Modulated String Searching[☆]

Alberto Apostolico[a,1], Péter L. Erdős[b,2], István Miklós[b,1,3], Johannes Siemons[c]

[a] *College of Computing, Georgia Institute of Technology, 801 Atlantic Drive, Atlanta, GA 30318, USA*
and
Istituto di Analisi dei Sistemi e Informatica, Consiglio Nazionale delle Ricerche, Viale Manzoni 30, Roma, Italy,
email: *axa@cc.gatech.edu*
[b] *Alfréd Rényi Institute of Mathematics, Reáltanoda u 13-15 Budapest, 1053 Hungary*
email: *<erdos.peter,miklos.istvan>@renyi.mta.hu*
[c] *School of Mathematics, University of East Anglia, Norwich, UK,*
email: *j.siemons@uea.ac.uk*
[d] *Institute of Computer Science and Control, Hungarian Academy of Sciences, Lágymányosi út 11, H-1111 Hungary*

Abstract

In his 1987 paper entitled *Generalized String Matching* Abrahamson introduced the concept of *pattern matching with character classes* and provided the first efficient algorithm to solve this problem. The best known solution to date is due to Linhart and Shamir (2009).

Another broad yet comparatively less intensively studied class of string matching problems is numerical string searching, such as for instance "less-than" or L_1-norm string searching. The best known solutions for problems in this class are based on FFT convolution after some suitable re-encoding.

The present paper introduces *modulated string searching* as a unified framework for string matching problems where the numerical conditions can be combined with some Boolean/numerical decision conditions on the character classes. One example problem in this class is the *locally bounded* L_1*-norm* matching problem with parameters b and τ : here the pattern "matches" a text of same length if their L_1-distance is at most b and if furthermore there is no position where the text element and pattern element differ by more than the local bound τ. A more general setup is that where the pattern positions contain character classes and/or each position has its own private local bound. While the first variant can clearly be handled by adaptation of the classic FFT method, the second one is far too complicated for this treatment. The algorithm we propose in this paper can solve all such problems efficiently.

The proposed framework contains two nested procedures. The first one, based on Karatsuba's fast multiplication algorithm, solves pattern matching with character classes within time $O\left(nm^{0.585}\right)$, where n and m are the text and pattern length respectively (under some reasonable conventions). This is slightly better than the complexity of Abrahamson's algorithm for generalized string matching but worse than algorithms based on FFT. The second procedure, which works as a plug-in within the first one and is tailored to the specific problem variant at hand, solves the numerical and/or Boolean matching problem with high efficiency. Some of the previously known constructions can be adapted to match or outperform

[☆]This research was carried out in part while A. Apostolico and J. Siemons were visiting the Rényi Institute, with support from the Hungarian Bioinformatics MTKD-CT-2006-042794 and Marie Curie Host Fellowships for Transfer of Knowledge.

[1]Additional support was provided by the United States-Israel Binational Science Foundation (BSF) Grant No. 2008217 and by the Research Program of Georgia Tech.

[2]Research supported in part by the Hungarian NSF under contract NK 78439 and K 68262.

[3]Research supported in part by the Hungarian NSF under contract PD 84297.

several (but not all) problem variations handled by the construction proposed here. The latter aims to be a general tool that provides a unified solution for all problems of this kind.

Key words: Pattern matching with character classes; Karatsuba's fast multiplication algorithm; locally bounded L_1-norm string matching on character classes; truncated L_1-norm string matching on character classes

---

1. Introduction

String searching is a basic primitive of computation. In the standard formulation of the problem, we are given a pattern and a text and are required to find all occurrences of the pattern in the text. Several variants of the problem have also been considered, such as allowing mismatches, insertions, deletions, swaps and so on.

In his paper [1] Abrahamson introduced the notion of *pattern matching with character classes* (or **PMCC** for short) which is specified as follows. The *pattern* P of length m is given as a sequence of character classes ($P[j] \subseteq \Sigma$) and the *text* T is a sequence from Σ^* (that is $T[i] \in \Sigma$). Here P occurs at location i in T if $\forall j : 1 \leq j \leq m,\ T[i+j-1] \in P[j]$. The problem of PMCC for a (typically long) text is to find all positions in the text T where the pattern P occurs. Standard string searching thus corresponds to the special case where each character class consists of exactly one element. In the original formulation PMCC was called *generalized pattern matching*.

