Multiple serial episode matching

Patrick Cegielski, Irene Guessarian, Yuri Matiyasevich

Multiple serial episodes matching****

Patrick Cégielski * — Irène Guessarian ** — Yuri Matiyasevich ***

* *LACL, UMR-FRE 2673, Université Paris 12, Route forestière Hurtault, F-77300 Fontainebleau, France, cegielski@univ-paris12.fr*

** *LIAFA, UMR 7089 and Université Paris 6, 2 Place Jussieu, 75254 Paris Cedex 5, France; send correspondence to ig@liafa.jussieu..fr*

*** *Steklov Institute of Mathematics, Fontanka 27, St. Petersburg, Russia. yumat@pdmi.ras.ru*

**** *Support by INTAS grant 04-77-7173 is gratefully acknowledged.*

ABSTRACT. *In [BCGM01] we have generalized the Knuth-Morris-Pratt (KMP) pattern matching algorithm and defined a non-conventional kind of RAM, the MP–RAMs which model more closely the microprocessor operations, and designed an $O(n)$ on-line algorithm for solving the serial episode matching problem on MP–RAMs when there is only one single episode. We here give two extensions of this algorithm to the case when we search for several patterns simultaneously and compare them. More preciseley, given $q+1$ strings (a text t of length n and q patterns $m_1, \ldots, m_q$) and a natural number w, the* multiple serial episode matching problem *consists in finding the number of size w windows of text t which contain patterns $m_1, \ldots, m_q$ as subsequences, i.e. for each m_i, if $m_i = p_1, \ldots, p_k$, the letters $p_1, \ldots, p_k$ occur in the window, in the same order as in m_i, but not necessarily consecutively (they may be interleaved with other letters).*

KEYWORDS: *Subsequence matching, algorithm, frequent patterns, episode matching, datamining.*

1. Introduction

The recent development of datamining induced the development of computing techniques, among them is episode searching and counting. An example of frequent serial episode search is as follows: let t be a text consisting of requests to a university webserver ; assume we wish to count how many times, within at most 10 time units, the sequence $e_1e_2e_3e_4$ appears, where e_1 = 'Computer Science', e_2 = 'Master', e_3 = 'CS318 homepage', e_4 = 'Assignment'. It suffices to count the number of 10-windows of t containing the subsequence $p = e_1e_2e_3e_4$. If e_1, e_2, e_3, e_4 must appear in that same order in the window, the episode is said to be *serial*, if they can appear in any order, the episode is said to be *parallel*; a partial order can also be imposed on the events composing an episode (see [MTV95], which proposes several algorithms for episode searching). Searching serial episodes is more complex than searching parallel episodes. Of course, if one has to scan a $\log$ file, it is better to do it for several episodes $e_1e_2 \dots e_n$, $f_1f_2 \dots f_m$, $g_1g_2 \dots g_p$ simultaneously. We will hence investigate the search of several serial episodes in the same window: each serial episode is ordered, but no order is imposed among occurrences of the episodes in the window.

The problem we address is the following: given a text t of length n, patterns $m_1, \dots, m_q$ on the same alphabet A and an integer w, we wish to determine the number of size w windows of text containing all q patterns as serial episodes, *i.e.* the letters of each m_i appear in the window, in the same order as in m_i, but they need not be consecutive because other letters can be interleaved. When searching for a single pattern m, this problem with arguments the window size w, the text t and pattern m is called *serial episode matching problem* in [MTV95], *episode matching* in [DFGGK97] and *subsequence matching* in [AHU74]; a related problem is the *matching with don't cares* of [MBY91, KR97].

This problem is an interesting generalisation of *pattern-matching*. Without the window size restriction, it is easy to find in linear time whether p occurs in the text: if $p = p_1 \dots p_k$, a finite state automaton with $k+1$ states $s_0, s_1, \dots, s_k$ will read the text; the initial state is s_0; after reading letter p_1 we go to state s_1, then after reading letter p_2 we go to state s_2, $\dots$; the text is accepted as soon as state s_k is reached. Episode matching within a w-window is harder; its importance is due to potential applications to datamining [M97, MTV95] and molecular biology[MBY91, KR97, NR02].

For the problem with a single episode in w-windows, a standard algorithm is described in [DFGGK97, MTV95]. It is close to the algorithms of *pattern-matching* [A90, AHU74] and its time complexity is $O(nk)$. Another *on-line* algorithm is described in [DFGGK97]: the idea is to slice the pattern in $k/\log k$ well-chosen pieces organised in a *trie*; its time complexity is $O(nk/\log k)$. We gave an *on-line* algorithm reading the text t, each text symbol being read only once and whose time complexity is $O(n)$ [BCGM01].

