ELSEVIER

Annals of Pure and Applied Logic 113 (2002) 59–80

ANNALS OF PURE AND APPLIED LOGIC

www.elsevier.com/locate/apal

Window-accumulated subsequence matching problem is linear ☆

Luc Boasson[a], Patrick Cegielski[b], Irène Guessarian[c,*,1], Yuri Matiyasevich[d,2]

[a] *LIAFA, Université Paris 7, 2 Place Jussieu, 75254 Paris Cedex 5, France*
[b] *LACL, Université Paris 12, Route forestière Hurtault, F-77300 Fontainebleau, France*
[c] *LIAFA, UMR 7089 and Université Paris 6, 2 Place Jussieu, 75254 Paris Cedex 5, France*
[d] *Steklov Institute of Mathematics, Fontanka 27, St. Petersburg, Russia*

Abstract

Given two strings, text t of length n, and pattern $p = p_1 \dots p_k$ of length k, and given a natural number w, the subsequence matching problem consists in finding the number of size w windows of text t which contain pattern p as a subsequence, i.e. the letters $p_1, \dots, p_k$ occur in the window, in the same order as in p, but not necessarily consecutively (they may be interleaved with other letters). Subsequence matching is used for finding frequent patterns and association rules in databases. We generalize the Knuth–Morris–Pratt (KMP) pattern matching algorithm; we define a non-conventional kind of RAM, the MP-RAMs which model more closely the microprocessor operations; we design an $O(n)$ on-line algorithm for solving the subsequence matching problem on MP-RAMs.

Keywords: Subsequence matching; Algorithms; Frequent patterns; Episode matching; Datamining

1. Introduction

We address the following problem. Given a text t of length n and a pattern $p = p_1 \cdots p_k$ of length $k \leqslant w$, both from the alphabet A, and given a natural number w,

☆ A first version of this paper appears in proceedings of PODS'99, Principles Of Database System [3].

* Corresponding author.

E-mail addresses: lub@liafa.jussieu.fr (L. Boasson), cegielski@univ-paris12.fr (P. Cegielski), ig@liafa.jussieu.fr (I. Guessarian), yumat@pdmi.ras.ru (Y. Matiyasevich).

[1] Partly supported by EVOLUTION grant, http://www.prism.uvsq.fr/dataware/coop/evolution.html.

[2] Visiting Institut für Informatik, Stuttgart University, Germany as Humboldt Research Award Winner.

PII: S0168-0072(01)00051-3

find the number of size w windows of text t which contain pattern p as a subsequence so that the letters $p_1,\ldots,p_k$ occur in the window in the same order as in p though not necessarily consecutively because they may be interleaved with additional letters. We call this problem with arguments the size w, the text t, and the pattern p, the *Window-Accumulated Subsequence matching Problem* or in short *WASP*.[3]

The subsequence matching problem is an intrinsically interesting generalization of the pattern-matching problem. It has not attracted attention earlier because the plain subsequence matching problem, where one stops as soon as an occurrence of p is found (regardless of any window size), is easily solved in linear time: a finite state automaton with $k+1$ states $s_0,s_1,\ldots,s_k$ scans the text; its initial state is s_0; when it scans letter p_1 it goes in state s_1, then when it scans letter p_2 it goes in state $s_2,\ldots$; the text is accepted as soon as it reaches state s_k. Subsequence matching within a w-window is a more difficult problem, which emerged due to its applications in knowledge discovery and datamining (in short KDD) [15, 16], and as a first step for solving a problem in molecular biology [14, 13]. One quite important use of subsequence matching in KDD consists in recognizing frequent patterns in sequences of data. Knowledge of frequent patterns is then used to determine association rules in databases and to predict the behavior of large data [15, 16]. Consider for instance a text t consisting of a university WWW-server logfile containing requests to see WWW pages, and suppose we want to see how often, within a time window of at most 10 units of time, the sequence of events $e_1e_2e_3e_4$ has occurred, where: $e_1=$ 'Computer Science Department homepage', $e_2=$ 'Graduate Course Descriptions', $e_3=$ 'CS586 homepage', $e_4=$ 'homework'. This will be achieved by counting the number of 10-windows of t containing $p=e_1e_2e_3e_4$ as a subsequence. This example calls for three remarks.

1. An *on-line* analysis is much preferable to an *off-line* analysis. In the second case, we record the text, and we then read it back and forth to process it. In the first case, we read the text once, and immediately process it: that is, we have both (i) a bounded amount of working memory available (much smaller that the size of the text, hence we cannot memorize the whole text), and (ii) a bounded amount of time available between reading two consecutive text symbols.
2. The length k of the pattern is usually much smaller than the length n of the text, hence an $O(f(w,k)+n)$ processing time may be preferable to an $O(nk)$ processing time, even if $f(w,k)$ is a rapidly growing function.
3. The pattern is often reused, hence a preprocessing of the pattern, done only once, can pay off in the long run. Moreover the preprocessing can be done during 'off-hours' and it is important to have an answer as fast as possible while a user is querying a text during 'peak-hours'.

The contribution of this paper is twofold: (i) we design two new efficient algorithms (Sections 3.2 and 3.3) for solving the subsequence matching problem, (ii) in so doing,

[3] The *WASP* is called *episode matching* in [9], and *serial episode matching* in [16]; we use *subsequence matching* in order to follow the terminology of [2]. A closely related problem is the *matching with don't cares* of [14, 13].

we establish two intrinsically interesting results (a) and (b) stated below. Let L be the language consisting of the strings t where pattern p occurs as a subsequence within a w-window.

(a) In Theorem 2 we *build the minimal finite state automaton* accepting L; this yields an $O(f(w,k)+n)$ on-line algorithm for solving the subsequence matching problem, with $f(w,k)$ exponential in k, w.

(b) We *introduce a nonconventional kind of RAMs*, the *MP-RAM*s, which are interesting *per se* because they model more closely the microprocessor basic operations. In Theorem 3 we show that the transitions of a finite state automaton accepting L can be encoded in such a way as to be *efficiently* computed on an MP-RAM using only the basic (and fast) operations of shifting bits, binary AND and addition; this yields an $O(\log w + k + n)$ on-line algorithm for solving the subsequence matching problem. We checked that the MP-RAM-based algorithm is much faster in practice. We believe that, for other algorithms too, a speed-up will be achieved by programming them on the MP-RAMs that we define in Section 3.3.

Text searching problems have been extensively studied for a long time: most important are pattern-matching problems, which consist in searching for (possibly constrained) occurrences of 'small' patterns in 'large' texts. Pattern-matching algorithms fall in four categories:

– *off-line* algorithms reading text and/or pattern back and forth, or
– *on-line* algorithms which can be coarsely divided into three types
  1. algorithms preprocessing a fixed text and then reading a variable pattern [19, 14] (organizing text by various methods as e.g. a suffix array [14] or a suffix tree [21]),
  2. algorithms preprocessing a fixed pattern and then scanning the text on-line [12, 10, 8],
  3. algorithms scanning both text and pattern on-line [17, 9, 16, 13].

For the subsequence matching problem, we study here *on-line* algorithms which scan text t forward, reading each text symbol only once. The algorithms of Sections 3.1 and 3.3 fall in the third category, while the algorithm of Section 3.2 falls in the second category. A standard on-line algorithm for the subsequence matching problem is described in [9, 16]. It has some similarities with the algorithms used for pattern-matching [1, 2] and runs in time $O(nk)$; it will be described in more detail in Section 3.1. Another on-line algorithm is described in [8]: its basic idea consists in cutting the pattern into $k/\log k$ suitably chosen pieces organized in a trie; it then runs in time $O(nk/\log k)$.

