

Parameterized matching with mismatches[☆]

Alberto Apostolico^{a,1}, Péter L. Erdős^{b,2}, Moshe Lewenstein^{c,*}

^a Department of Computer Sciences, Purdue University, West Lafayette, IN 47907, USA and Dipartimento di Ingegneria dell' Informazione, Università di Padova, Padova, Italy

^b A. Rényi Institute of Mathematics, Hungarian Academy of Sciences, Budapest, P.O. Box 127, H-1364 Hungary

^c Department of Computer Science, Bar-Ilan University, Ramat Gan 52900, Israel

Available online 3 May 2006

Abstract

The problem of approximate parameterized string searching consists of finding, for a given text $t = t_1 t_2 \dots t_n$ and pattern $p = p_1 p_2 \dots p_m$ over respective alphabets Σ_t and Σ_p , the injection π_i from Σ_p to Σ_t maximizing the number of matches between $\pi_i(p)$ and $t_i t_{i+1} \dots t_{i+m-1}$ ($i = 1, 2, \dots, n - m + 1$). We examine the special case where both strings are run-length encoded, and further restrict to the case where one of the alphabets is binary. For this case, we give a construction working in time $O(n + (r_p \times r_t) \alpha(r_t) \log(r_t))$, where r_p and r_t denote the number of runs in the corresponding encodings for y and x , respectively, and α is the inverse of the Ackermann's function.

© 2006 Elsevier B.V. All rights reserved.

Keywords: Algorithms; String matching; Pattern matching; Parameterized matching; Pattern matching with mismatches

1. Introduction

String searching is one of the basic primitives of computation, see [1]. In the standard formulation of the problem, we are given a pattern and a text and it is required to find all occurrences of the pattern in the text. Several variants of the problem have also been considered, e.g., allowing mismatches, insertions, deletions, swaps, etc. In the parameterized variant, a match exists at one position of the text if the alphabets of pattern and text can be consistently mapped into one another in such a way that all characters match pairwise. This variant was introduced and studied by B. Baker [2,3] and others, motivated by issues of program compaction in software engineering. In this paper, we study approximate variants of the problem where a (possibly controlled) number of mismatches is allowed.

Specifically, we seek to find, for given text $t = t_1 t_2 \dots t_n$ and pattern $p = p_1 p_2 \dots p_m$ over respective alphabets Σ_t and Σ_p , the injection π_i from Σ_p to Σ_t maximizing the number of matches between $\pi_i(p)$ and $t_i t_{i+1} \dots t_{i+m-1}$ ($i = 1, 2, \dots, n - m + 1$). We examine the special case where both strings are run-length encoded, and further restrict

[☆] Work performed in part while on leave at ZIF, University of Bielefeld, Germany, within framework of the Special Year on “General Theory of Information Transfer and Combinatorics”.

* Corresponding author.

E-mail addresses: axa@cs.purdue.edu (A. Apostolico), elp@renyi.hu (P.L. Erdős), moshe@cs.biu.ac.il (M. Lewenstein).

¹ Supported in part by an IBM Faculty Partnership Award, by the Italian Ministry of University and Research under the National Projects “Bioinformatics and Genomics Research” and FIRB RBNE01KNFP, and by the Research Program of the University of Padova.

² This work was supported in part by the Hungarian NSF, under contract Nos. T37846, T34702.

to the case where one of the alphabets is binary. For this case, we give a construction working in time $O(r_p \times r_t)$, where r_p and r_t denote the number of runs in the corresponding encodings for y and x , respectively.

This paper is organized as follows. In the next section, we give some basic definitions and properties, and derive the combinatorial facts used in our construction. Section 3 is devoted to the design and description of our algorithm. The last section lists conclusions and plans of future work.

2. Properties

Given a string $s = s_1s_2 \dots s_n$ of length $|s| = n$ over an alphabet Σ_1 , and an injection π from Σ_1 to some other alphabet Σ_2 , the image of s under π is the string $s' = \pi(s) = \pi(s_1) \dots \pi(s_n)$ that is obtained by applying π to all characters of s , where \cdot denotes concatenation.

