Proceedings of the Eindhoven FASTAR Days 2004 : Eindhoven, The Netherlands, September 3-4, 2004

Proceedings of the Eindhoven FASTAR Days 2004

Edited by Loek Cleophas and Bruce W. Watson

December 2004

FASTAR Research Group
http://fastar.org

Software Construction Group
Department of Mathematics and Computer Science
Technische Universiteit Eindhoven
P.O. Box 513, NL-5600 MB Eindhoven, The Netherlands

Sponsored by

NWO
Netherlands Organisation for Scientific Research

Contents

The Eindhoven FASTAR Days (EFD) 2004 were organized by the Software Construction group of the Department of Mathematics and Computer Science at the Technische Universiteit Eindhoven. On September 3rd and 4th 2004, over thirty participants—hailing from the Czech Republic, Finland, France, The Netherlands, Poland and South Africa—gathered at the Department to attend the EFD.

The EFD were organized in connection with the research on finite automata by the FASTAR Research Group, which is centered in Eindhoven and at the University of Pretoria, South Africa. FASTAR (Finite Automata Systems—Theoretical and Applied Research) is an international research group that aims to lead in all areas related to finite state systems. The work in FASTAR includes both core and applied parts of this field.

The EFD therefore focused on the field of finite automata, with an emphasis on practical aspects and applications. Eighteen presentations, mostly on subjects within this field, were given, by researchers as well as students from participating universities and industrial research facilities.

This report contains the proceedings of the conference, in the form of papers for twelve of the presentations at the EFD. Most of them were initially reviewed and distributed as handouts during the EFD. After the EFD took place, the papers were revised for publication in these proceedings.

We would like to thank the participants for their attendance and presentations, making the EFD 2004 as successful as they were. Based on this success, it is our intention to make the EFD into a recurring event.

Eindhoven, December 2004

Loek Cleophas
Bruce W. Watson

Program Committee
Loek Cleophas, Bruce W. Watson

Local Arrangements
Loek Cleophas, Hanneke Driever, Bruce W. Watson

Repetitions in Text and Finite Automata[1]

Bořivoj Melichar
Department of Computer Science & Engineering
Czech Technical University
Karlovo nám. 13, 121 35 Prague 2, Czech Republic
email: melichar@fel.cvut.cz

Abstract: A general way to find repetitions of factors in a given text is shown. We start with a classification of repetitions. The general models for finding exact repetitions in one string and in a finite set of strings are introduced. It is shown that d-subsets created during determinization of nondeterministic factor automata contain all information concerning repetitions of factors. The principle of the analysis of d-subsets is then used for finding approximate repetitions using several distances for a general finite alphabet and for an ordered alphabet including the case of presence of don't care symbols. Complexity of finding repetitions is shown for exact repetitions in one string.

1 Introduction

Let a text $T = t_1 t_2 \dots t_n$ be given. Finding a repetition in text T can be defined as determining whether some substring (factor) repeats in the given text T. Furthermore, we can distinguish between exact and approximate repetitions. For approximate repetitions we use several distances for both a general alphabet and an ordered alphabet. In some cases the problem of finding repetitions in a text concerns a specified factor. The goal of this work is to find a general way to find all these repetitions in the given text. The main idea is based on the use of finite automata.

The very first attempt to solve the problem of finding repetitions in a string is in a Master's thesis [Ma79]. The first paper on this topic is [Cr86]. Some other papers on this topic are [CR94], [CCI+99] and [CCI+00]. An approach used in this paper is based on the construction of deterministic factor automata. This approach is based on some ideas from Master's thesis [Me02].

After the overview of basic definitions in Chapter 2, the attempt to make a classification of repetition problems is shown in Chapter 3. Next two Chapters are devoted to exact repetitions in one string and in a finite set of strings. The proof of the correctness of our approach for finding exact repetitions in one string is given in Chapter 4. Subsequent Chapters describe the solution of the approximate repetition problem for six distances for both general and ordered alphabets including the case of presence of don't care symbols.

[1]This research has been partially supported by the Ministry of Education, Youth, and Sport of the Czech Republic under research program No. J04/98:212300014.

2 Definitions

Basic notions from the theory of automata follow [AU71] and [HU79]. Basic notions from stringology are from [CR94]. The notion of factor automaton was introduced in [BBE+87].

Definition 2.1 (Set of factors)
The set $Fact(x), x \in A^*$, is the set of all *substrings (factors)* of the string x:
$Fact(x) = \{y : x = uyv, u, v \in A^*, x, y \in A^*\}$. □

Definition 2.2 (d–subset)
Let $M_1 = (Q_1, A, \delta_1, q_{01}, F_1)$ be a nondeterministic finite automaton. Let $M_2 = (Q_2, A, \delta_2, q_{02}, F_2)$ be the deterministic finite automaton equivalent to automaton M_1. Automaton M_2 is constructed using the standard determinization algorithm based on subset construction. Every state $q \in Q_2$ corresponds to some subset d of Q_1. This subset will be called *d–subset* (deterministic subset). □

Definition 2.3 (Exact repetition in one string)
Let T be a string, $T = a_1a_2 \dots a_n$ and $a_i = a_j, a_{i+1} = a_{j+1}, \dots, a_{i+m} = a_{j+m},\ i < j,\ m \geq 0$. The string $x_2 = a_ja_{j+1} \dots a_{j+m}$ is an *exact repetition* of the string $x_1 = a_ia_{i+1} \dots a_{i+m}$. x_1 or x_2 is called the repeating factor in text T.

□

Definition 2.4 (Exact repetition in a set of strings)
Let S be a set of strings, $S = \{x_1, x_2, \dots, x_{|S|}\}$ and $x_{pi} = x_{qj}, x_{pi+1} = x_{qj+1}, \dots, x_{pm} = x_{qm}$, where $i \in \langle 1, x_p\rangle,\ j \in \langle 1, x_q\rangle,\ m \geq 0,\ p, q \in \langle 1, |S|\rangle$. The string $x_{qj}x_{qj+1} \dots x_{qm}$ is an *exact repetition* of the string $x_{pi}x_{pi+1} \dots x_{pm}$.
□

Definition 2.5 (Approximate repetition in one string)
Let $T = a_1a_2 \dots a_n$ be a string and $D(a_ia_{i+1} \dots a_{i+m}, a_ja_{j+1} \dots a_{j+m'}) \leq k$, where $m, m' \geq 0$, D is a distance, $0 < k < n$. The string $a_ja_{j+1} \dots a_{j+m'}$ is an *approximate repetition* of the string $a_ia_{i+1} \dots a_{i+m}$. □

Approximate repetition in a set of strings can be defined in a similar way.

Definition 2.6 (Type of repetition)
Let $x_2 = a_ja_{j+1} \dots a_{j+m'}$ be an exact or approximate repetition of $x_1 = a_ia_{i+1} \dots a_{i+m}, i < j$, in one string.
Then if $j - i < m$ then the repetition is with an *overlapping* (O),
if $j - i = m$ then the repetition is a *square* (S),
if $j - i > m$ then the repetition is with a *gap* (G). □

Definition 2.7 (Distances of strings - general alphabets)
Three variants of the distances between two strings X and Y are defined as the minimum number of editing operations:

1. *replace* (Hamming distance, R-distance),
2. *delete, insert* and *replace* (Levenshtein distance, DIR-distance),
3. *delete, insert, replace* and *transpose* of neighbour symbols (generalized Levenshtein distance, $DIRT$-distance),

needed to convert string X to string Y. □

The Hamming distance is a metrics on the set of strings of equal length. The Levenshtein and the generalized Levenshtein distances are metrics on the set of strings of not necessarily equal length.

Definition 2.8 (Distance of strings - ordered alphabets)
Let $A = \{a_1, a_2, \ldots, a_p\}$ be an ordered alphabet and $k > 0$ an integer. Two symbols $a_i, a_j \in A,\ i, j \in \langle 1, p\rangle$, are said to be Δ_k-*approximate*, if and only if

$$\Delta_k(a_i, a_j) = |i - j| \leq k.$$

Next, three variants of distance are defined as follows:

1. Δ*distance:*

   For a given integer $k > 0$ we say that two strings x, y are Δ_k-approximate, if and only if

   $$|x| = |y|, \text{ and } \Delta_k(x_i, y_i) \leq k, \quad \forall i \in \{1, \ldots, |x|\}. \tag{1}$$

2. Γ*distance:*

   For a given integer $k > 0$ we say that two strings x, y are Γ_k-*approximate*, if and only if

   $$|x| = |y|, \text{ and } \sum_{i=1}^{|x|} \Delta_k(x_i, y_i) \leq k. \tag{2}$$

3. (Δ, Γ)*distance*:

   We say that two strings x, y are (Δ_k, Γ_l)*-approximate*, if and only if x and y satisfy both conditions (1) and (2). □

All three distances are metrics on the sets of strings of equal length.

Definition 2.9
The "don't care" symbol is a special universal symbol $\circ$ that matches any other symbol, including itself.

3 Classification of repetitions

Problems of repetitions of factors in a string over a finite size alphabet can be classified according to various criteria. We will use four criteria for classification of repetition problems leading to four-dimensional space in which each point corresponds to the particular problem of repetition of a factor in a string. Let us make a list of all dimensions including possible

"values" in each dimension:

1. Number of strings:
   - one,
   - finite number greater than one,
   - infinite number.
2. Repetition of factors (see Definition 2.6):
   - with overlapping,
   - square,
   - with gap.
3. Specification of the factor:
   - repeated factor is given,
   - repeated factor is not given,
     - length l of the repeated factor is given exactly,
     - length of the repeated factor is less than given l,
     - length of the repeated factor is greater than given l,
     - finding the longest repeated factor.
4. The way of finding repetitions:
   - exact repetition,
   - approximate repetition with Hamming distance (R-repetition),
   - approximate repetition with Levenshtein distance (DIR-repetition),
   - approximate repetition with generalized Levenshtein distance ($DIRT$-repetition),
   - Δ-approximate repetition,
   - Γ-approximate repetition,
   - (Δ, Γ)-approximate repetition.
5. Importance of symbols in factor:
   - take care of all symbols,
   - don't care of some symbols.

The above classification is visualized in Figure 3.1. If we count the number of possible problems of finding repetitions in a string, we obtain $N = 3 * 3 * 2 * 7 * 2 = 272$.

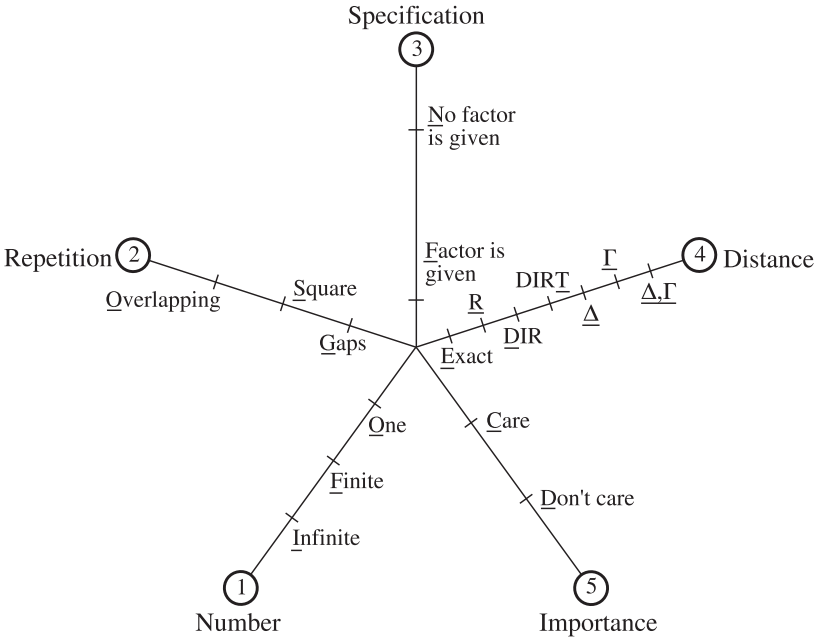


Figure 3.1: Classification of repetition problems

In order to facilitate references to a particular problem of repetition in a string, we will use abbreviations for all problems. These abbreviations are summarized in Table 3.1.

Using this method, we can, for example, refer to the overlapping exact repetition of a given factor where all symbols are considered as the $OFEC$ problem.

Instead of the single repetition problem we will use the notion of a family of repetitions in string problems. In this case we will use the symbol ? instead of a particular symbol. For example $S???$ is the family of all problems concerning square repetitions.

Dimension	1	2	3	4	5
	O	O	F	E	C
	F	S	N	R	D
	I	G		D	
				T	
				Δ	
				Γ	
				(Δ, Γ)	

Table 3.1: Abbreviations of repetition problems

Each repetition problem can have several instances:

1. verify whether some factor is repeated in the text or not,
2. find the first repetition of some factor,
3. find the number of all repetitions of some factor,
4. find all repetitions of some factor and where they are.

If we take into account all possible instances, the number of repetitions in string problems grows further.

4 Exact repetitions in one string

In this section we will introduce how to use a factor automaton for finding exact repetitions in one string (*O?NEC* problem). The main idea is based on the construction of the deterministic factor automaton. First, we construct a nondeterministic factor automaton for a given string. The next step is to construct the equivalent deterministic factor automaton. During this construction, we memorize d-subsets. The repetitions that we are looking for are obtained by analyzing these d-subsets. The next algorithm describes the computation of d-subsets of a deterministic factor automaton.

Algorithm 4.1
Computation of repetitions in one string.
Input: String $T = a_1 a_2 \dots a_n$.
Output: Deterministic factor automaton M_D accepting $Fact(T)$ and d–subsets for all states of M.
Method:

1. Construct nondeterministic factor automaton M_N accepting $Fact(T)$:
   (a) Construct finite automaton M accepting string $T = a_1 a_2 \dots a_n$ and all its prefixes.
   $M = (\{q_0, q_1, q_2, \dots, q_n\}, A, \delta, q_0, \{q_0, q_1, \dots, q_n\})$,
   where $\delta(q_i, a_{i+1}) = q_{i+1}$ for all $i \in \langle 0, n-1 \rangle$.
   (b) Construct finite automaton M_ε from the automaton M by inserting ε–transitions:
   $\delta(q_0, \varepsilon) = \{q_1, q_2, \dots, q_{n-1}\}$.
   (c) Replace all ε–transitions by non–ε–transitions. The resulting automaton is M_N.
2. Construct deterministic factor automaton M_D equivalent to automaton M_N and memorize the d–subsets during this construction.
3. Analyze d–subsets to compute repetitions. □

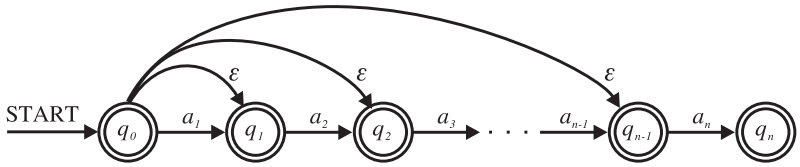


Figure 4.1: Transition diagram of factor automaton M_ε with ε–transitions constructed in step 1.b of Algorithm 4.1

The factor automaton M_ε constructed by Algorithm 4.1 has, after step 1.b, the transition diagram depicted in Fig. 4.1. The factor automaton M_N has, after step 1.c of Algorithm 4.1, the transition diagram depicted in Fig. 4.2.

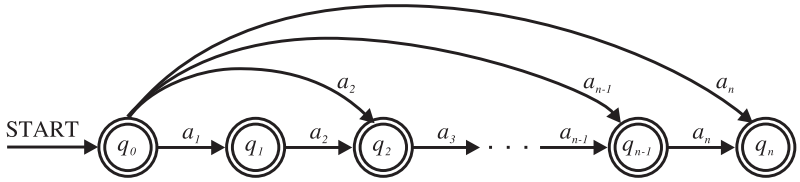


Figure 4.2: Transition diagram of factor automaton M_N after the removal of ε–transitions in step 1.c of Algorithm 4.1

The next example shows the construction of the deterministic factor automaton and the analysis of the d–subsets.

Let us make a note concerning labelling: Labels used as the names of states are selected in order to indicate positions in the string. This labelling will be useful later.

Example 4.2
Let us use the text $T = ababa$. At first, we construct a nondeterministic factor automaton $M_\varepsilon(ababa) = (Q_\varepsilon, A, \delta_\varepsilon, 0, Q_\varepsilon)$ with ε-transitions. Its transition diagram is depicted in Figure 4.3.

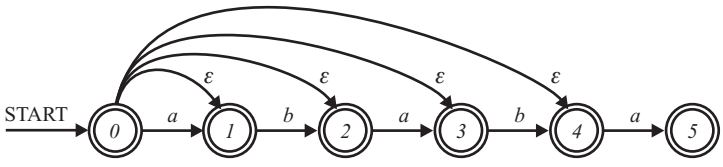


Figure 4.3: Transition diagram of factor automaton $M_\varepsilon(ababa)$ from Example 4.2

Then we remove ε-transitions and the resulting nondeterministic factor automaton $M_N(ababa) = (Q_N, A, \delta_N, 0, Q_N)$ is depicted in Figure 4.4 and its transition table is Table 4.1. □

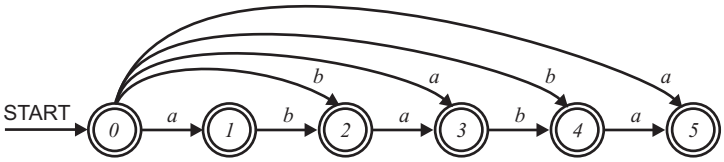


Figure 4.4: Transition diagram of nondeterministic factor automaton $M_N(ababa)$ from Example 4.2

State	a	b
0	$1, 3, 5$	$2, 4$
1		2
2	3	
3		4
4	5	
5		

Table 4.1: Transition table of nondeterministic factor automaton $M_N(ababa)$ from Example 4.2

As a next step, we construct the equivalent deterministic factor automaton $M_D(ababa) = (Q_D, A, \delta_D, 0, Q_D)$. During this operation we memorize the created d-subsets. We suppose, taking into account the labelling of the states of the nondeterministic factor automaton, that d-subsets are ordered in the natural way. The extended transition table (with ordered d-subsets) of the deterministic factor automaton $M_D(ababa)$ is shown in Table 4.2. The transition diagram of M_D is depicted in Figure 4.5.

State	d-subset	a	b
D_0	0	$1, 3, 5$	$2, 4$
D_1	$1, 3, 5$		$2, 4$
D_2	$2, 4$	$3, 5$	
D_3	$3, 5$		4
D_4	4	5	
D_5	5		

Table 4.2: Transition table of automaton $M_D(ababa)$ from Example 4.2

Now we start the analysis of the resulting d-subsets:
d-subset $d(D_1) = \{1, 3, 5\}$ shows that factor a repeats at positions $1, 3$ and 5 of the given string, and its length is one. d-subset $d(D_2) = \{2, 4\}$ shows that factor ab repeats, and its occurrence in the string ends at positions

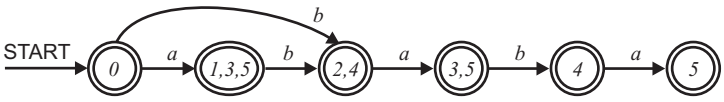


Figure 4.5: Transition diagram of deterministic factor automaton $M_D(ababa)$ from Example 4.2

2 and 4 and its length is two. Moreover, the suffix b of this factor also repeats at the same positions as factor ab. d-subset $d(D_3) = \{3, 5\}$ shows that factor aba repeats, and its occurrence in the string ends at positions 3 and 5 and its length is three. Moreover, its suffix ba also repeats at the same positions. Suffix a of factor aba also repeats at positions 3 and 5, but we have already obtained this information during analysis of the d-subset $d(D_1) = \{1, 3, 5\}$. Analysis of the d-subsets having only single states brings no further information on repeating factors.

Let us make a summary of these observations in the *repetition table* shown in Table 4.3.

d-subset	Factor	First occurrence	Repetitions
$1, 3, 5$	a	1	$(3, G), (5, G)$
$2, 4$	ab	2	$(4, S)$
$2, 4$	b	2	$(4, G)$
$3, 5$	aba	3	$(5, O)$
$3, 5$	ba	3	$(5, S)$

Table 4.3: Repetition table of $ababa$

The last column of the repetition table with the header *Repetitions* contains sequences of pairs (i, R) where i is the position where the repeated factor ends (let us recall the note on labelling of states) and R is the type of repetition (G, S, O). The other columns are selfexplaining. □

The construction of the repetition table is based on the following observations illustrated in Figure 4.6 and Lemmata 4.3 and 4.4.

Lemma 4.3
Let T be a string and $M_D(T)$ be the deterministic factor automaton for T with states labelled by corresponding d-subsets. If factor $u = a_1 a_2 \dots a_m$, $m \geq 1$, is repeats in string T and its occurrences start at positions $x + 1$ and $y + 1$, $x \neq y$ then there exists a d-subset in $M_D(T)$ containing the pair $\{x + m, y + m\}$.

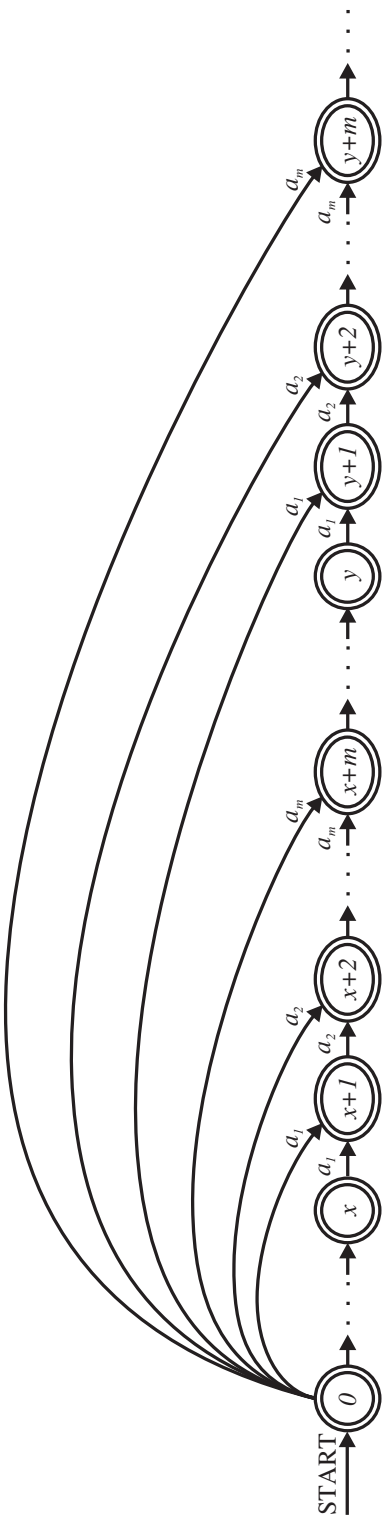


Figure 4.6: Repeated factor $u = a_1 a_2 \dots a_m$ in $M_N(T)$

Proof
Let $M_N(T) = (Q_N, A, \delta_N, q_0, Q_N)$ be the nondeterministic factor automaton for T and let $u = a_1 a_2 \dots a_m$ be the factor starting at positions $x+1$ and $y+1$ in T, $x \neq y$. Then there are transitions in $M_N(T)$ from state 0 to states $x+1$ and $y+1$ for symbol a_1, ($\delta_N(0, a_1)$ contains $x+1$ and $y+1$). It follows from the construction of $M_N(T)$ that:

$$\begin{array}{ll} \delta_N(x+1, a_2) = \{x+2\}, & \delta_N(y+1, a_2) = \{y+2\}, \\ \delta_N(x+2, a_3) = \{x+3\}, & \delta_N(y+2, a_3) = \{y+3\}, \\ \vdots & \vdots \\ \delta_N(x+m-1, a_m) = \{x+m\}, & \delta_N(y+m-1, a_m) = \{y+m\}. \end{array}$$

Deterministic factor automaton $M_D(T) = (Q_D, A, \delta_D, D_0, Q_D)$ then contains states $D_0, D_1, D_2, \dots D_m$ having this property:

$$\begin{array}{ll} \delta_D(D_0, a_1) = D_1, & \{x+1, y+1\} \subset D_1, \\ \delta_D(D_1, a_2) = D_2, & \{x+2, y+2\} \subset D_2, \\ \vdots & \vdots \\ \delta_D(D_{m-1}, a_m) = D_m, & \{x+m, y+m\} \subset D_m. \end{array}$$

We can conclude that the d-subset D_m contains the pair $\{x+m, y+m\}$.
□

Lemma 4.4
Let T be a string and let $M_D(T)$ be the deterministic factor automaton for T with states labelled by corresponding d-subsets. If a d-subset D_m contains two elements $x+m$ and $y+m$ then there exists the factor $u = a_1 a_2 \dots a_m$, $m \geq 1$, starting at both positions $x+1$ and $y+1$ in string T.

Proof
Let $M_N(T)$ be the nondeterministic factor automaton for T. If a d-subset D_m contains elements from $\{x+m, y+m\}$ then it holds for δ_N of $M_N(T)$:
$\{x+m, y+m\} \subset \delta_N(0, a_m)$, and
$\delta_N(x+m-1, a_m) = \{x+m\}$,
$\delta_N(y+m-1, a_m) = \{y+m\}$ for some $a_m \in A$.
Then the d-subset D_{m-1} such that $\delta_D(D_{m-1}, a_m) = D_m$ must contain
$x+m-1, y+m-1$ such that $\{x+m-1, y+m-1\} \subset \delta_N(0, a_{m-1})$,
$\delta_N(x+m-2, a_{m-1}) = \{x+m-1\}$,
$\delta_N(y+m-2, a_{m-1}) = \{y+m-1\}$
and for the same reason the D-subset D_1 must contain $x+1, y+1$ such that $\{x+1, y+1\} \subset \delta_N(0, a_1)$ and $\delta_N(x, a_1) = \{x+1\}$, $\delta_N(y, a_1) = \{y+1\}$.
Then there exists the sequence of transitions in $M_D(T)$:

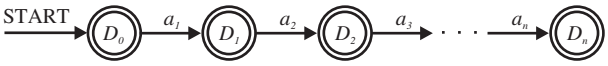


Figure 4.7: Repeated factor $u = a_1a_2 \dots a_m$ in $M_D(T)$

$$\begin{aligned}(D_0, a_1a_2 \dots a_m) &\vdash (D_1, a_2 \dots a_m)\\ &\vdash (D_2, a_3 \dots a_m)\\ &\vdots\\ &\vdash (D_{m-1}, a_m)\\ &\vdash (D_m, \varepsilon),\end{aligned}$$

where

$$\begin{aligned}&\{x+1, y+1\} \subset D_1,\\ &\vdots\\ &\{x+m, y+m\} \subset D_m.\end{aligned}$$

This sequence of transitions corresponds to two different sequences of transitions in $M_N(T)$ going through state $x+1$:

$$\begin{aligned}(0, a_1a_2 \dots a_m) &\vdash (x+1, a_2 \dots a_m)\\ &\vdash (x+2, a_3 \dots a_m)\\ &\vdots\\ &\vdash (x+m-1, a_m)\\ &\vdash (x+m, \varepsilon),\end{aligned}$$

$$\begin{aligned}(x, a_1a_2 \dots a_m) &\vdash (x+1, a_2 \dots a_m)\\ &\vdash (x+2, a_3 \dots a_m)\\ &\vdots\\ &\vdash (x+m-1, a_m)\\ &\vdash (x+m, \varepsilon).\end{aligned}$$

Similarly two sequences of transitions go through state $y+1$:

$$\begin{aligned}(0, a_1a_2 \dots a_m) &\vdash (y+1, a_2 \dots a_m)\\ &\vdash (y+2, a_3 \dots a_m)\\ &\vdots\\ &\vdash (y+m-1, a_m)\\ &\vdash (y+m, \varepsilon),\end{aligned}$$

$$\begin{aligned}(y, a_1a_2 \dots a_m) &\vdash (y+1, a_2 \dots a_m)\\ &\vdash (y+2, a_3 \dots a_m)\\ &\vdots\\ &\vdash (y+m-1, a_m)\\ &\vdash (y+m, \varepsilon).\end{aligned}$$

It follows from this that the factor $u = a_1 a_2 \dots a_m$ is present twice in string T in different positions $x+1, y+1$. □

The following Lemma is a simple consequence of Lemma 4.4.

Lemma 4.5
Let u be a repeating factor in string T. Then all factors of u are also repeating factors in T. □

Definition 4.6
If u is a repeating factor in text T and there is no longer factor of the form vuw, v or $w \neq \varepsilon$ but not both, which is also a repeating factor, then we will call u the *maximal repeating factor*. □

Definition 4.7
Let $M_D(T)$ be a deterministic factor automaton. The *depth* of each state D of M_D is the length of the longest sequence of transitions leading from the initial state to state D. □

If there exists a sequence of transitions from the initial state to state D which is shorter than the depth of D, it corresponds to the suffix of the maximal repeating factor.

Lemma 4.8
Let u be a maximal repeating factor in string T. The length of this factor is equal to the depth of the state in $M_D(T)$ indicating the repetition of u.□

Proof
The path for the maximal repeating factor $u = a_1 a_2 \dots a_m$ stars in the initial state, because the states $x+1$ and $y+1$ of the nondeterministic factor automaton $M_N(T)$ are direct successors of its initial state and therefore $\delta_D(D_0, a_1) = D_1$ and $\{x+1, y+1\} \subset D_1$. Therefore there exists a sequence of transitions in the deterministic factor automaton $M_D(T)$:

$$\begin{aligned}
(D_0, a_1 a_2 \dots a_m) &\vdash (D_1, a_2 \dots a_m) \\
&\vdash (D_2, a_3 \dots a_m) \\
&\vdots \\
&\vdash (D_{m-1}, a_m) \\
&\vdash (D_m, \varepsilon)
\end{aligned}$$

□

The remaining question is the decision on the type of repetition. Let us suppose that the length of the repeating factor is m. If we have in the d-subset in question two elements $i, j, j > i$, then:
If $j - i < m$ then there is a repetition with overlapping.

If $j - i = m$ then there is a square.
If $j - i > m$ then there is a repetition with gap.

There follows one more observation from Example 4.2:

Lemma 4.9
If some state in $M_D(T)$ has a corresponding d-subset containing one element only, then its successor also has a corresponding d-subset containing one element.

Proof
This follows from the construction of the deterministic factor automaton. The transition table of nondeterministic factor automaton $M_N(T)$ has more than one state in the row for the initial state only. All other states have at most one successor for a particular input symbol. Therefore in the equivalent deterministic factor automaton $M_D(T)$ the state corresponding to a d-subset having one element may have only one successor for one symbol, and this state has a corresponding d-subset containing just one element. □

We can use this observation during the construction of deterministic factor automaton $M_D(T)$ in order to find some repetition. It is enough to construct only the part of $M_D(T)$ containing d-subsets with at least two elements. The rest of $M_D(T)$ gives no information on repetitions.

We summarize the result of the computation of repetitions in *repetition table* $R(T)$. It contains the d–subset, the repeating factor, the ending position of its first occurrence, the ending positions of all repetitions and the type of each repetition. We will abbreviate this type as follows: O = repetition with Overlapping, S = Square repetition, G = repetition with Gap.

Algorithm 4.10
Constructing a repetition table containing exact repetitions in a given string.
Input: String $T = a_1 a_2 \dots a_n$.
Output: Repetition table R for string T.
Method:

1. Construct a deterministic factor automaton
   $M_D(T) = (Q_D, A, \delta_D, 0, Q_D)$ for a given string T.
   Memorize for each state $q \in Q_D$:
   (a) d-subset $D(q) = \{r_1, r_2, \dots, r_p\}$,
   (b) $d = depth(q)$,
   (c) maximal repeating factor for state q $maxfactor(q) = x, |x| = d$.

2. Create rows in the repetition table R for each state q having $D(q)$ with more than one element:

   (a) the row for the maximal repeating factor x of state q has the form:

$(\{r_1, r_2, \ldots, r_p\}, x, r_1, \{(r_2, X_2), (r_3, X_3), \ldots, (r_p, X_p)\}$,
where $X_i,\ 2 \leq i \leq p$, is equal to

i. O, if $r_i - r_{i-1} < d$,
ii. S, if $r_i - r_{i-1} = d$,
iii. G, if $r_i - r_{i-1} > d$,

(b) for each suffix y of x (such that the row for y was not created before) create the row of the form:
$(\{r_1, r_2 \ldots, r_p\}, y, r_1, \{(r_2, X_2), (r_3, X_3), \ldots, (r_p, X_p)\}$,
where $X_i,\ 2 \leq i \leq p$, is deduced in the same manner. □

An example of the repetition table is shown in Example 4.2 for the string $T = ababa$.

5 Complexity of computation of exact repetitions

The time and space complexity of the computation of exact repetitions in string is treated in this Chapter.

Definition 5.1
A d-subset is *simple* if it contains just one element. The corresponding state to it we call *simple state*. A d-subset is *multiple* if it contains more than one element. The corresponding state to it we will call *multiple state*.

□

The time complexity is composed of two parts:

1. The complexity of the construction of the deterministic factor automaton. If we take the number of states and transitions of the resulting factor automaton then the complexity is linear. More exactly the number of its states is

   $NS \leq 2n - 2$,
   and the number of transitions is

   $NT \leq 3n - 4$.

2. The second part of the overall complexity is the construction of repetition table. The number of rows of this table is the number of different multiple d-subsets. The highest number of multiple d-subsets has the factor automaton for text $T = a^n$. Repeating factors of this text are $a, a^2, \ldots, a^{n-1}$.

There is necessary, for the computation of repetitions using factor automata approach, to construct the part of deterministic factor containing only all multiple states. It is the matter of fact, that a simple state has at most one next state and it is simple one, too. Therefore, during the construction of

deterministic factor automaton, we can stop construction of the next part of this automaton as soon as we reach a simple state.

Example 5.2
Let us have text $T = a^n,\ n > 0$. Let us construct deterministic factor automaton $M_D(a^n)$ for text T. Transition diagram of this automaton is depicted in Fig. 5.1. Automaton $M_D(a^n)$ has $n+1$ states and n transitions.

Figure 5.1: Transition diagram of deterministic factor automaton $M_D(a^n)$ for text $T = a^n$ from Example 5.2

Number of multiple states is $n - 1$.

To construct this automaton in order to find all repetitions, we must construct the whole automaton including the initial state and the state n (terminal state). Repetition table R has the form shown in Table 5.1. □

d-subset	Factor	List of repetitions
$1, 2, \ldots, n$	a	$(1, F), (2, S), (3, S) \ldots, (n, S)$
$2, \ldots, n$	aa	$(2, F), (3, 0), (4, 0) \ldots, (n, 0)$
$\vdots$		
$n-1, n$	a^{n-1}	$(n-1, F), (n, 0)$

Table 5.1: Repetition table R for text $T = a^n$ from Example 5.2

The opposite case to the previous one is the text composed of symbols which are all different. The length of such text is limited by the size of alphabet.

Example 5.3
Let the alphabet be $A = \{a, b, c\}$ and text $T = abcd$. Deterministic factor automaton $M_D(abcd)$ for text T has transition diagram depicted in Fig. 5.2. Automaton $M_D(abcd)$ has $n+1$ states and $2n-1$ transitions. All respective d-subsets are simple. To construct this automaton in order to find all repetitions, we must construct all next states of the initial state for all symbols of the text. The number of these states is just n. The repetition table is empty. □

Now, after the presentation both limit cases, we will try to find some case inbetween with the maximal complexity. We guess, that the next example

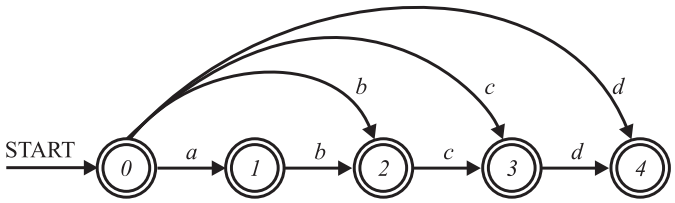


Figure 5.2: Transition diagram of deterministic factor automaton $M_D(abcd)$ for text $T = abcd$ from Example 5.2

is showing it. The text selected in such way, that all proper suffixes of the prefix of the text appear in it and therefore they are repeating.

Example 5.4
Let the text be $T = abcdbcdcdd$. Deterministic factor automaton $M_D(T)$ has the transition diagram depicted in Fig. 5.3. Automaton M_D has 17 states

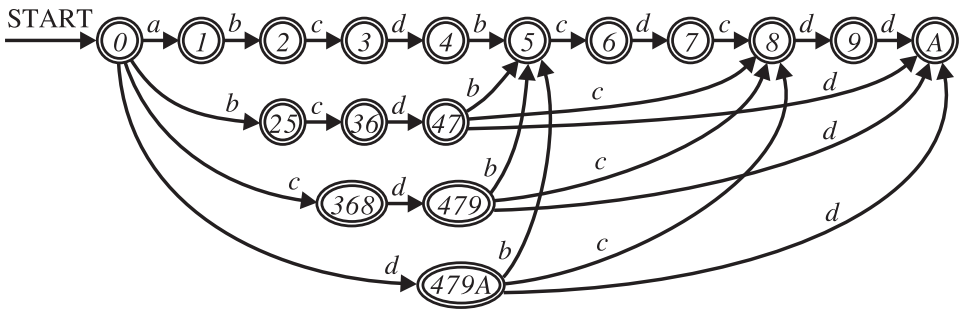


Figure 5.3: Transition diagram of deterministic factor automaton $M_D(T)$ for text $T = abcdbcdcdd$ from Example 5.4

and 25 transitions while text T has 10 symbols. The number of multiple d-subsets is 6. To construct this automaton in order to find all repetitions, we must construct all multiple states and moreover the states corresponding to single d-subsets: $0, 1, 5, 8, A$.

The results is, that we must construct 11 states from the total number of 17 states. Repetition table R is shown in Table 5.2. □

It is known, that the maximal state and transition complexity of the factor automaton is reached for text $T = ab^{n-2}c$. Let us show such factor automaton in this context.

Example 5.5
Let the text be $T = ab^4c$. Deterministic factor automaton $M_D(T)$ has transition diagram depicted in Fig. 5.4. Automaton $M_D(T)$ has 10 $(2 * 6 - 2)$ states and 14 $(3 * 6 - 4)$ transitions while the text has 6 symbols. The num-

d-subset	Factor	List of repetitions
25	b	$(2, F), (5, G)$
36	bc	$(3, F), (6, G)$
47	bcd	$(4, F), (7, S)$
368	c	$(3, F), (6, G), (8, G)$
479	cd	$(4, F), (7, G), (9, G)$
$479A$	d	$(4, F), (7, G), (9, G), (10, S)$

Table 5.2: Repetition table R for text $T = abcdbcdcdd$ from Example 5.4

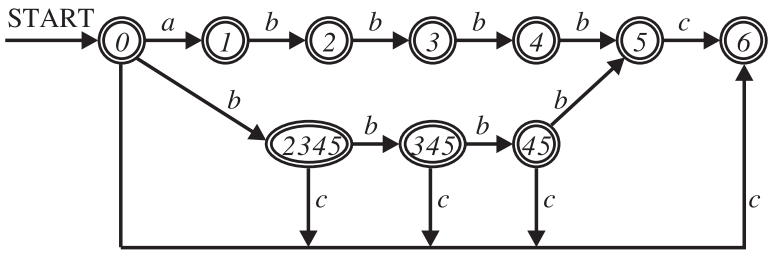


Figure 5.4: Transition diagram of deterministic factor automaton $M_D(T)$ for text $T = ab^4c$ from Example 5.5

ber of multiple states is 3 $(6 - 3)$. To construct this automaton in order to find all repetitions, we must construct the 3 multiple states and moreover 3 simple states. Therefore we must construct 6 states from the total number of 10 states. Repetition table R is shown in Table 5.3. □

d-subset	Factor	List of repetitions
2345	b	$(2, F), (3, S), (4, S), (5, S)$
345	bb	$(3, F), (4, O), (5, O)$
45	bbb	$(4, F), (5, O)$

Table 5.3: Repetition table R for text $T = ab^4c$ from Example 5.5

We have used, in the previous examples, three measures of complexity:

1. The number of multiple states of the deterministic factor automaton. This number is equal to the number of rows in the resulting repetition table, because each row of the repetition table corresponds to one multiple d-subset. Moreover, it corresponds to the number of repeating factors.
2. The number of states which are necessary to construct in order to get all information on repetitions. We must reach the simple state on all

pathes starting in the initial state. We already know that there is at most one successor of a simple state and it is a simple state, too (Lema 4.9). The number of such states which is necessary to construct is therefore greater than the number of multiple states.

3. The total number of repetitions (occurrences) of all repeating factors in text. This number corresponds to the number of items in the last column of the repetition table headed by "List of repetitions".

The results concerning this the measures of complexity from previous examples are summarised in the Table 5.4.

Text	No. of multiple states	No. of necessary states	No. of repetitions
a^n	$n-1$	$n+1$	$(n^2+n-2)/2$
$a_1a_2\ldots a_n$ (all symbols unique)	0	$n+1$	0
$a_1a_2\ldots a_ma_2\ldots a_m\ldots a_{m-1}a_ma_m$	$(m^2-m)/2$	$(m^2+m)/2$	$\sum_{i=1}^{m-1} i(m-i+1)$
$ab^{n-2}c$	$n-3$	n	$(n^2-3n)/2$

Table 5.4: Measures of complexity from Examples 5.2, 5.3, 5.4, 5.5

Let us show how the complexity measures from Table 5.4 have been computed.

Example 5.6
Text $T = a^n$ has been used in Example 5.2. The number of multiple states is $n-1$ which is the number of repeating factors. The number of necessary states is $n+1$ because the initial and the terminal states must be constructed. The number of repetitions is given by the sum:

$$n + (n-1) + (n-2) + \ldots + 2 = \frac{n^2+n-2}{2}.$$

□

Example 5.7
Text $T = abcd$ has been used in Example 5.3. This automaton has no multiple state. The number of necessary states is $n+1$. It means, that in order to recognize that no repetition exists in such text all states of this automaton must be constructed. □

Example 5.8
Text $T = abcdbcdcdd$ used in Example 5.4 has very special form. It consists of prefix $abcd$ followed by all its proper suffixes. It is possible to construct

such text only for some n. The length n of the text must satisfy the condition:

$$n = \sum_{i=1}^{m} i = \frac{m^2 + m}{2},$$

where m is the length of the prefix in question. It follows that

$$m = \frac{-1 \pm \sqrt{1 + 8n}}{2}$$

and therefore $m = O(\sqrt{n})$.

The number of multiple states is

$$(m-1) + (m-2) + \ldots + 1 = \frac{m^2 - m}{2}.$$

The number of necessary states we must increase by m which is the number of simple states being next states of the multiple states and the initial state. Therefore the number of necessary states is $(m^2 + m)/2$. The number of repetitions is

$$m + 2(m+1) + 3(m-2) + \ldots + (m-1)2 = \sum_{i=1}^{m-1} i(m - i + 1).$$

Therefore this number is $O(m^2) = O(n)$. □

Example 5.9
Text $T = ab^{n-2}c$ used in Example 5.5 leads to the factor automaton having maximal number of states and transitions. The number of multiple states is equal to $n-3$ and the number of necessary states is equal to n. The number of repetitions is

$$(n-2) + (n-3) + \ldots + 2 = \frac{n^2 - 3n}{2}.$$

□

It follows from the described experiments that the complexity of deterministic factor automata and therefore the complexity of computation of repetitions for a text of length n has these results:

1. The number of multiple states is linear. It means that the repetition table has $O(n)$ rows. It is the space complexity of the computation all repeated factors.
2. The number of necessary states is again linear. It means that time complexity of the computation all repeated factors is $O(n)$.
3. The number of repetitions is $O(n^2)$ which is the time and space complexity of the computation of all occurrences of repeated factors.

6 Exact repetitions in a finite set of strings

The idea of the use of a factor automaton for finding exact repetitions in one string can also be used for finding exact repetitions in a finite set of strings (*F?NEC* problem). We show in the next example the construction of a factor automaton and the analysis of d–subsets created during this construction.

Example 6.1
Let us construct the factor automaton for the set of strings $S = \{abab, abba\}$. First, we construct factor automata M_1 and M_2 for both strings in S. Their transition diagrams are depicted in Figs 6.1 and 6.2, respectively.

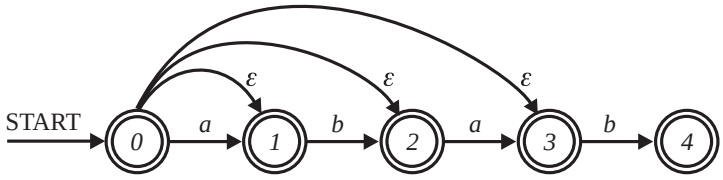


Figure 6.1: Transition diagram of factor automaton M_1 accepting $Fact(abab)$ from Example 6.1

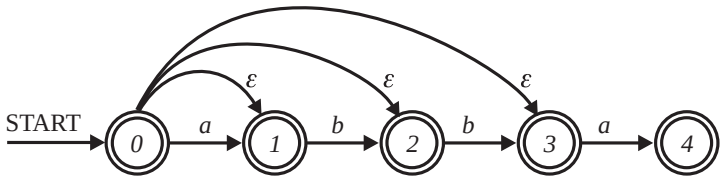


Figure 6.2: Transition diagram of factor automaton M_2 accepting $Fact(abba)$ from Example 6.1

In the second step we construct automaton M_ε accepting the language $L(M_\varepsilon) = Fact(abab) \cup Fact(abba)$. Its transition diagram is depicted in Fig. 6.3.

In the third step we construct automaton M_N by removing ε–transitions from automaton M_ε. Its transition diagram is depicted in Fig. 6.4.

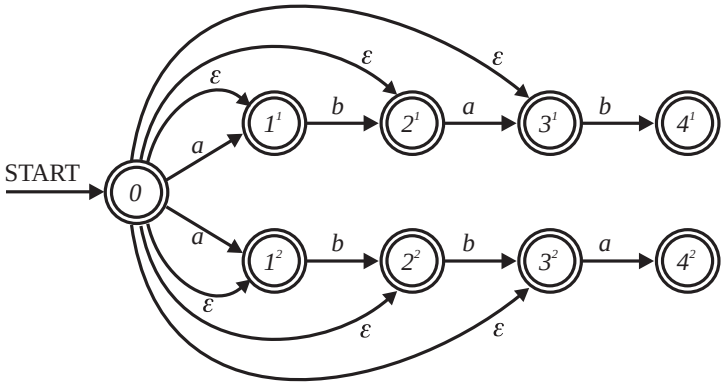


Figure 6.3: Transition diagram of factor automaton M_ε accepting the set $Fact(abab) \cup Fact(abba)$ from Example 6.1

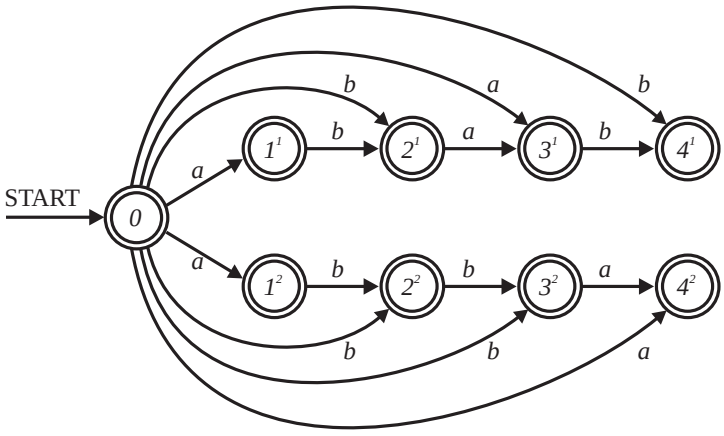


Figure 6.4: Transition diagram of nondeterministic factor automaton M_N accepting the set $Fact(abab) \cup Fact(abba)$ from Example 6.1

	a	b
0	$1^1 1^2 3^1 4^2$	$2^1 2^2 3^2 4^1$
$1^1 1^2 3^1 4^2$		$2^1 2^2 4^1$
$2^1 2^2 3^2 4^1$	$3^1 4^2$	3^2
$2^1 2^2 4^1$	3^1	3^2
$3^1 4^2$		4^1
3^1		4^1
3^2	4^2	
4^1		
4^2		

Table 6.1: Transition table of deterministic factor automaton M_D from Example 6.1

The last step is to construct deterministic factor automaton M_D. Its transition table is shown in Table 6.1 The transition diagram of the resulting deterministic factor automaton M_D is depicted in Fig. 6.5. Now we do

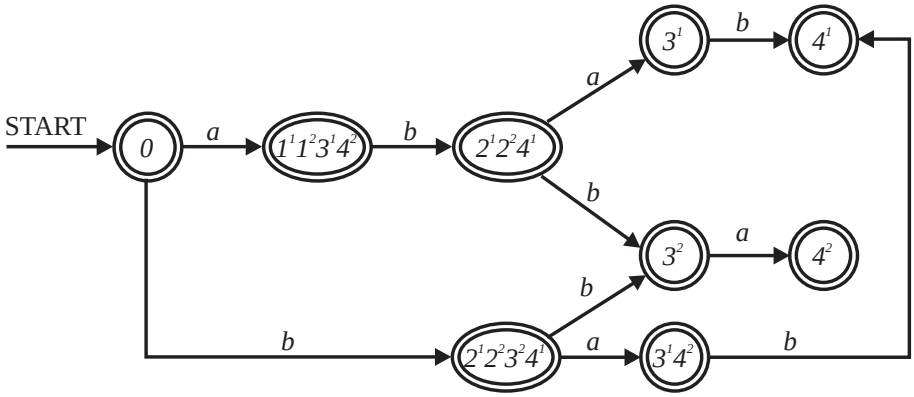


Figure 6.5: Transition diagram of deterministic factor automaton M_D accepting set $Fact(abab) \cup Fact(abba)$ from Example 6.1

the analysis of d–subsets of the resulting automaton M_D. The result of this analysis is the repetition table shown in Table 6.2 for the set $S = \{abab, abba\}$. □

Definition 6.2
Let S be a set of strings $S = \{x_1, x_2, \ldots, x_{|S|}\}$. The repetition table for S contains the following items:

1. d–subset,
2. corresponding factor,
3. list of repetitions of the factor containing elements of the form (i, j, X_{ij}),

d–subset	Factor	List of repetitions
$1^1 1^2 3^1 4^2$	a	$(1,1,F),(2,1,F),(1,3,G),(2,4,G)$
$2^1 2^2 4^1$	ab	$(1,2,F),(2,2,F),(1,4,G)$
$2^1 2^2 3^2 4^1$	b	$(1,2,F),(2,2,F),(2,3,S),(1,4,G)$
$3^1 4^2$	ba	$(1,3,F),(2,4,F)$

Table 6.2: Repetition table for set $S = \{abab, abba\}$ from Example 6.1

where i is the index of the string in S,
j is the position in string x_i,
X_{ij} is the type of repetition:
F - the first occurrence of the factor in string x_i,
O - repetition of the factor in x_i with overlapping,
S - repetition as a square in x_i,
G - repetition with a gap in x_i. □

7 Approximate repetitions – Hamming distance

We have used the factor automaton for finding exact repetitions in either one string or in a finite set of strings. A similar approach can be used for finding approximate repetitions in a string ($O?NRC$ problem). To find approximate repetitions, we use approximate factor automata. In this Section, we show how to find approximate repetitions using the Hamming distance.

Example 7.1
Let string $x = abba$. We construct an approximate factor automaton using Hamming distance $k = 1$.

1. We construct a finite automaton accepting string x and all strings with Hamming distance equal to 1. The set of these strings is denoted $H_1(abba)$. The resulting finite automaton has the transition diagram depicted in Fig. 7.1.

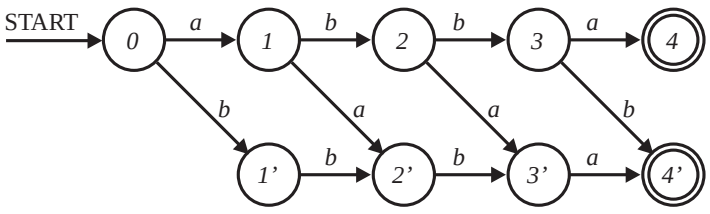


Figure 7.1: Transition diagram of the "Hamming" automaton accepting $H_1(abba)$ with Hamming distance $k = 1$ from Example 7.1

2. We use the principle of inserting the ε–transitions from state 0 to states 1, 2 and 3 and we fix all states as final states. The transition diagram with inserted ε–transition and fixed final states is depicted in Fig. 7.2.

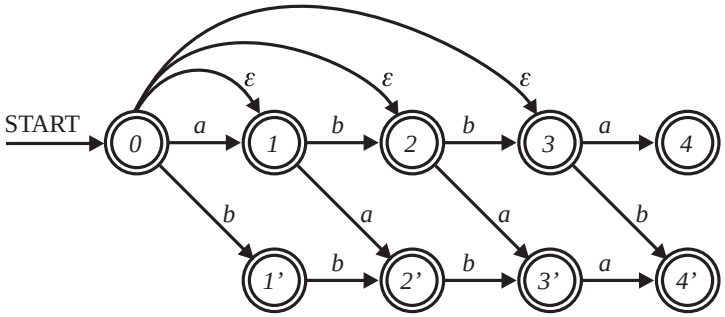


Figure 7.2: Transition diagram of the "Hamming" automaton with fixed final states and inserted ε–transitions from Example 7.1

3. We replace ε–transitions by non–ε–transitions. The resulting automaton has the transition diagram depicted in Fig. 7.3.

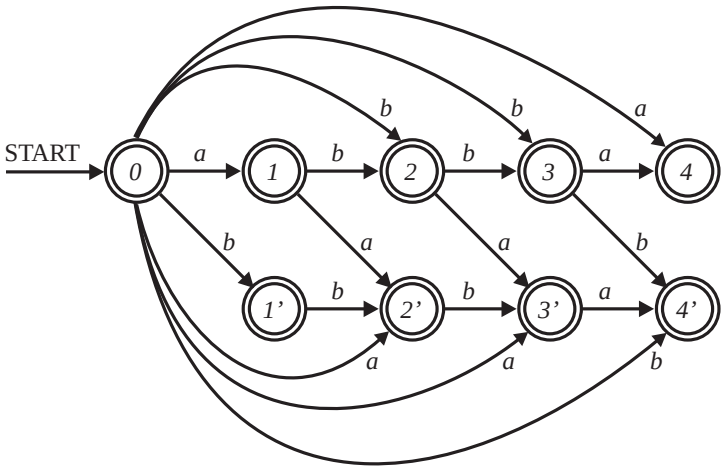


Figure 7.3: Transition diagram of the nondeterministic "Hamming" factor automaton after removal of ε–transitions from Example 7.1

4. The final operation is the construction of the equivalent deterministic finite automaton. Its transition table is Table 7.1

The transition diagram of the resulting deterministic approximate factor automaton is depicted in Fig. 7.4.

	a	b
0	$142'3'$	$231'4'$
$142'3'$	$2'4'$	$23'$
$231'4'$	$3'4$	$32'4'$
$2'4'$		$3'$
$23'$	$3'4'$	3
$3'4$	$4'$	
$32'4'$	4	$3'4'$
$3'$	$4'$	
$3'4'$	$4'$	
3	4	$4'$
$4'$		
4		

Table 7.1: Transition table of the deterministic "Hamming" factor automaton from Example 7.1

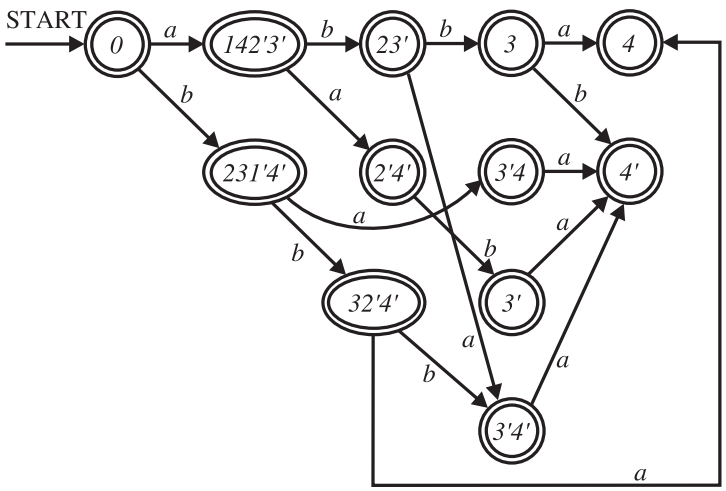


Figure 7.4: Transition diagram of the deterministic "Hamming" factor automaton for $x = abba$, Hamming distance $k = 1$ from Example 7.1

Now we construct the repetition table. We take into account the repetition of factors which are longer than k. In this case $k = 1$ and therefore we select repetitions of factors having length greater or equal to two. The next table contains information on the approximate repetition of factors of the string $x = abba$.

d–subset	Factor	Exact occurrence	Approximate repetitions
$23'$	ab	2	$(3, bb, O)$
$32'4'$	bb	3	$(2, ab, O), (4, ba, O)$
$3'4$	ba	4	$(3, bb, O)$

□

8 Approximate repetitions – Levenshtein distance

Let us note that Levenshtein distance between strings x and y is defined as the minimum number of editing operations delete, insert and replace which are necessary to convert string x into string y. In this section we show solution of *O?NDC* problem.

Example 8.1
Let string $x = abba$ and Levenshtein distance $k = 1$. Find all approximate repetitions in this string.
We construct an approximate factor automaton using Levenshtein distance $k = 1$.

1. We construct a finite automaton accepting string x and all strings with Levenshtein distance equal to 1. The set of these strings is denoted $L_1(abba)$. The resulting finite automaton has the transition diagram depicted in Fig. 8.1.

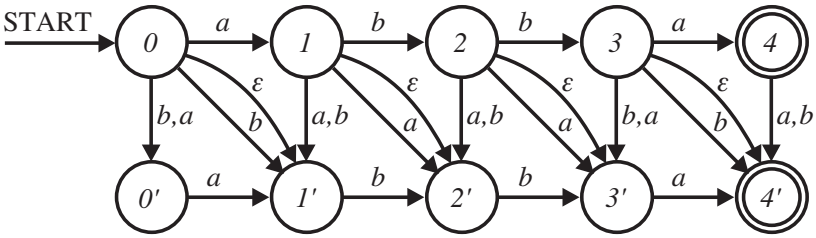


Figure 8.1: Transition diagram of the "Levenshtein" automaton accepting $L_1(abba)$, with Levenshtein distance $k = 1$ from Example 8.1

2. We use the principle of inserting the ε–transitions from state 0 to states 1, 2 and 3 and we fix all states as final states. The transition diagram with inserted ε–transitions and fixed final states is depicted in Fig. 8.2.

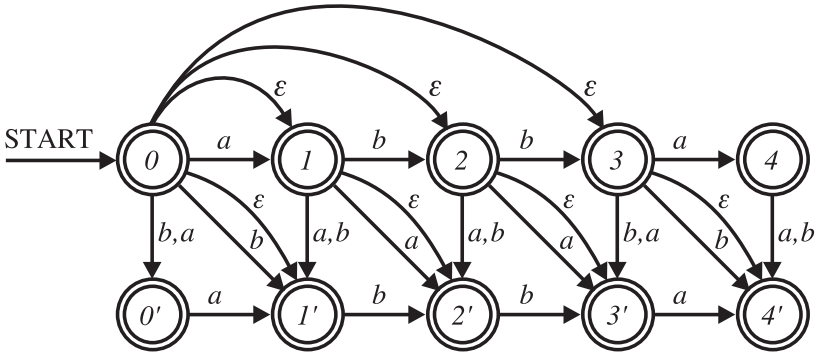


Figure 8.2: Transition diagram of the "Levenshtein" automaton with final states fixed and ε–transitions inserted from Example 8.1

3. We replace all ε–transitions by non–ε–transitions. The resulting automaton has the transition diagram depicted in Fig. 8.3. Its transition table is shown in Tab. 8.1

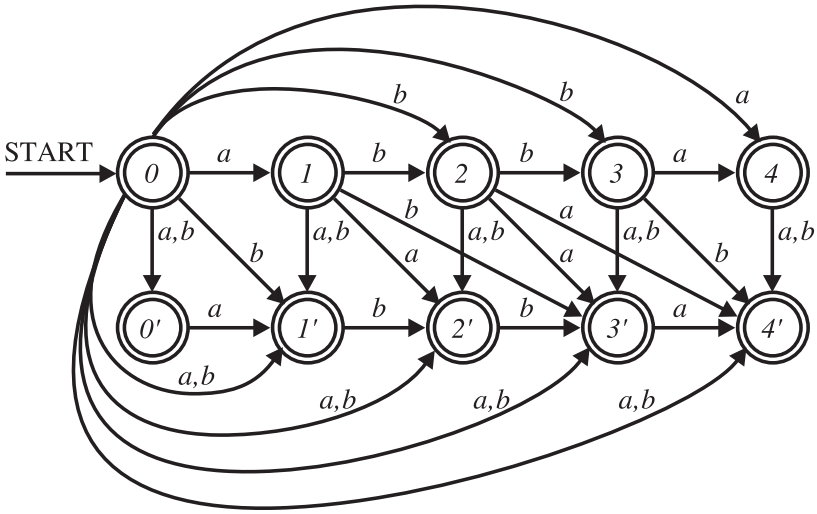


Figure 8.3: Transition diagram of the nondeterministic "Levenshtein" factor automaton after removal of ε–transitions from Example 8.1

4. The final operation is to construct the equivalent deterministic finite automaton. Its transition table is shown in Table 8.2.

	a	b
0	$140'1'2'3'4'$	$230'1'2'3'4'$
1	$1'2'$	$21'3'$
2	$2'3'4'$	$32'$
3	$43'$	$3'4'$
4	$4'$	$4'$
$0'$	$1'$	
$1'$		$2'$
$2'$		$3'$
$3'$	$4'$	
$4'$		

Table 8.1: Transition table of the nondeterministic "Levenshtein" factor automaton from Example 8.1

0	$140'1'2'3'4'$	$230'1'2'3'4'$
$140'1'2'3'4'$	$1'2'4'$	$21'2'3'4'$
$230'1'2'3'4'$	$41'2'3'4'$	$32'3'4'$
$1'2'4'$		$2'3'$
$21'2'3'4'$	$2'3'4'$	$32'3'$
$32'3'$	$43'4'$	$3'4'$
$41'2'3'4'$	$4'2'3'4'$	
$32'3'4'$	$43'4'$	$3'4'$
$43'4'$	$4'$	$4'$
$2'3'$	$4'$	$3'$
$2'3'4'$	$4'$	$3'$
$3'4'$	$4'$	
$3'$	$4'$	
$4'$		

Table 8.2: Transition table of the deterministic "Levenshtein" factor automaton from Example 8.1

The transition diagram of the resulting deterministic "Levenshtein" factor automaton is depicted in Fig. 8.4.

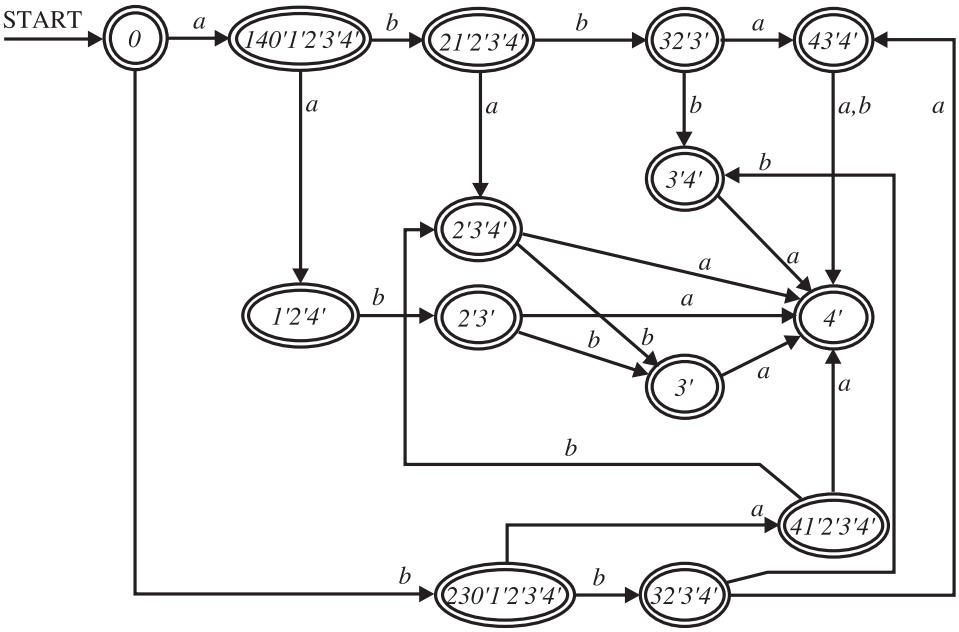


Figure 8.4: Transition diagram of the deterministic "Levenshtein" factor automaton for the string $x = abba$ from Example 8.1

Now we construct the repetition table. We take into account the repetition of factors longer than k (the number of allowed errors). Approximate repetition table R is shown in Table 8.3.

d–subset	Factor	Exact occurrence	Approximate repetitions
21'2'3'4'	ab	2	$(3, abb, O)$
32'3'	abb	3	$(2, ab, O), (3, bb, O)$
43'4'	$abba$	4	$(3, abb, O), (4, bba, O)$
32'3'4'	bb	3	$(2, ab, O), (3, abb, O), (4, ba, O), (4, bba, O)$
41'2'3'4'	ba	4	$(2, b, S), (3, bb, S)$

Table 8.3: Approximate repetition table R for string $x = abba$ with Levenshtein distance $k = 1$ from Example 8.1

9 Approximate repetitions – Δ distance

Let us note that the Δ distance is defined by Def. 2.8. This distance is defined as the local distance for each position of the string. The number of errors is not cumulated as in the previous (and following) cases of finding approximate repetitions. In this section we show solution of $O?N\Delta C$ problem.

Example 9.1
Let the string $x = abbc$ over the ordered alphabet $A = \{a, b, c\}$. We construct an approximate factor automaton using Δ distance equal to one.

1. We construct a finite automaton accepting string x and all strings having Δ distance equal to one. The set of all these strings is denoted $\Delta_1(abbc)$. This finite automaton has the transition diagram depicted in Fig. 9.1.

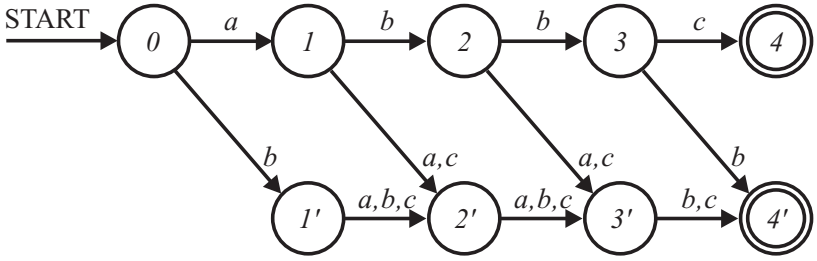


Figure 9.1: Transition diagram of the "Δ" automaton accepting $\Delta_1(abbc)$ with Δ distance $k = 1$ from Example 9.1

2. We use the principle of inserting ε–transitions from state 0 to states 1, 2 and 3 and making all states final states. The transition diagram of the automaton with inserted ε–transitions and fixed final states is depicted in Fig. 9.2.
3. We replace ε–transitions by non–ε–transitions. The resulting automaton has the transition diagram depicted in Fig. 9.3.

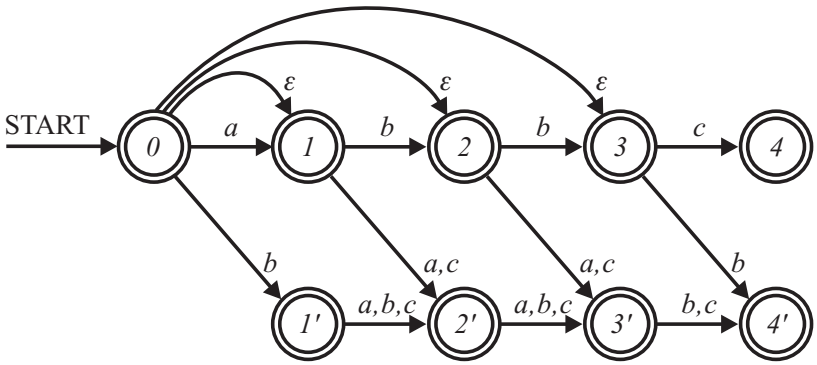


Figure 9.2: Transition diagram of the "Δ" automaton with final states fixed and ε–transitions inserted from Example 9.1

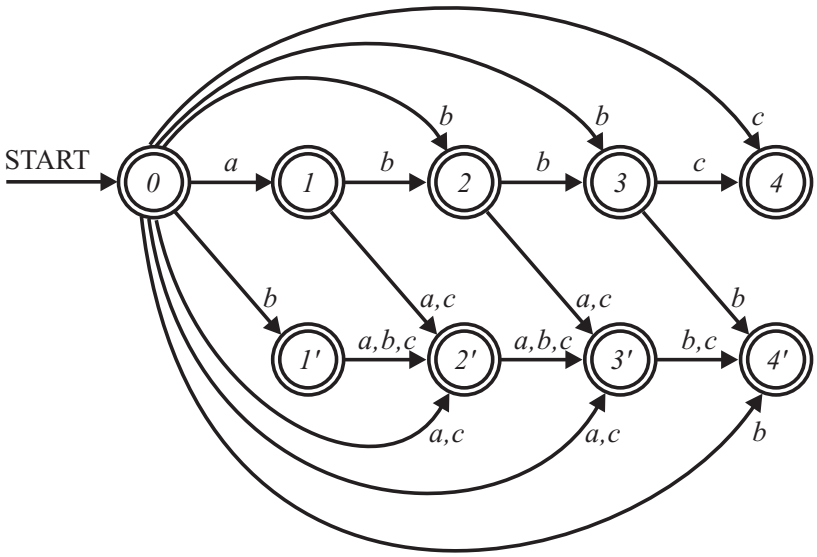


Figure 9.3: Transition diagram of the nondeterministic "Δ" factor automaton after the removal of ε–transitions from Example 9.1

4. The final operation is to construct the equivalent deterministic factor automaton. Table 9.1 is its transition table and its transition diagram is depicted in Fig. 9.4.

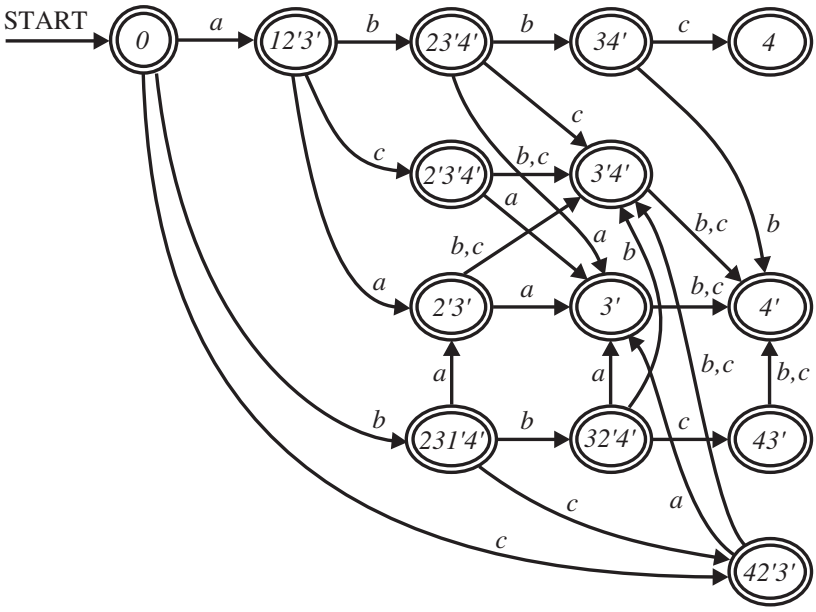


Figure 9.4: Transition diagram of the deterministic "Δ" factor automaton for $x = abbc, \Delta$ distance=1, from Example 9.1

Now we construct the repetition table. We take into account the repetitions of factors longer than the allowed distance. In this case, the distance is equal to 1 and therefore we select repetitions of factors having the length greater or equal to two. Table 9.2 contains information on the Δ–approximate repetitions of factors of the string $x = abbc$. This table is structured according to Def. 6.2. □

	a	b	c
0	$12'3'$	$231'4'$	$42'3'$
$12'3'$	$2'3'$	$23'4'$	$2'3'4'$
$23'4'$	$3'$	$34'$	$3'4'$
$231'4'$	$2'3'$	$32'4'$	$42'3'$
$32'4'$	$3'$	$3'4'$	$43'$
$34'$		$4'$	4
$43'$		$4'$	$4'$
4			
$42'3'$	$3'$	$3'4'$	$3'4'$
$2'3'4'$	$3'$	$3'4'$	$3'4'$
$2'3'$	$3'$	$3'4'$	$3'4'$
$3'4'$		$4'$	$4'$
$3'$		$4'$	$4'$
$4'$			

Table 9.1: Transition table of the deterministic "Δ" factor automaton from Example 9.1

d–subset	Factor	Exact occurrence	Approximate repetitions
23'4'	ab	2	$(3, bb, O), (4, bc, S)$
34'	abb	3	$(4, bbc, O)$
32'4'	bb	3	$(2, ab, O), (4, bc, O)$
42'3'	bc	4	$(2, ab, S), (3, bb, O)$
43'	bbc	4	$(3, abb, O)$

Table 9.2: Approximate repetition table for string $x = abbc$ from Example 9.1

10 Approximate repetitions – Γ distance

Γ distance is defined by Def. 2.8. This distance is defined as a global distance, which means that the local errors are cumulated. In this section we show solution of $O?N\Gamma C$ problem.