We briefly compare the subsequence matching problem with closely related problems studied in the literature:

1. the *matching with don't cares* problem: given k pattern strings $P_1,\ldots,P_k \in A^*$, search text t for occurrences of the form $P_1u_1\cdots u_{k-1}P_k$ for $u_1,\ldots,u_{k-1}\in A^*$.
2. the *subsequence matching* problem: given a pattern $p = p_1\cdots p_k \in A^*$, with $p_i \in A$, search text t for occurrences of the form $p_1u_1\cdots u_{k-1}p_k$ for $u_1,\ldots,u_{k-1}\in A^*$, with the constraint that the total length of $p_1u_1\ldots u_{k-1}p_k$ is not greater than a given integer w (i.e. $\leqslant w$).

3. the *pattern-matching* problem: given a pattern $p \in A^*$, find occurrences of p in text t.

The matching with don't cares has been studied, without bounds on the lengths of the u_is, in [13] and, with constraints on the lengths of the u_is, in [14]. From a purely algebraic viewpoint, pattern-matching is a particular instance of subsequence matching, which is in turn a particular instance of matching with don't cares. However, from the complexity viewpoint, these problems are different and not interreducible.

Noticing that the subsequence matching problem is a generalization of the pattern-matching problem, we introduce here two algorithms based on new ideas. We observe first that, when the window size w is equal to the pattern length k, the subsequence matching problem reduces to the pattern-matching problem. We note then that a very efficient pattern-matching algorithm, the Knuth–Morris–Pratt (KMP) algorithm, is based on preprocessing the pattern [12]. We thus use a similar approach for subsequence matching: given a window size w, we preprocess the pattern, in order to obtain a minimal finite-state automaton which then runs in time n on any length n text and computes the number of w-windows containing pattern p as a subsequence. Indeed, when $w = k$, after the preprocessing, our algorithm runs *exactly* like the KMP algorithm. Our automaton is based on an idea different from the ones used in suffix automata [8], suffix trees [21] and similar structures [13], or suffix arrays [14]: we use prefixes of the pattern and substrings of the text instead of the usually used suffixes.

The paper is organized as follows: in Section 2, we define the problem, in Section 3 we describe the algorithms and study their complexities; experimental results are stated in Section 4.

2. The problem

2.1. The subsequence matching problem

An *alphabet* is a finite nonempty set A. A *string* of *length* n over the alphabet A is a mapping t from the set of integers $\{1,\ldots,n\}$ into A. The only string of length zero is the *empty string*, denoted by ε. A nonempty string $t : i \mapsto t_i$ will be denoted by $t_1 t_2 \cdots t_n$. A *language* over alphabet A is a set of strings on the alphabet A.

Let $t = t_1 t_2 \cdots t_n$ be a string. A string $p = p_1 p_2 \cdots p_k$ is said to be a *substring* (or *factor*) of t iff there exists an integer j such that $t_{j+i} = p_i$ for $1 \leqslant i \leqslant k$. A *window* of *size* w on string t, in short w-window, is a substring $t_{i+1} t_{i+2} \cdots t_{i+w}$ of t of length w; there are $n - w + 1$ such windows. String p is a *subsequence* of t iff there exist integers $1 \leqslant i_1 < i_2 < \cdots < i_k \leqslant n$ such that $t_{i_j} = p_j$ for $1 \leqslant j \leqslant k$. If p is a subsequence of t and if we have $i_k - i_1 < w$, then p is a *subsequence* of t *within a* w-*window*.

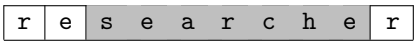

Fig. 1. A 7-window with a minimal substring containing 'see'.

Example 1. If $t =$ 'researcher' then 'sea' is a substring of t, hence 'sea' is also a subsequence of t. Further, 'see' is neither a substring, nor a subsequence of t within a 6-window, but 'see' is a subsequence of t within a 7-window. See Fig. 1.

Given an alphabet A, and strings p, t over A:

- the *pattern-matching problem* consists in finding whether p is a substring of t,
- the *plain subsequence matching problem* is to find whether p is a subsequence of t,
- given moreover a window size w,
  - the *Window-Existence Subsequence matching Problem*, in short *WESP*, consists in finding whether p is a subsequence of t within a w-window.
  - the *Window-Accumulated Subsequence matching Problem*, in short *WASP*, consists in counting *all* the w-windows within which p is a subsequence of t.

A naive solution exists for the pattern-matching problem whose time complexity on RAM is $O(nk)$. Knuth et al. [12] have given a well-known algorithm to solve the problem in linear time $O(n + k)$. A naive solution for the *WASP* is in $O(nkw)$. A more elaborate algorithm, we call it the **standard algorithm**, is in $O(nk)$ (see [16]). In [15] Mannila asks whether the *WASP* can be solved in $o(nk)$.

2.2. The $o(nk)$ notation

An important issue is to first explicit the meaning of the $o(nk)$ notation defined by Landau (cf. [11]). Originally, the $o(h(n))$ notation was introduced to compare growths of functions of a single argument; when we are to compare functions in several arguments, different non-equivalent interpretations of $o(h(n, m, \ldots))$ are possible. In our case, assume an algorithm in time $t(n, k)$; then $t(n, k) = o(nk)$ can be interpreted in two different ways:

1. either as $\lim_{n+k\to+\infty} t(n,k)/nk = 0$, i.e. $\forall\varepsilon, \exists N, \forall n, \forall k(n + k > N \Rightarrow t(n,k) < \varepsilon)$;
2. or as $\lim_{\substack{n\to+\infty \\ k\to+\infty}} t(n,k)/nk = 0$, i.e. $\forall\varepsilon, \exists N, \forall n, \forall k(n > N \text{ and } k > N \Rightarrow t(n,k) < \varepsilon)$.

With interpretation 1, no algorithm can solve the *WASP* in time $o(nk)$. Indeed, any algorithm for the *WASP* must read the text once, hence $t(n, k) \geqslant n$. Then, for a given k, e.g. $k = 2$, $t(n,k)/nk \geqslant 1/2$, hence $\lim_{n+k\to+\infty} t(n,k)/nk = 0$ is impossible. We thus choose interpretation 2.

Das et al. [9] give an $O(nk/\log k)$ algorithm for the *WASP* by using tries to represent the pattern, hence solves Mannila's problem. We improve this result in another direction by giving a linear ($O(n)$) on-line algorithm on MP-RAM for the *WASP* (Section 3.3.2).

3. Algorithms and upper bounds on their complexities

We describe first the standard algorithm, then two new algorithms counting the number of w-windows of t which contain p as a subsequence, and we study their complexity.

3.1. The standard algorithm

Introduction: Our intention is to give algorithms improving the algorithm of [16], called here the *standard algorithm*. Let us first recall this algorithm.

Main idea: Let $p = p_1 p_2 \cdots p_k$ be the pattern and let $t = t_1 t_2 \cdots t_n$ be the text. We note first that the subsequence matching problem reduces to finding the number of w-windows of t which contain a *minimal substring* containing p: a substring of t containing p is said to be *minimal* if no proper substring of it contains p.

Example 2. For instance, let $t = \texttt{researshers}$ (sic) and $p = \texttt{se}$; then, `rese` and `rshe` are substrings containing `se`, but are *not* minimal substrings containing `se`. On the other hand, `se` and `she` are minimal substrings containing `se`.

Definition 1. A string $t_r \dots t_s$ is said to be a *minimal substring* of t *containing* $p_1 \dots p_l$ iff

1. there exist integers $r \leqslant i_1 < i_2 < \dots < i_l \leqslant s$ such that $t_{i_j} = p_j$ for $1 \leqslant j \leqslant l$ (we say that $i_1, i_2, \dots, i_l$ is an *occurrence* of $p_1 \dots p_l$ in $t_r \dots t_s$),
2. and moreover, no proper substring of $t_r \dots t_s$ contains $p_1 \dots p_l$, i.e. for all occurrences of $p_1 \dots p_l$ in $t_r \dots t_s$, $i_1 = r$ and $i_l = s$.

Indeed any window containing p as a subsequence also contains a minimal substring containing p. See Fig. 1.

Proposition 1. *There exists an* $O(nk)$ *algorithm to solve the WASP with arguments* n, k, w *and dealing with integers less than* n.