Given two strings w and u of the same length over alphabets Σ_1 and Σ_2 and an injection π from Σ_1 to Σ_2 the π -match between w and u is the number of matches between $\pi(w)$ and u . The problem of *approximate parameterized matching* between w and u consists of finding a π of maximum π -match. For given input text x ($|x| = n$) and pattern y ($|y| = m$), the problem of *approximate parameterized searching (APS)* for y in x consists of computing the *approximate parameterized matching* between y and every (consecutive) substring of length m of x . Hence, approximate parameterized searching requires computing the π yielding maximum π -match for y at each position of x . Of course, the best π is not necessarily the same at every position.

The general problem of APS can be solved in $O(nm(\sqrt{m} + \log n))$ by reduction to bipartite graph matching (see, e.g., [6,7]): each relative alignment defines one such graph in which edges are weighted according to the number of matches induced by each character “hits”, and the problem is to choose a set of edges of maximum weight.

For fixed sized text and pattern alphabets there are a constant number of possible injections. Hence we can apply the convolution method [5] to try them out individually and then compare them point by point, in time $O(n \log n)$ overall. When $|\Sigma_p| = 1$, the problem has a trivial linear solution: slide over the text with a window of size equal to the pattern length m while keeping track of which text character scores the maximum number of matches. Following initialization at the first position, it is enough at every step to update the number of matches to check the text character leaving to face the pattern and the text character entering to face the pattern.

Such simplifications appear to fade as soon as either the text or the pattern alphabet alone can be arbitrary while the other is binary.

In this paper, we concentrate on special cases of APS where the assumption is made that t and p are presented in their respective run-length encodings, denoted $X = X_1X_2 \dots X_{r_t}$ and $Y = Y_1Y_2 \dots Y_{r_p}$, respectively. The generic run, say $X_k = t_i t_{i+1} \dots t_{i+l-1}$, corresponds to a maximal substring of consecutive occurrences of the same symbol, and it has an *interval* $l_k = [i, i + l - 1] = [L_{l_k}, R_{l_k}]$ naturally associated with it. For ease of notation we use the l_k 's to denote both the intervals and their respective lengths. Thus, the list of integers l_1, l_2, \dots, l_{r_t} specifies the run intervals for X . Clearly, knowing the first symbols and the run intervals for a binary string X entirely specifies that string. Moreover, note that the interval lengths can be derived from L_{l_k} , the left-end of the interval, and from $L_{l_{k+1}}$, the left-end of the following interval. Hence, another way of specifying the run intervals is simply by the *left-end list*, $L_{l_1}, L_{l_2}, \dots, L_{l_{r_p+1}}$ (or, $\dots, L_{l_{r_t+1}}$), where $L_{l_{r_p+1}} = |P| + 1$ (or, $L_{l_{r_t+1}} = |T| + 1$, respectively) is added for the sake of consistent computation. We also use Σ_t and Σ_p to denote the text and pattern alphabet, respectively.

3. Run-length algorithm

Our goal is to solve APS in time $O(r_p \times r_t)$, but we must be prepared to pay an additive term linear in n , the length of x , and perhaps some logarithmic overhead. In practice, one may expect less than one run of pattern and text to end up juxtaposed at any position of the pattern, whence $O(r_p \times r_t)$ would be sub-linear.

We consider first the easy variant where $|\Sigma_p| = |\Sigma_t| = 2$. Note that $O(n \times r_p)$ or $O(m \times r_t)$ is doable in this case. For example, initialize the matching corresponding to aligning the pattern beginning in the first position of the text. Clearly, the π -mismatch at this position results from trying out the two possible assignments for the zeroes and ones and choosing the best one. Next, following every unit shift of the pattern, for each one of the two competing injections scan the runs of the pattern and update the matches for each run. We show next that this scheme can be extended to run in time $O(r_p \times r_t)$. Towards this, we need to introduce additional notions.

Proof. The case $t_i = t_{i+1}$ does not require explanation. If $t_i \neq t_{i+1}$ and t_i matches the character of the run, then the number of the matches in the run decreases by 1. Otherwise, the number of the matches increases by 1. \square

Fact 1 supports a simple algorithm for this instance of APS. The algorithm works as follows.