Example 10.1
Let the string $x = abbc$ over the ordered alphabet $A = \{a, b, c\}$. We construct an approximate factor automaton using Γ distance equal to two.

1. We construct a finite automaton accepting string x and all strings having Γ distance equal to two. The set of all these strings is denoted $\Gamma_2(abbc)$. This finite automaton has the transition diagram depicted in Fig. 10.1.

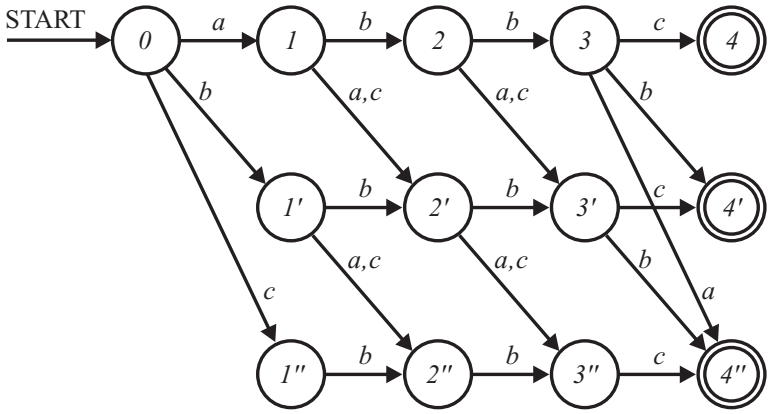


Figure 10.1: Transition diagram of the "Γ" automaton accepting $\Gamma_2(abbc)$ with Γ distance $k = 1$ from Example 10.1

2. Now we insert ε–transitions from state 0 to states 1, 2 and 3 and we make all states final. The transition diagram of the automaton with inserted ε–transitions and fixed final states is depicted in Fig. 10.2
3. We replace the ε–transition by non–ε–transitions. The resulting non-deterministic factor automaton has the transition diagram depicted in Fig. 10.3.
4. The final operation is to construct the equivalent deterministic finite automaton.

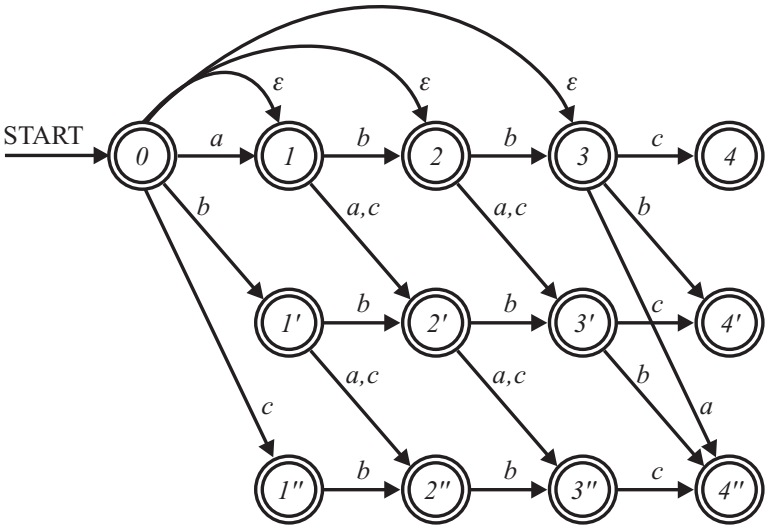


Figure 10.2: Transition diagram of the "Γ" automaton with final states fixed and ε–transitions inserted from Example 10.1

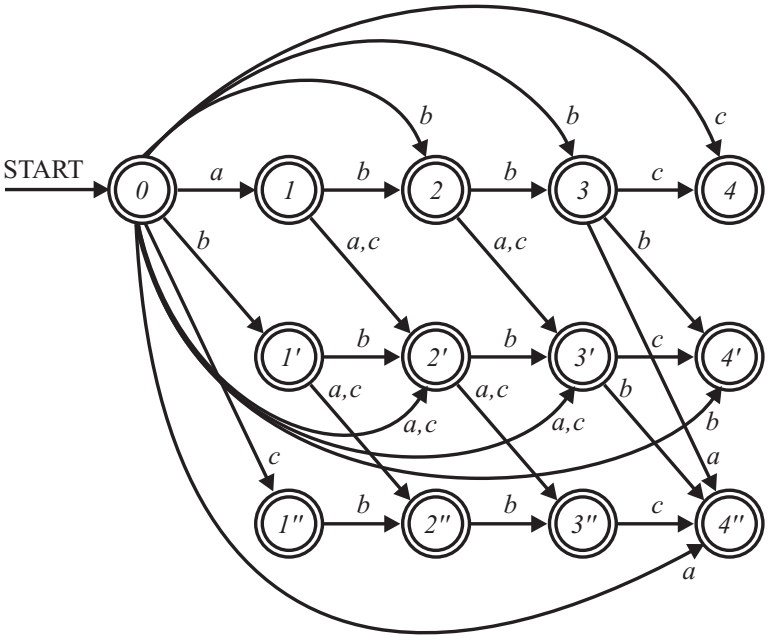


Figure 10.3: Transition diagram of the nondeterministic "Γ" factor automaton after the removal of ε–transitions from Example 10.1

	a	b	c
0	$12'3'4''$	$231'4'$	$42'3'1''$
$12'3'4''$	$2'3''$	$23'4''$	$2'4'3''$
$231'4'$	$3'2''4''$	$32'4'$	$43'2''$
$23'4''$	$3'$	$34''$	$3'4'$
$32'4'$	$3''4''$	$3'4'$	$43''$
$34''$	$4''$	$4'$	4
$42'3'1''$	$3''$	$3'2''4''$	$4'3''$
$43'2''$		$3''4''$	$4'$
$43''$			$4''$
4			
$2'4'3''$	$3''$	$3'$	$3''4''$
$2'3''$	$3''$	$3'$	$3''4''$
$3'2''4''$		$3''4''$	$4'$
$3'4'$		$4''$	$4'$
$3'$		$4''$	$4'$
$4'3''$			$4''$
$4'$			
$3''4''$			$4''$
$3'4''$		$4''$	$4'$
$3''$			$4''$
$4''$			

Table 10.1: Transition table of the deterministic "Γ" factor automaton for the string $x = abbc$ from Example 10.1

Analyzing Table 10.1 we can recognize that the following sets are sets of equivalent states:

$\{2'4'3'', 2'3''\}, \{3'4', 3'\}, \{43''4'3'', 3''4'', 3''\}, \{43'2'', 3'2''4''\}$.

Only states $43''$ and $43'2''$ have an impact on the repetition table. Let us replace all equivalent states by the respective sets. Then we obtain the transition diagram of the optimized deterministic factor "Γ" automaton depicted in Fig. 10.4. Now we construct the repetition table. We take into account the repetitions of factors longer than two (allowed distance). The Table 10.2 contains information on approximate repetition of one factor. The repetition of factor bc indicated by d-subset 43'2" is not included in the repetition table because the length of this factor is $|bc| = 2$ □

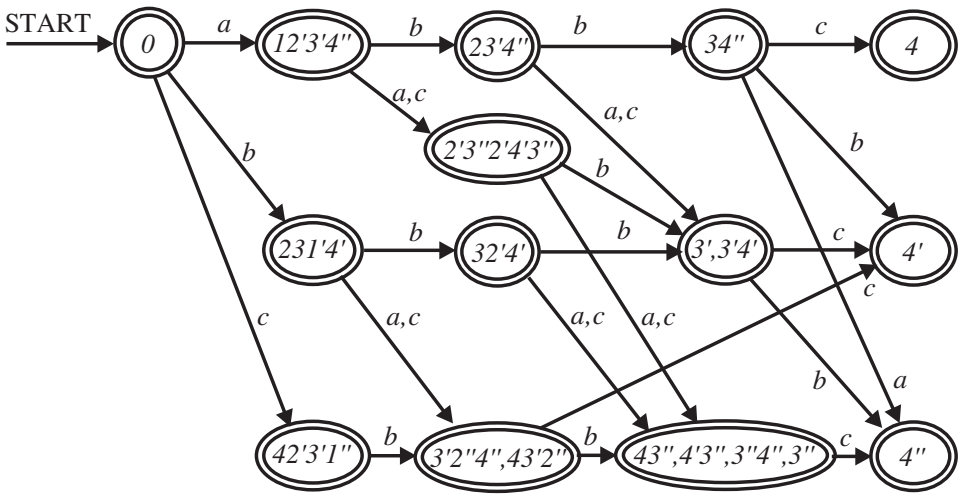


Figure 10.4: Transition diagram of the optimized deterministic "Γ" factor automaton for string $x = abbc$ from Example 10.1

d–subset	Factor	Exact occurrence	Approximate repetitions
34"	abb	3	$(4, bbc, O)$

Table 10.2: Approximate repetition table for string $x = abbc$, Γ distance equal to two, from Example 10.1

11 Approximate repetitions – (Δ, Γ) distance

(Δ, Γ) distance is defined by Def. 2.8. This distance is defined as a global distance, which means that the local errors are cumulated. In this section we show solution of *O?N(*Δ, Γ*)C* problem.

Example 11.1
Let string $x = abbc$ over ordered alphabet $A = \{a, b, c\}$. We construct an approximate factor automaton using (Δ, Γ) distance equal to (1,2).

1. We construct a finite automaton accepting string x and all strings having (Δ, Γ) distance equal to (1,2). The set of all these strings is denoted $(\Delta, \Gamma)_{1,2}(abbc)$. This finite automaton has the transition diagram depicted in Fig. 11.1.

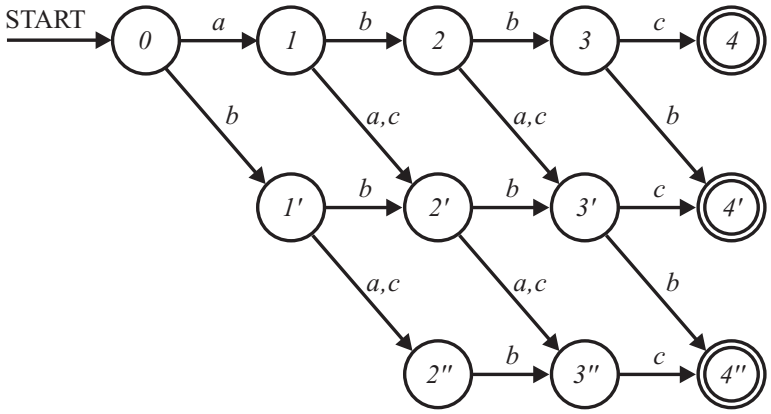


Figure 11.1: Transition diagram of the "(Δ, Γ)" automaton accepting $(\Delta, \Gamma)_{1,2}(abbc)$ with (Δ, Γ) distance $(k, l) = (1, 2)$ from Example 11.1

2. Now we insert ε–transitions from state 0 to states 1, 2 and 3 and we make all states final. The transition diagram of the automaton with inserted ε–transitions and fixed final states is depicted in Fig. 11.2.
3. We replace ε–transitions by non–ε–transitions. The resulting nondeterministic factor automaton has the transition diagram depicted in Fig. 11.3.

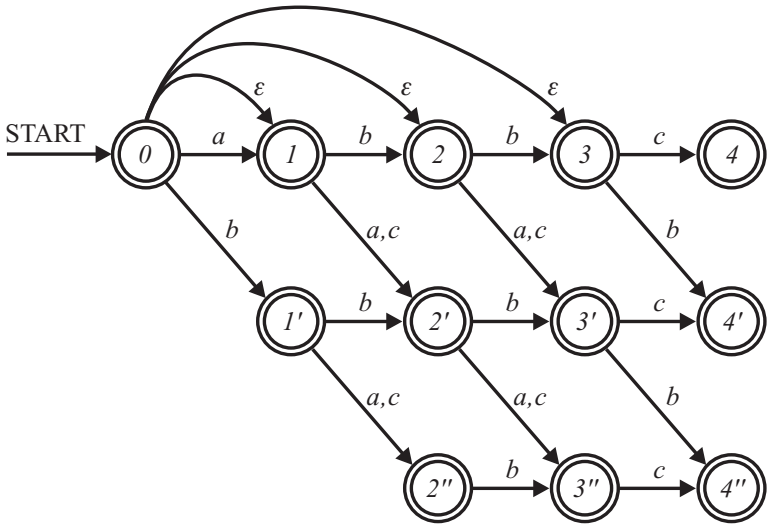


Figure 11.2: Transition diagram of the "(Δ, Γ)" automaton with final states fixed and ε–transitions inserted from Example 11.1

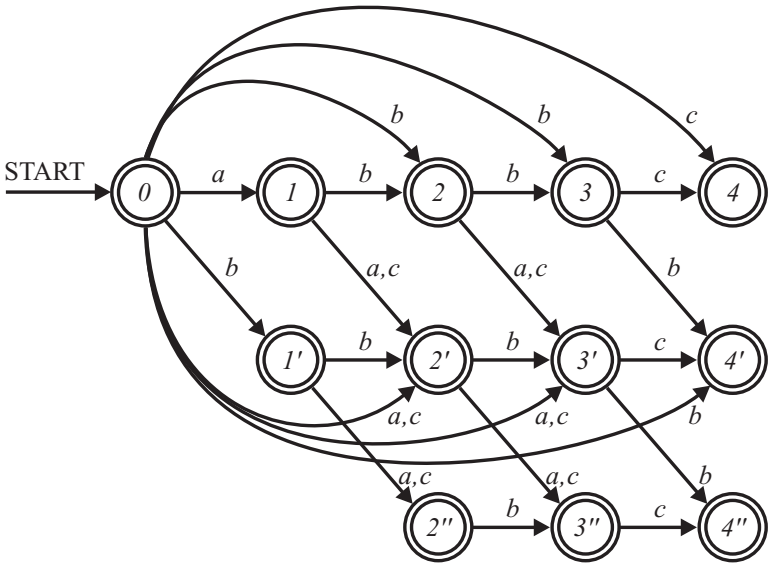


Figure 11.3: Transition diagram of the nondeterministic "(Δ, Γ)" factor automaton after the removal of ε–transitions from Example 11.1

4. The final operation is to construct the equivalent deterministic finite automaton. Table 11.1 is its transition table.

	a	b	c
0	$12'3'$	$231'4'$	$42'3'$
$12'3'$	$2'3''$	$23'$	$2'3''$
$231'4'$	$3'2''$	$32'4'$	$43'2''$
$23'$	$3'$	$34''$	$3'4'$
$32'4'$	$3''$	$3'4'$	$43''$
$34''$		$4'$	4
$42'3'$	$3''$	$3'4''$	$4'3''$
$43'2''$		$3''4''$	$4'$
$43''$			$4''$
4			
$2'3''$	$3''$	$3'$	$3''4''$
$3'2''$		$3''4''$	$4'$
$3'4'$		$4''$	$4'$
$3'4''$		$4''$	$4'$
$3'$		$4''$	$4'$
$4'3''$			$4''$
$4'$			
$3''4''$			$4''$
$3''$			$4''$
$4''$			

Table 11.1: Transition table of the deterministic "Γ, Δ" factor automaton from Example 11.1

Analyzing Table 11.1 we can recognize that the following sets are sets of equivalent states:

$\{3'4', 3', 3'4''\}, \{3''4'', 3'', 43'', 4'3''\}, \{43'2'', 3'2''\}$.

We replace all equivalent states by the respective sets. Then we obtain the transition diagram of the optimized deterministic "(Δ, Γ)" factor automaton depicted in Fig. 11.4.

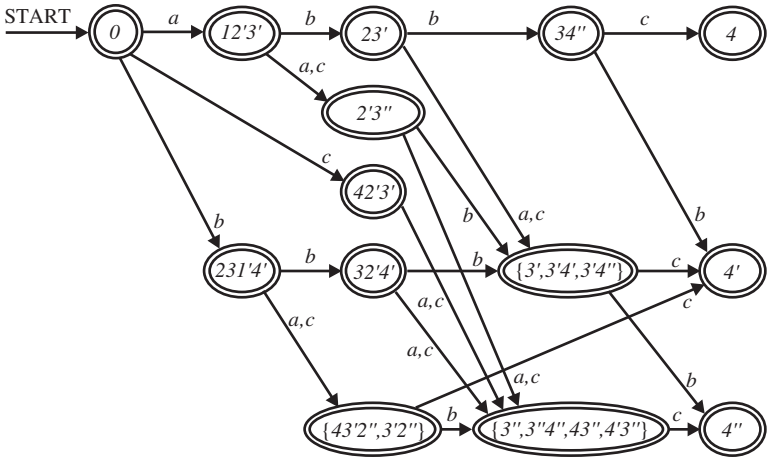


Figure 11.4: Transition diagram of the optimized deterministic "(Δ, Γ)" factor automaton for the string $x = abbc$ from Example 11.1

Now we construct the repetition table. We take into account the repetitions of factors longer than two (allowed distance). Table 11.2 contains information on approximate repetition of one factor. □

d–subset	Factor	Exact occurrence	Approximate repetitions
34"	abb	3	$(4, bbc, O)$

Table 11.2: Approximate repetition table for string $x = abbc$, (Γ, Δ) distance equal to (1,2), from Example 11.1

12 Exact repetitions in one string with don't care symbols

The "don't care" symbol ($\circ$) is defined by Def. 2.9. Next example shows principle of finding repetitions in the case of presence don't care symbols (*O?NED* problem).

Example 12.1
Let string $x = a \circ aab$ over alphabet $A = \{a, b, c\}$. Symbol $\circ$ is the don't care symbol. We construct a don't care factor automaton.

1. We construct a finite automaton accepting set of strings described by string x with don't care symbol. This set is $DC(x) = \{aaaab, abaab, acaab\}$. This finite automaton has the transition diagram depicted in Fig. 12.1.

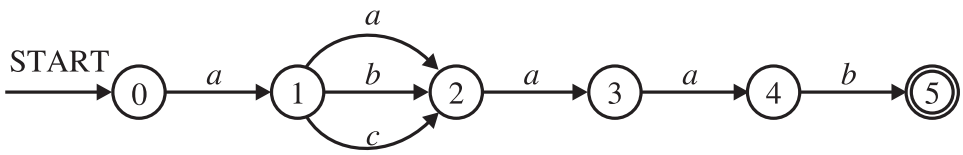


Figure 12.1: Transition diagram of the DC automaton accepting $DC(x)$ from Example 12.1

2. We insert ε-transitions from state 0 to states 1,2,3,4 and we make all states final. Transition diagram of automaton $DC_\varepsilon(a \circ aab)$ with inserted ε-transitions and fixed final states is depicted in Fig. 12.2.

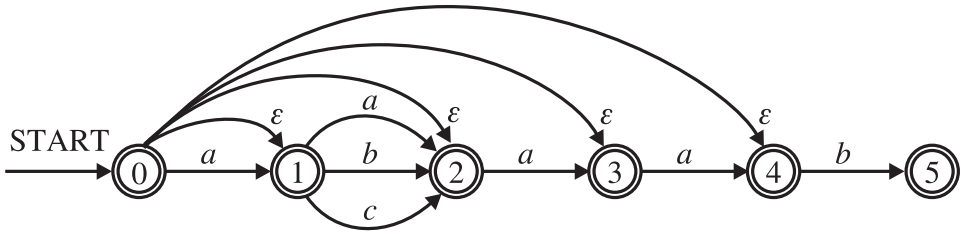


Figure 12.2: Transition diagram of the $DC_\varepsilon(a \circ aab)$ automaton with inserted ε-transitions and fixed final states from Example 12.1

3. We replace ε-transitions by non ε-transitions. Resulting nondeterministic factor automaton $DC_N(a \circ aab)$ has the transition diagram depicted in Fig. 12.3.

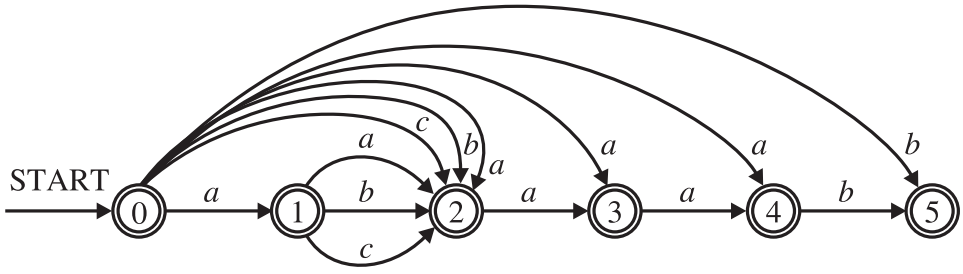


Figure 12.3: Transition diagram of nondeterministic factor automaton $DC_N(a \circ aab)$ after the removal of ε-transitions from Example 12.1

4. The final operation is construction of equivalent deterministic factor automaton $DC_D(a \circ aab)$. Table 12.1 is transition table of the nondeterministic factor automaton having transition diagram depicted in

Fig. 12.3. Table 12.2 is transition table of deterministic factor automaton $DC_D(a \circ aab)$. Transition diagram of deterministic factor automaton $DC_D(a \circ aab)$ is depicted in Fig. 12.4.

	a	b	c
0	$1, 2, 3, 4$	$2, 5$	2
1	2	2	2
2	3		
3	4		
4		5	
5			

Table 12.1: Transition table of nondeterministic factor automaton $DC_N(a \circ aab)$ from Example 12.1

	a	b	c
0	1234	25	2
1234	234	25	2
2	3		
25	3		
234	34	5	
3	4		
34	4	5	
4		5	
5			

Table 12.2: Transition table of deterministic factor automaton $DC_D(a \circ aab)$ from Example 12.1

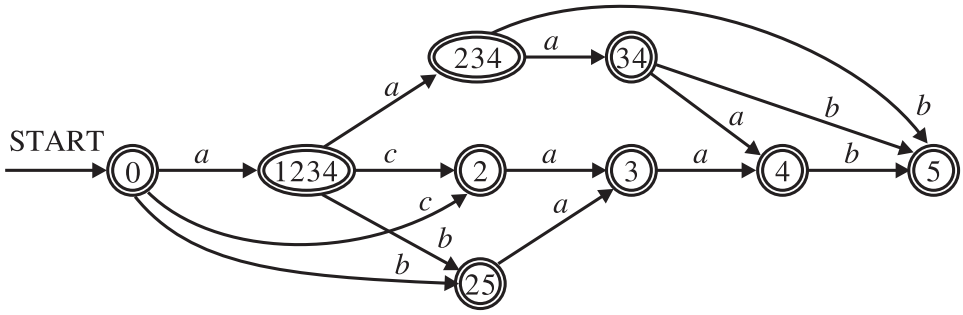


Figure 12.4: Transition diagram of deterministic factor automaton $DC_D(a \circ aab)$ from Example 12.1

The last step is the construction of the repetition table. It is shown in Table 12.3.

d–subset	Factor	First occurrence	Repetitions
1234	a	1	$(2,S),(3,S),(4,S)$
25	ab	2	$(5,G)$
234	aa	2	$(3,O),(4,O)$
34	aaa	3	$(4,O)$

Table 12.3: Repetition table for string $x = a \circ aab$ from Example 12.1

13 Conclusion

We have shown uniform and simple models for finding exact and approximate repetitions in a text. All models are based on the analysis of d-subsets created during the determinization of different types of nondeterministic factor automata. This leads to a very simple and straightforward solution of the repetition problems.

The time complexity is composed of two parts:

- construction of the deterministic factor automaton, which has linear time complexity for exact repetitions, and
- construction of the repetition table having quadratic time complexity for exact repetitions.

The time complexity of finding approximate repetitions is an open problem and should be investigated in detail.

The second topic for further study is the search for a connection between the general models shown here and existing algorithms for particular

repetition problems. This may lead to an understanding of the way how to simulate general models in different cases.

References

[AU71] Aho, A. V., Ullman, J. D.: *The Theory of Parsing, Translation and Compiling*. Vol. 1 Parsing, Vol. 2 Compiling. New York: Prentice-Hall 1971, 1972.

[BBE+87] Blumer, A., Blumer, J., Ehrenfeucht, A., Haussler, D., McConnel, R. *Complete inverted files for efficient text retrieval and analysis*. J. Assoc. Comput. Mach., 34(3):578–595, 1987.

[CCI+99] Cambouropoulos, E., Crochemore, M., Iliopoulos, C. S., Mouchard, L., Pinzon, Y. J.: *Algorithms for Computing Approximate Repetitions in Musical Sequences*. Australian Workshop On Combinatorial Algorithms, 1999.

[CCI+00] Iliopoulos, C. S., Lecroq, T., Mouchard, L., Pinzon, Y. J.: *Computing Approximate Repetitions in Musical Sequences*. Proceedings of the Prague Stringology Club Workshop '2000, Department of Computer Science, CTU, Prague, 2000.

[Cr86] Crochemore, M.: *Transducers and Repetitions*. Theoretical Computer Science, 1986, pp. 63-86.

[CR94] Crochemore, M., Rytter, W.: *Text algorithms*. Oxford University Press, 1994.

[HU79] Hopcroft, J. E., Ullman, J. D.: *Introduction to Automata, Languages and Computations*. Addison-Wesley, Reading, MA, 1979.

[Ma79] Main, M.: *An $\mathcal{O}(n \log n)$ algorithm for finding repetition in a string*. Master's thesis. Washington State University, 1979.

[Me02] Melichar, B. (jr.): *Repetitions in strings*. Master's thesis. Czech Technical University in Prague, 2002.

Extension of Selected ADFA Construction Algorithms to the Case of Cyclic Automata

Jan Daciuk
Department of Knowledge Engineering
Faculty of Electronics, Telecommunications and Informatics
Gdańsk University of Technology
e-mail: jandac@eti.pg.gda.pl

November 10, 2004

Abstract

In a recent paper, Carrasco and Forcada present an incremental algorithm for adding words to a minimal deterministic cyclic automaton. The algorithm is an extension of an incremental construction algorithm for acyclic deterministic automata (ADFAs). We present similar extensions of two other construction algorithms for ADFAs. Although the extensions were published in May and June 2004, we expose similarities between them, and we also provide a different, much simpler reformulation of one of them.

1 Introduction

Minimal finite-state automata (FSAs) are an ideal choice for representation of dictionaries in natural language processing (NLP). In addition to fast processing of words, they take little space once they are constructed. However, traditional construction needs much space. Incremental and semi-incremental methods build automata word-by-word (or string-by-string), reducing the number of states after each word is added to the language of the automaton. Incremental methods keep the automaton minimal (possibly except for as many states as the length of the longest word plus one) during construction, semi-incremental methods are less strict.

As the size of available computer memories constantly and quickly increases, and their prices fall, questions are raised whether memory-efficiency is still an issue. However, new applications require much more memory than the existing ones. They are developed precisely because more memory becomes available. The technological progress also results in new devices with smaller memories, like laptops, palmtops, etc.

Traditional methods for construction of acyclic automata construct a trie, and then minimize it. Incremental and semi-incremental methods merge both

processes so that they overlap. Traditional construction methods for cyclic automata are more complicated. First, a non-deterministic automaton is constructed, then it is determinized, and finally minimized. It is harder to design corresponding incremental and semi-incremental algorithms. An easier job is to extend existing algorithms for acyclic automata to the case when individual words are added to the language of a cyclic automaton. One such extension has been proposed by Carrasco and Forcada in [2]. We have proposed similar extensions of other algorithms [3], [4]. This paper exposes similarities between them. It also offers a new formulation of one of them.

The rest of the paper is organized as follows. Section 2 introduces basic definitions. The main points of an extension of an incremental construction algorithm for unsorted data is sketched in Section 3. Short descriptions of algorithms we extend, i.e. the incremental algorithm for sorted data, and the Watson's algorithm, can be found in Section 4 and Section 5 respectively. The corresponding extensions are described in Section 6 and Section 7. The similarities in both extensions are exposed in Section 8.

2 Mathematical Preliminaries

We define a deterministic finite-state automaton as $M = (Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, Σ is a finite set of symbols called the alphabet, $q_0 \in Q$ is the start (or initial) state, and $F \subseteq Q$ is a set of final (accepting) states. We define the transition function $\delta : Q \times \Sigma \to Q$ as a partial mapping. If $\delta(q, a) \notin Q$ for some $q \in Q$ and $a \in \Sigma$, we write $\delta(q, a) = \perp$. The extended transition function $\delta^* : Q \times \Sigma^* \to Q$ has a string (including the empty string ϵ) as an argument, and is defined as:

$$\begin{aligned} \delta^*(q, \epsilon) &= q \\ \delta^*(q, ax) &= \delta^*(\delta(q, a), x) \end{aligned}$$

The right language of a state q is defined as:

$$\overrightarrow{\mathcal{L}}(q) = \{x \in \Sigma^* : \delta^*(q, x) \in F\}$$

The language of the automaton $\mathcal{L}(M) = \overrightarrow{\mathcal{L}}(q_0)$. The right language can be defined recursively:

$$\overrightarrow{\mathcal{L}}(q) = \bigcup_{a \in \Sigma : \delta(q,a) \neq \perp} a \cdot \overrightarrow{\mathcal{L}}(\delta(q, a)) \cup \begin{cases} \{\epsilon\} & \text{if } q \in F \\ \emptyset & \text{otherwise} \end{cases}$$

Equality of right languages is an equivalence relation that partitions the set of states into abstraction classes (equivalence classes). The minimal automaton is the unique automaton (up to isomorphisms) that has the minimal number of states among automata recognizing the same language. It is also the automaton

where all states are useful (i.e. reachable from the start state, and from which a final state can be reached), and each equivalence class has exactly one member.

$$(q \equiv p) \iff (\overrightarrow{\mathcal{L}}\ (q) = \overrightarrow{\mathcal{L}}\ (p))$$

The length of a string $w \in \Sigma^*$ is denoted $|w|$, and the i-th symbol (starting from one) in the string w as w_i. A substring of a string w being a concatenation of $w_i w_{i+1} \dots w_{j-1} w_j$ is denoted as $w_{i\dots j}, 1 \leq i \leq j \leq |w|$.

The longest common prefix p (of the word w to be added to the language of an automaton M and the language of the automaton $\mathcal{L}(M)$) is the longest string $p \in \Sigma^* : \exists_{s \in \Sigma^*}\ w = ps \wedge \exists_{r \in \Sigma^*}\ pr \in \mathcal{L}(M)$. We denote the longest common prefix of strings w and v as $w \wedge v$, and the longest common prefix of a string w and a language $\mathcal{L}$ as $w \wedge \mathcal{L}$. The longest common prefix path is a set of states $\{q \in Q : q = \delta^*(q_0, s), p = sr, r \in \Sigma^*\}$ visited while recognizing the longest common prefix p.

A *confluence state* is a state with more than one in-transition.

3 Extension by Carrasco and Forcada

Two incremental algorithms for construction of minimal, acyclic, deterministic finite-state automata were presented in [6], and later in [5]. The first algorithm takes input lexicographically sorted, hence the nickname "the sorted data algorithm". The second algorithm accepts data in any order – "the unsorted data algorithm".

In [2], Carrasco and Forcada extended the unsorted data algorithm so that it adds strings to a cyclic automaton. They propose a two-stage construction process for cyclic automata. First, the cyclic core is constructed. It contains all cycles that are to be found in the target automaton. Then, strings are added to the language of the automaton.

Carrasco and Forcada derive their algorithm from the union of an automaton $M = (Q, \Sigma, \delta, q_0, F)$ with a single string automaton $M_w = (Q_w, \Sigma, \delta_w, q_{0w}, F_w)$, $\mathcal{L}(M_w) = w$. We will not go into details of that construction; the reader is referred to [2] for more information on the subject. We want to highlight the main points of their algorithm that make it different from the original algorithm for acyclic automata:

1. Before the algorithm starts, there is already a minimal, deterministic, cyclic automaton.
2. All states of the initial automaton are already in the Register (a set of states that have a unique right language in the automaton, i.e. a set of pairwise inequivalent states).
3. The start state is cloned (a copy of it with identical out-transitions is created).
4. The longest common prefix path is cloned.

4 Incremental Algorithm for Sorted Data

The incremental algorithm for sorted data is the first algorithm in [6] and [5]. The algorithm benefits from the fact that input strings are lexicographically sorted.

```
 1: function main;
 2:     R ← ∅; w′ ← ϵ;
 3:     while there is another word do
 4:         w ← next word;
 5:         p ← w ∧ w′; j ← |p|;
 6:         s ← δ*(q0, p);
 7:         v ← w_{j+1...|w|}; v′ ← w′_{j+1...|w′|};
 8:         if v ≠ ϵ then
 9:             replace_or_register(s, v′);
10:         end if;
11:         add_suffix(s, v); w′ ← w;
12:     end while;
13:     replace_or_register(q0, v′);
14: end function;

15: function replace_or_register(q, w);
16:     c ← δ(q, w1);
17:     if w_{2...|w|} ≠ ϵ then
18:         replace_or_register(c, w_{2...|w|});
19:     end if;
20:     if ∃_{r∈R} r ≡ q then
21:         δ(q, w1) ← r;
22:         delete c;
23:     else
24:         R ← R ∪ {c};
25:     end if;
26: end function;

27: function add_suffix(q, w);
28:     while w ≠ ϵ do
29:         δ(q, w1) ← new state;
30:         q ← δ(q, w1);
31:         w ← w_{2...|w|};
32:     end while;
33:     F ← F ∪ {q};
34: end function;
```

In contrast to the original formulation of the algorithm in [6] and [5], function *replace_or_register* takes an additional argument: the part of the previously

added string that is not included in the longest common prefix. Rather than looking for the last transition of the state being the first argument, we are now able to directly choose the transition based on its label. Both versions are equivalent, but the one presented here is closer to the extension presented in the paper.

In the listing, R represents the Register – a set of states that have a unique right language in the automaton. Instead of traversing the whole automaton in search for an equivalent state, it is sufficient to look for it in the Register. The Register lookup can be implemented in constant time. The Register is used in all algorithms mentioned in this paper. Lines 5 and 6 calculate the longest common prefix and the last state of the longest common prefix path. Line 7 calculates the part v that follows the longest common prefix p in the string to be added w, and the part v' that follows p in the string w' last added to the language of the automaton.

It is worth noting that confluence states are never found in the longest common prefix path, i.e. the path in the automaton traversed when adding a new string. Additional in-transitions are created only by the function *replace_or_register*. They can lead only to states that are already in the Register R. Such states are never in the longest common prefix path p. The input is lexicographically sorted. The longest common prefix of two adjacent words in the input is never shorter than the longest common prefix of the first of those words and any word that comes later in the input stream. Therefore, states representing v' will never be again in the longest common prefix path, and v' is an argument of the function *replace_or_register*. Functions *replace_or_register* and *add_suffix* are independent of each other; they can be called in any order.

5 Watson's Algorithm

The Watson's algorithm (see [9] for details) accepts strings in any order of decreasing length. As in the sorted data algorithm, the longest common prefix path is followed (lines 11–13), and then a chain of states and transitions is built to recognize the suffix if there is one (lines 14–18). If there is no suffix (the state returned by the function *add_word* – the last state of the longest common prefix path – has some out-transitions), the string is a prefix of another string already in the language of the automaton. Since strings come in decreasing lengths, all states reachable from the last state in the longest common prefix path will never be in the longest common prefix path of any string to be added later. Therefore, they can be considered for minimization. If an equivalent state is found in the register R, it replaces the current state, and in-transitions of the current state redirected to the equivalent one. That redirection of transitions creates confluence states, but they can never appear in the longest common prefix path, as that path cannot be longer than subsequent strings coming in the order of decresing length. If no equivalent state is found, the current state is added to the Register.

1: **function** main;

```
 2:     R ← ∅;
 3:     while there is another word then
 4:         w ← next word;
 5:         minim(R, build_stack(add_word(w), []));
 6:     end while;
 7:     minim(R, build_stack(q_0, []));
 8: end function;

 9: function add_word(w);
10:     q = q_0; i ← 0;
11:     while i < |w| and δ(q, w_i) ≠ ⊥ do
12:         q ← δ(q, w_i); i ← i + 1;
13:     end while;
14:     while i < |w| do
15:         δ(q, w_i) ← new state;
16:         q ← δ(q, w_i); i ← i + 1;
17:     end while;
18:     F ← F ∪ {q};
19:     return q;
20: end function;

21: function build_stack(q, X);
22:     push(X, q);
23:     for a ∈ Σ : δ(q, a) ≠ ⊥ do
24:         if δ(q, a) ∉ F then
25:             X ← build_stack(δ(q, a), X);
26:         end if;
27:     end for;
28:     return X;
29: end function;

30: function minim(R, S);
31:     while S ≠ [] do
32:         q ← pop(S);
33:         if ∃_{r∈R} q ≡ r then
34:             replace q with r;
35:         else
36:             R ← R ∪ {q};
37:         end if;
38:     end while;
39: end function
```

Line 34 in function *minim* means that state q is deleted, and all transitions leading to that state are redirected to state r. Note that such operation is difficult to implement without storing in-transitions. In practice, the stack S should contain transitions rather than states. Function *build_stack* (lines 21–29)

forms a team with function *minim* (30–39). It fulfills the same role as function *replace_or_register* in the sorted data algorithm. Actually, those two functions can be rewritten as *replace_or_register2*:

```
21: function replace_or_register2(q);
22:     for a ∈ Σ : δ(q, a) ≠ ⊥ do
23:         if δ(q, a) ∉ F then
24:             replace_or_register2(δ(q, a));
25:         end if;
26:     end for;
27:     if ∃_{r∈R} q ≡ r then
28:         replace q with r;
29:     else
30:         R ← R ∪ {q};
31:     end if;
32: end function;
```

6 Extension of the Sorted Data Algorithm

The extension of the sorted data algorithm is presented here differently than in [3]. The present version is simpler.

```
 1: function main;
 2:     R ← Q; r ← clone(q_0); w' ← ε;
 3:     while not eof do
 4:         w ← next_word; i ← 1; q ← r;
 5:         p ← w ∧ w'; j ← |p|;
 6:         while i < |w| and δ(q, w_i) ≠ ⊥ and fanin(δ(q, w_i)) < 2 do
 7:             q ← δ(q, w_i); i ← i + 1;
 8:         end while;
 9:         while i < |w| and δ(q, w_i) ≠ ⊥ do
10:             δ(q, w_i) ← clone(δ(q, w_i));
11:             q ← δ(q, w_i); i ← i + 1;
12:         end while;
13:         if |w'| ≥ j then
14:             replace_or_register(δ(r, w_{1...j−1}, w'_{j...|w'|});
15:         end if;
16:         add_suffix(q, w_{i...|w|}); w' ← w;
17:     end while;
18:     replace_or_register(r, w');
19:     if r ≠ q_0 then
20:         delete_branch(q_0);
21:     end if;
22: end function;
```

Functions *replace_or_register* and *add_suffix* are the same as in the version for acyclic automata. Function *fanin* returns the number of in-transitions of its argument. Function *delete_branch* deletes its argument state and all states reachable from it without traversing confluence states. The main difference between the original version and the extension is the presence of the loop in lines 9–12, and the additional condition for the loop in lines 6–8. Both loops traverse the longest prefix path. The first loop (lines 6–8) processes only non-confluence states. It is the corresponding loop from the original algorithm augmented with the condition that makes sure that the states are not confluence. The second loop handles confluence states. Each such state is cloned. When a state is cloned – an exact copy of it is created, the object of cloning and its result have an identical suite of out-transitions. Hence the targets of the transitions become confluence states themselves. Once we find a confluence state in the longest common prefix path, all states that follow it become confluence states themselves.

Another two differences are in line 2. Since we add strings to the language of an existing (cyclic) automaton, the Register cannot initially be empty. It is set to the set of all states of the automaton. The second difference is that we clone the start state, and the clone becomes the new start state of the automaton. Thus initially, all state in the longest common prefix path are confluence states. The loop in lines 9–12 creates their non-confluence clones.

Confluence states constitute a border between the pre-existing part of the automaton, and the new part created by adding new strings. By cloning states from the old part, we recreate conditions for the original algorithm that assumes no confluence states.

7 Extension of Watson's Algorithm

Because function *minim* is identical to the corresponding function in the original algorithm, it is omitted here. Note that instead of using the pair of functions *build_stack* and *minim*, we could use a modified version of *replace_or_register2*.

```
 1: function main;
 2:     R ← Q; r ← clone(q_0);
 3:     while there is another word do
 4:         w ← next word;
 5:         minim(R, build_stack(add_word(w), []));
 6:     end while;
 7:     minim(R, build_stack(q_0, [])); q_0 ← r;
 8: end function;

 9: function add_word(w);
10:     q = r; i ← 0;
11:     while i < |w| and δ(q, w_i) ≠ ⊥ and fanin(δ(q, w_i)) < 2 do
12:         q ← δ(q, w_i); i ← i + 1;
```

```
13:     end while;
14:     while i < |w| and δ(q, w_i) ≠ ⊥ do
15:         δ(q, w_i) ← clone(δ(q, w_i));
16:         q ← δ(q, w_i); i ← i + 1;
17:     end while;
18:     while i < |w| do
19:         q ← build_state(q, w_i); i ← i + 1;
20:     end while;
21:     F ← F ∪ {q};
22:     return q;
23: end function;

24: function build_stack(q, X);
25:     push(X, q);
26:     for a ∈ Σ : δ(q, a) ≠ ⊥ do
27:         if δ(q, a) ∉ F ∧ fanin(δ(q, a)) < 2 then
28:             X ← build_stack(δ(q, a), X);
29:         end if;
30:     end for;
31:     return X;
32: end function;
```

The extension is carried out in the same way as for the sorted data algorithm. In line 2, the Register is set to the set of states of the initial automaton. The start state is cloned. The loop in lines 11–13 has an additional condition that makes sure that state are not confluence. The loop that follows in lines 14–17 processes confluence states by cloning them. It provides a non-confluence copy of confluence states, so that the rest of the algorithm can behave very much as in the original. The final difference is an additional condition in function *build_stack*. Confluence states are not put onto stack. As in the extension for the sorted data algorithm, the confluence states form a boundary between the old and the new part of the automaton under construction.

8 Conclusions

We have presented two algorithms for incremental addition of strings to a cyclic automaton. They are extensions of the incremental algorithm for the construction of minimal, acyclic, deterministic, finite-state automata from sorted data [6], [5], and semi-incremental Watson's algorithm [9]. Both extensions have already been described in [3] and [4] respectively. They share a number of features with the extension by Carrasco and Forcada [2] of the incremental algorithm for the construction of minimal, acyclic, deterministic, finite-state automata from unsorted data [6], [5], [1], [8], [7]. All extensions assume that there is an existing, cyclic automaton, to the language of which we add new strings. All states of

the automaton are already in the Register. In both extensions presented here confluence states play a role of markers for the boundary between the old part of the automaton – the one that existed before the start of the algorithm, and the new part created as the result of the algorithm. Cloning those states restores original conditions for the algorithms that form the base of the extensions. To start the process of cloning, the start state is cloned, which makes target states of all its out-transitions confluence states.

Modifications that lead to extensions that are described here are small. They include cloning the start state, cloning the confluence states in the longest common prefix path, dividing one loop into two, and providing additional conditions for testing. The worst-case complexity is the same as in the acyclic case.

Both extensions require the data to be sorted (differently for each algorithm). They use this feature to perform the construction more efficiently than the algorithm provided by Carrasco and Forcada. States created by adding new strings to the language of the automaton are compared to states in the Register only once. This makes both extensions faster than the one by Carrasco and Forcada. Although the algorithms themselves are faster, sorting also takes time. However, in case of the extension of the Watson's algorithm, the data can be kept in files that hold strings of the same length. Carrasco and Forcada emphasize on-line maintenance of dictionaries; they also provide an algorithm for deletion of strings. Their model is suitable for dictionary developers who need to constantly update dictionaries. A different, off-line model is possible. In that model, dictionaries are updated only from time to time. The result of the construction process is not kept in memory directly afterwards. Automata are first compressed to minimize memory use. In such a model, our extensions do have an advantage over Carrasco and Foracada's algorithm. In the on-line model for dictionary developers, our algorithms can also prove useful. The strings to be added may come in small sorted chunks. In such case, our algorithms would update the automaton faster than the more general algorithm presented by Carrasco and Forcada.

9 Acknowledgements

The algorithms described in [3] and [4], and presented in this paper, were conceived during the author's stay in Alfa-informatica, Rijksuniversiteit Groningen, where he took part in the PIONIER Project *Algorithms for Linguistic Processing*, funded by NWO (Dutch Organization for Scientific Research) and the University of Groningen.

References

[1] J. Aoe, K. Morimoto, and M. Hase. An algorithm for compressing common suffixes used in trie structures. *Trans. IEICE*, 1992.

[2] Rafael C. Carrasco and Mikel L. Forcada. Incremental construction and maintenance of minimal finite-state automata. *Computational Linguistics*, 28(2), June 2002.

[3] Jan Daciuk. Comments on incremental construction and maintenance of minimal finite-state automata by Rafael C. Carrasco and Mikel Forcada. *Computational Linguistics*, 30(2):227–235, 2004.

[4] Jan Daciuk. Semi-incremental addition of strings to a cyclic finite automaton. In Krzysztof Trojanowski Mieczysław A. Kłopotek, Sławomir T. Wierzchoń, editor, *Proceedings of the International IIS: IIP WM'04 Conference*, Advances in Soft Computing, pages 201–207, Zakopane, Poland, May 2004. Springer.

[5] Jan Daciuk, Stoyan Mihov, Bruce Watson, and Richard Watson. Incremental construction of minimal acyclic finite state automata. *Computational Linguistics*, 26(1):3–16, April 2000.

[6] Jan Daciuk, Richard E. Watson, and Bruce W. Watson. Incremental construction of acyclic finite-state automata and transducers. In Kemal Oflazer and Lauri Karttunen, editors, *Finite State Methods in Natural Language Processing*, Bilkent University, Ankara, Turkey, June – July 1998.

[7] Dominique Revuz. Dynamic acyclic minimal automaton. In *CIAA 2000, Fifth International Conference on Implementation and Application of Automata*, pages 226–232, London, Canada, July 2000.

[8] K. Sgarbas, N. Fakotakis, and G. Kokkinakis. Two algorithms for incremental construction of directed acyclic word graphs. *International Journal on Artificial Intelligence Tools, World Scientific*, 4(3):369–381, 1995.

[9] Bruce Watson. A fast new (semi-incremental) algorithm for the construction of minimal acyclic DFAs. In *Third Workshop on Implementing Automata*, pages 91–98, Rouen, France, September 1998. Lecture Notes in Computer Science, Springer.

A Note on Join and Auto-Intersection of n-ary Rational Relations

Andre Kempe[1] Jean-Marc Champarnaud[2] Jason Eisner[3]

[1] Xerox Research Centre Europe – Grenoble Laboratory
6 chemin de Maupertuis – 38240 Meylan – France
Andre.Kempe@xrce.xerox.com – http://www.xrce.xerox.com

[2] Université de Rouen Faculté des Sciences et des Techniques
76821 Mont-Saint-Aignan – France
Jean-Marc.Champarnaud@univ-rouen.fr

[3] Johns Hopkins University – Computer Science Department
3400 N. Charles St. – Baltimore, MD 21218 — United States
jason@cs.jhu.edu – http://www.cs.jhu.edu/∼jason/

Abstract

A finite-state machine with n tapes describes a rational (or regular) relation on n strings. It is more expressive than a relational database table with n columns, which can only describe a *finite* relation.

We describe some basic operations on n-ary rational relations and propose notation for them. (For generality we give the semiring-weighted case in which each tuple has a weight.) Unfortunately, the join operation is problematic: if two rational relations are joined on more than one tape, it can lead to non-rational relations with undecidable properties. We recast join in terms of "auto-intersection" and illustrate some cases in which difficulties arise. We close with the hope that partial or restricted algorithms may be found that are still powerful enough to have practical use.

1 Introduction

Multi-tape finite-state machines (FSMs) (Rabin and Scott, 1959; Elgot and Mezei, 1965; Kay, 1987; Kaplan and Kay, 1994) are a natural generalization of the familiar one- and two-tape cases, known respectively as finite-state acceptors and transducers.

An n-tape FSM characterizes n-tuples of strings. The set of tuples that it accepts is called an n-ary relation. If the FSM is weighted, it defines a weighted n-ary relation that assigns each n-tuple a weight (in some semiring), such as a probability.

The relations defined by FSMs are known as *rational* (or *regular*) relations. Our interest in n-tuples stems from our view of these relations as relational databases. In the familiar case $n = 2$, a finite-state transducer can be regarded as a kind of (weighted) database of string pairs—for example, ⟨spelling, pronunciation⟩, ⟨French word, English word⟩, or ⟨parent concept, child concept⟩. An acyclic transducer can represent any finite database of this sort. Shared substrings can make the representation particularly efficient: a hypothesis lattice for speech processing (Mohri, 1997) represents exponentially many pairs in linear space.

Unlike a classical database, a transducer may even define infinitely many pairs. For example, it may characterize the pattern of the spelling-pronunciation relationship in such a way that it can map even a novel word's spelling to zero or more possible pronunciations (with various weights), and vice-versa. Another transducer may attempt to map not just a word but a sentence of unbounded length to an annotated, corrected, or translated version.

On this database view, it is natural to consider relations with more than 2 columns. In natural language processing, multi-tape machines have recently been used to represent lattices of ⟨speech, gesture, interpretation⟩ triples for processing multimodal input (Bangalore and Johnston, 2000). They have also been used in the morphological analysis of Semitic languages, using multiple tapes to synchronize the vowels, consonants, and templatic pattern into a surface form (Kay, 1987; Kiraz, 2000). They may be similarly useful for coordinating the multiple tiers of autosegmental phonology or articulator-based speech recognition (Livescu, Glass, and Bilmes, 2003).

Unfortunately, one pays a price for allowing infinite multi-column databases. Finite-state methods derive their power from a *rational* algebra, which can combine simple FSMs using operations such as union, closure, and composition. Databases similarly derive their power from a *relational* algebra. Cyclic FSMs are closed under the rational operations, but not under the relational operations, as finite databases are. For example, transducers are not closed under intersection (Rabin and Scott, 1959).

In this paper, we give a formal discussion of semiring-weighted n-ary relations (Section 2). We define several useful operators (Section 3), offering useful notation and taking care to distinguish cases that preserve the rationality of relations from those that do not.

The focus of the paper is a database join operator $\bowtie$ that generalizes intersection, composition, and cross product (Section 3.3). Certain cases of join (single-tape or finite) are guaranteed to preserve rationality and appear practically useful.

In Section 3.4, we reduce the join problem to a somewhat simpler problem of "auto-intersection" (Kempe, Guingne, and Nicart, 2004). In Section 4, we illustrate how auto-intersecting two tapes of a rational relation may produce a variety of non-rational weighted or unweighted relations, including context-sensitive languages whose emptiness is undecidable. We leave open the possibility that there may exist a partial or approximate algorithm with enough coverage to have some practical use.

2 Definitions

After recalling the basic definitions of a monoid and a semiring, we define n-ary weighted relations and n-tape weighted finite-state machines. Our definitions follow the usual definitions for multi-tape finite-state automata (Elgot and Mezei, 1965; Eilenberg, 1974), with semiring weights added just as for acceptors and transducers (Kuich and Salomaa, 1986; Mohri, Pereira, and Riley, 1998).

2.1 Semirings

A *monoid* is a structure $\langle M, \circ, \bar{1} \rangle$ consisting of a set M, an associative binary operation $\circ$ on M, and a *neutral* element $\bar{1}$ such that $\bar{1} \circ a = a \circ \bar{1} = a$ for all $a \in M$. A monoid is called *commutative* iff $a \circ b = b \circ a$ for all $a, b \in M$.

A *semiring* is a structure $\mathcal{K} = \langle \mathbb{K}, \oplus, \otimes, \bar{0}, \bar{1} \rangle$ consisting of a set $\mathbb{K}$, two binary operations, $\oplus$ (*collection*) and $\otimes$ (*extension*), and two neutral elements, $\bar{0}$ and $\bar{1}$, that satisfies the following properties:

- $\langle \mathbb{K}, \oplus, \bar{0} \rangle$ is a commutative monoid
- $\langle \mathbb{K}, \otimes, \bar{1} \rangle$ is a monoid
- extension is *left-* and *right-distributive* over collection:
  $a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)\,, \quad (a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c)\,, \quad \forall a, b, c \in \mathbb{K}$

- $\bar{0}$ is an annihilator for extension: $\bar{0} \otimes a = a \otimes \bar{0} = \bar{0}\,, \quad \forall a \in \mathbb{K}$

Examples of semirings are:

1. $\langle \{\text{FALSE}, \text{TRUE}\}, \vee, \wedge, \text{FALSE}, \text{TRUE} \rangle$: the boolean semiring, which can be used to define unweighted relations and machines.
2. $\langle \mathbb{N}, +, \times, 0, 1 \rangle$: a non-negative integer semiring.
3. $\langle \mathbb{R}_{\geq 0}, +, \times, 0, 1 \rangle$: a non-negative real semiring that can be used to model probabilities.
4. $\langle \mathbb{R}_{\geq 0} \cup \{\infty\}, \min, +, \infty, 0 \rangle$: a "tropical" semiring, sometimes used to model negative logarithms of probabilities.
5. $\langle 2^{\Sigma^*}, \cup, \cdot, \emptyset, \{\varepsilon\} \rangle$: the semiring of unweighted languages over an alphabet Σ under union $\cup$ and pairwise concatenation $\cdot$. Note that this has a subsemiring consisting of only the regular languages. (Similar semirings exist whose elements are weighted languages and relations, but we do not define them here.)

A semiring can have additional properties, and in this article we are interested in the following two:

1. commutativity: $a \otimes b = b \otimes a\,, \quad \forall a, b \in \mathbb{K}$
2. idempotency: $a \oplus a = a\,, \quad \forall a \in \mathbb{K}$

All examples above are commutative, except the last one, which is commutative only if $|\Sigma| = 1$. Examples 1, 4, and 5 are idempotent.

We will use the following notations for repeated collection and extension of a single value $k \in \mathbb{K}$:

$$ik = k \oplus k \oplus \cdots \oplus k \qquad (i \text{ times}) \tag{1}$$

$$k^i = k \otimes k \otimes \cdots \otimes k \qquad (i \text{ times}) \tag{2}$$

Note that ik does not in general mean $i \otimes k$. Usually the latter is not even defined, as the integer $i \in \mathbb{N}$ is usually not an element of the semiring.

2.2 Weighted n-ary Relations and Multi-Tape Weighted Finite-State Machines

A weighted n-ary relation is a function from $(\Sigma^*)^n$ to $\mathbb{K}$, for a given finite alphabet Σ and a given weight semiring $\mathcal{K} = \langle \mathbb{K}, \oplus, \otimes, \bar{0}, \bar{1} \rangle$. In other words, the relation assigns a weight to any n-tuple of strings. A weight of $\bar{0}$ can be interpreted as meaning that the tuple is not in the relation.

We are especially interested in *rational* (or *regular*) n-ary relations—that is, relations that can be encoded by n-tape weighted finite-state machines, which we now define.

We adopt a convention that variable names referring to n-tuples of strings include a superscript $^{(n)}$. Thus we write $s^{(n)}$ rather than $\vec{s}$ for a tuple of strings $\langle s_1, \ldots, s_n \rangle$. We also use this convention for the names of more complex objects that contain n-tuples of strings, such as n-tape automata and their transitions and paths.

An *n-tape weighted finite-state machine* (WFSM or n-WFSM),[1] $A^{(n)}$, is defined by a six-tuple

$$A^{(n)} = \langle \Sigma, Q, \mathcal{K}, E^{(n)}, \lambda, \varrho \rangle \tag{3}$$

with Σ being a finite alphabet, Q a finite set of states, $\mathcal{K} = \langle \mathbb{K}, \oplus, \otimes, \bar{0}, \bar{1} \rangle$ the semiring of weights, $E^{(n)} \subseteq (Q \times (\Sigma^*)^n \times \mathbb{K} \times Q)$ a finite set of weighted n-tape transitions, $\lambda : Q \to \mathbb{K}$ a function that assigns initial weights to states, and $\varrho : Q \to \mathbb{K}$ a function that assigns final weights to states.

[1] We follow some recent literature in using the term "machine" rather than "automaton." The acronym to refer to the general n-tape case is then FSM or n-FSM, which leaves the acronym FSA available to refer to the special case of a finite-state *acceptor* ($n = 1$). FST refers to the special case of a finite-state transducer ($n = 2$).

Any transition $e^{(n)} \in E^{(n)}$ has the form

$$e^{(n)} = \langle p, \ell^{(n)}, w, n \rangle \tag{4}$$

We refer to these four components as the transition's source state $p(e^{(n)}) \in Q$, its label $\ell(e^{(n)}) \in (\Sigma^*)^n$, its weight $w(e^{(n)}) \in \mathbb{K}$, and its target state $n(e^{(n)}) \in Q$.

A *path* $\gamma^{(n)}$ of length $\ell \geq 0$ is a sequence of transitions $e_1^{(n)} e_2^{(n)} \cdots e_\ell^{(n)}$ such that $n(e_i^{(n)}) = p(e_{i+1}^{(n)})$ for each $i \in [\![1, \ell - 1]\!]$. A path's label is defined to be the elementwise concatenation of the labels of its transitions:

$$\ell(\gamma^{(n)}) \stackrel{\text{def}}{=} \ell(e_1^{(n)}) \cdot \ell(e_2^{(n)}) \cdot \cdots \cdot \ell(e_\ell^{(n)}) \tag{5}$$

This is an n-tuple of strings having the form $s^{(n)} = \langle s_1, s_2, \ldots, s_n \rangle$. The path's weight is defined to be

$$w(\gamma^{(n)}) \stackrel{\text{def}}{=} \lambda(p(e_1^{(n)})) \otimes \left(\bigotimes_{j \in [\![1,\ell]\!]} w\left(e_j^{(n)}\right) \right) \otimes \varrho(n\left(e_\ell^{(n)}\right)) \tag{6}$$

The path is said to be *successful*, and to *accept* its label, if $w(\gamma^{(n)}) \neq \bar{0}$. We denote by $\Gamma_{A^{(n)}}$ the set of all successful paths of $A^{(n)}$, and by $\Gamma_{A^{(n)}}(s^{(n)})$ the set of successful paths (if any) that accept $s^{(n)}$:

$$\Gamma_{A^{(n)}}(s^{(n)}) \;=\; \{\; \gamma^{(n)} \in \Gamma_{A^{(n)}} \mid s^{(n)} = \ell(\gamma^{(n)}) \;\} \tag{7}$$

Now, the machine $A^{(n)}$ defines a weighted n-ary relation $\mathcal{R}(A^{(n)}) : (\Sigma^*)^n \to \mathbb{K}$ that assigns to each n-tuple, $s^{(n)}$, the total weight of all paths accepting it:

$$\mathcal{R}_{A^{(n)}}(s^{(n)}) \stackrel{\text{def}}{=} \bigoplus_{\gamma^{(n)} \in \Gamma_{A^{(n)}}(s^{(n)})} w(\gamma^{(n)}) \tag{8}$$

It is convenient to define the *support* of an arbitrary weighted relation $\mathcal{R}^{(n)}$, meaning the set of tuples to which the relation gives non-$\bar{0}$ weight:

$$\mathsf{support}(\mathcal{R}^{(n)}) \stackrel{\text{def}}{=} \{\; s^{(n)} \in (\Sigma^*)^n \mid \mathcal{R}^{(n)}(s^{(n)}) \neq \bar{0} \;\} \tag{9}$$

This support set can be regarded as an ordinary unweighted relation obtained from $\mathcal{R}^{(n)}$. A different perspective on unweighted relations is that they are weighted relations over the boolean semiring, i.e., functions from $(\Sigma^*)^n \to \{\text{FALSE}, \text{TRUE}\}$.

2.3 Infinite Sums

In defining $\mathcal{R}(A^{(n)})$, we glossed over one point for simplicity's sake. A sum over finitely many weights can be computed by repeated application of $\oplus$. But (8) may sometimes call for an infinite sum, whose meaning has not been defined. This case arises if $\mathcal{R}_{A^{(n)}}$ contains any cyclic paths with the label $\langle \epsilon, \epsilon, \ldots \epsilon \rangle$. Cyclic paths of this sort cannot simply be disallowed in a natural way, since they can be re-introduced by the closure and projection operations discussed below.

Briefly, the solution is to pre-compute the geometric sum $k^* = \bigoplus_{i=0}^{\infty} k^i \in \mathbb{K}$ for each $k \in \mathbb{K}$.[2] In practice, one simply defines a *closure* operator * that satisfies certain axioms, obtaining a so-called *closed semiring*.

This allows infinite sums over any *regular* set of paths, as required by (8) and by section 3's equations (11), (12), (13), and (21). One constructs a WFSM containing just those paths (e.g., $\Gamma_{A^{(n)}}(s^{(n)})$), and then sums their weights with an algorithm that generalizes the Kleene-Floyd-Warshall technique to closed semirings (Lehmann, 1977).

[2] Divergent sums can be represented by $k^* = \infty$, where $\infty \in \mathbb{K}$ is a distinguished value.

3 Operations

We now describe some central operations on n-ary weighted relations and their n-tape WFSMs, focusing on operations that affect the number of tapes. (See (Kempe, Guingne, and Nicart, 2004).) In particular, we introduce an "auto-intersection" operation that will simplify the discussion of multi-tape join.

Our notation is chosen throughout to highlight the connection to relational databases.

3.1 Simple Operations

The basic rational operations of union, concatenation, and closure can be used to construct any n-ary weighted rational relation.[3] Thus, the rational operations can be used to write *regular expressions* that specify particular relations. On the database perspective, such expressions are useful for specifying both actual databases (typically finite relations) and particular queries (typically infinite relations, i.e., the set of all tuples with a given property). (Section 3.3 will discuss how to intersect a database with a query.)

The union and concatenation of two weighted n-ary relations, $\mathcal{R}_1^{(n)}$ and $\mathcal{R}_2^{(n)}$, are the relations $\mathcal{R}_1^{(n)} \cup \mathcal{R}_2^{(n)}$ and $\mathcal{R}_1^{(n)} \cdot \mathcal{R}_2^{(n)}$ defined by

$$\left(\mathcal{R}_1^{(n)} \cup \mathcal{R}_2^{(n)}\right)(s^{(n)}) \stackrel{\text{def}}{=} \mathcal{R}_1^{(n)}(s^{(n)}) \oplus \mathcal{R}_2^{(n)}(s^{(n)}) \tag{10}$$

$$\left(\mathcal{R}_1^{(n)} \cdot \mathcal{R}_2^{(n)}\right)(s^{(n)}) \stackrel{\text{def}}{=} \bigoplus_{\substack{u^{(n)},v^{(n)}:\\ (\forall i \in [\![1,n]\!])\, s_i = u_i \cdot v_i}} \mathcal{R}_1^{(n)}(u^{(n)}) \otimes \mathcal{R}_2^{(n)}(v^{(n)}) \tag{11}$$

The closure of $\mathcal{R}^{(n)}$ is the relation

$$(\mathcal{R}^{(n)})^* \stackrel{\text{def}}{=} \bigcup_{\ell=0}^{\infty} \underbrace{\mathcal{R}^{(n)} \cdot \mathcal{R}^{(n)} \cdots \mathcal{R}^{(n)}}_{\ell \text{ times}}, \text{ implying that}$$

$$\left((\mathcal{R}^{(n)})^*\right)(\langle s_1, \ldots, s_n \rangle) = \bigoplus_{\ell=0}^{\infty} \bigoplus_{\substack{u_1^{(n)},\ldots u_\ell^{(n)}:\\ (\forall i \in [\![1,n]\!]) s_i = (u_1)_i \cdot (u_2)_i \cdots (u_\ell)_i}} \bigotimes_{j=1}^{\ell} \mathcal{R}^{(n)}(u_j^{(n)}) \tag{12}$$

These operations can be implemented by simple constructions on the corresponding nondeterministic n-tape WFSMs (Rosenberg, 1964). These n-tape constructions and their semiring-weighted versions are exactly the same as for acceptors ($n = 1$) and transducers ($n = 2$), as they are indifferent to the n-tuple transition labels.

3.2 Projection and Complementary Projection

Projection keeps certain columns of a database relation and discards the others. In the case of a rational relation implemented by a n-WFSM, it can be implemented by discarding the corresponding tapes of the n-WFSM, yielding an m-WFSM for $m < n$.

Projection may map several distinct n-tuples onto the same m-tuple. In this case, we will define the weight of the m-tuple by summing the several n-tuples' weights using $\oplus$. This resembles aggregation in databases, but note that only weights can be aggregated across n-tuples, not the (string) data in the n-tuples themselves.

[3] By combining the "atomic" weighted relations, namely, those whose support is a single tuple from the finite set $\{(s_1, s_2, \ldots s_n) : |s_1 s_2 \cdots s_n| \leq 1\}$.

For any $j_1, \ldots, j_m \in [\![1, n]\!]$, we formally define a *projection* operator $\pi_{\langle j_1,\ldots,j_m\rangle}$ that maps n-ary relations to m-ary relations:

$$\left(\pi_{\langle j_1,\ldots,j_m\rangle}(\mathcal{R}_1^{(n)})\right)(s^{(m)}) \stackrel{\text{def}}{=} \bigoplus_{\substack{u^{(n)}: \\ (\forall i \in [\![1,m]\!])\, s_i = u_{j_i}}} \mathcal{R}_1^{(n)}(u^{(n)}) \tag{13}$$

It retains only those component strings (i.e. tapes) of each tuple that are specified by the indices $j_1, \ldots j_m$, and places them in the specified order.

Notice that our definition allows projection indices to occur in any order, possibly with repeats. Thus the tapes of $s^{(n)}$ can be permuted or duplicated. For example, $\pi_{\langle 2,1\rangle}$ will invert a 2-ary relation.

As a convenience, we also define the *complementary projection* of a relation. For any $j_1, \ldots j_m \in [\![1, n]\!]$, we define an operator $\overline{\pi}_{\{j_1,\ldots j_m\}}$ that removes the tapes $j_1, \ldots j_m$ and preserves all other tapes in their original order. Without loss of generality we may assume that $j_1 < j_2 < \cdots < j_m$; then we can define $\overline{\pi}_{\{j_1,\ldots j_m\}}$ as equivalent to $\pi_{\langle 1,\ldots,j_1-1,j_1+1,\ldots j_m-1,j_m+1,\ldots n\rangle}$, which maps n-ary relations to $(n-m)$-ary relations.

3.3 Join and Generalized Composition

Applications: Our version of the join operation is quite powerful. It can be used to join two "databases" (typically finite relations), to conjoin two "queries" (typically infinite relations), or to select those database tuples that match a query, reweighting them if the query is weighted.

Another family of uses is inspired by natural language processing, where WFSTs ($n = 2$) are commonly used to construct noisy channel models (Knight and Graehl, 1998). Using $n > 2$ tapes allows us to generalize naturally to doing constraint programming or graphical modeling over string-valued variables. Given variables $V_1, \ldots V_n$ with unknown values in the *infinite* domain Σ^*, one can specify a (weighted) m-ary relation to express a (soft) constraint over some $m \leq n$ of the variables. All known constraint relations can be systematically joined together, along tapes that correspond to common variables. This yields a (weighted) n-ary relation that evaluates which n-tuples are appropriate as joint values of the n variables. If this n-ary relation specifies a probability distribution over n-tuples, one can intersect it with another n-ary relation describing incomplete data, in order to compute the probability of the data for purposes of parameter training or statistical inference.

As we will see, join is *too* powerful: rational relations are *not* closed under arbitrary joins. Section 4 will explore this point in detail. Nonetheless, we can mathematically define the possibly non-rational result of a join. The operation appears so useful that it would be helpful to have a partial or approximate algorithm.

Definition: The reader may already be familiar with the notion of natural join on databases. Our presentation differs from the standard database treatment in that our tapes are numbered, whereas the columns of a database are typically named. So our join operators, unlike a database join, must explicitly select tapes by number, and as a result are neither associative nor commutative.

A join of two relations is formed by finding "matching" pairs of tuples. For example, $\langle abc, def, \epsilon\rangle$ and $\langle def, ghi, \epsilon, jkl\rangle$ match on two of their tapes. We notate the matching of tapes in this case as $\{2{=}1, 3{=}3\}$. They combine to yield a tuple $\langle abc, def, \epsilon, ghi, jkl\rangle$, whose weight in the joined relation is the product (under $\otimes$) of the two original tuples' weights.

More precisely, for any distinct $i_1, \ldots i_r \in [\![1, n]\!]$ and any distinct $j_1, \ldots j_r \in [\![1, m]\!]$, we define a *join* operator $\bowtie_{\{i_1=j_1,\ldots,i_r=j_r\}}$. It combines an n-ary and an m-ary relation into an $(n+m-r)$-ary relation defined as follows:

$$\left(\mathcal{R}_1^{(n)} \bowtie_{\{i_1=j_1,\ldots,i_r=j_r\}} \mathcal{R}_2^{(m)}\right)(\langle u_1,\ldots,u_n,s_1,\ldots,s_{m-r}\rangle) \stackrel{\text{def}}{=} \mathcal{R}_1^{(n)}(u^{(n)}) \otimes \mathcal{R}_2^{(m)}(v^{(m)}) \tag{14}$$

where $v^{(m)}$ is the unique tuple such that $\overline{\pi}_{\{j_1,\ldots j_r\}}(v^{(m)}) = s^{(m-r)}$ and $(\forall \ell \in [\![1,r]\!])v_{j_\ell} = u_{i_\ell}$.

Relation to Cross Product: Taking $r = 0$ gives an important special case. The *cross product* operator $\times$, equivalent to $\bowtie_\emptyset$, combines an n-ary and an m-ary relation into an $(n+m)$-ary relation:

$$\mathcal{R}_1^{(n)} \times \mathcal{R}_2^{(m)} \stackrel{\text{def}}{=} \mathcal{R}_1^{(n)} \bowtie_\emptyset \mathcal{R}_2^{(m)} \tag{15}$$

with the result that

$$\left(\mathcal{R}_1^{(n)} \times \mathcal{R}_2^{(m)}\right)(\langle u_1,\ldots,u_n,v_1,\ldots,v_m\rangle) = \mathcal{R}_1^{(n)}(u^{(n)}) \otimes \mathcal{R}_2^{(m)}(v^{(m)}) \tag{16}$$

A WFSM for $\mathcal{R}_1^{(n)} \times \mathcal{R}_2^{(m)}$ can easily be constructed from WFSMs for $\mathcal{R}_1^{(n)}$ and $\mathcal{R}_2^{(m)}$, by concatenating them after appropriately "padding" their transition labels into $(n+m)$-tuples via extra epsilons. Thus, the cross product of weighted rational relations is always rational.