Proof. We first explain the idea of the algorithm, inspired from those of [16, 8]. In order to count the number of w-windows of t which contain a minimal substring containing p we maintain an array of integers $s[1 \dots k]$, where $s[l]$ contains the starting position in t of the most recently seen minimal substring containing $p_1 \dots p_l$ if such a substring exists, and 0 otherwise. We initialize $s[1 \dots k]$ with $[0 \dots 0]$. The algorithm runs by sliding a w-window on t. Let i be the index in t where the current window ends. If $s[l] = j \neq 0$, this means there is a minimal substring containing $p_1 \dots p_l$ and starting at index j in the substring $t_j \dots t_i$. Thus, if $s[k] = j$ and $i - j < w$ we conclude that the current window contains a minimal substring containing p. The variable *count* gives the number of w-windows containing p as a subsequence. □

Example 3. Let for instance $t = \texttt{researshers}$, $p = \texttt{see}$, and $w = 8$; we obtain the following table:

i	t	$s[1]$	$s[2]$	$s[3]$	$i - s[3]$	Accept?
1	r	$-\infty$	$-\infty$	$-\infty$	∞	No
2	e	$-\infty$	$-\infty$	$-\infty$	∞	No
3	**s**	3	$-\infty$	$-\infty$	∞	No
4	**e**	3	3	$-\infty$	∞	No
5	a	3	3	$-\infty$	∞	No
6	r	3	3	$-\infty$	∞	No
7	s	7	3	$-\infty$	∞	No
8	h	7	3	$-\infty$	∞	No
9	**e**	7	7	3	6	Yes
10	r	7	7	3	7	Yes
11	s	11	7	3	8	No

We accept when $i - s[3] < 8$.

Algorithm. The algorithm is given below, in a pseudo-language ('`DO forall` $l \in I$ `inst ENDDO`' performs `inst` simultaneously for all indices $l \in I$).

```
DO forall l ∈ [1...k] s[l] := 0 ENDDO
count := 0
DO for i ∈ [1...n] in increasing order
    IF t_i = p_1 then s[1] := i ENDIF
    DO forall l ∈ [2...k]
        IF t_i = p_l then s[l] := s[l − 1] ENDIF ENDDO
    IF i − s[k] < w then count := count + 1 ENDIF
ENDDO
```

We obtain easily the computational complexity which is announced.

Remark 1. Note that, as a practical improvement, to find more quickly the entries of s that need to be updated, implementations may maintain, for every letter a, a list $waits[a]$ of elements of A consisting of those entries of s that need to be updated if the next letter scanned is a.

3.2. An algorithm preprocessing the pattern

Remark 2. In the standard algorithm, we have to maintain an array of integers:

$s[1] \dots s[k]$

where $s[i]$ is a position in the string t, hence is an integer of length $ln(|n|)$.
A first variation is to replace this array by the array:

$l[1] \dots l[k]$

where $l[i]$ is the distance between the current position and the starting position of the most recently seen minimal substring containing $p_1 \dots p_i$ if such a substring exists, and $+\infty$ otherwise. Of course this distance is interesting when it is less than the size

of the window, hence we may truncate it by w. The consequence is we have to deal with integers of length $ln(|w|)$ instead of integers of length $ln(|n|)$.

Example 4. Let, as in example 3, $t = \texttt{researshers}$, $p = \texttt{see}$ and $w = 8$; we obtain the following table:

i	t	$l[3]$	$l[2]$	$l[1]$	Accept?
1	r	$+\infty$	$+\infty$	$+\infty$	No
2	e	$+\infty$	$+\infty$	$+\infty$	No
3	**s**	$+\infty$	$+\infty$	1	No
4	**e**	$+\infty$	2	2	No
5	a	$+\infty$	3	3	No
6	r	$+\infty$	4	4	No
7	s	$+\infty$	5	1	No
8	h	$+\infty$	6	2	No
9	**e**	7	3	3	Yes
10	r	8	4	4	Yes
11	s	$+\infty$	5	1	No

We accept when $l[3] \leqslant 8$. For this reason, we changed the order of presentation of $l[1]$, $l[2]$, $l[3]$.

Remark 3. A value of the last array may be seen as one of w^k states. As in the Knuth–Morris–Pratt algorithm, we may consider an automaton instead of computing a new state on the fly. We note that when the window size w is equal to the pattern length k, the *WASP* reduces to the usual pattern matching problem addressed by the Knuth–Morris–Pratt algorithm. Hence we generalize the KMP algorithm by preprocessing the pair (pattern + window size w). However, our automaton uses prefixes of the pattern instead of the more commonly used suffixes [8, 13].

3.2.1. A complexity result

Theorem 1. *There exists an* $O(f(w,k) + n)$ *algorithm to solve the WASP with arguments* n, k, w *on a classical RAM, dealing with integers less than* w.

Proof. The main point here is that the function $f(w,k)$ does not depend on n; we shall bound its growth rate later. The algorithm consists of two steps: the first step preprocesses the pattern and the second step scans the text.

Step 1: We construct a finite state automaton $\mathscr{A}$ by preprocessing pattern p. The alphabet of $\mathscr{A}$ is A; the states of $\mathscr{A}$ are k-tuples of numbers $\langle l_1, \ldots, l_k \rangle$ with $l_j \in \{1, \ldots, w, w+1\}$. Indeed we saw that the numbers in tuples may be truncated to w; here $w+1$ plays the rôle of $+\infty$.

We first informally describe the behaviour of $\mathscr{A}$. When automaton $\mathscr{A}$ is scanning a string t, it will be in state $\langle l_1, \ldots, l_k \rangle$ after reading $t_1 \ldots t_m$ iff, l_i is the length of the shortest suffix[4] of $t_1 \ldots t_m$ which is of length not greater than w and contains

[4] Recall that string s is a *prefix* (resp. *suffix*) of string t iff there exists a string v such that $t = sv$ (resp. $t = vs$).

$p_1 \dots p_i$ as a subsequence, for $i = 1, \dots, k$; if no suffix (of length not greater than w) of $t_1 \dots t_m$ contains $p_1 \dots p_i$ as a subsequence, we let $l_i = w + 1$. Namely, for every i such that $1 \leqslant i \leqslant k$ and $l_i < w + 1$, if we assume $t = t' t_{\max\{m-w+1,1\}} \dots t_m t''$, then string $s_i = p_1 \dots p_i$ is a subsequence of $t_{m-l_i+1} \dots t_m$ and is not a subsequence of $t_{m-l_i+2} \dots t_m$.

We now formally define automaton $\mathscr{A}$. Let $Next(l)$ be the auxiliary function

$$Next(l) = \begin{cases} l+1 & \text{if } l < w+1, \\ w+1 & \text{otherwise.} \end{cases}$$

1. The initial state of $\mathscr{A}$ is the k-tuple $\langle w+1, \dots, w+1 \rangle$.
2. The accepting states of $\mathscr{A}$ are the k-tuples $\langle l_1, \dots, l_k \rangle$ such that $l_k < w+1$, meaning that pattern p is a subsequence of the w-window ending at the currently scanned letter of t.
3. Transitions: starting from $\langle l_1, \dots, l_k \rangle$ and reading a, automaton $\mathscr{A}$ will go in $\langle l'_1, \dots, l'_k \rangle$, denoted by $\langle l_1, \dots, l_k \rangle \xrightarrow{a} \langle l'_1, \dots, l'_k \rangle$, where, for $i = 1, \dots, k$

$$l'_i = \begin{cases} Next(l_{i-1}) & \text{if } p_i = a, \\ Next(l_i) & \text{if } p_i \neq a. \end{cases}$$

(We take $l_0 = 0$ for $k = 1$.)

Step 2: Automaton $\mathscr{A}$ scans text $t_1 \dots t_n$, starting with $count = 0$ initially, and incrementing *count* by 1 each time an accepting state is encountered.