In this paper, we describe two efficient algorithms (Section 3) for solving the problems of simultaneous search of multiple episodes. These algorithms use the *MP–RAM*, that we introduced in [BCGM01], to model microprocessor basic operations, using only the fast operations on bits (*shifts*), and bit-wise addition; this gives an on-line algorithm in time $O(nq)$ (theorem 1). In practice, this algorithm based on MP–RAMs and a new implementation of *tries*, is much faster as shown in section 4. We believe that other algorithms can be considerably improved if programmed on MP–RAMs.

Our algorithm relies upon two ideas: 1) preprocess patterns and window size to obtain a finite automaton solving the problem as in Knuth, Morris, and Pratt algorithm [KMP77] (the solutions preprocessing the text [T02, MBY91, S71, U95] are prohibitive here because of their space complexity) and 2) code the states of this automaton to compute its transitions very quickly on MP-RAMs, without precomputing, nor storing the automaton: using the automaton itself is also prohibitive, not the least because of the number of states; we emulate the behaviour of the automaton without computing the automaton. We study: (a) the case when the patterns have no common part and (b) the case when they have similar parts. In each case, an appropriate preprocessing of the set of patterns enables us to build an automaton solving the problem and we show that the behaviour of this automaton can be emulated on-line on MP-RAMs. Moreover, the time complexity of the preprocessing is insignificant because it is smaller than the text size by several orders of magnitude: typically, window and patterns will consist of a few dozen characters while the text will consist of several million characters.

The paper is organised as follows: in section 2, we define the problem, in section 3 we describe the algorithms searching multiple episodes in parallel; we present the experimental results in section 4.

Figure 1: A French advertisement

Figure 2: A text with two 5-windows containing “vie" (in gray), and a single 5-window containing “vile".

2. The problem

2.1. *The (multiple) episode problem*

An *alphabet* is a finite non-empty set A. A *length* n *word* on A is a mapping t from $\{1, \ldots, n\}$ to A. The only length zero word is the *empty word*, denoted by ε. A non-empty word $t : i \mapsto t_i$ is denoted by $t_1 t_2 \cdots t_n$. A *language* on alphabet A is a set of words on A.

Let $t = t_1 t_2 \cdots t_n$ be a word which will be called the *text* in the paper. The word $p = p_1 p_2 \cdots p_k$ is a *factor* of t iff, there exists an integer j such that $t_{j+i} = p_i$ for $1 \leq i \leq k$. A size w *window* of on t, in short w-window, is a size w factor $t_{i+1} t_{i+2} \cdots t_{i+w}$ of t; there are $n - w + 1$ such windows in t. The word p is an *episode* (or *subsequence*) of t iff there exist integers $1 \leq i_1 < i_2 < \cdots < i_k \leq n$ such that $t_{i_j} = p_j$ for $1 \leq j \leq k$. If moreover, $i_k - i_1 < w$, p is an *episode* of t *in a* w*-window*.

Example 1 If $t =$ “dans ville il y a vie" (a French advertisement, see figure 1) then “vie" is a factor and hence a subsequence of t. “vile" is neither a factor, nor a subsequence of t in a 4-window, but it is a subsequence of t in a 5-window. See figure 2.□

Given an alphabet A, and words $t, m_1, \ldots, m_q$ on A:

– the simultaneous *pattern-matching* problem consists in finding whether $m_1, \ldots, m_q$ are factors of t,
– given moreover a window size w:

- the *subsequence existence* problem consists in finding whether $m_1, \ldots, m_q$ are subsequences of t in a w-window;
- the *multiple episode search* problem consists in counting the number of w-windows in which all of $m_1, \ldots, m_q$ are subsequences of t.

For the simultaneous search of several subsequences $m_1, \ldots, m_q$, we have various different problems:

– either we count the number of occurrences of each m_i in a w-window (not necessarily the same): this case will be useful for searching in parallel, with a single scan of the text, a set of patterns which are candidates for being frequent.

– or we count the number of windows containing all the m_is: this case will be useful for trying to verify association rules. For example, the association rule $m_2, \ldots, m_q \Longrightarrow m_1$ will be useful if the number of w-windows containing all the $m_2, \ldots, m_q$ is high enough, and to check that, we will count the w-windows containing *all of* $m_2, \ldots, m_q$. Our method will enable us to verify more easily both the validity of the association rule (“among the windows containing $m_2, \ldots, m_q$ many contain also m_1”) and the fact that it is interesting enough (“many windows contain $m_2, \ldots, m_q$”): it will suffice to count simultaneously the windows containing $m_2, \ldots, m_q$ and the windows containing $m_1, m_2, \ldots, m_q$.

A naive solution exists for *pattern-matching*. Its time complexity on RAM is $O(nk)$, where k is the pattern size. Knuth, Morris, and Pratt [KMP77] gave a well-known algorithm solving the problem in linear time $O(n + k)$. A solution in $O(nk)$ is given in [MTV95] for searching a single size k episode. We gave in [BCGM01] an algorithm with time complexity $O(n)$ (on MP–RAM) for searching a single episode.