Relation to Intersection: Taking $n = r = m$ gives another important special case. The *intersection* of two n-ary relations is another n-ary relation:

$$\mathcal{R}_1^{(n)} \cap \mathcal{R}_2^{(n)} \stackrel{\text{def}}{=} \mathcal{R}_1^{(n)} \bowtie_{\{1=1,2=2,\ldots n=n\}} \mathcal{R}_2^{(n)} \tag{17}$$

with the result that

$$\left(\mathcal{R}_1^{(n)} \cap \mathcal{R}_2^{(n)}\right)(s^{(n)}) = \mathcal{R}_1^{(n)}(s^{(n)}) \otimes \mathcal{R}_2^{(n)}(s^{(n)}) \tag{18}$$

It is known that the intersection of transducers ($n = 2$) is not necessarily rational (Rabin and Scott, 1959): $\{\langle a^j b^*, c^j\rangle \mid j \in \mathbb{N}\} \cap \{\langle a^* b^j, c^j\rangle \mid j \in \mathbb{N}\} = \{\langle a^j b^j, c^j\rangle \mid j \in \mathbb{N}\}$. Nor, for that matter, is intersection of acceptors ($n = 1$) if they are weighted by a non-commutative semiring. Thus rational relations are not closed under the more general join operation, either.

Generalized Composition: For distinct $i_1,\ldots i_r \in [\![1,n]\!]$ and distinct $j_1,\ldots j_r \in [\![1,m]\!]$, it is convenient to define a *generalized composition* operator $\boxtimes_{\{i_1=j_1,\ldots,i_r=j_r\}}$. It carries out a join and then discards the joined tapes:

$$\mathcal{R}_1^{(n)} \boxtimes_{\{i_1=j_1,\ldots,i_r=j_r\}} \mathcal{R}_2^{(m)} \stackrel{\text{def}}{=} \overline{\pi}_{\{i_1,\ldots i_r\}}\left(\mathcal{R}_1^{(n)} \bowtie_{\{i_1=j_1,\ldots,i_r=j_r\}} \mathcal{R}_2^{(m)}\right) \tag{19}$$

Note that $\boxtimes$ can result in aggregation because it uses $\bar{\pi}$. For example, the special case of ordinary composition $\circ$ of transducers

$$\mathcal{R}_1^{(2)} \circ \mathcal{R}_2^{(2)} \stackrel{\text{def}}{=} \mathcal{R}_1^{(2)} \boxtimes_{\{2=1\}} \mathcal{R}_2^{(2)} = \overline{\pi}_{\{2\}}(\mathcal{R}_1^{(2)} \bowtie_{\{2=1\}} \mathcal{R}_2^{(2)}) \tag{20}$$

results in a summation over strings v on the discarded tape that was joined:

$$\left(\mathcal{R}_1^{(2)} \circ \mathcal{R}_2^{(2)}\right)(u,w) = \bigoplus_v \mathcal{R}_1^{(2)}(u,v) \otimes \mathcal{R}_2^{(2)}(v,w) \tag{21}$$

The generalized composition of rational relations is not necessarily rational.

Single-Tape Join: We speak about *single-tape join* if only one tape is used in each relation ($r = 1$). Two well-known special cases are the join $\bowtie_{\{1=1\}}$ used to intersect two acceptors in (17) (where $n = 1$), and the join $\bowtie_{\{2=1\}}$ used during classical composition of two transducers in (20).

There are other uses of single-tape join. A composition cascade of several transducers, $\mathcal{R}^{(2)} = \mathcal{R}_1^{(2)} \circ \mathcal{R}_2^{(2)} \circ \mathcal{R}_3^{(2)}$, could be replaced by a join cascade, $\mathcal{R}^{(4)} = \mathcal{R}_1^{(2)} \bowtie_{\{2=1\}} \left(\mathcal{R}_2^{(2)} \bowtie_{\{2=1\}} \mathcal{R}_3^{(2)}\right)$. The intermediate results are now preserved on tapes 2 and 3 for subsequent inspection or further transduction (Kempe, 2004). In this way, single-tape join is adequate to combine several transducers into any tree topology: $\mathcal{R}^{(4)} = \left(\mathcal{R}_1^{(2)} \bowtie_{\{2=1\}} \mathcal{R}_2^{(2)}\right) \bowtie_{\{2=1\}} \mathcal{R}_3^{(2)}$. One can use this technique to implement a tree-structured directed graphical model (sometimes called a dendroid distribution) by joining weighted transducers that represent the conditional probability distributions of the model.

Sometimes one wishes to join an n-ary relation with a cross product of m languages. This operation can be regarded as m single-tape joins. It can be used to train the parameters of the dendroid distribution described above, as explained for $n = m = 2$ by (Eisner, 2002). The generalization to more tapes is particularly useful for training a cascaded noisy channel model when intermediate results along the channel are partly observed.

The single-tape join of weighted multi-tape rational relations is rational as long as the weights fall in a commutative weight semiring. One can construct a WFSM for the resulting relation, using a standard "cross-product of states" construction.

The commutativity of the weights is crucial to this construction. (The constructed WFSM's paths interleave weights from paths in the two input WFSMs.) No such construction is possible if the weight semiring $\mathcal{K}$ is not commutative. For example, let k, k' be weights that do not commute. Let $\mathcal{R}^{(1)}$ be a rational language such that $\forall j \in \mathbb{N}, \mathcal{R}^{(1)}(a^j) = k^j \otimes k'$. Then (18) implies that $\forall j, \left(\mathcal{R}^{(1)} \cap \mathcal{R}^{(1)}\right)(a^j) = k^j \otimes k' \otimes k^j \otimes k'$; this single-tape join cannot in general be computed by any WFSM.

Mohri, Pereira, and Riley (1998), writing about WFST composition, noted another subtlety in extending the "cross product of states" construction to weighted machines. Their observation and solution apply generally to single-tape join of WFSMs (and would presumably be relevant to any partial algorithm for multi-tape join). A pair of successful paths in the input machines are considered to "match" if they both accept the same string s on the single tape being joined. A pair of matched input paths is supposed to yield exactly one path in the composed machine. However, if both input paths allow ϵ transitions on the join tape at the same position in s, then a naive implementation of the construction may produce $i > 1$ identically labeled and weighted paths, corresponding to different alignments of the input paths. This "path multiplicity problem" will incorrectly contribute i copies of the path weight to the sum in (8), affecting the result unless the weight semiring is idempotent. The solution is to revise the construction to allow only a canonical alignment of matched input paths.

3.4 Auto-Intersection

Our discussion of join will be simplified by reducing it to a simpler problem. For any distinct $i_1, j_1, \ldots i_r, j_r \in [\![1, n]\!]$, we define an *auto-intersection* operator $\sigma_{\{i_1=j_1, i_2=j_2, \ldots, i_r=j_r\}}$ that maps a relation $\mathcal{R}^{(n)}$ to a "subset" of that relation, preserving tuples $s^{(n)}$ whose elements are equal in pairs as specified, but removing all other tuples from the support of the relation.[4]

$$\left(\sigma_{\{i_1=j_1,\ldots,i_r=j_r\}}(\mathcal{R}^{(n)})\right)(\langle s_1, \ldots, s_n\rangle) \stackrel{\text{def}}{=} \begin{cases} \mathcal{R}^{(n)}(\langle s_1, \ldots, s_n\rangle) & \text{if } (\forall \ell \in [\![1, r]\!]) s_{i_\ell} = s_{j_\ell} \\ \bar{0} & \text{otherwise} \end{cases} \quad (22)$$

Auto-intersection does not necessarily preserve the rationality of $\mathcal{R}^{(n)}$, as we will discuss in Section 4.

[4]The requirement that the $2r$ indices be distinct mirrors the similar requirement on join and is needed in (26). But it can be evaded by duplicating tapes: an illegal auto-intersection such as $\sigma_{\{1=2,2=3\}}(\mathcal{R})$ can be computed as $\overline{\pi}_{\{3\}}(\sigma_{\{1=2,3=4\}}(\pi_{\langle 1,2,2,3\rangle}(\mathcal{R})))$.

Note that auto-intersecting a relation is different from joining the relation with its own projections. For example, $\sigma_{\{1=2\}}(\mathcal{R}^{(2)})$ is supported by tuples of the form $\langle w, w\rangle \in \mathcal{R}^{(2)}$. By contrast, $\mathcal{R}^{(2)} \bowtie_{\{1=1\}} \left(\pi_{\langle 2\rangle}(\mathcal{R}^{(2)})\right)$ is supported by tuples $\langle w, x\rangle \in \mathcal{R}^{(2)}$ such that w can also appear on tape 2 of $\mathcal{R}^{(2)}$ (but not necessarily paired with a copy of w on tape 1).

An example of auto-intersection is shown in Figure 1. It encodes the relation

$$\mathcal{R}_1^{(3)} \;=\; \langle a, x, \varepsilon\rangle\ \langle b, y, a\rangle^*\ \langle \varepsilon, z, b\rangle \;=\; \{\ \langle ab^j, xy^jz, a^jb\rangle \mid j \in \mathbb{N}\ \} \tag{23}$$

$$\sigma_{\{1=3\}}(\mathcal{R}_1^{(3)}) \;=\; \{\ \langle ab^1, xy^1z, a^1b\rangle\ \} \tag{24}$$

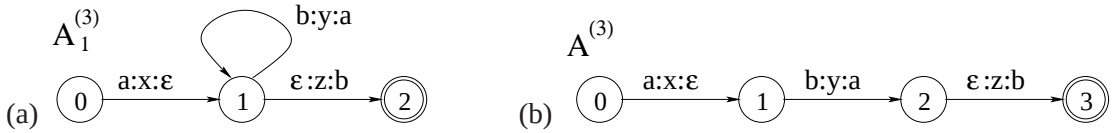


Figure 1: (a) A WFSM $A_1^{(3)}$ and (b) its auto-intersection $A^{(3)} = \sigma_{\{1=3\}}(A_1^{(3)})$. (Weights omitted)

It is possible to reduce join to auto-intersection using only rational operations (namely cross product and complementary projection). An arbitrary join can be implemented as

$$\mathcal{R}_1^{(n)} \bowtie_{\{i_1=j_1,\ldots,i_r=j_r\}} \mathcal{R}_2^{(m)} \;=\; \overline{\pi}_{\{n+j_1,\ldots,n+j_r\}}\left(\ \sigma_{\{i_1=n+j_1,\ldots,i_r=n+j_r\}}(\ \mathcal{R}_1^{(n)} \times \mathcal{R}_2^{(m)}\ )\ \right) \tag{25}$$

Conversely, it is possible to reduce any auto-intersection to a single join with a rational relation:

$$\sigma_{\{i_1=j_1,\ldots,i_r=j_r\}}(\mathcal{R}^{(n)}) \;=\; \mathcal{R}^{(n)} \bowtie_{\{i_1=1,j_1=2,\ldots,i_r=2r-1,j_r=2r\}} \left(\underbrace{(\pi_{\langle 1,1\rangle}(\Sigma^*) \times \cdots \times \pi_{\langle 1,1\rangle}(\Sigma^*)}_{r \text{ times}}\right) \tag{26}$$

Thus, for any class of "difficult" join instances whose results are non-rational or have undecidable emptiness (see section 4.4), there is a corresponding class of difficult auto-intersection instances, and vice-versa. Conversely, a partial solution to one problem would yield a partial solution to the other. In future work we hope to identify such a partial algorithm for auto-intersection.

The rest of this paper is therefore devoted to remarks on the auto-intersection problem only. Working in terms of auto-intersection rather than join will simplify our discussion. First, only one machine is involved. Second, in considering partial algorithms for auto-intersection, we do not have to worry about the order in which non-commutative weights from two joined machines are multiplied together, or the path multiplicity problem. Those issues have already been handled in the cross-product step of the join construction (25), and are not of further concern to the auto-intersection step.

For simplicity, we will focus on auto-intersections $\sigma_{\{i=j\}}$ that involve only a single pair of tapes. That is enough to expose the core difficulties. Indeed, the general case of auto-intersection can be defined in terms of this simple case:

$$\sigma_{\{i_1=j_1,\ldots,i_r=j_r\}}(\ \mathcal{R}^{(n)}\ ) \;\stackrel{\text{def}}{=}\; \sigma_{\{i_r=j_r\}}(\ \cdots \sigma_{\{i_1=j_1\}}(\ \mathcal{R}^{(n)}\ ) \cdots\ ) \tag{27}$$

Nonetheless, we caution that the general case might benefit from a more direct treatment. It may be wise to compute $\sigma_{\{i_1=j_1,\ldots,i_r=j_r\}}$ "all at once" rather than one tape pair at a time. The reason is that even when $\sigma_{\{i_1=j_1,\ldots,i_r=j_r\}}$ is rational, a finite-state strategy for computing it via (27) could "fail" by encountering non-rational intermediate results. For example, consider applying $\sigma_{\{2=3,4=5\}}$ to the rational 5-ary relation $\{\langle a^ib^j, c^i, c^j, x, y\rangle \mid i, j \in \mathbb{N}\}$. The final result is rational (the empty relation), but the intermediate result after applying just $\sigma_{\{2=3\}}$ would be the non-rational relation $\{\langle a^ib^i, c^i, c^i, x, y\rangle \mid i \in \mathbb{N}\}$.

4 Some Difficult Examples for Auto-Intersection

Some instances of auto-intersection are "easy." In particular, consider a finite relation (one with finite support, representable by an acyclic WFSM). Its auto-intersection is computable and is itself finite, since it just selects some tuples of the original relation. (Thus, by (25), $\mathcal{R}_1 \bowtie \mathcal{R}_2$ is finite if $\mathcal{R}_1$ or $\mathcal{R}_2$ is.) On such "easy" examples, the job of a good auto-intersection algorithm is merely to keep the resulting FSM small by preserving the sharing of substrings in the original FSM.

In this section, we will discuss some "difficult" classes of auto-intersection problems, where the result is non-rational or has undecidable properties. Each such class has a matching class of join problems, as discussed in section 3.4.

These difficulties imply that there is no general finite-state join algorithm. Nor is there an algorithm that produces the join whenever it is rational and returns an error code otherwise.

At the same time, the examples in this section may be instructive if one wishes to design a more limited join or auto-intersection algorithm that can succeed (exactly or approximately) on some practical cases. We leave such a task to future work.

4.1 Equal-Exponent Problem

Consider the unweighted binary relation $\mathcal{R}^{(2)} = \{\langle a^i b^j, a^j b^k \rangle \mid i,j,k \in \mathbb{N}\}$, interpreted as a weighted relation over the boolean semiring. The relation is rational because it can be encoded by a 2-FSM (Figure 2a). Its auto-intersection $\sigma_{\{1=2\}}(\mathcal{R}^{(2)}) = \{\langle a^i b^i, a^i b^i \rangle \mid i \in \mathbb{N}\}$ is, however, non-rational. Notice that the auto-intersection would in effect need to select just those paths in Figure 2a where all three cycles are traversed the same number of times.

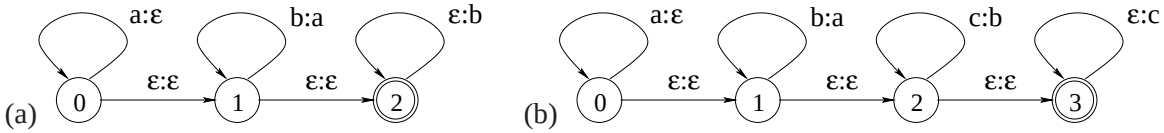


Figure 2: Two FSMs whose auto-intersection leads to equal-exponent problems

We can extend this example to any number of equal exponents. Consider for example the binary relation $\mathcal{R}^{(2)} = \{\langle a^i b^j c^k, a^j b^k c^\ell \rangle \mid i,j,k,\ell \in \mathbb{N}\}$, which is rational (Figure 2b) but has a non-rational auto-intersection $\sigma_{\{1=2\}}(\mathcal{R}^{(2)}) = \{\langle a^i b^i c^i, a^i b^i c^i \rangle \mid i \in \mathbb{N}\}$.

We say that such examples suffer from the *equal-exponent problem*. The equal-exponent problem may also appear on tapes other than the ones being intersected. The unweighted 3-ary relation $\{\langle a^i a, a a^j, x^i y z^j \rangle \mid i,j \in \mathbb{N}\}$ is rational (Figure 3a); but its auto-intersection under $\sigma_{\{1=2\}}$ is equal to $\{\langle a^i a, a a^i, x^i y z^i \rangle \mid i \in \mathbb{N}\}$, which is not rational because its projection onto tape 3 is not a regular language.

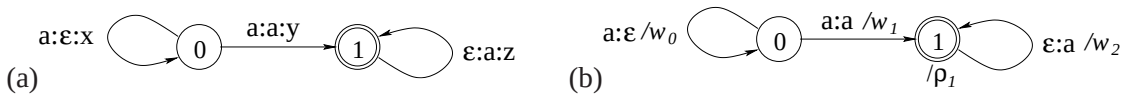


Figure 3: FSMs whose auto-intersection on tapes 1,2 requires equal exponents on tape 3 or in weights

Finally, the equal-exponent problem may appear in the weights assigned by the relation, if the weight semiring is not commutative. Figure 3b is a variant of Figure 3a that replaces the third tape with weights.

Its auto-intersection under $\sigma_{\{1=2\}}$ is the weighted relation $\mathcal{R}$ defined by

$$\mathcal{R}(\langle a^i a, a a^i \rangle) = w_0^i \otimes w_1 \otimes w_2^i \otimes \varrho_1 \quad (28)$$

$$\mathcal{R}(s^{(2)}) = \bar{0} \text{ otherwise} \quad (29)$$

This relation has rational support, but is not in general a rational relation. It does become rational if the weight semiring is commutative, in which case $w_0^i \otimes w_1 \otimes w_2^i \otimes \varrho_1$ can be computed as $(w_0 \otimes w_2)^i \otimes w_1 \otimes \varrho_1$. Notice that if the weights are rational languages over an alphabet Σ (see Section 2.1), so that they effectively act like a third tape, then they are guaranteed to commute only if $|\Sigma|$=1.

4.2 Shuffle Problem

The *shuffle product* of two strings $u \sqcup\!\sqcup v$ is defined, e.g., in (Sakarovitch, 2003) as:

$$u \sqcup\!\sqcup v \stackrel{\text{def}}{=} \{\, u_1 v_1 \dots u_j v_j \mid u = u_1 \dots u_j \,,\; v = v_1 \dots v_j \,,\; (\forall i \in [\![1, j]\!]) u_i, v_i \in \Sigma^* \,\} \quad (30)$$

This set contains all possible "interleavings" of the symbols from u and v. The symbols of u keep their respective order, as do the symbols of v, but any order is allowed between a symbol from u and a symbol from v. For example:

$$abc \sqcup\!\sqcup xy = \{abcxy, abxcy, abxyc, axbcy, axbyc, axybc, xabcy, xabyc, xaybc, xyabc\} \quad (31)$$

$$aa \sqcup\!\sqcup xx = \{aaxx, axax, axxa, xaax, xaxa, xxaa\} \quad (32)$$

$$aaa \sqcup\!\sqcup aaa = \{aaaaaa\} \quad (33)$$

The size of the set $u \sqcup\!\sqcup v$ grows exponentially in the lengths of u and v.

Consider the unweighted relation $\mathcal{R}^{(3)} = \{\langle a^i, a^j, x^i \sqcup\!\sqcup y^j \rangle \mid i, j \in \mathbb{N}\}$, interpreted as a weighted relation over the boolean semiring. It is rational because it can be encoded by a 3-FSM (Figure 4a). Its auto-intersection $\sigma_{\{1=2\}}(\mathcal{R}^{(3)}) = \{\langle a^i, a^i, x^i \sqcup\!\sqcup y^i \rangle \mid i \in \mathbb{N}\}$ is, however, non-rational, as its projection onto tape 3 is the non-rational language of strings having equal numbers of x's and y's.

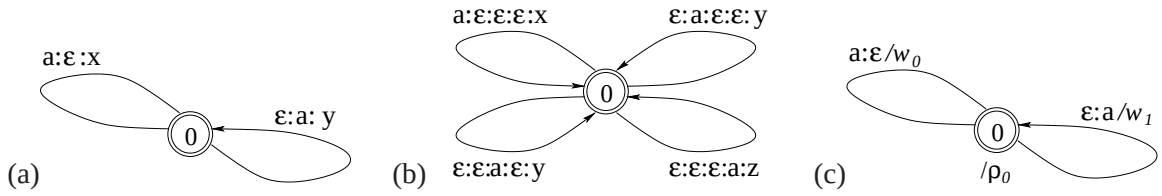


Figure 4: Three (W)FSMs whose auto-intersection leads to shuffle problems

Using additional tapes lets us extend this example to any number of equal exponents. For example, the relation $\mathcal{R}^{(5)} = \{\langle a^i, a^j, a^j, a^k, x^i \sqcup\!\sqcup y^j \sqcup\!\sqcup z^k \rangle \mid i, j, k \in \mathbb{N}\}$ is rational (Figure 2b) but has a non-rational auto-intersection $\sigma_{\{1=2,3=4\}}(\mathcal{R}^{(5)}) = \{\langle a^i, a^i, a^i, a^i, x^i \sqcup\!\sqcup y^i \sqcup\!\sqcup z^i \rangle \mid i \in \mathbb{N}\}$.

This *shuffle problem* can be regarded as the source of other failures of rationality. If $\mathcal{R}^{(1)}$ is any rational language, then the single-tape join $\{\langle a^i, a^j, (x^i \sqcup\!\sqcup y^j) \rangle \mid i, j \in \mathbb{N}\} \bowtie_{\{3=1\}} \mathcal{R}^{(1)}$ is also rational. Auto-intersecting it using the $\sigma_{\{1=2\}}$ operator yields a relation whose tape 3 recognizes a "restricted shuffle," namely, the potentially non-rational language $\{x^i \sqcup\!\sqcup y^i \mid i \in \mathbb{N}\} \cap \mathcal{R}^{(1)}$. For example, taking $\mathcal{R}^{(1)}$ to be the language $x^* y^*$ creates the equal-exponent language $\{x^i y^i \mid i \in \mathbb{N}\}$ of section 4.1.

Beyond simply restricting the shuffle language, one can also transduce it to obtain further examples. Consider the rational 3-relation $\{\langle a^i, a^j, (x^i \sqcup\!\sqcup y^j) \rangle \mid i, j \in \mathbb{N}\} \boxtimes_{\{3=1\}} \mathcal{R}^{(2)}$, where $\mathcal{R}^{(2)}$ is any rational

2-ary relation. Applying the $\sigma_{\{1=2\}}$ operator yields a relation whose tape 3 recognizes the transduction of $\{x^i \sqcup\!\sqcup y^i \mid i \in \mathbb{N}\}$ by $\mathcal{R}^{(2)}$. The transduction can replace x^i and y^i by arbitrary languages while restricting their shuffling.

The shuffle problem may also appear in the weights assigned by the relation, if the weight semiring is not both commutative and idempotent. Figure 4c is a variant of Figure 4a that replaces the third tape with weights.[5] Applying the $\sigma_{\{1=2\}}$ operator yields a relation $\mathcal{R}$ such that $\forall i \in \mathbb{N}$, $\mathcal{R}(a^i) = (w_0^i \sqcup\!\sqcup w_1^i) \otimes \varrho_0$, where the informal notation $w_0^i \sqcup\!\sqcup w_1^i$ denotes the "shuffle sum of two products of weights." For example, if $k, l, p, q \in \mathbb{K}$, we would write

$$\begin{aligned}
(k \otimes l) \sqcup\!\sqcup (p \otimes q) &= (k \otimes l \otimes p \otimes q) \oplus (k \otimes p \otimes l \otimes q) \oplus (k \otimes p \otimes q \otimes l) \oplus \\
&\quad (p \otimes k \otimes l \otimes q) \oplus (p \otimes k \otimes q \otimes l) \oplus (p \otimes q \otimes k \otimes l) && (34)\\
k^2 \sqcup\!\sqcup p^2 &= (k \otimes k \otimes p \otimes p) \oplus (k \otimes p \otimes k \otimes p) \oplus (k \otimes p \otimes p \otimes k) \oplus \\
&\quad (p \otimes k \otimes k \otimes p) \oplus (p \otimes k \otimes p \otimes k) \oplus (p \otimes p \otimes k \otimes k) && (35)\\
k^2 \sqcup\!\sqcup k^2 &= 6\,(k \otimes k \otimes k \otimes k) \;=\; 6\,(\,k^4\,) && (36)
\end{aligned}$$

In general, the weighted relation $\mathcal{R}$ in our example is non-rational. However, it is rational if the semiring is both commutative and idempotent. In that case, $w_0^i \sqcup\!\sqcup w_1^i = j_i(w_0 \otimes w_1)^i = (w_0 \otimes w_1)^i$, where $j_i \in \mathbb{N}$ is the number of summands in the shuffle sum and is irrelevant thanks to idempotency.

4.3 Presentation Problems

Our next example illustrates how a partial auto-intersection algorithm might be affected by the presentation of its input.

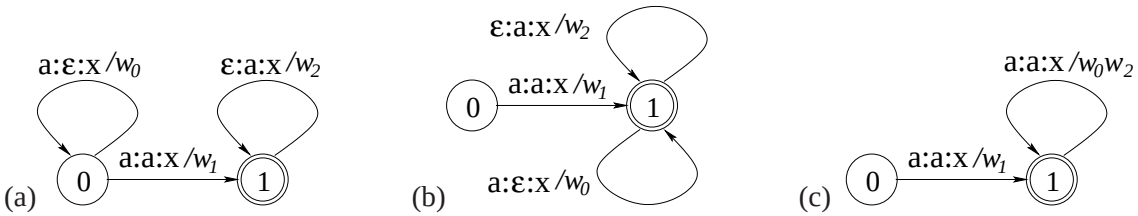


Figure 5: (a), (b) Different presentations of the same relation $\mathcal{R}^{(3)}$; (c) the auto-intersection $\sigma_{\{1,2\}}(\mathcal{R}^{(3)})$

Provided that the weight semiring is commutative, the WFSMs in Figures 4.3a and 4.3b describe the same relation, which for each $i \in \mathbb{N}$ maps $\langle a^{i+1}, a^{i+1}, x^{2i+1} \rangle$ to $w_0^i \otimes w_1 \otimes w_2^i$. A naive algorithm modeled on WFST determinization would fail to terminate on either machine, constructing a successful path of length $2i+1$ for each $i \in \mathbb{N}$. For example, on Figure 4.3a, it would allow unrolling the first cycle i times and then transitioning to the second cycle to allow the second tape to "catch up" with the first.

A partial algorithm for auto-intersection might attempt to detect and handle some such cases, allowing it to compute the correct auto-intersection (Figure 4.3c). It seems potentially easier to detect the Figure 4.3b case than the Figure 4.3a case.

[5] Again, this example can be derived by transducing the original shuffle example of Figure 4a. If all transitions in that example are given weight $\bar{1}$ in the semiring of interest, then its generalized composition $\boxtimes_{\{3=1\}}$ with a simple weighted machine will produce Figure 4c by replacing all instances of x with w_0, etc.

4.4 Post's Correspondence Problem

Post's Correspondence Problem or PCP (Post, 1946) is a classical undecidable problem that is sometimes used to prove the undecidability of other problems. Mark-Jan Nederhof (personal communication) pointed out its relevance to auto-intersection.

Definition: Given an alphabet Σ, an instance of PCP is a list of pairs of strings in Σ^*: $\langle u_1, v_1\rangle, \ldots \langle u_p, v_p\rangle$. A solution is a string s such that $s = u_{i_1}u_{i_2}\ldots u_{i_r} = v_{i_1}v_{i_2}\ldots v_{i_r}$ for some non-empty index sequence $i_1, i_2, \ldots i_r \in [\![1, p]\!]$. This sequence may contain duplicates.

Taking an example from (Zhao, 2002), the instance $\langle \texttt{abb}, \texttt{a}\rangle, \langle \texttt{b}, \texttt{abb}\rangle, \langle \texttt{a}, \texttt{bb}\rangle$ has among its solutions the string $abbaabbabbabb = u_1u_3u_1u_1u_3u_2u_2 = v_1v_3v_1v_1v_3v_2v_2$, obtained from the index sequence 1311322. For the sake of clarity, we show here both the instance and the solution in tabular form:

i	1	2	3
u_i	abb	b	a
v_i	a	abb	bb

i	1	3	1	1	3	2	2
u_i	abb	a	abb	abb	a	b	b
v_i	a	bb	a	a	bb	abb	abb

The language of solutions to a given instance is context-sensitive. That is, it is possible for a linear bounded automaton to determine whether a given string s is a solution, simply by considering all index sequences of length $\leq 2|s|$.[6]

What is not decidable, in general, is whether this context-sensitive language of solutions is non-empty. To put this another way, the set of PCP instances with at least one solution is not recursive (although it is recursively enumerable).

An instance of PCP can be represented as a 2-tape automaton, $A^{(2)}$, with a unique state, that is both initial and final, and p transitions labeled with pairs of strings $u_i : v_i$, as illustrated in Figure 6a. The set of all solutions to this instance equals $\pi_{\langle 1\rangle}(\sigma_{\{1=2\}}(\mathcal{R}(A^{(2)})))$. If one wishes instead to obtain the language of index sequences of each solution, one can represent the instance as a 3-tape automaton $A^{(3)}$ with an additional tape of indices $i \in [\![1, p]\!]$, as illustrated in Figure 6b, and construct $\pi_{\langle 1\rangle}(\sigma_{\{2=3\}}(\mathcal{R}(A^{(3)})))$.

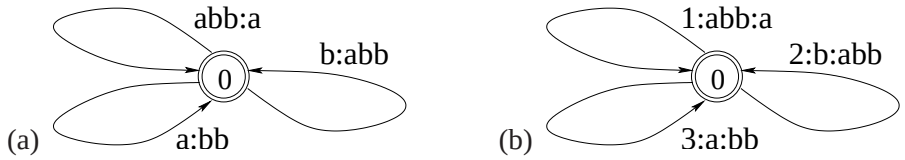


Figure 6: An instance of a PCP (a) without and (b) with an additional tape of indices

This reduction from PCP to auto-intersection demonstrates that it is undecidable whether the result of an unweighted 2-tape auto-intersection is empty.

Furthermore, this implies that there can be no partial auto-intersection algorithm that is "complete" in that it always returns a correct FSM if the auto-intersection is rational, and always terminates with an error code otherwise. If such an algorithm did exist, one could use it as follows to determine the emptiness of an unweighted auto-intersection (and hence to determine the existence of a solution to a PCP instance, which is impossible in general). If the algorithm returned an FSM, we would test it for emptiness by determining whether there was at least one path from an initial to a final state. If the algorithm returned an error code, we would know that the result was non-rational and hence could not be empty.

Despite this gloomy result, some recent work (Zhao, 2002) has explored heuristic tests that can identify some PCP instances as empty, as well as heuristic search methods that try to find a single solution

[6] We may assume without loss of generality that $\langle \varepsilon, \varepsilon\rangle$ is not among the strings in the instance. Then if $s = u_{i_1}u_{i_2}\ldots u_{i_r} = v_{i_1}v_{i_2}\ldots v_{i_r}$ is a solution, we have $r \leq |u_{i_1}u_{i_2}\ldots u_{i_r}v_{i_1}v_{i_2}\ldots v_{i_r}| = |ss| = 2|s|$.

to a PCP quickly (although not the full language of solutions). These methods might provide a starting point for constructing a useful partial algorithm for auto-intersection.

5 Conclusion

We have provided definitions and notation for the central operations on weighted n-ary relations and the finite-state machines that describe the rational cases. Our notation is informed by regarding these objects as weighted databases. This perspective is pedagogically useful and motivates potential applications.

We focused primarily on the important join operation $\bowtie$, and the related operations of generalized composition $\boxtimes$ and auto-intersection $\sigma_{\{1=2\}}$. In some cases, these operators preserve rationality. In general, they do not, and we showed that the resulting relations, while individually decidable (at the level of individual tuples), can have undecidable emptiness as a class.

Our question for future research is whether there exists a partial or approximate algorithm for auto-intersection that can handle some practical cases of infinite relations. This would imply the existence of a similar algorithm for join.

There is some precedent for such investigations. Regarding partial algorithms, we already noted the work of (Zhao, 2002) on partial solutions to the generally undecidable Post's Correspondence Problem, which reduces to our problem. In the speech and language processing community, researchers manage to make practical use of a WFSM determinization algorithm that is not guaranteed to terminate when no answer exists (Mohri, 1997).[7] As for approximations, context-free languages are not in general rational, but they can be usefully approximated by FSMs that accept a close superset or subset (Nederhof, 2000). Approximation by pruning is an option for FSMs weighted by probabilities. Where a naive auto-intersection algorithm would run forever, in an attempt to generate an infinite-state machine, it might be possible to obtain a reasonable finite-state machine by pruning away work on low-probability paths.

Acknowledgments

We wish to thank Mark-Jan Nederhof for discussion of our work at an earlier stage. It was he who saw the relationship between auto-intersection and Post's correspondence problem.

References

Bangalore, Srinivas and Michael Johnston. 2000. Finite-state multimodal parsing and understanding. In *Proc. of the 17th COLING*, pages 369–375, Saarbrücken, Germany, August.

Eilenberg, Samuel. 1974. *Automata, Languages, and Machines*, volume A. Academic Press, San Diego.

Eisner, Jason. 2002. Parameter estimation for probabilistic finite-state transducers. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, Philadelphia, July.

Elgot, Calvin C. and Jorge E. Mezei. 1965. On relations defined by generalized finite automata. *IBM Journal of Research and Development*, 9(1):47–68.

Kaplan, Ronald M. and Martin Kay. 1994. Regular models of phonological rule systems. *Computational Linguistics*, 20(3):331–378.

[7] Even though Mohri also exhibits a terminating algorithm for that problem.

Kay, Martin. 1987. Nonconcatenative finite-state morphology. In *Proc. 3rd Int. Conf. EACL*, pages 2–10, Copenhagen, Denmark.

Kempe, André. 2004. NLP applications based on weighted multi-tape automata. In *Proc. 11th Conf. TALN*, pages 253–258, Fes, Morocco, April.

Kempe, André, Franck Guingne, and Florent Nicart. 2004. Algorithms for weighted multi-tape automata. Research report 2004/031, Xerox Research Centre Europe, Meylan, France. (available from www.xrce.xerox.com and www.arXiv.org/abs/cs.CL/0406003).

Kiraz, George Anton. 2000. Multitiered nonlinear morphology using multitape finite automata: a case study on Syriac and Arabic. *Computational Lingistics*, 26(1):77–105, March.

Knight, Kevin and Jonathan Graehl. 1998. Machine transliteration. *Computational Linguistics*, 24(4).

Kuich, Werner and Arto Salomaa. 1986. *Semirings, Automata, Languages*. Number 5 in EATCS Monographs on Theoretical Computer Science. Springer Verlag, Berlin, Germany.

Lehmann, Daniel J. 1977. Algebraic structures for transitive closure. *Theoretical Computer Science*, 4(1):59–76.

Livescu, Karen, James Glass, and Jeff Bilmes. 2003. Hidden feature models for speech recognition using dynamic Bayesian networks. In *8th European Conference on Speech Communication and Technology (Eurospeech)*.

Mohri, Mehryar. 1997. Finite-state transducers in language and speech processing. *Computational Linguistics*, 23(2):269–312.

Mohri, Mehryar, Fernando C. N. Pereira, and Michael Riley. 1998. A rational design for a weighted finite-state transducer library. *Lecture Notes in Computer Science*, 1436:144–158.

Nederhof, Mark-Jan. 2000. Practical experiments with regular approximation of context-free languages. *Computational Linguistics*, 26(1).

Post, Emil. 1946. A variant of a recursively unsolvable problem. *Bulletin of the American Mathematical Society*, 52:264–268.

Rabin, Michael O. and Dana Scott. 1959. Finite automata and their decision problems. *IBM Journal of Research and Development*, 3(2):114–125.

Rosenberg, Arnold L. 1964. On n-tape finite state acceptors. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 76–81.

Sakarovitch, Jacques. 2003. *Éléments de theorie des automates*. Éditions Vuibert, Paris, France.

Zhao, Ling. 2002. Tackling Post's Correspondence Problem. In Jonathan Schaeffer, Martin Müller, and Yngvi Björnsson, editors, *Proceedings of the Third International Conference on Computers and Games, CG 2002*, volume 2883 of *Lecture Notes in Computer Science*, pages 326–344, Edmonton, Canada, July 25-27. Springer-Verlag. Revised edition (January 1, 2004).

Practical Experiments with NFA Simulation

Jan Holub* and Petr Špiller

Department of Computer Science and Engineering, Czech Technical University,
Karlovo nám. 13, CZ-121 35, Prague 2, Czech Republic
WWW: http://cs.felk.cvut.cz/~holub

Abstract. We present results of practical experiments with simulation of nondeterministic finite automata (NFA) in approximate string matching. NFA cannot be directly used because of its nondeterminism. Simulation of NFA run together with determinisation are the ways of NFA usage. We present practical experiments with two simulation methods called bit parallelism and dynamic programming. Furthermore, we introduce the resolution system which is capable of deciding which method is the most beneficial (in the meaning of the time and space requirements) for a given approximate string matching task. The decision is made according to the experimental result obtained during the project development.

1 Introduction

In Computer Science there is a class of tasks solvable by finite automata. For a task from this class we can construct deterministic finite automaton (DFA). Then we run the DFA and get the solution for the task. However, in practice nondeterministic finite automaton (NFA) for the given task is usually constructed much easier than DFA. That is why NFAs are in use. We cannot run NFA directly because of its nondeterminism. Once we have NFA we have two options what to do. We can either transform NFA to the corresponding DFA (using the standard subset construction algorithm [12, 15]) or simulate a run of the NFA.

The transformation NFA to DFA can lead up to exponential blow up of states (up to $2^{|Q_{NFA}|}$ DFA states where $|Q_{NFA}|$ is number of states of NFA). Although the blow up is not always as bad it may be a source of problems. The time complexity of preprocessing (the transformation NFA to DFA) is dependent on the number of states of the resulting DFA. However, once we have DFA we can run the DFA very quickly in time $\mathcal{O}(n)$ where n is a given input text.

The simulation of NFA runs always slower but has no such huge requirements for the memory and has very simple preprocessing. Basic simulation method [9] runs in time $\mathcal{O}(n|Q_{NFA}|^2)$ and space $\mathcal{O}(|\Sigma||Q_{NFA}|^2)$. Implementation using bit vectors [9] then runs in time $\mathcal{O}(n|Q_{NFA}|\lceil\frac{|Q_{NFA}|}{w}\rceil)$ and space $\mathcal{O}(|\Sigma||Q|\lceil\frac{|Q|}{w}\rceil)$, where w is a length of used computer word in bits.

Since the NFA simulation can be considered as a just-in-time transformation NFA to DFA where only the deterministic state needed in the next step is constructed from the current deterministic state, we can also consider third approach which is a combination of the previous two.

Theoretical works on the topic were already introduced (see [9–11]). This paper presents experimental results of NFA simulation in the area of the exact and approximate string matching. At the end of the paper we present our resolution system which resulted from our experiments. The resolution system decides which method is the most suitable for a given string matching task. The considered methods are the transformation to DFA with consequent DFA run or one of the NFA simulation methods. Input data for the decision are length of string, length of input text, and type and level of edit distance.

The algorithms that use the bit parallelism were developed without the knowledge that they simulate NFA solving the given problem. At first an algorithm using the bit parallelism

* This research has been partially supported by MŠMT research program No MSM 212300014.

was used for the exact string matching (Shift-And in [5]), the multiple exact string matching (Shift-And in [18]), the approximate string matching using the Hamming distance (Shift-Add in [2]), the approximate string matching using the Levenshtein distance (Shift-Or in [2] and Shift-And in [21]) and for the generalized pattern matching (Shift-Or in [1]), where the pattern consists not only from symbols but also from sets of symbols. Later [8, 9] it was discovered they simulate the corresponding Σ-version of NFA.

The original dynamic programming algorithm [17, 19] was designed for the approximate string matching using the Levenshtein distance also without the knowledge it simulates the NFA. It was designed in order to directly compute the edit distance. Moreover, it was used for construction of DFA with $\min(3^m, (k+1)!(k+2)^{m-k})$ states. Later [8, 9] it was shown that it simulates the corresponding Σ-version of NFA. The original algorithm was improved in [6]. However, this improvement cannot be applied for other edit distances nor for $\bar{p}$-version of NFA.

At the beginning we present DFA and simulation methods on the exact string matching. Then comparison of these methods for individual edit distances follows. At the end we shortly describe our resolution system.

2 Preliminaries

Let Σ be a nonempty input alphabet, Σ^* be the set of all strings over Σ, ε be the *empty string*, and $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$. If $w \in \Sigma^*$, then $|w|$ denotes the *length* of w ($|\varepsilon| = 0$). If $a \in \Sigma$, then $\bar{a} = \Sigma \setminus \{a\}$ denotes a *complement* of a over Σ. If $w = xyz$, $x, y, z \in \Sigma^*$, then x, y, z are *factors* (substrings) of w, moreover, x is a *prefix* of w and z is a *suffix* of w.

Deterministic finite automaton (DFA) is a quintuple $(Q, \Sigma, \delta, q_0, F)$, where Q is a set of states, Σ is a set of input symbols, δ is a mapping (transition function) $Q \times \Sigma \mapsto Q$, $q_0 \in Q$ is an initial state, and $F \subseteq Q$ is a set of final states. We extend δ to a function $\hat{\delta}$ mapping $Q \times \Sigma^+ \mapsto Q$.

Nondeterministic finite automaton (NFA) is a quintuple $(Q, \Sigma, \delta, q_0, F)$, where Q, Σ, q_0, F are the same like in DFA and δ is a mapping $Q \times (\Sigma \cup \{\varepsilon\}) \mapsto 2^{|Q|}$. We also extend δ to $\hat{\delta}$ mapping $Q \times \Sigma^* \mapsto 2^{|Q|}$. DFA (resp. NFA) accepts a string $w \in \Sigma^*$ if and only if $\hat{\delta}(q_0, w) \in F$ (resp. $\hat{\delta}(q_0, w) \cap F \neq \emptyset$).

An *active state* of NFA, when the last symbol of a prefix w of an input string is processed, denotes each state q, $q \in \hat{\delta}(q_0, w)$. At the beginning, only q_0 is active state.

An algorithm $\mathcal{A}$ *simulates a run of an NFA*, if $\forall w$, $w \in \Sigma^*$, it holds that $\mathcal{A}$ with given w at the input reports all information associated with each final state q_f, $q_f \in F$, after processing w, if and only if $q_f \in \hat{\delta}(q_0, w)$.

Hamming distance [7] $D_H(v, w)$ between two strings $v, w \in \Sigma^*$, $|v| = |w|$ is a minimum number of edit operations *replace* needed to convert v to w. *Levenshtein distance* [16] $D_L(v, w)$ between two strings $v, w \in \Sigma^*$, is a minimum number of edit operations *replace*, *insert*, and *delete* needed to convert v to w. *Damerau distance* [4] (also called *generalized Levenshtein distance*) $D_D(v, w)$ between two strings $v, w \in \Sigma^*$, is a minimum number of edit operations *replace*, *insert*, *delete*, and *transpose* needed to convert v to w. Each symbol of v can participate at most in one edit operation *transpose*.

The exact string matching is the task of searching for all occurrences of pattern P (of length $m = |P|$) in text T (of length $n = |T|$). The approximate string matching is then searching for all occurrences of string $w \in \Sigma^*$ in T so that $D(w, P) \leq k$ for given k and D. Given edit distance D can be one of D_H, D_L, and D_D.

First, let us summarize theoretical results [9] for all the implemented methods. Table 1 holds a well-arranged characteristics of each algorithm. We present space and time complexity of each method. w denotes length of computer word used. Reduced NFAs are described later.

method	space complexity	time complexity
Exact string matching		
Boyer-Moore	$\mathcal{O}(m + \lvert\Sigma\rvert)$	preprocess $\mathcal{O}(m + \lvert\Sigma\rvert)$ run $\mathcal{O}(n)$
direct construction of DFA	$\mathcal{O}(m\lvert\Sigma\rvert)$	preprocess $\mathcal{O}(m\lvert\Sigma\rvert)$ run $\mathcal{O}(n)$
Bit parallelism	$\mathcal{O}(\lceil\frac{m}{w}\rceil + m\lvert\Sigma\rvert)$	preprocess $\mathcal{O}(m + \lceil\frac{m}{w}\rceil\lvert\Sigma\rvert)$ run $\mathcal{O}(\lceil\frac{m}{w}\rceil n)$
Dynamic programming	$\mathcal{O}(m)$	preprocess $\mathcal{O}(m)$ run $\mathcal{O}(mn)$
Approximate string matching Finite automata		
NFA, Hamming distance	$\mathcal{O}((km - \frac{k^2}{2} + \frac{3k}{2})\lvert\Sigma\rvert + km)$	$\mathcal{O}((km - \frac{k^2}{2} + \frac{3k}{2})\lvert\Sigma\rvert + km)$
NFA, Levenshtein distances	$\mathcal{O}(k^2m\lvert\Sigma\rvert - \frac{2}{3}k^3\lvert\Sigma\rvert + km\lvert\Sigma\rvert)$	$\mathcal{O}(k^2m\lvert\Sigma\rvert - \frac{2}{3}k^3\lvert\Sigma\rvert + km\lvert\Sigma\rvert)$
reduced NFA, Hamming d.	$\mathcal{O}(mk\lvert\Sigma\rvert - k^2 + k\lvert\Sigma\rvert)$	$\mathcal{O}(mk\lvert\Sigma\rvert - k^2 + k\lvert\Sigma\rvert)$
reduced NFA, Levenshtein d.	$\mathcal{O}(k^2m\lvert\Sigma\rvert - k^3\lvert\Sigma\rvert + km\lvert\Sigma\rvert)$	$\mathcal{O}(k^2m\lvert\Sigma\rvert - k^3\lvert\Sigma\rvert + km\lvert\Sigma\rvert)$
DFA	$\mathcal{O}(\lvert\Sigma\rvert\lvert Q_{DFA}\rvert)$	run $\mathcal{O}(n)$
Approximate string matching Bit parallelism		
non-reduced NFA	$\mathcal{O}(k\lceil\frac{m}{w}\rceil + m\lvert\Sigma\rvert)$	preprocess $\mathcal{O}(m + \lceil\frac{m}{w}\rceil\lvert\Sigma\rvert + k\lceil\frac{m}{w}\rceil)$ run $\mathcal{O}(kn\lceil\frac{m}{w}\rceil)$
reduced NFA	$\mathcal{O}(k\lceil\frac{m-k}{w}\rceil + k(m-k)\lvert\Sigma\rvert)$	preprocess $\mathcal{O}(m + \lceil\frac{m}{w}\rceil\lvert\Sigma\rvert + k\lceil\frac{m-k}{w}\rceil\lvert\Sigma\rvert)$ run $\mathcal{O}(kn\lceil\frac{m-k}{w}\rceil)$
Approximate string matching Dynamic programming		
non-reduced NFA	$\mathcal{O}(m)$	preprocess $\mathcal{O}(m)$ run $\mathcal{O}(mn)$
reduced NFA	$\mathcal{O}((m-k) + m\lvert\Sigma\rvert)$	preprocess $\mathcal{O}((m-k) + m\lvert\Sigma\rvert)$ run $\mathcal{O}((m-k)n)$

Table 1. Space and time complexities of the methods used for the exact and approximate string matching

We used six document classes for testing so that we can see the different behavior of algorithms for various structures of strings and various alphabet sizes. For each class we use one file from Canterbury and Calgary Corpora. The classes, their numeric identifiers and the files are listed in Tab. 2.

id	file name	$\lvert\Sigma\rvert$	description
1	search2	105	binary file—executable system program
2	bliss2.bmp	122	binary file—true color bitmap noised image
3	c.html	82	text file—c source converted to hypertext document
4	book1	63	text file—English text, book Madding Crowd by T.Hardy
5	pi.txt	10	text file—number π, sequences of ascii decimal numbers
6	E.coli	4	text file—DNA sequence of bacteria Escherichia coli

Table 2. Text classes used for testing of the string matching methods

For testing we used a computer with processor AMD Athlon XP 2000+, memory 256MB RAM running GNU/Linux SUSE 8.2.

3 Exact String Matching

In the exact string matching we are interested in all occurrences of pattern P in text T. Number of algorithms have been developed for this task (e.g. [14, 3, 13]). We focus only on finite automata approach in this paper. For comparison we also consider Boyer-Moore algorithm.

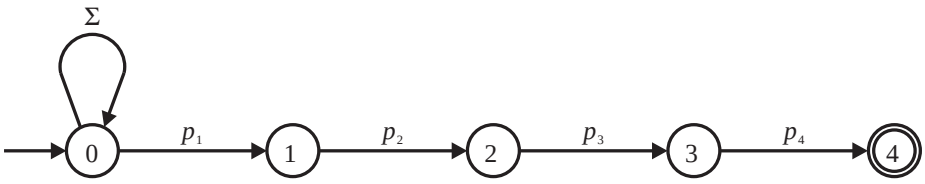


Fig. 1. NFA for the exact string matching ($m = 4$)

NFA for the exact string matching is shown in Fig. 1. The NFA can be transformed to DFA which has the same number of states. Moreover, there is a direct construction algorithm available working in $\mathcal{O}(m)$ time. For comparison of the transformation and the direct construction algorithm see Fig. 2 and 3. Note that for short DNA patterns (id = 6), the direct DFA construction is slower although it is linear.

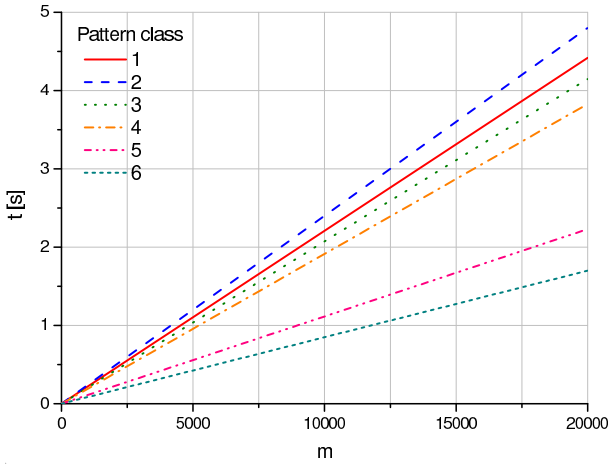


Fig. 2. Example of dependence of the preprocessing time (t) on the length of pattern (m) for direct DFA creation for different pattern classes (measured 100 cycles)

3.1 Bit Parallelism

In the bit parallel simulation for the exact string matching we use one bit vector $R = r_1 r_2 \cdots r_m$ where each state except the initial state is represented by one bit. R_i represents value of R in step i (after reading ith input symbol) and R_{i-1} in the previous step. Formula 1 shows the simulation. Table D contains mask vectors for all alphabet symbols. **shl** and OR represent bitwise operations `left shift` and `or` respectively. For more details see [9, 10].

$$\begin{aligned} r_{j,0} &\leftarrow 1, & 0 < j \leq m \\ R_i &\leftarrow \mathbf{shl}(R_{i-1}) \text{ OR } D[t_i], & 0 < i \leq n \end{aligned} \tag{1}$$

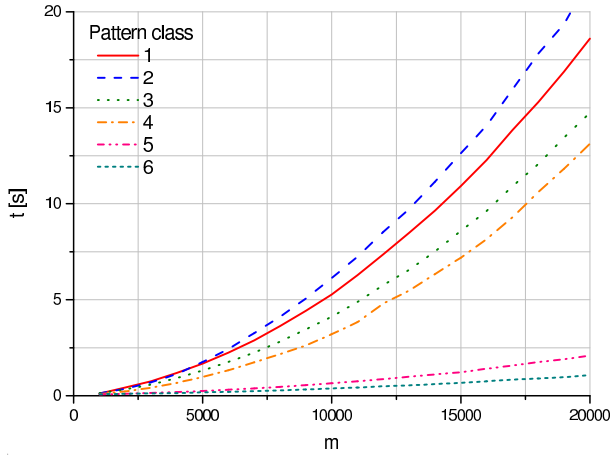


Fig. 3. Example of dependence of the preprocessing time (t) in dependence on the length of pattern (m) for transformation of NFA to the equivalent DFA for different pattern classes

3.2 Dynamic Programming

Dynamic programming simulation for the exact string matching uses integer vector D—one integer for each state. Formula 2 shows the simulation. For more details see [9, 11].

$$\begin{aligned} &d_{j,0} \leftarrow 1, && 0 < j \leq m \\ &d_{0,i} \leftarrow 0, && 0 \leq i \leq n \\ &d_{j,i} \leftarrow \textbf{if } t_i = p_j \textbf{ then } d_{j-1,i-1} \textbf{ else } d_{j-1,i-1} + 1, && 0 < i \leq n, 0 < j \leq m \end{aligned} \tag{2}$$

While the bit parallelism for the exact string matching runs in time $\mathcal{O}(\lceil \frac{m}{w} \rceil n)$ the dynamic programming runs in time $\mathcal{O}(mn)$.

Comparison of all above-mentioned methods is in Fig. 4. As one can expect, Boyer-Moore algorithm is the winner. The simulation methods as well as the DFA applies in the approximate string matching where the idea of Boyer-Moore algorithm is not efficient.

4 Approximate String Matching—Hamming Distance

Nondeterministic finite automaton for the approximate string matching using the Hamming distance can be constructed by connecting $k+1$ copies of NFA for the exact string matching by transitions representing edit operation *replace* and removing unreachable states. The resulting NFA is shown in Fig. 5. Transition *replace* can be labeled either by complement of matching symbol or by any symbol of alphabet. Thus we distinguish $\bar{p}$-version and Σ-version respectively. Both versions do correctly the approximate string matching and the resulting DFAs are isomorphic [9].

We consider also reduced NFA where we omit states needed only to distinguish the exact number of errors. These states are states 3, 4 and 8 in Fig. 5. The reduced NFA does not tell what is the exact number of errors in the found string but ensures that the number is not greater than k. See [9] for more details.

We compared time for transformation of NFA for the approximate string matching using the Hamming distance to DFA for both versions. As you can see in Fig. 6, $\bar{p}$-version needs less time for transformation NFA to DFA for all document classes and thus it is more suitable. Further in this section we consider $\bar{p}$-version for DFA.

When we look at Fig. 7 we can see dependence of DFA size on pattern length m and number k of errors allowed. It seems that while the dependence on k is exponential, the

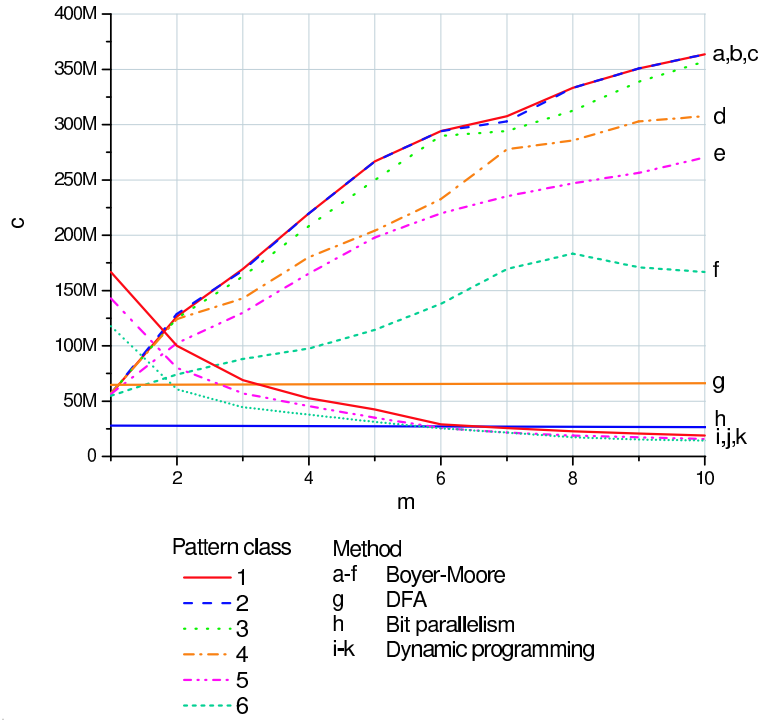


Fig. 4. Comparison of the average character matching speed (c) for various exact string matching methods in dependence on the length of pattern (m) for various pattern classes

dependence on m tends to be linear. As expected, there is a linear dependence of transformation time on DFA size.

The number of states of the resulting DFA strongly depends on the structure of pattern P. Both the size of the input alphabet $|\Sigma|$ and the sequences of the repeating characters are the main factors that affect the resulting DFA. Unfortunately, we are not able to tell how many states the DFA will have even if we know the structure of pattern P. Fig. 7 is an example of the dependence of total number of DFA states $|Q|$ on the number of errors k and pattern length m for the approximate string matching using the Hamming distance. The increase of states is still exponential but very far from the theoretical upper bound for this type of DFA. The number of states for $m = 19, k = 9$ is about 6×10^5 in this example (a sequence of repeating characters of variable length has been used) while the upper bound is 5.12×10^{17}. Still, the exponential growth of the number of DFA states limits us to use small m and k.

The structure of the pattern also applies as we can see on Fig. 8. We see that the binary patterns and common English text produce a relatively small number of states. For the pattern from C source html code, where the periodicity is increased already, we see an increase in the number of DFA states. For the number π and the DNA sequence, where the huge overlapping of the pattern occurs, the increase of the DFA size is even more significant.

A surprising result of the experiments was that the reduced approximate string matching NFA using the Hamming distance produces sometimes even slightly larger DFA than than non-reduced NFA. Significant decrease of DFA size is for $k > m/2$ which may not be too useful in practice. See Fig. 9 and 10.

We may also try to minimize the resulting DFA. Since the matching speed is the same for the both minimal and non-minimal automata, this makes sense only if the minimal automata should have significantly less states. The amount of states removed during the minimization is dependent primarily on the structure of the pattern. For patterns with small alphabets, like a DNA sequences, the minimization is very efficient (see Fig. 11)—

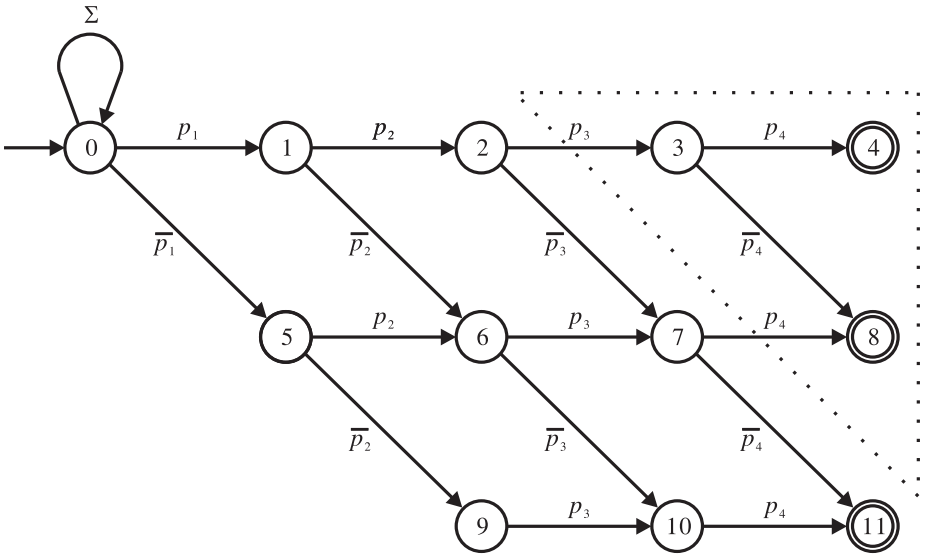


Fig. 5. NFA for the approximate string matching using the Hamming distance ($m = 4$, $k = 2$, $\bar{p}$-version)

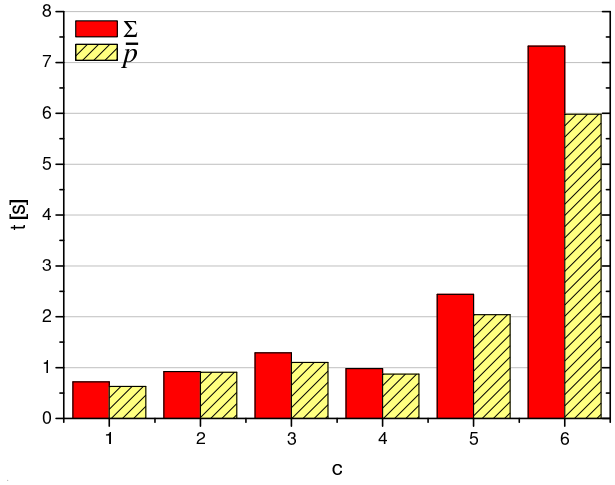


Fig. 6. Comparison of the time (t) required for the transformation of NFA to equivalent DFA for different pattern classes (c) and NFA versions using the Hamming distance

the experiments show that up to 70% of DFA states can be removed. For the common text, there is about 25% of space saved. The minimization also depends on the number of errors k. The automata with more levels should be relatively more minimized. This is also illustrated in Fig. 11. We should also consider the time required for minimization. Unfortunately, the increase of time is very significant and shows an exponential response to the length of pattern m.

4.1 Bit Parallelism

When simulating a run of an approximate string matching NFA for the Hamming distance, bit parallelism uses one bit vector for each error level of the NFA. The running phase time is $\mathcal{O}(kn\lceil\frac{m}{w}\rceil)$. The increase of the running time for increasing k is illustrated in Fig. 12. Our experiments show that the simulation of Σ-version of NFA runs faster than $\bar{p}$-version. We confine ourselves to the statement here that the Σ version should be used in every case.

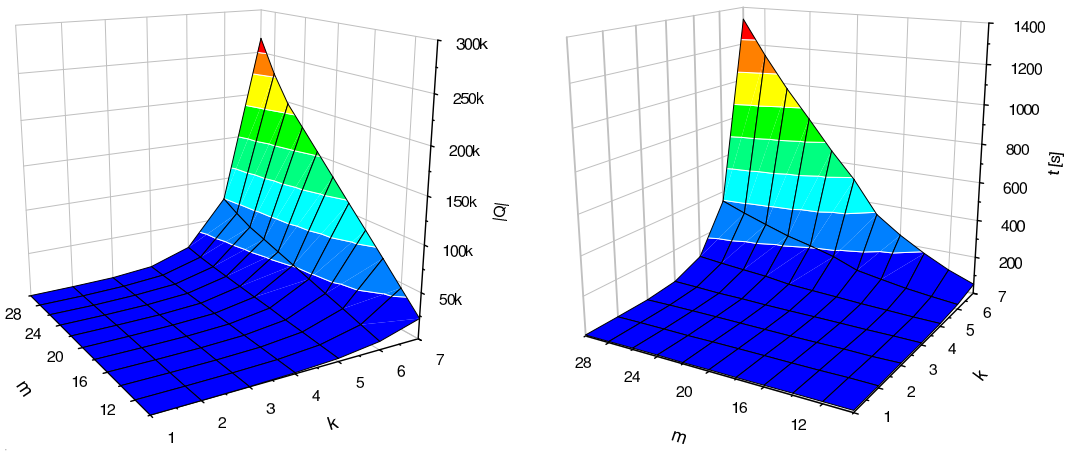


Fig. 7. Example of increase of the number of DFA states ($|Q|$) and the time (t) in dependence on the length of pattern (m) and the number of errors (k) for the approximate string matching using the Hamming distance

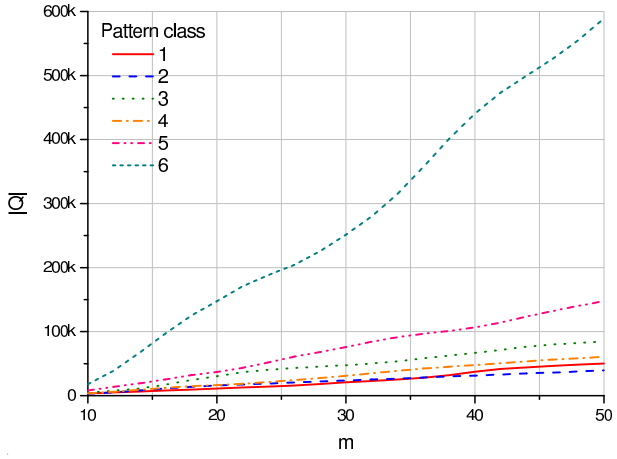


Fig. 8. Increase of the number of DFA states ($|Q|$) in dependence on the length of pattern (m) for various pattern structures for the approximate string matching using the Hamming distance

$$
\begin{aligned}
r^l_{j,0} &\leftarrow 1, && 0 < j \leq m, 0 \leq l \leq k \\
R^0_i &\leftarrow \mathbf{shl}(R^0_{i-1}) \text{ OR } D[t_i], && 0 < i \leq n \\
R^l_i &\leftarrow (\mathbf{shl}(R^l_{i-1}) \text{ OR } D[t_i]) \text{ AND } \mathbf{shl}(R^{l-1}_{i-1}), && 0 < i \leq n, 0 < l \leq k
\end{aligned}
\tag{3}
$$

We expected that reduced version of NFA would run faster by approximately 20% due to the omitted **shl** operation. However, this is not always true. The processing time of the simulation differs very significantly for different pattern classes unexpectedly, while the simulation of the non-reduced NFA remains the same. The situation is illustrated in Fig. 13 for $k = 20$.

It seems that the processing speed is dependent on the size of the pattern alphabet. The reason is probably in the way the array of reduced vectors is accessed—for smaller range of index j the code is running significantly faster.

As in the case of DFA, the use of reduced NFAs is efficient only when the number of errors allowed is significant with respect to the length of the pattern.

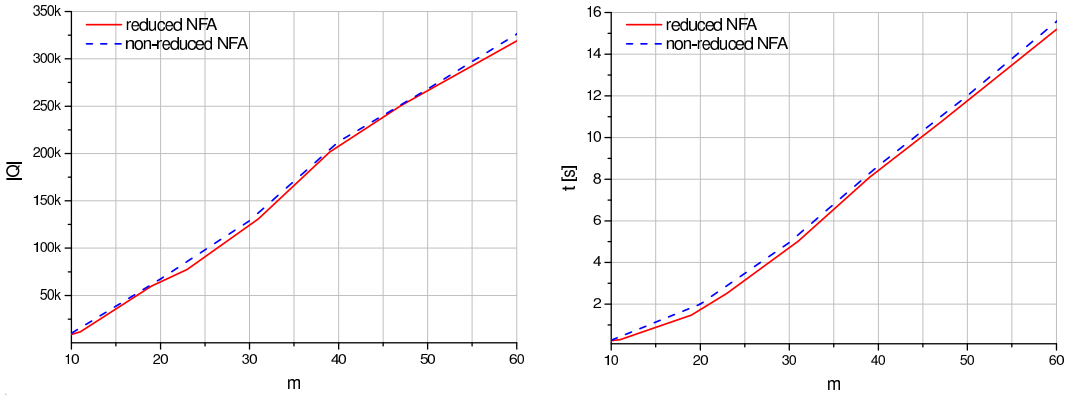


Fig. 9. Comparison of the number of DFA states ($|Q|$) and the transformation time (t) resulting from reduced and non-reduced approximate string matching NFA using the Hamming distance in dependence on the length of pattern (m) ($k = 6$)

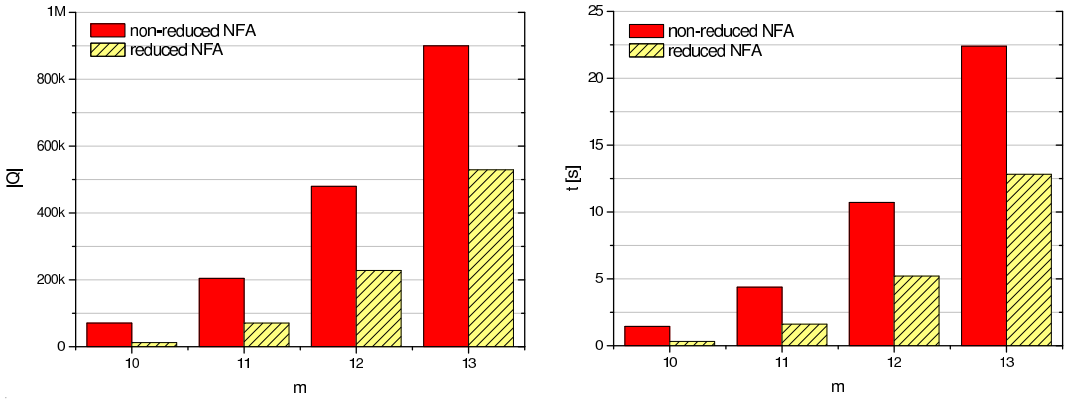


Fig. 10. Comparison of the number of DFA states ($|Q|$) and the transformation time (t) resulting from reduced and non-reduced approximate string matching NFA using the Hamming distance in dependence on the length of pattern (m) ($k = 9$)

4.2 Dynamic Programming

Using the dynamic programming for the approximate string matching is exactly the same as for the exact string matching since the method is not sensitive to the number of errors k. For different k the number of operations remains the same, only the condition testing the string match is changed. This is especially useful for the large values of k as discussed below.

$$\begin{aligned} &d_{j,0} \leftarrow k+1, && 0 < j \leq m \\ &d_{0,i} \leftarrow 0, && 0 \leq i \leq n \\ &d_{j,i} \leftarrow \textbf{if } t_i = p_j \textbf{ then } d_{j-1,i-1} \textbf{ else } d_{j-1,i-1} + 1, && 0 < i \leq n, 0 < j \leq m \end{aligned} \tag{4}$$

The first aspect to consider when choosing the most efficient approximate matching method is the number of errors k and the length of pattern m. It is obvious that, especially for larger number of errors, we are not always able to transform approximate string matching NFA to the equivalent DFA. To be more concrete, we were able to create a DFA for at most 11 errors using the non-reduced NFA. For the reduced NFA, the limit is raised to 14. Beyond these limits, the DFA have too many states to fit in the memory of our testing system.

In such cases we should use one of the simulation methods. Even when the DFA is "small" enough to fit in the memory, we should consider the length of the input text. Since

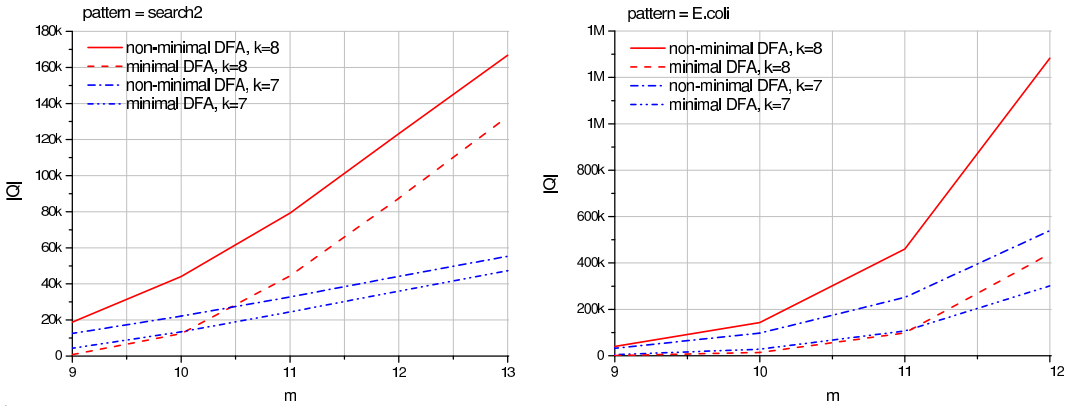


Fig. 11. Comparison of the number of DFA states ($|Q|$) for non-minimal and minimal DFA for various pattern classes and different values of k in dependence on the length of pattern (m) for the approximate string matching using the Hamming distance

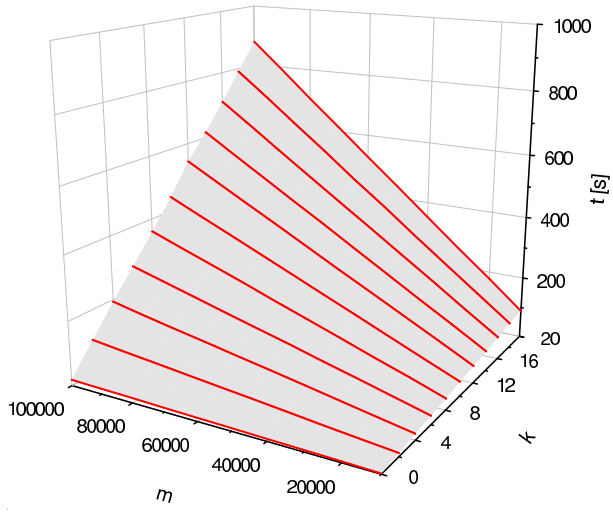


Fig. 12. Example of increase of the run time (t) for bit parallelism simulation in dependence on number of errors (k) and size of input text (n) for the approximate string matching using the Hamming distance ($m = 10\,000$)

the generation of the DFA takes a relatively long time, it does not make much sense to use it for very short input texts where the simulation methods with almost no preprocessing are very efficient.