The second step takes time n on a classical RAM, and the first step takes time $f(w,k)$ related to the number of states of $\mathscr{A}$, which is $(w+1)^k$. □

3.2.2. Minimization of automaton $\mathscr{A}$

The algorithm of Theorem 1 can be optimized. To this end,

1. consider only k-tuples representing reachable states and discard the other k-tuples, and
2. moreover, substitute $w+1$ for l_i if the length $k-i$ of $p_{i+1} \dots p_k$ is $> w - l_i$, because then s_i is too short and p cannot be a subsequence in the next $w - l_i$ windows.

The optimized automaton $\mathscr{A}_{\text{opt}}$ is in state $\langle l_1, \dots, l_k \rangle$ after reading $t_1 \dots t_m$ iff l_i is the length of the shortest suffix of $t_{\max\{m-w+k-i+1,1\}} \cdots t_m$ containing $p_1 \dots p_i$ as a subsequence, for $i = 1, \dots, k$. The transitions of $\mathscr{A}_{\text{opt}}$ are defined by $\langle l_1, \dots, l_k \rangle \xrightarrow{a} \langle l'_1, \dots, l'_k \rangle$ where

$$l'_i = \begin{cases} l_{i-1} + 1 & \text{if } p_i = a, \\ l_i + 1 & \text{if } p_i \neq a \quad \text{and} \quad l_i + 1 \leqslant w - k + i, \\ w+1 & \text{otherwise.} \end{cases}$$

It will be shown below that, in the worst case, the number N of states of $\mathscr{A}_{\text{opt}}$ satisfies $\binom{w+1}{k} \leqslant N \leqslant \binom{w+k}{k}$. Hence, even for the optimized algorithm, $f(w,k)$ is exponential in k.

When $w = k$, our optimized construction gives an automaton having the *same* number of states as the KMP algorithm for pattern-matching. Moreover:

Theorem 2. *Assume that alphabet A contains at least one letter x which does not occur in pattern p; then* $\mathscr{A}_{\mathrm{opt}}$ *is the minimal finite state automaton* (*i.e. having the minimal number of states*) *accepting the strings t where pattern p occurs as a subsequence within a w-window.*

Proof. We show that any automaton $\mathscr{B}$ accepting the same strings as $\mathscr{A}_{\mathrm{opt}}$ has at least as many states as $\mathscr{A}_{\mathrm{opt}}$. Let $s = \langle l_1, \ldots, l_{i-1}, l_i, \ldots, l_k \rangle$ and $s' = \langle l_1, \ldots, l_{i-1}, l'_i, \ldots, l'_k \rangle$ be two states of $\mathscr{A}_{\mathrm{opt}}$, both reachable from $\langle w+1, \ldots, w+1 \rangle$, who first differ at their ith components. Let $t_1 \ldots t_m$ and $t'_1 \ldots t'_{m'}$ be corresponding input strings bringing $\mathscr{A}_{\mathrm{opt}}$ from $\langle w+1, \ldots, w+1 \rangle$ to s and s', respectively. Let us show that, after scanning inputs $t_1 \ldots t_m$ and $t'_1 \ldots t'_{m'}$, automaton $\mathscr{B}$ should come to different states $s_{\mathscr{B}}$ and $s'_{\mathscr{B}}$, respectively. Without loss of generality, we further assume that $l_i < l'_i$; let x be a letter not occurring in pattern p, then $s_{\mathscr{B}}$ accepts the text when the next scanned $w - l_i$ letters consist of $x^{w-l_i+i-k} p_{i+1} \ldots p_k$ while $s'_{\mathscr{B}}$ rejects the text in the same circumstances. □

3.2.3. About the size of automaton $\mathscr{A}_{\mathrm{opt}}$

As in Theorem 1, our algorithm consists in running $\mathscr{A}_{\mathrm{opt}}$ on a text t: counting the number of times we pass through an accepting state gives the number of w-windows of t containing p as a subsequence in time exactly n. Hence, after the preprocessing, we scan the text t in time merely n. We now study how long takes our preprocessing: this is related to N, the number of states of $\mathscr{A}_{\mathrm{opt}}$.

Lemma 1. 1. *In the worst case, the size N of* $\mathscr{A}_{\mathrm{opt}}$ *satisfies* $\binom{w+1}{k} \leqslant N$.

2. *In all cases, the size M of the* (*nonoptimized*) *automaton* $\mathscr{A}$ *satisfies* $M \leqslant \binom{w+k}{k}$.

Proof. In all cases, $M \leqslant \binom{w+k}{k}$ because the states of $\mathscr{A}$ assume the form $\langle l_1, l_2, \ldots, l_k \rangle$ with $l_1 \leqslant l_2 \leqslant \cdots \leqslant l_k \leqslant w+1$. In the case when $p = a^k$, we have $N = \binom{w+1}{k}$ because, when $p = a^k$, the states of $\mathscr{A}_{\mathrm{opt}}$ assume the form $\langle l_1, l_2, \ldots, l_i, w+1, \ldots, w+1 \rangle$ with $l_1 < l_2 < \cdots < l_i \leqslant w-k+i$ and because there are $\binom{w+1}{k}$ such sequences. Hence, in the worst case, $\binom{w+1}{k} \leqslant N$. □

Corollary 1. *The number of states of* $\mathscr{A}_{\mathrm{opt}}$ *is exponential in the worst case.*

Remark 4. Using hashing methods divides the preprocessing time by a factor of 10, and thus our algorithm with hashing can still run about twice as fast as the standard algorithm for windows of size 14 or 15.

Corollary 2 (Space complexity). *The* $\mathscr{A}_{\mathrm{opt}}$*-based algorithm uses at most* $\mathrm{O}(n + \binom{w+k}{k})$ *locations of size* $\mathrm{O}(k \log w)$.

Proof. Indeed, up to $O(\binom{w+k}{k})$ additional memory locations may be needed to store the states of $\mathscr{A}_{\mathrm{opt}}$, and a state is a k-tuple of numbers $\leqslant w+1$, hence each state needs $k \log w$ bits. □

3.3. An algorithm on MP-RAMs

For 'large' windows, for instance on a PC for windows of size $w \geqslant 14$ and patterns of size $k \geqslant 6$, the previous preprocessing explodes, due to the exponential growth in the number of states of $\mathscr{A}_{\mathrm{opt}}$, and the standard algorithm is better than our algorithm. Whence the idea of a smaller preprocessing which is almost independent of the pattern and of the window; this method is described in the present section.

3.3.1. MP-RAMs

It is usual to give pattern matching algorithms on RAMs. Indeed the RAM model of computation is well-suited for computational complexities greater than n^2. For low complexities though, RAMs are not a well-suited model of computation, because *any* random access to the memory is counted as *one* elementary operation: this is no longer a valid model when there are too many different values to be stored, as for instance the $\binom{w+1}{k}$ states of $\mathscr{A}$.

Already in 1974, the motivation of Pratt et al. [18] for introducing vector machines was the remark that bitwise boolean operations and shift are implemented on commercial computers and are ideally suited for certain problems. This paper started a quite interesting series of papers comparing the computational complexities of various models of machines accounting for bitwise boolean operations and shifts with those of conventional machines, such as Turing machines, RAMs, etc. [20, 6]. Going back to the first motivation of Pratt et al. [18], concrete applications of this technique to varieties of string-matching problems began with [4, 22]: they are known as *bit-parallelism* or *shift-OR*. We follow this path with our problem, which is close to the problems treated in [4, 22, 5], although it is different from these problems.

In what follows, we use a refinement of the RAM model, which is a more realistic model of computation. Moreover, we encode $\mathscr{A}$ in such a way that (i) each state of $\mathscr{A}$ can be stored in a single memory location, and (ii) only the most basic micro-processor operations are needed to compute the transitions of $\mathscr{A}$. We use a RAM with the same control structures as those of conventional RAMs,[5] but with a set of initial operations enlarged by including bitwise boolean operations and shifts; whenever possible, these operations will be preferred. Such RAMs are closer to microprocessors, hence we call them MP-RAMs.