Abrahamson proved that PMCC is harder than standard string searching and gave an algorithm for it. Since the algorithm deals with unrestricted alphabets, both the text and the pattern are encoded over the fixed auxiliary alphabet $\phi, 0, 1$. More precisely the text alphabet Σ is presumed to be the infinite set $\{\phi, a_l, a_2, \ldots\}$, where a_i is represented by the string $\#\bar{\imath}$, where $\bar{\imath}$ is the binary representation of i, without leading zeros. Symbol ϕ is represented by itself.

Now, let $\hat{M}$ denote the number of symbols used over the original alphabet to describe the pattern elements, and let M be the total length of the encoding of the pattern. Likewise, let n be the number of symbols in the text sequence, and N the total length of the encoding of the text. Then the time complexity of Abrahamson's algorithm is

$$O\left(M + N + n\hat{M}^{1/2}\text{polylog}(m)\right).$$

The state of the art for PMCC is due to Linhart and Shamir [7]. Their algorithm has the following impressive time complexity: having set $\kappa = \log_{|\Sigma|}(\log n / \log m)$, then it is $O(|\Sigma|^{1-\kappa} n \log m)$ for $\kappa \leq 1$, while for $\kappa > 1$ it becomes $O(n \log(m/\kappa))$. Their approach can be extended to solve PMCC with mismatches and to PMCC with subset matching. It is based on encoding the text and pattern using large prime numbers, and on an FFT-based convolution process. It is suitable for checking "element(s) in a subset relation" but not for more complicated conditions.

The problem of searching for strings consisting of numerical values rather than characters arises in countless applications and some variants have already been studied in combinatorial pattern matching. In these problems the *fitting* conditions are described in numerical terms. For example, in the *less-than* string searching problem (Amir and Farach [2]), the pattern fits the text if at each positionof the alignment the pattern value does not exceed the corresponding text value. Additional variants require the computation of the L_1-*distance* of the pattern from the text at each starting position (Amir, Landau and Vishkin [4], Lipsky [8]). Yet another version, known as the $k - L_1$-*distance problem* (Amir, Lipsky, Porat and Umanski [3]), consists of computing approximate matching in the L_1-metric.

These fast methods are also based on suitable encoding processes and on FFT, with corresponding time complexity. These algorithms do not seem to be applicable to numerical string searching with character classes and in general to those cases where a pointwise evaluation of individual comparisons is required.

In the next section we introduce the *modulated string searching* framework (or **MSS** for short) which combines the flexibility of PMCC with numerical calculations and/or more complicated Boolean conditions. We will give first a simple and naïve solution for the problem. (See Section 2.)

Our proposed approach for MSS is by a pair of nested procedures. The first one (see Section 3) is an algorithm to solve PMCC, based on Karatsuba's fast multiplication method. Its complexity is

$$O\left(nm^{0.585}\right)$$

where n and m are the text and pattern lengths, respectively, provided that all other parameters involved such as character class number etc. can be treated as constants. Here one could argue that the application of the Toom-Cook or the Schönhage algorithms [5, 9] yields a better performance. This is true, however, only for certain values of the text and pattern lengths. In addition, those algorithms require higher overheads, offsetting the overall gain. The above complexity is also worse than the complexity achieved in, say, [7]. However, the present method allows us to design a second procedure which works as a plug-in within the first one (see Section 4) and which solves a variety of numerical and/or Boolean problems. Indeed, the first procedure of our framework is always the same, while the plug-in procedure and its complexity depend heavily on specific matching conventions. Some of the previously known constructions can be adapted to match or outperform several (but not all) problem variations handled by the method proposed here, which therefore aims to be a general tool that provides a unified solution for all problems of this kind.