The construction of the resolution system is based on the practical observation of the behavior of each approximate string matching algorithm. Using the measured data for each simulation method, we are able to tell what the resulting time will be for any input variable combination since the simulation algorithms are—with some minor exceptions—strictly linear to all the variables. For the finite automata, the situation is little more complicated. However, we are still able to gather enough data to estimate the properties of the resulting DFA.

Fig. 14 is an example of linear dependence of time with an increasing length of pattern for the dynamic programming. However, we can see some minor abnormalities here. For the $m = 14\,000$ and $m = 20\,000$ (approximately) the line changes its tangent for all classes of patterns. The way the processor handles the memory probably changes a little at these points. We can then approximate the run of the algorithm by three line segments, one for each section (separated by dashed vertical line in Fig. 14).

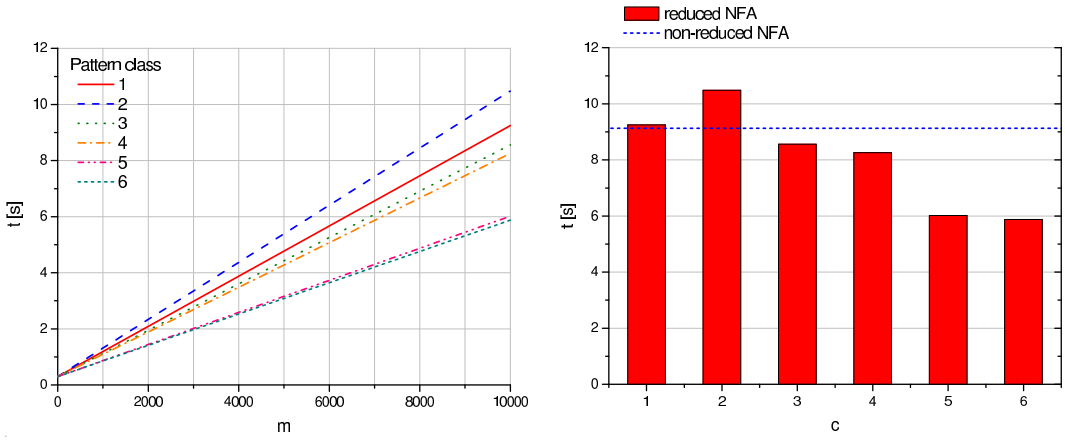


Fig. 13. Example of dependence of the run time (t) for bit parallelism simulating reduced NFA using the Hamming distance on the length of pattern (m) for various pattern classes (c)

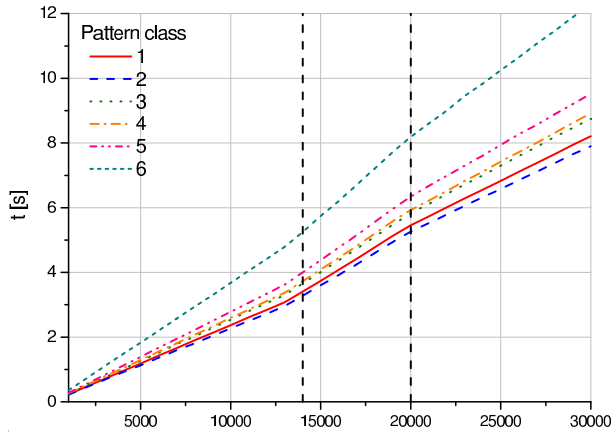


Fig. 14. Dependence of the running time (t) on the length of pattern (m) using dynamic programming for various pattern classes for the approximate string matching using the Hamming distance ($n = 100\,000$)

Let us demonstrate the functionality of the resolution system on the example of bit parallelism in comparison to dynamic programming. Suppose we have a pattern of length $m = 1\,000$. The decision should be made according to the number of errors k. The method of bit parallelism benefits from the fact that the number of computer words processed during one step of the process is equal to $k\lceil\frac{m}{w}\rceil$. However, for the number of errors $k = w$, the m words have to be processed, which is exactly the same number as for the dynamic programming. For increasing k the situation becomes even worse. Since the dynamic programming is not sensitive to the k, we should expect bit parallelism to be over-performed by the dynamic programming for some value of the k. To find the limit value of k we can use the resolution system. We can try to enter a different number of k for fixed m and length of input text n. The system provides us with the results shown in Fig. 15. The limit value of k, where the dynamic programming is becoming more efficient depends on the pattern class. For example, for the matching pattern, being the DNA sequence, we read the value of k should be greater than 12. This is verified by real measurements as shown also in Fig. 15.

For finite automata the situation is much more complicated. The time and space complexity may change in a more complex way for the increasing length of pattern. Take a look at Fig. 16 depicting the situation for DFA with $k = 1, 2$. We see that the increase of the number of states is pretty linear and therefore can be described by a simple linear equation. However, the time dependance is far from being linear.

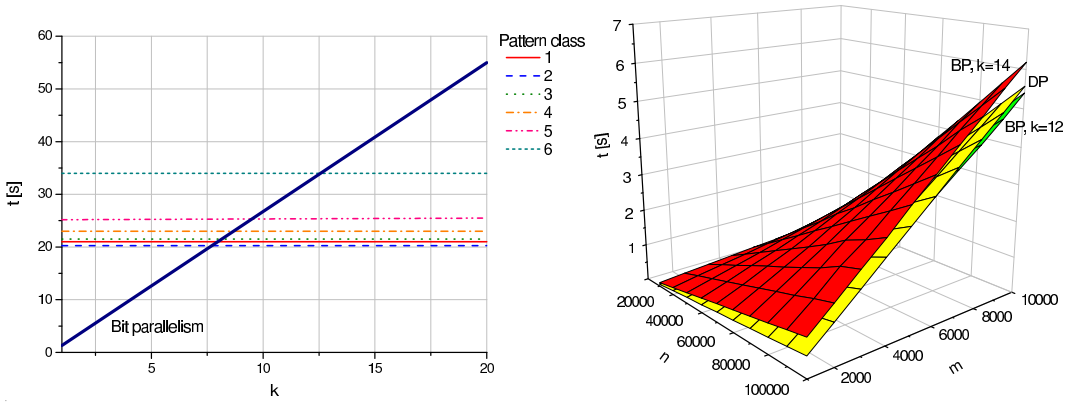


Fig. 15. Comparison of bit parallelism and dynamic programming for various number of errors (k) for the approximate string matching using the Hamming distance

The solution is to use the available experimental measurements and construct a curve that fits the data. Curve fitting is a very advanced area of data analysis and is not the subject of this paper. The package called SIMFIT[1] has been used for the data analysis and curve fitting. The method used for curve fitting is based on the process of unconstrained weighted least squares regression using sum of sequence of increasing number of exponentials.

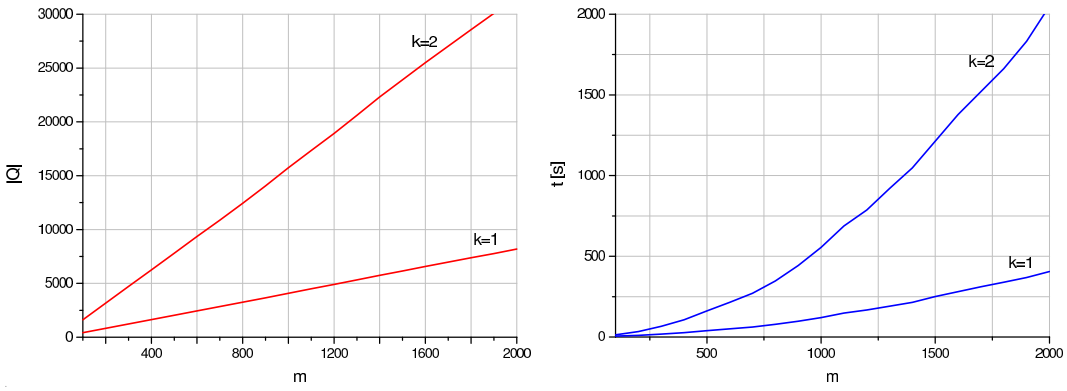


Fig. 16. Example of dependance of the number of DFA states ($|Q|$) and the construction time (t) on the length of pattern (m) for different number of errors (k) for the approximate string matching using the Hamming distance

As mentioned above the huge preprocessing time of DFA is compensated with the fast matching phase. Therefore, we may ask how long the input text should be so the DFA becomes efficient. The resolution system should be used to find the solution. Lets take a look at the example of pattern of length $m = 500$ being an English text. Suppose we are looking for the occurrences of the pattern in the input text with at most $k = 4$ mismatches. The estimation is, that the construction of the DFA should take 26 seconds and the resulting DFA should have 112 239 states. We can also estimate the running times for each simulation method depending on the size of input text. This situation is illustrated in Fig. 17. We see that for $n > 18 \times 10^6$ the resulting time is less than for any other simulation method. The time for the DFA is displayed as a constant, since the change of the running time over a given interval is hardly noticeable. We can verify this estimation with real measurement.

[1] A Package for Simulation, Curve fitting, Graph plotting and Data Analysis, W. G. Bardsley.

The "real" DFA has been constructed in a time almost equal to the estimation and the number of states has been 112987, which is less than 1% deviation.

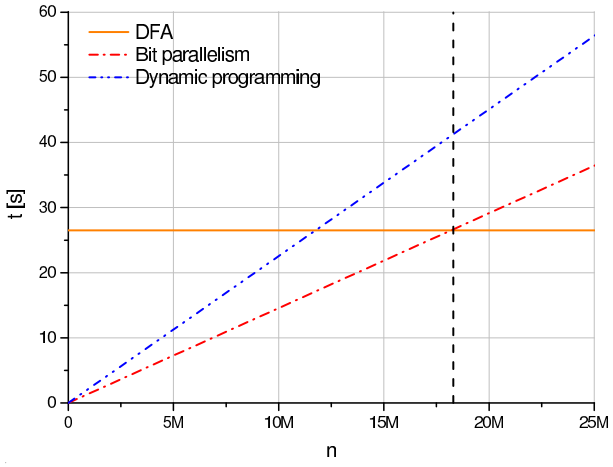


Fig. 17. Comparison of the approximate string matching methods using the Hamming distance ($m = 500, k = 4$)

At this point, we also have to mention one important attribute of the deterministic automata. The DFA can be precomputed in advance and stored on any data storage. At the moment we actually need to perform the matching task, the DFA can be quickly loaded into the memory and the matching should begin immediately. The advantages of such an approach are obvious. For example, DFA should be created on one dedicated system and the resulting DFA should be sent to other systems participating in the matching task.

5 Approximate String Matching—Levenshtein Distance

When using the approximate string matching which utilizes the Levenshtein distance, there are two more edit operations *delete* and *insert* allowed. Fig. 18 shows the corresponding NFA. Vertical transitions represent *insert* and diagonal ε-transitions represent *delete*. The same procedures as with the Hamming distance are used, however, the processing times are basically higher because of additional computing because of added edit operations.

The main difference between NFA for the Levenshtein distance and the Hamming distance automaton is the presence of ε-transitions. Since these transitions expand through all levels of the NFA, the inner structure is totally different.

We should first consider using the Σ, or $\bar{p}$ version of the NFA for the transformation. Fig. 19 illustrates that the Σ version NFA produces DFA with fewer states and in less time. We have experienced the difference about 10%. Because of the expanded ε-transitions, the states are no longer so "isolated" as in the case of the Hamming distance NFA. For the Σ version NFA, where there are even more transitions, a greater number of states are rendered synonymous during the DFA construction.

The next thing we are interested in is a response to the different pattern classes. Fig. 20 shows that the behavior of DFA for the Levenshtein distance is just an opposite to DFA for the Hamming distance. The patterns with the smaller alphabets, and large overlapping results in DFA which have significantly fewer states. We should also notice that the increase of states is non-linear as is the increase of the transformation time[2].

[2] Compare to the Hamming distance DFA, with the linear increase of the number of states.

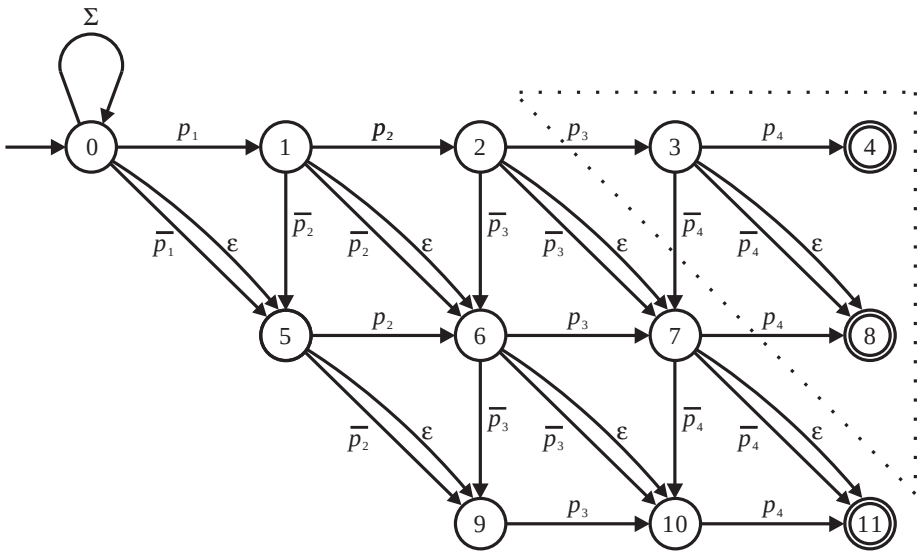


Fig. 18. NFA for the approximate string matching using the Levenshtein distance ($m = 4$, $k = 2$, $\bar{p}$-version)

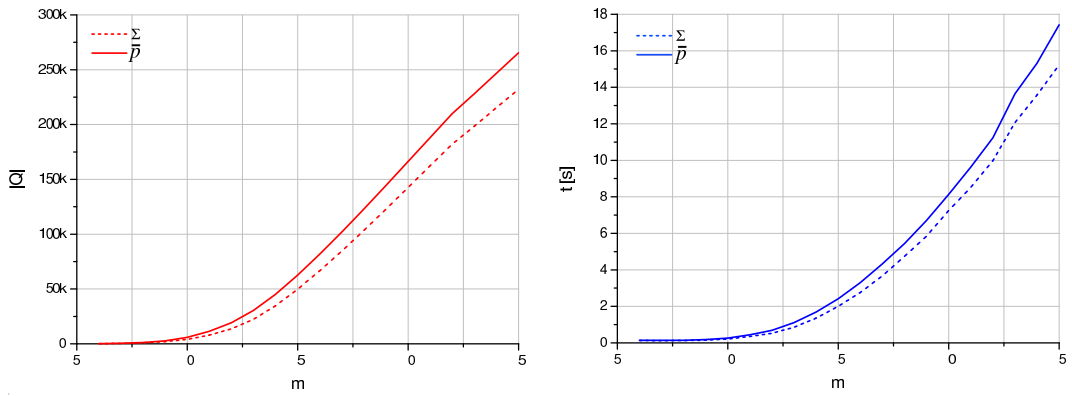


Fig. 19. Dependence of the number of DFA states ($|Q|$) and the transformation time (t) on the length of pattern (m) for various versions of approximate string matching NFA using the Levenshtein distance ($k = 5$)

As for the Hamming distance, we can also use reduced NFA. If you remove states 3, 4, and 8 from Fig. 18, you get the reduced NFA for the approximate string matching using the Levenshtein distance. Using reduced NFA follows the same rules as for the Hamming distance. The reduction is especially noticeable for the $k = m-1$. In such a case, where only two diagonals remain, the resulting DFA has very small number of states being in $\mathcal{O}(k)$. This is caused by the ε-transitions which render the diagonal states to behave similarly.

There is also a question of the minimal automata. Our experiments have proven that the DFA which results from the NFA using the Levenshtein distance can be minimized to approximately half its size. Fig. 21 illustrates the difference between the number of states of minimal and non-minimal DFA being created for English text. For other pattern classes, the situation is very similar—the variations are about 10%. The increase in the time is also illustrated in Fig. 21.

The description of the DFA behavior for the resolution system, constructed from the measured data, is then based on two estimated exponential curves—one for the states expansion estimation and the second for the time estimation.

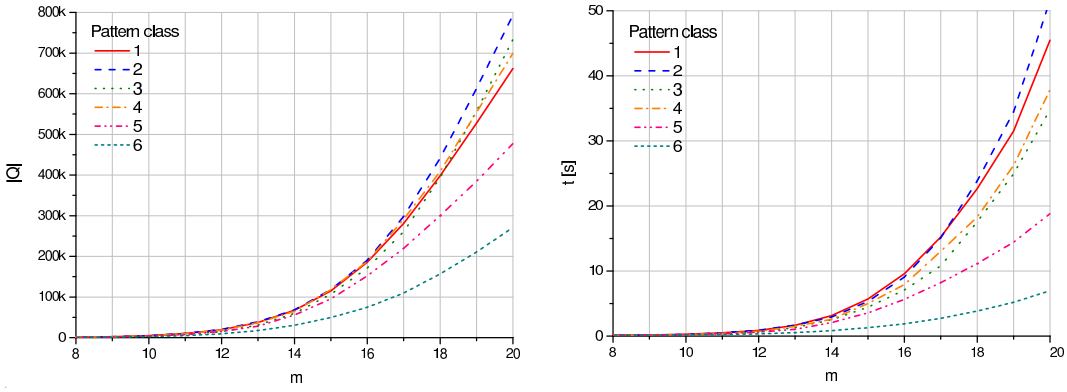


Fig. 20. Dependence of the number of DFA states ($|Q|$) and the transformation time (t) on the length of pattern (m) for various pattern classes for the approximate string matching using the Levenshtein distance ($k = 6$)

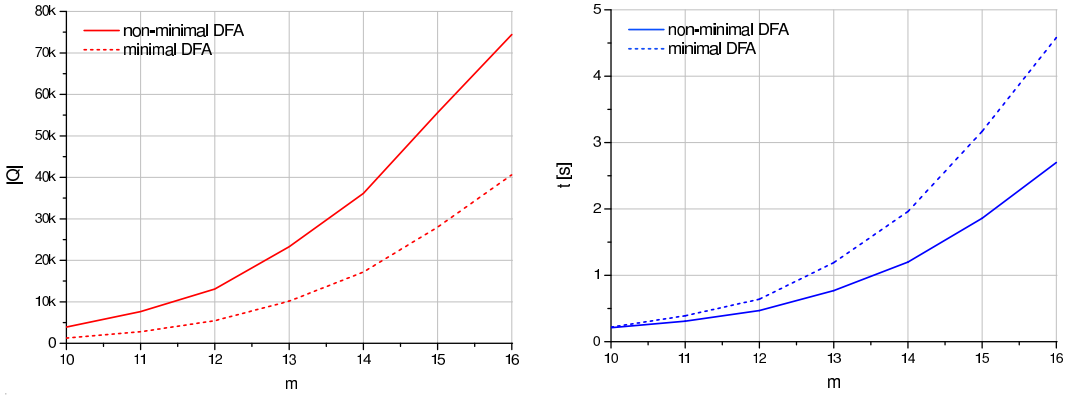


Fig. 21. Comparison of the number of DFA states ($|Q|$) and the creation time (t) for non-minimal and minimal DFA in dependence on the length of pattern (m) for the approximate string matching using the Levenshtein distance ($k = 7$)

5.1 Bit Parallelism

The implementation of the simulation using the Levenshtein distance differs only in the number of bitwise operations performed in one step of the algorithm in comparison to the Hamming distance NFA simulation. Both Σ and $\bar{p}$ NFA versions of the simulation are available. We use the Σ as long as it does not require any additional preprocessing, and the running phase is much more efficient than for the $\bar{p}$ version.

$$
\begin{aligned}
r^l_{j,0} &\leftarrow 0, && 0 < j \leq l, 0 < l \leq k \\
r^l_{j,0} &\leftarrow 1, && l < j \leq m, 0 \leq l \leq k \\
R^0_i &\leftarrow \mathbf{shl}(R^0_{i-1}) \text{ OR } D[t_i], && 0 < i \leq n \\
R^l_i &\leftarrow (\mathbf{shl}(R^l_{i-1}) \text{ OR } D[t_i]) \\
&\quad \text{AND } \mathbf{shl}(R^{l-1}_{i-1} \text{ AND } R^{l-1}_i) \text{ AND } (R^{l-1}_{i-1} \text{ OR } V), 0 < i \leq n, 0 < l \leq k
\end{aligned}
\tag{5}
$$

The simulation of the non-reduced NFA shows a strict linearity to all variables as shown in Fig. 22 and does not show any response to the pattern classes.

For the simulation of the reduced NFA using the bit parallelism, the same problem occurs as in the case of the Hamming distance. The processing time differs for the different pattern again—though not so drastically as for the Hamming distance. We must also understand that the process running the simulation of the reduced automata can never run faster than the non-reduced simulation for the case when the vectors are not "physically" shortened.

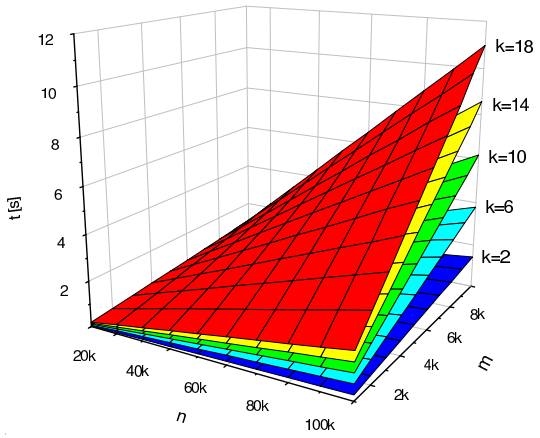


Fig. 22. Example dependence of the computing time (t) on the length of pattern (p) and the length of input text (n) for various values of number of errors (k)

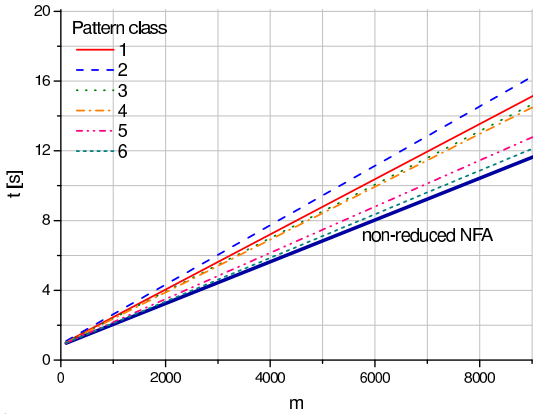


Fig. 23. Comparison of the processing time (t) required for the simulation of reduced NFA using bit parallelism for various pattern classes in dependence on the length of pattern (m) for the approximate string matching using the Levenshtein distance ($k = 20$)

5.2 Dynamic Programming

We can apply the same principles as to the dynamic programming using the Hamming distance. For dynamic programming using the Levenshtein distance we can choose from either the implementation of Σ or $\bar{p}$ version of the simulated NFA. Fig. 24 is comparison of the running times for the various pattern structures as well as a comparison of the Σ and $\bar{p}$ version implementation.

$$\begin{aligned} &d_{j,0} \leftarrow j, && 0 \leq j \leq m \\ &d_{0,i} \leftarrow 0, && 0 \leq i \leq n \\ &d_{j,i} \leftarrow \textbf{min}(\textbf{if } t_i = p_j \textbf{ then } d_{j-1,i-1} \textbf{ else } d_{j-1,i-1} + 1, && \\ &\qquad \textbf{if } j < m \textbf{ then } d_{j,i-1} + 1, && \\ &\qquad d_{j-1,i} + 1), && 0 < i \leq n, 0 < j \leq m \end{aligned} \tag{6}$$

One significant difference in comparison to the dynamic programming using the Hamming distance is the availability of the reduced NFA simulation. However, even if we are not interested in the exact number of errors, we should carefully consider using of the reduced NFA here. Two things ought to be considered. The preprocessing of the G matrix in time $\mathcal{O}((m+1)|\Sigma|)$ is not negligible, and the algorithm itself is running slower than the algorithm simulating the non-reduced NFA. Fig. 25 is a comparison of a run of both

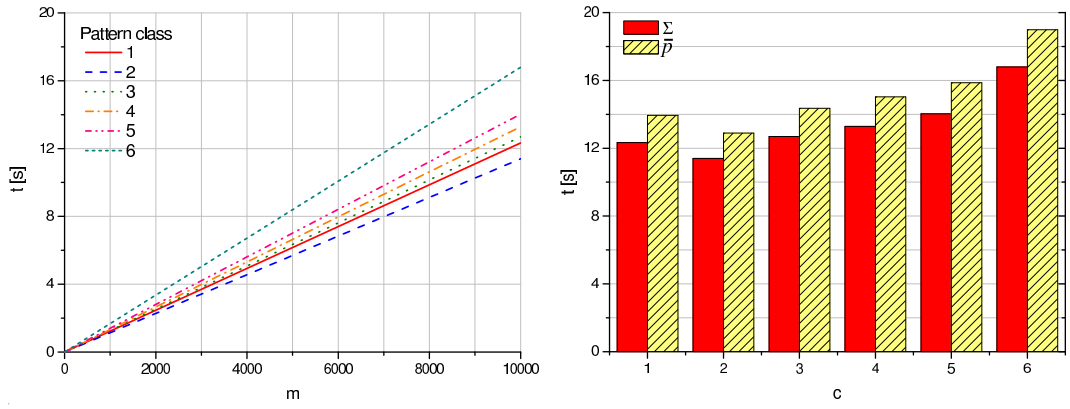


Fig. 24. Comparison of the running time (t) for the NFA simulation using dynamic programming for various pattern classes (c) and different NFA versions in dependence on length of pattern (m) for the approximate string matching using the Levenshtein distance

reduced and non-reduced simulations using the dynamic programming with the number of differences $k = 1$. To benefit from the processing of the shortened vector d_i we should increase the value of k. An example of such a process is also shown in Fig. 25. The pattern of length $m = 100$ being a "acgt" DNA sequence has been used in this example. We see that the lines representing the elapsed time in dependence on the value of k cross each other for $k \doteq 21$. So, if we are looking for the occurrences of the pattern in a text with less than 21 differences, the simulation of non-reduced NFA should be used even if we do not care about the exact number of errors. Let us note this example has been simulated using our resolution system and has been verified with the real measurements afterwards.

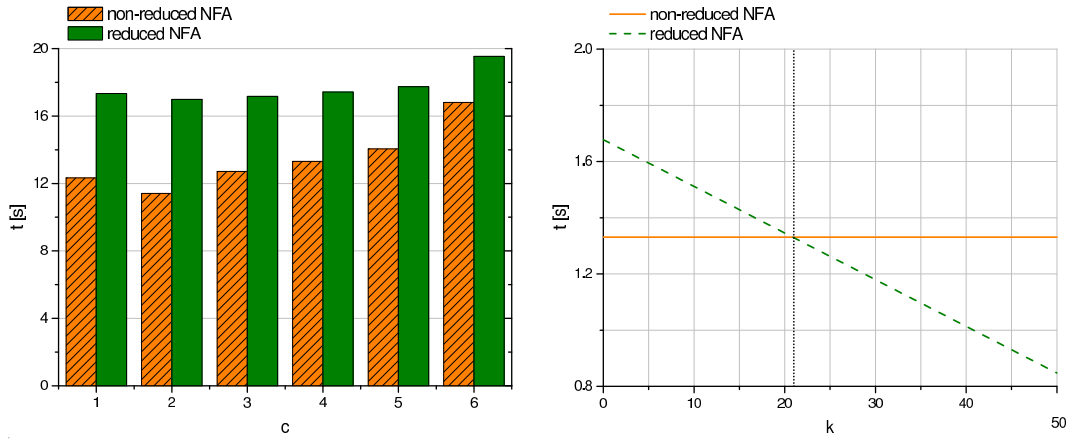


Fig. 25. Comparison of the simulation of reduced and non-reduced NFA using dynamic programming for the approximate string matching using the Levenshtein distance

5.3 Comparison

Here, we will only show the basic comparison of the simulation methods as we did in the previous section. The situation is shown in Fig. 26. We see that the value of k, where the simulation methods are almost equal in their performance, is shifted in the direction of increasing k. For example, for the DNA sequence—the simulation using the bit parallelism is faster than the dynamic programming for the number of errors k being up to 18 (was 12

for the Hamming distance). This increase is caused by the different ratio of performance between the implementation of the Hamming and Levenshtein distance simulation for both methods. In the next section, we present a visual comparison where all of the possible edit distance simulation methods performances are matched together.

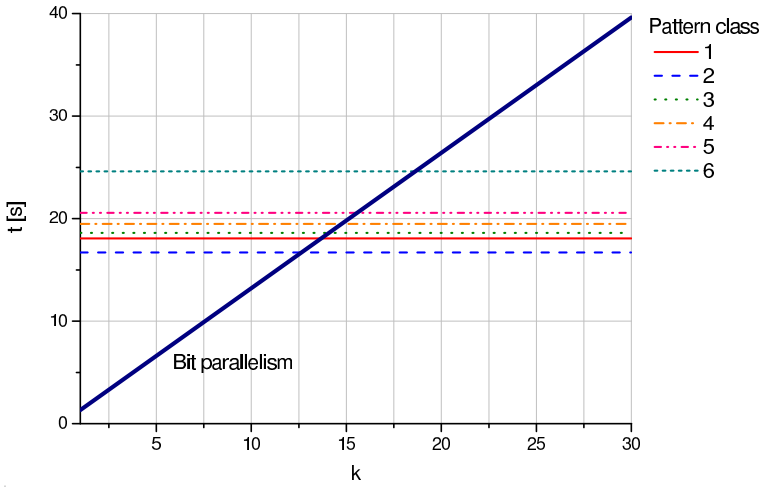


Fig. 26. Comparison of the processing time (t) for the simulation of NFA using dynamic programming for various pattern classes in dependence on the number of errors (k) for the approximate string matching using the Levenshtein distance

6 Approximate String Matching—Damerau Distance

The definition of the Damerau distance allows one more edit operation in addition to the Levenshtein distance—*transpose*. Again, the complexity of the process is increased. For the details see [9].

DFA size is increased by approximately 20% when compared with the Levenshtein distance. Otherwise the behaviour is the same. Σ-version produces smaller DFA. In Fig. 27 we can see DFA sizes for all three edit distances considered in this paper. One can see that for shorter patterns the size of the DFA utilizing the Levenshtein distances may be smaller than for the Hamming distance. However, due to the exponential increase of the number of DFA states the situation is to be inverted for longer patterns.

Bit parallel simulation requires higher preprocessing time since we need precomputed $\mathbf{shr}(D[t_i])$ vectors to speed up the processing. Again, the bit parallel simulation of reduced NFAs cannot run faster than non-reduced ones unless the vectors are "physically shortened".

Dynamic programming simulation for the Damerau distance behaves in the same way as for the Levenshtein distance. The same sensitivity applies to pattern class. Simulation for Σ-version runs faster than for $\bar{p}$-version. Since the formula for computing matrix D is more complex, the simulation of reduced NFA outperform non-reduced for higher k than for the Levenshtein distance (see Fig. 28).

Fig. 29 shows the comparison of the simulation methods as we did for the Hamming and Levenshtein distances. We can see, that the limit value of k when the dynamic programming simulation outperforms bit parallelism is little higher again then for the case of the Levenshtein distance[3]. For the DNA sequence pattern type the time line crosses at the value of $k \doteq 26$ (was 18 for the Levenshtein distance).

[3] We are still taking about the non-reduced Σ version of the NFA.

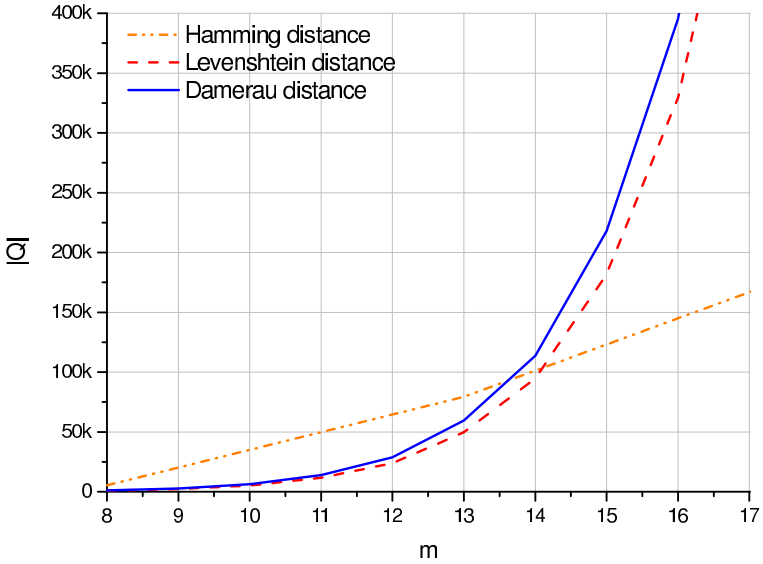


Fig. 27. Comparison of the number of DFA states ($|Q|$) in dependence on the length of pattern (m) for various classes of the DFA ($k = 7$)

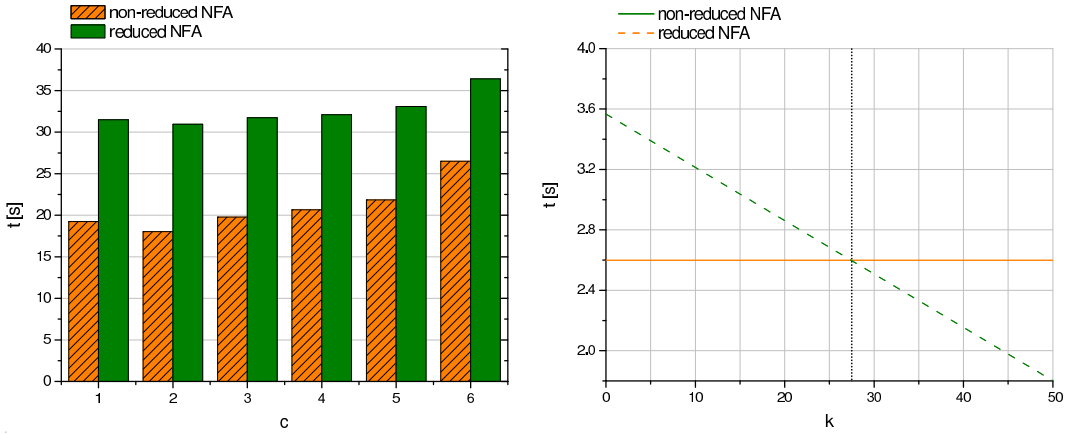


Fig. 28. Comparison of the simulation of reduced and non-reduced NFA using dynamic programming for the approximate string matching using the Damerau distance

The increase of processing time for all discussed edit distances is shown in Fig. 30. It follows the expectation that the more edit operation allowed the more complicated simulation formula and the higher processing time.

7 Resolution system

The resolution system is a quite simple console application handling the method description files. As mentioned above, each of the possible method is described in such a way, we can estimate the behavior of the method for any combination of the input variables. Each such description file resides in a directory named after the edit distance used in the approximate string matching NFA, and its version. Together, there are six such directories (three edit distances, each having two versions). Each directory contains number of the description files identified by the method abbreviation string, pattern type and the reduction flag. For DFA there ought to be also a set of files describing the minimized automata. The full complete set should consist of 86 files for each edit distance. However, the size of the current set is much smaller. Once we have determined which of the version (Σ or $\bar{p}$) is more effective for each

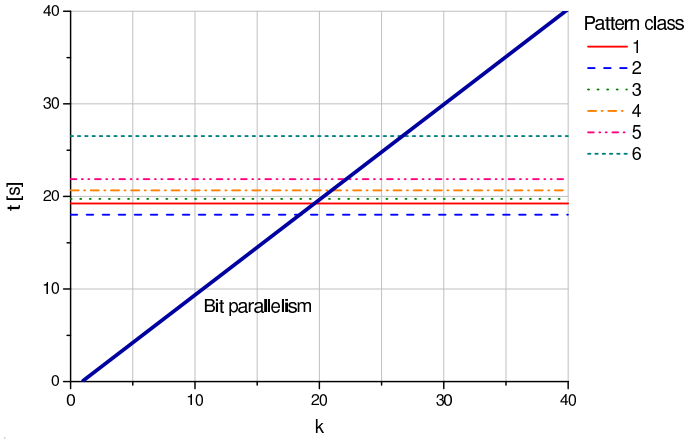


Fig. 29. Comparison of the processing time (t) for the simulation of NFA using dynamic programming for various pattern classes in dependence on the number of errors (k) for the approximate string matching using the Damerau distance

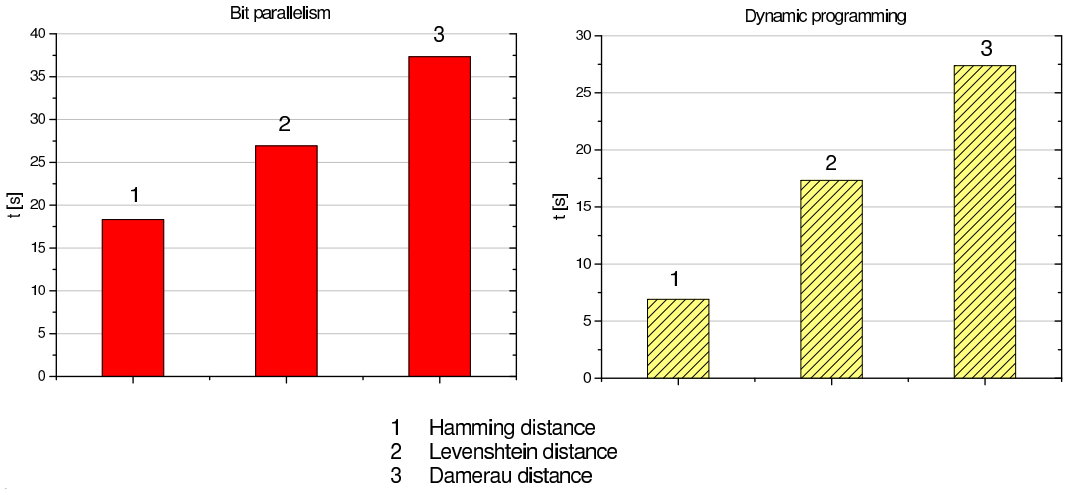


Fig. 30. Comparison of the time consumption for different simulation method running the simulation of various approximate string matching NFAs (Σ versions)

of the implemented method, the extensive measurements have been performed only for the more profitable version. The essential set of the description files consists of 29 files at the moment. However, it is not a problem to add the description files later at any moment. The resolution system is designed is such a way, it searches for all possible description files—if the one of the files is missing, it is skipped and treated as not currently available.

The application reads all the available description files and then produces output as shown in Tab. 3. The time (and state for DFA) estimations are computed and the method with the minimal estimated value is recommended. The "real" measurements can be performed afterwards and the estimated and elapsed values are matched together. The process of matching is performed by the program called simply `search` which encapsulates all of the implemented approximate string matching algorithms.

Since the description of the methods behavior fits only to the system where the measurement have been performed, we should match the speed of the actual system to our reference system somehow. This should be done with the provided setup program, which runs a synthetic benchmark procedure, widely know as Dhrystone [20]. Any actual system is benchmarked and all the time estimations are being corrected according to the ratio of performance of the actual system and the reference system.

```
Time estimations (after time adjustment)
=========================================================
[non-reduced NFA]
DFA,non-reduced NFA,p-version:not available
DP,non-reduced NFA,S/p-version:time=7.13397s
BP,non-reduced NFA,S-version:time=11.02s

Recommendations
=========================================================
DP,non-reduced NFA,S/p-version:time=7.13397s

Real test results
=========================================================
Elapsed time=7.34s
Estimated time=7.13397s
Deviation=2.80691%
```

Table 3. Sample output from the approximate string matching method resolution system

There may be a question of accuracy of the estimated results. The system is not intended to provide a precise results. The purpose of this project is to determine the basic features of each method used in the approximate string matching. The resolution system should be treated as a guide for choosing the appropriate string matching method for any particular matching task. No statistical evaluation of the results has been made to verify the system accuracy.

The precision of the results depends mainly on the actually used pattern. For the patterns and input files used for the reference measurements the result should be close enough to the estimation. However, for pattern with different structures, the result may be also different. We have specified the pattern classes in the beginning of this chapter. Any actual pattern may differ in size of the alphabet and the characteristic structure. Unfortunately, the performance of the alphabet sensitive methods is not affected only by the alphabet size. Therefore, we cannot adapt the estimated results to the actual alphabet size.

For the approximate string matching methods utilizing the NFA simulation the situation is simplified with the mostly linear dependence of the method behavior on the input variables. The estimated processing times are usually quite precise—we have experienced deviations up to 10%. For the deterministic automata the situation is quite different. The increase of the number of states as well as the increase of the construction time is mostly exponential with increasing number of errors k, and the length of the pattern m. For the range of input variables for which we have performed the measurements, and the data has been recorded in the description files, the estimations ought to be right. However, in the cases where there are not enough measured data available for precise estimation, the result may be significantly different. In the direction of increasing m with fixed k this is not the problem since we usually have a data set of good quality, so the estimation curve can be constructed with high precision. However, for the direction of increasing k we were not able to find suitable curve to fit the experimental results in most case. In such a case, the estimation is being made from last two measured values using simple linear equation.

Fig. 31 shows an example of time required for creation of DFA for pattern being a DNA sequence for number of errors $k = 2$ and an attempt to fit curve to these data. We can see a very significant fluctuation of time in dependance on the length of pattern. Such a behavior is nearly impossible to effectively describe. This especially happens for long patterns, where huge periodicity of pattern take place.

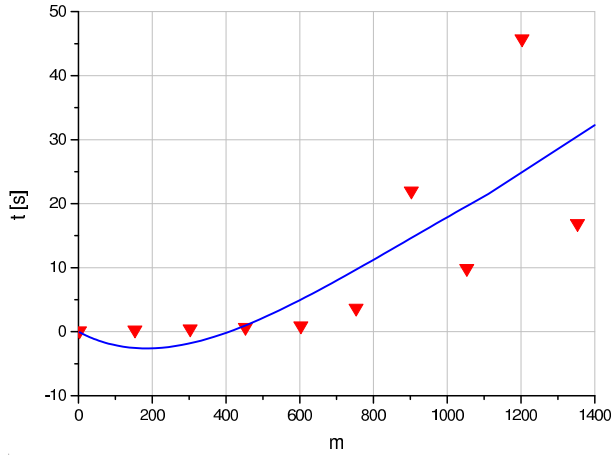


Fig. 31. Example of time dependence (t) on the length pattern (m) for pattern being a DNA sequence ($k = 2$)

Use of the deterministic automata, in general, is quite problematic. To fully understand the behavior of DFA for any given pattern, more extensive research should be carried on this subject. However, this research is beyond limits of this project.

8 Conclusions

We presented the results of our experiments with DFA and simulation of NFA for the exact and approximate pattern matching. They verify expectations leading from our theoretical work. However, there is one exception. We expected the use of reduced NFAs would be efficient. In practice they are efficient only when the number of allowed errors k is big with respect to the length of pattern m.

DFA size grows reasonable with increasing m and exponentially with increasing k. When DFA cannot fit in memory or for short texts, simulation methods apply. Comparing the simulation methods we see that dynamic programming is independent on k.

Based on the experimental results we designed a resolution system which for given configuration of the approximate string matching task decides which of the presented method would be most efficient. Since we found a sensitivity of the methods to the alphabet size and class of the text, for a practical application would be advisable to do more detailed classification of input texts appearing in the application and test these classes. Thus one can improve precision of the resolution system.

References

1. K. Abrahamson. Generalized string matching. *SIAM J. Comput.*, 16(6):1039–1051, 1987.
2. R. A. Baeza-Yates and G. H. Gonnet. A new approach to text searching. *Commun. ACM*, 35(10):74–82, 1992.
3. R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, 1977.
4. F. Damerau. A technique for computer detection and correction of spelling errors. *Commun. ACM*, 7(3):171–176, 1964.
5. B. Dömölki. An algorithm for syntactical analysis. *Computational Linguistics*, 3:29–46, 1964. Hungarian Academy of Science, Budapest.
6. Z. Galil and K. Park. An improved algorithm for approximate string matching. In G. Ausiello, M. Dezani-Ciancaglini, and S. Ronchi Della Rocca, editors, *Proceedings of the 16th International Colloquium on Automata, Languages and Programming*, number 372 in Lecture Notes in Computer Science, pages 394–404, Stresa, Italy, 1989. Springer-Verlag, Berlin.
7. R. W. Hamming. Error detecting and error correcting codes. *The Bell System Technical Journal*, 29(2):147–160, 1950.

8. J. Holub. Dynamic programming for reduced NFAs for approximate string and sequence matching. In J. Holub and M. Šimánek, editors, *Proceedings of the Prague Stringology Club Workshop '98*, pages 73–82, Czech Technical University, Prague, Czech Republic, 1998. Collaborative Report DC–98–06.
9. J. Holub. *Simulation of Nondeterministic Finite Automata in Pattern Matching.* Ph.D. Thesis, Czech Technical University, Prague, Czech Republic, February 2000.
10. J. Holub. Bit parallelism – NFA simulation. In B.W. Watson and D. Wood, editors, *Implementation and Application of Automata*, number 2494 in Lecture Notes in Computer Science, pages 149–160. Springer-Verlag, Heidelberg, 2002.
11. J. Holub. Dynamic programming – NFA simulation. In J.-M. Champarnaud and D. Maurel, editors, *Implementation and Application of Automata*, number 2608 in Lecture Notes in Computer Science, pages 295–300. Springer-Verlag, Heidelberg, 2003.
12. J. E. Hopcroft and J. D. Ullman. *Introduction to automata, languages and computations.* Addison-Wesley, Reading, MA, 1979.
13. R. N. Horspool. Practical fast searching in strings. *Softw. Pract. Exp.*, 10(6):501–506, 1980.
14. D. E. Knuth, J. H. Morris, Jr, and V. R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(1):323–350, 1977.
15. D. C. Kozen. *Automata and Computability.* Springer-Verlag, Berlin, 1997.
16. V. I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Sov. Phys. Dokl.*, 6:707–710, 1966.
17. P. H. Sellers. The theory and computation of evolutionary distances: Pattern recognition. *J. Algorithms*, 1(4):359–373, 1980.
18. R. K. Shyamasundar. A simple string matching algorithm. technical report, Tata Institute of Fundamental Research, India, 1976. 9 pages.
19. E. Ukkonen. Finding approximate patterns in strings. *J. Algorithms*, 6(1–3):132–137, 1985.
20. R. P. Weicker. Dhrystone: A synthetic systems programming benchmark. *Commun. ACM*, 27(10):1013–1030, 1984.
21. S. Wu and U. Manber. Fast text searching allowing errors. *Commun. ACM*, 35(10):83–91, 1992.

A Two-dimensional Online Tessellation Automata Approach to Two-dimensional Pattern Matching *

Tomáš Polcar and Bořivoj Melichar

Abstract

A new general approach to exact and approximate two-dimesional pattern matching is presented. This method, based on two-dimensional online tessellation automata, is a generalization of a well known one-dimensional pattern matching approach based on finite automata. It creates a nondeterministic two-dimensional online tessellation automaton and transforms it, by newly presented method, to a deterministic one. Then this automaton processes the input two-dimensional text and whenever it reaches a final state it reports an occurrence of the pattern. Since the text processing depends only on the size of the text, the searching phase of presented algorithms requires only $\mathcal{O}(n^2)$ time for the text of size (n, n). The only disadvantage of this method is the possibility of the exponential (in the size of the pattern) time of the preprocessing phase.

1 Introduction

Pattern matching in text is a well known theoretical problem in computer science. The expansion of multimedia requires an appropriate generalization of the pattern matching to higher dimensions. The first intuitive extension is the seeking of rectangular patterns in rectangular text. This leads to the two-dimensional pattern matching that can be also defined for non-rectangular patterns.

Two-dimensional exact and approximate pattern matching has many applications, especially in image processing. It is used for content based information retrieval from image databases, for image analysis, and for medical diagnostics. It is also used by some methods of detecting edges, where a set of 'edge detectors' is matched against the picture, and by some OCR systems.

Let us suppose squared pattern P of size (m, m), squared text T of size (n, n), and $\sigma = \min\left(|A|, m^2\right)$, where $|A|$ is the size of the alphabet A. The first linear time two-dimensional exact pattern matching algorithm, which takes $\mathcal{O}\left(\left(m^2 + n^2\right) \log \sigma\right)$ time, was introduced by Bird [Bir77] and, independently,

*This research has been partially supported by FRVŠ grant No. 2060/04, by CTU grant No. CTU0409213, and by the Ministry of Education, Youth, and Sports of the Czech Republic under research program No. J04/98:212300014.

by Baker [Bak78]. Using two-dimensional periodicity studied by Amir and Benson [AB92], Galil and Park [GP92] proposed the truly alphabet independent algorithm, which requires $\mathcal{O}\left(m^2+n^2\right)$ time. The best average case result is due to Baeza-Yates and Régnier [BYR93], who obtained $\mathcal{O}\left(n^2/m\right)$ time on average and $\mathcal{O}\left(n^2\right)$ time in the worst case. Two-dimensional online tessellation automata were foresaid by Inoue and Nakamura [IN77]. They also proved some basic properties of these acceptors. An algorithm for two-dimensional pattern matching using these automata, which simulates the Bird-Baker algorithm [Bir77, Bak78], was given by Toda, Inoue, Takanami [TIT83]. Slightly different notation, adopted from [GR97], will be used for two-dimensional online tessellation automata in this paper. The best result in the pattern matching with at most k substitutions (in this paper called Hamming distance defined in Definition 8) gave Amir and Landau [AL91] achieving $\mathcal{O}\left((k+\log\sigma)\,n^2\right)$ using $\mathcal{O}\left(n^2\right)$ space. The same problem was solved by Ranka and Heywood [RH91] using less space $\mathcal{O}\left(kn\right)$ but requiring $\mathcal{O}\left((k+m)\,n^2\right)$ time.

New two-dimensional pattern matching method presented in this paper is based on two-dimensional online tessellation automata. As these automata can be comprehended as a generalization of finite automata (well known from one-dimensional case), the presented method can be seen as a generalization of the one-dimensional pattern matching algorithm based on finite automata. The biggest advantage of this method is that the searching phase always requires only time linear with the size of the text $\mathcal{O}\left(n^2\right)$, and does not depend on the size of the pattern or the number of allowed errors. Unfortunately, there is no better than exponential estimation of the preprocessing phase time, which is $\mathcal{O}(\sigma\ 2^{3m^2})$ in case of exact 2D pattern matching and $\mathcal{O}(\sigma\ 2^{3k^2m^2})$ in case of 2D pattern matching with at most k substitutions.

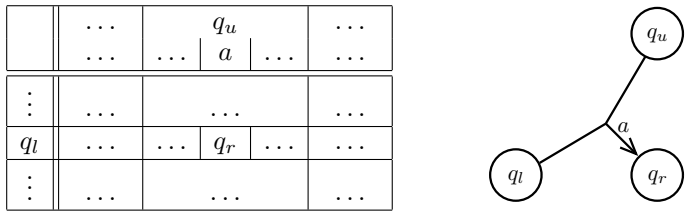
This paper is organized as follows. After the short introduction given in this section, Section 2 introduces basic definitions and terminology. After that, Section 3 presents new algorithm that transforms special type of two-dimensional online tessellation automata into equivalent deterministic two-dimensional online tessellation automata. Results from this section are then used in Section 4 that describes a new approach to the two-dimensional pattern matching using two-dimensional online tessellation automata. Finally, conclusions and outlines for future work are given in Section 5.

2 Basic notions and notations

Let A be a finite alphabet.

Definition 1 (picture) *A* picture (two-dimensional string) *over A is a two-dimensional rectangular array of elements of A. The* set of all pictures *over A is denoted by A^{**}. $A^{m,n}$ denotes the set of all pictures of size (m,n).* □

Given a picture u, the number of rows of u is denoted by $\mathrm{row}(u)$ and the number of columns of u is denoted by $\mathrm{col}(u)$. Let $1 \leq i \leq i' \leq \mathrm{row}(u)$,



a) b)

Figure 1: An example of a part of the transition table (at part a)) and the transition diagram (at part b)) describing $\delta(q_u, q_l, a) = q_r$

$1 \leq j \leq j' \leq \mathrm{col}(u)$. $u(i,j)$ denotes the symbol in u with coordinates (i,j), $u[(i,j),(i',j')]$ denotes the subarray of u of size $(i'-i+1, j'-j+1)$ starting at position (i,j).

Definition 2 (subpicture, prefix, suffix) *A picture $v \in A^{m,n}$ is said to be a* subpicture *of a picture $u \in A^{m',n'}$, $m \leq m'$, $n \leq n'$ if there exists (i,j) such that $v = u[(i,j),(i+m-1, j+n-1)]$. It is a* prefix *if $(i,j) = (1,1)$ and a* suffix *if $(i,j) = (m'-m+1, n'-n+1)$.* □

Since it will be necessary to identify the boundary of a picture u a picture $\hat{u}$ is defined as a picture of size $(\mathrm{row}(u)+2, \mathrm{col}(u)+2)$ obtained by surrounding u with a special boundary symbol $\# \notin A$. As $\hat{u}$ is a special picture for a special purpose, a different coordinate origin will be used for it. The coordinates of upper left corner of u are $(1,1)$ but of $\hat{u}$ are $(0,0)$. It ensures that $u(i,j) = \hat{u}(i,j)$ for all $1 \leq i \leq \mathrm{row}(u)$, $1 \leq j \leq \mathrm{col}(u)$.

Definition 3 (2D Online Tessellation Automaton) *A nondeterministic (deterministic) two-dimensional online tessellation automaton, which is referred as 2OTA (2DOTA), is a 5-tuple $\mathcal{A} = (A, Q, \delta, q_0, F)$ where:*

- *A is the input alphabet,*
- *Q is the finite set of states,*
- *$\delta : Q \times Q \times A \to \mathcal{P}(Q)$ ($\delta : Q \times Q \times A \to Q$) is the transition function,*
- *$q_0 \in Q$ is the initial state,*
- *$F \subseteq Q$ is the set of final states.* □

As well as transition function of finite automata, transition function of 2OTA can be described by several manners. The most common is a transition table and the most intuitive is a transition diagram. Let us note that the transition diagram can depict one state more than once. It helps to preserve clearness of the diagram. An example of both (transition table and transition diagram) is shown in Figure 1.

Definition 4 (Run of 2OTA) *A run of $\mathcal{A}$ on a picture $u \in A^{**}$ consists of associating a state (from the set Q) with each position (i,j) of u. Such state is given by the transition function δ and depends on the states already associated with positions $(i-1,j)$ and $(i,j-1)$ and on the symbol $u(i,j)$. Therefore, a run of $\mathcal{A}$ is a picture $r_{\mathcal{A}}(u)$ of size $(row(u), col(u))$ over the alphabet Q. The set of all possible runs of $\mathcal{A}$ on u is denoted by $\mathcal{R}_{\mathcal{A}}(u)$.*

In the beginning, the initial state q_0 is associated with all positions of the first row and of the first column of the picture $\hat{u}$. After that, the run consists of $row(u) + col(u) - 1$ steps. At each step l, one diagonal of picture $r_{\mathcal{A}}(u)$, which consists of such states that $i + j - 1 = l$, is computed. 2OTA $\mathcal{A}$ accepts (or recognizes) a picture u if there exists a run $r_{\mathcal{A}}(u) \in \mathcal{R}_{\mathcal{A}}(u)$ such that $r_{\mathcal{A}}(u)(row(u), col(u)) \in F$. A set of all pictures accepted by 2OTA $\mathcal{A}$ (language recognized by $\mathcal{A}$) is denoted by $\mathcal{L}(\mathcal{R}_{\mathcal{A}})$. The transition function is not necessarily a total function. In case it is a partial function, once the automaton cannot continue, the run ends and the picture is not accepted. □

Definition 5 (2OTA equivalency) *Two 2OTA $\mathcal{A}_1$ and $\mathcal{A}_2$ are equivalent if and only if they accept the same language, i.e. $\mathcal{L}(\mathcal{R}_{\mathcal{A}_1}) = \mathcal{L}(\mathcal{R}_{\mathcal{A}_2})$* □

Definition 6 (2OTA simulation) *Simulation of $\mathcal{A} = (A, Q, \delta, q_0, F)$ on a picture $u \in A^{**}$ is a picture $s_{\mathcal{A}}(u)$ of size $(row(u), col(u))$ over the alphabet $\mathcal{P}(Q)$. In the beginning, the initial set of states $\{q_0\}$ is associated with all positions of the first row and of the first column of $\hat{u}$. After that, a set of states at a position (i,j) is computed by the following expression:*

$$s_{\mathcal{A}}(u)(i,j) = \bigcup_{p \in s_{\mathcal{A}}(u)(i-1,j)} \bigcup_{q \in s_{\mathcal{A}}(u)(i,j-1)} \delta(p, q, u(i,j)).$$

Simulation accepts the picture u if $s_{\mathcal{A}}(u)(row(u), col(u)) \cap F \neq \emptyset$. A set of all pictures accepted by simulation of 2OTA $\mathcal{A}$ is denoted by $\mathcal{L}(s_{\mathcal{A}})$. □

Definition 7 (2OTA simulatability) *If a given 2OTA $\mathcal{A}$ satisfies*

$$\mathcal{L}(\mathcal{R}_{\mathcal{A}}) = \mathcal{L}(s_{\mathcal{A}}), \tag{1}$$

it is called simulatable two-dimensional online tesselation automaton. □

In order to use approximate pattern matching, it is necessary to define an error distance.

Definition 8 (2D Hamming distance) *A* Two-dimensional Hamming distance *between pictures u, v of the same size (m,n) is the number of coordinates (i,j), $1 \leq i \leq m$, $1 \leq j \leq n$ such that $u(i,j) \neq v(i,j)$. It will be denoted by $H(u,v)$.* □

3 2OTA to 2DOTA transformation

As was shown by Inoue and Nakamura in [IN77], 2OTA are more powerful than 2DOTA, because there is at least one 2OTA accepting language that is not recognizable by any 2DOTA. It means that it is not possible to create a universal algorithm for 2OTA to 2DOTA transformation.

Fortunately, this algorithm can be constructed for simulatable 2OTA by generalization of the subset construction, which is well know from one-dimensional case (presented e.g. in [Yu97] page 50).

Lemma 1 *Let $\mathcal{A} = (A, Q, \delta, q_0, F)$ be a simulatable 2OTA. Then a 2DOTA $\mathcal{A}' = (A, Q', \delta', q_0', F')$ where:*

- $Q' = \mathcal{P}(Q)$,
- $\delta'(p, q, a) = \bigcup_{r \in p} \bigcup_{s \in q} \delta(r, s, a)$ $\forall p, q \in Q'$, $\forall a \in A$,
- $q_0' = \{q_0\}$
- $F' = \{q \mid q \in Q', q \cap F \neq \emptyset\}$.

is a deterministic 2D online tessellation automaton equivalent to $\mathcal{A}$.

Proof 1 *Let us show by induction on (i, j) that $r_{\mathcal{A}'}(u)(i, j) = s_{\mathcal{A}}(u)(i, j)$. It simply holds for $(i, 0)$, $0 \leq i \leq row(u)$ and $(0, j)$, $0 \leq j \leq col(u)$, because $r_{\mathcal{A}'}(u)(i, 0) = s_{\mathcal{A}}(u)(i, 0) = r_{\mathcal{A}'}(u)(0, j) = s_{\mathcal{A}}(u)(0, j) = \{q_0\}$.*

Let us suppose that it also holds for coordinates $(i-1, j)$ and $(i, j-1)$, $1 \leq i \leq row(u)$, $1 \leq j \leq col(u)$. From the definition of the 2OTA simulation and δ', $s_{\mathcal{A}}(u)(i, j) = \bigcup_{p \in s_{\mathcal{A}}(u)(i-1,j)} \bigcup_{q \in s_{\mathcal{A}}(u)(i,j-1)} \delta(p, q, u(i, j))$ and $r_{\mathcal{A}'}(u)(i, j) = \bigcup_{p \in r_{\mathcal{A}'}(u)(i-1,j)} \bigcup_{q \in r_{\mathcal{A}'}(u)(i,j-1)} \delta(p, q, u(i, j))$. Thus $r_{\mathcal{A}'}(u)(i, j) = s_{\mathcal{A}}(u)(i, j)$.

The 2OTA simulation accepts u if and only if $s_{\mathcal{A}}(u)(row(u), col(u)) \cap F \neq \emptyset$, $F' = \{q \mid q \in Q', q \cap F \neq \emptyset\}$, and $r_{\mathcal{A}'}(u)(row(u), col(u)) = s_{\mathcal{A}}(u)(row(u), col(u))$. Thus $\mathcal{L}(\mathcal{R}_{\mathcal{A}'}) = \mathcal{L}(s_{\mathcal{A}})$.

Since $\mathcal{L}(s_{\mathcal{A}}) = \mathcal{L}(\mathcal{R}_{\mathcal{A}})$ ($\mathcal{A}$ is the simulatable 2OTA), $\mathcal{L}(\mathcal{R}_{\mathcal{A}'}) = \mathcal{L}(\mathcal{R}_{\mathcal{A}})$. □

Well, how to show that a particular 2OTA is the simulatable one? It is necessary to show, that it satisfies condition (1). Lemma 2 shows that one part of this condition is fulfilled by all 2OTA.

Lemma 2 *Given a 2OTA $\mathcal{A}$, $\mathcal{L}(\mathcal{R}_{\mathcal{A}}) \subseteq \mathcal{L}(s_{\mathcal{A}})$.*

Proof 2 *Consider a picture $u \in \mathcal{L}(\mathcal{R}_{\mathcal{A}})$. Let us show by induction on (i, j) that for any run $r_{\mathcal{A}} \in \mathcal{R}_{\mathcal{A}}$, $r_{\mathcal{A}}(u)(i, j) \in s_{\mathcal{A}}(u)(i, j)$, $0 \leq i \leq row(u)$, $0 \leq j \leq col(u)$. It simply holds for $(i, 0)$, $0 \leq i \leq row(u)$ and $(0, j)$, $0 \leq j \leq col(u)$ because $r_{\mathcal{A}}(u)(i, 0) = r_{\mathcal{A}}(u)(0, j) = q_0$ and $s_{\mathcal{A}}(u)(i, 0) = s_{\mathcal{A}}(u)(0, j) = \{q_0\}$.*

Let us suppose that it also holds for the coordinates $(i-1, j)$ and $(i, j-1)$, $1 \leq i \leq row(u)$, $1 \leq j \leq col(u)$. Then $r_{\mathcal{A}}(u)(i, j) \in s_{\mathcal{A}}(u)(i, j)$ because $s_{\mathcal{A}}(u)(i, j) = \bigcup_{p \in s_{\mathcal{A}}(u)(i-1,j)} \bigcup_{q \in s_{\mathcal{A}}(u)(i,j-1)} \delta(p, q, u(i, j))$.

Since $r_{\mathcal{A}}(u)(row(u), col(u)) \in s_{\mathcal{A}}(u)(row(u), col(u))$, $u \in \mathcal{L}(s_{\mathcal{A}})$. □

As it could be quite difficult to prove the remainder, following sufficient condition, which is easier to prove, was defined.

Lemma 3 *Let $\mathcal{A} = (A, Q, \delta, q_0, F)$ be a 2OTA that satisfies*

$$\begin{array}{l} \forall u \in A^{**}, \forall (i,j), 1 \leq i \leq row(u), 1 \leq j \leq col(u), \\ \forall o \in s_{\mathcal{A}}(u)(i-1,j), \forall p \in s_{\mathcal{A}}(u)(i,j-1), \\ either\ \exists\ r_{\mathcal{A}}(u) \in \mathcal{R}_{\mathcal{A}}(u),\ o = r_{\mathcal{A}}(u)(i-1,j), p = r_{\mathcal{A}}(u)(i,j-1), \\ or\ \delta(o,p,u(i,j)) = \emptyset. \end{array} \tag{2}$$

Then $\mathcal{A}$ is the simulatable 2OTA.

Proof 3 *It is enough to show that $\mathcal{L}(\mathcal{R}_{\mathcal{A}}) \supseteq \mathcal{L}(s_{\mathcal{A}})$. First, let as show that if a given 2OTA $\mathcal{A}$ satisfies*

$$\begin{array}{l} \forall u \in A^{**}, \forall (i,j), 0 \leq i \leq row(u), 0 \leq j \leq col(u), \\ \forall q \in s_{\mathcal{A}}(u)(i,j), \exists\ r_{\mathcal{A}}(u) \in \mathcal{R}_{\mathcal{A}}(u),\ q = r_{\mathcal{A}}(u)(i,j), \end{array} \tag{3}$$

$\mathcal{L}(\mathcal{R}_{\mathcal{A}}) \supseteq \mathcal{L}(s_{\mathcal{A}})$. Since the condition is satisfied for all (i,j) it is also satisfied for $(row(u), col(u))$. Thus when the picture u is accepted by the simulation of $\mathcal{A}$, it also exists an accepting run of $\mathcal{A}$.

Now, it suffice to prove that condition (2) implies condition (3). Let us suppose that $\mathcal{A}$ satisfies condition (2) but it does not satisfy condition (3). It means there is a picture u, (i,j), $1 \leq i \leq row(u)$, $1 \leq j \leq col(u)$, and $q \in s_{\mathcal{A}}(u)(i,j)$, such that for all runs $r_{\mathcal{A}}(u) \in \mathcal{R}_{\mathcal{A}}(u)$, $r_{\mathcal{A}}(u)(i,j) \neq q$. It means that q is not accessible by any run of $\mathcal{A}$, but it is accessible by the simulation of $\mathcal{A}$. The only possibility how q can get into $s_{\mathcal{A}}(u)(i,j)$ is that there are $o \in s_{\mathcal{A}}(u)(i-1,j)$, $p \in s_{\mathcal{A}}(u)(i,j-1)$, $\delta(o,p,u(i,j)) \neq \emptyset$, $q \in \delta(o,p,u(i,j))$. Since q at position (i,j) is not accessible by any run of $\mathcal{A}$, o and p were not accessible by the same run of $\mathcal{A}$ (i.e. $\nexists\ r_{\mathcal{A}}(u) \in \mathcal{R}_{\mathcal{A}}(u),\ o = r_{\mathcal{A}}(u)(i-1,j), p = r_{\mathcal{A}}(u)(i,j-1)$). Otherwise q would be also accessible by this run because $q \in \delta(o,p,u(i,j))$. This is the contradiction with fact that $\mathcal{A}$ satisfies condition (2) and the proof is complete. □

What is the space and time complexity of this generalized subset construction algorithm? Let $\mathcal{A}_N = (A, Q_N, \delta_N, q_0^N, F_N)$ be a simulatable 2OTA and $\mathcal{A}_D = (A, Q_D, \delta_D, q_0^D, F_D)$ be equivalent 2DOTA. Since states of $\mathcal{A}_D$ are created as the sets of states of $\mathcal{A}_N$, $|Q_D| = \mathcal{O}(2^{|Q_N|})$. The total number of transitions is $\mathcal{O}(|A||Q_D|^2) = \mathcal{O}(|A|2^{2|Q_N|})$, because δ function is defined on $Q \times Q \times A$. Moreover, one state of $\mathcal{A}_D$ is formed by at most $|Q_N|$ states of $\mathcal{A}_N$. Thus the total time required by $\mathcal{A}_D$ construction is $\mathcal{O}(|A||Q_N||Q_D|^2) \leq \mathcal{O}(|A||Q_D|^3) = \mathcal{O}(|A|2^{3|Q_N|})$.

The space required by resulting automaton $\mathcal{A}_D$ is equal to the number of its states plus the number of its transitions. Thus it requires $\mathcal{O}(2^{|Q_N|} + |A|2^{2|Q_N|}) = \mathcal{O}(|A|2^{2|Q_N|})$ space.

Algorithm 1 Two-dimensional pattern matching by 2OTA.

Input: Pattern P, text T, type of pattern matching problem *type*.
Output: Coordinates of lower rights corners of all occurrences of P.

1: Create simulatable 2OTA $\mathcal{A}_N$ for pattern matching problem *type* and given pattern P as described in Section 4.1 or 4.2.

2: Transform 2OTA $\mathcal{A}_N$ to 2DOTA $\mathcal{A}_D$.

3: Run A_D on T.

4: Output = all positions (i, j) associated with final state of A_D.

4 Pattern matching by 2OTA

Now it is possible to generalize one-dimensional pattern matching algorithm, which was described by Melichar and Holub in [MH97], into two dimensions as it is described in Algorithm 1. This algorithm works in two independent phases. Preprocessing phase creates deterministic two-dimensional pattern matching automaton, for given pattern and pattern matching problem. This phase depends on the transformation of 2OTA to 2DOTA described in previous Section 3 and on the pattern matching automata construction, which is described in Section 4.1 and 4.2. Processing phase then runs the automaton created by preprocessing phase on the input text.

What is the space and time complexity of this algorithm? Since it works in two independent phases, it is possible to analyze them separately. As the construction of nondeterministic automaton is quite faster than its transformation into deterministic one, the time of the preprocessing phase is given by the time of the 2OTA to 2DOTA transformation, which was analyzed in Section 3. Processing phase requires always linear time with the size of the input text.

The extra space required by this algorithm is given by the space required by the pattern matching automaton (analyzed in Section 3), and by space required for automaton run, which is $\mathcal{O}(n)$ for the input text of size (n, n).

4.1 Exact two-dimensional pattern matching

Let us create a 2OTA for exact two-dimensional pattern matching. In order to do that, let us generalize the pattern matching automata from the one-dimensional case. Therefore, a 2OTA for matching pattern P will be a 2OTA accepting all pictures with P as its suffix. Such automaton is created by Algorithm 2.

Let us show the analogy to the one-dimensional case (described in [MH97]). $q_0 \in \delta(q_0, q_0, x)$ for all $x \in A$ represents the self loop at the initial state. The rest of δ function accepts exactly the picture P. The only nondeterminism of the δ function is created in step 5 of Algorithm 2. The key question is, whether this automaton is the simulatable one.

Algorithm 2 Construction of 2OTA for two-dimensional exact pattern matching.

Input: A pattern P.
Output: A 2OTA $\mathcal{A}$ accepting all pictures with P as its suffix.

1: $m = \text{row}(P)$, $n = \text{col}(P)$,
2: $Q = \{q_0\} \cup \{q_{i,j} \mid 1 \leq i \leq m, 1 \leq j \leq n\}$,
3: $F = \{q_{m,n}\}$,
4: $\delta(q_0, q_0, x) = \{q_0\}$ for $x \in A$, $x \neq P(1,1)$
5: $\delta(q_0, q_0, P(1,1)) = \{q_0, q_{1,1}\}$,
6: $\delta(q_{i-1,1}, q_0, P(i,1)) = \{q_{i,1}\}$ for $2 \leq i \leq m$,
7: $\delta(q_0, q_{1,j-1}, P(1,j)) = \{q_{1,j}\}$ for $2 \leq j \leq n$,
8: $\delta(q_{i-1,j}, q_{i,j-1}, P(i,j)) = \{q_{i,j}\}$ for $2 \leq i \leq m, 2 \leq j \leq n$,
9: $\mathcal{A} = (A, Q, \delta, q_0, F)$.