Definition 2. An MP-RAM is a RAM extended by the following new operations:
1. bitwise AND, denoted by &,

[5] For a formal definition of classical RAMs see e.g. [2, pp. 5–11.]

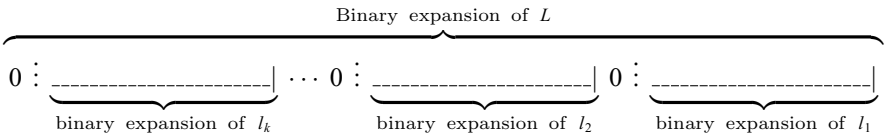

Fig. 2. Encoding of $\langle l_1, \ldots, l_k \rangle$.

2. left shift, denoted by $\ll$, or by *shl*, and
3. right shift, denoted by $\gg$, or by *shr*.

The new operations are low-level operations which will be executed much faster than the complex $MULT$, DIV operations.

Example 5. Assuming MP-RAMs with unbounded memory locations, we have for instance:

$$(10110 \ \& \ 01101) = 100;\ (10110 \ll 4) = 101100000;\ \ (10110 \gg 3) = 10.$$

If memory locations were bounded to 8 bits, we would have: $(10110 \ll 4) = 1100000$, which we may write in the form $(00010110 \ll 4) = 01100000$.

3.3.2. A complexity result

The idea of the algorithm consists in encoding automaton $\mathscr{A}$ of Theorem 1 so that its transitions can be computed by an MP-RAM without using the $MULT$, DIV operations. We describe the encoding of $\mathscr{A}$. Let Ω be the least natural number such that $w + 2 \leqslant 2^{\Omega}$. The rôle of $+\infty$ is now played by the number $2^{\Omega} - 1$ whose binary representation consists of Ω units. We redefine the function $Next$ as:

$$Next_{\Omega}(l) = \begin{cases} l + 1 & \text{if } l < 2^{\Omega} - 1, \\ 2^{\Omega} - 1 & \text{otherwise.} \end{cases}$$

State $\langle l_1, \ldots, l_k \rangle$ is then coded by the number:

$$\begin{aligned} L &= \sum_{i=1}^{k} l_i (2^{\Omega+1})^{i-1} \\ &= \sum_{i=1}^{k} l_i \ll ((\Omega + 1)(i - 1)). \end{aligned} \tag{1}$$

The binary expansion of L consists of the binary expansions of the l_is padded by leading zeros up to the length $\Omega + 1$; see Fig. 2. Note that all these padded representations begin with 0 because all l_is are less than $2^{\Omega} - 1$. These 0s play an important rôle in the implementation of the function $Next_{\Omega}$.

According to the definition of Eq. (1), the initial state $\langle 2^{\Omega}-1,\ldots,2^{\Omega}-1\rangle$ is coded as

$$I_0=\sum_{i=1}^{k}(2^{\Omega}-1)2^{(\Omega+1)(i-1)}$$
$$=\sum_{i=1}^{k}((1\ll\Omega)-1)\ll((\Omega+1)(i-1)).$$

Respectively, accepting states are exactly those L satisfying $L<F$ where $F=(w+1)2^{(\Omega+1)(k-1)}$, which means that $l_k\leqslant w$.

Proposition 2. *The codes of the transitions of $\mathscr{A}$ are computed by an MP-RAM as follows*:

$$L\xrightarrow{a}L' \text{ iff } L'=T-((T\&E_2)\gg\Omega)$$

where

$$T=((L\ll(\Omega+1))\&M_a)+(L\&N_a)+E_1\ ,$$

$$E_1=\sum_{i=1}^{k}1\ll((\Omega+1)(i-1)),$$

$$E_2=\sum_{i=1}^{k}(1\ll\Omega)\ll((\Omega+1)(i-1)),$$

and for $a\in A$,

$$M_a=\sum_{\substack{p_i=a\\1\leqslant i\leqslant k}}((1\ll\Omega)-1)\ll((\Omega+1)(i-1)),$$
$$N_a=\sum_{\substack{p_i\neq a\\1\leqslant i\leqslant k}}((1\ll\Omega)-1)\ll((\Omega+1)(i-1)).$$

Proof. A state $l=\langle l_1,\ldots,l_k\rangle$ is encoded by the binary expansion L defined in equation 1. The binary expansion L is obtained by concatenating the binary expansions $\overline{l_i}$s of the l_is padded by leading zeros up to the length $\Omega+1$; see Fig. 2. We chose the basis $2^{\Omega+1}$ rather than the basis 2^{Ω} to encode the l_is: indeed, basis 2^{Ω} is sufficient to encode all the l_is, but the larger basis $2^{\Omega+1}$ will simplify the treatment in case of overflows.

Consequently, the binary expansion of an integer less than $2^{k(\Omega+1)}$ consists of k *large blocks* of $(\Omega+1)$ bits, the first bit is called the *overflow digit* and the remaining Ω bits constitute a *small block*. The blocks are numbered 1 to k leftward (the rightmost

$$L = \boxed{0 \vdots \overline{l_5}} \boxed{0 \vdots \overline{l_4}} \boxed{0 \vdots \overline{l_3}} \boxed{0 \vdots \overline{l_2}} \boxed{0 \vdots \overline{l_1}}$$

Fig. 3. Encoding of $\langle l_1, \ldots, l_k \rangle$; $\overline{l_i}$ is the binary expansion of l_i.

block is block 1, and the leftmost block is block k). When no ambiguity arises we will just say 'block' instead of 'small block'.

Example 6. Assume pattern p has length $k = 5$ and $w = 14$; hence $\Omega = 4$. The encoding L of state $l = \langle l_1, \ldots, l_5 \rangle$ is depicted in Fig. 3; L consists of 5 'large blocks' of the form $0 \vdots \overline{l_i}$, where 0 is the overflow digit and each $\overline{l_i}$ is a 'small block'.

For instance if $l = \langle 3, 5, 10, \infty, \infty \rangle$,

$$\boxed{0 \vdots 1111} \boxed{0 \vdots 1111} \boxed{0 \vdots 1010} \boxed{0 \vdots 0101} \boxed{0 \vdots 0011}$$

and, with the notation of Fig. 3, depicted as

$$L = \boxed{0 \vdots \overline{15}} \boxed{0 \vdots \overline{15}} \boxed{0 \vdots \overline{10}} \boxed{0 \vdots \overline{5}} \boxed{0 \vdots \overline{3}}$$

The initial state is coded by

$$I_0 = \sum_{i=1}^{k} (2^{\Omega} - 1) 2^{(\Omega+1)(i-1)}$$
$$= \sum_{i=1}^{k} ((1 \ll \Omega) - 1) \ll ((\Omega + 1)(i - 1)).$$

Example 6 (*continued*). $\Omega = 4$ and I_0 is depicted by

$$I_0 = \boxed{0 \vdots 1111} \boxed{0 \vdots 1111} \boxed{0 \vdots 1111} \boxed{0 \vdots 1111} \boxed{0 \vdots 1111}$$

if the sequence consisting of Ω 1s and representing $\overline{w+1}$ is shortened into $\underline{1}$, the picture becomes

$$I_0 = \boxed{0 \vdots \underline{1}} \boxed{0 \vdots \underline{1}} \boxed{0 \vdots \underline{1}} \boxed{0 \vdots \underline{1}} \boxed{0 \vdots \underline{1}}$$

Accepting states are exactly those Ls satisfying $L < F$ where $F = (w + 1)2^{(\Omega+1)(k-1)}$; indeed, $l = \langle l_1, \ldots, l_k \rangle$ is an accepting state if $l_k < w + 1$.

Example 6 (*continued*). Here, $w = 14$ and F is depicted by

$$F = \boxed{0 \vdots 1111} \boxed{0 \vdots 0000} \boxed{0 \vdots 0000} \boxed{0 \vdots 0000} \boxed{0 \vdots 0000}.$$

We now will describe the proof, illustrating it by Example 6 where we assume $p = aabca$.