2. Modulated string searching framework

The framework for *modulated string matching on character classes* is as follows. The alphabet Σ is some set of natural numbers and b denotes an absolute constant. The pattern is a string of character classes (each class being a finite subset of Σ) whose length, that is the total number of the character classes, is denoted by m and the text is a finite string over Σ. The matching conditions are dictated by two functions with the following features. One of them depends on the particular variant of the problem and takes as arguments a character class and a character, and returns in constant time a score of the match. The second function takes as arguments the scores at the m positions of an alignment of the pattern against the text and returns true in case they add up to at most b, false otherwise.

Examples: Consider first the locally bounded L_1-distance string matching problems on character classes: Assume we are given two strings of equal length m over the natural numbers. Then the *L_1-distance* of these strings is $\sum_{i=1}^{m} |P_i - T_i|$, as usual. When one of the strings is given with character classes, the *L_1-distance on character classes* at a given position is defined as the smallest L_1-distance between the element of the first string at that position and any of the elements in the effacing class. For given pattern and text strings, the total distance at some starting text position is the sum of the above local distances. Let now b and τ be absolute constants. We say that the pattern *fits* at a given position of the text in *locally bounded L_1-distance* with parameters b and τ if there is no position in the corresponding substring of the text where the L_1-distance from the pattern element is bigger than τ and, in addition, the total L_1-distance of the two strings is at most b. The *locally bounded L_1-distance string matching problem on character classes* then is to find all positions of the text where the pattern fits.

When each pattern class consists of only one element then one can easily design a two-phase FFT based algorithm to solve this problem efficiently. However, if the classes are not singletons and / or each pattern position has its own private local bound then this is not feasible anymore.

A closely related notion is the *τ-truncated L_1-distance*. For two strings of length m this parameter is defined as $\sum_{i=1}^{m} \min(|P_i - T_i|, \tau)$. This can be visualized as testing a sequence of manufactured items against a standard of reference: the difference at each position describes, e.g., the cost to repair a token *in situ* while it is more economical to replace that token when the repair becomes too costly. When the pattern is given

in form of character sets then the obvious analogous definition applies. Then the *τ-truncated L_1-distance string matching problem on character classes* becomes to find all positions of the text where the τ-truncated L_1-distance of the pattern is at most b. In particular, in case of singleton character classes in the pattern and a big enough constant τ this yields the standard L_1-distance problem.

We finally list a third example which can be called *fitting assignment*. Here we simply have a map $\mathcal{A}$ from the pairs of text characters and pattern character classes (which may each specify its position within the pattern), say, to the integers $\mathbb{Z}$, or even more generally, to the reals $\mathbb{R}$. In this case we have to store the values for all possible pairs, but, in exchange, there is no need to calculate anything. It is clear that no FFT based method can solve this problem. On the other hand, these and other problems can be described and solved in the framework proposed here.

Modulated string searching can be solved easily by the following direct method. Align the pattern with the text starting at every position of the text. Each text character is matched against its corresponding set P_i. In most cases (like in the first two described above) finding out whether a text character fits into P_i can be managed with the help of a simple merge operation and so requires roughly $\log |P_i|$ time. The algorithm then compares the distance of the text element and the neighboring pattern elements against the threshold τ. Adding up for all text characters this yields $n \sum_{i=1}^{m} \log \left|P_i\right|$ time.

Before proceeding further we specify a more convenient representation of the pattern elements. For each pattern position i we have in general a character class P_i and we will represent this subset of the alphabet Σ by a binary characteristic vector $\bar{p}_i$ of length $|\Sigma|$. Since there are several kinds of "length" in this paper, we will use the term *dimension* for the length of a vector. So $\bar{p}_i$ is a vector of dimension $|\Sigma|$. If we represent our text symbols analogously by characteristic vectors $\bar{t}_j$ for all $j = 1, \ldots, n$, each one of which contains exactly one non-zero element, then the text character T_j and the pattern character class P_i match if and only if the scalar product $\langle \bar{p}_i, \bar{t}_j \rangle$ of the corresponding characteristic vectors is exactly 1.

