Information Processing Letters 111 (2011) 318–322

Contents lists available at ScienceDirect

Information Processing Letters

www.elsevier.com/locate/ipl

The indexing for one-dimensional proportionally-scaled strings ☆

Yung-Hsing Peng, Chang-Biau Yang *, Chiou-Ting Tseng, Chiou-Yi Hor

Department of Computer Science and Engineering, National Sun Yat-sen University, Kaohsiung, Taiwan

ARTICLE INFO

Article history:
Received 7 February 2009
Received in revised form 21 November 2010
Accepted 1 December 2010
Available online 21 December 2010
Communicated by F.Y.L. Chin

Keywords:
Design of algorithms
String matching
Scale
Proportional

ABSTRACT

Related problems of scaled matching and indexing, which aim to determine all positions in a text T that a pattern P occurs in its scaled form, are considered important because of their applications to computer vision. However, previous results only focus on enlarged patterns, and do not allow shrunk patterns since they may disappear. In this paper, we give the definition and an efficient indexing algorithm for proportionally-scaled patterns that can be visually enlarged or shrunk. The proposed indexing algorithm takes $O(|T|)$ time in its preprocessing phase, and achieves $O(|P| + U_p + \log m)$ time in its answering phase, where $|T|$, $|P|$, U_p, and m denote the length of T, the length of P, the number of reported positions, and the length of T under run-length representation, respectively.

1. Introduction

In the field of string processing, *exact string matching* is a classical problem which asks for all positions of a pattern string P in a text string T. When both P and T are one-dimensional strings, this problem can be solved in $O(|T|+|P|)$ time with the well-known Knuth–Morris–Pratt algorithm [12], where $|T|$ and $|P|$ denote the length of T and P, respectively.

Aside from the exact string matching problem, the *exact string indexing problem* asks one to preprocess T, so that the positions of P in T can be determined more efficiently. That is, T can be thought of as a database whereas P is the target string. Therefore, the performance of an indexing algorithm can be measured by its *preprocessing phase* with T and *answering phase* with P. For fixed alphabets, related techniques of suffix trees [10,16] and suffix arrays [11] can achieve both the optimal preprocessing time $O(|T|)$ and answering time $O(|P| + U)$, denoted as $\langle O(|T|), O(|P| + U)\rangle$, where U represents the number of reported positions.

Problems of *inexact matching* [2,3,6,8] have also drawn much attention recently. Among them, related problems that involve matching [2–5] or indexing [14,15,17] scaled patterns are considered not only interesting, but also realistic in the field of computer vision. One should note that, however, these algorithms [2–5,14,15,17] only discuss enlarged patterns but avoid shrunk ones. As mentioned in the previous literature [2], this is because the pattern may disappear, which would cause a scaled match at every position in T.

Nonetheless, from the perspectives of computer vision and algorithm, it is still worth studying the effect of a shrunk pattern, even if some presumptions must be made to prevent the pattern from disappearance. Therefore, we refer to Eilam-Tzoreff and Vishkin's multiplying transformation [9], which is known as the first matching problem that involves scaling. With a slight modification to their model, we define the *proportionally-scaled pattern*, which could be enlarged or shrunk, but never disappears. Also, with our modification, a proportionally-scaled pattern, which is different from those derived in the past [2–4,9], is natural (visually proportional) to human eyes. In this paper, we propose an efficient algorithm for indexing proportionally-scaled patterns. To the authors' knowledge, this is the first indexing algorithm for both enlarged and shrunk patterns.

☆ This research work was partially supported by the National Science Council of Taiwan under contract NSC-97-2221-E-110-064.

* Corresponding author.
E-mail address: cbyang@cse.nsysu.edu.tw (C.-B. Yang).

0020-0190/$ – see front matter
doi:10.1016/j.ipl.2010.12.001

The rest of this paper is organized as follows. In Section 2, we give an overview for required techniques. In Section 3, we give our definition for a proportionally-scaled pattern, and then explain a simple linear time matching algorithm adapted from previous results [3,9]. After that, we propose our indexing algorithm in Section 4. Finally, in Section 5 we give two interesting problems for future study.

2. Required techniques

For any string S, let $S[i]$ denote the ith character in S, and $S[i, j]$ denote the substring ranging from $S[i]$ to $S[j]$, for $1 \leqslant i \leqslant j \leqslant |S|$, where $|S|$ denotes the length of S. Assume $T' = t_1^{r_1} t_2^{r_2} \dots t_m^{r_m}$ be the run-length representation of T with $|T'| = m$, where $t_i \in \Sigma$, for $1 \leqslant i \leqslant m$, $t_j \neq t_{j+1}$, $1 \leqslant j \leqslant m-1$ and r_i denotes the run length of t_i. Therefore, one can easily map each character $T'[i]$ to the run $T[\sum_{j=1}^{i-1} r_j + 1, \sum_{j=1}^{i} r_j]$ in T. Also, let $P' = p_1^{s_1} p_2^{s_2} \dots p_u^{s_u}$ be the run-length represented string of P with $|P'| = u$. In the following, we briefly describe the required techniques.