Lemma 4 *A 2OTA $\mathcal{A} = (A, Q, \delta, q_0, F)$ for exact pattern matching of pattern P created by Algorithm 2 is a simulatable 2OTA.*

Proof 4 *As was shown in Lemma 3 it is enough to show that it satisfies condition 2. In other words, it is necessary to show that for all pictures $u \in A^{**}$ and for all $o \in s_{\mathcal{A}}(u)(i-1,j)$, $p \in s_{\mathcal{A}}(u)(i,j-1)$, $1 \leq i \leq row(u)$, $1 \leq j \leq col(u)$ it is satisfied that o and p are either accessible by the same run of $\mathcal{A}$ ($\exists\, r_{\mathcal{A}}(u) \in \mathcal{R}_{\mathcal{A}}(u)$, $o = r_{\mathcal{A}}(u)(i-1,j), p = r_{\mathcal{A}}(u)(i,j-1)$) or $\delta(o,p,u(i,j)) = \emptyset$.*

Let $o = q_{i',j'}$ and $p = q_{i'',j''}$. From the definition of δ function in Algorithm 2, $\delta(q_{i',j'}, q_{i'',j''}, a) \neq \emptyset$ only if $i'' = i' + 1$, $j' = j'' + 1$, and $a = P(i'', j')$. Since there is a run $r_{\mathcal{A}}(u) \in \mathcal{R}_{\mathcal{A}}(u)$ such that $r_{\mathcal{A}}(u)(i-1,j) = q_{i',j'}$ only if $P[(1,1),(i',j')]$ is the suffix of $u[(1,1),(i-1,j)]$, $r_{\mathcal{A}}(i,j-1) = q_{i'',j''}$ only if $P[(1,1),(i'',j'')]$ is the suffix of $u[(1,1),(i,j-1)]$, and $a = P(i'',j')$, it holds that $P[(1,1),(i'',j')]$ is the suffix of $u[(1,1),(i,j)]$ and the condition holds. Since q_0 represents the empty prefix, the condition also holds for $o = q_0$ or $p = q_0$. □

And what is the number of states of 2OTA $\mathcal{A}$ created by Algorithm 2? It creates one state for each prefix of the pattern, so the number of states of $\mathcal{A}$ is linear with the size of the pattern – m^2 for the pattern of size (m,m). An example of the exact pattern matching is shown in Example 1.

Example 1 *Let us create a deterministic two-dimensional online tessellation automaton for the exact matching of a picture* $\boxed{\begin{smallmatrix} a & b \\ b & a \end{smallmatrix}}$. *At first, a nondeterministic 2OTA is created using Algorithm 2. Its δ function of this automaton is described in Figures 2 and 3. After that, this automaton is transformed by the subset construction into a deterministic one (its transition table is shown in Figure 4). Once the 2DOTA is created, it can be used for the pattern matching,*

	0			1, 1			1, 2			2, 1			2, 2		
	a	b	x	a	b	x	a	b	x	a	b	x	a	b	x
0	0; 1, 1	0	0		2, 1										
1, 1		1, 2													
1, 2															
2, 1							2, 2								
2, 2															

Figure 2: The δ function (described by the transition table) of 2OTA for exact pattern matching from Example 1 (As the state names are in form $q_{i,j}$, only subscripts of the state names are shown in this table.)

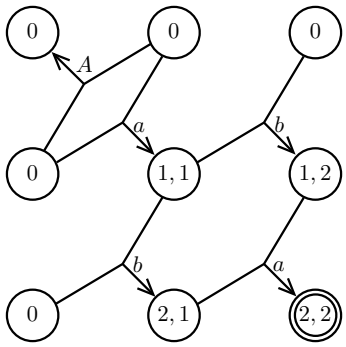


Figure 3: The δ function (described by the transition diagram) of 2OTA for exact pattern matching from Example 1 (As the state names are in form $q_{i,j}$, only subscripts of the state names are shown in this diagram.)

e.g. in a picture shown in Figure 5 a). Run of the 2DOTA over this picture and found occurrences are shown in Figure 5 b). □

4.2 Two-dimensional pattern matching using Hamming distance

Basic principles presented in Section 4.1 can be used to create an online tessellation automaton for two-dimensional pattern matching using Hamming distance.

Each state of the exact pattern matching automaton represents a particular prefix of searched pattern. Hamming pattern matching automaton requires an additional information about a number of errors occurred in certain prefix. However, it is still not enough. The issue is in fact, that it is not possible to compute the number of errors of prefix with coordinates (i, j) from the number of errors of prefixes with coordinates $(i-1, j)$ and $(i, j-1)$, because it is necessary to know the number of errors common for both prefixes. Thus, all states but initial one of a pattern matching 2OTA using Hamming distance will be formed by 4-tuple (i, j, α, β), where i and j represents the size of found prefix (as in exact case), α is the number of errors in a current column of a prefix, and β is the number of all errors in a prefix. Thus the transition function computes new

	0			0; 1, 1			0; 1, 1; 2, 2		
	a	b	x	a	b	x	a	b	x
0	0; 1, 1	0	0	0; 1, 1	0; 2, 1	0	0; 1, 1	0; 2, 1	0
0; 1, 1	0; 1, 1	0; 1, 2	0	0; 1, 1	0; 1, 2; 2, 1	0	0; 1, 1	0; 1, 2; 2, 1	0
0; 1, 1; 2, 2	0; 1, 1	0; 1, 2	0	0; 1, 1	0; 1, 2; 2, 1	0	0; 1, 1	0; 1, 2; 2, 1	0
0; 1, 2	0; 1, 1	0	0	0; 1, 1	0; 2, 1	0	0; 1, 1	0; 2, 1	0
0; 1, 2; 2, 1	0; 1, 1	0	0	0; 1, 1	0; 2, 1	0	0; 1, 1	0; 2, 1	0
0; 2, 1	0; 1, 1	0	0	0; 1, 1	0; 2, 1	0	0; 1, 1	0; 2, 1	0

	0; 1, 2			0; 1, 2; 2, 1			0; 2, 1		
	a	b	x	a	b	x	a	b	x
0	0; 1, 1	0	0	0; 1, 1	0	0	0; 1, 1	0	0
0; 1, 1	0; 1, 1	0; 1, 2	0	0; 1, 1	0; 1, 2	0	0; 1, 1	0; 1, 2	0
0; 1, 1; 2, 2	0; 1, 1	0; 1, 2	0	0; 1, 1	0; 1, 2	0	0; 1, 1	0; 1, 2	0
0; 1, 2	0; 1, 1	0	0	0; 1, 1	0	0	0; 1, 1	0	0
0; 1, 2; 2, 1	0; 1, 1; 2, 2	0	0	0; 1, 1; 2, 2	0	0	0; 1, 1	0	0
0; 2, 1	0; 1, 1; 2, 2	0	0	0; 1, 1; 2, 2	0	0	0; 1, 1	0	0

Figure 4: The δ function of 2DOTA for exact pattern matching from Example 1 (As the state names are in form $q_{i,j}$, only subscripts of the state names are shown in this table.)

a	b	x	a	a	b	b
a	b	a	b	x	a	b
x	b	b	a	b	a	x
b	b	x	b	a	a	b
a	a	b	b	a	x	a

a)

0; 1, 1	0; 1, 2	0	0; 1, 1	0; 1, 1	0; 1, 2	0
0; 1, 1	0; 1, 2	0; 1, 1	0; 1, 2; 2, 1	0	0; 1, 1	0; 1, 2
0	0	0; 2, 1	0; 1, 1; 2, 2	0; 1, 2	0; 1, 1	0
0	0	0	0; 2, 1	0; 1, 1; 2, 2	0; 1, 1	0; 1, 2
0; 1, 1	0; 1, 1	0; 1, 2	0	0; 1, 1	0	0; 1, 1

b)

Figure 5: a) Input picture for pattern matching from Example 1 (squares denotes pattern occurrences) b) Run of the 2DOTA from Example 1 over the picture from a) (As the state names are in form $q_{i,j}$, only subscripts of the state names are shown in this table. Squares denotes found pattern occurrences - the only final state is $0; 1, 1; 2, 2$.)

state as follows:

$$\delta(q_{i-1,j,\alpha,\beta}, q_{i,j-1,\alpha',\beta'}, a) = \begin{cases} q_{i,j,\alpha,\beta'+\alpha} & \text{when } a = P(i,j), \\ q_{i,j,\alpha+1,\beta'+\alpha+1} & \text{otherwise.} \end{cases}$$

Such 2OTA is created by Algorithm 3. It is possible to see, that the only nondeterminism of the δ function, created by steps 4 and 5 of Algorithm 3, is again caused by the self loop at the initial state.

Lemma 5 *A 2OTA $\mathcal{A} = (A, Q, \delta, q_0, F)$ for pattern matching using Hamming distance of pattern P created by Algorithm 3 is a simulatable 2OTA.*

Proof 5 *This proof is nearly the same as the proof of Lemma 4. The only difference is that the δ function is little bit more complicated than in the exact case.*

Algorithm 3 Construction of 2OTA for two-dimensional pattern matching using Hamming distance.

Input: A pattern P, a number of allowed errors k.
Output: A 2OTA $\mathcal{A}$ accepting all pictures u that exists a picture v, $H(u,v) \leq k$, v is the suffix of P.

1: $m = \mathrm{row}(P)$, $n = \mathrm{col}(P)$,

2: $Q = \{q_0\} \cup \{q_{i,j,\alpha,\beta} \mid 1 \leq i \leq m, 1 \leq j \leq n, 0 \leq \beta \leq \min(i \cdot j, k), 0 \leq \alpha \leq \min(i, \beta)\}$,

3: $F = \{q_{m,n,\alpha,\beta} \mid 0 \leq \beta \leq k, 0 \leq \alpha \leq \min(m, \beta)\}$,

4: $\delta(q_0, q_0, P(1,1)) = \{q_0, q_{1,1,0,0}\}$,

5: $\delta(q_0, q_0, a) = \{q_0, q_{1,1,1,1}\}$ for $a \in A$, $a \neq P(1,1)$,

6: $\delta(q_{i-1,1,\alpha,\alpha}, q_0, P(i,1)) = \{q_{i,1,\alpha,\alpha}\}$ for $2 \leq i \leq m$, $0 \leq \alpha \leq \min(i-1, k)$,

7: $\delta(q_{i-1,1,\alpha,\alpha}, q_0, a) = \{q_{i,1,\alpha+1,\alpha+1}\}$ for $a \in A$, $a \neq P(i,1)$, $2 \leq i \leq m$, $0 \leq \alpha \leq \min(i-1, k-1)$,

8: $\delta(q_0, q_{1,j-1,\alpha,\beta}, P(1,j)) = \{q_{1,j,0,\beta}\}$ for $2 \leq j \leq n$, $0 \leq \beta \leq \min(j-1, k)$, $0 \leq \alpha \leq \min(1, \beta)$,

9: $\delta(q_0, q_{1,j-1,\alpha,\beta}, a) = \{q_{1,j,1,\beta+1}\}$ for $a \in A$, $a \neq P(1,j)$, $2 \leq j \leq n$, $0 \leq \beta \leq \min(j-1, k-1)$, $0 \leq \alpha \leq \min(1, \beta)$,

10: $\delta(q_{i-1,j,\alpha,\beta}, q_{i,j-1,\alpha',\beta'}, P(i,j)) = \{q_{i,j,\alpha,\beta'+\alpha}\}$ for $2 \leq i \leq m, 2 \leq j \leq n$, $0 \leq \beta \leq \min((i-1)j, k)$, $0 \leq \alpha \leq \min(i-1, \beta)$, $0 \leq \beta' \leq \min(i(j-1), k-\alpha)$, $0 \leq \alpha' \leq \min(i, \beta')$,

11: $\delta(q_{i-1,j,\alpha,\beta}, q_{i,j-1,\alpha',\beta'}, a) = \{q_{i,j,\alpha+1,\beta'+\alpha+1}\}$ for $a \in A$, $a \neq P(i,j)$, $2 \leq i \leq m, 2 \leq j \leq n$, $0 \leq \beta \leq \min((i-1)j, k-1)$, $0 \leq \alpha \leq \min(i-1, \beta)$, $0 \leq \beta' \leq \min(i(j-1), k-\alpha-1)$, $0 \leq \alpha' \leq \min(i, \beta')$,

12: $\mathcal{A} = (A, Q, \delta, q_0, F)$.

Let $o = q_{i',j',\alpha',\beta'}$ and $p = q_{i'',j'',\alpha'',\beta''}$. From the definition of δ function in Algorithm 3, $\delta(q_{i',j',\alpha',\beta'}, q_{i'',j'',\alpha'',\beta''}, a) \neq \emptyset$ only if $i'' = i'+1$, $j' = j''+1$, and either $a = P(i'', j')$ and $\beta''+\alpha' \leq k$ or $\beta''+\alpha'+1 \leq k$. $r_{\mathcal{A}}(u)(i-1, j) = q_{i',j',\alpha',\beta'}$ only if $P[(1,1),(i',j')]$ is the suffix of $u[(1,1),(i-1,j)]$, with the number of errors α' in the j'-th column and β' in total. $r_{\mathcal{A}}(i, j-1) = q_{i'',j'',\alpha'',\beta''}$ only if $P[(1,1),(i'',j'')]$ is the suffix of $u[(1,1),(i,j-1)]$, with the number of errors α'' in the j''-th column and β'' in total. Thus it holds that $P[(1,1),(i'',j')]$ is the suffix of $u[(1,1),(i,j)]$ with either $\beta'' + \alpha' \leq k$ errors if $a = P(i'', j')$ or $\beta'' + \alpha' + 1 \leq k$ errors if $a \neq P(i'', j')$ and the condition holds. Since q_0 represents the empty prefix, the condition also holds for $o = q_0$ or $p = q_0$. □

And what is the number of states of 2OTA $\mathcal{A}$ created by Algorithm 3? It creates at most k^2 states for each prefix of the pattern, where k is the number of allowed errors. So, the number of states of $\mathcal{A}$ is $\mathcal{O}(k^2m^2)$ for the pattern of

	0			1, 1, 0, 0			1, 1, 1, 1		
	a	b	x	a	b	x	a	b	x
0	0 1, 1, 0, 0	0 1, 1, 1, 1	0 1, 1, 1, 1	2, 1, 1, 1	2, 1, 0, 0	2, 1, 1, 1		2, 1, 1, 1	
1, 1, 0, 0	1, 2, 1, 1	1, 2, 0, 0	1, 2, 1, 1						
1, 1, 1, 1		1, 2, 0, 1							

	1, 2, 0, 0			1, 2, 0, 1			1, 2, 1, 1		
	a	b	x	a	b	x	a	b	x
2, 1, 0, 0	2, 2, 0, 0	2, 2, 1, 1	2, 2, 1, 1	2, 2, 0, 0			2, 2, 1, 1		
2, 1, 1, 1	2, 2, 0, 1			2, 2, 0, 1					

Figure 6: The δ function (described by the transition table) of 2OTA for pattern matching using Hamming distance from Example 2 (As the state names are in form $q_{i,j,\alpha,\beta}$, only subscripts of the state names are shown in this table. Moreover, only non-empty rows and columns are shown here.)

size (m, m). An example of the pattern matching using Hamming distance is shown in Example 2.

Example 2 *Let us create a deterministic two-dimensional online tessellation automaton for the matching of the picture* $\boxed{\begin{smallmatrix} a & b \\ b & a \end{smallmatrix}}$ *using Hamming distance with at most one error allowed. At first, a nondeterministic 2OTA is created using Algorithm 3. The δ function of this automaton is described in Figures 6 and 7.*

After that, this automaton is transformed into a deterministic one. Once the 2DOTA is created, it can be used for the pattern matching, e.g. in a picture shown in Figure 8 a) (it is the same picture as was used in Example 1). Run of the 2DOTA over this picture and found occurrences are shown in Figure 8 b). □

5 Conclusion

A new algorithm that transforms some special types of nondeterministic two-dimensional online tessellation automata to deterministic ones was presented in this paper. This algorithm was built as a generalization of the subset construction well known from the one-dimensional case [Yu97]. Moreover, this algorithm was used to build a completely new algorithm for two-dimensional pattern matching – especially for exact two-dimensional pattern matching and two-dimensional pattern matching using Hamming distance. This new algorithm is also a generalization of the one-dimensional case. It creates an appropriate nondeterministic two-dimensional online tessellation automaton for given pattern and pattern matching problem, transforms it to a deterministic version a performs the pattern matching.

The biggest advantage of this approach is the fact that the matching phase time complexity is always linear with the size of the text – $\mathcal{O}(n^2)$ for the text of size (n, n). It means that the matching phase is always very fast and its speed does not depend on the type of the pattern matching problem.

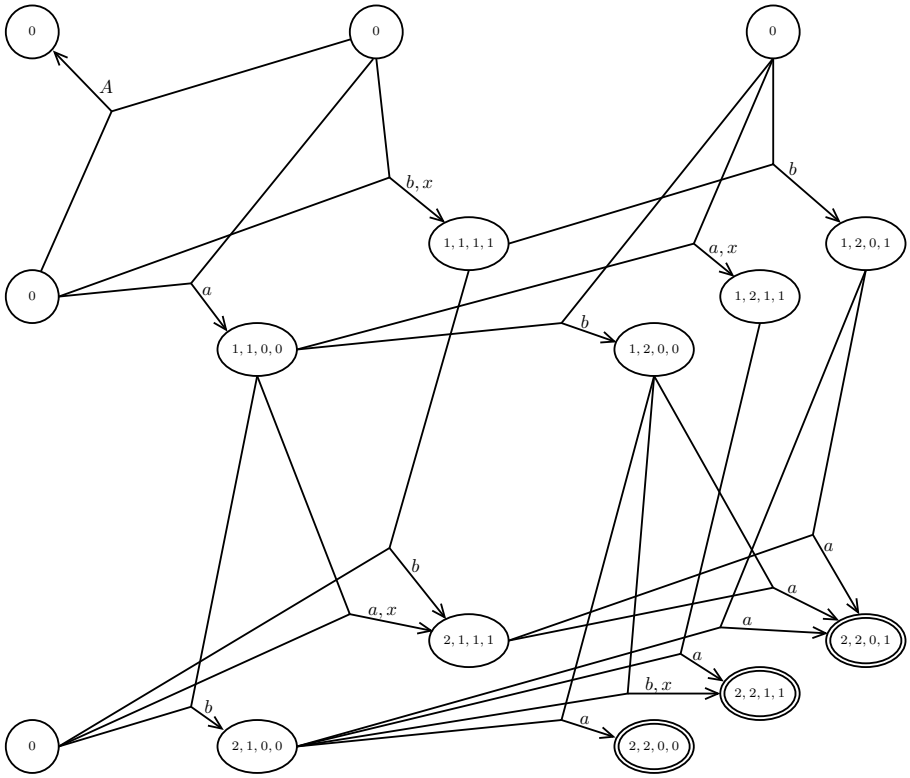


Figure 7: The δ function (described by the transition diagram) of 2OTA for pattern matching using Hamming distance from Example 2 (As the state names are in form $q_{i,j,\alpha,\beta}$, only subscripts of the state names are shown in this diagram.)

Moreover, the time complexity of the preprocessing phase depends only on the size of the pattern, size of the alphabet, the number of allowed errors, and the type of used error distance. Disadvantage is that there is not better than exponential (with the size of the pattern) estimation of the time complexity of the preprocessing phase. For pattern of size (m, m) it is $\mathcal{O}(|A|2^{3m^2})$ in case of exact pattern matching and $\mathcal{O}(|A|2^{3k^2m^2})$ in case of pattern matching using Hamming distance, where $|A|$ is the size of the alphabet, and k is the number of allowed errors. Thus this pattern matching algorithm is very useful in case that the size of the text is much greater than the size of the pattern ($m \ll n$), or when the pattern is searched in many input texts.

The other advantage of this method is, that the input picture can be read row by row or column by column. It means that it is not necessary to have the whole input picture loaded in the memory. Thus, the space required by the presented algorithm is $\mathcal{O}(|A|2^{2m^2} + n)$ in case of exact pattern matching, and $\mathcal{O}(|A|2^{2k^2m^2} + n)$ in case of pattern matching using Hamming distance.

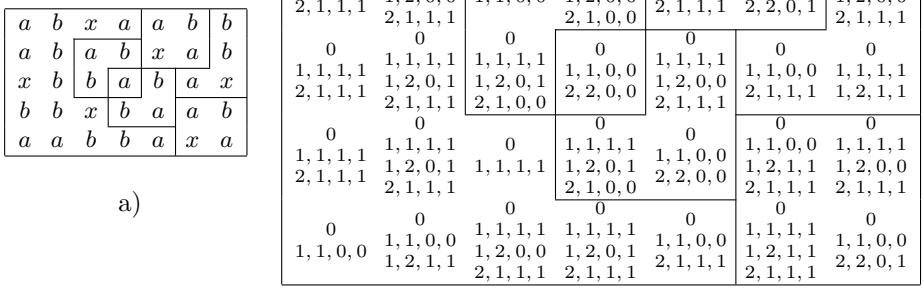


Figure 8: a) Input picture for pattern matching from Example 2 (squares denotes pattern occurrences) b) Run of the 2DOTA from Example 2 over the picture from a) (As the state names are in form $q_{i,j,\alpha,\beta}$, only subscripts of the state names are shown in this table. Squares denotes found pattern occurrences.)

Future research should extend presented principle for other types of two-dimensional pattern matching problems – e.g. two-dimensional approximate pattern matching using KS [KS87], R, RC, and L [BY98] distances including matching of a finite set of patterns. Moreover it might improve estimations of the space and time complexities of the preprocessing phase of the presented algorithm.

References

[AB92] A. Amir and G. Benson. Two-dimensional periodicity and its applications. In *Proceedings of the 3rd ACM-SIAM Annual Symposium on Discrete Algorithms*, pages 440–452, 1992.

[AL91] A. Amir and G. Landau. Fast parallel and serial multidimensional approximate array matching. *Theoretical Computer Science*, 81:97–115, 1991.

[Bak78] T. Baker. A techique for extending rapid exact string matching to arrays of more than one dimension. *SIAM J. Comput.*, 6:533–541, 1978.

[Bir77] R. S. Bird. Two dimensional pattern matching. *Information Processing Letters*, 6(5):168–170, October 1977.

[BY98] Ricardo A. Baeza-Yates. Similarity in two-dimensional strings. In *Annual International Conference on Computing and Combinatorics*, LNCS 1449, pages 319–328. Springer-Verlag, Berlin, August 1998.

[BYR93] R. Baeza-Yates and M. Régnier. Fast two dimensional pattern matching. *Inf. Process. Lett.*, 45:51–57, 1993.

[GP92] Z. Galil and K. Park. Truly alphabet-independent two-dimensional pattern matching. In *Proceedings of the 33rd IEEE Annual Symposium on Foundations of Computer Science*, pages 247–256, 1992.

[GR97] D. Giammarresi and A. Restivo. Two-dimensional languages. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 3 Beyond Words, pages 215–267. Springer-Verlag, Berlin, 1997.

[IN77] K. Inoue and A. Nakamura. Some properties of two-dimensional on-line tessellation acceptors. *Information Sciences*, 13:95–121, 1977.

[KS87] K. Krithivasan and R. Sitalakshmi. Efficient two-dimensional pattern matching in the presence of errors. *Information Sciences*, 43:169–184, 1987.

[MH97] B. Melichar and J. Holub. 6D classification of pattern matching problems. In J. Holub, editor, *Proceedings of the Prague Stringology Club Workshop '97*, pages 24–32, 1997.

[RH91] S. Ranka and T. Heywood. Two–dimensional pattern matching with k mismatches. *Pattern Recognition*, 24(1):31–40, 1991.

[TIT83] M. Toda, K. Inoue, and I. Takanami. Two-dimensional pattern matching by two-dimensional on-line tesselation acceptors. *Theoretical Computer Science*, 24:179–194, 1983.

[Yu97] S. Yu. Regular languages. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 1 Word Language Grammar, pages 41–110. Springer-Verlag, Berlin, 1997.

Compiling Contextual Restrictions on Strings into Finite-State Automata

Anssi Yli-Jyrä and Kimmo Koskenniemi
Department of General Linguistics
P.O. Box 9, FIN-00014 University of Helsinki, Finland
firstname.lastname@helsinki.fi

Abstract

The paper discusses a language operation that we call *context restriction*. This operation is closely associated with *context restriction rules* (Koskenniemi, 1983; Kiraz, 2000), *right-arrow rules* or *implication rules* (Koskenniemi et al., 1992; Voutilainen, 1997) and the *restriction operator* (Beesley and Karttunen, 2003). The operation has been used in finite-state phonology and morphology in certain limited ways. A more general setting involves restricting overlapping occurrences of a center language under context conditions. Recently, star-free regular languages (and all regular languages) have been shown to be closed under context restrictions with such "overlapping centers" (Yli-Jyrä, 2003), but the construction involved is overly complex and becomes impractical when the number of operands grows.

In this paper, we improve this recent result by presenting a more practical construction. This construction is not only simpler but it also leads to a generalization where contexts and centers may appear as conditions at both sides of the implication arrow ($\Rightarrow$): licensing conditions on the right-hand side specify the restriction and triggering conditions on the left-hand side regulate activation of the restriction. One application of the generalization is to facilitate splitting certain context restriction rules in grammars into a conjunction of separate rules.

1 Introduction

There are different definitions for context restriction operations and rules, but they share a common idea that substrings that belong to a so-called *center language* $\mathcal{X}$ are either *accepted* or *rejected* according to the *context* where they occur.[1] A set

[1] In context restriction rules that are used in morphology, the alphabet of the strings consists of same-length correspondences. However, we avoid this complication in the current paper.

of *licensing context conditions* ($\mathfrak{C}_1, \mathfrak{C}_2, \cdots, \mathfrak{C}_n$) is specified, and each of the conditions is a pair of a *left-hand context language* and a *right-hand context language*. The context of an occurrence of the center $\mathcal{X}$ *satisfies* a context condition $\mathfrak{C}_i$ if its left-hand and right-hand contexts belong, respectively, to the left-hand and right-hand context languages. An occurrence is accepted if its context satisfies at least one of the context conditions.

The strings where different occurrences of $\mathcal{X}$ overlap each other are problematic. To treat such a string, the occurrences of the center are divided into those that are *focused*[2] and those that are *unfocused*. The string is included to the language described by context restriction operations and rules if and only if it all the focused occurrences are accepted. However, the existing definitions for context restrictions choose the focused occurrences in different ways. Some definitions for context restrictions focus all the occurrences (Yli-Jyrä, 2003). Some other definitions (related to Karttunen, 1997; Kempe and Karttunen, 1996; Karttunen, 1996) focus, non-deterministically or deterministically, a set of non-overlapping occurrences in such a way that the unfocused occurrences that remain in the string would overlap with the focused ones. There are further definitions that are partition-based (Grimley-Evans et al., 1996; Kiraz, 2000) or do not really work for long occurrences (Kaplan and Kay, 1994), which means that the occurrences cannot overlap each other at all.

Context restriction is a widely useful operation, and it is closely connected to several formalisms:

- In the classical rule formalism for the *two-level morphology*, the centers of context restriction rules are restricted to single character correspondences (Koskenniemi, 1983). Two-level context restriction rules can be compiled into finite-state transducers (FST) according to a suggestion by Ron Kaplan who solved the problem of multiple contexts in 1980's by means of context markers (Karttunen et al., 1987; Kaplan and Kay, 1994).

- Alternative two-level and multi-tiered formalisms have also been proposed (Ritchie et al., 1992; Grimley-Evans et al., 1996; Kiraz, 2000). In these formalisms, the occurrences of the center cannot overlap at all.

- In the framework of Finite State Intersection Grammar (FSIG) (a flavor of finite-state syntax) (Koskenniemi et al., 1992; Yli-Jyrä, 2003), overlapping occurrences can be focused simultaneously because the centers are not necessarily sets of unary strings as it is the case in the classical two-level morphology. In the literature, there are also cases where context restrictions

[2] A focused occurrence corresponds – as a notion – to the occurrence of $\mathcal{X}$ on a particular *application* of the context restriction rule (Karttunen et al., 1987).

could have been used as a shorthand notation for combinations of other operations (Wrathall, 1977; Grimley-Evans, 1997) and to cover **If-Then** functions of (Kaplan and Kay, 1994, p.370), if the operation only had been available as a pre-defined primitive. In these cases, context restriction can be viewed as a general purpose language operation whose arguments can be e.g. context-free languages (Wrathall, 1977; Yli-Jyrä, 2004 (in print)).

- The *replace(ment) operators* (e.g. (Karttunen, 1997; Kempe and Karttunen, 1996; Beesley and Karttunen, 2003) and the *context-dependent rewrite rules* (Kaplan and Kay, 1994; Mohri and Sproat, 1996) are also related to context restrictions, but multiple applications of the replacement / rewriting rules motivate defining restrictions in such a way that simultaneous foci do not overlap each other.

Various flavors of context restrictions differ from each other mainly due to different conceptions on possible foci in the accepted strings. We will now restrict ourselves to the definition where each string is associated with only one set of focused occurrences of the center substrings. In this set, all occurrences of the center language are focused simultaneously and each occurrence must, thus, be accepted. According to this definition, each string of length n has in the worst case $O(n^2)$ focused occurrences, and it is, therefore, not immediately obvious that regular languages are closed under the operation that has this property.

We will now give an exact definition for the flavor of context restriction (context restriction with "overlapping centers") we are concerned with. Let Σ to be the alphabet for building strings. A *context restriction of a center $\mathcal{X}$ in contexts* $\mathfrak{C}_1, \mathfrak{C}_2, \cdots, \mathfrak{C}_n$ is a operation where $\mathcal{X}$ is a subset of Σ^* and each context $\mathfrak{C}_i$, $1 \leq i \leq n$, is of the form $\mathcal{V}_i \_\_\_ \mathcal{Y}_i$, where $\mathcal{V}_i, \mathcal{Y}_i \subseteq \Sigma^*$. The operation is expressed using a notation

$$\mathcal{X} \Rightarrow \mathcal{V}_1 \_\_\_ \mathcal{Y}_1, \mathcal{V}_2 \_\_\_ \mathcal{Y}_2, \ldots, \mathcal{V}_n \_\_\_ \mathcal{Y}_n \tag{1}$$

and it defines the set of all strings $w \in \Sigma^*$ such that, for every possible $v, y \in \Sigma^*$ and $x \in \mathcal{X}$ for which $w = vxy$, there exists some context $\mathcal{V}_i \_\_\_ \mathcal{Y}_i$, $1 \leq i \leq n$, where both $v \in \Sigma^*\mathcal{V}_i$ and $y \in \mathcal{Y}_i\Sigma^*$. If all these sets $\mathcal{X}$, $\mathcal{V}_i$ and $\mathcal{Y}_i$ are regular (or star-free regular) then the result will also be regular (resp. star-free regular) (Yli-Jyrä, 2003).

The reader should note that, in the current paper, we define context languages $\mathcal{V}_i$ and $\mathcal{Y}_i$, $1 \leq i \leq n$, directly as *total contexts*. [3]

[3] In the literature (e.g. in Beesley and Karttunen, 2003), it is most often assumed that left-hand context languages $\mathcal{V}_i$ of the form Σ^*L and right-hand context languages $\mathcal{Y}_i$ of the form $L\Sigma^*$ can be

In this paper, we present a previously unpublished construction for the language denoted by context restrictions with "overlapping centers". The construction is based on a combination of usual regular operations that are easy to implement in practice. In contrast to various previous compilation methods, our new construction restricts all overlapping occurrences (vs. Kaplan and Kay, 1994; Grimley-Evans et al., 1996), and avoids exponential growth in the size of the expanded formula (vs. Yli-Jyrä, 2003). Our construction resembles the compilation method by Grimley-Evans et al. (1996) and Kiraz (2000). However, our method deals with overlapping occurrences, while theirs assumes a partitioning where the occurrences corresponds to disjoint blocks in the strings. The new method has been communicated to the Xerox group and it has already been adopted – due to its generality and speed – in XFST (Beesley and Karttunen, 2003)[4], a proprietary finite-state compiler, since XFST version 8.3.3. The new construction also generalizes so that it involves *triggering conditions* and *licensing conditions*. The *generalized restriction* has a lot of applications. A part of this paper is devoted to illustration of a possible application that allows for *decomposed context restrictions*.

The paper is structured as follows. The new construction is presented in Section 2 and generalized in Section 3. In Section 4, we describe a special problematic setting where the context restriction has bad state complexity, and then show that the generalized context restriction can be used to split context restrictions into subconstraints that are recognized with smaller automata. We list some areas of further work in Section 5, and conclude in Section 6. The appendix presents a summary of the previous solutions, being of interests only to a portion of the intended audience.

2 New Construction

2.1 Notation

We use the following usual language operations: concatenation ($L_1 L_2$), exponentiation (L^n), concatenation closure (L^*), union ($L_1 \cup L_2$), intersection ($L_1 \cap L_2$) and asymmetric difference ($L_1 - L_2$). We use parenthesis $(,)$ for grouping. When L is a regular language, we denote by $|L|$ the size of a minimal deterministic automaton that recognizes L – this is what we mean by the *state complexity* of L.

The default alphabet for building strings is Σ. Let $\diamond \notin \Sigma$. We use $\diamond$ as a special marker symbol, called a *diamond*, that is not present in the final result. For

simplified by replacing them with L in the notation. This would require prepending and appending Σ^* respectively to $\mathcal{V}_i$ and $\mathcal{Y}_i$ when the meaning of the operation is concerned. We avoid such expansions for the sake of clarity. By doing so, we do not sacrifice generality.

[4]The original XFST version attached to the book by Beesley and Karttunen (2003) would interprete simple and multi-context restrictions under different, mutually inconsistent definitions.

all alphabets M such that $M \cap \Sigma = \emptyset$ we denote the union $\Sigma \cup M$ by Σ_M. By $h_M : \Sigma_M^* \to \Sigma^*$ we denote a homomorphism w.r.t. string concatenation such that it just deletes marker symbols M from the strings. The inverse homomorphism h_M^{-1} obviously inserts symbols M freely in any string position. By $s_{\alpha/L} : \Sigma_{\{\alpha\}}^* \to \Sigma^*$ we denote the substitution that replaces in strings of $\Sigma_{\{\alpha\}}^*$ the symbol $\alpha \notin \Sigma$ with the language $L \subseteq \Sigma^*$.

2.2 Compiling Basic Restrictions

The semantics of a context restriction is formulated in four stages:

(I) We take all possible strings $w \in \Sigma^*$ where there is some focused occurrence of any substring $x \subseteq \mathcal{X}$ and insert a pair of diamonds $\diamond$ into these strings in such a way that the occurrence of x is marked by a preceding diamond $\diamond$ and a following diamond $\diamond$. The obtained set is obviously

$$\Sigma^* \diamond \mathcal{X} \diamond \Sigma^*. \tag{2}$$

Now $h_{\{\diamond\}}(\Sigma^* \diamond \mathcal{X} \diamond \Sigma^*)$ can be interpreted as the set

$$\{w \in \Sigma^* \mid \exists vxy : w = vxy \wedge x \in \mathcal{X}\}.$$

(II) We describe all strings $w = vxy$ where the string pair $v \_\_\_ y$ satisfies some licensing context $\mathcal{V}_i \_\_\_ \mathcal{Y}_i$, and we insert a pair of diamonds $\diamond$ into these strings in such a way that x is marked by a preceding and a following diamond. The obtained set is obviously

$$\cup_{i=1}^n \mathcal{V}_i \diamond \Sigma^* \diamond \mathcal{Y}_i \tag{3}$$

Now $h_{\{\diamond\}}(\cup_{i=1}^n \mathcal{V}_i \diamond \Sigma^* \diamond \mathcal{Y}_i)$ can be interpreted as the set

$$\{w \in \Sigma^* \mid \exists vxy : w = vxy \wedge \exists\, i : 1 \leq i \leq n \wedge v \in \mathcal{V}_i \wedge y \in \mathcal{Y}_i\}.$$

(III) A string $w \in \Sigma^*$ is obviously rejected by the context restriction if and only if it contains some focused occurrence of any $x \in \mathcal{X}$ such that the occurrence does not have a licensing context. If we take all rejected string and add diamonds around an arbitrary x that fails to have a licensing context, we obtain the set:

$$\Sigma^* \diamond \mathcal{X} \diamond \Sigma^* - \cup_{i=1}^n \mathcal{V}_i \diamond \Sigma^* \diamond \mathcal{Y}_i. \tag{4}$$

Now $h_{\{\diamond\}}(\Sigma^* \diamond \mathcal{X} \diamond \Sigma^* - \cup_{i=1}^n \mathcal{V}_i \diamond \Sigma^* \diamond \mathcal{Y}_i)$ can be interpreted as the set

$$\{w \in \Sigma^* \mid \exists vxy : w = vxy \wedge x \in \mathcal{X} \wedge \neg \exists\, i : 1 \leq i \leq n \wedge v \in \mathcal{V}_i \wedge y \in \mathcal{Y}_i\}.$$

(IV) The set (4) consists of all possible rejected strings, with the extra diamonds included. The desired interpretation of the restriction is therefore achieved by deleting the diamonds and taking the complement:

$$\Sigma^* - h_{\{\diamond\}}(\Sigma^* \diamond \mathcal{X} \diamond \Sigma^* - \cup_{i=1}^{n} \mathcal{V}_i \diamond \Sigma^* \diamond \mathcal{Y}_i). \tag{5}$$

This can be interpreted as the set:

$$\{w \in \Sigma^* \mid \neg(\exists vxy : w = vxy \wedge x \in \mathcal{X} \wedge \neg\exists\, i : 1 \leq i \leq n \wedge v \in \mathcal{V}_i \wedge y \in \mathcal{Y}_i)\}$$
$$= \{w \in \Sigma^* \mid \forall vxy : w = vxy \wedge x \in \mathcal{X} \rightarrow \exists\, i : 1 \leq i \leq n \wedge v \in \mathcal{V}_i \wedge y \in \mathcal{Y}_i)\}.$$

The language defined by expression 5 is the language denoted by expression 1. Given this equivalence, we are now able to construct a finite automaton corresponding to expression 1 via usual algorithms on automata, if the argument languages ($\mathcal{X}$, $\mathcal{V}_1$, $\mathcal{Y}_1$, $\mathcal{V}_2$, $\mathcal{Y}_2$, etc.) are regular and given as finite automata.

Those readers who are interested to relate our new construction with previous solutions are directed to the appendix where some earlier solutions are discussed.

3 Generalized Restriction

3.1 Definition

Now we define a new operator $\overset{g\diamond}{\Rightarrow}$, called the *generalized restriction operator*, as follows. Recall that $\mathfrak{W}$ is a Fraktur capital for 'w' and that we used variable 'w' for complete strings. Let $\mathfrak{W}_1, \ldots, \mathfrak{W}_m, \mathfrak{W}'_1, \ldots, \mathfrak{W}'_n$ be subsets of $\Sigma^*(\diamond\Sigma^*)^g$, where $g \in \mathbb{N}$. The expression

$$\mathfrak{W}_1, \mathfrak{W}_2, \ldots, \mathfrak{W}_m \overset{g\diamond}{\Rightarrow} \mathfrak{W}'_1, \mathfrak{W}'_2, \ldots, \mathfrak{W}'_n,$$

denotes the language

$$\Sigma^* - h_{\{\diamond\}}(\cup_{i=1}^{m} \mathfrak{W}_i - \cup_{i=1}^{n} \mathfrak{W}'_i). \tag{6}$$

3.2 Basic Applications

The generalized restriction operation has a potential to express many different kinds of restrictions in ways that are very similar to each other. This flexibility is illustrated by the following examples.

Context restriction Context restriction defined in (1) can be expressed as a generalized restriction as follows:

$$\Sigma^* \diamond \mathcal{X} \diamond \Sigma^* \quad \overset{2\diamond}{\Rightarrow} \quad \mathcal{V}_1 \diamond \Sigma^* \diamond \mathcal{Y}_1,\ \mathcal{V}_2 \diamond \Sigma^* \diamond \mathcal{Y}_2,\ \ldots,\ \mathcal{V}_n \diamond \Sigma^* \diamond \mathcal{Y}_n. \tag{7}$$

Coercion In analogy to the surface coercion rule in the two-level morphology (Koskenniemi, 1983; Kaplan and Kay, 1994; Grimley-Evans et al., 1996), we can define a coercion operation where satisfaction of any *triggering context condition* implies that the focused substrings are drawn from the *licensing center language*. This operation can be defined easily as follows. The expression

$$\mathcal{X}' \Leftarrow \mathcal{V}'_1 \_\_ \mathcal{Y}'_1, \mathcal{V}'_2 \_\_ \mathcal{Y}'_2, \ \ldots, \ \mathcal{V}'_m \_\_ \mathcal{Y}'_m,$$

where the backward arrow indicates that the roles of the sides are exchanged, denotes the language

$$\mathcal{V}'_1 \diamond \Sigma^* \diamond \mathcal{Y}'_1, \ \mathcal{V}'_2 \diamond \Sigma^* \diamond \mathcal{Y}'_2, \ \cdots, \ \mathcal{V}'_m \diamond \Sigma^* \diamond \mathcal{Y}'_m \quad \overset{2\diamond}{\Rightarrow} \quad \Sigma^* \diamond \mathcal{X}' \diamond \Sigma^*.$$

If-Then Kaplan and Kay (1994) defined the following functions:

$$\textit{If-P-then-S}(L_1, L_2) \overset{def}{=} \Sigma^* - L_1(\Sigma^* - L_2)$$
$$\textit{If-S-then-P}(L_1, L_2) \overset{def}{=} \Sigma^* - (\Sigma^* - L_1)L_2.$$

These functions can also be defined very intuitively using generalized restrictions with one diamond:

$$\textit{If-P-then-S}(L_1, L_2) \overset{def}{=} L_1 \diamond \Sigma^* \overset{1\diamond}{\Rightarrow} \Sigma^* \diamond L_2$$
$$\textit{If-S-then-P}(L_1, L_2) \overset{def}{=} \Sigma^* \diamond L_2 \overset{1\diamond}{\Rightarrow} L_1 \diamond \Sigma^*.$$

Nowhere Often we want to say that strings belonging to $\mathcal{X}$ do not occur anywhere in the accepted strings as substrings. This can be expressed as a context restriction $\mathcal{X} \Rightarrow \emptyset \_\_ \emptyset$ or as a generalized restriction $\Sigma^* \mathcal{X} \Sigma^* \overset{0\diamond}{\Rightarrow} \emptyset$.

More than Two Diamonds The number g of diamonds involved in the generalized restriction operation $\overset{g\diamond}{\Rightarrow}$ can also be greater than two. Such extensions can be used to express restrictions on discontinuous parts of the strings, but these possibilities are not discussed in this paper.

3.3 Adding Preconditions

The most appealing property of generalized restrictions is that they contain simultaneously centers and contexts of two different kinds: (i) licensing and (ii) triggering. This allows expressing more complicated rules in a simple and elegant manner:

Context Restriction with Preconditions When we reduce context restrictions into generalized restrictions, the left hand-hand side of the generalized restriction is of the form

$$\mathfrak{W} = \mathcal{V}' \diamond \mathcal{X} \diamond \mathcal{Y}' \text{ where } \mathcal{V}', \mathcal{Y}' = \Sigma^*.$$

The languages $\mathcal{V}'$ and $\mathcal{Y}'$ form a triggering context condition. If we make $\mathcal{V}'$ or $\mathcal{Y}'$ more restrictive, the context restriction focuses only those occurrences whose contexts satisfy the triggering context condition. When the triggering context condition $\mathcal{V}' \_\_\_ \mathcal{Y}'$ is not satisfied by the context of an occurrence, the occurrence is not focused at all and the acceptance of the whole string does not depend on it.

Coercion with Preconditions When we reduce coercions to generalized restrictions, the left hand-hand side of the generalized restriction is of the form

$$\mathcal{V}'_1 \diamond \mathcal{X}_1 \diamond \mathcal{Y}'_1,\ \mathcal{V}'_2 \diamond \mathcal{X}_2 \diamond \mathcal{Y}'_2,\ \cdots,\ \mathcal{V}'_m \diamond \mathcal{X}_m \diamond \mathcal{Y}'_m, \text{where } \mathcal{X}_i = \Sigma^* \text{ for all } 1 \leq i \leq m.$$

Now the sets $\mathcal{X}_i$, where $1 \leq i \leq m$, could also differ from Σ^*. If we make a $\mathcal{X}_i$ more restrictive, an actual context $v \_\_\_ y$, where $v \in \mathcal{V}'_i$ and $y \in \mathcal{Y}'_i$, triggers the coercion on the substring if only if it the focused substring x in the context $v \_\_\_ y$ belongs to set $\mathcal{X}_i$.

Bracketing Restriction Coercion with preconditions can be used to express the meaning of constraints called *bracketing restrictions* (Yli-Jyrä, 2003 (in print)). A bracketing restriction constraint is expressed through the notation

$$\#\mathcal{V}' \_\_\_ \mathcal{Y}'\# \Rightarrow \mathcal{X}',$$

where $\mathcal{V}', \mathcal{Y}', \mathcal{X}' \subseteq \Sigma_{\{\Delta\}}$ denote regular languages. The constraint denotes the language

$$\begin{aligned}\{w \in \Sigma^* \mid\ & w \in \Delta' \wedge \forall vxy : w = vxy \wedge x \in \Delta'\ \wedge \\ & v \in (s_{\Delta/\Delta'}(\mathcal{V}'))\ \wedge\ y \in (s_{\Delta/\Delta'}(\mathcal{Y}')) \longrightarrow x \in (s_{\Delta/\Delta'}(\mathcal{X}'))\},\end{aligned}$$

where $\Delta' \subseteq \Sigma^*$ is a bracketed *language* that is substituted for *symbol* Δ. This language can be expressed using a coercion with preconditions as follows:

$$\Delta' \cap \Big((s_{\Delta/\Delta'}(\mathcal{V}')) \diamond \Delta' \diamond (s_{\Delta/\Delta'}(\mathcal{Y}')) \overset{2\diamond}{\Rightarrow} \Sigma^* \diamond (s_{\Delta/\Delta'}(\mathcal{X}')) \diamond \Sigma^* \Big).$$

When the language D is a regular language, such as Δ_d in Section 4.1, every bracketing restriction denotes a regular language. Yli-Jyrä (2003 (in print)) discusses its state complexity and suggests decomposing each restriction into sub-constraints.

Decomposed Context Restriction Context restrictions with preconditions allow us to decompose a context restriction into a set of generalized restrictions whose intersection represents the accepted language. The context restriction

$$\mathcal{X} \Rightarrow \mathcal{V}_1 \_\_ \mathcal{Y}_1, \mathcal{V}_2 \_\_ \mathcal{Y}_2, \ldots, \mathcal{V}_n \_\_ \mathcal{Y}_n,$$

can now be expressed as an intersection of generalized restrictions

$$\begin{array}{rcl} \mathcal{V}'_1 \diamond \mathcal{X} \diamond \mathcal{Y}'_1 & \stackrel{2\diamond}{\Rightarrow} & \mathcal{V}_1 \diamond \Sigma^* \diamond \mathcal{Y}_1,\ \ldots,\ \mathcal{V}_n \diamond \Sigma^* \diamond \mathcal{Y}_n \\ & \vdots & \\ \mathcal{V}'_m \diamond \mathcal{X} \diamond \mathcal{Y}'_m & \stackrel{2\diamond}{\Rightarrow} & \mathcal{V}_1 \diamond \Sigma^* \diamond \mathcal{Y}_1,\ \ldots,\ \mathcal{V}_n \diamond \Sigma^* \diamond \mathcal{Y}_n \end{array} \tag{8}$$

such that $\cup_{i=1}^{m} \mathcal{V}'_i \diamond \mathcal{X} \diamond \mathcal{Y}'_i = \Sigma^* \diamond \mathcal{X} \diamond \Sigma^*$.

4 A Blow-Up Problem with Context Restrictions

In some practical applications, the languages described by context restrictions have bad state complexity. In the sequel, we will present the background of this problem and then show how it could be solved using decomposed context restrictions. The solution represents each context restriction by means of an intersection of simpler languages. Such an intersection corresponds to a direct product of small automata, where the direct product can be computed lazily, on demand. The improved representation is more compact and better organized, which facilitates efficient application of context restrictions.

4.1 The Background

Bracketed Finite-State Intersection Grammars An approach for natural language parsing and disambiguation based on regular languages as constraints was proposed in (Koskenniemi et al., 1992). It formed the theoretical basis for a Finite-State Intersection Grammar (FSIG) for English (Voutilainen, 1997). This particular FSIG has, however, suffered from parsing difficulties (Tapanainen, 1997). Recently, the parsing approach has been developed into new kinds of FSIG grammars that could be called Bracketed FSIGs (Yli-Jyrä, 2003; 2003 (in print); 2004 (in print); 2004a). In these FSIGs, a great deal of the state complexity of the grammar derives from balanced bracketing that is to be validated by means of finite-state constraints.

Marking the Sentence Structure In the FSIG approach, the parses for natural language sentences are represented as annotated sentences. An annotated sentence might be a sequence of multi-character symbols, including word tokens and correctly inserted part-of-speech tags and brackets, as in the following:

```
the DET man N [ who PRON walked V PAST on PREP the DET
street N ] was V happy A
```

or

```
this PRON is V the DET dog N [ that PRON chased V the
DET cat N ] [ that PRON killed V the DET mouse ].
```

In these examples, the brackets are used to mark a part of the clause structure. In some Bracketed FSIGs, the brackets mark very detailed constituency (Yli-Jyrä, 2003 (in print)) or dependency structures (Yli-Jyrä, 2004 (in print)). Nevertheless, brackets can be used economically (cf. Koskenniemi et al., 1992; Yli-Jyrä, 2003 (in print), 2004a) so that the depth of nested brackets remains small in practice.

The Language with Balanced Brackets In all the annotated sentences of FSIG, the brackets belonging to a class of left "super" brackets is balanced with the brackets belonging to the corresponding class of right "super" brackets. This property can be expressed as follows: Let B_L and B_R be respectively a class of left and right "super" brackets in the grammar. There may be other, non-"super" brackets that are not balanced, but are closed (or opened) with a "super" bracket. The set $\Delta_d \subseteq \Sigma^*$ contains all bracketed strings whose "super" brackets are balanced w.r.t. these bracket classes, and where the "super" brackets have at most d nested levels (level 0 = no brackets). This language is derived inductively as follows:

$$\Delta_d = \begin{cases} (\Sigma - B_L - B_R)^* & \text{if } d = 0 \\ (\Delta_{d-1} \cup (\ B_L\ \Delta_{d-1}\ B_R\ ))^* & \text{if } d > 0 \end{cases}$$

Voutilainen (1997) uses in his FSIG for English a special pre-defined language *dots* '**...**' (equivalent to $\boxed{\cdots}^d$ in Yli-Jyrä, 2003) to refer to arbitrary strings with balanced bracketing, where the bracketing marks boundaries of center embedded clauses. It is obtained as the language $\Delta_1 - \Sigma^*$ @@ Σ^* if we let $B_L = \{$@<$\}$ and $B_R = \{$@>$\}$.

Typical Context Restriction Rules Typical FSIG rules express contextual restrictions on features within clauses by requiring the presence of other syntactic functions and structures to the left and/or to the right. Normally only features within the same clause will be used as a context, but in case of relative pronouns

and conjunctions the required features might be after or before the clause boundary (cf. e.g. Voutilainen, 1997).

Most FSIG rules are written in such a way that they apply to deeper clauses as well as to top-level clauses. This is made possible by means of a language of balanced bracketings – denoted by '**...**', $\boxed{\cdots}^d$, or Δ_d – that is pre-defined in all FSIG frameworks. For example, the rule

$$\texttt{@SUBJ} \;\Rightarrow\; \Sigma^* \;\texttt{VFIN}\; \boxed{\cdot\cdot}^d \_\_\_ \Sigma^*, \quad \Sigma^* \_\_\_ \boxed{\cdot\cdot}^d \;\texttt{VFIN}\; \Sigma^*$$

requires that a subject feature (`@SUBJ`) is allowed only in clauses where one can also find a feature indicating a finite verb (`VFIN`). The *dotdot* $\boxed{\cdot\cdot}^d$ denotes the set of strings that are in the same clause. It is defined by the formula $\boxed{\cdot\cdot}^d = \boxed{\cdots}^d - \boxed{\cdot}^d \;\texttt{@/}\; \boxed{\cdot\cdot}^d$, where `@/` is a multi-character symbol that is used in bracketed strings to separate adjacent clauses at the same bracketing level.

4.2 Large Compilation Results

The size of the automata obtained from context restriction rules has turned out to be an obstacle for large-scale grammar development (Voutilainen, 1997). Usually we are not able to combine many context restriction rules into a single automaton. Therefore, the compiled grammar must be represented as a lazily evaluated combination (intersection) of individual context restriction rules.

Unfortunately, even separate context restriction rules can be too complex to be compiled into automata. When the center $\mathcal{X}$ or a context $\mathcal{C}_i$ is defined using the $\boxed{\cdots}^d$ or Δ_d language, the automaton obtained from a context restriction seems to grow in the worst case exponentially according to the parameter d. This effect can be perhaps most easily understood as follows: If the automaton that enforces a context restriction on unbracketed strings has k states, the automaton that enforces the restriction on one-level bracketings needs, on one hand, $O(k)$ states for keeping track of the restriction on the top level and, on the other hand, $O(k)$ states for keeping track of the restriction inside bracketed embeddings at each $O(k)$ possible states of the the top level automaton. In practice, this means that the worst-case state complexity of this automaton is in $O(k^2)$. Furthermore, it seems that the state complexity will grow exponentially according to d.

Due to these observations on the state complexity of context restriction rules, the second author of this paper proposed that we should try to split the rules into sub-rules each of which takes care of a different bracketing level[5].

[5]The idea of splitting FSIG rules into separate levels is due to the second author (Koskenniemi) who communicated it privately to the first author several years before we learned to do it constructively.

4.3 Decomposition w.r.t. Bracketing Level

We will now present a new compilation method that decomposes an arbitrary context restriction involving Δ_d into a set of generalized restrictions. Each of these generalized restrictions can then be compiled separately into finite automata.

One of the underlying assumptions in FSIGs is that the accepted strings belong to the set Δ_d (or $\boxed{\cdots}^d$). For this reason, we can replace each context restriction

$$\mathcal{X} \Rightarrow \mathcal{V}_1 \_\_ \mathcal{Y}_1, \mathcal{V}_2 \_\_ \mathcal{Y}_2, \ldots, \mathcal{V}_n \_\_ \mathcal{Y}_n$$

with

$$\Delta_d \ \cap \ (\mathcal{X} \Rightarrow \mathcal{V}_1 \_\_ \mathcal{Y}_1, \mathcal{V}_2 \_\_ \mathcal{Y}_2, \ldots, \mathcal{V}_n \_\_ \mathcal{Y}_n). \tag{9}$$

Observe that the prefixes of balanced bracketings Δ_d belong to the language $P = \cup_{i=0}^{d} \Delta_d (B_L \Delta_d)^i$. All the other prefixes in Σ^* are in the set $\Sigma^* - P$, but they are not possible prefixes for the strings in Δ_d. Accordingly, we can replace Formula (9) with an intersection of generalized restrictions as follows:

$$\cap_{i=0}^{d} L_i, \tag{10}$$

where the languages L_i are defined by the formula

$$L_i = \Delta_d \cap ((\Delta_d (B_L \Delta_d)^i) \diamond \mathcal{X} \diamond \Sigma^* \overset{2\diamond}{\Rightarrow} \mathfrak{W}'), \text{ where } \mathfrak{W}' = \cup_j^n (\mathcal{V}_j \diamond \Sigma^* \diamond \mathcal{Y}_j).$$

The language L_i, $0 \leq i \leq d$, restricts occurrences of $\mathcal{X}$ that start at the bracketing level i. Each L_i can be compiled into a separate constraint automaton and the intersection of the languages of these automata will be computed lazily during FSIG parsing.

Our hypothesis is that the obtained new lazy representation (sometimes such representations are called *virtual networks*) for the decomposed context restriction (10) will be substantially smaller than the single automaton that results from Formula (9). This hypothesis motivates the following experiment.

4.4 An Experiment

We carried out a small experiment with different representations of context restriction rules in FSIG. In the experiment we investigated the following possibilities:

- each rule corresponds to a separate constraint language $R, S, \ldots$,
- the rules are combined into a single, big constraint language $R \cap S \cap \cdots$,

- each rule is decomposed into $d + 1$ separate languages $R_0, R_1, \ldots, R_d$ as proposed in Section 4.3,
- languages R_i, $S_i, \ldots$ for each bracketing level i, $0 \leq i \leq d$, are combined into a big constraint language $R_i \cap S_i \cap \cdots$.

The Set of Rules Interesting rules (Voutilainen, 1997) taken from a full-scale grammar would have been too complicated to be investigated here. The following context restriction rules, expressing simple generalizations about the presence of syntactic categories, were used in the experiment:

$$\Delta_d \cap (\ \texttt{IOBJ} \ \Rightarrow\ \Sigma^*\ \texttt{OBJ}\ \Delta_d \_\_ \Sigma^*,\ \ \Sigma^* \_\_ \Delta_d\ \texttt{OBJ}\ \Sigma^*) \qquad (R)$$

$$\Delta_d \cap (\ \texttt{OBJ} \ \Rightarrow\ \Sigma^*\ \texttt{SUBJ}\ \Delta_d \_\_ \Sigma^*,\ \ \Sigma^* \_\_ \Delta_d\ \texttt{SUBJ}\ \Sigma^*) \qquad (S)$$

$$\Delta_d \cap (\ \texttt{SUBJ} \ \Rightarrow\ \Sigma^*\ \texttt{VFIN}\ \Delta_d \_\_ \Sigma^*,\ \ \Sigma^* \_\_ \Delta_d\ \texttt{VFIN}\ \Sigma^*) \qquad (T)$$

These rules correspond to automata that are similar to each other up to relabeling of certain transitions. One should note, however, that rules R and S (and rules S and T) have more features in common than rules R and T.

The Size of Automata The state complexity of the rules R, S, and T grows exponentially according to d, the bound for nested brackets. This is shown in Table 1. For comparison, the language R' is defined as $(\texttt{IOBJ} \ \Rightarrow\ \Sigma^*\ \texttt{OBJ}\ \Delta_d \_\_ \Sigma^*,\ \ \Sigma^* \_\_ \Delta_d\ \texttt{OBJ}\ \Sigma^*)$.

d	$\lvert\Delta_d\rvert$	$\lvert R'\rvert$	$\lvert R\rvert$	$3\lvert R\rvert + 3$	3^d
0	1	3	3	12	9
1	2	9	12	39	36
2	3	27	39	120	108
3	4	81	120	363	324
4	5	243	363	1092	972

Table 1: The state complexity of language R grows exponentially according to d.

Assuming that $d = 2$, we constructed automata for each of the languages R, S and T and splitted them for each bracketing level in order to obtain languages R_i, S_i and T_i for all i, $0 \leq i \leq d$. These automata were then combined in different ways in order to see how the sizes of automata differ in the four theoretical possibilities. The results are shown in Table 2. It shows that combinations of full rules grow substantially faster than combinations of rules decomposed with respect to the bracketing level. When we decompose the language R w.r.t. the bracketing

L	$\|L\|$	$\|L_0\|$	$\|L_1\|$	$\|L_2\|$	sum
R	39	9	7	5	21
S	39	9	7	5	21
T	39	9	7	5	21
$R \cap S$	258	18	13	8	39
$R \cap T$	819	27	19	11	57
$R \cap S \cap T$	1884	36	25	14	75

Table 2: State complexity: the rules without decomposition compared to rules that have been decomposed w.r.t. bracketing levels.

depth using the formula (10), we see in Table 3 that the state complexity of the top-most component (R_0) grows linearly to d and $|\Delta_d|$.

	$d=0$	$d=1$	$d=2$	$d=3$	d
$\|\Delta_d\|$	1	2	3	4	$(d+1)$
$\|R_0\|$	3	6	9	12	$3(d+1)$

Table 3: The state complexity of R_0 grows linearly to the parameter d.

Application Relevance For purposes of FSIG parsing, it would be nice if we could compute a minimal deterministic automaton that recognizes the intersection of all rules in the grammar. Because this cannot be done in practice, a parsing algorithm based on backtracking search has been used earlier (Tapanainen, 1997) in addition to automata construction algorithms. The search algorithm operated mostly in a left-to-right fashion.

We observe that $R_i \cap S_i \cap T_i$ have substantially smaller state complexity than $R \cap S \cap T$. This is an important finding, having applications in FSIG parsing. So far, it has not been feasible to apply all the rule automata one by one with a kind of word lattice called a *sentence automaton* (Tapanainen, 1997) by computing successive direct products of automata and minimizing the results. This might become feasible when rules have been decomposed with the current method. Decomposition of constraint restrictions w.r.t. bracketing levels apparently facilitates new techniques (Yli-Jyrä, 2004b) where the parsing proceeds partly in a bottom-up or top-down fashion rather than only in a left-right fashion.

5 Further Work

The definition of context restriction used in this paper is not the only possibility. Further research in this area is still needed due to the following reasons:

- It is not obvious whether the current flavor for the context restriction is practical if we extend context restrictions by attaching weights to contexts and centers.
- It is our experience that, for real context restrictions rules, different definitions for the operation may result in equivalent languages. Understanding when the results coincide might have practical relevance.
- It is an open question whether the proposed or the other definitions for context restrictions have a more natural interpretation when they are not interchangeable but yield different results.

While the current results on decomposed context restrictions significantly improve our possibilities to combine large portions of the original FSIG grammars into singe automata, there is still place for further research that is related to methods that organize the computation of the lazy intersection in an efficient manner.

Generalized restriction with multiple diamonds has many useful applications that remain to be studied later.

6 Conclusion

This paper discussed the definition and the representation of the context restriction operation. In particular, an alternative compilation method for generalized context restriction operation was given, with immediate applications to arrangement of constraint automata in finite-state parsers that apply multiple automata to the input.

The main result of this paper is that context restrictions can itself be restricted to be applied only in certain contexts, which means that there are two kinds of contexts that can occur in one rule: those that trigger the restriction and those that license the restricted occurrences. This observation can be used, on one hand, in defining different kinds of operations and, on the other hand, to split large context restrictions into components that can be compiled separately.

The applicability of these results are not restricted to the formalisms where so-called context restriction rules are used. The general context restriction is a useful operation that is derived from the relative complement of languages, and it allows expressing complex situations in an intuitive manner.

Acknowledgments

We are grateful to L. Karttunen, L. Carlson, A. Kempe, S. Wintner and Y. Cohen-Sygal for useful discussions, and anonymous ACL and COLING referees for critical comments. The new constructions, the experiments and the comparisons are due to the first author, whose work was funded by NorFA under his personal Ph.D. scholarship (ref.nr. 010529).

Appendix: Some Previous Solutions

An Approach that Does not Use Transducers

A Well Known Special Case There is a well known formula (Koskenniemi, 1983, p.106; Kaplan and Kay, 1994, p.371)[6] that compiles simple context restrictions with "overlapping centers":

$$\Sigma^* - ((\Sigma^* - \mathcal{V}_1)\mathcal{X}\Sigma^* \cup \Sigma^*\mathcal{X}(\Sigma^* - \mathcal{Y}_1))\,. \tag{11}$$

Compound Restriction Operations with Two Contexts The following compilation method[7] covers all 2-context cases:

$$\begin{array}{lll} \Sigma^* - (& \Sigma^*\mathcal{X}((\Sigma^* - \mathcal{Y}_1) \cap (\Sigma^* - \mathcal{Y}_2)) & \text{“at least } \mathcal{Y}_1 \text{ and } \mathcal{Y}_2 \text{ fail”} \\ & \cup\, (\Sigma^* - \mathcal{V}_1)\mathcal{X}(\mathcal{Y}_1 - \mathcal{Y}_2) & \text{“at least } \mathcal{V}_1 \text{ and } \mathcal{Y}_2 \text{ fail”} \\ & \cup\, (\Sigma^* - \mathcal{V}_2)\mathcal{X}(\mathcal{Y}_2 - \mathcal{Y}_1) & \text{“at least } \mathcal{V}_2 \text{ and } \mathcal{Y}_1 \text{ fail”} \\ & \cup\, ((\Sigma^* - \mathcal{V}_1) \cap (\Sigma^* - \mathcal{V}_2))\mathcal{X}(\mathcal{Y}_1 \cap \mathcal{Y}_2) \;\;) & \text{“only } \mathcal{V}_1 \text{ and } \mathcal{V}_2 \text{ fail”} \end{array} \tag{12}$$

It is possible to show that the last line of (12) can be simplified without changing the meaning of the whole formula:

$$\Sigma^* - (\ldots \cup ((\Sigma^* - \mathcal{V}_1) \cap (\Sigma^* - \mathcal{V}_2))\mathcal{X}\Sigma^* \;\;) \quad \text{“at least } \mathcal{V}_1 \text{ and } \mathcal{V}_2 \text{ fail”} \tag{13}$$

This modified formula is a special case of (14):

[6] Grimley-Evans et al., 1996, p.458, quote Kaplan and Kay imprecisely, suggesting that this formula for simple context restrictions does not work when context language portions overlap with portions of the center language.

[7] This formula is given in a slightly different form (with **If-Then** functions) on the web page "Operators in XFST, FSA Utilities and FSM – A synopsis. Explanation of operators for FSA Utilities.". This page has been available since year 2003 at the location `http://cs.haifa.ac.il/~shuly/teaching/03/lab/fst.html`. In addition to Jason Eisner who made the first version, many anonymous scholars have worked on this page and it is not obvious who contributed the additional sections. In 2003, Yli-Jyrä discussed with Shuly Wintner about the author of the context restriction section. That section can probably be attributed to Dale Gerdemann in Tübingen.

The General Case Yli-Jyrä (2003) generalized the latter formula (13) independently to arbitrary number of contexts. The language denoted by the context restriction with "overlapping centers" is obtained with the formula[8]

$$\Sigma^* - \bigcup_{t_1} \cdots \bigcup_{t_n} \left(\bigcap_{i=1}^{n} \phi(t_i, \mathcal{V}_i) \right) \mathcal{X} \left(\bigcap_{i=1}^{n} \phi(\overline{t_i}, \mathcal{Y}_i) \right), \tag{14}$$

where $t_1, t_2, \ldots, t_n$ are Boolean variables and the function $\phi : \{\textbf{true}, \textbf{false}\} \times 2^{\Sigma^*} \rightarrow 2^{\Sigma^*}$ is defined in such a way that $\phi(q, Q)$ returns $\Sigma^* - Q$ if q is **true**, and Σ^* otherwise. In practice, when n is small, Formula (14) is very efficient. However, when n grows, the expansion of the formula results in an exponentially growing regular expression.

To derive (14), consider the situation where a context $v \_\_\_ y$ does not satisfy any licensing context condition. It equals to $\bigwedge_{i=1}^{n} v \notin \mathcal{V}_i \vee y \notin \mathcal{Y}_i$. The disjunction $v \notin \mathcal{V}_i \vee y \notin \mathcal{Y}_i$ is true if and only if the formula $t_i \rightarrow v \notin \mathcal{V}_i \wedge \overline{t_i} \rightarrow y \notin \mathcal{Y}_i$ is satisfiable (i.e. true for some value of t_i). Thus, the formula $\bigwedge_{i=1}^{n} v \notin \mathcal{V}_i \vee y \notin \mathcal{Y}_i$ can rewritten as the following formula:

$$\exists t_1 \ldots t_n : \left(\bigwedge_{i=1}^{n} t_i \rightarrow v \notin \mathcal{V}_i \wedge \overline{t_i} \rightarrow y \notin \mathcal{Y}_i \right) \Leftrightarrow \bigvee_{t_1} \cdots \bigvee_{t_n} \left(\bigwedge_{i=1}^{n} t_i \rightarrow v \notin \mathcal{V}_i \wedge \overline{t_i} \rightarrow y \notin \mathcal{Y}_i \right)$$
$$\Leftrightarrow \bigvee_{t_1} \cdots \bigvee_{t_n} \left(v \in \bigcap_{i=1}^{n} \phi(t_i, \mathcal{V}_i) \right) \wedge \left(y \in \bigcap_{i=1}^{n} \phi(\overline{t_i}, \mathcal{Y}_i) \right).$$

When the failure condition is in this form, it is easy to see how it has been used to obtain Formula (14).

Kaplan and Kay's Approach

In the two-level model of morphology (Koskenniemi, 1983, p.106), centers of context restrictions are normally of length of one symbol (of a symbol pair alphabet). Compilation of this special case with arbitrary number of contexts has been solved with aid of marker symbols (Karttunen et al., 1987; Kaplan and Kay, 1994). This solution was originally presented using a pair symbol alphabet and rational relations and it defined such same-length relations that were used in two-level morphology.

In the following, we will reformulate Kaplan and Kay's method in the domain of regular languages (rather than in the domain of same-length relations). The one-context restriction $\mathcal{X}_{\Sigma_M} \Rightarrow_{\Sigma_M} \mathcal{V}_{\Sigma_M} \_\_\_ \mathcal{Y}_{\Sigma_M}$ for arguments $\mathcal{X}_{\Sigma_M}, \mathcal{V}_{\Sigma_M}, \mathcal{Y}_{\Sigma_M} \subseteq$

[8] To make the current presentation more coherent, the original formula of Yli-Jyrä (2003) is turned here "up-side down" by an application of DeMorgan's law.

Σ_M^* is used as a primitive operation, which is interpreted as the language $\Sigma_M^* - ((\Sigma_M^* - \mathcal{V}_{\Sigma_M})\mathcal{X}_{\Sigma_M}\Sigma_M^* \cup \Sigma_M^*\mathcal{X}_{\Sigma_M}(\Sigma_M^* - \mathcal{Y}_{\Sigma_M}))$.

The Original Method Kaplan and Kay's method allocates 2 marker symbols for each context C_i. These marker symbols form the set $M = \{\langle_i | 1 \leq i \leq n\} \cup \{\rangle_i | 1 \leq i \leq n\}$. A context restriction with n contexts is compiled as

$$h_M(\ (\cup_{i=1}^n (\mathcal{X} \Rightarrow_{\Sigma_M} \Sigma_M^* \langle_i \_\_ \rangle_i \Sigma_M^*))^* \\ \cap\ \cap_{i=1}^n (\langle_i \Rightarrow_{\Sigma_M} h_M^{-1}(\mathcal{V}_i) \_\_ \Sigma_M^*) \cap (\rangle_i \Rightarrow_{\Sigma_M} \Sigma_M^* \_\_ h_M^{-1}(\mathcal{Y}_i))). \quad (15)$$

This method works, indeed[9], only if $\mathcal{X} \subseteq \Sigma$. For example, if $\mathcal{X} = \{aa\} \subseteq \Sigma^*$, the language described by the sub-formula $(\mathcal{X} \Rightarrow_{\Sigma_M} \Sigma_M^* \langle_i \_\_ \rangle_i \Sigma_M^*)$ contains all unary strings Σ, and thus (15) yields the universal language Σ^* regardless of the licensing context conditions.[10]

An Improved Method A slightly more general solution, where $\mathcal{X}$ has the limitation $\mathcal{X} \subseteq \Sigma^*\Sigma$, is

$$h_M((\Sigma^* - \Sigma^*\mathcal{X}\Sigma^*)((\cup_{i=1}^n \langle_i \mathcal{X} \rangle_i)\ \ (\Sigma^* - \Sigma^*\mathcal{X}\Sigma^*))^* \\ \cap \cap_{i=1}^n (\langle_i \Rightarrow_{\Sigma_M} h_M^{-1}(\mathcal{V}_i) \_\_ \Sigma_M^*) \cap (\rangle_i \Rightarrow_{\Sigma_M} \Sigma_M^* \_\_ h_M^{-1}(\mathcal{Y}_i))). \quad (16)$$

This improvement is closely related to the replace operator (Karttunen, 1997; Kempe and Karttunen, 1996)[11]. It handles all context restrictions with "non-overlapping centers", but works with "overlapping centers" in a way that differs from our definition (1).

Crucial difference There are examples of context restrictions that can be used to test different definitions and differentiate them from each other. For example,

$$a\Sigma^*b \Rightarrow \Sigma^*c \_\_ \Sigma^*, \Sigma^* \_\_ d\Sigma^*$$

[9] From Kaplan and Kay, 1994, p.369, one might be able to read that the limitation $\mathcal{X} \subseteq \Sigma$ is inessential.