Recall that, if $l = \langle l_1, \ldots, l_k \rangle \xrightarrow{\sigma} l' = \langle l'_1, \ldots, l'_k \rangle$, then l'_i is either $Next_\Omega(l_{i-1})$ or $Next_\Omega(l_i)$ according to whether the scanned letter σ is equal to p_i or not. The cases $l'_i = Next_\Omega(l_{i-1})$ and $l'_i = Next_\Omega(l_i)$ will be respectively called *computations of the first kind* and of *the second kind.*

Step 1: M_σ is a filter designed to prepare the computations of the first kind. Precisely, let

$$M_\sigma = \sum_{\substack{p_i=\sigma \\ 1 \leqslant i \leqslant k}} ((1 \ll \Omega) - 1) \ll ((\Omega + 1)(i - 1)).$$

For i ranging from 1 to k, the ith small block of M_σ from the right-hand side consists of Ω ones or Ω zeros according to whether p_i is equal to σ or not.

Example 6 (*continued*). Here $p = aabca$, hence

$$M_a = \boxed{0 \vdots \underline{1} \mid 0 \vdots \underline{0} \mid 0 \vdots \underline{0} \mid 0 \vdots \underline{1} \mid 0 \vdots \underline{1}}$$

$$M_b = \boxed{0 \vdots \underline{0} \mid 0 \vdots \underline{0} \mid 0 \vdots \underline{1} \mid 0 \vdots \underline{0} \mid 0 \vdots \underline{0}}$$

$$M_c = \boxed{0 \vdots \underline{0} \mid 0 \vdots \underline{1} \mid 0 \vdots \underline{0} \mid 0 \vdots \underline{0} \mid 0 \vdots \underline{0}}$$

where $\underline{0} = 0000$ and $\underline{1} = 1111$.

A left shift of L by $\Omega + 1$ will shift the $\overline{l_i}$s by one large block leftward.

Example 7. If $l = \langle l_1, l_2, l_3, l_4, l_5 \rangle$, then

$$L = \boxed{0 \vdots \overline{l_5} \mid 0 \vdots \overline{l_4} \mid 0 \vdots \overline{l_3} \mid 0 \vdots \overline{l_2} \mid 0 \vdots \overline{l_1}}$$

and

$$L \ll (\Omega + 1) = \boxed{0 \vdots \overline{l_5} \mid 0 \vdots \overline{l_4} \mid 0 \vdots \overline{l_3} \mid 0 \vdots \overline{l_2} \mid 0 \vdots \overline{l_1} \mid 0 \vdots \underline{0}}$$

Note that the rightmost small block of $L \ll (\Omega + 1)$ is always $\underline{0}$.

"Adding" $L \ll (\Omega + 1)$ with M_σ results

1. in erasing the leftmost large block of $L \ll (\Omega + 1)$ and,
2. for i ranging from 2 to k, in setting the ith small block from the right to, respectively, $\overline{l_{i-1}}$ or $\underline{0}$ according to whether σ is equal to p_i or not.

Thus $(L \ll (\Omega + 1)) \& M_\sigma$ screens off the blocks for which $\sigma \neq p_i$ and shifts everything by one large block leftward; more precisely, for $i > 1$, the ith block will contain $\overline{l_{i-1}}$ if $p_i = \sigma$ and $\underline{0}$ otherwise.

Example 6 (*continued*). With $p = aabca$, we have

$$(L \ll (\Omega+1))\&M_a = \boxed{0 \vdots \overline{l_4}}\boxed{0 \vdots \underline{0}}\boxed{0 \vdots \underline{0}}\boxed{0 \vdots \overline{l_1}}\boxed{0 \vdots \underline{0}}$$

$$(L \ll (\Omega+1))\&M_b = \boxed{0 \vdots \underline{0}}\boxed{0 \vdots \underline{0}}\boxed{0 \vdots \overline{l_2}}\boxed{0 \vdots \underline{0}}\boxed{0 \vdots \underline{0}}$$

$$(L \ll (\Omega+1))\&M_c = \boxed{0 \vdots \underline{0}}\boxed{0 \vdots \overline{l_3}}\boxed{0 \vdots \underline{0}}\boxed{0 \vdots \underline{0}}\boxed{0 \vdots \underline{0}}$$

Similarly, N_σ is a filter designed to prepare the computations of the second kind. Let

$$N_\sigma = \sum_{\substack{p_i \neq \sigma \\ 1 \leqslant i \leqslant k}} ((1 \ll \Omega) - 1) \ll ((\Omega+1)(i-1)).$$

For i ranging from 1 to k, the ith block of N_σ is $\underline{0}$ or $\underline{1}$ if $p_i = \sigma$ or $p_i \neq \sigma$, respectively. Namely, $L\&N_\sigma$ screens off the blocks for which $\sigma = p_i$. More precisely, for $i \geqslant 1$, the ith block of $L\&N_\sigma$ will contain $\overline{l_i}$ if $p_i \neq \sigma$ and 0 otherwise.

Example 6 (*continued*). With $p = aabca$,

$$N_a = \boxed{0 \vdots \underline{0}}\boxed{0 \vdots \underline{1}}\boxed{0 \vdots \underline{1}}\boxed{0 \vdots \underline{0}}\boxed{0 \vdots \underline{0}}$$

$$L\&N_a = \boxed{0 \vdots \underline{0}}\boxed{0 \vdots \overline{l_4}}\boxed{0 \vdots \overline{l_3}}\boxed{0 \vdots \underline{0}}\boxed{0 \vdots \underline{0}}$$

$$N_b = \boxed{0 \vdots \underline{1}}\boxed{0 \vdots \underline{1}}\boxed{0 \vdots \underline{0}}\boxed{0 \vdots \underline{1}}\boxed{0 \vdots \underline{1}}$$

$$L\&N_b = \boxed{0 \vdots \overline{l_5}}\boxed{0 \vdots \overline{l_4}}\boxed{0 \vdots \underline{0}}\boxed{0 \vdots \overline{l_2}}\boxed{0 \vdots \overline{l_1}}$$

We note that M_σ and N_σ are complementary in the following sense: the small blocks of M_σ and N_σ can assume only the values $\underline{1}$ and $\underline{0}$, and the ith small block of M_σ is $\underline{1}$ iff the ith small block of N_σ is $\underline{0}$.

The ith block of $((L \ll (\Omega+1))\&M_\sigma) + (L\&N_\sigma)$ is equal to $\overline{l_{i-1}}$ or $\overline{l_i}$, respectively when $\sigma = p_i$ or $\sigma \neq p_i$. The rightmost block is special: its value is $\underline{0}$ when $\sigma = p_1$ and $\overline{l_1}$ when $\sigma \neq p_1$.

Example 6 (*continued*). With $p = aabca$, $((L \ll (\Omega+1))\&M_a) + (L\&N_a)$ is depicted by:

$$\boxed{0 \vdots \overline{l_4}}\boxed{0 \vdots \overline{l_4}}\boxed{0 \vdots \overline{l_3}}\boxed{0 \vdots \overline{l_1}}\boxed{0 \vdots \underline{0}}$$

Let

$$E_1 = \sum_{i=1}^{k} 1 \ll ((\Omega+1)(i-1)).$$

Each block of E_1 consists of the binary expansion of 1.

Example 8. E_1 is depicted by

$0 \vdots 0001$	$0 \vdots 0001$	$0 \vdots 0001$	$0 \vdots 0001$	$0 \vdots 0001$

Adding E_1 to $((L \ll (\Omega+1))\&M_\sigma) + (L\&N_\sigma)$ results in adding 1 to each block.
Let $T = ((L \ll (\Omega+1))\&M_\sigma) + (L\&N_\sigma) + E_1$.