2.1. Suffix arrays for integer alphabets

The suffix array T_A of T is an $O(|T|)$-space data structure that stores each suffix of T according to their lexical order. With additional information for the longest common prefixes, to search a given string P in T, one can perform a binary search on T_A, which achieves the answering time $O(|P| + U + \log|T|)$ [13]. The required time for constructing T_A is equal to that for constructing the suffix tree of T [10,16]. For constant-sized alphabets, T_A can be constructed in $O(|T|)$ time [16]. For integer alphabets, Farach-Colton et al. [10] first proposed the following result.

Theorem 1. *(See [10].) Given a string T over $\{1, 2, \dots, |T|\}$, the suffix array T_A of T can be constructed in $O(|T|)$ time.*

Based on Theorem 1, one can construct the suffix array of T in $O(|T| + Sort)$ time, where *Sort* denotes the required time to transform T into a string over $\{1, 2, \dots, |T|\}$. Therefore, for unbounded alphabets, it takes $\Omega(|T| \log |T|)$ time to construct the suffix array with existing sorting algorithms. Note that it is not necessary to transform the suffix array over $\{1, 2, \dots, |T|\}$ back into Σ, since the lexical order still holds.

2.2. The range minimum query and the three-sided query

Given an array A of n numbers, the *range minimum query* (*RMQ*) asks for the minimum element in the subarray $A[i_1, i_2]$, for any given interval $1 \leqslant i_1 \leqslant i_2 \leqslant n$. Bender and Farach-Colton [7] gave an elegant algorithm for preprocessing A, so that each RMQ can be answered in constant time. Let $RMQ_A(i_1, i_2)$ be the index of the minimum element in the subarray $A[i_1, i_2]$. We summarize their result as follows.

Theorem 2. *(See [7].) Given an array A of n numbers, one can preprocess A in $O(n)$ time such that for any given interval $[i_1, i_2]$, one can determine $RMQ_A(i_1, i_2)$ in $O(1)$ time.*

Applying Theorem 2 recursively, one can easily verify the following lemma, which also summarizes the three-sided query [1].

Lemma 1. *(See [1].) Given an array A of n numbers and a threshold c, one can preprocess A in $O(n)$ time, so that for any given interval $[i_1, i_2]$, it takes $O(U_x)$ time to report all indices $i_1 \leqslant i' \leqslant i_2$ satisfying $A[i'] \leqslant c$, where U_x is the number of reported indices.*

3. Matching proportionally-scaled patterns

In this section, we give our definition for a proportionally-scaled pattern, which is a modification to previous results [3,9]. In addition, to provide a better understanding to Section 4, we explain a simple linear time matching algorithm adapted from Eilam-Tzoreff and Vishkin's algorithm [3,9].

3.1. Definition

Recall that in Eilam-Tzoreff and Vishkin's scaling model [9], each character is a real number. Therefore, the α-scaling of a character r is written as αr, where α is also a real number. Taking $T = (3.8)(2.55)(3.3)(8.1)(3.45)(5.6)$ and $P = (1.7)(2.2)(5.4)(2.3)$ for example, $T[2, 5]$ is the (1.5)-scaling of P. However, one should note that this scheme is not the case for matching run-length represented strings. Taking $T' = b^4c^2a^5b^9$ and $P' = b^3c^2a^5b^5$ for example, P' still matches with T', even though there does not exist any α-scaling of $(3)(2)(5)(5)$ that equals to $(4)(2)(5)(9)$. In the following, we explain how to modify Eilam-Tzoreff and Vishkin's model, obtaining the definition for proportionally-scaled patterns.

For clarity, we begin with the *scaling function* defined by Amir et al. [2]. Given $P = p_1^{s_1} p_2^{s_2} \dots p_u^{s_u}$, the α-scaling of P, denoted by $\delta_\alpha(P)$, for any real number $\alpha > 0$, represents the string $p_1^{\lfloor \alpha s_1 \rfloor} p_2^{\lfloor \alpha s_2 \rfloor} \dots p_u^{\lfloor \alpha s_u \rfloor}$ [2]. As an example, suppose we have $T = a^2b^4c^2a^5b^9c^4a^4b^4$ and $P = a^3b^6c^3a^3$. In this case, one can verify that $\delta_{\frac{3}{2}}(P) = a^4b^9c^4a^4$ is a substring of T, but P is not. To apply the scale $0 < \alpha < 1$ to the same example, however, one can see that for $\frac{1}{6} \leqslant \alpha < \frac{1}{3}$, $\delta_\alpha(P)$ has only one symbol "b". In addition, the pattern disappears for any $\alpha < \frac{1}{6}$. To avoid the problem of symbol disappearance, which causes invalid matches, we give a new definition of a proportionally-scaled pattern of P as follows.