[10] This resembles an XFST regular expression `[[a => b _ c] | [a => d _ e]]*` by Beesley and Karttunen (2003, p. 65). That cannot be seen as a compilation approach for a context restriction of the form `[a => b _ c, d _ e]`, because it is similarly defective if `a`,`b`,`c`,`d` and `e` are constants denoting arbitrary regular languages.

[11] This method was also related although not precisely identifiable with the obsolete implementation of the restriction operator in some earlier XFST versions (before v.8.3.0). According to Kempe (2004, priv.comm.), e.g. an XFST regular expression `[? - %@]* & [X => L1 _ R2, ..., Ln _ Rn]`, where `%@` is a special symbol not occurring in `X`,`L1`,`R1`,...,`Ln`,`Rn`, would have been compiled in such a way that it corresponds to `[[? - %@]*.o.[X -> %@ || L1 _ R2, ..., Ln _ Rn].o. ~[?* X ?*]].u`. This is *roughly* the method that was used in earlier XFST versions.

demonstrates that the improved method is not equivalent to our definition (1): the result of Formula (16) accepts e.g. strings $caab$, $abdb$, $abbd$, $acab$ while the result of Formula (5) rejects them. Here, the improved method (16) produced a bigger result (7 states) than our method (4 states). In the FSIG framework (Koskenniemi et al., 1992), one can easily find more restrictions rules with "overlapping centers". Some of them would produce different results if compiled using these alternative methods (cf. Yli-Jyrä, 2003).[12]

A Partition-Based Approach

The Original Method Grimley-Evans et al. (1996) implemented a morphological grammar system that involved context restriction rules. This grammar system is partition-based, which means that possible strings (or lexical/surface correspondences) inside the system are sequences of element substrings $E \subseteq \Sigma^*$ (or lexical/surface correspondences) that are separated with a separator $\sigma \notin \Sigma$. The set of all possible sequences that are filtered with context restriction rules is, thus, $\sigma(E\sigma)^*$. The center language $\mathcal{X}$ is a subset of E. Each context restriction focuses only occurrences that are complete elements substrings. When we restrict ourselves to strings instead of correspondences, this compilation method can be formulated as

$$\sigma(E\sigma)^* - s_{\diamond/\sigma\mathcal{X}\sigma}(h^{-1}_{\{\sigma\}}(\Sigma^* \diamond \Sigma^* - \bigcup_{i=1}^{n} \mathcal{V}_i \diamond \mathcal{Y}_i)). \tag{17}$$

A Non-Partition-Based Variant Formula (17) is closely related to our construction (5). However, sequences of element substrings in the partition-based system is an unnecessary complication when the restriction operates with languages rather than regular relations. We can simplify Formula (17) by eliminating this feature. The simplification is formulated as follows:

$$\Sigma^* - s_{\diamond/\mathcal{X}}(\Sigma^* \diamond \Sigma^* - \bigcup_{i=1}^{n} \mathcal{V}_i \diamond \mathcal{Y}_i). \tag{18}$$

Formula (18) captures the definition (1) and it can be seen, therefore, as an optimization for Formula (5).

[12] The rule compiler used by Voutilainen (1997) was implemented by Pasi Tapanainen in Helsinki, and it produced in 1998 results that seem to coincide with our definition (1). The method used in the compiler has not been published, but according to Tapanainen (1992) he has used a transducer-based method for compiling implication rules.

References

Beesley, Kenneth R. and Lauri Karttunen. 2003. *Finite State Morphology*. CSLI Studies in Computational Linguistics. Stanford, CA, USA: CSLI Publications.

Grimley-Evans, Edmund. 1997. Approximating context-free grammars with a finite-state calculus. In *Proc. ACL 1997*, pages 452–459. Madrid, Spain.

Grimley-Evans, Edmund, George Anton Kiraz, and Stephen G. Pulman. 1996. Compiling a partition-based two-level formalism. In *Proc. COLING 1996*, vol. 1, pages 454–59.

Kaplan, Ronald M. and Martin Kay. 1994. Regular models of phonological rule systems. *Computational Linguistics* 20(3):331–378.

Karttunen, Lauri. 1996. Directed replacement. In *Proc. ACL 1996*, pages 108–115. Santa Cruz, CA, USA.

Karttunen, Lauri. 1997. The replace operator. In E. Roche and Y. Schabes, eds., *Finite-State Language Processing*, chap. 4, pages 117–147. Cambridge, MA, USA: A Bradford Book, the MIT Press.

Karttunen, Lauri, Kimmo Koskenniemi, and Ronald M. Kaplan. 1987. A compiler for two-level phonological rules. Report CSLI-87-108, Center for Study of Language and Information, Stanford University, CA, USA.

Kempe, André and Lauri Karttunen. 1996. Parallel replacement in finite state calculus. In *Proc. COLING 1997*, vol. 2, pages 622–627. Copenhagen Denmark.

Kiraz, George Anton. 2000. Multitiered nonlinear morphology using multitape finite automata: A case study on Syriac and Arabic. *Computational Linguistics* 26(1):77–105.

Koskenniemi, Kimmo. 1983. *Two-level morphology: a general computational model for word-form recognition and production*. No. 11 in Publ. of the Department of General Linguistics, University of Helsinki. Helsinki: Yliopistopaino.

Koskenniemi, Kimmo, Pasi Tapanainen, and Atro Voutilainen. 1992. Compiling and using finite-state syntactic rules. In *Proc. COLING 1992*, vol. 1, pages 156–162. Nantes, France.

Mohri, Mehryar and Richard Sproat. 1996. An efficient compiler for weighted rewrite rules. In *Proc. ACL 1996*, pages 231–238. Santa Cruz, CA, USA.

Ritchie, Graeme D., Graham J. Russel, and Alan W. Black. 1992. *Computational Morphology: Practical Mechanisms for the English Lexicon*. Cambridge, MA, USA: A Bradford Book, the MIT Press.

Tapanainen, Pasi. 1992. Äärellisiin automaatteihin perustuva luonnollisen kielen jäsennin. Licentiate thesis C-1993-07, Department of Computer Science, University of Helsinki, Helsinki, Finland.

Tapanainen, Pasi. 1997. Applying a finite-state intersection grammar. In E. Roche and Y. Schabes, eds., *Finite-State Language Processing*, chap. 10, pages 311–327. Cambridge, MA, USA: A Bradford Book, the MIT Press.

Voutilainen, Atro. 1997. Designing a (finite-state) parsing grammar. In E. Roche and Y. Schabes, eds., *Finite-State Language Processing*, chap. 9, pages 283–310. Cambridge, MA, USA: A Bradford Book, the MIT Press.

Wrathall, Celia. 1977. Characterizations of the Dyck sets. *RAIRO – Informatique Théorique* 11(1):53–62.

Yli-Jyrä, Anssi. 2003. Describing syntax with star-free regular expressions. In *Proc. EACL 2003*, pages 379–386. Agro Hotel, Budapest, Hungary.

Yli-Jyrä, Anssi. 2003 (in print). Regular approximations through labeled bracketing (revised version). In G. Jäger, P. Monachesi, G. Penn, and S. Wintner, eds., *Proc. FGVienna, The 8th conference on Formal Grammar, Vienna, Austria 16–17 August, 2003*, CSLI Publications Online Proceedings. Stanford, CA, USA: CSLI Publications.

Yli-Jyrä, Anssi. 2004a. Axiomatization of restricted non-projective dependency trees through finite-state constraints that analyse crossing bracketings. In G.-J. M. Kruijff and D. Duchier, eds., *Proc. Workshop of Recent Advances in Dependency Grammar*, pages 33–40. Geneva, Switzerland.

Yli-Jyrä, Anssi. 2004b. Simplification of intermediate results during intersection of multiple weighted automata. In M. Droste and H. Vogler, eds., *Weighted Automata: Theory and Applications, Dresden, Germany*, no. TUD-FI04-05 — May 2004, ISSN-1430-211X in Technische Berichte der Fakultät Informatik, pages 46–48. D-01062 Dresden, Germany: Techniche Universität Dresden.

Yli-Jyrä, Anssi. 2004 (in print). Approximating dependency grammars through regular string languages. In *Proc. CIAA 2004. Ninth International Conference on Implementation and Application of Automata. Queen's University, Kingston, Canada*, Lecture Notes in Computer Science. Springer-Verlag.

Symmetric Difference NFAs: Extended Abstract

Lynette van Zijl
Stellenbosch University, South Africa.
lynette@cs.sun.ac.za

November 10, 2004

Abstract

We give a motivation for the study of symmetric difference nondeterministic finite automata, followed by an overview of current results first on unary and then on binary symmetric difference nondeterministic finite automata. The focus point of the overview is a comparison of the descriptionl complexity of symmetric difference nondeterministic finite automata as compared to that of traditional nondeterministic finite automata. We conclude with a list of some open questions on symmetric difference nondeterministic finite automata.

1 Introduction

Symmetric difference nondeterministic finite automata ($\oplus$-NFAs) were first defined by Van der Walt and Van Zijl [7] in the late 1990s. There is a close analogy between unary $\oplus$-NFAs and linear feedback shift registers (LFSRs) [3, 7] and hence also between unary $\oplus$-NFAs and linear additive cellular automata (LACAs) [2]. Accordingly, all the applications of LFSRs and LACAs can be implemented using unary $\oplus$-NFAs. Such applications include random number generation [8], cryptology, hashing, and others. These are different applications for nondeterministic automata than those of the traditional NFAs. In addition, the reader may note that the unary $\oplus$-NFAs are directly implementable in off the shelf hardware as LFSRs.

The main interest of this author in $\oplus$-NFAs is in their ability to provide succinct representations for regular languages which cannot be succinctly represented with traditional NFAs. Moreover, we are interested not only in the question *whether* a certain representation has the ability to succinctly represent a regular language, but rather for *which* and for *how many* languages it provides a succinct description. In this aspect $\oplus$-NFAs show behaviour quite different to that of traditional NFAs.

In addition, many interesting automata-theoretic questions about $\oplus$-NFAs arise in the investigation of their succinctness properties, and these questions had not been answered for LFSRs or LACAs either.

In the next section, we cover the definition of $\oplus$-NFAs.

2 Definitions

$\oplus$-NFAs are instances of so-called generalized NFAs ($\star$-NFAs). We therefore first define $\star$-NFAs [7]. We assume that the reader has a basic knowledge of automata theory and formal languages, as for example in [5].

Note that we use symmetric difference here in the usual set theoretic sense; that is, for any two sets A and B, the symmetric difference is defined as $A \oplus B = (A \cup B) \setminus (A \cap B)$.

2.1 Definition of $\star$-NFAs

Definition 1 *A $\star$-NFA M is a 6-tuple $M = (Q, \Sigma, \delta, q_0, F, \star)$, where Q is the finite non-empty set of states, Σ is the finite non-empty input alphabet, $q_0 \subseteq Q$ is the set of start states and $F \subseteq Q$ is the set of final states. δ is the transition function such that $\delta : Q \times \Sigma \to 2^Q$. Here $\star$ is any associative commutative binary operation on sets.*

The transition function δ can be extended to $\delta : 2^Q \times \Sigma \to 2^Q$ by defining

$$\delta(A, a) = \overset{\star}{_{q \in A}} \delta(q, a) \tag{1}$$

for any $a \in \Sigma$ and $A \in 2^Q$.

δ can also be extended to $\delta : 2^Q \times \Sigma^* \to 2^Q$ as follows:

$$\delta(A, \epsilon) = A$$

and

$$\delta(A, aw) = \delta(\delta(A, a), w)$$

for any $a \in \Sigma$, $w \in \Sigma^*$ and $A \in 2^Q$.

We use the terminology $\star$-DFA to indicate the DFA equivalent to a given $\star$-NFA, as obtained by using the subset construction. We shall also say that a $\star$-NFA *needs* m deterministic states iff the minimal DFA of the $\star$-DFA has m states.

Definition 2 *Let M be a $\star$-NFA $M = (Q, \Sigma, \delta, q_0, F, \star)$, and let w be a word in Σ^*. Then M accepts w iff $F \cap \delta(q_0, w) \neq \emptyset$.*

Other definitions of acceptance were investigated in [7].

To define a $\oplus$-NFA, simply replace $\star$ in the definitions above with $\oplus$.

Example 1 *Let M be a $\oplus$-NFA defined by*

$$M = (\{q_1, q_2, q_3\}, \{a\}, \delta, \{q_1\}, \{q_3\}, \oplus)$$

with δ given by

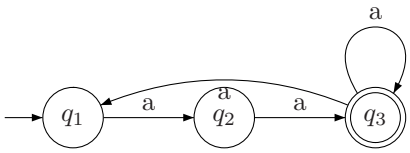


Figure 1: The $\oplus$-NFA for Ex. 1.

δ	a
q_1	$\{q_2\}$
q_2	$\{q_3\}$
q_3	$\{q_1, q_3\}$.

To find the $\oplus$-DFA M' equivalent to M, we apply the subset construction, but using symmetric difference instead of union. The transition function δ' of M' is then given by:

δ'	a
$[q_1]$	$[q_2]$
$[q_2]$	$[q_3]$
$[q_3]$	$[q_1, q_3]$
$[q_1, q_3]$	$[q_1, q_2, q_3]$
$[q_1, q_2, q_3]$	$[q_1, q_2]$
$[q_1, q_2]$	$[q_2, q_3]$
$[q_2, q_3]$	$[q_1]$.

□

Any unary $\oplus$-NFA is clearly an autonomous linear machine (see [4, 6, 8] for more detail). Therefore, the transition table of a unary $\oplus$-NFA M can be encoded as a binary matrix $\mathbf{A}$, and successive matrix multiplications in the Galois field $GF(2)$ reflect the subset construction on M. We assume that the reader is familiar with the theory of linear fields and with Galois fields, as for example in [3, 6]. Suffice it to say that $GF(2)$ contains the numbers 0 and 1, and addition is defined such that $1+1=0$, but $1+0=0+1=1$. This corresponds to the parity characteristic of the symmetric difference operation on sets.

To encode a unary $\oplus$-NFA M as a binary matrix in $GF(2)$, we use the encoding rule

$$a_{ji} = \begin{cases} 1 & \text{if } q_j \in \delta(q_i, a) \\ 0 & \text{otherwise.} \end{cases}$$

In other words, we encode every row of the transition table of M as a column in $\mathbf{A}$. The matrix $\mathbf{A}$ is called the *characteristic matrix* of M, and the polynomial $c(X) = \det(\mathbf{A} - \mathbf{I}X)$ is known as its characteristic polynomial.

Example 2 *Consider the $\oplus$-NFA in Ex. 1 above. Then* $\mathbf{A}$ *is given by*

$$\mathbf{A} = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix}.$$

Calculating $c(X)$ using standard algebra leads to $c(X) = X^3 - X^2 - 1$. The interested reader may note that $c(X)$ is a primitive irreducible polynomial in $GF(2)$. If we encode the start state as a column vector $y(0)$, and compute $\mathbf{A}^k y(0)$, we end up with the k-th entry in the on-the-fly subset construction on M. For example, with the start state q_1 encoded as $y(0) = [1\ 0\ 0]^T$, we see that $\mathbf{A}^4$ is given by

$$\mathbf{A}^4 = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix},$$

and hence $\mathbf{A}^4 y(0)$ is given by $[1\ 1\ 1]$. This corresponds to the state $[q_1, q_2, q_3]$, which is reached after four applications of the subset construction on M. Similarly, $\mathbf{A}^6 y(0)$ is given by $[0\ 1\ 1]$, which corresponds to $\{q_2, q_3\}$.

□

3 Unary $\oplus$-NFAs

Given any n-state unary $\oplus$-NFA M, one can immediately determine the size and cycle structure of its equivalent $\oplus$-NFA M' by the properties of its characteristic matrix $\mathbf{A}$ and characteristic polynomial $c(X)$. In particular, we have the following results from [3, 9]:

- If $\mathbf{A}$ is non-singular and $c(X)$ is primitive and irreducible, then M' has $2^n - 1$ reachable states in a single cycle of length $2^n - 1$.
- If $\mathbf{A}$ is non-singular and $c(X)$ is irreducible but not primitive, then M' has a cycle of length b, where b is a factor of $2^n - 1$. No other cycle lengths are possible.
- If $\mathbf{A}$ is singular, then M' has a cycle length of b, or b transient states which ends in a cycle of length one. Here b is either less than 2^{n-1}, or can be constructed as $\mathrm{lcm}(2^{n_1} - 1, 2^{n_2} - 1, \ldots, 2^{n_j} - 1)$, where $n = n_1 + n_2 + \ldots + n_j$.
- If $\mathbf{A}$ is non-singular and $c(X)$ is reducible, then M' has a number of states already occuring in one of the above cases.

Note that the results above concern the number of reachable states of the $\oplus$-DFA, and not the size of the minimal $\oplus$-DFA. However, a careful choice of final states can usually lead to examples where the $\oplus$-DFA in question is in fact minimal. Such examples were given in [7, 9, 11, 10], and we note the following results:

- There is a family of unary languages $\mathcal{L}_{k>0}$ such that each $\mathcal{L}_k$ can be accepted by a $\oplus$-NFA with n states for which the smallest equivalent $\oplus$-DFA has $2^n - 1$ states [7].

- The unary regular languages that have succinct representations with $\oplus$-NFAs can be defined precisely, namely,

$$\mathcal{L} = \bigcup_{k \in A} \{a^{i(2^n-1)+k} | i \geq 0\},$$

  for some set $A \subseteq \{0, 1, \ldots, 2^n - 2\}$.

- There are at least

$$2^n \times \frac{1}{n}\varphi(2^n - 1)$$

  distinct languages accepted by an n-state $\oplus$-NFA [11] such that the minimal $\oplus$-DFA has $2^n - 1$ states. Here $\varphi(m)$ is the Euler function which denotes the number of positive integers less than m which are relatively prime to m.

- There are regular languages that have succinct representations with $\oplus$-NFAs, but for which there are no succinct representations with traditional NFAs [11].

- There are 'magic' numbers k between 2^{n-1} and $2^n - 1$ for which there are no n-state unary $\oplus$-NFAs with equivalent minimal $\oplus$-DFAs with k states [10].

- The probability that a $\oplus$-NFA may have a succinct representation is higher than in the case of traditional NFAs [1].

Note that it is possible to exploit co-deterministic examples in constructive existence proofs, as traditional NFAs and $\oplus$-NFAs accept the same language if the NFA is co-deterministic.

We list some results not previously published in detail:

- Shortest word accepted by n-state unary $\oplus$-NFA: $n - 1$.

- k-entry $\oplus$-DFAs: Worse succinctness properties than traditional k-entry DFAs. This is easy to see: The union operation can build up all sets larger than the (deterministic) sets with cardinality one. However, the symmetric difference operation can only build the sets up to the size of the cardinality of the set of start states.

- Reversible $\oplus$-NFAs: As for LACAs; requires somewhat intricate construction for non-trivial cases. Note that Brzozowski's result on the reverse of DFAs for minimization purposes does not hold.

- Minimization of $\oplus$-NFAs: in progress (Daciuk, Mueller).

- Homogeneous $\oplus$-NFAs: Again, reversibility leads to most results in traditional case. Not applicable in $\oplus$-NFA case.

Note that the known special cases where traditional NFAs perform well as far as succinctness is concerned, appears not to behave well with $\oplus$-NFAs.

Some automata-theoretic properties of $\oplus$-NFAs that still need investigation include:

- Are there families of regular languages that can be succinctly represented by traditional NFAs, but for which there exists no succinct description with $\oplus$-NFAs? *The answer is most probably affirmative; no proof exists yet.*
- Is there a normal form for $\oplus$-NFAs, similar to Chrobak's normal form for unary traditional NFAs?
- Ambiguity in $\oplus$-NFAs, and the effect on succinctness.
- Forms of primitive words for $\oplus$-NFAs: *This relates to the positions of final states in a $\oplus$-DFA cycle, and has not been investigated either for LFSRs or LACAs, to the author's knowledge. Of special interest is the primitive words that are formed by combining two primitive words from different cycles.*
- Graphical representation of $\oplus$-NFAs.

4 Binary $\oplus$-NFAs

Many results in the unary case translate readily into the binary case, but little work has been done on binary $\oplus$-NFAs *per se*. We list only two results:

- There are no magic numbers for binary $\oplus$-NFAs [10].
- Binary $\oplus$-NFAs often generate all reachable states [1, 9].

5 Conclusion

We motivated the investigation into $\oplus$-NFAs, and gave an overview of the current state of the art. We listed a number of open questions in this field, mostly concerning the automata-theoretic properties of $\oplus$-NFAs.

Acknowledgement

This research was supported by NRF grant #2053436.

References

[1] J.-M. Champarnaud, G. Hansel, T. Paranthoën, and D. Ziadi. NFAs bitstream-based random generation. In *Proceedings of the 4th Conference on Descriptional Complexity of Formal Systems*, London, Ontario, Canada, 2002.

[2] P.P Chaudhuri, D.R. Chowdhury, S. Nandi, and S. Chattaopadhyay. *Additive Cellular Automata Theory and Applications Volume 1.* IEEE Computer Society Press, Los Alamitos, California, 1997.

[3] L.L. Dornhoff and F.E. Hohn. *Applied Modern Algebra.* MacMillan Publishing Co., Inc., New York, 1977.

[4] S. Eilenberg. *Automata, Languages and Machines, Vol. A, B.* Academic Press, New York, 1974.

[5] M. Sipser. *Introduction to the Theory of Computation.* PWS Publishing Company, 1997.

[6] H. Stone. *Discrete Mathematical Structures.* Science Research Associates, Chicago, 1973.

[7] L. van Zijl. *Generalized Nondeterminism and the Succinct Representation of Regular Languages.* PhD thesis, Stellenbosch University, 1997.

[8] L. van Zijl. Random number generation with symmetric difference NFAs. In *Proceedings of CIAA2001*, volume 2494, pages 263–273, Pretoria, South Africa, 2001. Lecture Notes in Computer Science.

[9] L. van Zijl. Nondeterminism and succinctly representable regular languages. In *Proceedings of SAICSIT'2002*, pages 212–223, Port Elizabeth, South Africa, 2002. ACM International Conference Proceedings Series.

[10] L. van Zijl. Magic numbers for symmetric difference NFAs. In *Proceedings of DCFS2004*, pages 274–284, London, Ontario, Canada, 2004.

[11] L. van Zijl. On binary $\oplus$-NFAs and succinct descriptions of regular languages. *Theoretical Computer Science*, in press 2004.

Efficient Weighted Expressions Conversion

Faissal Ouardi and Djelloul Ziadi
L.I.F.A.R., University of Rouen, France
E-mails:{Faissal.Ouardi,Djelloul.Ziadi}@univ-rouen.fr

Abstract

J. Hromkovič et *al.* have given an elegant method to convert a regular expression of size n into an ε-free nondeterministic finite automaton having $O(n)$ states and $O(n\log^2(n))$ transitions. This method has been implemented efficiently in time $O(n\log^2(n))$ by C. Hagenah and A. Muscholl. In this paper we extend this method to weighted regular expressions and we give an $O(n\log^2(n))$ implementation of this method.

1 Introduction

Weighted automata are efficient data structures for manipulating regular series. They are used in a lot of practical and theoretical applications such as computer algebra, image encoding, speech recognition or text processing. Regular series are also encoded by regular weighted expressions. The equivalence between these two representations, weighted regular expressions and finite automata has been proved in 1961 by Schützenberger [16]. For the conversion of weighted regular expression into weighted automaton there is principally three algorithms. The first one due to P. Caron and F. Flouret [4]. Their algorithm works on a subset of weighted regular expressions and produces the position automaton. Champarnaud et *al.* [5] given an efficient algorithm that convert a weighted regular expression into its position automaton in quadratic time on the size of the expression. Their algorithm is mainly based on the use of the $\mathbb{K}\mathcal{Z}\mathcal{PC}$-structure for weighted expressions. The position automaton is a particular one (see [6]) that has a quadratic number of transitions and $(n+1)$ states where n is the alphabetic width of the expression. S. Lombardy and J. Sakarovitch [14] algorithm constructs a weighted automaton that turns out to be the generalization of Antimirov automaton for the Boolean case. The Antimirov automaton [1] has less states than the positions automaton. As the position automaton, Antimirov automaton has a quadratic number of transitions.

In the Boolean case many algorithms has been developed for the problem of conversion [8, 15, 2, 12, 17]. The best one in term of complexity is that proposed by J. Hromkovič et *al.* [11] and implemented by C. Hagenah and A. Muscholl [9]. This automaton called the Common Follow Sets automaton has a $O(n)$ of states and $O(n\log^2(n))$ transitions. C. Hagenah and A. Muscholl proved that this automaton can be constructed from a regular expression of size n in time $O(n\log^2(n))$ which constitutes the best algorithm for the conversion problem.

In this paper, we extend J. Hromkovič et al. algorithm to weighted regular expressions and we present an efficient algorithm to convert a weighted regular expression of size n into an automaton having $O(n)$ states and $O(n\log^2(n))$ in $O(n\log^2(n))$ time.

In section 2, we first recall notions of $\mathbb{K}$-expressions and formal series. In section 3 we present the position automaton construction. In section 4 we introduce the notion of Common Follow Polynomials automaton which is the generalization of the notion of Common Follow Sets automaton presented in [11]. In section 5 we recall the $\mathbb{K}\mathcal{ZPC}$-structure in order to implement efficiently the algorithm introduced in section 4. Finally in Section 7 and 8 we describe our algorithm.

2 Preliminaries

Let A be a finite alphabet, and $(\mathbb{K},\oplus,\otimes,0,1)$ be a semiring (commutative or not). The operator star $\circledast$ can be partially defined, the scalar $y^{\circledast}\in\mathbb{K}$ being a solution (if there exists) of the equations $y\otimes x\oplus 1=y$ and $x\otimes y\oplus 1=y$ [10, 13]. In the following definition, we introduce the notion of $\mathbb{K}$-expression:

Definition 1 *$\mathbb{K}$-expressions over an alphabet A are inductively defined as follows:*

- *$a\in A$ and $k\in\mathbb{K}$ are $\mathbb{K}$-expressions,*
- *if F and G are $\mathbb{K}$-expressions, then $(F+G)$, $(F\cdot G)$, and (F^*) are $\mathbb{K}$-expressions.*

When there is no ambiguity, the $\mathbb{K}$-expression $(F\cdot G)$ will be denoted (FG). Let E be a $\mathbb{K}$-expression. We will denote A_E the alphabet of E. The linearized version $\overline{E}$ of E is the $\mathbb{K}$-expression deduced from E by ranking every letter occurrence with its position in E. Subscripted letters are called positions. The size of E, denoted $|E|$ is the size of the syntactical tree of E. For example, if $E=(\frac{1}{2}\cdot a^*+\frac{1}{3}\cdot b^*)^*\cdot a^*$, we get $A_E=\{a,b\}$, $\overline{E}=(\frac{1}{2}\cdot a_1^*+\frac{1}{3}\cdot b_2^*)^*\cdot a_3^*$, $A_{\overline{E}}=\{a_1,b_2,a_3\}$ and $|\overline{E}|=13$.

We define inductively the *null term* of a $\mathbb{K}$-expression E, denoted $c(E)$, by:

$$\begin{aligned} c(k) &= k \text{ for all } k\in\mathbb{K}\\ c(a) &= 0 \text{ for all } a\in A\\ c(F+G) &= c(F)\oplus c(G)\\ c(FG) &= c(F)\otimes c(G)\\ c(F^*) &= c(F)^{\circledast}\end{aligned}$$

The null term of $E=(\frac{1}{2}a^*+\frac{1}{3}b^*)^*a^*$ is $c(E)=6$.

In the following, we present a brief description of formal series and we define a subset of the set of $\mathbb{K}$-expressions, usually called regular $\mathbb{K}$-expressions, which are associated to regular series [3].

Definition 2 *A (non-commutative) formal series with coefficients in $\mathbb{K}$ and variables in A is a map from the free monoid A^* to $\mathbb{K}$ which associates with the word $w\in A^*$ a coefficient $(S,w)\in\mathbb{K}$.*

A formal series is usually written as an infinite sum: $S = \sum_{u \in A^*} (S, u)u$. The *support* of the formal series S is the language $\mathrm{supp}(S) = \{u \in A^* \mid (S, u) \neq 0\}$. The set of formal series over A with coefficients in $\mathbb{K}$ is denoted by $\mathbb{K}\langle\langle A \rangle\rangle$. A structure of semiring is defined on $\mathbb{K}\langle\langle A \rangle\rangle$ as follows [3, 13]:

- $(S + T, u) = (S, u) \oplus (T, u)$,
- $(ST, u) = \bigoplus_{u_1 u_2 = u} (S, u_1) \otimes (T, u_2)$, with $S, T \in \mathbb{K}\langle\langle A \rangle\rangle$.

A polynomial is a formal series with finite support. The set of polynomials is denoted by $\mathbb{K}\langle A \rangle$. It is a subsemiring of $\mathbb{K}\langle\langle A \rangle\rangle$. The star of series is defined by : $S^* = \sum_{n \geq 0} S^n$ with $S^0 = \varepsilon,\ S^n = S^{n-1}S$ if $n > 0$. Notice that the star of a formal series does not always exist:

Proposition 1 *[13] The star of a formal series $S \in \mathbb{K}\langle\langle A \rangle\rangle$ is defined if and only if $(S, \varepsilon)^{\circledast}$ is defined in $\mathbb{K}$. In this case:*

$$S^* \quad = \quad (S, \varepsilon)^{\circledast}(S_0(S, \varepsilon)^{\circledast})^* \tag{1}$$

where the formal series S_0 is defined by $(S_0, \varepsilon) = 0$ and $(S_0, u) = (S, u)$ for any word u.

In the following, we will consider the previous construction of star of formal series.

Definition 3 *The semiring of regular series $\mathbb{K}rat(A^*) \subset \mathbb{K}\langle\langle A \rangle\rangle$ is the smallest set of $\mathbb{K}\langle\langle A \rangle\rangle$ which contains the polynomials semiring $\mathbb{K}\langle A \rangle$, and which is stable by the operations of addition, product and star when this latter is defined.*

The following definition introduces the notion of regular $\mathbb{K}$-expression.

Definition 4 *A regular $\mathbb{K}$-expression is defined inductively by:*

- *$a \in A, k \in \mathbb{K}$ are regular $\mathbb{K}$-expressions which respectively denote the regular series $S_a = a$ and $S_k = k$,*
- *if F, G and H (s.t. $c(H)^{\circledast}$ exists) are regular $\mathbb{K}$-expressions which respectively denote the regular series S_F, S_G and S_H, then $(F + G), (FG)$, and (H^*) are regular $\mathbb{K}$-expressions respectively denote the regular series $S_F + S_G$, $S_F S_G$ and ${S_H}^*$.*

Definition 5 *Let A be a finite alphabet, and $\mathbb{K}$ be a semiring (commutative or not). We define an automaton with multiplicities $\mathcal{A} = (Q, q_0, \delta, F, \mu)$ as follows:*

- *Q a finite set of states,*
- *q_0 the initial state with 1 as initial coefficient,*
- *$\delta : Q \times \mathbb{K} \times A \to 2^Q$ the transition function,*
- *$F \subseteq Q$ is the set of final states,*
- *$\mu : Q \to \mathbb{K}$ the final function, (we have $q \in F$ if and only if $\mu(q) \neq 0$).*

The definition of an automaton with multiplicities is more general but this one is sufficient for our construction.

A recognized path p in $\mathcal{A}$ is a sequence $(q_0, q_1), (q_1, q_2), \cdots, (q_{n-1}, q_n)$ of transitions in $\mathcal{A}$. It is written $p = (q_0, q_1, q_2, \cdots, q_{n-1}, q_n)$. We denote by $\mathrm{coef}(p)$ the cost of the path p in $\mathcal{A}$. Formally, we get $\mathrm{coef}(p) = \alpha_{q_1} \otimes \alpha_{q_2} \otimes \cdots \otimes \alpha_{q_n} \otimes \mu(q_n)$ with $q_i \in \delta(q_{i-1}, \alpha_{q_i}, a_{q_i})$, where $q_i \in Q$, $a_{q_i} \in A$ and $\alpha_{q_i} \in \mathbb{K}$ for all $1 \leq i \leq n$. We denote by $w(p)$ the word obtained in $\mathcal{A}$ by going across the path p, i.e. $w(p) = a_{q_1} a_{q_2} \cdots a_{q_n}$ where $a_{q_i} \in A$ for all $1 \leq i \leq n$. We denote by $\mathcal{C}_\mathcal{A}$ the set of all recognized paths in $\mathcal{A}$. From these functions, we can define the series $S_\mathcal{A}$ associated with the automaton $\mathcal{A}$:

$$S_\mathcal{A} = \sum_{u \in A^*} (S_\mathcal{A}, u) u$$

where:

$$(S_\mathcal{A}, u) = \bigoplus_{\substack{p \in \mathcal{C}_\mathcal{A} \\ w(p) = u}} \mathrm{coef}(p)$$

We say that the automaton $\mathcal{A}$ *realizes* the series $S_\mathcal{A}$.

Definition 6 *A formal series $S \in \mathbb{K}\langle\langle A \rangle\rangle$ is called recognizable if there exists an automaton that realizes it.*

The following result due to Schützenberger [16] is classical:

Theorem 1 *(Schützenberger, 1961). A formal series is recognizable if and only if it is regular.*

3 Position automaton

In this section, we recall some basic notions related to the construction of position automata from regular $\mathbb{K}$-expressions. Let E be a regular $\mathbb{K}$-expression over an alphabet A. The position automaton associated with E is computed from three functions First, Last and $\mathrm{Follow}(\cdot, E)$. These functions are defined from $L(\overline{E})$ ($L(\overline{E})$ is the language associated to the linearized version of E) [5]. They can be computed inductively according to the following rules:

$$\begin{aligned}
\mathrm{First}(k) &= 0 \text{ for all } k \in \mathbb{K} && (2)\\
\mathrm{First}(a) &= 1a_i \text{ (a_i is the position associated to a in } A_{\overline{E}}) && (3)\\
\mathrm{First}(F + G) &= \mathrm{First}(F) + \mathrm{First}(G) && (4)\\
\mathrm{First}(FG) &= \mathrm{First}(F) + c(F)\,\mathrm{First}(G) && (5)\\
\mathrm{First}(F^*) &= c(F)^{\circledast}\,\mathrm{First}(F) && (6)
\end{aligned}$$

We obtain the same rule for Last by substituting Last for First and replacing the formula (4) by:

$$\mathrm{Last}(FG) = \mathrm{Last}(G) + c(G)\,\mathrm{Last}(F) \quad (7)$$

Given a position $x \in A_{\overline{E}}$, the function $\mathrm{Follow}(x, E)$ is inductively computed as follows:

$$\begin{aligned}
\mathrm{Follow}(x, k) &= 0 \text{ for all } k \in \mathbb{K} \\
\mathrm{Follow}(x, a) &= 0 \text{ for all } a \in A \\
\mathrm{Follow}(x, F + G) &= \mathrm{Follow}(x, F) + \mathrm{Follow}(x, G) \\
\mathrm{Follow}(x, FG) &= \mathrm{Follow}(x, F) + (\mathrm{Last}(F), x)\,\mathrm{First}(G) + \mathrm{Follow}(x, G) \\
\mathrm{Follow}(x, F^*) &= \mathrm{Follow}(x, F) + (\mathrm{Last}(F^*), x)\,\mathrm{First}(F)
\end{aligned}$$

Proposition 2 *[5] Let E be a regular $\mathbb{K}$-expression. The formal series* First, Last *and* Follow *of E are polynomials.*

Let h be the mapping from $A_{\overline{E}}$ to A_E induced by the linearization of E over $A_{\overline{E}}$. It maps every position to its value in A_E. For example, if $\overline{E} = 2a_1 + 3b_2 + a_3$ then $h(a_1) = h(a_3) = a$ and $h(b_2) = b$. The First, Last and Follow polynomials of a regular $\mathbb{K}$-expression E can be used to define an automaton with multiplicities realizing S_E. We define the *position automaton* for E, denoted $\mathcal{A}_E$, as the 5-tuple (Q, q_0, F, δ, μ) where, for some $q_0 \notin A_{\overline{E}}$,

- $Q = \{q_0\} \cup A_{\overline{E}}$,
- $\delta(q_0, \alpha, h(q)) = q$ if $(\mathrm{First}(E), q) = \alpha$,
- $\delta(p, \alpha, h(q)) = q$ if $(\mathrm{Follow}(p, E), q) = \alpha$ for all p, q in $A_{\overline{E}}$,
- $q \in F \subseteq Q$ if and only if $\mu(q) \neq 0$ with $\mu(q) = \begin{cases} c(E) & \text{if } q = q_0, \\ (\mathrm{Last}(E), q) & \text{otherwise.} \end{cases}$

Proposition 3 *[5] Let E be a regular $\mathbb{K}$-expression and $\mathcal{A}_E$ its position automaton. Then $\mathcal{A}_E$ realizes the regular series S_E.*

4 Common Follow Polynomials automaton

In this section, we introduce the notion of Common Follow Polynomials system which is the generalization of the notion of the Common Follow Sets introduced by J. Hromkovič et *al.* [11] in order to compute an automaton having less transitions than the position automaton.

Next, we prove that the CFP automaton induced by the Common Follow Polynomials system realizes the same series as the position automaton.

Finally, we recall the $\mathbb{K}\mathcal{ZPC}$-structure in order to implement an efficient algorithm to convert a regular $\mathbb{K}$-expression into its CFP automaton.

From now, we will consider the expression $E' = (\overline{E} \cdot \sharp)$ with $\sharp \notin A_{\overline{E}}$ and we will use the operator $\times$ as follows:
If $C = \alpha_1 x_1 + \cdots + \alpha_n x_n$ is a polynomials in $\mathbb{K}\langle A \rangle$ and $k \in \mathbb{K}$ then $k \times C = (k \otimes \alpha_1)x_1 + \cdots + (k \otimes \alpha_n)x_n$.

Definition 7 (Commun Follow Polynomials System) *Let E be a regular $\mathbb{K}$-expression and $E' = \overline{E}\sharp$. A CFP system for E is given as $S(E) = (\mathrm{dec}(x))_{x\in A_{\overline{E}}}$, where each $dec(x) \subseteq \mathbb{K}\langle A_{\overline{E}} \cup \{\sharp\}\rangle$ is a decomposition of* $\mathrm{Follow}(x, E')$*, that is:*

$$\mathrm{Follow}(x, E') = \sum_{k\times C\in \mathrm{dec}(x)} k \times C$$

The family of common follow polynomials $\mathcal{F}_E$ associated with this system is defined by

$$\mathcal{F}_E \;=\; \{\mathrm{First}(E')\} \cup \bigcup_{x\in A_{\overline{E}}} \{C \mid \exists k \in \mathbb{K} \textit{ such that } k \times C \in \mathrm{dec}(x)\}$$

Notice that $\mathcal{F}_E$ is not unique as we can see in the following example.

Example 1 *Let $E = (\frac{1}{3}a + \frac{1}{6}b)(\frac{1}{2}a^*b)$. One has:*

$$\begin{aligned} E' &= (\frac{1}{3}a_1 + \frac{1}{6}b_2)(\frac{1}{2}a_3^*b_4)\sharp \\ \mathrm{First}(E') &= \frac{1}{3}a_1 + \frac{1}{6}b_2 \end{aligned}$$

$\mathrm{Follow}(a_1, E') = \frac{1}{2} \times (a_3 + b_4)$	$\mathrm{Follow}(a_1, E') = \frac{1}{2} \times (a_3) + \frac{1}{2} \times (b_4)$
$\mathrm{dec}(a_1) = \{\frac{1}{2} \times (a_3 + b_4)\}$	$\mathrm{dec}(a_1) = \{\frac{1}{2} \times (a_3), \frac{1}{2} \times (b_4)\}$
$\mathrm{Follow}(b_2, E') = \frac{1}{2} \times (a_3 + b_4)$	$\mathrm{Follow}(b_2, E') = \frac{1}{2} \times (a_3) + \frac{1}{2} \times (b_4)$
$\mathrm{dec}(b_2) = \{\frac{1}{2} \times (a_3 + b_4)\}$	$\mathrm{dec}(b_2) = \{\frac{1}{2} \times (a_3), \frac{1}{2} \times (b_4)\}$
$\mathrm{Follow}(a_3, E') = 1 \times (a_3 + b_4)$	$\mathrm{Follow}(a_3, E') = 1 \times (a_3 + b_4)$
$\mathrm{dec}(a_3) = \{1 \times (a_3 + b_4)\}$	$\mathrm{dec}(a_3) = \{1 \times (a_3 + b_4)\}$
$\mathrm{Follow}(b_4, E') = 1 \times \sharp$	$\mathrm{Follow}(b_4, E') = 1 \times \sharp$
$\mathrm{dec}(b_4) = \{1 \times \sharp\}$	$\mathrm{dec}(a_1) = \{1 \times \sharp\}$
$\mathcal{F}_E = \{\frac{1}{3}a_1 + \frac{1}{6}b_2, a_3 + b_4, \sharp\}$	$\mathcal{F}_E = \{\frac{1}{3}a_1 + \frac{1}{6}b_2, a_3, b_4, a_3 + b_4, \sharp\}$

Definition 8 (CFP automaton) *Let E a regular $\mathbb{K}$-expression and $S(E)$ its Common Follow Polynomials system. The Common Follow Polynomials automaton associated to E is defined by $\mathcal{A}_{S(E)} = (Q, A, q_0, \delta_1, F, \mu_1)$ where*

- $Q = \mathcal{F}_E$,
- $q_0 = \{\mathrm{First}(E')\}$,
- $\delta_1 = \{(C, \alpha a, C') | \alpha = \bigoplus_{1\le i\le m} \beta_i \otimes k_{a_i}$ *where* $\sum_{1\le i\le m} \beta_i a_i$ *is a subpolynomials of C s.t. $h(a_i) = a$ and $k_{a_i} \times C' \in \mathrm{dec}(a_i)$ for $0 \le i \le m$}$,
- $C \in F$ *if and only if* $\mu_1(C) = (C, \sharp) \neq 0$.

Theorem 2 *Let E be a regular $\mathbb{K}$-expression and $\mathcal{A}_{S(E)}$ its common follow polynomials automaton. Then $\mathcal{A}_{S(E)}$ realizes the regular series S_E.*

To prove this theorem, we introduce the automaton $\mathcal{A}_{S(\overline{E})} = (Q, A_{\overline{E}}, q_0, \delta_2, F, \mu_1)$ defined as follows:

- $Q = \mathcal{F}_E$,
- $q_0 = \{\mathrm{First}(E')\}$,
- $\delta_2 = \{(C, k_j\beta_i a_i, C') \quad | \quad \beta_i a_i$ a monomials of C and $k_j \times C' \in \mathrm{dec}(a_i)\}$,
- $C \in F$ if and only if $\mu_1(C) = (C, \sharp) \neq 0$.

Lemma 1 *Let E be a regular $\mathbb{K}$-expression. The automaton $\mathcal{A}_{S(\overline{E})}$ realizes the series $S_{\overline{E}}$.*

Proof. Let E be a regular $\mathbb{K}$-expression and $\mathcal{A}_{\overline{E}}$ the position automaton associated to its linearized version $\overline{E}$ and let $S(E) = (\mathrm{dec}(x))_{x \in A_{\overline{E}}}$ a *CFP system* associated with E.
Let $w = a_1 a_2 \cdots a_n \in {A_{\overline{E}}}^*$, From Proposition 3, $\mathcal{A}_{\overline{E}}$ realizes the series $S_{\overline{E}}$, so to prove this Lemma it suffices to show that for each path $p \in \mathcal{C}_{\mathcal{A}_{\overline{E}}}$, such that $m(p) = w$, there exists a unique equivalent path $p' \in \mathcal{C}_{\mathcal{A}_{S(\overline{E})}}$ such that $m(p') = w$ and $\mathrm{coef}(p) = \mathrm{coef}(p')$, and conversely.
Let p be a path in $\mathcal{C}_{\mathcal{A}_{\overline{E}}}$ such that $m(p) = w$ and $\mathrm{coef}(p) = \alpha_1 \otimes \cdots \otimes \alpha_n \otimes \mu_1(a_n)$. From the definition of a *CFP system*, one has, for each position $a_i \in A_{\overline{E}}$ there exists a family of polynomials $(C_j)_{l \leq j \leq i_l}$, such that $\mathrm{dec}(a_i) = \cup_{1 \leq j \leq i_l} k_j \times C_j$ and $\mathrm{Follow}(E', a_i) = \sum_{k \times C \in \mathrm{dec}(a_i)} k \times C$. Thus, by a straightforward induction on the length of w:

- In $\mathcal{A}_{\overline{E}}$, one has $\delta(q_0, \alpha_1 a_1) = a_1$, then there exists a unique $k_1 \times C_1 \in \mathrm{dec}(a_1)$ such that $\beta_2 = (C_1, a_2) \neq 0$. Thus, in $\mathcal{A}_{S(\overline{E})}$, there exists a unique equivalent transition from the state $\mathrm{First}(E')$ to the state C_1 realizing $\delta_2(\mathrm{First}(E'), \gamma_1 a_1) = C_1$, where $\gamma_1 = \alpha_1 \otimes k_1$.

- Similarly, in $\mathcal{A}_{\overline{E}}$, one has $\delta(a_{i-1}, \alpha_i a_i) = a_i$ and $\delta(a_i, \alpha_{i+1} a_{i+1}) = a_{i+1}$, then there exists a unique $k_i \times C_i \in \mathrm{dec}(a_i)$ such that $\beta_{i+1} = (C_i, a_{i+1}) \neq 0$ and $\alpha_i = k_{i-1} \otimes \beta_i$ for all $2 \leq i \leq n$. Thus, in $\mathcal{A}_{S(\overline{E})}$, there exists a unique equivalent transition from the state C_{i-1} to the state C_i realizing $\delta_2(C_{i-1}, \gamma_i a_i) = C_i$, where $\gamma_i = \beta_i \otimes k_i$ and $\alpha_i = k_{i-1} \otimes \beta_i$ for all $2 \leq i \leq n$. When we arriving at the state a_n in $\mathcal{A}_{\overline{E}}$, in $\mathcal{A}_{S(\overline{E})}$, one has an equivalent state C_n, where there exists $k_n \times C_n \in \mathrm{dec}(a_n)$ and $\beta_{n+1} = (C_n, \sharp) \neq 0$.

Finally, there exists a unique path c' in $\mathcal{A}_{S(\overline{E})}$ such that $m(p') = w$ and
$\mathrm{coef}(p') = (\underbrace{\alpha_1}_{\alpha_1} \otimes \underbrace{k_1) \otimes (\beta_2}_{\alpha_2} \otimes \underbrace{k_2) \otimes \cdots}_{\alpha_3} \cdots \otimes \underbrace{(\beta_n}_{\alpha_n} \otimes \underbrace{k_n) \otimes \beta_{n+1}}_{\mu_1(a_n)} = \mathrm{coef}(p)$.
Conversely, let p' be a path in $\mathcal{A}_{S(\overline{E})}$ with $m(p') = w$ and $\mathrm{coef}(p') = \gamma_1 \otimes \gamma_2 \otimes \cdots \gamma_n \otimes \beta_{n+1}$. From the definition of the CFP automaton, one has $\gamma_i = \beta_i \otimes k_i$ with $\delta_2(C_{i-1}, \alpha_i a_i) = C_i$, $k_i \times C_i \in \mathrm{dec}(a_i)$ and $\beta_i a_i$ a term in C_{i-1}, we have $\beta_{i+1} = (C_i, a_{i+1}) \neq \overline{O}$ then there exists a unique equivalent transition $\delta(a_{i-1}, \alpha_i a_i) = a_i$ in $\mathcal{A}_{\overline{E}}$ such that $\alpha_i = k_{i-1} \otimes \beta_i$ and $\gamma_i = \beta_i \otimes k_i$ for all $2 \leq i \leq n$, and for $\delta_2(\mathrm{First}(E'), \gamma_1 a_1) = C_1$, one has an equivalent transition in $\mathcal{A}_{\overline{E}}$, $\delta(q_0, \alpha_1 a_1) = a_1$ such that $\gamma_1 = \alpha_1 \otimes k_1$. ∎

Proof of theorem 2. Let E be a regular $\mathbb{K}$-expression and $w \in {A_E}^*$. We must prove that $(S_{\mathcal{A}_{S(E)}}, w) = (S_E, w)$. By Lemma 1, it's suffices to prove that

$$\bigoplus_{\substack{p' \in \mathcal{C}_{\mathcal{A}_{S(E)}} \\ m(p')=w}} \text{coef}(p') = \bigoplus_{\substack{p \in \mathcal{C}_{\mathcal{A}_{S(\overline{E})}} \\ h(m(p))=w}} \text{coef}(p)$$

Let $(p_i)_{1 \leq i \leq n}$ the family of paths in $\mathcal{A}_{S(\overline{E})}$ realizing $h(m(p_i)) = w$, for all $1 \leq i \leq n$. We suppose without loss of generality that $m(p_i) = a_1 a_2 \cdots a_{t_i} \cdots a_l$, $\text{coef}(p_i) = \alpha_1 \otimes \alpha_2 \otimes \cdots \alpha_{t_i} \cdots \otimes \alpha_n$ and $h(a_{t_i}) = a$, for all $1 \leq i \leq n$. Then, in $\mathcal{A}_{S(\overline{E})}$ if there exists two states C and C' such that $(C, a_{t_i}) \neq 0$ and $k_i \times C' \in \text{dec}(a_{t_i})$, one has an unique equivalent path p' in $\mathcal{A}_{S(E)}$ such that $m(p') = w$ and

$$\bigoplus_{1 \leq i \leq n} (\alpha_1 \otimes \alpha_2 \otimes \cdots \alpha_{t_i} \cdots \otimes \alpha_n) = \alpha_1 \otimes \alpha_2 \otimes \cdots \underbrace{(\bigoplus_{1 \leq i \leq n} \alpha_{t_i})}_{\alpha} \cdots \otimes \alpha_n = \text{coef}(p'). \qquad \blacksquare$$

Example 2 *For* $E = (\frac{1}{3}a + \frac{1}{6}b)(\frac{1}{2}a^*b)$. *Consider the following family of Common Follow Polynomials system* $\mathcal{F}_E = \{\frac{1}{3}a_1 + \frac{1}{6}b_2\} \cup \{a_3 + b_4, \sharp\}$.

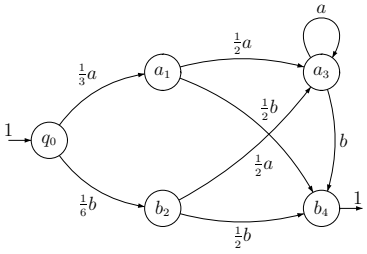


Figure 1: The position automaton for $E = (\frac{1}{3}a + \frac{1}{6}b)(\frac{1}{2}a^*b)$.

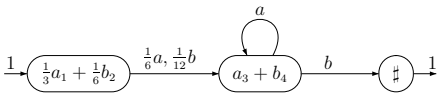


Figure 2: A CFP automaton for $E = (\frac{1}{3}a + \frac{1}{6}b)(\frac{1}{2}a^*b)$.

Note that for the previous example, the position automaton associated with E has 5 states and 8 transitions (See Figure 1). However there exists a CFP automaton realizing the series S_E with only 3 states and 4 transitions (See Figure 2).

The number of transitions and the number of states in a Common Follow Polynomials Automaton obviously depends on the choice of the Common Follow Polynomials System. The next section deals with the problem of finding appropriate common follow polynomials.
A good choice can be found by resolving the following system

$$\left\{ \; Minimize(f(E) \;\; = \;\; \sum_{x \in A_{\overline{E}}} n_x m_x). \right.$$

Where m_x denotes the number of polynomials $C \in \mathcal{F}_E$ such that $(C, x) \neq 0$ and n_x denotes the size of $\text{dec}(x)$.

From the definition of the automaton $\mathcal{A}_{S(\overline{E})}$, the function f represent the number of its transitions.

Unfortunately, In the practice this method is not efficient. In the Boolean case J. Hromkovič et *al*. [11], presented an elegant method that computes a particular Common Follow Sets system which yields to a Common Follow Sets automaton having $O(n)$ states and $O(n\log^2(n))$ transitions where n denote the size of the regular expression. In [9] C. Hagenah and A. Muscholl have shown that this particular automaton can be computed in time $O(n\log^2(n))$ which is the best algorithm for the problem of conversion in the boolean case.

In the next sections, we prove that for a regular $\mathbb{K}$-expression of size n there exists a Common Follow Polynomials automaton with $O(n\log^2(n))$ transitions and $O(n)$ states. This constitutes the generalization of the result proved by J. Hromkovič et *al.*[11]. Next we give an efficient algorithm that computes this automaton in time $O(n\log^2(n))$.

Our algorithm is based on the $\mathbb{K}\mathcal{ZPC}$-structure . This structure has been introduce in [5], in order to compute the position automaton. The following section gives a brief description of this structure.

5 $\mathbb{K}\mathcal{ZPC}$-structure

Let E be a regular $\mathbb{K}$-expression. The $\mathbb{K}\mathcal{ZPC}$-structure of E is based on two labeled trees ($\mathrm{TL}(E)$ and $\mathrm{TF}(E)$) deduced from its syntactical tree $\mathrm{T}(E)$. These trees encode respectively the Last and the First polynomials associated to the subexpressions of E. The edges of these trees are labeled by elements of $\mathbb{K}$.
A node in $\mathrm{T}(E)$ will be noted ν. If the arity of ν is two, we write respectively ν_l and ν_r its left son and its right son. If its arity is 1, its son will be noted ν_s. The relation of descendance over the syntax tree is denoted $\preceq$. For a tree whose edges are labeled by elements of the semiring $\mathbb{K}$, we define the function of cost π as follows:

$$\pi(\vartheta,\vartheta') = \begin{cases} \text{the cost of the path } \vartheta \rightsquigarrow \vartheta' & \text{if } \vartheta \preceq \vartheta' \text{or } \vartheta' \preceq \vartheta, \\ 0 & \text{otherwise.} \end{cases}$$

By convention we set $\pi(\vartheta,\vartheta) = 1$.

A subtree t of $\mathrm{T}(E)$ is a tree associated to a subexpression of E', or a tree obtained from $\mathrm{T}(E)$ by deleting a set of trees which represent some subexpressions of E'. This definition is also applied to t. Let t_1 be a subtree of t, we denote by $\mathrm{t} \setminus \mathrm{t}_1$ the subtree of t resulting from t by deleting the subtree t_1. We denote by $\mathrm{Pos}(\mathrm{t})$ the set of positions of expressions $a \in A_E$ being leaves of t. As a measure of t we use the cardinality of $\mathrm{Pos}(\mathrm{t})$, we set $|\,\mathrm{t}\,| = |\,\mathrm{Pos}(\mathrm{t})|$. For a node ν in $\mathrm{T}(E)$, the regular $\mathbb{K}$-expression E_ν denotes the subexpression resulting from the node ν and $c(\nu)$ its null term. The Last tree $\mathrm{TL}(E)$ is a labeled copy of $\mathrm{T}(E)$, where an edge going from a node λ labeled "$\cdot$" to its left son λ_l is marked by $c(\lambda_l)$ and for each edge going from a node λ labeled "$*$" to its son λ_s is marked by $c(\lambda_s)^{\circledast}$. The node λ represents the polynomials

$$e(\lambda) \;=\; \sum_{x\in A_E} \pi(x,\lambda)x$$

Thus we have

$$e(\lambda) \;=\; \mathrm{Last}(E_\lambda)$$

The First tree $\mathrm{TF}(E)$ is computed in a similar way, by marking an edge going from a node φ labeled "$\cdot$" to its right son φ_r by $c(\varphi_l)$ and a edge going from a node φ to its son φ_s is marked by $c(\varphi_s)^{\circledast}$. The node φ represents the polynomials

$$e(\varphi) \quad = \quad \sum_{x \in A_E} \pi(\varphi, x)x$$

Thus we have

$$e(\varphi) \quad = \quad \mathrm{First}(E_\varphi)$$

Any edge not marked will be marked by 1 and the marking of these edges is implicit.
The two trees are connected as follows. if a node λ of $\mathrm{TL}(E)$ is labeled by "$\cdot$", its left son λ_l is linked to the right son φ_r of the corresponding node φ in $\mathrm{TF}(E)$. If a node in $\mathrm{TL}(E)$ labeled "$*$" its son node is linked to its corresponding node in $\mathrm{TF}(E)$. Such links are called *follow links*. The set of follow links is denoted by Δ. We denote by Δ_x the set of follow links associated to the position x. That is: $\Delta_x = \{(\lambda, \varphi) \in \Delta \mid x \preceq \lambda\}$.

Proposition 4 *Let E be a regular $\mathbb{K}$-expression and $x \in \mathrm{Pos}(E)$. Then*

$$\mathrm{Follow}(x, E) \quad = \quad \sum_{(\lambda,\varphi)\in\Delta_x} \pi(x, \lambda)e(\varphi) \tag{8}$$

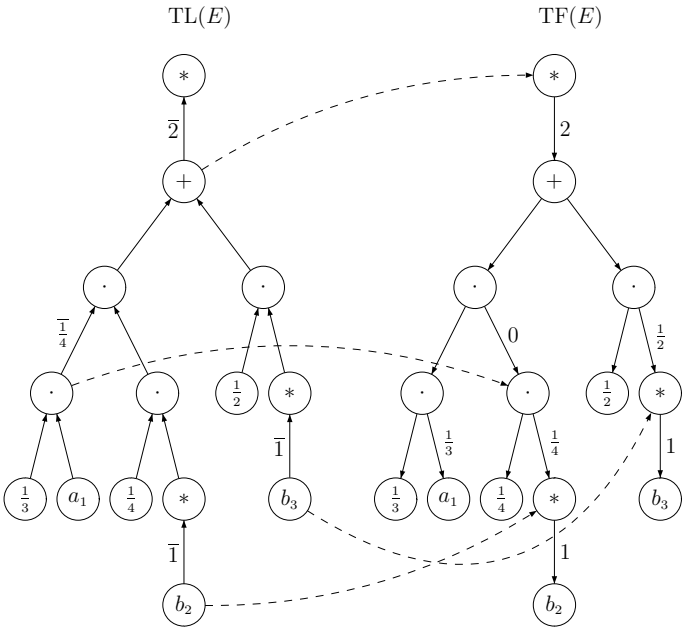


Figure 3: The $\mathbb{K}\mathcal{ZPC}$-structure associated to the expression $E = (\frac{1}{3}a\frac{1}{4}b^* + \frac{1}{2}b^*)^*$.

6 A CFP computation

Now we describe a procedure which allows us to produce a CFP system that yields to a CFP automaton with $O(n\log^2(n))$ transitions and $O(n)$ states. This procedure is a generalization of J. Hromkovič et *al.* one.

We introduce the function $f : \mathrm{TL}(E') \to \mathrm{TF}(E') \cup \{\perp\}$ defined by:

$$f(\lambda) = \begin{cases} \varphi & \text{if } (\lambda, \varphi) \in \Delta_x, \\ \perp & \text{otherwise.} \end{cases}$$

where $\perp$ denotes an artificial node such that $\pi(x, \perp) = 0$.
In order to simplify the writings in the following parts, we extend the polynomial Follow to the nodes of the tree $\mathrm{TL}(E')$ as follows:
Lets $\lambda_1 \preceq \lambda_2$ be two nodes from the tree $\mathrm{TL}(E')$, we denote by

$$\mathrm{Follow}(\lambda_1, \lambda_2) = \sum_{\lambda_1 \preceq \lambda \prec \lambda_2} \pi(\lambda_1, \lambda) e(f(\lambda))$$

Notice that for a polynomial $P(?)$ and a subtree t of $\mathrm{T}(E')$, the restriction of $P(?)$ to t, denoted by $P_{\mathrm{t}}(?)$ is defined as

$$P_{\mathrm{t}}(?) = \sum_{x \in \mathrm{Pos(t)}} (P(?), x)x$$

Consider a subtree t of $\mathrm{T}(E')$ and a position $x \in \mathrm{Pos(t)} \setminus \{\sharp\}$. We denote by tf (respectively tl) the labeled copy of t in $\mathrm{TF}(E')$ (respectively in $\mathrm{TL}(E')$). The procedure recursively computes a particular CFP system. It is defined as
If $|\,\mathrm{t}\,| > 1$:
Divide t into two subtrees t_1 and t_2 according to the following rules: $\frac{|\,\mathrm{t}\,|}{3} \leq |\,\mathrm{t}_1\,| \leq \frac{2|\,\mathrm{t}\,|}{3}$ and let $\mathrm{t}_2 = \mathrm{t} \setminus \mathrm{t}_1$ with λ_1 be the root of the subtree tl_1 and φ_1 its corresponding node in tf_1. Let

$$\begin{aligned} k_1(x) &= \pi(x, \lambda_1), \\ P_1 &= \mathrm{Follow}_{\mathrm{t}_2}(\lambda_1, E'), \\ k_2(x) &= \sum_{x \preceq \lambda \prec E'} \pi(x, \lambda)\pi(f(\lambda), \varphi_1) \\ P_2 &= e(\varphi_1). \end{aligned}$$

Let us suppose that $x \in \mathrm{Pos}(\mathrm{t}_1)$. If $\pi(x, \lambda_1) = 0$, that is in the tree tl, one cannot reach the follow links which are in above the root of the tree tl_1 i.e. From Formula 12, the positions of the tree t_2 not occur in the polynomial $\mathrm{Follow}_{\mathrm{t}}(x, E')$. In this case, one has $\mathrm{dec}(x, \mathrm{t}) = \mathrm{dec}(x, \mathrm{t}_1)$. In the contrary case, we write $\mathrm{Follow}_{\mathrm{t}}(x, E')$ as a sum of polynomials having disjoined supports. We can write $\mathrm{Follow}_{\mathrm{t}}(x, E') = \mathrm{Follow}_{\mathrm{t}_1}(x, E') + \mathrm{Follow}_{\mathrm{t}_2}(x, E')$ and we search to compute the polynomial $\mathrm{Follow}_{\mathrm{t}_2}(x, E')$. Let us show that $\mathrm{Follow}_{\mathrm{t}_2}(x, E') = k_1(x) \times P_1$ and $\mathrm{dec}(x, \mathrm{t}) = \mathrm{dec}(x, \mathrm{t}_1) \cup \{k_1(x) \times P_1\}$. From Formula 12, one has:

$$\begin{aligned} \mathrm{Follow}_{\mathrm{t}_2}(x, E') &= \sum_{\substack{(\lambda, f(\lambda)) \in \Delta_x \\ \lambda \prec E'}} \pi(x, \lambda) e_{\mathrm{t}_2}(f(\lambda)) \\ &= \sum_{x \preceq \lambda \prec E'} \pi(x, \lambda) e_{\mathrm{t}_2}(f(\lambda)) \\ &= \pi(x, \lambda_1) \times \sum_{\lambda_1 \preceq \lambda \prec E'} \pi(\lambda_1, \lambda) e_{\mathrm{t}_2}(f(\lambda)). \end{aligned}$$

Notice that the polynomial $P_1 = \sum_{\lambda_1 \preceq \lambda \prec E'} \pi(\lambda_1, \lambda) e_{\mathrm{t}_2}(f(\lambda))$ is independent of the positions of the tree t_1 and the polynomials $\mathrm{Follow}_{\mathrm{t}_2}(x, E')$ with $x \in \mathrm{Pos}(\mathrm{t}_1)$ are equal except a coefficient $k_1(x) = \pi(x, \lambda_1)$. Thus we get:

$$\mathrm{dec}(x, \mathrm{t}) \;=\; \begin{cases} \mathrm{dec}(x, \mathrm{t}_1) & \text{if } \pi(x, \lambda_1) = 0, \\ \mathrm{dec}(x, \mathrm{t}_1) \cup \{k_1(x) \times P_1\} & \text{otherwise.} \end{cases}$$

Let us suppose that $x \in \mathrm{Pos}(\mathrm{t}_2)$. Similarly, if $\mathrm{Follow}_{\mathrm{t}_1}(x, E') = 0$, one has $\mathrm{dec}(x, \mathrm{t}) = \mathrm{dec}(x, \mathrm{t}_2)$. In the contrary case, $\mathrm{dec}(x, \mathrm{t}) = \mathrm{dec}(x, \mathrm{t}_2) \cup \{\mathrm{Follow}_{\mathrm{t}_1}(x, E')\}$. Let us show that $\mathrm{Follow}_{\mathrm{t}_1}(x, E') = k_2(x) \times P_2$. Lets F be the root node of t_2 and φ_1 be the root node of tf_1. From Formula 12 one has:

$$\begin{aligned} \mathrm{Follow}_{\mathrm{t}_1}(x, E') &= \sum_{\substack{(\lambda, f(\lambda)) \in \Delta_x \\ \lambda \prec E'}} \pi(x, \lambda) e_{\mathrm{t}_1}(f(\lambda)) \\ &= \sum_{x \preceq \lambda \prec E'} \pi(x, \lambda) e_{\mathrm{t}_1}(f(\lambda)) \\ &= \sum_{x \preceq \lambda \prec E'} \pi(x, \lambda) \pi(f(\lambda), \varphi_1) e_{\mathrm{t}_1}(\varphi_1) \\ &= \sum_{x \preceq \lambda \prec E'} \pi(x, \lambda) \pi(f(\lambda), \varphi_1) \times e(\varphi_1). \end{aligned}$$

Let $k_2(x) = \sum_{x \preceq \lambda \prec E'} \pi(x, \lambda) \pi(f(\lambda), \varphi_1)$ and $P_2 = e(\varphi_1)$, we get

$$\mathrm{dec}(x, \mathrm{t}) \;=\; \begin{cases} \mathrm{dec}(x, \mathrm{t}_2) & \text{if } \mathrm{Follow}_{\mathrm{t}_1}(x, E') = 0, \\ \mathrm{dec}(x, \mathrm{t}_2) \cup \{k_2(x) \times P_2\} & \text{otherwise.} \end{cases}$$

If $|\,\mathrm{t}\,| = 1$: Using the same idea of the previous cases, we get

$$\mathrm{dec}(x, \mathrm{t}) \;=\; \{(P_0, x) \times x\}$$

With $P_0 = \sum_{x \preceq \lambda \prec E'} (\pi(x, \lambda) \pi(f(\lambda), x)) x$.

Proposition 5

$$\mathrm{Follow}_{\mathrm{t}}(x, E') = \sum_{k \times C \in \mathrm{dec}(x, t)} k \times C.$$

Proof. By induction on the size of the regular $\mathbb{K}$-expression E.
If $|\,\mathrm{t}\,| = 1$, one has $\mathrm{Follow}_{\mathrm{t}}(x, E') = P_0 = (P_0, x)x$. We suppose that the property is true for all subtree t' such that $1 < |\,\mathrm{t}'\,| < |\,\mathrm{t}\,|$ and we prove that it is true for t. We have

$$\begin{aligned} \mathrm{Follow}_{\mathrm{t}}(x, E') &= \mathrm{Follow}_{\mathrm{t}_1}(x, E') + \mathrm{Follow}_{\mathrm{t}_2}(x, E') \\ &\overset{\text{Ind.Hyp.}}{=} \sum_{k \times C \in \mathrm{dec}(x, \mathrm{t}_1)} k \times C + \sum_{k \times C \in \mathrm{dec}(x, \mathrm{t}_2)} k \times C \end{aligned}$$

One has

$$\mathrm{dec}(x,\mathrm{t}) \;=\; \begin{cases} \mathrm{dec}(x,\mathrm{t}_1) & \text{if } x \in \mathrm{Pos}(\mathrm{t}_1) \text{ and } k_1(x)=0,\\ \mathrm{dec}(x,\mathrm{t}_1)\cup\{k_1(x)\times P_1\} & \text{if } x \in \mathrm{Pos}(\mathrm{t}_1) \text{ and } k_1(x)\neq 0,\\ \mathrm{dec}(x,\mathrm{t}_2) & \text{if } x \in \mathrm{Pos}(\mathrm{t}_2) \text{ and } k_2(x)=0,\\ \mathrm{dec}(x,\mathrm{t}_2)\cup\{k_2(x)\times P_2\} & \text{if } x \in \mathrm{Pos}(\mathrm{t}_2) \text{ and } k_2(x)\neq 0. \end{cases}$$

So, we can conclude that $\mathrm{Follow}_{\mathrm{t}}(x,E') = \sum_{k\times C\in \mathrm{dec}(x,\mathrm{t})} k\times C$. ∎

Thus, we get a CFP family $\mathcal{F}_{\mathrm{t}}$ restricted to t

$$\mathcal{F}_{\mathrm{t}} = \bigcup_{x\in \mathrm{Pos}(\mathrm{t})\setminus\{\sharp\}} \{C | \exists k\in\mathbb{K} \text{ such that } k\times C\in \mathrm{dec}(x,\mathrm{t})\}$$

The following example illustrates the different stages of the recursive procedure.

Example 3 *Let* $E=(((a+\frac{1}{3})+(b+\frac{1}{6}))(b+1))^*$.

Step 1:
$|\,\mathrm{T}(E')| = 4$. *We divide the tree* $\mathrm{t}=\mathrm{T}(E')$ *into two subtrees* t_1 *and* t_2 *such that* t_1 *be the subtree representing the subexpression* $(a_1+\frac{1}{3})+(b_2+\frac{1}{6})$ *and* $\mathrm{t}_2=\mathrm{t}\setminus\mathrm{t}_1$. *In this case one has*

$$\begin{aligned} \mathrm{dec}(a_1,\mathrm{t}) &= \mathrm{dec}(a_1,\mathrm{t}_1)\cup\{1\times(2b_3+2\sharp)\},\\ \mathrm{dec}(b_2,\mathrm{t}) &= \mathrm{dec}(b_2,\mathrm{t}_1)\cup\{1\times(2b_3+2\sharp)\},\\ \mathrm{dec}(b_3,\mathrm{t}) &= \mathrm{dec}(b_3,\mathrm{t}_2)\cup\{1\times(2a_1+2b_2)\}. \end{aligned}$$

Step 2:
Recursively, we divide the subtree t_1 *into two subtrees* t_{1_1} *(the subtree representing the subexpression* $a_1+\frac{1}{3}$ *and* $\mathrm{t}_{1_2}=\mathrm{t}_1\setminus\mathrm{t}_{1_1}$*). Similarly, we divide the subtree* t_2 *into two subtrees* t_{2_1} *(the subtree representing the subexpression* b_3+1 *and* $\mathrm{t}_{2_2}=\mathrm{t}_2\setminus\mathrm{t}_{2_1}$*). Thus we get*

$$\begin{aligned} \mathrm{dec}(a_1,\mathrm{t}) &= \mathrm{dec}(a_1,\mathrm{t}_{1_1})\cup\{2\times(b_2)\}\cup\{1\times(2b_3+2\sharp)\},\\ \mathrm{dec}(b_2,\mathrm{t}) &= \mathrm{dec}(b_2,\mathrm{t}_{1_2})\cup\{2\times(a_1)\}\cup\{1\times(2b_3+2\sharp)\},\\ \mathrm{dec}(b_3,\mathrm{t}) &= \mathrm{dec}(b_3,\mathrm{t}_{2_1})\cup\{2\times(\sharp)\}\cup\{1\times(2a_1+2b_2)\}. \end{aligned}$$

Step 3:
One has $|\,\mathrm{t}_{1_1}\,| = |\,\mathrm{t}_{1_2}\,| = |\,\mathrm{t}_{2_1}\,| = 1$*, thus:*

$$\begin{aligned} \mathrm{dec}(a_1,\mathrm{t}) &= \{2\times(a_1)\}\cup\{2\times(b_2)\}\cup\{1\times(2b_3+2\sharp)\},\\ \mathrm{dec}(b_2,\mathrm{t}) &= \{2\times(b_2)\}\cup\{2\times(a_1)\}\cup\{1\times(2b_3+2\sharp)\},\\ \mathrm{dec}(b_3,\mathrm{t}) &= \{2\times(b_3)\}\cup\{2\times(\sharp)\}\cup\{1\times(2a_1+2b_2)\}. \end{aligned}$$

Thus the resulting CFP family $\mathcal{F}_E$ *associated with* $S(E)$ *is*

$$\mathcal{F}_E = \{2a_1+2b_2+b_3+2\sharp\}\cup\{(a_1),(b_2),(b_3),(\sharp),(2b_3+2\sharp)\}$$

Using this procedure, we can prove in similar way as the Boolean case the following Lemma.