We assign two counters to the task of keeping track of the π -match under either bijection $\pi : \Sigma_p \rightarrow \Sigma_t$. Specifically, c will tally the π -match under the *identity* $\pi(0_p) = 0_t$, and \bar{c} will do the same for the *switch* $\pi(0_p) = 1_t$, with transparent notation. To fix the ideas, we illustrate the operation with c , the one with \bar{c} being dual. At the beginning, the pattern is aligned with the text, left justified, and we compute the fusion and the values $c(1)$ by straightforward methods, in $O(m)$ time. As part of this computation, we also initialize the following items at $k = 1$.

- The number $S(k)$ of pattern runs of which the contribution to the match won't undergo a change upon unit shift.
- The number $DU(k)$ of pattern runs that will each witness a unit decrement in their contribution upon a single shift.
- The number $IU(k)$ of pattern runs that symmetrically experience unit increments upon shift.

The idea of the algorithm is that as long as there is no bump, we will be able to compute $c(k)$ at each location in constant time by the following observation.

Lemma 1. *Let $k + 1$ be a location without a bump then $c(k + 1) = c(k) + IU(k) - DU(k)$.*

Proof. This follows from the fact that $IU(k + 1) = IU(k)$ and $DU(k + 1) = DU(k)$. \square

Hence, within this regime the value of c can be computed in constant time per shift, whether the shift itself be unitary or multiple. In order to know the APS at every position of the text, we still need to compute the winning match $\text{Max}(c, \bar{c})$ between identity and switch. It is clear how to do that by comparing the two scores at each position of the text, at a cost of $O(n)$ operations. However, in view of the bi-modal behavior of $\text{Max}(c, \bar{c})$ in between bumps, this extra term is not needed here: just giving this value at bumps and the slope of the straight lines of c and \bar{c} implicitly encodes all intermediate values.

We now look at the management of bumps. Figuratively, an external left-end delimiter of a text run enters the fusion at the right end, and exits it from the left end. During this life cycle, the run to the right of this delimiter will in general alternate between *inactive* phases, characterized by the run being entirely contained in a wider pattern run, and *active* phases, that correspond to the run being broken by a pattern run delimiter. More formally, the run with left-end delimiter L_i in fusion $\mathcal{L} = (L_1, L_2, \dots, L_p)$ is inactive if its neighbor L_j has $j < i - 1$ and is active if $j = i - 1$. In the corresponding fusion, an inactive run is simply re-copied shift after shift, whereas an active run appears as broken, and its first and last segments undergo unit increases and decreases in size, respectively, at every shift. Note that there are only at most r_p active runs, hence r_p bumps awaiting at any given time. The corresponding sizes all vary in time, but updating them all at every step would charge an unacceptable $O(n \times r_p)$. Luckily, we do not need to, as we are only interested in the run responsible for the bump distance, and this does not change in between bumps.

Observe that, when measured relative to the origin of the pattern, the delimiters of pattern runs in the fusion do not change in time, only text run delimiters “travel backwards” inside the pattern run partition.

It remains to be said how it is determined that a bump has to take place. But this is always associated with a specific alignment of the pattern relative to the text, which we can compute as follows. There are $r_p \times r_t$ bumps that occur, each by an alignment of the text-delimiter with a pattern-delimiter. Say location i of the text has a text-delimiter immediately preceding it and location j of the pattern has a pattern-delimiter immediately preceding it. The bump happens when the pattern is aligned to begin at location $i - j + 1$, the *bump alignment*. So, to find all the bump alignments we go over all text-delimiters and pattern-delimiters and mark sentinels at their bump alignment in an n length array. We then traverse the array and have an ordered list of all the bump-alignments and, hence, we know exactly where all the bumps take place and can advance from one to the next with simplicity.

As part of the management of the bump, we also update $SU(k)$, $DU(k)$ and $IU(k)$, where k is a bump alignment, in an obvious way. The remaining details are easy and left for an exercise. As there are only $r_p \times r_t$ bumps, then clearly the time complexity of this algorithm is $O(r_p \times r_t + n)$.