Definition 1. Given

$$P = p_1^{s_1} p_2^{s_2} \dots p_u^{s_u}, \quad \delta_\alpha(P) = p_1^{\lceil \alpha s_1 \rceil} p_2^{\lceil \alpha s_2 \rceil} \dots p_u^{\lceil \alpha s_u \rceil}$$

is a proportionally-scaled pattern of P if

(1) $\alpha > 0$,
(2) $\frac{s_{j+1}}{s_j} = \frac{\lceil \alpha s_{j+1} \rceil}{\lceil \alpha s_j \rceil}$, for $2 \leqslant j \leqslant u-2$, and
(3) $\frac{s_2}{s_1} \geqslant \frac{\lceil \alpha s_2 \rceil}{\lceil \alpha s_1 \rceil}$ and $\frac{s_u}{s_{u-1}} \leqslant \frac{\lceil \alpha s_u \rceil}{\lceil \alpha s_{u-1} \rceil}$.

Based on Definition 1, it is clear that a scaled pattern never disappears. In addition, $\delta_\alpha(P)$ could be a shrunk pattern or an enlarged one. Considering the previous example $T = a^2b^4c^2a^5b^9c^4a^4b^4$ and $P = a^3b^6c^3a^3$, one can verify that both $\delta_{\frac{2}{3}}(P) = a^2b^4c^2a^2$ and $\delta_{\frac{5}{3}}(P) = a^5b^{10}c^5a^5$ are proportionally-scaled patterns of P, where $\delta_{\frac{2}{3}}(P)$ is also a shrunk pattern in T. Taking $\delta_{\frac{3}{2}}(P) = a^5b^9c^5a^5$ as another example, it is not a proportionally-scaled pattern of P, because it breaks the second condition.

In Definition 1, we set the constraints $\frac{s_2}{s_1} \geqslant \frac{\lceil \alpha s_2 \rceil}{\lceil \alpha s_1 \rceil}$ and $\frac{s_u}{s_{u-1}} \leqslant \frac{\lceil \alpha s_u \rceil}{\lceil \alpha s_{u-1} \rceil}$. Therefore, a minor distortion with no more than two characters is allowed on $\lceil \alpha s_1 \rceil$ and $\lceil \alpha s_u \rceil$, by which we increase the flexibility for matching proportionally-scaled patterns. This is motivated by the result of Amir et al. [4], where each character (pixel) is treated like an interval. For example, given $T = a^2b^1c^1b^1a^4$ and $P = a^3b^2c^2b^2a^5$, one can see that $\delta_{\frac{1}{2}}(P) = a^2b^1c^1b^1a^3$ is also a proportionally-scaled pattern in T, because the pattern $a^{1.5}b^1c^1b^1a^{2.5}$ is visually existent in T. In this example, $a^{1.5}$ is an interval of length 1.5 and therefore a suffix of a^2.

Note that in Eilam-Tzoreff and Vishkin's scaling model [9], they adopt the constraints $\frac{s_2}{s_1} = \frac{\lceil \alpha s_2 \rceil}{\lceil \alpha s_1 \rceil}$ and $\frac{s_u}{s_{u-1}} = \frac{\lceil \alpha s_u \rceil}{\lceil \alpha s_{u-1} \rceil}$ (see the third condition in Definition 1).

3.2. A simple linear time matching algorithm

For completeness, in the following we briefly explain a linear time matching algorithm for proportionally-scaled patterns. Given $T' = t_1^{r_1}t_2^{r_2}\ldots t_m^{r_m}$ and $P' = p_1^{s_1}p_2^{s_2}\ldots p_u^{s_u}$, the *quotient strings* [3,9] of T' and P' can be written as $\hat{T}' = t_2^{r_2/r_1}t_3^{r_3/r_2}\ldots t_m^{r_m/r_{m-1}}$ and $\hat{P}' = p_2^{s_2/s_1}p_3^{s_3/s_2}\ldots p_u^{s_u/s_{u-1}}$, respectively. Intuitively, the following lemma holds.

Lemma 2. *A proportionally-scaled pattern $\delta_\alpha(P)$ occurs at the ith position of T' iff*

(1) $\hat{P}'[2, u-1] = \hat{T}'[i+1, i+u-2]$,
(2) $p_1p_2 = t_it_{i+1}$ *and* $\frac{s_2}{s_1} \geqslant \frac{r_{i+1}}{r_i}$, *and*
(3) $p_u = t_{i+u-1}$ *and* $\frac{s_u}{s_{u-1}} \leqslant \frac{r_{i+u-1}}{r_{i+u-2}}$.