2.2. *The notation* $o(nk)$

Let us first make precise the meaning of the notation $o(nk)$.

The notation $o(h(n))$ was introduced to compare growth rates of functions with one argument; for comparing functions with several arguments, various non-equivalent interpretations $o(h(n, m, ...))$ are possible. Consider a function $t(n, k)$; $t(n, k) = o(nk)$ could mean:

1) either $\lim\limits_{n+k\to+\infty} t(n, k)/nk = 0$;

2) or $\lim\limits_{\substack{n\to+\infty \\ k\to+\infty}} t(n, k)/nk = 0$, i.e. $\forall\epsilon, \exists N,\ \forall n, \forall k\big(n > N$ and $k > N \Longrightarrow t(n, k) < \epsilon nk\big)$.

With meaning 1, no algorithm can solve the *single episode within a window* problem in time $o(nk)$. Indeed, any algorithm for the *episode within a window* problem must scan the text at least once, hence $t(n, k) \geq n$. For a given k, for example $k = 2$,we have $t(n, k)/nk \geq 1/2$. Hence $\lim\limits_{n+k\to+\infty} t(n, k)/nk = 0$ is impossible. We thus have to choose meaning 2.

2.3. *Algorithms on MP–RAM*

Given a window size w and q patterns, we preprocess (patterns + window size w) to build a virtual finite state automaton $\mathcal{A}$; we will then emulate on-line the behaviour of $\mathcal{A}$ to scan text t and count in time nq the number of windows containing our patterns as episodes. Note that our method is different from both: 1) methods preprocessing the text [T02, MBY91, S71, U95] (we preprocess the pattern) and 2) methods using suffixes of the pattern [C88, MBY91, KR97, U95] (we use prefixes of the patterns). We encode the subset of states of $\mathcal{A}$ needed to compute the transitions on-line on an MP-RAM. Indeed, $\mathcal{A}$ has $O(w+1)^k$ state, where k is the size of the structure encoding the q patterns $m_1, \ldots, m_q$; for w and q large, the time and space complexity for computing the states of $\mathcal{A}$ becomes prohibitive, whence the need to compute the states on-line quickly without having to precompute nor store them. We introduced MP-RAMs to this end.

Pattern-matching algorithms are often given on RAMs. This model is not good when there are too many different values to be stored, for example $O(w + 1)^k$ states for $\mathcal{A}$. As early as 1974, the motivation of [PRS74] for introducing "vector machines" was the remark that boolean bit-wise operations and shifts which are implemented on computers are faster and better suited for many problems. This work was the starting point of a series of papers: [TRL92, BG95] comparing the complexities of computations on various models of machines allowing for boolean bit-wise operations and shifts with computation complexities on classical machines, such as Turing machines, RAMs etc. The practical applications of this technique to various *pattern-matching* problems start with [BYG92, WM92]: they are known as *bit-parallelism*, or *shift-OR* techniques. We follow this track with the episode search problem, close to the problems studied in [BYG92, WM92, BYN96], albeit different from these problems.

In the sequel, we use a variant of RAMs, which is a more realistic computation model in some aspects, and we encode $\mathcal{A}$ to ensure that (i) each state of $\mathcal{A}$ is stored in a single memory cell and (ii) only the most basic microprocessor operations are used to compute the transitions of $\mathcal{A}$. Our RAMs have the same control structures as classical RAMs[1], but the operations are enriched by allowing for boolean bit-wise operations and shifts, which we will preferably use whenever possible. Such RAMs are close to microprocessors, this is why we called them MP–RAMs.

Definition 1 *An MP–RAM is a RAM extended by allowing new operations:*

1) the bit-wise and, *denoted by* $\&$,
2) the left shift, *denoted by* $\ll$ *or* shl, *and*
3) the right shift, *denoted by* $\gg$ *or* shr.

The new operations are low-level operations, executable much faster than the more complex *MULT, DIV* operations.

1. See [AHU74] pages 5–11, for a definition of classical RAMs.

Figure 3: Trie representing tu, tue and $tutu$. The full black circles indicate ends of patterns.

Example 2 *Assume our MP–RAMs have unbounded memory cells. We will have for example:* $(10110 \,\&\, 01101) = 100$, $(10110 \ll 4) = 101100000$ *and* $(10110 \gg 3) = 10$*. If memory cells have at most 8 bits, we will have:* $(10110 \ll 4) = 1100000$*, that will be written as* $(00010110 \ll 4) = 01100000$*.*

3. Parallel search of several patterns

Let us recall the problems. Given patterns $m_1, m_2, \ldots, m_q$, we can:

– either count the number of occurrences of each m_i in a w-window (not necessarily the same one);

– or count the number of w-windows containing $m_1, m_2, \ldots, m_q$.

The algorithm we described in [BCGM01] for counting the number of w-windows containing *a single* pattern m can be adapted to all these cases, only the acceptance or counting condition will change.