Note that there is still an $O(n)$ term charged for finding the sentinels. An alternative to the above scheme is offered by the observation that when a text run issues a bump for the first time, it is possible to predict all of its subsequent r_p bumps, and these correspond to the sequence of pattern runs, in reverse order. Assuming all bumps caused by the same text run are kept in a list, every such run will require the insertion of $r_p + 1$ items in a list of currently issued runs containing at most m elements at any given time. Each insertion can be performed in $O(r_p \log \log(m/r_p))$ through resort to integer finger merging techniques, leading to an $O((r_p \times r_t) \log \log(m/r_p))$ time algorithm that uses $O(m)$ auxiliary space.

4. General alphabet algorithm

We now relax the condition of binary alphabet and consider the more general case where $\Sigma_p = \{0, 1\}$ while Σ_t is arbitrary.

The situation is now that at every given text location where the pattern is aligned the parameterized match maps the two pattern symbols to two (different!) text symbols such that the number of p-matches is maximized. This may happen in a couple of scenarios. Say 0 of the pattern maximally maps to $a \in \Sigma_t$ and 1 maximally maps to $b \in \Sigma_t$. Then, obviously, the best mapping is 0 to a and 1 to b . However, it may be the case that 0 maximally maps to a and 1 also maximally maps to a . Since we seek a p-match we require that they map to different symbols. So, in this case, the best mapping is defined by the second best maximizing map of the symbols. Namely, say 0 maps maximally to b (if the mapping to a is not taken into consideration) and 1 maps maximally to c (when the a mapping is dropped). Obviously, the best match is either 0 to a and 1 to c or 0 to b and 1 to a . (Note that b and c can be the same character.)

This seemingly requires us to count the mappings of each text symbol separately against 0 and 1 for both the maximal and second maximal values for every location of the text. This is exactly what we did in the previous section, but since the text was binary we could tuck this into the additive $O(n)$ complexity that we had anyway. In our current scenario where Σ_t can be large the cost of just saving all these values can be as large as $O(n|\Sigma_t|)$ which is more than we desire to pay. Hence we will need a finer technique to achieve an improved time complexity.

We start by reducing the text to a collection of binary strings, one for each character of the text. For each character σ , we create text t^σ which is obtained from t by applying the characteristic function f_σ on each character of t , specifically, $t_i^\sigma = 1$ if $t_i = \sigma$ and $t_i^\sigma = 0$ if $t_i \neq \sigma$. See the following example.

$$\begin{aligned} t &= \text{aaaaabcccaabbaadd} \\ t^a &= 1111100000111001100 \\ t^b &= 000011000000110000 \\ t^c &= 000000111000000000 \\ t^d &= 000000000000000011 \end{aligned}$$

Note that it follows from the above definition that we can use the t^σ 's instead of t in the following sense. Let i be a location of the text t and let p be aligned with t starting at i . Then the number of matches between $0 \in \Sigma_p$ (or, respectively, of $1 \in \Sigma_p$) and $\sigma \in \Sigma_t$ equals the number of matches between 0 (or, respectively, of $1 \in \Sigma_p$) and 1 of the text of t^σ in the alignment of p with t^σ . What makes this really neat is the following.

Lemma 2. *Let t be a text over an alphabet Σ_t with r_t runs. Let p be a binary pattern with r_p runs. Then*

$$\sum_{\sigma \in \Sigma_t} r_p \times r_{T^\sigma} \leq 2r_p \times r_t$$

Proof. Every text-delimiter in t (besides the ones at the extreme ends) appears in exactly $2r^\sigma$ strings. \square

This immediately gives a method which will run in $O(r_p \times r_t + n|\Sigma_t|)$ as we can apply the method of the previous section to each t^σ separately (for an $O(r_p \times r_t + n|\Sigma_t|)$ time) and compute the best p-match at each location by the above-mentioned discussion in $O(n)$ time. Our current goal is to reduce the $O(n|\Sigma_t|)$ term to $O(n)$ in order to achieve the same time bounds as in the previous section.

We define $S(k, \sigma)$, $DU(k, \sigma)$ and $IU(k, \sigma)$ to be the values $S(k)$, $DU(k)$ and $IU(k)$ for function c_σ which computes the mapping count for $\pi(0_p) = \sigma$.