- On the one hand, for i ranging from 2 to k, the ith large block of T contains
  - $\overline{l_i'}$ if the contents of $((L \ll (\Omega+1))\&M_\sigma) + (L\&N_\sigma)$ is strictly less than $\underline{1}$ $(= 2^\Omega - 1)$,
  - 2^Ω otherwise. In this latter case, we have obtained 2^Ω whilst $\overline{l_i'}$ is equal to $2^\Omega - 1$, because $(2^\Omega - 1) + 1 = 2^\Omega - 1$ (recall $2^\Omega - 1$ plays the rôle of ∞); thus the ith block of T does not contain the proper result.
- On the other hand, the rightmost block of T contains $\overline{l_1} + 1$ or 1. In the first case, the contents of that block is again the proper result $\overline{l_1' c}$ or not, respectively when $\underline{1} \neq \overline{l_1}$ or $\overline{l_1} = \underline{1}$.

At the end of Step 1, all the large blocks of T, except may be the first block, contain a number of the form $\lambda + 1$ when we would like them to contain $Next_\Omega(\lambda)$.

Example 6 (*continued*). With $p = aabca$, $w = 14$, and $l = \langle 3, 5, 10, w+1, w+1\rangle$, $((L \ll (\Omega+1))\&M_a) + (L\&N_a) + E_1$ is depicted by

$1 \vdots 0000$	$1 \vdots 0000$	$0 \vdots 1011$	$0 \vdots 0100$	$0 \vdots 0001$

Step 2: We replace $\lambda + 1$ by $Next_\Omega(\lambda)$ wherever needed: it suffices to reset the large blocks where an overflow has occurred, by substituting $2^\Omega - 1$ for 2^Ω in all such blocks. Let

$$E_2 = \sum_{i=1}^{k} (1 \ll \Omega) \ll ((\Omega+1)(i-1)).$$

In each large block of E_2 the overflow digit is 1 and the small block is $\underline{0}$.

Example 9. E_2 is depicted by

$1 \vdots \underline{0}$	$1 \vdots \underline{0}$	$1 \vdots \underline{0}$	$1 \vdots \underline{0}$	$1 \vdots \underline{0}$

Hence, for each large block of $T\&E_2$, the small block is $\underline{0}$ and the overflow digit is 1 if there is an overflow in the corresponding block of T, i.e. that block consists of 2^Ω, and 0 otherwise.

Example 6 (*continued*).

$(T\&E_2) =$

$1 \vdots \underline{0}$	$1 \vdots \underline{0}$	$0 \vdots \underline{0}$	$0 \vdots \underline{0}$	$0 \vdots \underline{0}$

Dividing $(T\&E_2)$ by 2^{Ω} is realized by a right shift of every bit of $(T\&E_2)$. The ith large block of the result of this division contains the binary expansion of 1 or 0, respectively when the overflow digit of the ith large block of $(T\&E_2)$ was 1 or 0.

Example 6 (*continued*). $((T\&E_2)\ll\Omega)$ is depicted by

$0\vdots 0001$	$0\vdots 0001$	$0\vdots 0000$	$0\vdots 0000$	$0\vdots 0000$

Hence, in $T-((T\&E_2)\gg\Omega)$ the ith large block coincides with the corresponding block of T if there was no overflow, and is the binary expansion of $2^{\Omega}-1$ (namely $0\vdots\overline{w+1}$) otherwise.

Example 6 (*continued*). $T-((T\&E_2)\gg\Omega)$ is depicted by

$0\vdots 1111$	$0\vdots 1111$	$0\vdots 1011$	$0\vdots 0100$	$0\vdots 0001$

In both cases, the ith large block is equal to $\overline{l_i'}$. Hence,

$$L' = T-((T\&E_2)\gg\Omega).$$

Lemma 2. *Preprocessing of w and the pattern* (*step* 1 *of the algorithm on MP-RAM*) *runs in time* $O(k+\log(w))$.

Proof. In the preprocessing step 1. we compute the $2\alpha+6$ numbers $\Omega, I_0, F, E_1, E_2, M_\sigma, N_\sigma$. We compute Ω by the following algorithm:

```
DO u := w + 1; Ω := 0 ENDDO
WHILE u > 0 DO
        u := u ≫ 1; Ω := Ω + 1 ENDWHILE
```

Hence the time needed is $O(\log(w))$.
We compute I_0 by the following algorithm:

```
DO z := (1 ≪ Ω) − 1; I_0 := 0 ENDDO
DO for i ∈ [1 ... k] in increasing order
        I_0 := (I_0 ≪ (Ω + 1)) + z ENDDO
```

Hence the time needed is $O(k)$, and similarly for E_1 and E_2.
We compute F by the following $O(k)$ algorithm:

```
F := w + 1
DO for i ∈ [1 ... k − 1] in increasing order
        F := (F ≪ (Ω + 1)) ENDDO
```

We compute M_σ by the following $O(k)$ algorithm, using Hörner's method:

```
DO z := (1 ≪ Ω) − 1; M_σ := 0 ENDDO
DO for i ∈ [1 ... k] in decreasing order
        M_σ := (M_σ ≪ (Ω + 1))
        IF p_i = σ then M_σ := M_σ + z ENDIF ENDDO
```

Similarly for N_σ.

Theorem 3. *There exists an* $O(n+k+\log(w))=O(n)$ *on-line algorithm to solve the WASP with arguments* n,k,w *on an MP-RAM.*

Proof. Let α denote the number of letters in alphabet A,[6] and let $|w|$ denote the length of the binary representation of w; the preprocessing consists simply in computing $2\alpha+5$ numbers of size $k(|w|+2)$ which will be used in computing *on-line and without preprocessing* the transitions of the automaton $\mathscr{A}$ of Theorem 1 by an MP-RAM. The algorithm consists of the four steps below:

1. compute Ω, I_0, F, E_1, E_2, and M_a, N_a for $a \in A$;
2. set $count = 0$;
3. set $L = I_0$;
4. scan the text t; after reading t_i, calculate the new state L, and if $L < F$ increment *count* by 1.

Our algorithm uses only the simple and fast operations &, $\ll$, $\gg$, and addition. We have shown in Lemma 2 that the preprocessing step 1 takes time $O(k+\log(w))$. In step 4 we scan text t in time $O(n)$. Hence the complexity is $O(n+k+\log(w))$. Because $k \leqslant n$ and $w \leqslant n$, the complexity of the algorithm is $O(n)$. □

4. Experimental results

We implemented all algorithms in C. The experiments have been run on a PC-Cyrix 166 under Linux, on a DEC-alpha shared by several users and on an Apple Mac PowerPC 7100. The text consisted of a randomly generated text file; the patterns were of the form $a^{k_1}ba^{k_2}$, which give a high complexity for the preprocessing. For other patterns, the results are much better (see Fig. 5, pattern 2). We counted the complexity in machine clock-ticks, because quite often our algorithms ran in 0 s. In Fig. 5 the patterns for sub-figure pattern 1 are prefixes of length 4, 6, 8, 10 of $p_1 = aabaaaaaaa$ (i.e. for $k=4$ we have $p = aaba$, for $k=6$ we have $p = aabaaa$, etc.), and the patterns for sub-figure pattern 2 are prefixes of length 4, 6, 8, 10 of $p_2 = abababababab$. The experiments clearly confirm our complexity analysis and show that

- for a fixed window size w, the maximum complexity of preprocessing is reached for $w \sim 2k$ (see Fig. 5): the theoretical explanation is that, for a fixed w, $\binom{w}{k}$ has its maximum when $k = w/2$,
- if we do not count the preprocessing, then our first method is 5–10 times faster than the standard method: indeed the standard method takes time nk and ours takes time n (see Fig. 5),
- even including the preprocessing and taking patterns which give the worst-case complexity, our first method outperforms the standard method as soon as the data is large (texts of length $n \geqslant 10^6$, see Fig. 4),

[6] Our algorithm is alphabet-dependent, however, as noted in [9], we can map all letters not occurring in the pattern on a single letter, and hence assume that the alphabet has at most $k+1$ letters.