To search simultaneously several patterns $m_1, \ldots, m_q$, [WM92] propose a method concatenating all the patterns. To search simultaneously several episodes $m_1, \ldots, m_q$, we generalise our algorithm [BCGM01]: we use q counters $c_1, \ldots, c_q$ initially set to 0, and we define an appropriate multiple counting condition such that each time m_i is in a w-window, the corresponding counter c_i is incremented. This method has a drawback: if the patterns are too long, it will need more than one memory cell for coding the states of the automaton. For searching multiple patterns the method proposed by [DFGGK97] to optimise the search, when words $m_1, \ldots, m_q$ have common prefixes, is to organise $m_1, \ldots, m_q$ in a *trie* [K97] before applying the standard algorithm. We apply our algorithm on MP-RAMs in a similar way, and implement *tries* in a new way. We thus can encode the set of patterns compactly, and then encode the states of the automaton on a single memory cell.

3.1. *Representing patterns by a trie*

Consider for example episodes $m_1 = tu$, $m_2 = tue$, and $m_3 = tutu$. We choose this example because it illustrates most of the difficulties in encoding the automaton: episode *taie* is very simple because all letters are different, *tati* is less simple because there are two occurrences of t which must be distinguished, *tutu* a bit more complex (the first occurrence of *tu* must be distinguished from the second one), *turlututu* would be even more complex. We represent these three episodes by the trie t pictured in figure 3.

We implement this trie t by the three tables below:

$tr =$

t	u	e	t	u

$pr =$

0	1	2	2	4

$f =$

2	3	5

Table tr represents the "flattened" trie. Predecessors are in table pr: $pr[i]$ gives the index in tr of the parent of $tr[i]$ in the trie; 0 means there is no predecessor and hence it is a pattern start[2]. Finally f marks patterns ends: $f[i]$ is the index in tr of the end of pattern i.

3.2. *Preprocessing the trie and algorithm*

We preprocess the trie of patterns and this gives us a finite state automaton $\mathcal{A}$. Its alphabet is A. The states are the k-tuples of integers $\langle l_1, \ldots, l_k \rangle$ with l_j belonging to $\{1, \ldots, w, +\infty\}$, where k is the size of table tr and w the window size.

2. Numbering of indices starts at 1 in order to indicate pattern starts by 0.

$$\overbrace{0\,\vdots\underbrace{\qquad\qquad\qquad}_{\text{binary expansion of } l_k}| \quad \dots \quad 0\,\vdots\underbrace{\qquad\qquad\qquad}_{\text{binary expansion of } l_2}| \; 0\,\vdots\underbrace{\qquad\qquad\qquad}_{\text{binary expansion of } l_1}|}^{\text{binary expansion of } L}$$

Figure 4: Encoding of $\langle l_1, \dots, l_k \rangle$.

$$L = \begin{array}{|c|c|c|c|c|}\hline 0\,\vdots\,\overline{l_5} & 0\,\vdots\,\overline{l_4} & 0\,\vdots\,\overline{l_3} & 0\,\vdots\,\overline{l_2} & 0\,\vdots\,\overline{l_1} \\ \hline\end{array}$$

Figure 5: Encoding of $\langle l_1, \dots, l_5 \rangle$; $\overline{l_i}$ is the binary expansion of l_i.

We describe informally the behaviour of $\mathcal{A}$. $\mathcal{A}$ scans t, it will be in state $\langle l_1, \dots, l_k \rangle$ after scanning $t_1 \dots t_m$ iff l_i is the length of the shortest suffix[3] of $t_1 \dots t_m$ shorter than w and containing $tr[j_i] \dots tr[i]$ as subsequence for $i = 1, \dots, k$, where $tr[j_i] \dots tr[i]$ is the sequence of letters labelling the path going from the root of the trie to the node represented by $tr[i]$. If no suffix (of length less than w) of $t_1 \dots t_m$ contains $tr[j_i] \dots tr[i]$ as a subsequence, we let $l_i = +\infty$.

Let us now describe our algorithm. Let Ω be the least integer such that $w + 2 \leq 2^\Omega$. The rôle of $+\infty$ is played by $2^\Omega - 1$, whose binary encoding is a sequence of Ω ones. We define the function Next_Ω by:

$$\mathrm{Next}_\Omega(l) = \begin{cases} l+1, & \text{if } l < 2^\Omega - 1; \\ 2^\Omega - 1, & \text{else.} \end{cases}$$

State $\langle l_1, \dots, l_k \rangle$ is encoded by integer:

$$L = \sum_{i=1}^{k} l_i (2^{\Omega+1})^{i-1} = \sum_{i=1}^{k} \Big(l_i \ll \big((\Omega+1)(i-1)\big)\Big). \qquad [1]$$

Let $\overline{l_i}$ denote the binary expansion of l_i, $i = 1, \dots, k$, prefixed by zeros in such a way that $\overline{l_i}$ occupies Ω bits (all l_is are smaller than $2^\Omega - 1$, hence they will fit in Ω bits). The binary expansion of L is obtained by concatenating the $\overline{l_i}$s, each prefixed by a zero (figure 4). These initial zeros are needed for implementing function Next_Ω to indicate overflows. Every integer smaller than $2^{k(\Omega+1)}$ can be written as k *big blocks* of $(\Omega + 1)$ bits, the first bit of each big block is 0 (and is called the *overflow bit*) and the Ω remaining bits constitute a *small block*. The blocks are numbered 1 to k from right to left (the rightmost block is block 1, the leftmost block is block k).