Lemma 3. *Let k_1, k_2, \dots, k_h be the locations with bumps in the fusion between p and t^σ . Then for $k_i \leq k < k_{i+1}$, $S(k, \sigma) = S(k_i, \sigma) + (k - k_i)(DU(k_i, \sigma) + IU(k_i, \sigma))$.*

Proof. Follows from Lemma 1. \square

Hence if we can compute the values $S(k, \sigma)$ at all the bumps of p and t^σ the rest can be computed directly from them. In fact, we can compute these values $S(k, \sigma)$ at all the bumps in $O(r_p \times r_t)$ time by using the management of the bumps from the previous section. However, while the computation of the other values is now easy the number of values to be computed still has the unacceptable size $O(n|\Sigma_t|)$.

Nevertheless, we can conclude that the values of $S(k, \sigma)$ between bumps k_i and $k_{i+1} - 1$ are defined by a linear segment, $I_{k_i, k_{i+1}, \sigma}$, on the reals which has endpoints $S(k_i, \sigma)$ and $S(k_i, \sigma) + ((k_{i+1} - 1) - k_i)(DU(k_i, \sigma) + IU(k_i, \sigma))$ and are defined by the linear function $h(x) = S(k_i, \sigma) + (x - k_i)(DU(k_i, \sigma) + IU(k_i, \sigma))$.

Note that $r_p \times r_t^\sigma$ segments define the value of the matches between 0_p and σ over all locations of the text. Combining this with Lemma 2 we have that $O(r_p \times r_t)$ linear segments define the match count between 0_p and each of the $|\Sigma_t|$ symbols of the text.

This converts our problem into a geometric problem: we are given a set of $O(r_t \times r_p)$ linear segments and wish to find their upper envelope. This problem is solved for k segments in time $O(k\alpha(k) \log k)$ and $O(k\alpha(k))$ storage [4], where $\alpha(k)$ is the (extremely slowly growing) inverse of the Ackermann's function. To be specific, we have two systems of linear segments, where one system lists the hits of all characters of Σ_t against character $0 \in \Sigma_p$, while the other similarly lists hits against $1 \in \Sigma_p$. At any text position, the optimum APS consists of the maximum achievable by taking the hits scored by two distinct text characters at that position. This is clearly the sum of the maxima in the two systems if the characters producing these maxima are different. If, on the other hand, these are the same text character, then we need to know the characters producing the maximum and second maximum at that position. Trying both combinations will produce the winning bijection. This can be obtained, e.g., by peeling the segments atop the upper envelope and then recomputing the latter. Note that this operation may result in a set of segments that constitutes substantial alteration of the original set, however, the number of segments at the outset is still linear in the original number.

The above arguments adapt to the case of binary Σ_t and arbitrary Σ_p . This leads to the following

Fact 2. There is an $O(n + (r_p \times r_t)\alpha(r_t) \log(r_t))$ algorithm for computing parameterized matching with either Σ_t or Σ_p is binary and the other one is arbitrary.

References

- [1] A. Apostolico, Z. Galil (Eds.), Pattern Matching Algorithms, Oxford University Press, New York, 1997.
- [2] B.S. Baker, Parameterized duplication in strings: Algorithms and an application to software maintenance, SIAM Journal of Computing 26 (5) (1997) 1343–1362.
- [3] B.S. Baker, Parameterized pattern matching: Algorithms and applications, Journal Computer System Science 52 (1) (1996) 28–42.
- [4] H. Edelsbrunner, L.J. Guibas, M. Sharir, The upper envelope of piecewise linear functions: Algorithms and applications, Discrete Computational Geometry 4 (1989) 311–336.
- [5] M. Fischer, M. Paterson, String matching and other products, in: R. Karp (Ed.), Complexity of Computation, SIAM–AMS Proceedings, 1974, pp. 113–125.
- [6] M.L. Fredman, R.E. Tarjan, Fibonacci heaps and their use in improved network optimizations algorithms, Journal of the ACM 34 (3) (1987) 596–615.
- [7] M.Y. Kao, T.W. Lam, W.K. Sung, H.F. Ting, A decomposition theorem for maximum weight bipartite matching, SIAM Journal of Computing 31 (1) (2001) 18–26.