With this notation, for each $j = 0, \ldots, n-m-1$, the substring of T starting at position $j+1$ and ending at $j+m$ fits the pattern string if and only if

$$v_{j+1} := \sum_{i=1}^{m} \langle \bar{p}_i, \bar{t}_{j+i} \rangle \tag{1}$$

equals m exactly. Furthermore, when $v_{j+1} = m - \ell$, then we have exactly ℓ mismatches.

The direct computation of the above sums would require $O(nm)$ scalar products where each one may take $O(|\Sigma|)$ time to compute. In the next section we show how one can speed up this algorithm for the MSS problem using a convolution-type argument.

3. PMCC with Karatsuba's fast multiplication algorithm

In this section we develop an algorithm to solve the PMCC problem based on Karatsuba's fast multiplication (Karatsuba and Ofman [6]). Recall that Karatsuba's algorithm requires

$$O\left(m^{\log_2 3}\right)$$

single digit multiplications and as many additions to multiply two polynomials of degree $m-1$. Now if we want to multiply two polynomials f and g of degrees n and $m = n/q$ respectively we first split f into segments $f_1, \ldots, f_q$ of length m, then carry out all multiplications $f_i g$ and finally add up the results using the corresponding place values for all results. In other words, we compute

$$f \cdot g = \sum_{i=0}^{q-1} (f_i \cdot g) x^{im} .$$

We shall see (Expression 6) that for a constant number of different character classes this requires altogether roughly $O\left(nm^{0.585}\right)$ time.

In conclusion, we carry out $q = n/m$ polynomial multiplications and suitable combine the results into our final answer. Therefore, at the heart of our application we face the following problem:

We are given two strings of equal length m consisting of binary vectors of dimension $|\Sigma|$ where each vector in the first sequence has exactly one non-zero element. We want to compute the "product" of the two strings in such a way that for each position j the result is exactly v_j as defined above. Note that this actually corresponds to solving an extension of exact search, since the v_j's now yield as a byproduct also the number of possible mismatches in correspondence with each alignment. For the simplicity of this discussion it is convenient to assume that the length of the strings is a power of 2. This does not affect generality since any string can be padded suitably with zeroes. We remark in passing that we could extend our approach by allowing character classes in the text as well (that is, several 1s in the corresponding characteristic vectors). Leaving such a generalization for an exercise, we will now establish the following fact:

Theorem 1. *Under suitable bounds on the number of distinct character classes, numerical values and position-specific thresholds, the problem of modulated string searching (with possible mismatches) can be solved by an adaptation of Karatsuba's multiplication algorithm in time* $O\left(nm^{0.585}\right)$.

A detailed analysis of complexity will follow after Equation (6).

The proof requires us to revisit Karatsuba's algorithm carefully. Recall that this algorithm is based on the following trick originally invented by Gauss for multiplying complex numbers: Let f, g be two polynomials of degree $2k-1$, let a, b, c and d be polynomials of degree $k-1$ and set $f = ax^k + b$ and $g = cx^k + d$. Then

$$(ax^k + b)(cx^k + d) = ac \cdot x^{2k} + [(a+b)(c+d) - ac - bd] \cdot x^k + bd \tag{2}$$

The algorithm computes all products recursively. Figure 1 displays the basic recursion. Its control structure borrowed from the pseudocode in Weimerskirch and Paar [10] is reproduced here for the convenience of the reader.

Algorithm KAM $\quad z = KAM(f, g)$
Input: `Polynomials` $f(x), g(x)$; $\quad 2k = degree(f) + 1 = degree(g) + 1$.
Output: $\quad z(x) = f(x) \times g(x)$
if $2k = 1$ **return** $f \times g$
set $\quad f(x) = a(x)x^k + b(x)$; $\quad g(x) = c(x)x^k + d(x)$
create $r_1(x) = a(x) + b(x), \quad r_2(x) = c(x) + d(x)$
$t_1 \leftarrow$ **KAM**(a, c)
$t_2 \leftarrow$ **KAM**(b, d)
$t_3 \leftarrow$ **KAM**(r_1, r_2)
return $t_1 x^{2k} + (t_3 - t_1 - t_2)x^k + t_2$