By the definition in equation (1), the initial state $\langle +\infty, \dots, +\infty \rangle$ is encoded by:

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

One might see a multiplication here. In fact we will need a loop for $i = 1$ to k. We will execute each time we go through the loop a shift of $\Omega + 1$, and the multiplication will disappear. All equations below are treated in the same way.

Assume that the window size is $w = 13$ hence $\Omega = 4$. With the notations of figure 5, state $l = \langle 2, 5, \infty, 5, \infty \rangle$ is encoded by:

$$L = \begin{array}{|c|c|c|c|c|}\hline 0\,\vdots\,\overline{15} & 0\,\vdots\,\overline{5} & 0\,\vdots\,\overline{15} & 0\,\vdots\,\overline{5} & 0\,\vdots\,\overline{2} \\ \hline\end{array}$$

The initial state is represented by:

$$I_0 = \begin{array}{|c|c|c|c|c|}\hline 0\,\vdots\,1111 & 0\,\vdots\,1111 & 0\,\vdots\,1111 & 0\,\vdots\,1111 & 0\,\vdots\,1111 \\ \hline\end{array}$$

3. Word s is a *prefix* (resp. *suffix*) of word t iff there exists a word v such that $t = sv$ (resp. $t = vs$).

or, writing $\underline{1}$ instead of the Ω ones representing ∞:

$$I_0 = \begin{array}{|c|c|c|c|c|}\hline 0\vdots\underline{1} & 0\vdots\underline{1} & 0\vdots\underline{1} & 0\vdots\underline{1} & 0\vdots\underline{1} \\ \hline\end{array}$$

In transition $l = \langle l_1, \ldots, l_k\rangle \xrightarrow{\sigma} l' = \langle l'_1, \ldots, l'_k\rangle$, the l'_i component of the new state l' is either $\mathrm{Next}_\Omega(l_{pr[i]})$ or $\mathrm{Next}_\Omega(l_i)$ according to whether the scanned letter σ is equal to $tr[i]$ or not. The cases $l'_i = Next_\Omega(l_{pr[i]})$ and $l'_i = Next_\Omega(l_i)$ respectively yield a *first type computation* and a *second type computation*.

To generalise the algorithm of [BCGM01], we must define several *masks* M_σ for each letter σ of alphabet A. If σ has several occurrences in table tr, we will need as many masks M_σ as occurrences $tr[i]$ and $tr[i']$ of σ with $j = i - pr[i] \neq i' - pr[i'] = j'$ (a single mask will suffice for the set of all occurrences such that $i - pr[i]$ has the same value j, because they correspond to the same shift of j big blocks). The M^j_σ are the masks preparing first type computations. Precisely, if $tr[i] = \sigma$ and $i - pr[i] = j$, the operation $\big(L \ll j(\Omega+1)\big)\&M^j_\sigma$ will shift everything of j big blocks leftwards and will erase the blocks for which $\sigma \neq p_i$ or $i - pr[i] \neq j$. For $i > 1$, the i-th block will thus contain $\overline{l_{pr[i]}}$ iff $tr[i] = \sigma$ and $i - pr[i] = j$. It will contain $\underline{0}$ otherwise.

In our example ($m_1 = tu$, $m_2 = tue$, and $m_3 = tutu$), we will need two masks M_t but a single mask M_u will suffice:

$$M^1_t = \begin{array}{|c|c|c|c|c|}\hline 0\vdots\underline{0} & 0\vdots\underline{0} & 0\vdots\underline{0} & 0\vdots\underline{0} & 0\vdots\underline{1} \\ \hline\end{array}$$

$$M^2_t = \begin{array}{|c|c|c|c|c|}\hline 0\vdots\underline{0} & 0\vdots\underline{1} & 0\vdots\underline{0} & 0\vdots\underline{0} & 0\vdots\underline{0} \\ \hline\end{array}$$

$$M^1_u = \begin{array}{|c|c|c|c|c|}\hline 0\vdots\underline{1} & 0\vdots\underline{0} & 0\vdots\underline{0} & 0\vdots\underline{1} & 0\vdots\underline{0} \\ \hline\end{array}$$

$$M^1_e = \begin{array}{|c|c|c|c|c|}\hline 0\vdots\underline{0} & 0\vdots\underline{0} & 0\vdots\underline{1} & 0\vdots\underline{0} & 0\vdots\underline{0} \\ \hline\end{array}$$