Lemma 2 *[11] For $\mathcal{F}_E = \{\mathrm{First}(E')\} \cup \mathcal{F}_{E'}$ the following holds:*

1. $|\mathcal{F}_E| = O(|E|)$,
2. $\sum_{C \in \mathcal{F}_E} |C| = O(|E| \log |E|)$,
3. $|\operatorname{dec}(x, E')| = O(\log |E|)$.

7 Efficient CFP system computation

Algorithm 1 summarizes the CFP system computation. It has a cubic time complexity. Indeed the computation of polynomials P_0, P_1, and P_3 is quadratic on the size of E independently of the size of subtree t.

Algorithm 1 A CFP system computation.

CFP-system(t)
Begin
1. F:=root_of(tl)
2. **if** ($|\,\mathrm{t}\,|$=1)
3. **then**
4. Let $x \in \mathrm{Pos}(\mathrm{t}) \setminus \{\#\}$
5. P_0:=$\mathrm{Follow}_{\mathrm{t}}(x, E')$
6. $\operatorname{dec}(x)$:=$\{(P_0, x) \times x\}$
7. **else**
8. decompose t into t_1 and $\mathrm{t}_2 = \mathrm{t} \setminus \mathrm{t}_1$ such that $\frac{|\mathrm{t}|}{3} \leq |\,\mathrm{t}_1\,| \leq \frac{2|\mathrm{t}|}{3}$
10. λ_1:=root_of(tl_1)
11. φ_1:=root_of(tf_1)
12. P_1:=$\mathrm{Follow}_{\mathrm{t}_2}(\lambda_1, E')$
13. P_2:=$e(\varphi_1)$
14. P_3:=$\mathrm{Follow}_{\mathrm{t}_1}(\varphi_1, E')$
15. **for** $x \in \mathrm{Pos}(\mathrm{t}_1) \setminus \{\#\}$ **do**
16. $\operatorname{dec}(x)$:=$\operatorname{dec}(x) \cup \{\pi(x, \lambda_1) \times P_1\}$
17. **for** $x \in \mathrm{Pos}(\mathrm{t}_2) \setminus \{\#\}$ **do**
18. $\operatorname{dec}(x)$:=$\operatorname{dec}(x) \cup \{(P_3, x) \times P_2\}$
19. **endif**
20. CFP-system(t_1)
21. CFP-system(t_2)
End

In the following section, we first show how these polynomials can be computed in linear time on the size of E, next we present a refinement of Algorithm 1 in order to compute these polynomials in linear time on the size of the subtree t.

7.1 Efficient Follow computation

Let t be a subtree deduced from a decomposition of $\mathrm{T}(E')$. Let λ_1 be the root of tl, and let λ_2 a node in tl such that $\lambda_1 \preccurlyeq \lambda_2$. We have

$$\mathrm{Follow}(\lambda_1, \lambda_2) \;=\; \sum_{\lambda_1 \preccurlyeq \lambda \prec \lambda_2} \pi(\lambda_1, \lambda) e(f(\lambda))$$

The supports of polynomials $(e(f(\lambda)))_{\lambda_1 \preccurlyeq \lambda \prec \lambda_2}$ are not disjoints. So the computation of $\mathrm{Follow}_{\mathrm{t}}(\lambda_1, \lambda_2)$ according to Formula (1) requires a quadratic time on the size of t.

Let λ' and λ'' be to nodes such that $\lambda_1 \preccurlyeq \lambda' \prec \lambda'' \preccurlyeq \lambda_2$ and $\mathrm{supp}(e(f(\lambda')) \cap \mathrm{supp}(e(f(\lambda)) \neq \varnothing$.

In this case we have

$$e(f(\lambda'')) \;=\; \pi(f(\lambda''), f(\lambda')) e(f(\lambda')) + e(f(\lambda'') - f(\lambda'))$$

where $e(f(\lambda'') - f(\lambda'))$ denote the polynomials induced by the subtree $\mathrm{t}_{\lambda''} \setminus \mathrm{t}_{\lambda'}$. So the polynomial

$$\pi(\lambda_1, \lambda') e(f(\lambda')) + \pi(\lambda_1, \lambda'') e(f(\lambda''))$$

can be written

$$[\pi(\lambda_1, \lambda') + \pi(\lambda_1, \lambda'') \pi(f(\lambda''), f(\lambda')\,] e(f(\lambda')) + \pi(\lambda_1, \lambda'') e(f(\lambda'') - f(\lambda'))$$

Here, supports of polynomials $e(f(\lambda'))$ and $e(f(\lambda'') - f(\lambda'))$ are disjoints. Algorithm 2 is based on this Formula. The call Follow(λ_2,$\overline{1}$,λ_2), computes the polynomial $\mathrm{Follow}(\lambda_1, \lambda_2)$ in linear time on the size of t. Indeed, each node in tl_2 is treated once.

Notice as polynomials Follow are ordered on $\mathrm{positions}$, the restriction of $\mathrm{Follow}_{\mathrm{t}'}(\lambda_1, \lambda_2)$ to some subtree t' of t needs a linear time on the size of t.

In a similar way, the polynomial $\mathrm{Follow}(\varphi_1, \varphi)$ is computed in linear time of the size of t where φ is the root of tf and φ_1 is a node in tf.

Now polynomials P_0, P_1, and P_3 are computed in linear time on the size of E. So Algorithm 1 runs in quadratic time on the size of E. However it do some redundant computations. In the following section we give a refinement of Algorithm 1, in which the P_0, P_1, and P_3 are computed in linear time on the size of subtree t.

Lemma 3 *Let E be a regular $\mathbb{K}$-expression. Let* t,t' *be subtrees of* $\mathrm{T}(E')$. *Let* λ_1 *be the root of* tl, *and let* λ_2 *a node in* tl *such that* $\lambda_1 \preccurlyeq \lambda_2$. *Then the polynomials* $\mathrm{Follow}_{\mathrm{t}'}(\lambda_1, \lambda_2)$ *can be computed in* $O(\max(|\,\mathrm{t}\,|, |\,\mathrm{t}'\,|)$.

7.2 Redundant computations

At each call of CFP-system procedure we compute the following polynomials $\mathrm{Follow}(\lambda_1, E')$, $\mathrm{Follow}(\varphi_1, E')$. We have

$$\mathrm{Follow}(\lambda_1, E') \;=\; \mathrm{Follow}(\lambda_1, \lambda) + \pi(\lambda_1, \lambda)\,\mathrm{Follow}(\lambda, E')$$

Algorithm 2 Follow computation.

Follow(λ_1,coef,λ_2) : polynomials
// λ_1 and λ_2 are two nodes s.t. $\lambda_1 \preceq \lambda_2$
Begin
1. coef:=$\pi(\lambda_1, \lambda_2) \oplus$ coef $\otimes \pi$(father($f(\lambda_2)$), $f(\lambda_2)$)
2. **case** (arity($f(\lambda_2)$) **of**
3. 0 :
4. **If** ((coef $\neq 0$) **and** ($f(\lambda_2)$ is a position))
5. **then**
6. **return**(coef $f(\lambda_2)$)
7. **else**
8. **return**(0)
9. **endif**
10. 1 :
11. **return**(Follow(λ_1,coef,son($f(\lambda_2)$)))
12. 2 :
13. **return**(Follow(λ_1,coef,leftson($f(\lambda_2)$))+Follow(λ_1,coef,rightson($f(\lambda_2)$)))
14. **endcase**

End

where λ and φ are respectively the root of tl and tf and λ_1 and φ_1 are respectively the root of tl_1 and tf_1.

As $\mathrm{Follow}(\lambda, E')$ is computed before $\mathrm{Follow}(\lambda_1, E')$ we can store it in some variable and use it when computing $\mathrm{Follow}(\lambda_1, E')$. Denote by $R(\mathrm{t}) = \mathrm{Follow}(\lambda, E')$. Then for the subtree t_1 of t we get

$$\mathrm{Follow}(\lambda_1, E') \quad = \quad \mathrm{Follow}(\lambda_1, \lambda) + \pi(\lambda_1, \lambda) R(\mathrm{t})$$

Thus $R(\mathrm{t})$ can be inductively computed as follows

$$R(\mathrm{t}_1) \quad = \quad \pi(\lambda_1, \lambda) R(\mathrm{t})$$

and as the root of t is the same as t_2 we get

$$R(\mathrm{t}_2) \quad = \quad R(\mathrm{t})$$

In a similar way $\mathrm{Follow}(\varphi_1, E')$ is computed. Denote by $S(\mathrm{t}) = \mathrm{Follow}(\varphi, E')$. So we get

$$\mathrm{Follow}(\varphi_1, E') \quad = \quad \mathrm{Follow}(\varphi_1, \varphi) + \pi(\varphi_1, \varphi) S(\mathrm{t})$$

where

$$S(\mathrm{t}_1) \quad = \quad \pi(\varphi_1, \varphi) S(\mathrm{t})$$

and

$$S(\mathrm{t}_2) \;=\; S(\mathrm{t})$$

It is obvious that if $R_{\mathrm{t}_2}(\mathrm{t})$ and $S_{\mathrm{t}_2}(\mathrm{t})$ are computed then $\mathrm{Follow}_{\mathrm{t}_2}(\lambda_2, E')$ and $\mathrm{Follow}_{\mathrm{t}_2}(\varphi_1, E')$ can be computed in time $O(|\,\mathrm{t}\,|)$. Finlay Algorithm 3 runs in $O(|E|\log|E|)$ time.

Algorithm 3 A CFP system computation.

CFP-system(t,R,S)
Begin
1. F:=root_of(tl)
2. **if** (|t|=1)
3. **then**
4. Let $x \in \mathrm{Pos}(\mathrm{t}) \setminus \{\#\}$
5. P_0:=$\mathrm{Follow}_{\mathrm{t}}(x, F) + R$
6. $\mathrm{dec}(x)$:=$\{(P_0, x) \times x\}$
7. **else**
8. decompose t into t_1 and $\mathrm{t}_2 = \mathrm{t} \setminus \mathrm{t}_1$ such that $\frac{|\,\mathrm{t}\,|}{3} \leq |\,\mathrm{t}_1\,| \leq \frac{2|\,\mathrm{t}\,|}{3}$
10. λ_1:=root_of(tl_1)
11. φ_1:=root_of(tf_1)
12. P_1:=$(\mathrm{Follow}(\lambda_1, F) + R)_{\mathrm{t}_2}$
13. P_2:=$e(\varphi_1)$
14. P_3:=$(\mathrm{Follow}(\varphi_1, F) + S)_{\mathrm{t}_1}$
15. **for** $x \in \mathrm{Pos}(\mathrm{t}_1) \setminus \{\#\}$ **do**
16. $\mathrm{dec}(x)$:=$\mathrm{dec}(x) \cup \{\pi(x, \lambda_1) \times P_1\}$
17. **for** $x \in \mathrm{Pos}(\mathrm{t}_2) \setminus \{\#\}$ **do**
18. $\mathrm{dec}(x)$:=$\mathrm{dec}(x) \cup \{(P_3, x) \times P_2\}$
19. **endif**
20. CFP-system($\mathrm{t}_1, (\pi(\lambda_1, F)R + \mathrm{Follow}(\lambda_1, F))_{\mathrm{t}_2}, (\pi(\varphi_1, \varphi)S + \mathrm{Follow}(\varphi_1, F))_{\mathrm{t}_2}$)
21. CFP-system(t_2,R,S)
End

Lemma 4 *Let E be a regular $\mathbb{K}$-expression. The CFP system associated to E can be computed in $O(|E|\log|E|)$ time.*

8 CFP automaton computation

In the previous section, we have given an algorithm that compute the CFP system. So the set of states of our automaton is computed. It remains to construct the set of transitions. Algorithm 4 computes in a first stage the set of transitions over $A_{\overline{E}}$, and in the second stage deduces the set of transitions over the alphabet A_E.

Let us now describe this algorithm. Let $C = \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_l x_l$ be a state in $\mathcal{F}_E$. Let $\beta_i x_i$ be a monomials in C. From the definition of $\mathcal{A}_{S(E)}$, if $\mathrm{dec}(x_i, E') = \{k_1 \times C_1, k_2 \times$

Algorithm 4 Transitions computation.

Transitions-from(C)
Begin
1. // Transitions over $A_{\overline{E}}$
1. **for** $C \in \mathcal{F}_E$ **do**
2. // $C = \beta_1 x_1 + \ldots + \beta_l x_l$
3. **for** $x \in \mathrm{supp}(C)$ **do**
4. // $\mathrm{supp}(C) = \{x_1, x_2, \ldots, x_l\}$
5. **for** $k \times C' \in \mathrm{dec}(x, E')$ **do**
6. // $\mathrm{dec}(x, E') = \{k_1 \times C_1, k_2 \times C_2, \ldots, k_m \times C_m\}$
7. Create a transition labelled $((C, x) \otimes k)x$ from C to C'
8. // Transition over A_E
9. **for** $C \in \mathcal{F}_E$ **do**
10. Let $q^+(C) = \{\alpha_1 x_1 C_1, \alpha_2 x_2 C_2, \ldots, \alpha_r x_r C_r\}$
11. // $q^+(C)$ are the set of transitions going from the state C
12. **for** $a \in A_E$ **do**
13. **for** $\alpha x C' \in q^+(C)$ **do**
14. $p(a, C')$=0
15. **for** $\alpha x C' \in q^+(C)$ **do**
16. $p(h(x), C')$:=$p(h(x), C') + \alpha$
End

$C_2, \cdots, k_m \times C_m\}$, we must create a transition from state C to each C_j $1 \leqslant j \leqslant m$, labelled $\beta_i k_j x_i$.

Now transitions are labelled by letter in $A_{\overline{E}}$. So in the second stage of the algorithm, we replace each symbol $x_i \in A_{\overline{E}}$ by $h(x_i)$. Next if there is two transitions from a state C to a state C' labelled respectively $\mathrm{kh}(x)$ and $k'h(x')$ where $h(x) = h(x')$, we merge these two transitions into a unique one having $(k + k')h(x)$ as label.

Finlay, as $\sum_{C \in \mathcal{F}_E} |C| = O(|E| \log |E|)$ and $|\mathrm{dec}(x, E')| = O(\log |E|)$, lines $1-7$ and lines $9-16$ of procedure transition() are achieved in $O(|E| \log |E|^2)$ times.

Theorem 3 *Let E be a regular $\mathbb{K}$-expression. The CFP automaton $\mathcal{A}_{S(E)}$ can be computed in $O(|E| \log^2(|E|))$ time.*

References

[1] V. Antimirov, *Partial derivatives of regular expressions and finite automaton constructions*, Theoret. Comput. Sci. **155** (1996), 291-319.

[2] Brüggemann-Klein A., *Regular Expressions into Finite Automata*, Theoret. Comput. Sci., 120, 197–213, 1993.

[3] J. Berstel and C. Reutenauer, *Rational series and their languages*, Springer-Verlag, Berlin, (1988).

[4] P. Caron and M. Flouret, *Glushkov construction for multiplicities*, In: S. Yu, A. Paun(eds.), Proc. 5th Int. Conf. on Implementations and Applications of Automata (CIAA). Lecture Notes in Computer Science **2088**, Springer-Verlag, (2001), 67-79.

[5] J.-M. Champarnaud, E. Laugerotte, F. Ouardi, and D. Ziadi, *From Regular Weighted Expressions to finite automata*, accepted in IJFCS, (2003).

[6] P. Caron and D. Ziadi, *Characterization of Glushkov Automata*, Theoret. Comp. Sc., **231**, (2000), 75-90.

[7] J.-M. Champarnaud and D. Ziadi, *Computing the equation automaton of regular expression in $O(s^2)$ space and time*, in CPM 2001, Combinatorial Pattern Matching, Lecture Notes in Computer Science, A. Amir and G. M. Landau eds., Springer-Verlag, **2089** (2001), 157-168.

[8] V.-M. Glushkov, *The abstract theory of automata*, Russian Mathematical Surveys, **16** (1961), 1-53.

[9] C. Hagenah and A. Muscholl, *Computing ε-Free NFA from Regular Expression in $O(n\log^2(n))$ Time*, Theor. Inform. Appl. 34 (4) (2000), 257-277.

[10] U. Hebisch and H. J. Weinert, *Semirings: algebraic theory and applications in computer science*, World Scientific, Singapore, (1993).

[11] U. Hromkovič, J. Seibert, and T Wilke, *Translating regular expressions into small ε-free nondeterministic finite automata*, J. Comput. System Sci. **62** (4), (2001), 565-588.

[12] Ilie L. and Yu S., *Algorithms for computing small NFAs*, In: K. Diks, W. Rytter (Eds.), Proc 27th MFCS, Warszawa, 2002, Lect. Notes in Comp. Sci., (Springer, Berlin, 2002), 328-340.

[13] W. Kuich and J. Salomaa, *Semirings, automata, languages.* Springer-Verlag, Berlin, (1986).

[14] S. Lombardy and J. Sakarovitch, *Derivatives of regular expression with multiplicity*, Proc. of MFCS 2002, Lect. Notes in Comp. Sci., 2420 (Springer, 2002), 471-482.

[15] R. F. McNaughton and H. Yamada, *Regular expressions and state graphs for automata*, IEEE Tans. Electronic Comput. **9** (1960), 39-47.

[16] M. P. Schützenberger, *On the definition of a family of automata*, Information and control, **6** (1961), 245-270.

[17] D. Ziadi, J.-L. Ponty and J.-M. Champarnaud, *Passage d'une expression rationnelle à un automate fini non-déterministe* , Bull. Bel. Math. Soc., **4** (1997), 177-203.

[18] D. Ziadi, *Quelques aspects théoriques et algorithmiques des automates*, Thèse d'habilitation à diriger des recherches, Université de Rouen, (2002).

Compact Representation of a Set of String-Classes

A. Fatholahzadeh
Supélec - Campus de Metz
2, rue Édouard Belin, 57078 Metz, France.
Email: Abolfazl.Fatholahzadeh@supelec.fr

Abstract

This paper introduces a new method for compact representation associated with a set of finite string-classes, where here, for the sake of clarity, each class is characterized by a non empty common maximal suffix (CMS). The data structure for compact representation of our method is based on q+1 automata and one m-ary decision tree (mdt), where q stands for the number of CMSes. The last automaton (g_{q+1}) is for spelling out the reverse-CMSes, each CMS is pointed (by way of the mdt) to its corresponding automaton which makes the process of lookup for an input string easy and fast because the search is done in an appropriate and learned subspace rather than in a complete space which can be dramatically very large. Experiments done on several datasets confirm the effectiveness of our method from the time and space requirements of the computation.

1 Introduction

When turning to the literature on algorithms of stringogloy, formal language and finite automata, as far as we know, one surprisingly realizes that there is no work done on how to use advantageously the benefit of having a set of string-classes both for compact representation of a finite set of strings and fast lookup process.

Given an input language (L), if L has the characteristics of String-class, the essential idea is to form $q + 1$ automata and one m–ary decision tree (mdt), where q stands for the number of CMSes. The last automaton (g_{q+1}) is for spelling out the reverse-CMSes, each of which is pointed (by way of the mdt) to its corresponding automaton which makes the process of lookup for an input string easy and fast. In order to explain intuitively the benefit

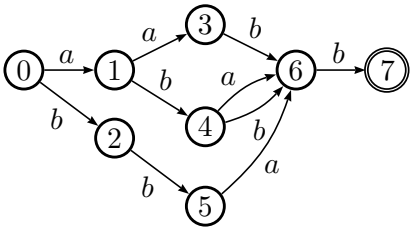


Figure 1: A (8,10) automaton of L = $\{aabb, abab, abbb, bbab\}$.

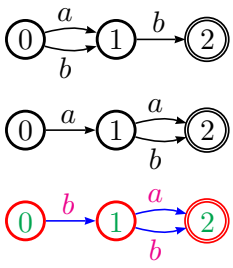


Figure 2: The 2 first automata are for prefix-sub-languages. The third one is for order-preserving reverse suffixes of L.

of our method, we give a very simple example.

Example 1: Although, the language $L = \{aabb, abab, abbb, bbab\}$ can be represented by a Deterministic Finite Automaton (DFA) of 8 states and 10 transitions (See Figure 1), because of the existence of CMSes, one can also divide L into two sub-prefix-languages: $\mathtt{P}_1 = \{ab, bb\}$ and $\mathtt{P}_2 = \{aa, ab\}$ along with $\mathtt{CMS}_1$ = "ab" and $\mathtt{CMS}_2$ = "bb", respectively. By forming an additional automaton associated with reverse suffix-sub-language of L, namely, $\{ba, bb\}$ and `learning` a decision tree indicating that "ba" (resp. "bb") is the reverse-suffix of first (resp. second) sub-prefix-language, then one has the ability to access directly to the right sub–space, namely, $\mathtt{g}_1$ (resp. $\mathtt{g}_2$) allowing a fast retrieval for variable-length strings and quick unsuccessful search determination.

The rest of paper argues in favor of our method and organized as follows. In Section 2, we give the basic definitions and notation. How one can learn the right sub-space is described in Section 3. Our algorithm for compact representation is presented in Section 4 along with a step-by-step trace of Example 1. How that representation can be used for fast lookup along with on demand (*i.e.,* just-in-time load the right sub-space in the main memory) is described in Section 5. Then, Section 6 compares our method with other techniques. The paper ends by giving the final words.

2 Preliminary

For more information on automata we refer the reader to *e.g.,* [8]. For a general reference on machine learning, the reader is referred to *e.g.,* [5]. For learning the values associated with a set of strings, we refer the reader to

our own method introduced in [3].

`String-Class:` A *string* (word) is a sequence of zero or more symbols from an alphabet Σ. The set of all strings over Σ is denoted by $\Sigma^\star$. The length of a string x is denoted by $|x|$. The *empty* string, the string of length zero is denoted by ϵ. The i^{th} symbol of a string x is denoted by $x[i]$. A string w is a suffix of x if $x = uyw$, for $w \in \Sigma^\star$. Similarly, u and y is a *prefix* and a *factor* of x, respectively. The concatenation of two strings x and y will be shown by xy. We write $x \oplus K$ (resp. $K \oplus x$) to denote the concatenation of one string x (resp. a set of finite strings) and a set of finite strings K (resp. one string).

The set of finite strings with a *common maximal suffix* (CMS) will be called a string-class (SC) *i.e.,* $SC = \{x = y_i z | i = 1, \ldots, n_1\}$. The reveres form of a CMS will be denoted by r_{cms}.

Example 2: The set {aabb, abbb} is a string-class whereas the set {aabb, abbb, abab} is not.

`Separate States:` A trie is essentially an ℓ–ary-tree whose nodes (states) are ℓ–places vectors with components corresponding to digits or characters. Each state on level h represents the set of all keys that begins with a certain sequence of h characters; the state specifies an ℓ–way branch, depending on the $(h+1)$st character. To avoid confusion between keys like "by" and "bye", usually a special end marker symbol, '#', is added to the end of all keys, so no prefix of a key can be a key itself. We write $g(r, a) = t$ to indicate that a transition labeled with $a \in \Sigma$ from r to t. The number of transitions outgoing from state s is defined as $OD(s)$ (*i.e.,* out degree). A state s with an $OD(s) = 0$ is called a terminal state. For $g(s, a) = t$ such that $OD(s) \geq 2$ and $OD(t) = 1$, state t is called a separate state (s_p) if and only if there is no state k such that $OD(k) \leq 2$ on the sequence of transitions from state t to the corresponding terminal. A set of separate states is denoted by S_p.

Example 3: Figure 3 shows a trie with one separate state, namely, state 6.

`Finite automaton:` A trie is used to form an acyclic finite-state automaton: a graph of the form $g = (Q, \Sigma, \delta, q_0, F)$ where Q is a finite set of states, q_0 is the start state, $F \subseteq Q$ is the accepting states. δ is a partial mapping $\delta : Q \times \Sigma \longrightarrow Q$ denoting *transition.* If $a \in \Sigma$, the notation $\delta(q, a) = \perp$ is used to mean that $\delta(q, a)$ is undefined. The extension of the partial δ mapping with $x \in \Sigma^\star$ is a function $\delta^\star : Q \times \Sigma^\star \longrightarrow Q$ and defined as follows:

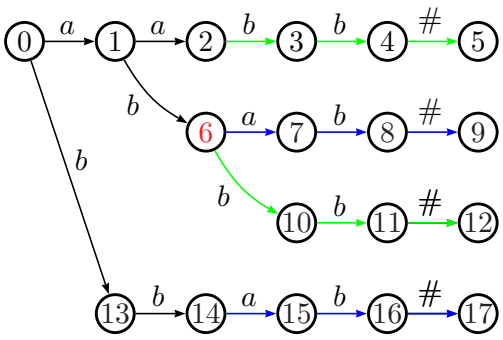


Figure 3: Trie for $\mathtt{L} = \{aabb, abab, abbb, bbab\}$

$$\delta^\star(\mathtt{q}, \varepsilon) = \mathtt{q}$$
$$\delta^\star(\mathtt{q}, \mathtt{ax}) = \begin{cases} \delta^\star(\delta(q,a),x) & \text{If } \delta(q,a) \neq \perp \\ \perp & \text{otherwise} \end{cases}$$

The property $\delta^\star$ allows fast retrieval for variable-length strings and quick unsuccessful search determination. The pessimistic time complexity of $\delta^\star$ is $\mathcal{O}(|\mathtt{x}|)$ *w.r.t.* a string $\mathtt{x}$. A finite automaton is said to be $(\mathtt{n_s}, \mathtt{n_t})$–automaton if $|\mathtt{Q}| = \mathtt{n_s}$ and $|\mathtt{E}| = \mathtt{n_t}$ where $\mathtt{E}$ denotes the set of the edges (transitions) of $\mathtt{g}$. An example of such graph is shown by Figure 1.

3 Learning right sub-space

How one can learn the right sub-space for being used in search process of user-inputs is the raison d'être of this section. The right sub-space means the answer to the following question:

> "Among $\mathtt{q}$ automata (*e.g.,* $\mathtt{g_1}$ and $\mathtt{g_2}$ of Figure 1) which one should be selected for being used in the lookup process?"

We answer to this question by learning the decision tree of the last automaton. That is to say, our idea is to form a reservoir of the following form: $\mathtt{f_{q+1}} = \{(\mathtt{rcms_1}, \mathtt{1}), \cdots, (\mathtt{rcms_q}, \mathtt{q})\}$ and using that form for discrimination purpose. More precisely, here, our task is selecting which `character(s)` of the input language of $\mathtt{f_{q+1}}$ *i.e.,* $\mathtt{rcms_1}, \cdots, \mathtt{rcms_q}$ is(are) the best for such a refinement.

Example 4: $\mathtt{f_3} = \{(\mathtt{ba}, \mathtt{1}), (\mathtt{bb}, \mathtt{2})\}$ is a form *w.r.t.* Example 1 our which can be refined as follows: $(\mathtt{rcms}[\mathtt{2}] = \text{`}\mathtt{a}' \wedge \mathtt{v} = \mathtt{1}) \vee (\mathtt{rcms}[\mathtt{2}] = \text{`}\mathtt{b}' \wedge \mathtt{v} = \mathtt{2})$ meaning that for an already recognized reverse-CMS (*i.e.,* spelled out using $\mathtt{g_3}$) select first or second automaton depending that the second character of

Table 1: **Left**: The profit trend of 10 restaurants where CP, HB and CK stand for competition, Hamburger and Chelo–Kabab (Iranian hum), respectively. **Right**: Its compressed form.

Age	CP	Type	Profit
old	no	CK	down
midlife	yes	CK	down
midlife	no	HB	up
old	no	HB	down
new	no	HB	up
new	no	CK	up
midlife	no	CK	up
new	yes	CK	up
midlife	no	HB	down
old	yes	CK	down

String	Value
onC	down
myC	down
mnH	up
onH	down
nnH	up
nnC	up
mnC	up
nyC	up
mnH	down
oyC	down

`rcms` be 'a' or 'b'.

`Illustration:` The framework for learning the output language of an input language is described in [3]. In this section, we only illustrate that method using an example. A decision tree (`dt`) is a direct acyclic graph of nodes and arcs. At each `node` a simple test is made; at the `leaves` a decision is made with respect to the class labels (values associated to a word in our case).

Example 5: The left part of Table 1 shows the data for 10 restaurants using four attributes. One can find out the attribute `age` is the best to be selected at first; this indicates that it is most likely that a decision can be made quickly if one first asks for the age of a restaurant. If the answer to this question is 'new' or 'old', then the profit can be predicted by 'up' or 'down', respectively. If the answer is 'midlife', then another question must be posed, about the presence of competition. After this answer is known, the profit trend can be determined.

Example 6: The right part of Table 1 is the compressed form of its counterpart *i.e.,* the strings of third first columns of each row of the left part of Table 1 is compressed in one string using the first character of each string (*e.g.,* "old" is transformed into 'o'). So, the decision tree concerning the selection of best characters for 10 strings can be built like the restaurant's example.

`Data structure of m-ary decision tree:` Usually, the decision tree is a binary one. However, for a compact space purpose, our data structure

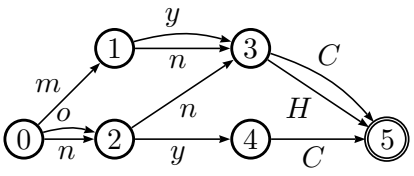


Figure 4: A (6,10) automaton for recognizing ten keys of the right part of Table 1.

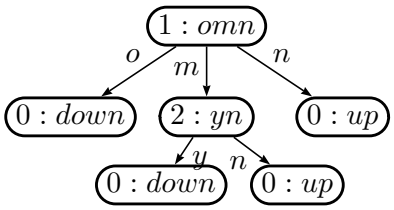


Figure 5: Learned m-ary decision tree of the right part of Table 1

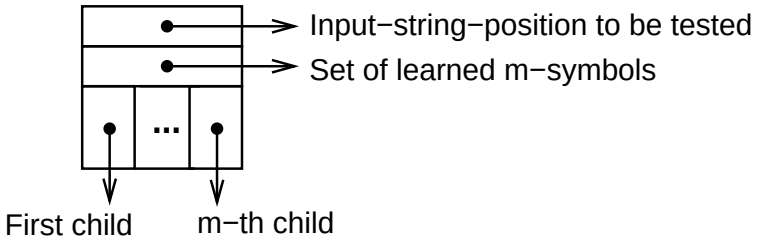


Figure 6: Node of `m`–ary decision tree.

is `m`-ary tree where `m` is the number of children *i.e.,* the length of the `best string` (the best learned characters of the learning process) at each node of the tree, except in the leaf one containing the output value. Figure 6 shows the data structure of a node of `m`-ary decision. The first field contains a nonnegative integer, say `i` for $\texttt{0} \leq \texttt{i} \leq \ell$ where ℓ denotes the length of the longest word(s) of the input language. If $\texttt{i} = \texttt{0}$ this means that the node is a leaf one, otherwise the node is an internal one (including the root node). The second filed represents either a best string or the output value. The third filed is `m`-pointers to other nodes, each indicating which node has to be followed in the tree when searching the output value.

`Searching value:` If a word has already been recognized key via an unlabeled automation, then how to use a decision tree for determining its value? This question is answered by the function SearchValue which works as follows: Given `x`, starting at the rode of the node, a simple test, namely $\texttt{x}[\texttt{i}] = \texttt{bs}[\texttt{i}]$, is made, where `i` and `bs` stand for the value placed in the first field and the best string of the second field, respectively. Depending on one test-equality which for sure will hold (because the word exists and we have already our best string for discrimination purpose.), $\texttt{i}^{\texttt{th}}$ pointer of the third filed will tell us which branch has to be followed. This process continues until we reach to a leaf node (asserted by 0 in the first field) providing the

output value.

Example 7: Figures 4 and 5 show an unlabeled automaton and a decision tree of ten keys mentioned in right part of Table 1. Given `x`, if it can be spelled out by that automaton, the outpout of `x` will be determined as follows: starting with the root-node of Figure 5, we have "1:omn" meaning that: (1) If `x[1]` = 'o' then gets the value (of `x`) by descending in the sub-tree of first child. Since that sub-tree has only one node - a leaf (asserted by zero) - then the value is "down"; (2) If `x[1]` = 'm', this time the value has to be selected using the sub-tree of the second child which itself depends on the value of second character ("2:yn") of `x` *i.e.,* the value is either "down" or "up"; and finally (3) If `x[1]` = 'n' the value is "up".

`Remark:` We will use above search process for determining the right sub-space called hereafter the function `SerachSubSpace` which will be used in our refinement algorithm described in the next Section.

4 Algorithm

Algorithm 1 shows the main function of our new method. The input of our algorithm is the user-reservoir (for short `R`) containing a collection of strings, where, each string, may be associated with one string (value) or more strings (alternative values).

---

func CompactRepresentation(R) // R = user-reservoir
  f ← ReduceReservoir(R); {Step 1, f = L ∪ O}
  TR ← FormTrie(L); S_p ← Separate States(TR); {Step 2}
  if $S_p = \emptyset$ **then**
    g ← CompactInputLanguage(TR); {Step 3: Form minimal DFA.}
    if O ≠ ∅ {*i.e.,* R has the output language.} **then**
      mdt ← LearnOutputLanguage(TR, O); {Step 4: Decision tree.}
    end if
  else
    Refine(TR, S_p, O); { Step 5: Refinement.}
  end if
cnuf

Algorithm 1: Compact Representation of String-Classes.

`Step 1:` If `R` contains alternative values, we assign to a word a unique ambiguity value-class (*e.g.,* 'Noun/Adj'), although a value-class represents

a set of alternative values that a given word can occur with. So, we may obtain a new reduced reservoir (noted by f).

Example 8: f = {(cabba, [xxxxx,xxyyx,xtzyx]), (cabca, [yzxxy,yzyyy])} is a reduced reservoir of R. The latter may be composed of five entries of two distinct words and five values.

Step 2: This stage will tell us whether the input language can be refined into a set of q sub–languages. This question is answered by the existence of the set of separate states in the trie associated with the input language. Figure 3 shows a trie with one separate state, namely state 6. This state will be used to refine the input language.

Step 3: If there is no a separate state in the trie, this means that the input language cannot be divided into suffix–sub-languages. In this case, we form a minimal DFA using (1) the trie of the second step and (2) our own work: In [2] two elegant algorithms for the incremental construction of minimal acyclic finite state automata and transducers from both sorted and unsorted data is described. We adapted their former one such that the length of the longest key be calculated for being used in the learning process of the decision tree.

Step 4: If the reservoir has the output language (*i.e.,* $O \neq \emptyset$) the decision tree associated with O will be performed by the function LearnOutputLanguage (Please refers to [3]).

Step 5: If the set of separate states is not empty, the function Refine will be called to form $q + 1$ graphs (automata) and one or $q + 1$ decision trees depending on $O = \emptyset$ or $O \neq \emptyset$, respectively. This work is done by the function Refine described below.

4.1 Refinement

Refinement is our method for learning how to divide user-reservoir into $q + 1$ input sub–languages? The formal description of the refinement method is shown in Algorithm 2.

Step 5-1: The first task of the refinement process is to collect the paths representing one or more CMSes. A CMS (or path) is just the sequence of transitions from state s_p to a terminal state (t). For each iteration of this step, the sequence from that current s_p to its corresponding t will be selected.

Example 9: State 6 of Figure 3 is a separate one. So the sequences from

that state to the terminal states (*i.e.,* 9 and 12) are selected as the CMSes. So, we have $\texttt{CMS}_1$ = "ab#" and $\texttt{CMS}_2$ = "bb#".

`Step 5-2:` The second stage of the refinement process is to determine the set of prefix-sub-languages. This is done as follows: for each CMS obtained in step 5-1, we use it to select other elements of the same prefix-sub-language.

Example 10: Given $\texttt{CMS}_1$ = "ab#", first we collect the current prefix into a set, say $\texttt{P}_1$. The first prefix *w.r.t.* $\texttt{CMS}_1$ is calculated from the path "abab". So $\texttt{P}_1 = \{ab\}$. Then, we look for other possible paths *w.r.t.* $\texttt{CMS}_1$. If any, then we update the current prefix–sub–language, namely $\texttt{P}_1$. There is only one path (*i.e.,* "bbab") having common $\texttt{CMS}_1$ (*i.e.,* from the state 14 to 17 of Figure 3). This means that the value of new prefix is "bb". So, we have $\texttt{P}_1 = \{ab, bb\}$. Similarly, *w.r.t.* $\texttt{CMS}_2$ = "bb#", we obtain $\texttt{P}_2 = \{aa, ab\}$. Consequently, two sub-languages of L are:

$$\begin{array}{ll} \texttt{L}_1 = \texttt{P}_1 \oplus \text{ ab} & \texttt{P}_1 = \{ab, bb\} \\ \texttt{L}_2 = \texttt{P}_2 \oplus \text{ bb} & \texttt{P}_2 = \{aa, ab\} \end{array}$$

---

func Refine($\texttt{TR}, \texttt{S}_\texttt{p}, \texttt{O}$)
  $\texttt{q} \leftarrow 1$; $\texttt{P}_\texttt{q} \leftarrow \emptyset$; $\texttt{K}_\texttt{r} \leftarrow \emptyset$;
  for $\texttt{s}_\texttt{p} \in \texttt{S}_\texttt{p}$ **do**
    $\texttt{T} \leftarrow$ CollectTerminalStates(TR);
    for $\texttt{t} \in \texttt{T}$ **do**
      $\texttt{CMS}_\texttt{q} \leftarrow$ SubPath($\texttt{s}_\texttt{p}$, t); {*i.e.,* Sequence from $\texttt{s}_\texttt{p}$ to t.}
      $\texttt{P}_\texttt{q} \leftarrow$ CurrentPrefix($\texttt{s}_\texttt{p}$); { *i.e.,* Sequence from 0 to $\texttt{s}_\texttt{p}$.}
      $\texttt{P}_\texttt{q} \leftarrow \texttt{P}_\texttt{q} \cup$ OtherPrefixes($\texttt{CMS}_\texttt{q}$); {*i.e.,* Other possible sequences.}
      $\texttt{TR}_\texttt{q} \leftarrow$ FormTrie($\texttt{P}_\texttt{q}$);
      $\texttt{g}_\texttt{q} \leftarrow$ CompactInputLanguage($\texttt{TR}_\texttt{q}$); {*i.e.,* Minimal DFA.}
      if $\texttt{O} \neq \emptyset$ {Dealing with output language.} **then**
        $\texttt{O}_\texttt{q} \leftarrow$ CollectOutputs($\texttt{P}_\texttt{q}$);
        $\texttt{mdt}_\texttt{q} \leftarrow$ LearnOutputLanguage($\texttt{TR}_\texttt{q}, \texttt{O}_\texttt{q}$); {*i.e.,* Decision tree.}
      end if
    end for
    $\texttt{K}_\texttt{r} \leftarrow \texttt{K} \oplus \texttt{rcms}_\texttt{q}$; $\{\texttt{rcms}_\texttt{q} = \texttt{rev}(\texttt{CMS}_\texttt{q})\}$
    $\texttt{q} \leftarrow \texttt{q} + 1$;
  end for
  $\texttt{TR}_\texttt{q} \leftarrow$ FormTrie($\texttt{K}_\texttt{r}$);
  $\texttt{g}_{\texttt{q}+1} \leftarrow$ CompactInputLanguage($\texttt{TR}_\texttt{q}$);
  $\texttt{mdt}_\texttt{rcms} \leftarrow$ LearnOutputLanguage($\texttt{TR}_\texttt{q}, \{1, \cdots, \texttt{q}\}$);

Algorithm 2: The refinement method.

`Step 5-3:` In this stage, we form `q` automata of the prefix–sub-languages of the reservoir, where recall that `q` denotes the number of prefix–sub–languages obtained in step 5-2. For each prefix-sub-language, like in Step 3, we form a minimal DFA and possibly a learned `m`-ary decision tree.

Example 11: The two first automata of Figure 1 are the outputs of this stage. There is no need to learn two decision trees because there is no output language *w.r.t.* the reservoir at hand, namely, $\mathtt{f} = \{\mathtt{aabb}, \mathtt{abab}, \mathtt{abbb}, \mathtt{bbab}\}$.

`Step 5-4:` This stage works as follows: (1) form the reservoir of reverse-CMSes, namely, $\mathtt{f}_{\mathtt{q}+1} = \{(\mathtt{rcms}_1, 1), \cdots, (\mathtt{rcms}_\mathtt{q}, \mathrm{q})\}$; (2) Form an automaton associated with $\mathtt{K}_\mathtt{r} = \{\mathtt{rcms}_1, \cdots, \mathtt{rcms}_\mathtt{q}\}$; and finally, (3) Learn the values associated with the strings of $\mathtt{K}_\mathtt{r}$.

Example 12: The bootom automaton (third one) of Figure 2 represents the minimal DFA, while the expression

$$(\mathtt{rcms}[2] = \text{`}\mathtt{a}' \wedge \mathtt{v} = 1) \vee (\mathtt{rcms}[2] = \text{`}\mathtt{b}' \wedge \mathtt{v} = 2)$$

stands for its associated decision tree.

5 On demand fast lookup

The complete storage space of this work is composed of $\mathtt{q}+1$ graphs (automata) plus one or $\mathtt{q}+1$ `m`-ary decision trees: one if there is no output language, $\mathtt{q}+1$ otherwise. However, loading that space, at the beginning stage of the lookup process can be problematic for large datasets. Furthermore, given a user-input (`x`), we are interested in performing the lookup process of `x` using only a `small` fragment-number of that complete space. In our work, that number is ranged from 1 to 4: `at least` one automaton (the last one: $\mathtt{g}_{\mathtt{q}+1}$); `at most` two graphs, namely, $\mathtt{g}_{\mathtt{q}+1}$ and $\mathtt{g}_\mathtt{j}$ plus two decision trees, namely, $\mathtt{mdt}_\mathtt{j}$, $\mathtt{mdt}_{\mathtt{rcms}}$, where $\mathtt{j} = \mathrm{SearchSubSpace}(\mathtt{rcms}, \mathtt{mdt}_{\mathtt{rcms}})$.

Consequently, by on demand, we mean that, depending `x`, only, one, two, three or four mentioned sub-storage(s) will be loaded in the main memory , not the complete storage which can be huge.

The formal description of the refinement method is shown in Algorithm 3 where by the function SpellOut(`x`, `g`) we mean $\delta^\star(\mathtt{q}_0, \mathtt{x})$ using the transitions of the graph `g`.

func OnDemandLookUp() // Using our method

$\alpha \leftarrow \ \bot$; {LoadTag of the last decision tree.}
for $\mathtt{i} = 1$ to $\mathtt{q}$ {Step 1} **do**
$\mathtt{AutomataTag[i]} \leftarrow \ \bot$; $\mathtt{LearningTag[i]} \leftarrow \ \bot$;
end for
LoadInMainMemory($\mathtt{g_{q+1}}$); {First ermenanet storage.}
while $\exists$ x {*i.e.,* user-input string.} **do**
$\mathtt{x'} \leftarrow$ rev(x); $\mathtt{rcms} \leftarrow$ SpellOut($\mathtt{x'}$, $\mathtt{g_{q+1}}$); {Step 2: $\delta^\star(\mathtt{q_0}, \mathtt{x'})$}
if $\mathtt{rcms} = \mathtt{nil}$ {*i.e.,* $\delta(\mathtt{q_0}, \mathtt{x'}[1]) = \bot$} **then**
return($\bot$); {*i.e.,* x is not a word of the input language.}
end if
if $\alpha = \bot$ {Now, on demande loading} **then**
LoadInMainMemory($\mathtt{mdt_{rcms}}$); $\alpha = \top$; {2th permenanet storage.}
end if
$\mathtt{j} \leftarrow$ SearchSubSpace($\mathtt{mdt_{rcms}}$, rcms); {See Example 3.}
$\mathtt{x_p} \leftarrow$ Substr(x, $|\mathtt{x}| - |\mathtt{rcms}|$);
if $\mathtt{AutomataTag[j]} = \bot$ {*i.e.,* On demand loading.} **then**
LoadInMainMemory($\mathtt{g_j}$); $\mathtt{AutomataTag[j]} \leftarrow \ \top$;
end if
$\mathtt{r_p} \leftarrow$ SpellOut($\mathtt{x_p}$, $\mathtt{g_j}$);
if $\mathtt{r_p} =$ nil {*i.e.,* $\delta(\mathtt{q_0}, \mathtt{x_p}[1]) = \bot$} **then**
return($\bot$); {$\mathtt{x} \notin \mathtt{L}$}
end if
if $\mathtt{O} = \emptyset$ {Reservoir doesn't contain the output language.} **then**
return($\top$); {$\mathtt{x} \in \mathtt{L}$}
end if
if $\mathtt{LearningTag[j]} = \bot$ **then**
LoadInMainMemory($\mathtt{mdt_j}$); $\mathtt{LearningTag[j]} \leftarrow \ \top$;
end if
$\mathtt{v} \leftarrow$ SearchSubSpace($\mathtt{x_p}$, $\mathtt{mdt_j}$)
return(v); {Output value of $\mathtt{x} \in \mathtt{L}$.}
end while
cnuf

Algorithm 3: Fast lookup along with on demand main memory loading.

`Step 1:` One permanent storage is loaded in the main memory: The last graph: $\mathtt{g_{q+1}}$. The second permanent storage may be $\mathtt{mdt_{rcms}}$: the decision tree associated with $\mathtt{g_{q+1}}$ see Algorithm 3. We maintain two load-tags, namely AutomataTag and LearningTag, each of which is one-dimension of size q, with false initialization meaning that $\mathtt{g_i}$ (for $1 \leq \mathtt{i} \leq \mathtt{q}$) and $\mathtt{mdt_i}$ (if

R has the output language) has not been loaded in the main memory. We may use them only if user-input dictate (see Step 2).

Step 2: We write x' to denote the reverse form of user input-string. This stage starts to spell out x' using g_{q+1}. This is done by the extension of the partial δ mapping defined in Section 2. Let $(k > 0) \wedge (k \leq |x|)$ be the position where spelling fails or succeeds. Note that the case of $k = |x|$ means that the word x has an empty prefix (see Figure 8). If the transition from the starting state (q_0) is undefined *i.e.,* $\delta(q_0, x'[1]) = \perp$ this means that $x \notin L$, otherwise the third step will be called.

Example 13: The following input-string x = "aabc" is not an element of the input language of Example 1, because using g_3 of Figure 2 doesn't allow that the transition be defined *i.e.,* $\delta(q_0, 'c') = \perp$. Therefore, we save the amount of spaces of g_1 et g_2 *i.e.,* they are not loaded in the main memory. As for x = "aabb" two deux graphs are loaded : one permanent, another on demand, namely, g_3, g_2, respectively.

Step 3: In this stage, we determine which automaton has be to inspected. This is done using the output of Step-2, namely rcms and mdt_{q+1}. That is to say, first the output of the function SearchSubspace(mdt_{q+1}, rcms) which is a positive integer (say j for $1 \leq j \leq q$) will determine the right graph to be used. Then, the prefix part of x, namely, x_p is spelled out using g_j.

Example 14: Let x' = "bbaa", in Step 2, the transitions of "bb" can be done using g_3 of Figure 2. It remains that the right sub–space to be learned for inspection of the remained part: the prefix of x, namely x_p = "aa". Given rcms = "bb" and the following learned decision tree: (rcms[2] = 'a' $\wedge$ v = 1) $\vee$ (rcms[2] = 'b' $\wedge$ v = 2). Now, we know that x_p has to spelled out using g_2: the second graph of Figure 2.

Step 4: In this stage, x_p is spelled out using its corresponding automaton. If spelling fails this means that $x \in L$. If spelling succeeds, we look for the value associated with x in case the reservoir contains the output language.

Example 15: Since x_p can be spelled by g_2 of Figure 2 and there is no output language in Example 1, then only the value true asserting that $x \in L$ will be returned.

6 Related work

Sheng Yu et al. [1, 7] proposed an interesting method called FDCA (Finite Deterministic Cover Automata) for reducing the size of the automaton as-

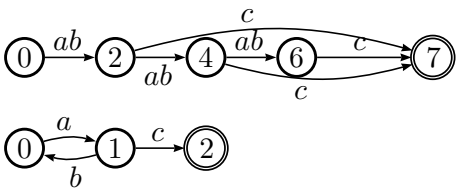


Figure 7: A (8,9) DFA and its FDCA of $\mathtt{L} = \{abc, ababc, abababc\}$, respectively.

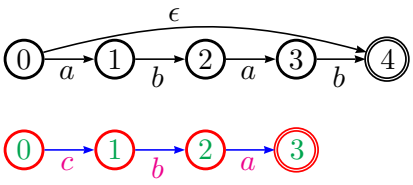


Figure 8: Two sub-languages of L.

sociated with input language L. A FDCA for L is a DFA that accepts all the words in L and possibly additional words of length greater than the length of the longest word(s) in L.

Example 16: ([7]) Figure 7 shows a (8,9)-automaton associated of $\mathtt{L} = \{abc, ababc, abababc\}$ and its counterpart, respectively.

Although, *w.r.t.* this example, FDCA `beats` advantageously our alternative in terms of the size of the automaton, but not `always`. The reason is that, unfortunately, there is no discussion on which kind of reservoir (read input language - L) is appropriate for transforming a DFA into a FDCA. Consequently, one doesn't `know` in advance when FDCA is good for being applied.

As mentioned above, if user-reservoir has no string-class, rather than losing the formed trie (see Step 2 of Algorithm 1), our method use it for forming a unique unlabeled DFA, possibly, along with the decision tree if the reservoir has the output language [3].

The main feature of our method is that spelling out user-input strings is done in a `learned` appropriate sub-space rather than in a complete one. When one uses FDCA there is no way that the search be done in the right sub-space. In addition that FDCA not having our feature mentioned above, one also has to consider the cost of forming possibly several distinct FDCAs.

Example 17: The following language $L = \{a, ab, ba, aba, abb, baa, bab\}$ has two distinct FDCAs. Consequently, one has to decide which one is good for being used.

It is worth mentioning that our method allows to determine the output values of user-input with a "less price" from the time and space requirements of the computation. Either our method be applicable (read dealing with string-classes) or not, if we are dealing with output language, then, instead of using the transducer (*e.g.,* Figure 9) we learn the decision tree (*e.g.,* Figure 10). The rasion of our policy is as follows: although, finite-state

transducers (automata with outputs) can be used to map an input language onto a set of values, our method uses an alternate representation method for such a mapping, consisting of associating a finite-state automaton accepting the input language with a decision tree representing the output values. The advantages of this approach are that it leads to more compact representations than transducers, and that decision trees can easily be synthesized by machine learning techniques. Combining automata with machine learning has the following desired properties:

1. The number of the states (and hence the transitions) representing the input language of our method is less than compared to the transducers.

2. In constructing transducers, we have to represent every transition by a data structure of at least two fields: one for the symbol representing the transition, another for the label-value (for short label) associated with the symbol. So in order to properly calculate the outputs, the labels set needs to have the algebraic structure *e.g.,* semiring in the case of weighted automata [6]. In our approach the transitions are not labeled with outputs; the cost of exploring the automata is low.

3. In most applications (*e.g.,* those of using part of speech tagging) there may be (many) identical output values. When you use the transducers there is no way to save the amount of space for those identical information, whereas in our approach such economy is allowed.

Example 18: Let us consider the following reduced user-reservoir taken form [4]: `f` = {(cabba, [xxxxx,xxyyx,xtzyx]), (cabca, [yzxxy,yzyyy])}. Figure 9 shows that the size of the transducer (of `f`) is composed of 13 nodes and 16 edges, whereas our method outputs a smaller size: a (7,7) unlabeled automaton along with a decision tree built on the best character, namely the second character from right to left of the words (of `f`), as Figure 10 asserts.

`Double gain:` Note that if user-reservoir has the output language and if it can be refined into sub-languages, then, the "gain' is double when searching a value of a word of `f`: (1) we are allowed to access to the right sub-space; (2) Rather than dealing with the transducers, the unlabeled automata along with the decision trees are used.

Example 19: Let `f` be {(aabb,1),(abab,2),(abbb,3), (bbab, 4)}. This reservoir is exactly, the extended form of the input language of `Example 1`. Given for instance, `x` = "aabb", one can obtain 2 as the output value of `x` using

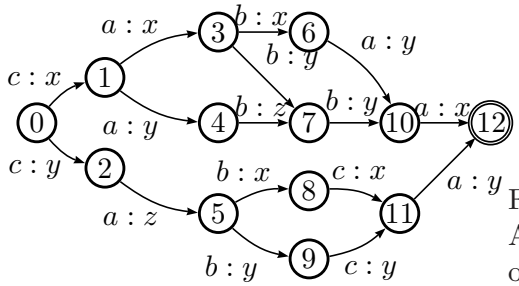


Figure 9: A (13,16)-transducer [4, Page 158] associated with $\mathtt{f}=$ {(cabba, [xxxxx,xxyyx,xtzyx]), (cabca, [yzxxy,yzyyy])}.

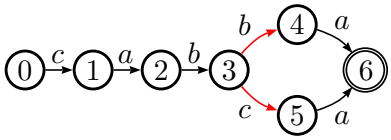


Figure 10: Our alternative: A (7,7)-automaton along with one decision rule: if $\mathtt{b}_2$ = 'b' then [xxxxx,xxyyx,xtzyx] else [yzxxy,yzyyy], where $\mathtt{b}_2$ denotes the second character from right to left.

$\mathtt{g}_3$, $\mathtt{mdt}_3$, $\mathtt{g}_2$ and $\mathtt{mdt}_2$, respectively.

Example 20: The reservoir Example 17 can be refined into two sub-languages, but this time using the common maximal prefix (CMP), where $\mathtt{CMP}=$ "cab" along with $\mathtt{S}_1 = \{$ba,ca$\}$, as the suffix-sub-language. The value of a word of $\mathtt{f}$ can be obtained in a similar way of the previous exmaple.

7 Final words

There are many data of the real world with the properties of having non empty common maximal suffixes, prefixes or both. Our method is designed for such a data, under consideration of the following `main` feature: Spelling out the user-input string is done in a `learned` appropriate subspace rather than in a complete space *i.e.,* a unique automaton which can be dramatically very large. The reader may have noticed that if the data has not the mentioned properties our current work based in part on our previous one [3] outputs a smaller representation compared to the transducer.

The following is among future works to be investigated:

1. If user-reservoir ($\mathtt{R}$) has both common maximal prefixes and suffixes then, which one is good for compact representation and fast lookup?

2. Although, experiments done on several datasets confirm the effectiveness of our method from the time and space requirements of the computation, but the theoretical complexities and establishing the tight upper bounds *w.r.t.* $\mathtt{R}$ is desired.

Note that, as far as we know, when using the transducers, the questions of time and space complexities *w.r.t.* the size of $\mathtt{R}$ are still opens.

Acknowledgments

I thank Bruce W. Watson and Loek Cleophas for their valuable helps about the preliminary version of this paper. This work is partially supported by le Conseil Régional de Lorrain (France) and by the Dutch Research Council.

References

[1] C. Câmpenau, N. Sântean, and S. YU. Minimal cover automata for finite languages. In *3th Int. Workshop on Implementation Automata*, pages 32–42, 1998.

[2] J. Daciuk, S. Mihov, B.W. Watson, and R.E. Watson. Incremental construction of finite-state automata. In *6th Int. Conf. on Association for Computational Linguistics*, pages 3–16. ACL, 2000.

[3] A. Fatholahzadeh. Implementation of dictionaries via automata and decision trees. In J.M. Champarnaud and D. Maurel, editors, *LNCS (Lecture Notes in Computer Science): Implementation and Application of Automata*, volume 2608, pages 95–105. Springer–Verlag, Berlin Heidelberg, 2003. Also in CIAA-2002.

[4] A. Kempe. Factorizations of ambiguous finite-state transducer. In *International Conference on Implementation and Application of Automata*, pages 157–164, 2000.

[5] T.M. Mitchel. *Machine Learning.* Mc Graw–Hill, 1993.

[6] M. Mohri. Finite-state transducers in language and speech processing. *Computational Linguistics*, pages 269–311, 1997.

[7] A. Păun, N. Sântean, and S. YU. An $o(n^2)$ algorithm for minimal cover automata for finite languages. In *5th Int. Conf. on Implementation and Application of Automata*, pages 243–251, 2000.

[8] G. Rozenberg and A. Salomaa, editors. *Handbook of formal language.* Springer–Verlag, Berlin Heidelberg, 1997.

Reward Variance in Markov Chains: A Calculational Approach

Tom Verhoeff *

May 2004

Abstract

We consider the variance of the reward until absorption in a Markov chain. This variance is usually calculated from the second moment (expectation of the square). We present a direct system of equations for the variance, involving the first moment (expectation) but not the second moment. This method is numerically superior to the calculation from the second moment.

1 Introduction

Consider the following problem. A spider is located on the ceiling of a cubic room. Each day it travels from one face to a neighboring face, crossing a single edge. The spider randomly chooses among the four neighboring faces, with uniform probability. Successive choices are independent. What is the expected number of days for the spider to reach the floor? And, in particular, what is the corresponding variance?

This problem can be modeled as a Markov chain, where each transition contributes a fixed 'reward' of one day. The ceiling is the initial state, the walls are other transient states, and the floor is an absorbing state. The problem can then be reformulated as finding the expected reward until absorption, and its variance.

We present, what we believe to be, a new method for calculating the variance in the reward until absorption. We came to this method when analyzing strategies for playing a solitaire version of the dice game Yahtzee.

Markov chains are not only useful for analyzing puzzles and games of chance, but also play a prominent role in economics and engineering. Attention is often focused on the expected reward (or cost). However, in practice, the variance is also important, because it relates to risk and buffer capacity needed for handling the swings around the expected value. For instance, it can be more economical to aim for a suboptimal expected value in favor of a lower variance, because this improves the predictability of a budget. The Dutch government only recently decided [1] that its goal in addressing the traffic jam problem would no longer be a reduction of the expected travel time, but rather a reduction in the variability of travel time to improve predictability even if that means longer travel times.

*Technische Universiteit Eindhoven, Fac. of Math. & CS, P.O. Box 513, 5600 MB Eindhoven, The Netherlands

In the remainder of this section, we explain some general concepts and results from probability theory and our notations. In Section 2, we do the same for Markov chains. Section 3 contains our new result. The spider in the cubic room is a running example. Finally, Section 4 concludes the article.

1.1 Probability Theory

For an introduction to probability theory see for instance [2]. We summarize the concepts needed for understanding this article.

A sample space is a set Ω of (mutually exclusive) sample points or outcomes. To model a stochastic experiment, each outcome $s \in \Omega$ is assigned a probability $P.s$, measuring the likelihood that s occurs. We have $0 \leq P.s \leq 1$ and $\sum_{s\in\Omega} P.s = 1$. The pair (Ω, P) is also called a probability space.

For example, the set of six faces (ceiling, four walls, and floor) of the cubic room forms a sample space. Assuming the spider is on the ceiling, let $P.s$ be the probability that the spider will be on face s the next day. We then have $P.s = \frac{1}{4}$ if s is one of the four walls, and $P.s = 0$ if s is the ceiling or the floor.

Given a probability space (Ω, P), a random variable X is a function $\Omega \to \mathbb{R}$, where $X.s$ is the value of $s \in \Omega$. The expectation (first moment) $\mathcal{E}[X]$ of this random variable X is given by

$$\mathcal{E}[X] \quad = \quad \sum_{s\in\Omega} P.s * X.s \ . \tag{1}$$

It is also called the mean, and it captures the central tendency of the random variable. More generally, $\mathcal{E}\left[X^k\right]$ is the k-th moment of X. We will write $\mathcal{E}_\Omega[X]$, $\mathcal{E}_{\Omega,P}[X]$, $\mathcal{E}_s[X.s]$, or $\mathcal{E}_{s\in\Omega}[X.s]$ to make the relevant probability space and random variable more explicit.

For example, a roll of a fair die can be modeled by

- the sample space Ω consisting of the six faces of a cube,
- the probability $P.s = \frac{1}{6}$ for each face s to appear on top, and
- a function D associating a unique value in the range 1 through 6 with each face.

For the expectation of D we have $\mathcal{E}[D] = \sum_{s\in\Omega} \frac{1}{6} * D.s = \sum_{k=1}^{6} \frac{1}{6} * k = 3.5$.

An important property of the expectation is its linearity. For constants c and d (whose value does not depend on the sample space) and random variables X and Y on the same probability space, $X + cY + d$ is also a random variable on that probability space, having expectation

$$\mathcal{E}[X + cY + d] \quad = \quad \mathcal{E}[X] + c\mathcal{E}[Y] + d \ . \tag{2}$$

Note that, in general, $\mathcal{E}[X * Y] = \mathcal{E}[X] * \mathcal{E}[Y]$ does *not* hold, but it does hold if X and Y are *independent*. In particular, $\mathcal{E}\left[X^2\right]$ and $\mathcal{E}[X]^2$ are not necessarily equal. For instance, for the fair die we have $\mathcal{E}\left[D^2\right] = \sum_{k=1}^{6} \frac{1}{6} * k^2 = 15\frac{1}{6}$, whereas $\mathcal{E}[D]^2 = 3.5^2 = 12\frac{1}{4}$.

The amount of variability of a random variable X around its mean can be measured by its variance $\mathcal{V}[X]$, defined by

$$\mathcal{V}[X] \quad = \quad \mathcal{E}\left[\,(X - \mathcal{E}[X])^2\,\right] \ . \tag{3}$$

The variance can be expressed in terms of the first and second moment, as the following calculation[1] shows:

$$\begin{array}{ll} & \mathcal{V}[X] \\ = & \quad \{ \text{ definition of } \mathcal{V} \ \} \\ & \mathcal{E}\left[(X - \mathcal{E}[X])^2 \right] \\ = & \quad \{ \text{ algebra } \} \\ & \mathcal{E}\left[X^2 - 2X\mathcal{E}[X] + \mathcal{E}[X]^2 \right] \\ = & \quad \{ \text{ linearity of } \mathcal{E}, \text{ observing that } \mathcal{E}[X] \text{ is a constant } \} \\ & \mathcal{E}\left[X^2\right] - 2\mathcal{E}[X]\mathcal{E}[X] + \mathcal{E}[X]^2 \\ = & \quad \{ \text{ algebra } \} \\ & \mathcal{E}\left[X^2\right] - \mathcal{E}[X]^2 \end{array} \tag{4}$$

Numerically, however, the expression $\mathcal{E}\left[X^2\right] - \mathcal{E}[X]^2$ is inferior to (3), because of the risk of cancelation as illustrated by the following example. Consider a random variable X which takes on one of two values $a = 999$ and $b = 1001$ with probabilities $p = 0.1$ and $q = 0.9$ respectively. We then have $\mathcal{V}[X] = pq(a-b)^2 = 0.36$. When evaluating (3) and (4) using the IEEE-754 single format for floating-point numbers, the following dramatic results are obtained

$$\begin{array}{rcl} \mathcal{E}\left[(X - \mathcal{E}[X])^2 \right] & = & 0.36000\cdots \\ \mathcal{E}\left[X^2\right] - \mathcal{E}[X]^2 & = & 0.50660\cdots \end{array}$$

The latter value is 40% too high!

Note that $\mathcal{V}[X]$ is expressed in the square of the units of X. If X is expressed in m/s, then $\mathcal{V}[X]$ is expressed in $\mathrm{m}^2/\mathrm{s}^2$. The standard deviation σ measures the variability in the same units as the random variable by taking the square root of the variance:

$$\sigma[X] \quad = \quad \sqrt{\mathcal{V}[X]}\ . \tag{5}$$

In general, the variance does not satisfy a linearity property like (2), but we do have:

$$\begin{array}{ll} & \mathcal{V}\left[\, cX + d \,\right] \\ = & \quad \{ \text{ definition of } \mathcal{V} \ \} \\ & \mathcal{E}\left[(cX + d - \mathcal{E}[cX + d])^2 \right] \\ = & \quad \{ \text{ linearity of } \mathcal{E} \ \} \\ & \mathcal{E}\left[(cX + d - (c\mathcal{E}[X] + d))^2 \right] \\ = & \quad \{ \text{ algebra } \} \\ & \mathcal{E}\left[c^2(X - \mathcal{E}[X])^2 \right] \\ = & \quad \{ \text{ linearity of } \mathcal{E}, \text{ definition of } \mathcal{V} \ \} \\ & c^2\mathcal{V}\left[X\right] \end{array} \tag{6}$$

[1] This way of recording calculations is due to Feijen, see [3]. The hint in braces explains why the relationship shown on the left holds between the expressions above and below it.

2 Markov Chains

For an introduction to Markov chains see for instance [4, 5, 6]. We summarize the concepts needed for understanding this article.

A Markov chain models a stochastic process, where an experiment with outcomes in a sample space Ω is repeated and where the probability distribution for the outcome of each experiment can depend on the outcome of the preceding experiment.

It is often more convenient to view the sample space Ω of a Markov chain as a state space. At each time step, the system is in a state $s \in \Omega$. The transition from state s to state t in the next time step occurs with probability $p.s.t$. For all $s \in \Omega$, the transition probabilities $p.s.t$ satisfy

$$\sum_{t \in \Omega} p.s.t \quad = \quad 1 \; . \tag{7}$$

Let us define a Markov chain for the spider in the cubic room. The state space consists of the six faces where the spider can be located. We abbreviate them as C (Ceiling), W_i (Wall, $0 \le i < 4$), and F (Floor). The initial state is C. The transition probabilities are given by

$p.s.t$	C	W_0	W_1	W_2	W_3	F
C	0	1/4	1/4	1/4	1/4	0
W_0	1/4	0	1/4	0	1/4	1/4
W_1	1/4	1/4	0	1/4	0	1/4
W_2	1/4	0	1/4	0	1/4	1/4
W_3	1/4	1/4	0	1/4	0	1/4
F	0	1/4	1/4	1/4	1/4	0

(8)

Note that the matrix of transition probabilities is symmetric. However, this is not generally the case.

We are interested only in the spider's behavior until it reaches the floor. Therefore, the transition probabilities from the floor, given in the last row of (8), are irrelevant. We might as well make the spider stay on the floor: $p.\mathsf{F}.\mathsf{F} = 1$ and $p.\mathsf{F}.s = 0$ for $s \neq \mathsf{F}$.

Furthermore, the four states W_i are *equivalent* in view of what they offer for the future. We can collapse them into a single state W. This yields the following simplified Markov chain, which is also pictured in Fig. 1.

$p.s.t$	C	W	F
C	0	1	0
W	1/4	1/2	1/4
F	0	0	1

(9)

2.1 Walks

We now turn to sequences of successive state transitions. A nonempty sequence $s_0 s_1 \cdots s_n$ of $n+1$ states $s_i \in \Omega$ is called a walk of length n from s_0 to s_n. This sequence represents n successive state transitions $s_{i-1} \to s_i$. The length of a walk is the number of state transitions it involves. Note that for a walk w of

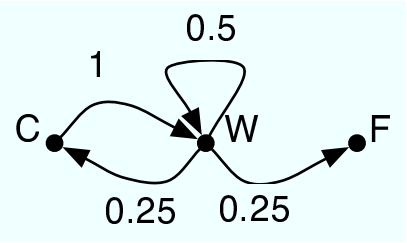


Figure 1: Simplified Markov chain for spider in cubic room (transitions with probability 0 not drawn)

length zero we have $w = s_0$. We denote catenation of sequences by juxtaposition. Variables s and t range over states, whereas variables v and w range over sequences of states.

The successive experiments in a Markov chain are independent and, hence, the transition probabilities can be multiplied. Thus, the probability $P.sw$ of a walk sw, given that it starts in s, satisfies the recurrence:

$$P.s = 1 , \tag{10}$$

$$P.stv = p.s.t * P.tv . \tag{11}$$

We are interested in the analysis of walks until their first arrival in some nonempty subset $A \subseteq \Omega$. In the example of the spider, our observation of the walk ends when the spider arrives at the floor for the first time, that is, $A = \{ \mathsf{F} \}$. In this article, we will use 'first arrival in A' and 'absorption in A' interchangeably.[2]

For nonempty $A \subseteq \Omega$, let $W_A.s$ be the set of walks starting in $s \in \Omega$ until first arrival in A. We will leave out the subscript A, because A will not vary. If $s \in A$, then the walk ends immediately. If $s \notin A$, then such walks involve at least one transition to some state t, from where the walk proceeds until first arrival in A. Formally:

$$W.s = \{ s \} \qquad \text{if } s \in A , \tag{12}$$

$$W.s = \biguplus_{t \in \Omega} \{ stv \mid tv \in W.t \} \qquad \text{if } s \notin A . \tag{13}$$

For this article, we make one important assumption about A: starting in state s, the probability that a walk eventually ends in A is 1. That is, for all s, we have

$$\sum_{sw \in W.s} P.sw = 1 . \tag{14}$$

Hence, $W.s$ is a sample space, and $P.sw$ for $w \in W.s$ is a probability function on it.

2.2 Rewards

In the example of the spider, we are interested in the expected *length* of a walk until absorption. More generally, we associate with each transition $s \to t$ a

[2]Strictly speaking, the transition probabilities should be redefined to make states $s \in A$ truly absorbing: $p.s.s = 1$ and $p.s.t = 0$ for $t \neq s$.

reward $r.s.t$. If we are only interested in the walk length, then we take $r.s.t = 1$. The (total) reward $R.sw$ of walk sw is defined inductively by

$$R.s = 0 , \tag{15}$$

$$R.stv = r.s.t + R.tv . \tag{16}$$

That is, successive rewards are independent and, hence, are simply added.

The reward $R.sw$ obtained when starting in state s and walking until first arrival in A is a random variable on the probability space $(W.s, P.sw)$. In the remainder of this section, we deal with the expectation $\mathcal{E}_{W.s}[R.sw]$. Even though this is a well-known result, we have included it here in detail for two reasons:

- We have not seen it treated in this way elsewhere.
- The treatment of the variance follows the same pattern.

First, however, we derive a pair of convenient properties for the expectation $\mathcal{E}_{W.s}[X]$ of an arbitrary random variable X on walks from s until first arrival in A.

Property For $s \in A$:

$$\begin{aligned}
& \mathcal{E}_{W.s}[X] \\
= & \quad \{ \text{ definition of } \mathcal{E} \ \} \\
& \sum_{w \in W.s} P.w * X.w \\
= & \quad \{ \ W.s = \{ \, s \, \}, \text{ because } s \in A \ \} \\
& P.s * X.s \\
= & \quad \{ \text{ by definition } P.s = 1 \ \} \\
& X.s
\end{aligned}$$

Property (conditioning on the first step toward absorption) For $s \notin A$:

$$\begin{aligned}
& \mathcal{E}_{W.s}[X] \\
= & \quad \{ \text{ definition of } \mathcal{E} \ \} \\
& \sum_{w \in W.s} P.w * X.w \\
= & \quad \{ \text{ write } w = sv \text{ for } v \in W.t, \text{ because } s \notin A, \text{ cf. (13)} \ \} \\
& \sum_{t \in \Omega} \sum_{v \in W.t} P.sv * X.sv \\
= & \quad \{ \text{ recurrence for walk probability: } P.sv = p.s.t * P.v \text{ for } v \in W.t \ \} \\
& \sum_{t \in \Omega} \sum_{v \in W.t} p.s.t * P.v * X.sv \\
= & \quad \{ \text{ distribute } p.s.t * \text{ outside } \textstyle\sum_v, \text{ using that } p.s.t \text{ does not depend on } v \ \} \\
& \sum_{t \in \Omega} p.s.t * \sum_{v \in W.t} P.v * X.sv \\
= & \quad \{ \text{ definition of } \mathcal{E} \ \} \\
& \sum_{t \in \Omega} p.s.t * \mathcal{E}_{v \in W.t}[X.sv] \\
= & \quad \{ \text{ definition of } \mathcal{E} \ \} \\
& \mathcal{E}_{t \in \Omega} \, [\, \mathcal{E}_{v \in W.t}[X.sv] \,]
\end{aligned}$$

Concerning the expected reward $\mathcal{E}_{W.s}[R]$ on a walk from state s until absorption in A, we can now calculate the following (well-known) result.