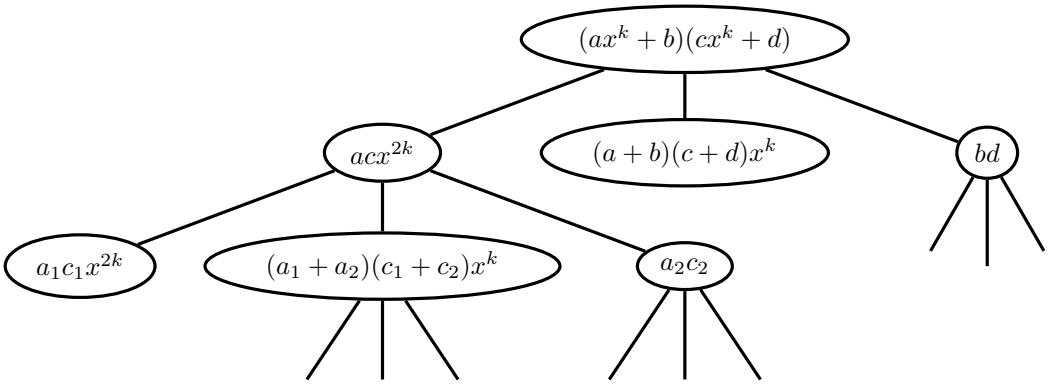
Thus, for suitable constants γ and δ the number of elementary operations performed by the algorithm is governed by the recurrence

$$T(2k) = 3T(k) + 2\gamma k + \delta \tag{3}$$

for which the Master Theorem gives the asymptotic bound $T(k) = \Theta(k^{\log_2 3})$. More specifically, the recursive procedure requires altogether $k^{\log_2 3}$ multiplications and not more than $6k^{\log_2 3} - 8k + 2$ elementary additions and subtractions (see [10]).

The main idea behind our proposed nested procedure is to substitute the regular multiplications (over the underlying number domain) performed in the leaves of our recursion tree with some symbolic computation

Figure 1: The structure of Karatsuba's algorithm.



over a not necessarily commutative ring structure $\mathcal{R}$ over the ring $\mathbb{Z}$ of integers. Strictly speaking, this symbolic computation constitutes our plug-in procedure and provides the flexibility of our proposed approach. However, once the Karatsuba algorithm is extended in this way to work on the polynomials in $\mathcal{R}[x]$, the complexity analysis above does no longer apply automatically. We thus need to take a closer look at the original algorithm's mechanics and complexity.

Observe that *before* issuing the recursive calls, the procedure needs only to perform two additions of polynomials. When control *returns* from the recursion, it needs to perform 4 such operations (2 additions and 2 subtractions), whence a total of 6. The mechanism of the original Karatsuba algorithm within the recursive calls can be subdivided into three phases:

Phase 1 Proceeding top-down, the procedure computes the items at each branching point and (finally) at the leaves of the recursion tree. At each of the 3^h vertices at level h it performs additions of two polynomials of length $k/2^h$, each requiring $k/2^h$ elementary additions over the ring $\mathcal{R}$. So the actual time complexity of these operations in our generalized Karatsuba algorithm will depend on the complexity of the coefficients from $\mathcal{R}$ and their representations.

Phase 2 At each of the $k^{\log_2 3}$ leaves, the procedure performs the required pairwise product between monomial coefficients. (Again this cost depends on the actual formal ring.)

Phase 3 Proceeding now bottom-up, the rocedure computes the actual polynomial values in correspondence with each branching vertex. This is done by shifting the values of the three children by 0, ℓ and 2ℓ positions accordingly, and performing two addition and two subtraction polynomials of length 4ℓ.

In Phase 1 the original algorithm performs roughly $2k^{\log_2 3}$ elementary additions (over the underlying number domain) while in Phase 2 there are $k^{\log_2 3}$ elementary pairwise products and additions. Finally, in Phase 3 roughly $4k^{\log_2 3}$ elementary additions of $\mathcal{R}[x]$-elements are performed.