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

Mask N_σ is the complement of $\sum_j M^j_\sigma$, preparing second type computations. The operation $L\&N_\sigma$ will erase the blocks for which $\sigma = tr[i]$. For our example, we have:

$$N_t = \begin{array}{|c|c|c|c|c|}\hline 0\vdots\underline{1} & 0\vdots\underline{0} & 0\vdots\underline{1} & 0\vdots\underline{1} & 0\vdots\underline{0} \\ \hline\end{array}$$

$$N_u = \begin{array}{|c|c|c|c|c|}\hline 0\vdots\underline{0} & 0\vdots\underline{1} & 0\vdots\underline{1} & 0\vdots\underline{0} & 0\vdots\underline{1} \\ \hline\end{array}$$

$$N_e = \begin{array}{|c|c|c|c|c|}\hline 0\vdots\underline{1} & 0\vdots\underline{1} & 0\vdots\underline{0} & 0\vdots\underline{1} & 0\vdots\underline{1} \\ \hline\end{array}$$

Generally, if k is table tr size,

$$M^j_\sigma = \sum_{\substack{tr[i]=\sigma \text{ and } pr[i]=i-j \\ 1\leq i\leq k}} \Big(\big((1 \ll \Omega) - 1\big) \ll \big((\Omega+1)(i-1)\big)\Big).$$

and

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

N_σ is the complement of $\sum_j M^j_\sigma$.

Transition $l = \langle l_1, \ldots, l_k\rangle \xrightarrow{\sigma} l' = \langle l'_1, \ldots, l'_k\rangle$ is computed by:

$$T = \sum_j \big((L \ll j(\Omega+1))\&M^j_\sigma\big) + (L\&N_\sigma) + E_1$$

where:

$$E_1 = \begin{array}{|c|c|c|c|c|}\hline 0 \vdots 0001 & 0 \vdots 0001 & 0 \vdots 0001 & 0 \vdots 0001 & 0 \vdots 0001 \\ \hline \end{array}$$

Adding E_1 amounts to add 1 to each small block.

In our example, if we scan letter t, the transition is computed by:

$$T = \big((L \lll 2(\Omega+1))\&M_t^2\big) + \big((L \lll (\Omega+1))\&M_t^1\big) + (L\&N_t) + E_1$$

yielding for $l = \langle 2, 5, \infty, 5, \infty \rangle$, encoded by:

$$L = \begin{array}{|c|c|c|c|c|}\hline 0 \vdots \overline{15} & 0 \vdots \overline{5} & 0 \vdots \overline{15} & 0 \vdots \overline{5} & 0 \vdots \overline{2} \\ \hline \end{array}$$

the result:

$$T = \begin{array}{|c|c|c|c|c|}\hline 1 \vdots \overline{0} & 0 \vdots \overline{6} & 1 \vdots \overline{0} & 0 \vdots \overline{6} & 0 \vdots \overline{1} \\ \hline \end{array}$$

All the blocks contain the correct result, except for the leftmost block and the middle block where an overflow occurred. To treat blocks where overflow occurred it suffices of initialise again these blocks by replacing T with $L' = T - \big((T\&E_2) \ggg \Omega\big)$, where:

$$E_2 = \begin{array}{|c|c|c|c|c|}\hline 1 \vdots \underline{0} & 1 \vdots \underline{0} & 1 \vdots \underline{0} & 1 \vdots \underline{0} & 1 \vdots \underline{0} \\ \hline \end{array}$$

We find:

$$T\&E_2 = \begin{array}{|c|c|c|c|c|}\hline 1 \vdots \underline{0} & 0 \vdots \underline{0} & 1 \vdots \underline{0} & 0 \vdots \underline{0} & 0 \vdots \underline{0} \\ \hline \end{array}$$

Hence:

$$(T\&E_2) \ggg \Omega = \begin{array}{|c|c|c|c|c|}\hline 0 \vdots \overline{1} & 0 \vdots \underline{0} & 0 \vdots \overline{1} & 0 \vdots \underline{0} & 0 \vdots \underline{0} \\ \hline \end{array}$$

and finally:

$$L' = T - \big((T\&E_2) \ggg \Omega\big) = \begin{array}{|c|c|c|c|c|}\hline 0 \vdots \overline{15} & 0 \vdots \overline{6} & 0 \vdots \overline{15} & 0 \vdots \overline{6} & 0 \vdots \overline{1} \\ \hline \end{array}$$

Last we define a counter c_i for each pattern m_i, and increment it whenever $l_{f[i]} < w+1$, which is implanted by: $M_i\&L < (w+1)2^{(\Omega+1)(f[i]-1)}$, for $i = 1, \ldots, k$, where $M_i = \big((1 \lll \Omega) - 1\big) \lll \big((\Omega+1)(f[i]-1)\big)$.