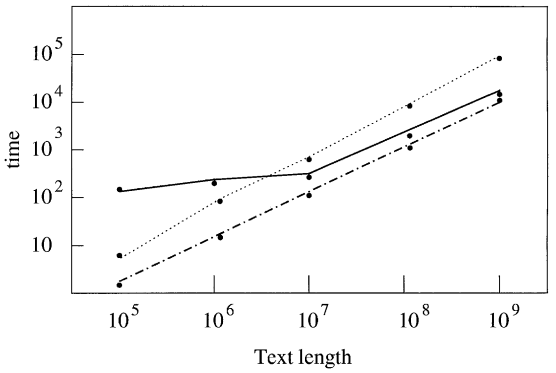

Fig. 4. Window width $w = 12$, and pattern *aabaaa*. The thick solid line represents the total running time of our first algorithm including preprocessing, the dashed-and-dotted line represents the running time of our MP-RAM algorithm, and the dotted line represents the running time of the standard algorithm.

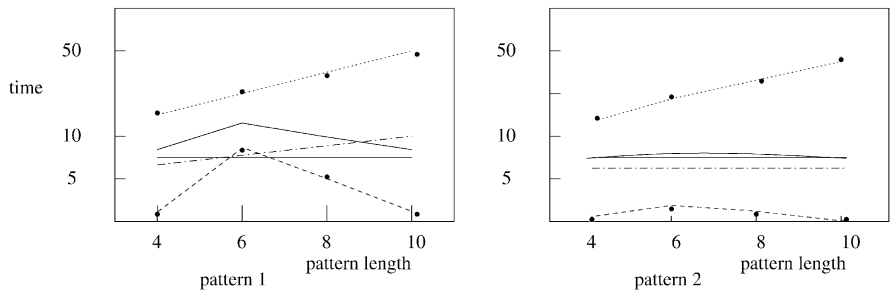

Fig. 5. Text of length 10^7, window width $w = 12$, and varying pattern lengths for 2 different patterns. The upper solid line represents the total running time of our first algorithm, the lower solid line its running time after preprocessing, the dashed line its preprocessing time; the dashed-and-dotted line represents the running time of our MP-RAM algorithm, and the dotted line the running time of the standard algorithm. The time scale is *linear for the preprocessing* times, and *logarithmic for the running times*.

- Fig. 5 shows that (i) the preprocessing of our first method is quite dependent of the pattern, but after the preprocessing our algorithm runs in constant time equal to the length of the text, for all patterns, and (ii) our MP-RAM algorithm is less dependent on the pattern.
- our MP-RAM algorithm is 2–10 times faster than the standard algorithm: the speed-up is by a factor of 2 for small patterns ($k < 5$) and can reach a factor of 10 for large windows and large patterns ($w \geqslant 30$, $k \geqslant 20$); in average, our MP-RAM algorithm is 3 times faster than the standard algorithm.

The *agrep* program of [22] solves the *WESP* (and not the *WASP*) in the special case when the window size is close to the pattern size, i.e. $w < \min(2k, 2k + 9)$. We compared our MP-RAM algorithm with *agrep*: we ran both programs on a DEC-alpha, and measured time using the UNIX *time* command. In all the cases that we tested (we

tested cases when the encoding of a state of automaton $\mathscr{A}$ of Theorem 1 fits in one or two computer words, i.e. $k \log w \leqslant 64$), our MP-RAM algorithm is 20% faster than *agrep* for elapsed time in seconds and it is 2 times faster than *agrep* for CPU user time.

5. Conclusion

We presented two new efficient algorithms for the *WASP*, linear in the size of the text. The complexity analysis showed that

1. our first algorithm (including preprocessing) is faster than the standard algorithm for large data and small windows (it blows up for large windows);
2. our second algorithm, based on MP-RAMs, is more efficient in all cases.

This was clearly confirmed by the implementation. Note that for both methods, implementing the *WASP* is no more difficult than implementing the *WESP*. This does not hold in general; usually counting problems are much harder that the corresponding existence problems: e.g., for the related problem of matching strings with don't cares the existence problem is in linear time while the counting problem is in polynomial time in [13], and in the special case of [14], the existence problem is in logarithmic time while the counting problem is in sublinear time.

References

[1] A. Aho, Algorithms for Finding Patterns in Strings, in: van Leeuwen (Ed.), Handbook of Theoretical Computer Science, vol. 1, North-Holland, Amsterdam, 1990, pp. 255–300.
[2] A. Aho, J. Hopcroft, J. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, London, 1974.
[3] L. Boasson, P. Cegielski, I. Guessarian, Y. Matiyasevich, Window-accumulated subsequence matching problem is linear, Proc. PODS'99 (Principles of Database Systems), ACM Press, New York, 1999, pp. 327–336.
[4] R. Baeza-Yates, G. Gonnet, A new approach to text searching, Comm. ACM 3 (1992) 74–82.
[5] R. Baeza-Yates, G. Navarro, A faster algorithm for approximate string matching, Proc. 1996 Combinatorial Pattern Matching Conf., Lecture Notes in Computer Science, vol. 1075, Springer, Berlin, 1996, pp. 1–23.
[6] A. Ben-Amram, Z. Galil, On the power of the shift instruction, Inform and Comput. 117 (1995) 19–36.
[7] S. Cook, Linear time simulation of deterministic two-way pushdown automata, in: C. Freiman (Ed.), Proc. IFIP Congress, North-Holland, Amsterdam, 1971, pp. 75–80.
[8] M. Crochemore, String-matching with constraints, Proc. MFCS'88, Lecture Notes in Computer Science, vol. 324, Springer, Berlin, 1988, pp. 44–58.
[9] G. Das, R. Fleischer, L. Gąsienic, D. Gunopoulos, J. Kärkkäinen, Episode matching, Proc. 1997 Combinatorial Pattern Matching Conf., Lecture Notes in Computer Science, vol. 1264, Springer, Berlin, 1997, pp. 12–27.
[10] Z. Galil, String matching in real time, J. ACM 28 (1981) 134–149.
[11] D.E. Knuth, Big omicron and big omega and big theta, SIGACT News (1976) 18–24.
[12] D. Knuth, J. Morris, V. Pratt, Fast pattern matching in strings, SIAM J. Comput. 6 (2) (1977) 323–350.
[13] G. Kucherov, M. Rusinovitch, Matching a set of strings with variable length don't cares, Theoret. Comput. Sci. 178 (1997) 129–154.
[14] U. Manber, R. Baeza-Yates, An algorithm for string matching with a sequence of don't cares, Inform. Process. Lett. 37 (1991) 133–136.

[15] H. Mannila, Methods and Problems in Data Mining, Proc. 1997 ICDT Conf., Lecture Notes in Computer Science, vol. 1186, Springer, Berlin, 1997, pp. 41–55.
[16] H. Mannila, H. Toivonen, A. Verkamo, Discovering frequent episodes in sequences, Proc. 1995 KDD Conf., 1995, pp. 210–215.
[17] Y. Matiyasevich, Real-time recognition of the inclusion relation, Zapiski Nauchnykh Leningradskovo Otdeleniya Mat. Inst. Steklova Akad. Nauk SSSR 20 (1971) 104–114. (Translated into English, J. Soviet Math. 1 (1973) 64–70).
[18] V. Pratt, M. Rabin, L. Stockmeyer, A charaterization of the power of vector machines, Proc. SToC 74, pp. 122–134.
[19] A. Slissenko, String-matching in real time, Lecture Notes in Computer Science, vol. 64, Springer, Berlin, 1978, pp. 493–496.
[20] J. Trahan, M. Loui, V. Ramachandran, Multiplication, division and shift instructions in parallel random access machines, Theoret. Comput. Sci. 100 (1992) 1–44.
[21] E. Ukkonen, On-line construction of suffix-trees, Algorithmica 14 (1995) 249–260.
[22] S. Wu, U. Manber, Fast text searching, Comm. ACM 3 (1992) 83–91.