We give next a formal description of the objects of our algorithm: Let the set Ω consist of one symbol for each subset of Σ that appears anywhere in the pattern. Furthermore let $\Gamma := \Sigma \cup \Omega$. In what follows, the generic elements of these sets are denoted by γ, σ and ω and we adopt these symbols as the appropriate characteristic vectors of the subsets. When, like in the case of locally bounded L_1-distance problem, the pattern positions contain their private local bounds as well, then we have ω's representing the same subset

but containing different private local bounds. Next we consider the free ring $\mathcal{R} = \mathbb{Z}\Gamma$ with generators Γ over the integers[4].

Recall that we are representing any pair formed by the pattern P and a corresponding segment T' of length m from our text T by the polynomials

$$T'(x) \in \mathbb{Z}\Gamma[X], \qquad T'(x) = \sum_{i=1}^{m} \mu_i x^{m-i}, \tag{4}$$

and

$$P(x) \in \mathbb{Z}\Gamma[X], \qquad P(x) = \sum_{i=1}^{m} \nu_i x^{i-1}; \tag{5}$$

where each $\mu_i \in \Sigma$ and $\nu_i \in \Omega$. In Phase 1, in the "middle" child of each vertex, we need to evaluate the sum of two polynomials. The coefficients of these polynomials are elements of the free ring, that is, they are formal linear combinations of the generators since, except at the leaves, along the algorithm no (symbolic) multiplication takes place among ring elements. Furthermore these combinations never consist of more than $|\Sigma|$ generators in the left polynomials (coming from the text) and more than $|\Omega|$ generators in the right polynomials (coming from the pattern).

For the sake of our argument we perform these symbolic summations by representing a general element γ of the free ring by a *formal characteristic* vector $v(\gamma)$: this contains the (integer) coefficients of the generators (and there are $|\Sigma|$ formal generators in the left polynomial and $|\Omega|$ formal generators in the right polynomials). To perform the addition of two general elements we add the corresponding characteristic vectors component-wise. Therefore the complexity of such a formal summation is $O(|\Gamma|)$ elementary additions over the integers. Therefore, the overall complexity of our "generalized" Phase 1 would be $O(|\Gamma| \cdot m^{\log_2 3})$ elementary operations over the integers. Fortunately, as will be seen shortly, we can organize this step much more efficiently.

In our Phase 2, we must compute at each leaf the product of two general integer elements of the free ring. This amounts to computing the pairwise products, each accompanied by the product of the two integer coefficients. At each leaf there are at most $|\Sigma||\Omega|$ such pairs.

Instead of evaluating these standard products, we apply a map Ψ from the products of any two generators into the ring $\mathbb{Z}$ of integers. We take, as an example, the case where a fit happens when the text character belong to the subset in the pattern. Then each double product $\sigma\omega$ maps to 1 iff the text symbol matches to the pattern element and 0 otherwise. This is exactly Equation (1), just the scalar product of the corresponding $\bar{t}$ and $\bar{p}$ characteristic vectors. Then we extend this map to the product of two general elements in the usual way: the Ψ-image of each double product will be accompanied by the product (over the integers) of the two coefficients. (If the fitting conditions are different, then the definition -and the computing method- of the pair-wise product of one element from Σ and one from Ω will differ accordingly.)

In fact, it is easy to see that this is a group homomorphism from the additive group of the free ring to the additive group of $\mathbb{Z}$. In this way, each coefficient in the final product, which is a linear combination of double products, is mapped into $\mathbb{Z}$.

We remark that by the distributive and commutative laws for integer numbers the formal linear combination of generator elements and the double products of the generators at the bottom are fully interchangeable. Therefore, in Phase I, instead of using formal linear combinations of generating elements, we can perform the standard linear combinations of vectors of dimension $|\Sigma|$ and $|\Omega|$ over $\mathbb{Z}$. Thus, in each step of Phase I,

[4]As is well known, the elements of this ring are formal linear combinations of finite words over Γ with coefficients in $\mathbb{Z}$. The additive group is commutative. The multiplication of two finite words is the concatenation of the words and hence is non-commutative. This multiplication extends distributively to the ring. The multiplication by scalars is distributive.

we have to calculate the linear combination of two (integer) characteristic vectors of dimension $O(|\Gamma|)$ and store the result as a new integer vector over $\mathbb{Z}$.