Our algorithm treats the more complex case where we demand that all episodes appear in a same window, a case that cannot be treated by the separate counting of the number of windows containing each episode. A simple modification of the counting condition enables us to also count *with a single scan of the text* the number of windows containing each individual episode, in a more efficient way than if the text were to be scanned for each episode.

Theorem 1 *There exists an on-line algorithm in time $O(nq)$ solving the parallel search of q serial episodes in a size n text (assuming the episode alphabet has at most $\sqrt{n}/q$ letters) on MP–RAM.*

Proof: Let α be the number of letters of the alphabet. As in [DFGGK97], we treat in the same way all letters not occurring in the patterns; this leads to defining two masks M_{other} and N_{other} common to all such letters. Let $|w|$ be the length of the binary expansion of w. The algorithm consists of four steps:

1) compute (at most) $q \times (k+1)$ integers representing the masks M_σ^j, $(k+1)$ integers representing the masks N_σ and the integers $\Omega, \Delta, I_0, F, E_1, E_2$; all these integers are of size $k(|w|+2)$ and are computed *simultaneously* in k iterations at most. The integer k is the size of the trie representing the patterns: $k \leq \sum_{i=1}^{q} |m_i| \leq \sqrt{n}$.

2) let $c = 0$ (c is the number of w-windows containing all the patterns).

Figure 6: The continuous thin lines represent the execution time of the MP–RAM algorithm (with trie); the dotted line represents the execution time of the MP–RAM algorithm (with concatenation); the dashed lines the execution time of the standard algorithm (with concatenation) and the continuous thick lines the execution time of the standard algorithm (with trie).

3) let $L = I_0$.

4) scan text t; after scanning t_i, compute the new state L (*on-line and without preprocessing* with an MP–RAM) and if $c_i < w$ for $i = 1, \ldots, q$, increment c by 1.

Our algorithm uses only the simple and fast operations $\&$, together with a careful implementation of $\ll, \gg$ and addition. Step 1 of preprocessing is in time $qk(k+1)+q(k+1)+\log(w) \leq q(\sqrt{n})^2+2q\sqrt{n}+q+\log(w) = O(nq)$; in general, k, q and w are smaller than n by several orders of magnitude and we will have: $qk(k+1) + q(k+1) + \log(w) = o(n)$. In step 4 we scan text t linearly in time $O(n)$ and perform q comparisons (one for each counter c_i). Complexity is thus in time nq, hence finally a time complexity $O(nq)$ for the algorithm. $\Box$

4. Experimental results

The algorithm on MP–RAM has a better complexity than the standard algorithm, however, the underlying computation models being different, we checked experimentally that the MP–RAM algorithm is faster. We implemented all algorithms in C++. Experiments were realised on a PC (256 Mo, 1Ghz) with Linux. The text was a randomly generated file. We measured the time with machine clock ticks.

For searching multiple patterns, we took 3 to 5 patterns of length 2 to 4; in figure 6, case (a) is the case of patterns having no common prefix, and case (b) is the case of patterns having common prefixes. In case (a), the MP–RAM algorithm where we concatenate the patterns is at least twice as fast as the standard “naive” algorithm where patterns are concatenated; both standard algorithms (with patterns concatenated or organised in a trie) are equivalent, the algorithm with concatenation being slightly faster; this was predictable since a trie organisation will not give a significant advantage in that case; the MP–RAM algorithm where the patterns are organised in a trie is 30 to 50% faster than the standard algorithm with trie, and 10 to 15% slower than the MP–RAM algorithm where the patterns are concatenated. However, as soon as the total length of the patterns is larger than 7 or 8, or the window size is larger than 30, if patterns are concatenated, the automaton state can no longer be encoded in a single 32 bits memory cell, and it is better to use the MP–RAM algorithm with trie (figure 6 case (b)). Figure 6 case (b) shows that, for patterns having common prefixes, the MP–RAM algorithm with trie is 1.3 to 1.5 times faster than the standard algorithm with trie, itself 1.4 to 1.6 times faster than the standard algorithm with concatenation.

5. Conclusion

We presented new algorithms for multiple episode search, much more efficient than the standard algorithms. This was confirmed by our experimental analysis. Note that with our method, counting the number of windows containing several episodes is not harder than checking the existence of one window containing these episodes. This is not true with most other problems; usually counting problems are much harder than the corresponding existence problems: for example, for the “matching with don’t cares” problem, the existence problem is in linear time while the counting problem is in polynomial time [KR97] and in the particular case of [MBY91], the existence problem is in logarithmic time while the counting problem is in sub-linear time.

6. References

[A90] A. Aho, Algorithms for Finding Patterns in Strings, in Handbook of Theoretical Computer Science, Vol. 1, van Leeuwen Ed., North-Holland, Amsterdam (1990), pp. 255–300.