For $s \in A$:

$$\begin{array}{ll} & \mathcal{E}_{W.s}[R] \\ = & \quad \{ \text{ property above, using } s \in A \ \} \\ & R.s \\ = & \quad \{ \text{ definition of } R \ \} \\ & 0 \end{array}$$

For $s \notin A$:

$$\begin{array}{ll} & \mathcal{E}_{W.s}[R] \\ = & \quad \{ \text{ conditioning on first state } t \text{ after state } s, \text{ using } s \notin A \ \} \\ & \mathcal{E}_{t \in \Omega}\,[\,\mathcal{E}_{v \in W.t}[R.sv]\,] \\ = & \quad \{ \text{ recurrence for walk reward: } R.sv = r.s.t + R.v \text{ for } v \in W.t \ \} \\ & \mathcal{E}_{t \in \Omega}\,[\,\mathcal{E}_{v \in W.t}[\,r.s.t + R.v\,]\,] \\ = & \quad \{ \text{ linearity of expectation, using that } r.s.t \text{ is independent of } v \ \} \\ & \mathcal{E}_{t \in \Omega}\,[\,r.s.t + \mathcal{E}_{v \in W.t}[R.v]\,] \\ = & \quad \{ \text{ simplify notation } \} \\ & \mathcal{E}_{t \in \Omega}\,[\,r.s.t + \mathcal{E}_{W.t}[R]\,] \end{array}$$

This gives us a system of linear equations with as unknowns $\mu_s = \mathcal{E}_{W.s}[R]$ for each $s \in \Omega$:

$$\mu_s \quad = \quad \sum_{t \in \Omega} p.s.t * (r.s.t + \mu_t) \ . \tag{17}$$

If $r.s.t = 1$ (measuring the length of a walk), then this can be simplified to

$$\mu_s \quad = \quad 1 + \sum_{t \in \Omega} p.s.t * \mu_t \ . \tag{18}$$

Consider the three-state Markov chain (9) for the spider in the cubic room. We take $A = \{\, \mathsf{F} \,\}$. According to (18), the system of equations for the expected walk lengths $\mu_s = \mathcal{E}_{W.s}[R]$ from face s to absorption on the floor is:

$$\begin{array}{rcl} \mu_{\mathsf{C}} & = & 1 + \mu_{\mathsf{W}} \\ \mu_{\mathsf{W}} & = & 1 + \left(\frac{1}{4}\mu_{\mathsf{C}} + \frac{1}{2}\mu_{\mathsf{W}} + \frac{1}{4}\mu_{\mathsf{F}} \right) \\ \mu_{\mathsf{F}} & = & 0 \end{array}$$

This has as solution:

$$\begin{array}{rcl} \mu_{\mathsf{C}} & = & 6 \\ \mu_{\mathsf{W}} & = & 5 \\ \mu_{\mathsf{F}} & = & 0 \end{array}$$

Thus, when starting on the ceiling, the expected duration for the spider to hit the floor is exactly 6 days.

3 Reward Variance

We now turn to the variance $\mathcal{V}_{W.s}[R]$ in the reward on a walk from state s to absorption in A. For $s \in A$, we calculate

$$\begin{array}{ll} & \mathcal{V}_{W.s}[R] \\ = & \quad \{ \text{ definition of } \mathcal{V} \ \} \\ & \mathcal{E}_{W.s}\left[\left(R - \mathcal{E}_{W.s}[R]\right)^2 \right] \\ = & \quad \{ \ \mathcal{E}_{W.s}[R] = 0, \text{ because } s \in A \text{ and } R.s = 0 \ \} \\ & \mathcal{E}_{W.s}\left[R^2 \right] \\ = & \quad \{ \text{ property of } \mathcal{E}_{W.s}, \text{ using } s \in A \text{ and } R.s = 0 \ \} \\ & 0 \end{array}$$

Before tackling $s \notin A$, we observe that for constant c and random variable X:

$$\mathcal{V}[X] = \mathcal{V}[c + X] = \mathcal{E}[(c+X)^2] - \mathcal{E}[c+X]^2 = \mathcal{E}[(c+X)^2] - (c + \mathcal{E}[X])^2 \ .$$

And, hence,

$$\mathcal{E}[(c+X)^2] \quad = \quad (c + \mathcal{E}[X])^2 + \mathcal{V}[X] \ . \tag{19}$$

Finally, for $s \notin A$, we calculate

$$\begin{array}{ll} & \mathcal{V}_{W.s}[R] \\ = & \quad \{ \text{ definition of } \mathcal{V} \ \} \\ & \mathcal{E}_{W.s}\left[\left(R - \mathcal{E}_{W.s}[R]\right)^2 \right] \\ = & \quad \{ \text{ conditioning on first state } t \text{ after state } s, \text{ using } s \notin A \ \} \\ & \mathcal{E}_{t \in \Omega}\left[\mathcal{E}_{v \in W.t}\left[\left(R.sv - \mathcal{E}_{W.s}[R]\right)^2 \right] \right] \\ = & \quad \{ \text{ recurrence for reward: } R.sv = r.s.t + R.v \text{ for } v \in W.t \ \} \\ & \mathcal{E}_{t \in \Omega}\left[\mathcal{E}_{v \in W.t}\left[\left(r.s.t + R.v - \mathcal{E}_{W.s}[R]\right)^2 \right] \right] \\ = & \quad \{ \ (19), \text{ using that } r.s.t - \mathcal{E}_{W.s}[R] \text{ does not depend on } v \ \} \\ & \mathcal{E}_{t \in \Omega}\left[\left(r.s.t + \mathcal{E}_{W.t}[R] - \mathcal{E}_{W.s}[R]\right)^2 + \mathcal{V}_{W.t}[R] \right] \end{array}$$

This yields a system of linear equations with as unknowns $\sigma_s^2 = \mathcal{V}_{W.s}[R]$ for each $s \in \Omega$, involving $\mu_s = \mathcal{E}_{W.s}[R]$ as parameters:

$$\sigma_s^2 \quad = \quad \sum_{t \in \Omega} p.s.t * \left((r.s.t + \mu_t - \mu_s)^2 + \sigma_t^2 \right) \ . \tag{20}$$

My earlier derivations of this result were quite messy. The derivation presented here is kept simple by using (19).

When applying (20) to the example of the spider, we obtain as system of equations for the variance in walk length $\sigma_s^2 = \mathcal{V}_{W.s}[R]$ from face s to absorption on the floor:

$$\begin{array}{rcl} \sigma_{\mathsf{C}}^2 & = & \left(1 + \mu_{\mathsf{W}} - \mu_{\mathsf{C}}\right)^2 + \sigma_{\mathsf{W}}^2 \\ \sigma_{\mathsf{W}}^2 & = & \frac{1}{4}\left(\left(1 + \mu_{\mathsf{C}} - \mu_{\mathsf{W}}\right)^2 + \sigma_{\mathsf{C}}^2 \right) + \frac{1}{2}\left(\left(1 + \mu_{\mathsf{W}} - \mu_{\mathsf{W}}\right)^2 + \sigma_{\mathsf{W}}^2 \right) + \frac{1}{4}\left(\left(1 + \mu_{\mathsf{F}} - \mu_{\mathsf{W}}\right)^2 + \sigma_{\mathsf{F}}^2 \right) \\ \sigma_{\mathsf{F}}^2 & = & 0 \end{array}$$

The first equation yields $\sigma^2_{\mathsf{C}} = \sigma^2_{\mathsf{W}}$, because $\mu_{\mathsf{C}} = 1 + \mu_{\mathsf{W}}$. This is understandable, since the probability for a transition from Ceiling to Wall equals 1 and, hence, there is no variability on this part of the walk.

The solution to the equation system is:

$$\begin{aligned} \sigma^2_{\mathsf{C}} &= 22 \\ \sigma^2_{\mathsf{W}} &= 22 \\ \sigma^2_{\mathsf{F}} &= 0 \end{aligned}$$

Hence, the standard deviation in the walk length from the ceiling to the floor is $\sqrt{22} \approx 4.69$. This is considerable compared to the expectation $\mu_{\mathsf{C}} = 6$. It means[3] that almost two million simulation runs are needed to estimate the expectation with an accuracy of 0.01 and a confidence level within 3σ.

The equations (20) can be generalized for covariance. Given two random variables X and Y on the same probability space, their *covariance* Cov.X.Y is defined by

$$\text{Cov}.X.Y \quad = \quad \mathcal{E}[\,(X - \mathcal{E}[X])(Y - \mathcal{E}[Y])\,]\ . \tag{21}$$

Note that Cov.X.$X = \mathcal{V}[X]$. Similar to (4), one can derive

$$\text{Cov}.X.Y \quad = \quad \mathcal{E}[XY] - \mathcal{E}[X]\mathcal{E}[Y] \tag{22}$$

Hence, we have (compare this to (2))

$$\mathcal{V}[X + cY + d] \quad = \quad \mathcal{V}[X] + c^2\mathcal{V}[Y] + 2c\text{Cov}.X.Y \tag{23}$$

Now consider two reward functions r and q on the same Markov chain. These induce the reward functions R and Q on walks. The covariances z_s between R and Q on walks starting in state s until absorption in A satisfy

$$z_s \quad = \quad \sum_{t \in \Omega} p.s.t * ((r.s.t + \mu_t - \mu_s)(q.s.t + \nu_t - \nu_s) + z_t)\ . \tag{24}$$

where $\mu_s = \mathcal{E}_{W.s}[R]$ and $\nu_s = \mathcal{E}_{W.s}[Q]$.

4 Conclusion

We have presented a new system of equations (20) for determining the variance of the reward until absorption in a Markov chain. Compared to the standard approach using the second moment, these equations have a lower risk of cancelation when solved numerically.

We applied this technique in our analysis of the dice game Yahtzee [7, 8]. The Markov chain for solitaire Yahtzee involves close to 10^9 states. Because it has no cycles, the resulting equations for expectation and variance are recurrence equations. These can be solved simply by backward substitution and dynamic programming. The analysis yields the optimal expected final score and its variance, also broken down by the individual scoring categories and their covariances. Cremers [9] extended the analysis to the beating of high scores.

I would like to acknowledge the helpful comments from Onno Boxma and Ivo Adan on an earlier version of this article.

[3] The variance in the average taken over n simulation runs equals the variance in a single run divided by n, and hence, the standard deviation in the average taken over n simulation runs equals $\sigma/\sqrt{n}$. The number of runs needs to be at least $\left(3\sqrt{22}/0.01\right)^2$.

References

[1] Karla Peijs. Nota Mobiliteit. Ministerie van Verkeer en Waterstaat, 2004. On-line: `http://www.notamobiliteit.nl/`.

[2] W. Feller. *Introduction to Probability Theory and Its Applications*, volume 1. Wiley, London, third edition, 1970.

[3] E. W. Dijkstra and C. S. Scholten. *Predicate Calculus and Program Semantics*. Springer, Berlin, 1990.

[4] John G. Kemeny and J. Laurie Snell. *Finite Markov Chains*. Springer, Berlin, 1976.

[5] Edward P. C. Kao. *An Introduction to Stochastic Processes*. Duxbury Press, Bonn, 1997.

[6] Sheldon M. Ross. *Introduction to Probability Models*. Academic Press, San Diego, eigth edition, 2003.

[7] T. Verhoeff. Optimal solitaire Yahtzee advisor and Yahtzee proficiency test, 1999. On-line since July 1999: `http://www.win.tue.nl/~wstomv/misc/yahtzee/`.

[8] H. van Maanen. Ook Yahtzee bezwijkt (Eng.: "Also Yahtzee succumbs"). *Het Parool*, 1999. Dutch daily newspaper, 9 oktober 1999.

[9] C. Cremers. How best to beat high scores in Yahtzee: A caching structure for evaluating large recurrent functions. Master's thesis, Fac. of Math. and CS, Technische Universiteit Eindhoven, The Netherlands, 2002.

Hardcoding and Dynamic Implementation of Finite Automata

Ernest Ketcha Ngassam[a], Bruce W. Watson[b], and Derrick G. Kourie[b]
[a] School of Computing, University of South Africa,
Pretoria 0003, South Africa
[b] Department of Computer Science, University of Pretoria,
Pretoria 0002, South Africa
E-mail: [a]ngassek@unisa.ac.za, [b]{bwatson, dkourie}@cs.up.ac.za

November 15, 2004

Abstract

The theoretical complexity of a string recognizer is linear to the length of the string being tested for acceptance. However, for some kind of strings the processing time largely depends on the number of states visited by the recognizer at run-time. Various experiments are conducted in order to compare the time efficiency of both hardcoded and table-driven algorithms when using such strings patterns. The results of the experiments are cross-compared in order to show the efficiency of the hardcoded algorithm over its table-driven counterpart. This help further the investigations on the problem of the dynamic implementation of finite automata. It is shown that we can rely on the history of the states previously visited in the dynamic framework in order to predict the suitable algorithm for acceptance testing.

1 Introduction

Previous work on Finite Automata (FAs) implementation revealed that the traditional table-driven (TD) algorithm may not be the sole approach for encoding a string recognizer. Another implementation approach using a hardcoded (HC) algorithm suggested by Knuth in [Kmp77] showed a time gain over the TD algorithm up to some threshold. In further experiments conducted in [Kwk04], it was shown that the processing time required to recognize a string largely depends on the structure of the string being recognized in relation to the overall structure of the automaton the recognizer is based upon. The possibility to improve the processing time of string recognizers using a dynamic algorithm called DIFAP[1] that handles both TD and HC algorithms simultaneously was

[1]Dynamic Implementation of Finite Automata for Performance

also suggested in [Kwk04]. However, the work only introduced the DIFAP algorithm without a complete analysis of the algorithm in terms of complexity and implementation. In this paper, we further the idea of DIFAP through analysis of its critical parts. The history of strings already processed by the recognizer is used to suggest the suitable algorithm to be used at run-time. Further improvements of DIFAP are suggested such as the extension of the original threshold of efficiency as well as a mixed-mode implementation of FAs using both HC and TD algorithms.

The structure of the remaining of this paper is as follows. In section 2 below, the implementation of FAs using both TD and HC algorithms is revisited. Section 3 reviews the experiments performed on string recognizers in order to capture the break-even variations of both algorithms. Various string patterns are investigated in this section showing the advantage of using HC over TD above the threshold of efficiency. In section 4, DIFAP is revisited followed by the analysis of its most critical section referred to as the knowledge table (KT). Additional improvements of DIFAP are suggested in Section 5, and we conclude and provide future directions to this work in Section 6.

2 Finite Automata Implementation

This section summarizes a review of automata implementation and the complexity of string recognizers already discussed in [Ket03]. Finite automata can be implemented using hardcode or softcode. The softcoded or TD algorithm requires a driver program made of few instructions to access the transition table during the entire recognition process. Algorithm 1 below depicts a TD algorithm that tests whether the string *str* is part of the language of the automaton represented by its transition matrix referred to as *transition*. The overall complexity of the recognizer is in the order of $O(len)$ where len is the length of the string being tested for acceptance. A hardcoded algorithm depicted in Algorithm 2 clearly shows that more instructions are required to represent the overall recognizer. Again, the complexity of the recognizer still remains in the order of $O(len)$ as for the TD algorithm. Both algorithms are different in terms of instructions and external data usage. The HC algorithm requires many instructions, which is not the case for the TD algorithm. The transition matrix is loaded into memory for the TD algorithm whereas only simple instructions are needed for its representation in the HC algorithm. Such observations clearly show that practical experiments are necessary to evaluate up to what extend the processing time of both algorithms differ. Section 3 below depicts various experiments on strings recognizers using both approaches.

Algorithm 1. *Table-driven string recognition*

```
function recognize(str,transition):boolean
  state := 0;
  stringPos := 0;
  while(stringPos < len) ∧(state ≥ 0) do
    state := transition[state][str[stringPos]];
```

```
      stringPos := stringPos+1;
    end while
    if state ≤ 0
      return(false);
    else
      return(true);
    end if
  end function
```

Algorithm 2. *Hardcoded string recognition*

```
 function recognize(str):boolean
  state_0 :
  if str[0] ∉ validsymbol_0 return(false);
  else if len = 1 return(true);
  else goto nextStates_1;
  end if
  state_1 :
  if str[1] ∉ validsymbol_1 return(false);
  else if len = 2 return(true);
  else goto nextStates_2;
  end if
...
...
...
  state_{numberOfStates-1} :
  if str[numberOfStates − 1] ∉ validsymbol_{numberOfStates-1} return(false);
  else return(true);
  end if
 end function
```

3 String Recognition Experiments

In this section, we present the experiments carried out on various kind of strings using both HC and TD algorithms. The experiments where conducted on an Intel Pentium IV at 1.8 GHz with 512MB of RAM and 20GB of hard drive. The TD algorithm was implemented using the gnu C++ compiler, and NASM (Netwide Assembler) was used for the HC implementation. The experiments were conducted under the Linux operating system. Randomly generated strings were investigated as well as various kind of strings that may offer some time gain when the HC algorithm is considered as opposed to the TD algorithm. This section summarizes experiments carried out in [Kwk04]. The subsection below depicts experiments based on random strings.

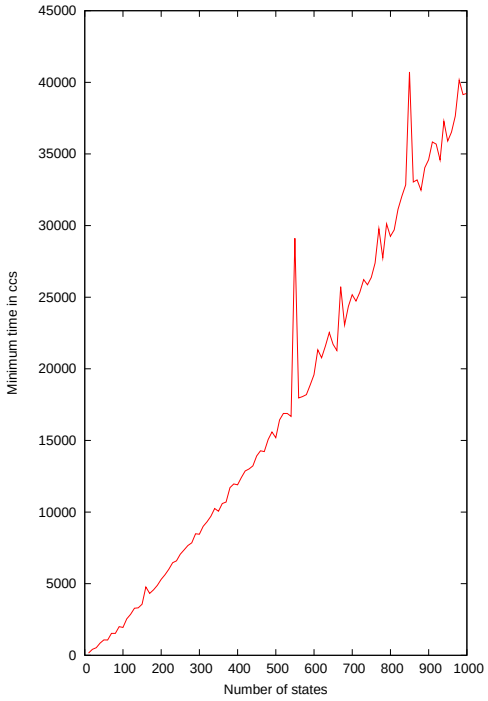


Figure 1: TD performance: 25 symbols

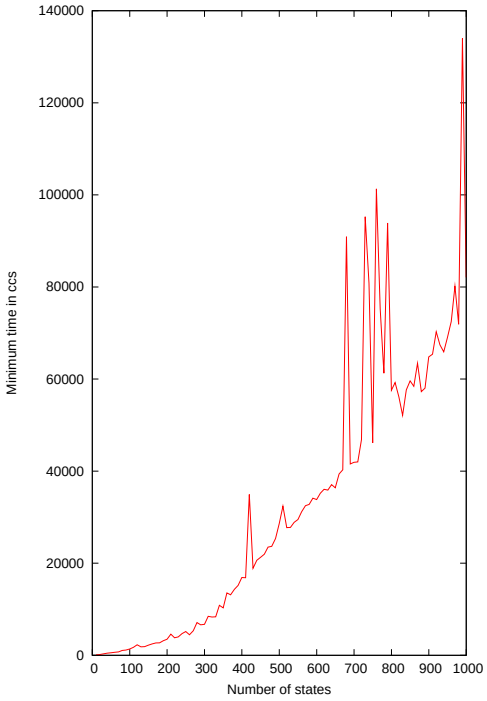


Figure 2: HC performance: 25 symbols

3.1 Experiments based on Random Strings

Experiments were based on alphabet size varying between 10 and 50 symbols with an increment of 5. For each alphabet size under consideration, 100 random automata of sizes ranging from 10 states to 1000 states with an increment of 10 were generated. For each case, a random accepting string of length $n-1$ was also generated for acceptance testing. Figures 1 and 2 depict the performance for both TD and HC algorithms for automata based on 25 alphabet symbols. It is observed that both graphs show a superlinear growth on the number of states. However, the HC experiment shows a slow growth in the region between 10 states and about 400 states. A plausible explanation to this may lies on the effect of cache on automata of smaller sizes. For such automata, the entire code size can fit into cache and reducing therefore the processing time since the probability of cache misses is very low in that region. Above the 400 states, the HC processing becomes inefficient due to the high probability of cache misses. A comparison between the two algorithms is depicted in Figure 3. It is clearly observed that the HC algorithm outperforms the TD algorithm up to the region between 300 and 400 states. As a result to this, FAs implementers should consider using the HC algorithm when solving computational problems based on automata of size less than about 360 states. However, above that threshold, the TD algorithm

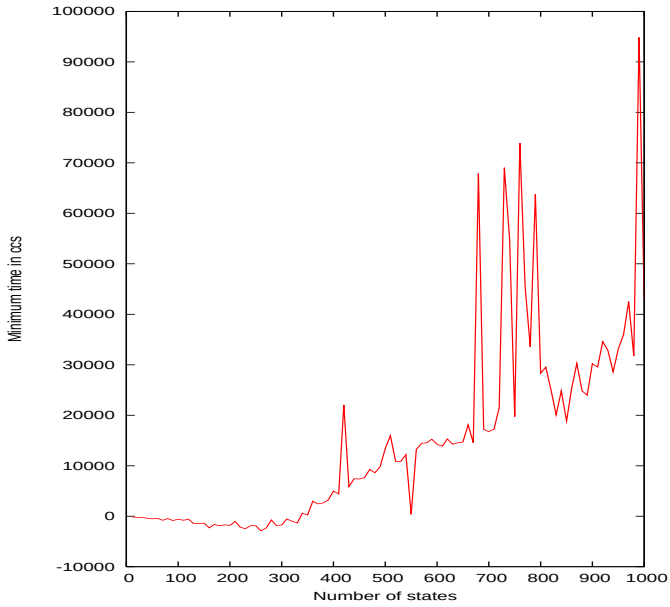


Figure 3: HC-TD performance aphabet size = 25

may be the suitable implementation approach. Since in practice, various strings of same size may require different processing time, it is of importance to conduct experiments in order to cross-compare the time efficiency for both algorithms. The following subsections depict various such experiments.

3.2 The single jump experiments

In this subsection, experiments were performed on strings that keep the FA only on a single state. The recognizer only jumps once on a single states and remains there for the entire recognition process. Thus the title *single jump experiment* suggested by the HC algorithm. The section summarizes experiments already suggested in [Kwk04]. Let consider the automaton modelled in Figure 4 having 5 states with two accepting states 3 and 4. The strings *abab* and *cdef* of size 4 are both part of the language modelled by the automaton. In theory, the total time required to accept or reject each string should be roughly the same. However, we notice that for the string *abab*, once the device reads the first symbol *a*, it jumps to the final state 3 and remains there until the entire string is processed. On the other hand, the recognizer will transverse several different states in order to accept the string *cdef*. This observation indicates that in practice, the time required to accept strings of same size with different patterns may differ considerably from one another. For the experiments, we randomly generated various automata of sizes between 10 and 1000 states using alphabet size varying between 10 and 50. Figure 5 depicts the difference in time between

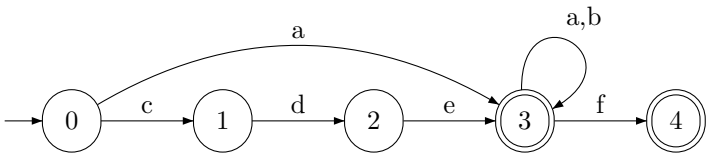


Figure 4: A state diagram: accepts the strings *"abab"* and *"cdef"*

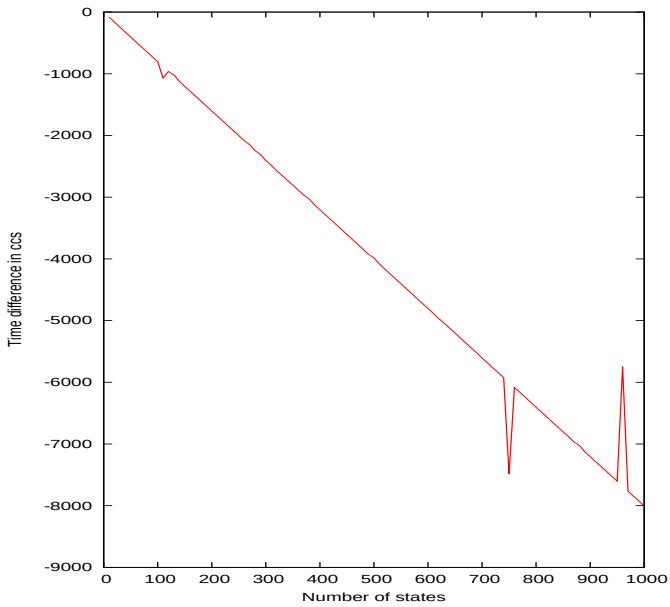


Figure 5: HC-TD performance for single jump

the table-driven and hardcoded implementation. It clearly shows that the HC algorithm is superlinearly faster than its TD counterpart. The average time efficiency per symbol is about 8 ccs. That is, for an automaton with n states, the HC algorithm is $8n$ times faster than the TD algorithm. These results illustrate that for strings following the pattern above described, the processor has sufficient space in its cache to hold the code relating to a single state. Since it always visits the same state over and over, there is a very low probability of cache misses. The experiment is therefore the best case scenario for both algorithms although the HC algorithm appears to be the most efficient. In the next subsection we explore another variant of such strings whereby the automaton visits only the starting state and the final state.

3.3 The far jump experiments

The experiments was suggested by the HC algorithm in the sense that from its initial state, the recognizer jumps to the final state and loops between the two states during the entire recognition process. The state diagram in figure 6 depicts such strings. We consider two strings $ab...ab$ and $cd...ef$ of length $n-1$ that are both accepting strings. The recognition process of the second string requires that the recognizer visits several states in the automaton whereas the first string only visits two states. Unlike the fact that far jumps are required to

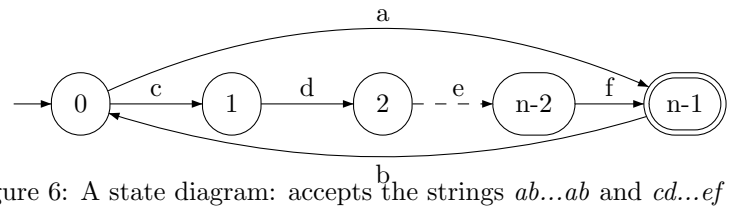


Figure 6: A state diagram: accepts the strings $ab...ab$ and $cd...ef$

move from the initial states to the final state for the first string, only limited number of states are visited. Therefore, we expect that such strings are tested at a very efficient time due to the very low probability of cache misses the processor is subject to. As show in figure 7, experiments revealed that the HC algorithm still outperforms its TD counterpart in such context. As a result to this, although in an average case behaviour there is a threshold of efficiency of HC over TD, there is still room for improvements above that threshold when some string patterns are considered. In the next section, we use the original

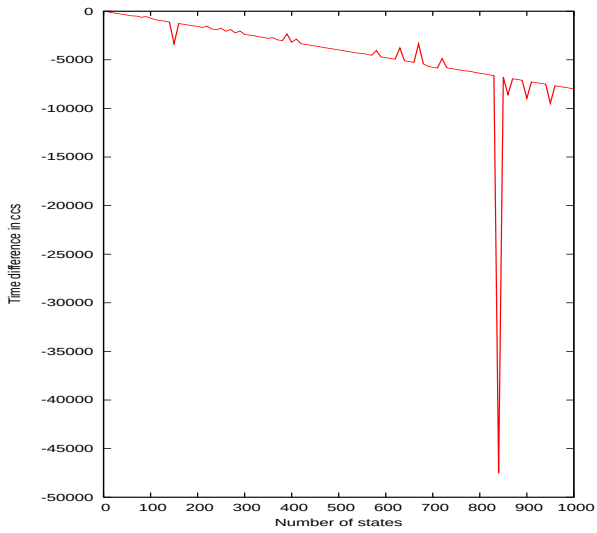


Figure 7: Performance HC-TD for strings that loop on two states

threshold of efficiency to conduct a particular experiment whereby the string remains on a single state after visiting some random states below the threshold.

3.4 A random string experiment followed by single jump

This section summarizes experiments already discussed in [Kwk04]. The random string experiments suggested that the HC outperformed the TD algorithm up to some threshold. In this section, we combine the threshold of efficiency with the experiment conducted in subsection 3.2. Consider the automaton modelled in figure 8. State t depicts some "sink state" in which the FA will remain after some other arbitrary set of states within the automaton have been visited. The state t is also assumed to be an accepting state. Based on the previous experiment, we would expect that the longer the FA remained in state t, the more the hardcoded implementation would enjoy an advantage over the table-driven version.

To verify this observation, and as a sort of sanity check on our results up to this point, hardcoded and table-driven implementations were set up to test strings of length *n-1* in FAs with n states. The experiment was designed so that the behaviour in processing the first 300 states was random - in the same sense as previously described. However, thereafter the FA remains in the same state – i.e. the best case scenario prevails.

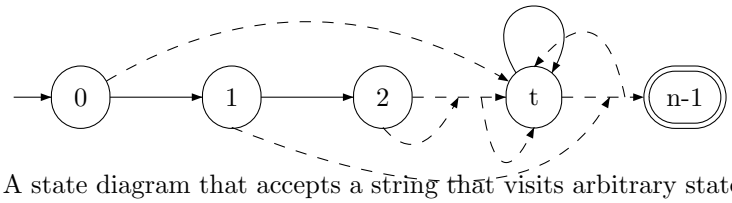


Figure 8: A state diagram that accepts a string that visits arbitrary states and remains on state t for some time

Figure 9 depicts the graphs obtained from the experiment. Unsurprisingly, it shows that hardcoding generally outperforms the table-driven implementation. However, there is also a suggestion in the data that in the longer term, the asymptotic improvement tends towards the 8ccs improvement observed in figure 5.

Of course, many more experiments similar to those described above could be run. An overall and general observation in regard to all these experiments is that they enable us to identify various ways in which the hardcoded implementation of FAs may outperform the traditional table-driven implementation. The next section considers how such information could be used to capitalize on the advantages offered by both approaches.

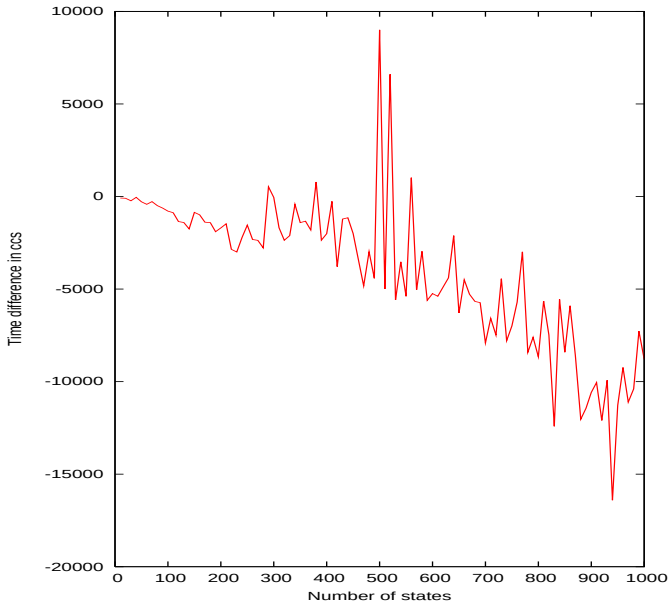


Figure 9: Comparison based on limited access on states

4 The Dynamic Implementation of FAs: DIFAP

The experiments depicted in the previous sections clearly show that the efficiency of a string recognizer is highly dependent on the nature of the string being recognized. This suggests that if likely patterns of strings to be input are known in advance – at least in some probabilistic sense – then it may be possible to put in place a time optimizing mechanism to carry out the string recognition. Consequently, the idea of dynamically adapting the implementation strategy of the FA according to the expected input (or partially inspected) string may be considered. We use the acronym DIFAP to refer to this notion, designating Dynamic Implementation of FAs for Performance enhancement. Figure 10 depicts the overall design of a DIFAP system. When it is first invoked, the implementer provides the specification of the automaton to be used, regardless the type of string to be recognized. DIFAP then analyzes the specification and choose the appropriate way of implementing the automaton depending of the size of the derived automaton. In terms of the currently available data, if the size is less than 360 states, this means that hardcoding is likely to be the optimal approach in representing the automaton irrespective of the kind of string to be tested. On the other hand, if the size of the automaton is above 360 states, as suggested in Section 2, a hardcode implementation might be indicated if long term behaviour is likely to tend towards best case behaviour, a table-driven implementation will be the appropriate choice in the absence of such information. However, in the latter case, DIFAP relies on the kind of string received as input to adapt it-

self progressively to an implementation approach that is optimal in some sense (e.g. optimal in relation to the history of strings processed to date), resulting in improved average processing speed.

The figure indicates that for bigger automata size, a knowledge table (KT) is first checked. At this stage, we do not prescribe what information should be kept in the KT. We merely observe that the current input string could, in principle, undergo some preliminary scan to identify whether its overall structure conforms to some set of general patterns that favour hardcode over table-driven. If that is not the case, the table-driven version of the automaton specification is generated and is used by the recognizer to check whether the string is part of the language described by the FA or not. Otherwise, the hardcoded version of the FA's specification is generated and used for recognition.

Not indicated in the figure is the possibility of post-processing: after a string has been tested, the string and the test outcome could be used to update information in the KT. As a very simple example, we might decide to concretely implement the KT as a table of the FA's states, in which a count is kept of the number of times a state has been visited. This information could be used to rearrange the order of rows (which represent states) in the transition matrix used by the table-driven approach, in the hope of minimizing data cache misses when this implementation strategy is used. Alternatively, the same information could be used to dictate the blocks of hardcode that should preferentially be loaded into cache, in circumstances in which hardcoding is indicated. However, the foregoing should not be construed as the only way in which the KT can be implemented. We conjecture that there are many creative possibilities within this broad model that merit deeper investigation in the future.

One of the advantage of using such a dynamic algorithm is that the structure of the automaton does not always remains in the system after processing. Each automaton is always regenerated into its executable when the system is invoked. The only structure that permanently remains in the system is the algebraic specification of the automaton. This results therefore in some degree of minimization of memory load for automata of considerable size. However, there is no need to always regenerate automata of size less than 360 states since they will always be implemented in hardcode. That is the reason why in the figure no deletion of the generated hardcode is indicated when the "size less than 360" path is followed.

In an implementation of DIFAP, attention should be given to the following parts of the algorithm to minimize latencies:

- *Time to generate the recognizer:* Unless directly implemented by hand, any FA-related problem always requires a formal specification of the grammar that describes the automaton before its corresponding automaton is encoded. This is a general problem, and one specific to DIFAP. The DIFAP implementation could therefore use generator techniques similar to those used in efficient code generator tools such as YACC[2] which as been proven to be amongst the best tool available to create directly executable

[2]Yet Another Compilers Compiler

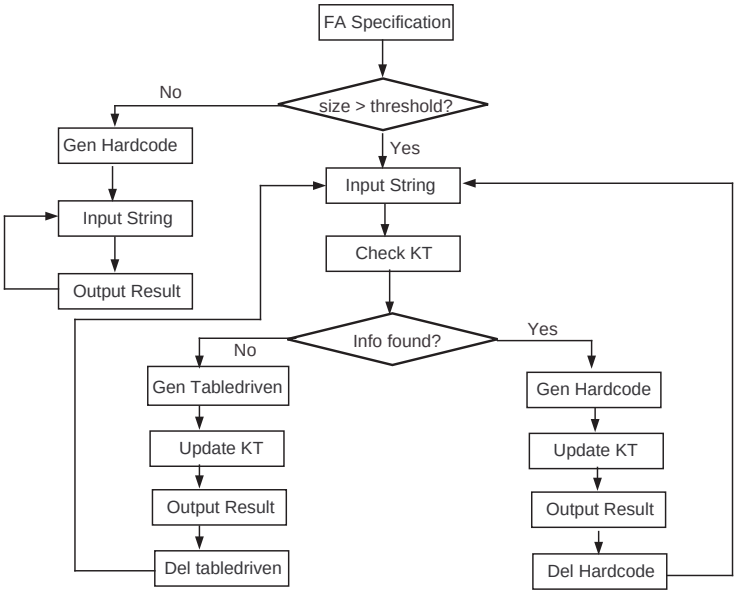


Figure 10: A Preliminary design of DIFAP

parsers. Unlike parsers, DIFAP's code generator will generate directly executable string recognizers.

- *Time taken to check the knowledge table:* One should take care to ensure that the matter of checking the KT does not degenerate into a time-inefficient exercise that negates any benefit from using the optimal string recognition strategy. Efficient algorithms should be devised that take minimal time to access the table and to chose the appropriate path to follow. This part of DIFAP may constitute a bottleneck. Intensive investigations will be made to provide an efficient approach to access the table and retrieve appropriate information.

- *Time required to update the knowledge table:* Although the precise nature and scope of the KT have not been identified here, it is envisaged that it will, itself, be a dynamic structure, changing over time in relation to the history of strings analyzed to date. However, there does not appear to be any reason for adapting the KT prior to processing the input string. Its update is something that can happen at a post-processing stage, and does not appear to be time-critical.

4.1 Structure of KT

The entire DIFAP algorithm relies on the KT in order to provide optimal information to the entire framework for efficient processing of a given recognizer. Various strategies can be used for its representation. Of course the most important notion to bear in mind must be efficiency. Since intensive investigations have not yet been made in order to test any of the strategy we have in mind. We have chosen to solve the problem in a step by step fashion so that each idea is eliminated from our list of choice once its weakness is noticeable. Up to date, we have identified three approaches by which the KT can be represented for optimal processing:

- *The KT is a single variable with recent nodes visited:* Our DIFAP algorithm heavily rely on the number of states visited by the automaton in order to take relevant action on whether to use TD or KT algorithm. In some applications, the kind of strings input might be predefined. We chose therefore to use a single variable to represent the KT since it will only contain the most recent number of states visited by the recognizer. In other words, during the recognition process, the number of states visited is recorded and then saved in the KT variable at the *update* of the KT. We then use this information for later processing. The Algorithm checks the value in the KT and takes relevant decision. If the value is zero, it means that no state have been visited and therefore the suitable algorithm should be the TD. If the number of states is less than or equal to the threshold of efficiency established above, then HC is the suitable algorithm. Otherwise, we use the TD algorithm. This approach seems

to be straightforward and very simple. However, one of its major drawback is that it is highly unpredictable as may result to a very inefficient framework.

- *The KT is a single variable with average nodes visited:* This approach is an alternative to the above. Using average node visited instead of most recent nodes visited reduce the probability of unpredictability but does not solve the entire problem. The approach is still therefore highly unpredictable.

- *The KT is an history structure:* This approach is still under investigation. We envisage it to be a structure containing detailed information on the history of each nodes of the automaton. The structure is aimed to be dynamic in the sense that, not all states can have a history if they have not yet been visited. The overall idea is to define a number of category of nodes such as *most likely to be visited*, *likely to be visited* and *unlikely to be visited*. The rate at which a state is visited is updated on a regular basis during the recognition process whether TD or HC was chosen. For each calculated rate of *visits*, the state can fall under each of the above defined category at any stage of the recognition. The role of the *check KT* routine will then be to probabilistically evaluate the number of states that are likely to be visited and take relevant action. Of course, the routine is still at a brainstorming stage and requires more though. However, this might be a better way to overcome efficiency problem of DIFAP as defined above.

5 Conclusion and Future Work

In this paper, we have made a review of the performance evaluation of both HC and TD algorithms already present in the literature. The HC algorithm outperforms the TD algorithm up to some threshold. Moreover, unlike the fact that the theoretical complexity of a recognizer is linear to the length of the input string. We have shown that the way the states of the automaton are visited at run-time plays a major role on the overall processing time. The more a state is visited the less he cache misses and the less the processing time. This observation helped us to have an idea on the way the KT of DIFAP can be implemented. We use a probabilistic concept to calculate the overall rate at which states are visited. This therefore yield to the actual implementation of DIFAP. However, many other challenges are still under investigation in order to make DIFAP a very flexible and efficient framework. The most important is the problem of mixed mode implementation of FAs whereby TD and HC are use simultaneously in order to provide a balance execution environment that uses small tables and small code.

References

[Kim02] Paul Kimmel. The Visual Basic .Net Developper's Book. Addison-Wesley, 2003.

[Ket03] E. Ketcha Ngassam. Hardcoding Finite Automata. MSC Dissertation. University of Pretoria, 2003.

[Kmp77] D. E Knuth and J.H Morris, Jr and V. R. Pratt. Fast Pattern Matching in Strings. SIAM J. Comput. Volume 6, 323-350, 1977.

[Kwk03a] E. Ketcha Ngassam, Bruce. W. Watson, and Derrick. G. Kourie, Preliminary Experiments in Hardcoding Finite Automata, Poster paper, CIAA, Santa Barbara, 299-300, September 2003.

[Kwk03b] E. Ketcha Ngassam, Bruce. W. Watson, and Derrick. G. Kourie, Hardcoding Finite State Automata Processing, SAICSIT, Johannesburg, 111-121, September 2003.

[Kwk04] E. Ketcha Ngassam, Bruce. W. Watson, and Derrick. G. Kourie, A Framework for the Dynamic Implementation of Finite Automata for Performance Enhancement, PSC, Prague, September 2004.

[Nwk03] Noud De Beijer, Bruce W. Watson and Derrick G. Kourie, Stretching and Jamming of Automata, SAICIST, Johannesburg, 198-207, September 2003.

[Nr02] Gonzalo Navarro, and Mathieu Raffinot. Flexible Pattern Matching In String: Practical on-line search for texts and biological sequences. Cambridge University Press 2002.

[Wat95] Bruce W. Watson. Taxonomies and Toolkits of Regular Languages Algorithms. PhD Thesis. Technical University of Eindhoven, 1995.

Stretching and Jamming of Finite Automata

Noud de Beijer* Derrick G. Kourie† Bruce W. Watson*†

Abstract

In this paper we present two transformations on automata, called stretching and jamming. These transformations will, under certain conditions, reduce the size of the transition table, and under other conditions reduce the string processing time. Given a finite automaton, we can stretch it by transforming each single transition into two or more sequential transitions, thereby introducing additional intermediate states. Jamming is the inverse transformation, in which two or more successive transitions are transformed into a single transition, thereby removing redundant intermediate states. We will present formal definitions of stretching and jamming and we will calculate theoretical bounds, when stretching/jamming is effective in terms of memory consumption.

1 Introduction

In this paper we present two new transformations on automata. A classical application area of automata theory is compiler construction.

In a compiler, a lexical analyzer is used to read input characters and to produce as output a sequence of tokens that the parser uses for syntax analysis [ASU86, p. 84]. Since the process of lexical analysis occupies a reasonable portion of the compiler's time, the lexical analyzer should minimize the number of operations it performs per input character [ASU86, p. 144]. The lexical analyzer uses finite automata to recognize languages. This finite automaton uses a transition function to process strings but there are different ways to implement this transition function.

The easiest and fastest way is to use a transition table in which there is a row for each state and a column for each input symbol. Unfortunately, this representation can take up a lot of space [ASU86, p. 114]. Of course, next to compilers there are numerous other applications in computing science where automata are used.

Thus, transformations on automata that increase their performance in terms of memory consumption or string processing time are potentially useful (see for example [Wat95]). We propose two transformations on automata: stretching and jamming. Under certain conditions, these transformations will produce more efficient automata in terms of memory consumption and string processing time.

*Department of Mathematics and Computer Science, Technische Universiteit Eindhoven, P.O. Box 513, 5600 MB Eindhoven, The Netherlands

†Department of Computer Science, University of Pretoria, Pretoria 0002, South Africa

Given a deterministic finite automaton (DFA), we can stretch it by transforming each single transition into two or more sequential transitions, thereby introducing additional intermediate states. For example, an ASCII DFA can be stretched by transforming each single ASCII (8-bit) character transition into two transitions, each of 4-bit characters.

Jamming is the inverse transformation, in which two successive transitions (based on, for example, input characters represented in 8-bits) are transformed into a single transition. This single transition will then be based on an input character represented by 16-bits. The same transformations can be used on a nondeterministic finite automaton (NFA).

2 Preliminaries

In this section we present the basic notions and notations used in this paper. Most of the notations used are standard (see for example [HMU01]) but a few new notations are introduced.

A deterministic finite automaton, *DFA*, is a 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of *states*, Σ is the *alphabet*, $\delta : Q \times \Sigma \nrightarrow Q$ is the (partial) *transition function*, q_0 is the *initial state* and F is a subset of Q whose elements are *final states*. $|Q|$ is the number of states and $|\Sigma|$ is the number of elements in the alphabet, or *alphabet size*.

The *n-closure* of an alphabet is the set of all symbols that consist of concatenating n symbols from Σ. Σ^+ is the *plus-closure* of the alphabet, the set of symbols obtained by concatenating one or more symbols from Σ.

$|Q||\Sigma|$ is the theoretical *transition table size*. Note that since cells represent states, the minimum cell size is determined by the minimum space requirements to represent a state, which is in turn determined by the total number of states. Although stretching and jamming will change the number of states in an automaton we will assume that the transition table cell size does not change in either transformation. We expect that in most cases the practical effects of this assumption are unlikely to be significant. Preliminary benchmarking results indicate that our theoretical transition table size is indeed a good estimate for the real transition table size.

A transition in a DFA M from p to q with label a will be denoted by (p, a, q) where $(p, a, q) \in Q \times \Sigma \times Q$ and $q = \delta(p, a)$. We will also use the notation $((p, a), q) \in \delta$.

A *path* of length k in a DFA M is a sequence $\langle (r_0, a_0, r_1), \ldots, (r_{k-1}, a_{k-1}, r_k) \rangle$, where $(r_i, a_i, r_{i+1}) \in Q \times \Sigma \times Q$ and $r_{i+1} = \delta(r_i, a_i)$ for $0 \leq i < k$. The *string*, or *word* $a_0 a_1 \cdots a_{k-1} \in \Sigma^k$ is the *label* of the path.

The *extended transition function* of a DFA M, $\hat{\delta} : Q \times \Sigma^+ \nrightarrow Q$, is defined so that $\hat{\delta}(r_i, w) = r_j$ iff there is a path from r_i to r_j, labeled w.

A nondeterministic finite automaton, *NFA*, is a 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$, defined in the same way as a DFA, with the following exception: $\delta : Q \times \Sigma \nrightarrow \mathcal{P}(Q)$ is the transition function. Note that $\mathcal{P}(Q)$ is the powerset of Q. For present purposes, ϵ-transitions can be ignored without loss of generality.

A transition in an NFA M from p to q with label a will also be denoted by (p, a, q) where $(p, a, q) \in Q \times \Sigma \times Q$ and $q \in \delta(p, a)$.

A path of length k in an NFA M is a sequence $\langle (r_0, a_0, r_1), \ldots, (r_{k-1}, a_{k-1}, r_k) \rangle$, where $(r_i, a_i, r_{i+1}) \in Q \times \Sigma \times Q$ and $r_{i+1} \in \delta(r_i, a_i)$ for $0 \leq i < k$.

In an NFA M, the extended transition function, $\hat{\delta} : Q \times \Sigma^+ \nrightarrow \mathcal{P}(Q)$, is also defined so that $r_j \in \hat{\delta}(r_i, w)$ iff there is a path from r_i to r_j, labeled w.

To stretch a transition we need to split up a single symbol in 2 or more subsymbols. Therefore, we conceive of alphabet elements as strings of *subelements* (typically bit substrings). If alphabet element $a \in \Sigma$ has length $|a|$ then we number the subelements $a.0, \ldots, a.(|a| - 1)$. Thus, if $a = 0111$ then $a.0 = 0$, $a.1 = 1$, $a.2 = 1$ and $a.3 = 1$.

By definition, a word is a string of symbols over an alphabet. We use the same notation to number the individual symbols of a word. So, for the word $w \in \Sigma^k$, we number the individual symbols $w.0 \cdots w.(k-1)$.

Because in our paper it will always be clear whether w is a word or a single symbol, no conflicts will rise because of this definition.

3 Formal Definitions

3.1 General Definitions

In this section we give formal definitions of the stretching and jamming operations. One way in which we can stretch an automaton is by transforming each transition into k sequential transitions. This stretching operation on a single transition is pictured in figure 1.

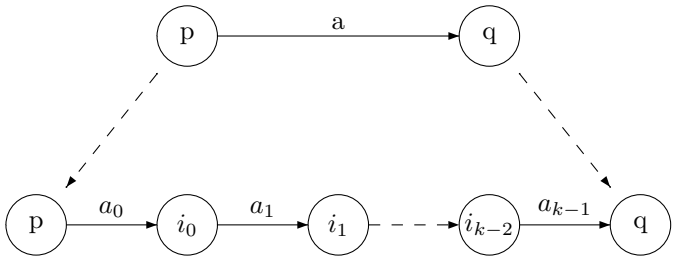


Figure 1: Stretching transition (p, a, q) into k sequential transitions.

In this example we see that transition (p, a, q) is stretched into k sequential transitions and $k - 1$ new states are introduced. In this sequence of transitions we call p and q the *original states*, and $i_0, \ldots, i_{k-2}$ the *additional intermediate states*.

Jamming is the inverse transformation, in which k sequential transitions are transformed into a single transition. In figure 1 this can be seen as performing a transformation in the opposite direction to the stretching operation.

This means that the intermediate states are removed. In the case of jamming we call these states *redundant intermediate states*.

We can stretch NFAs as well as DFAs. We will define the stretching transformation on NFAs, and the result of a transformation will also be an NFA. Because DFAs are a subset of NFAs, the stretching of DFAs is automatically defined. If we stretch a DFA, in some cases the resulting automaton may have more than one transition with the same label from a given state and therefore

the result of stretching a DFA might be an NFA. In section 3.2 we will present an example that will clarify this.

If NFA FA_0 can be stretched into NFA FA_1, we call FA_1 *a stretch* of FA_0. The set of states of FA_1 consists of a subset S_1 of original states and a subset I of newly introduced additional intermediate states. Definition 3.1 formally describes a general stretching transformation.

Firstly, there is an injection τ from the alphabet of FA_0, Σ_0 to Σ_1^+, the plus-closure of the alphabet of FA_1 (property 1). Secondly, there is a one-to-one relation φ between the original states of FA_0 and the original states of FA_1.

This bijection connects the start states of FA_0 and FA_1 (property 2). It also defines a one-to-one relationship between the final states of both automata (property 3). Property 4 states that for every transition from state p to q with label a in FA_0 there exists a path from $\varphi(p)$ to $\varphi(q)$ with label $\tau(a)$ in FA_1, which travels from a state in S_1 via a number of intermediate states to another state in S_1. The inverse of this property is also true, therefore property 5 also holds.

Definition 3.1. *Let* $\mathrm{FA}_0 = (S_0,\ \Sigma_0,\ \delta_0,\ q_0,\ F_0)$ *be an NFA, and let* $\mathrm{FA}_1 = (S_1 \cup I,\ \Sigma_1,\ \delta_1,\ q_1,\ F_1)$ *be an NFA.* FA_1 *is a stretch of* FA_0 *iff:*

- *There is an injection* $\tau : \Sigma_0 \to \Sigma_1^+$*, thus:*

$$- (\forall a_0, a_1 : a_0, a_1 \in \Sigma_0 : \tau(a_0) = \tau(a_1) \Rightarrow a_0 = a_1) \tag{1}$$

- *There is a bijection* $\varphi : S_0 \leftrightarrow S_1$*, with the following properties:*

$$- \varphi(q_0) = q_1 \tag{2}$$
$$- (\forall f_0 \in F_0 : (\exists f_1 \in F_1 : \varphi(f_0) = f_1)) \wedge |F_0| = |F_1| \tag{3}$$
$$- (\forall p, a, q : q \in \delta_0(p, a) : (\exists k, r_0, \ldots, r_k, w : r_k \in \hat{\delta}_1(r_0, w) : \varphi(p) = r_0 \wedge \varphi(q) = r_k \wedge \tau(a) = w)) \tag{4}$$
$$- (\forall k, r_0, \ldots, r_k, w : r_k \in \hat{\delta}_1(r_0, w) : \varphi(r_k) \in \delta_0(\varphi(r_0), \tau^{-1}(w))) \tag{5}$$

where $p, q \in S_0$*,* $a \in \Sigma_0$*,* $r_0, r_k \in S_1$*,* $r_1, \ldots, r_{k-1} \in I$*, and* $w \in \Sigma_1^k$ *for* $k \geq 1$*. Furthermore,* $\delta_1(r_i, w.i) = r_{i+1}$*, for* $0 \leq i < k$*.*

We define jamming as the inverse transformation of stretching. Because of symmetry, if we jam certain NFAs the result will be a DFA.

If NFA FA_0 is jammed into NFA FA_1 (FA_1 is *a jam* of FA_0) then FA_0 is a stretch of FA_1. The set of states of FA_0 consists of a subset S_0 of original states and a subset R of redundant intermediate states. These redundant intermediate states will be removed by the jamming transformation.

Definition 3.2. *Let* $\mathrm{FA}_0 = (S_0 \cup R,\ \Sigma_0,\ \delta_0,\ q_0,\ F_0)$ *be an NFA, and let* $\mathrm{FA}_1 = (S_1,\ \Sigma_1,\ \delta_1,\ q_1,\ F_1)$ *be an NFA.* FA_1 *is a jam of* FA_0 *iff* FA_0 *is a stretch of* FA_1*, with* R *being the set of additional intermediate states, resulting from stretching.*

3.2 Stretching and Jamming by a Factor f

In the previous section we presented definition 3.1. This definition is very general: every single transition can be stretched into a different number of sequential transitions, according to properties 4 and 5.

In the next section our major application of stretching and jamming will be introduced. To be able to use the definitions in that application, we need to make a restriction on the previous definition. In the definition below we only allow all transitions to be stretched into a fixed number of sequential transitions.

Therefore, we introduce the factor f in stretching and jamming. If NFA FA_0 is stretched by a factor f into NFA FA_1, we call FA_1 *an f-stretch* of FA_0. This means that the relation τ specializes to a one-to-one relationship between the alphabet of FA_0 and the f-closure of the alphabet of FA_1. Furthermore, for each transition in FA_0 there are exactly f sequential transitions in FA_1.

The formal differences between the new definition and the previous definition are expressed in properties 7, 10 and 11. In this definition, the relation τ is not only an injection, but because of property 7 also a surjection, and therefore a bijection. Also, because of property 10, for every transition (p, a, q) in the original NFA, there is a path of length f from $\varphi(p)$ to $\varphi(q)$ with label $\tau(a)$. Property 11 states that the inverse is also true.

Definition 3.3. *Let* $\mathrm{FA}_0 = (S_0,\ \Sigma_0,\ \delta_0,\ q_0,\ F_0)$ *be an NFA, and let* $\mathrm{FA}_1 = (S_1 \cup I,\ \Sigma_1,\ \delta_1,\ q_1,\ F_1)$ *be an NFA.* FA_1 *is an f-stretch of* FA_0 *iff:*

- FA_1 *is a stretch of* FA_0*, such that the injection* τ *specializes to a bijection* $\tau : \Sigma_0 \leftrightarrow \Sigma_1^f$*, thus:*

  $$- (\forall a_0, a_1 : a_0, a_1 \in \Sigma_0 : \tau(a_0) = \tau(a_1) \Rightarrow a_0 = a_1) \tag{6}$$
  $$- (\forall w \in \Sigma_1^f : (\exists a \in \Sigma_0 : \tau(a) = w)) \tag{7}$$

- *The bijection* $\varphi : S_0 \leftrightarrow S_1$*, is characterized by:*

  $$- \varphi(q_0) = q_1 \tag{8}$$
  $$- (\forall f_0 \in F_0 : (\exists f_1 \in F_1 : \varphi(f_0) = f_1)) \wedge |F_0| = |F_1| \tag{9}$$
  $$- (\forall p, a, q : q \in \delta_0(p, a) : (\exists r_0, \ldots, r_f, w : r_f \in \delta_1(r_0, w) : \varphi(r_0) = p \wedge \varphi(r_f) = q \wedge \tau(a) = w)) \tag{10}$$
  $$- (\forall r_0, \ldots, r_f, w : r_f \in \delta_1(r_0, w) : \varphi(r_f) \in \delta_0(\varphi(r_0), \tau^{-1}(w))) \tag{11}$$

  where $p, q \in S_0$*,* $a \in \Sigma_0$*,* $r_0, r_f \in S_1$*,* $r_1, \ldots, r_{f-1} \in I$*, and* $w \in \Sigma_1^f$ *for* $f \geq 2$*.*
  Furthermore, $\delta_1(r_i, w.i) = r_{i+1}$*, for* $0 \leq i < f$*.*

Jamming by a factor f is defined analogously to stretching by a factor f. Note that, by definition, jamming by a factor f is not always possible. NFA FA_0 can only be jammed if there exists an NFA FA_0 which can be stretched into FA_0.

Definition 3.4. *Let* $\mathrm{FA}_0 = (S_0 \cup R,\ \Sigma_0,\ \delta_0,\ q_0,\ F_0)$ *be an NFA, and let* $\mathrm{FA}_1 = (S_1,\ \Sigma_1,\ \delta_1,\ q_1,\ F_1)$ *be an NFA.* FA_1 *is an f-jam of* FA_0 *iff* FA_0 *is an f-stretch of* FA_1*. The set* R *of* FA_0 *is the set of additional intermediate states, resulting from stretching.*

To illustrate these definitions we give an example:

Example 3.5. *The graph in figure 2 represents the DFA* $\mathrm{FA}_0 = (\{q, r, s\}, \{a, b, c, d\}, \delta_1, q, \{s\})$*. DFA* FA_0 *can be stretched by a factor 2 into NFA* FA_1 *of figure 3. NFA* $\mathrm{FA}_1 = (S_2 \cup I, \{e, f, g, h, i, j, k\}, \delta_2, t, \{y\})$*, with*

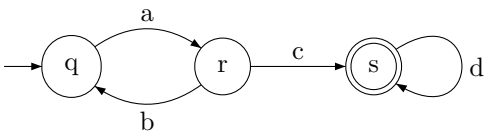


Figure 2: DFA FA_0

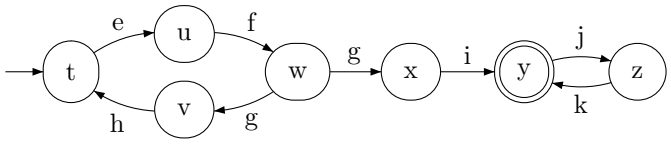


Figure 3: NFA FA_1, a 2-stretch of DFA FA_0

$$\begin{aligned} \tau(a) &= ef \\ \tau(b) &= gh \\ \tau(c) &= gi \\ \tau(d) &= jk \end{aligned}$$

Figure 4: Injection τ

$$\begin{aligned} \varphi(q) &= t \\ \varphi(r) &= w \\ \varphi(s) &= y \end{aligned}$$

Figure 5: Bijection φ

$S_2 = \{t, w, y\}$ and I, the set of additional intermediate states, is $\{u, v, x, z\}$. Injection τ and bijection φ are shown in figures 4 and 5 respectively.

In this example we can see why stretching a DFA can result in an NFA. Because we chose $\tau(b) = gh$ and $\tau(c) = gi$, there are two outgoing transitions with label g from state w. Therefore FA_1 *is an NFA.*

4 Bit-level Stretching and Jamming

4.1 Overview

In order to highlight how stretching and jamming can improve performance, we present an application of both transformations in this section. We look at stretching and jamming on a bit-level. We will only consider automata in which each element of the alphabet is a bit string. An *n-bit automaton* is an automaton whose alphabet consists of all the 2^n bit strings of length n.

Proposition 4.1. *Let f be a factor of n. Then we can f-stretch the n-bit DFA* FA_0 *into NFA* FA_1 *in the following way:*

- FA_1 *is an $\frac{n}{f}-$bit NFA.*
- *There is a bijection between Σ_0 and Σ_1^f ie. for every bit string of length n in Σ_0 there is a sequence of f bit strings of length $\frac{n}{f}$ in Σ_1^f and vice versa.*
- *For every transition in* FA_0 *there are f sequential transitions in* FA_1*, obeying the above bijection between Σ_0 and Σ_1^f for the labels of the transitions.*

Of course, this specialization of stretching is only allowed if n is divisible by f. In that case we call the DFA *f-stretchable*. Again, jamming is the inverse transformation: if an n-bit NFA is *f-jammable*, the resulting automaton is an $nf - bit$ DFA.

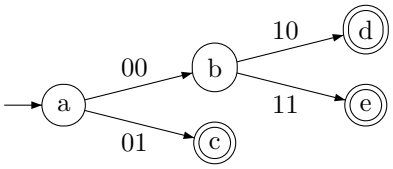


Figure 6: DFA FA_0

	00	01	10	11
a	{b}	{c}	-	-
b	-	-	{d}	{e}
c	-	-	-	-
d	-	-	-	-
e	-	-	-	-

Figure 7: Transition table of DFA FA_0

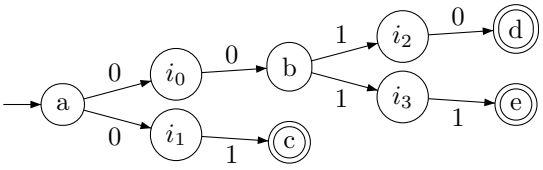


Figure 8: NFA FA_1, a 2-stretch of DFA FA_0

	0	1
a	$\{i_0,i_1\}$	-
i_0	{b}	-
i_1	-	{c}
b	-	$\{i_2,i_3\}$
i_2	{d}	-
i_3	-	{e}
c	-	-
d	-	-
e	-	-

Figure 9: Transition table of NFA FA_1

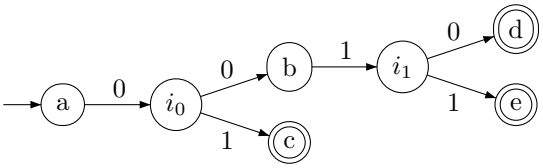


Figure 10: DFA FA_2, determinized NFA FA_1

	0	1
a	$\{i_0\}$	-
i_0	{b}	{c}
b	-	$\{i_1\}$
i_1	{d}	{e}
c	-	-
d	-	-
e	-	-

Figure 11: Transition table of DFA FA_2

Example 4.2. *To illustrate the stretching of n-bit automata we give an example. The 2-bit DFA* FA_0 *of figure 6 can be stretched by a factor* 2 *into the 1-bit NFA* FA_1 *of figure 8. Also, the 1-bit NFA* FA_1 *can be jammed into the 2-bit DFA* FA_0*. Furthermore, NFA* FA_1 *can be determinized into DFA* FA_2 *of figure 10.*

Note that in the previous example, we stretched a transition by using the most significant bit first. For example, we stretched transition $(a, 01, c)$ into $(a, 0, i_1)$ and $(i_1, 1, c)$, taking the 0 first and then the 1. In practice, we will usually use the least significant bit first, because that is the most natural way to process a bit string. It can be done as presented here in practice too, however.

4.2 Theoretical Results

In this section we will prove a number of propositions about stretching and jamming. From these propositions we can draw conclusions about when stretching or jamming will be useful in terms of memory consumption and string processing time. The first four propositions will deal with bit-level stretching. The following propositions hold:

Proposition 4.3. $|\Sigma_1| = \sqrt[f]{|\Sigma_0|}$

Proof. If FA_0 is an n-bit NFA, then the alphabet size, $|\Sigma|$, is 2^n. If FA_0 is stretched by a factor f into FA_1, FA_1 is an $\frac{n}{f}$-bit NFA, so the alphabet size, $|\Sigma_1|$, is $2^{\frac{n}{f}} = \sqrt[f]{2^n} = \sqrt[f]{|\Sigma_0|}$. □

Proposition 4.4. $|Q_1| = |\delta_0|(f-1) + |Q_0|$

Proof. Stretching by a factor f introduces $f-1$ additional intermediate states, for each single transition in the original DFA. Therefore $|Q_1|$ is equal to the additional intermediate states, $|\delta_0|(f-1)$, plus the number of states in the original DFA, $|Q|$. □

Proposition 4.5. $|Q_0| \leq |Q_1| \leq |Q_0||\Sigma_1|(f-1) + |Q_0|$

Proof. From proposition 4.4 we know that if DFA FA_0, with $|\delta_0|$ transitions, is stretched by a factor f into FA_1, $|Q_1| = |\delta_0|(f-1) + |Q_0|$. The number of transitions is at least 0 and at most $|Q_0||\Sigma_0|$. Therefore, $|Q_0| \leq |Q_1| \leq |Q_0||\Sigma_1|(f-1) + |Q_0|$. □

Proposition 4.6. *Let* $z = \frac{|Q_0|(|\Sigma_0| - \sqrt[f]{|\Sigma_0|})}{\sqrt[f]{|\Sigma_0|}(f-1)}$.

$$
\begin{array}{lcl}
|\delta_0| < z & \Leftrightarrow & |Q_1||\Sigma_1| < |Q_0||\Sigma_0| \\
|\delta_0| = z & \Leftrightarrow & |Q_1||\Sigma_1| = |Q_0||\Sigma_0| \\
|\delta_0| > z & \Leftrightarrow & |Q_1||\Sigma_1| > |Q_0||\Sigma_0|
\end{array}
$$

Proof. From proposition 4.3 we know that if DFA FA_0 is stretched by a factor f into FA_1, $|\Sigma_1| = \sqrt[f]{|\Sigma_0|}$. From proposition 4.4 we know that if DFA FA_0, with $|\delta_0|$ transitions, is stretched by a factor f to into FA_1, $|Q_1| = |\delta_0|(f-1) + |Q_0|$. Thus, $|Q_1||\Sigma_1| = (|\delta_0|(f-1) + |Q_0|)\sqrt[f]{|\Sigma_0|}$.
Therefore, if $|\delta_0| = z = |Q_0|(|\Sigma_0| - \sqrt[f]{|\Sigma_0|})/\sqrt[f]{|\Sigma_0|}(f-1)$ then:

$$
\begin{array}{ll}
|Q_1||\Sigma_1| & = ((|Q_0|(|\Sigma_0| - \sqrt[f]{|\Sigma_0|})/\sqrt[f]{|\Sigma_0|}(f-1))(f-1) + |Q_0|)\sqrt[f]{|\Sigma_0|} \\
& = |Q_0||\Sigma_0|
\end{array}
$$

The inequalities follow directly by similar reasoning. □

From proposition 4.6 we can conclude that bit-stretching a DFA reduces the transition table size when $|\delta_0| < z$. Therefore it can reduce the amount of memory needed for a transition table representation.

The *transition density* of an automaton is the number of transitions divided by the transition table size (the maximum number of possible transitions). For our DFA FA_0 this is $|\delta_0|/|Q_0||\Sigma_0|$. In the last proposition we saw that if the number of transitions $|\delta_0|$ is equal to z, the transition table size is the same before and after stretching. This means that the density in that case is $z/|Q_0||\Sigma_0| =$

$(\frac{1}{|\Sigma_0|} - |\Sigma_0|^{-\frac{1}{f}})/(1-f)$. This value is not dependent on the number of states, $|Q_0|$. In figure 12, $z/|Q_0||\Sigma_0|$ is set out against the alphabet size $|\Sigma_0|$ for different values of f. From these graphs we can conclude how low the transition density has to be to obtain a smaller transition table size by stretching. For example, if we stretch an 8-bit DFA (256 alphabet symbols) by a factor 2 (f=2), we reduce the transition table size if the transition density is lower than 6%.

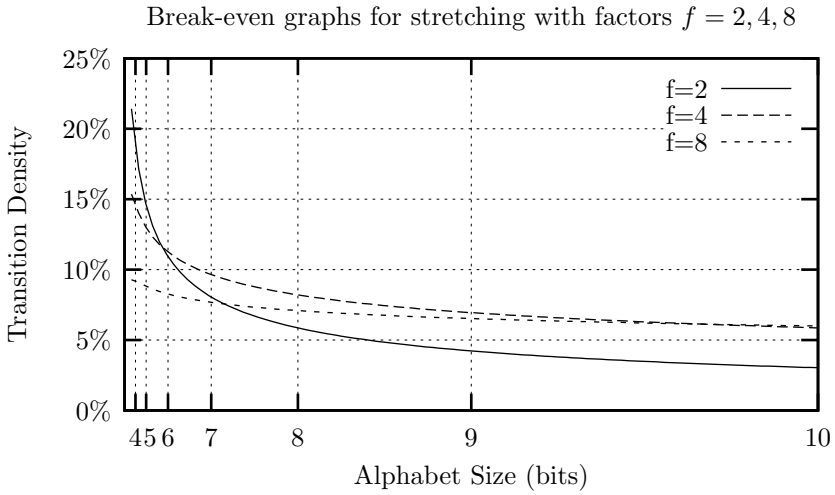


Figure 12: Break-even graphs for stretching

The next three propositions will deal with bit-level jamming.

Proposition 4.7. $|\Sigma_1| = |\Sigma_0|^f$

Proof. If FA_0 is an n-bit DFA, then the alphabet size, $|\Sigma_0|$, is 2^n. If FA_0 is jammed by a factor f into FA_1, FA_1 is an nf-bit DFA, so the alphabet size, Σ_1, is $2^{nf} = (2^n)^f = |\Sigma_0|^f$. □

Proposition 4.8. $0 \leq |Q_1| \leq |Q_0|$

Proof. Every automaton has at least 0 states, thus $|Q_1| \geq 0$.
Jamming removes all redundant intermediate states so $|Q_1| \leq |Q_0|$. □

We could include a proposition here that states in which cases jamming results in a smaller transition table. Unfortunately, in almost all cases jamming results in a larger transition table. However, one of the reasons for investigating the jamming operation is that it might reduce the string processing time because multiple symbols are processed at once.

This concludes our overview of bit-level stretching and jamming. We end this section with a few notes.

Of course, at this point the question rises how stretching and jamming perform in practice. We have implemented the stretching and jamming operations as algorithms and performed benchmarking studies under different conditions. These benchmarking studies confirm our theoretical results. Furthermore, these

studies show that jamming indeed reduces the string processing time in most cases. The algorithms as well as the results of the benchmarking studies can be found in [dB04].

In this paper, we will only consider stretching or jamming the complete transition table. As we saw in this section, stretching is useful if the transition density is low. Therefore, it might be interesting to stretch only certain parts of the transition table where the transition density is low. For jamming we can argue that if a certain part of the transition table contains many redundant states, jamming this part only might be interesting. We call this approach local stretching and jamming but leave its details as future work.

Furthermore, we will only consider transition tables that can be implemented with a regular matrix that has a row for each state and a column for each input symbol. Of course, there are cases where this does not apply. For example, sometimes character classes or sparse matrices are used for implementing the transition table. We will not discuss these situations in this paper.

As a general guideline however, if a certain transition table implementation leads to a smaller transition table density it will have a positive effect on stretching. If it leads to a higher transition density it will have a negative effect.

5 Conclusions and Future Work

We have defined the notions of stretching and jamming and shown the theoretical conditions under which they influence performance. In the case of stretching, performance can be improved by reducing the memory usage. Jamming on the other hand, increases memory usage.

During the theoretical discussion of jamming we hinted that it might reduce the string processing time but we did not present a theoretical model for that. However, preliminary benchmarking studies in [dB04] confirm our theoretical results and show that jamming indeed reduces the string processing time.

There are a number of interesting problems that can still be investigated further. We only considered stretching or jamming the complete transition table. Transforming only a small part of the transition table, in other words local stretching and jamming, is an interesting problem for further research.

Of course, any method that is used to change the transition table, for example character classes and sparse matrices, influences stretching and jamming. Therefore, these situations can be investigated further.

Lastly, we only looked into transforming automata and not regular expressions. The stretching and jamming of regular expressions is also a candidate for further research.

References

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers : principles, techniques and tools.* Addison-Wesley, 1986.

[dB04] A.A. de Beijer. Stretching and jamming of automata. Master's thesis, Technische Universiteit Eindhoven, The Netherlands, 2004.

[HMU01] J.E. Hopcroft, R. Motwani, and J.D. Ullman. *Introduction to automata theory, languages, and computation.* Addison-Wesley, 2001.

[Wat95] B.W. Watson. *Taxonomies and toolkits of regular language algorithms.* PhD thesis, Eindhoven University of Technology, The Netherlands, 1995.