In conclusion, instead of introducing the formal characteristic vectors $v(\gamma)$ we just keep the original representations of our text symbols and pattern elements and treat them as integer vectors. Therefore, we need $O(|\Gamma|)$ space to store the current polynomials at each step in Phase I, and the time complexity of Phase I is altogether $O(|\Gamma|m^{\log_2 3})$.

Clearly, a perfect match occurs if and only if the coefficient equals m. On the other hand, if this coefficient is $m' < m$ then there are exactly $m - m'$ mismatches between T' and P.

In Phase 2 of KAM we perform $O(m^{\log_2 3})$ pairwise multiplications between elements of the free ring. In our case the pairwise multiplication is simply the scalar product of two characteristic vectors of dimension $|\Sigma|$, whence our Phase 2 charges $O\left(|\Sigma||\Omega||\Sigma|m^{\log_2 3}\right)$ multiplications over $\mathbb{Z}$ and the same number of additions overall.

In Phase 3 we compute the Ψ-image of every addition instead of the additions of general elements of the free ring. In all such steps we have thus just integers as factors. Therefore, Phase 3 has exactly the same complexity as in the original Karatsuba algorithm, that is, $O(m^{\log_2 3})$.

We can conclude that the overall product of $T'(x)$ and $P(x)$ involves no more than $3rm^{\log_2 3} + 4m^{\log_2 3}$ additions and $O(|\Sigma||\Omega|m^{\log_2 3})$ pairwise products. Running the algorithm for all consecutive non-overlapping segments of the text and putting together the resulting product polynomials will complete the procedure, resulting in

$$O\left(\frac{n}{m}|\Sigma|^2|\Omega|m^{\log_2 3}\right) \;=\; O\left(n|\Sigma|^2|\Omega|m^{\log_2 3-1}\right) \;=\; O\left(n|\Sigma|^2|\Omega|m^{0.585}\right). \tag{6}$$

In cases when $|\Omega|$ is not too big (say it remains constant while n becomes longer) then this term is $O\left(nm^{0.585}\right)$. It is also possible to calculate all possible fitting scores in advance: taking all possible text characters against all possible pattern character classes (including the local numerical values), calculating and storing all occurring values in a corresponding two dimensional array. Then, at the cost of $|\Sigma|^2|\Omega|$ extra space we can reduce the time complexity from $O\left(n|\Sigma|^2|\Omega|m^{0.585}\right)$ to $O\left(n|\Sigma|m^{0.585} + |\Sigma|^2|\Omega|\right)$ which is again only $O\left(nm^{0.585}\right)$.

This concludes the discussion of our claim. □

4. The plug-in procedure

In this section we detail two additional plug-in procedures in order to exemplify the variety of incarnations of modulated string searching. Consider first a simple solution for the τ-truncated L_1-distance string matching on character classes in which we assume an infinite value for the constant b (see the detailed description at the beginning of Section 2). We examine the formal multiplications between elements of Σ and Ω, found at the bottom of the recursion tree. Their representative characteristic vectors were called t and p, respectively. Each text element (a characteristic vector) contains one character, while the pattern class may contain several characters. To calculate the "product" of these characteristic vectors, one should find the two characters in the pattern class which are closest to the text character (using, for example, a standard merge), then calculate the smaller L_1-distance, and finally truncate it with the constant τ. Instead we can check the closed τ-ball around the text element to see whether it contains elements of the pattern class and, if the answer is affirmative, perform the necessary calculation. This requires no more than $2\tau + 1$ steps at each multiplication. The subsequent steps are obvious. The solution of the locally bounded L_1-distance string matching on character classes is similar.

Next, to demonstrate the method in a more complex context, we compute the L_1-distances at points where pattern and text meet modulated pattern matching conditions such as the above. For this we will

need to manage a second characteristic vector pair for text and pattern, respectively, which will store the actual text and pattern elements in form of symbolic linear combinations.