[AHU74] A. Aho, J. Hopcroft, J. Ullman, Design and Analysis of Computer Algorithms, Addison-Wesley, London (1974).

[BYRN99] R. Baeza-Yates, B. Ribeiro-Neto, Modern Information Retrieval, ACM Press Books, New-York (1999).

[BYG92] R. Baeza-Yates, G. Gonnet, A new approach to text searching, Communications of the ACM, Vol 35 (1992), 74–82.

[BYN96] R. Baeza-Yates, G. Navarro, A faster algorithm for approximate string matching, Proc. 1996 Combinatorial Pattern Matching Conf., LNCS 1075, Springer-Verlag, Berlin (1996), pp. 1–23.

[BG95] A. Ben-Amram, Z. Galil, On the power of the shift instruction, Inf. Comput. Vol 117 (1995), pp. 19–36.

[BCGM01] L. Boasson, P. Cegielski, I. Guessarian, Y. Matiyasevich, Window Accumulated Subsequence Matching is linear, Annals of Pure and Applied Logic Vol. 113 (2001), pp. 59-80.

[C88] M. Crochemore, String-matching with constraints, Proc. MFCS'88, LNCS 324, Springer-Verlag, Berlin (1988), pp. 44–58.

[CR94] M. Crochemore, W. Rytter, Text Algorithms, Oxford University Press, Oxford (1994).

[CHL01] M. Crochemore, C. Hancart, T.. Lecroq, Algorithmique du text, Vuibert, Paris (2001).

[DFGGK97] G. Das, R. Fleischer, L. Gąsienic, D. Gunopoulos, J. Kärkkäinen, Episode Matching, Proc. 1997 Combinatorial Pattern Matching Conf., LNCS 1264, Springer-Verlag, Berlin (1997), pp. 12–27.

[G81] Z. Galil, String matching in real time, J. Assoc. Comput. Mac. Vol 28, (1981), pp. 134–149.

[K97] D. Knuth, The art of computer programming, Vol. 1, Fundamental algorithms, Addison-Wesley, Reading (1997).

[KMP77] D. Knuth, J. Morris, V. Pratt, Fast Pattern Matching in Strings, SIAM Journal of Comput. Vol 6(2), (1977), pp. 323–350.

[KR97] G. Kucherov, M. Rusinovitch, Matching a Set of Strings with variable Length Don't Cares, Theor. Comput. Sc. Vol 178, (1997), pp. 129–154.

[MBY91] U. Manber, R. Baeza-Yates, An Algorithm for String Matching with a Sequence of Don't Cares, Inform. Proc. Letters Vol 37, (1991), pp. 133–136.

[M02] H. Mannila, Local and Global Methods in Data Mining: Basic Techniques and open Problems, Proc. ICALP 2002, LNCS 1186, Springer-Verlag, Berlin (2002).

[M97] H. Mannila, Methods and Problems in Data Mining, Proc. 1997 ICDT Conf., LNCS 1186, Springer-Verlag, Berlin (1997), pp. 41–55.

[MTV95] H. Mannila, H. Toivonen, A. Verkamo, Discovering Frequent Episodes in Sequences, Proc. 1995 KDD Conf., (1995), pp. 210–215.

[Ma71] Y. Matiyasevich, Real-time recognition of the inclusion relation, Zapiski Nauchnykh Leningradskovo Otdeleniya Mat. Inst. Steklova Akad. Nauk SSSR, Vol. 20, (1971), pp. 104–114. Translated into English, Journal of Soviet Mathematics, Vol. 1, (1973), http://logic.pdmi.ras.ru/˜yumat/Journal, pp. 64–70.

[NR02] G. Navarro, M. Raffinot, Flexible Pattern Matching in Strings Practical on-line search algorithms for texts and biological sequences, Cambridge University Press, Cambridge (2002).

[PRS74] V. Pratt, M. Rabin, L. Stockmeyer, A charaterization of the power of vector machines, Proc. SToC 74, pp. 122-134.

[S71] A. Slissenko, String-matching in real time, LNCS 64, Springer-Verlag, Berlin (1978), pp. 493–496.

[TRL92] J. Trahan, M. Loui, V. Ramachandran, Multiplication, division and shift instructions in parallel random access machines, Theor. Comput. Sc. Vol. 100, (1992), pp.. 1–44.

[T02] Z. Tronicek, Episode matching, 12th Annual Symposium, Combinatorial Pattern Matching 2001, Jerusalem, LNCS 2089, Springer-Verlag, Berlin (2002), pp. 143-146.

[U95] E. Ukkonen, On-line construction of suffix-trees, Algorithmica, Vol. 14, (1995), pp.. 249–260.

[WM92] S. Wu, U. Manber, Fast text searching, Communications of the ACM, Vol 35 (1992), 83–91.