We revisit the multiplication of the largest linear combinations found at the bottom of the recursion tree. These were called t and p, respectively. They consist of two characteristic vectors storing $\min(m, |\Sigma|)$ elements from the text and $\min(m, |\Omega|)$ terms from the pattern. This time, our multiplication consists of computing the sum of the differences $(\ell_i - \ell_j)$ that can be formed by taking one symbol from t and one from p. Consider first pattern elements such that $\ell_i \geq \ell_j$. Letting r and r' be the number of symbols in p and t respectively, we assume the existence of index tables I and I' that take from the value of ℓ_h to h, respectively for $1 \leq h \leq r'$ and $1 \leq h \leq r$. We need the array S containing at the h-th position the value

$$S_{h-1} = \sum_{j=h}^{r} (\ell_j - \ell_{h-1}) f_j, \tag{7}$$

where f_j denotes the multiplicity of run length ℓ_j. Clearly,

$$(\ell_j - \ell_{j-1})(f_r + ... + f_j) = S_{j-1} - S_j$$

so that S can be filled in linear time using

$$S_{j-1} = S_j + (\ell_j - \ell_{j-1}) \times F_j$$

where $F_j = f_r + ... + f_j$ is obtained for all values of j by a single suffix computation on the frequencies. With the array S in place, the cumulative distance Δ of a text run length ℓ from the pattern runs is computed as follows. Let $\ell_{j-1} \leq \ell < \ell_j$. Then,

$$\Delta = S_j + (\ell_j - \ell) \times F_j.$$

We deal with the cases $\ell_i \leq \ell_j$ analogously. The overall procedure results in no increase in the time complexity.

It is easy to formulate many additional variants of the problem. For instance, assume that we are still interested in the L_1-distance between the pattern and the text at each possible starting position. However, we require now in addition that at each position the text element must fall within a possibly varying, specified neighbourhood of the pattern element. For example, their difference must be never bigger than some *a priori* assigned value τ (the previous problem), or it must be always an even number, or, it must be even whenever the difference is at most h, odd otherwise. And so on.

Acknowledgement We are indebted to Amihood Amir for his encouragement and for enlightening discussions.

References

[1] Abrahamson, K.: Generalized String Matching. *SIAM J. on Computing* **16**(6), 1039-1051, 1987.

[2] Amir, A. - Farach, M.: Efficient 2-dimensional approximate matching of half-rectangular figures, *Inform. and Comput.* **118** (1995), 1–11.

[3] Amir, A. - Lipsky, O. - Porat, E. - Umanski, J.: Approximate Matching in the L_1 Metric, *CPM 2005* (A. Apostolico, M. Crochemore, and K. Park (Eds.)) **LNCS 3537** (2005), 91-103.

[4] Amir, A. - Landau, G. M. - Vishkin, U.: Efficient Pattern Matching with Scaling. *Journal of Algorithms* **13**, 1, 2–32 (1992)

[5] Crandall, R. - Pomerance, C.: *Prime Numbers A Computational Perspective* Second Edition, Springer, (2005), pp 473.

[6] Karatsuba, A. - Ofman, Y.: Multiplication of Multidigit Numbers on Automata. *Soviet Physics - Doklady*, **7**, 595–596 (1963)

[7] Linhart, C. - Shamir, R.: Faster pattern matching with character classes using prime number encoding, *Journal of Computer and System Sciences* **75** (2009), 155–162.

[8] Lipsky, O.: Efficient distance computations. *Masters thesis, Bar-Ilan University, Department of Computer Science* (2003).

[9] Schönhage, A.: Asymptotically fast algorithms for the numerical multiplication and division of polynomials with complex coefficients. In *Computer Algebra, EUROCAM 82* **LNCS 144** (1982), 3–15.

[10] Weimerskirch, A. - Paar, C.: Generalizations of the Karatsuba Algorithm for Efficient Implementations. `www.crypto.ruhr-uni-bochum.de` . Technical report, 1–17 (2003) Alternative address: `eprint.iacr.org/2006/224.ps`