Lemma 2 can be realized as an extension to Eilam-Tzoreff and Vishkin's matching rule [9], in which they apply $\frac{s_2}{s_1} = \frac{r_{i+1}}{r_i}$ and $\frac{s_u}{s_{u-1}} = \frac{r_{i+u-1}}{r_{i+u-2}}$ (see the second and the third conditions in Lemma 2). According to Lemma 2, one can use any linear time string matching algorithm to check the first condition. Then, it takes $O(1)$ time to check the remaining two conditions for each position i in T that satisfies the first condition. Therefore, for $|P'| \geqslant 3$ we have an $O(|T| + |P|)$-time matching algorithm for finding each proportionally-scaled P in T'. For each position i in T' where $\delta_\alpha(P)$ occurs, its corresponding position in T is $(\sum_{j=1}^{i} r_j) - \lceil \alpha s_1 \rceil + 1 = (\sum_{j=1}^{i} r_j) - \lceil \frac{r_{i+1}s_1}{s_2} \rceil + 1$, which can be determined in $O(1)$ time with an $O(|T|)$-time preprocessing on T.

To end this section, we briefly discuss the remaining cases for $|P'| \leqslant 2$. For $|P'| = 1$, it is clear that $p_1^{\lceil \alpha s_1 \rceil}$ occurs at every position i that $T[i] = p_1$, for some $\alpha > 0$. For $|P'| = 2$, one can locate every position i in T' that satisfies $t_it_{i+1} = p_1p_2$, then report consecutive positions $(\sum_{j=1}^{i} r_j) - min(r_i, \lceil \frac{r_{i+1}}{\beta} \rceil) + 1$ to $\sum_{j=1}^{i} r_j$, where $\beta = \frac{s_2}{s_1}$. Hence, we have an $O(|T| + |P|)$-time algorithm for matching proportionally-scaled patterns.

4. Indexing proportionally-scaled patterns

Briefly, our indexing approach is to construct the suffix array of $\hat{T}'$. Though this approach is intuitive, note that the ith character in $\hat{T}'$ is of the form $t_{i+1}^{r_{i+1}/r_i}$, for $1 \leqslant i \leqslant m-1$. That is, the alphabet of $\hat{T}'$ is not fixed, which means a naive construction based on sorting takes $\Omega(|T| \log |T|)$ time, even if T is over fixed alphabets. Also, checking for every position of $\hat{P}'[2, u-1]$ takes $O(|T|)$ time in the worst case, by which one would obtain an $\langle O(|T| \log |T|), O(|P| + |T|)\rangle$-algorithm. In the following, we propose an $\langle O(|T|), O(|P| + U_p + \log m)\rangle$-algorithm, where U_p denotes the number of positions in T that a proportionally-scaled P occurs.

4.1. Constructing the suffix array of $\hat{T}'$

To show that the required suffix array can be constructed in $O(|T|)$ time, we give an $O(|T|)$-time sorting algorithm that transforms $\hat{T}'$ into a string over $\{1, 2, \ldots, m-1\}$, where $m \leqslant |T|$, for any T over a fixed alphabet or an integer alphabet. Suppose $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_{|\Sigma|}\}$, where $\sigma_i < \sigma_{i+1}$, for $1 \leqslant i \leqslant |\Sigma| - 1$. In addition, let $|\sigma_i|$ denote the number of occurrences of σ_i in T. Let σ_i^Y be the sequence of characters $\sigma_i^{x/y}$ in $\hat{T}'$, where x and y are both positive integers that $y > x$. For simplicity, let Y denote the maximal y in σ_i^Y. Likewise, let σ_i^X denote the sequence of characters $\sigma_i^{x/y}$ in $\hat{T}'$ for $x > y$, and X stands for the maximal x in σ_i^X. Let $D(\sigma_i^Y)$ and $D(\sigma_i^X)$ be the set of distinct fractions in σ_i^Y and σ_i^X, respectively. Taking $T' = c^{10}a^4b^6a^3b^7c^8a^{12}c^{10}b^5a^2$ for example, we have $\hat{T}' = a^{2/5}b^{3/2}a^{1/2}b^{7/3}c^{8/7}a^{3/2}c^{5/6}b^{1/2}a^{2/5}$, $a^Y = a^{2/5}a^{1/2}a^{2/5}$, $a^X = a^{3/2}$, $D(a^Y) = \{\frac{2}{5}, \frac{1}{2}\}$, and $D(a^X) = \{\frac{3}{2}\}$. Also, for a^Y and a^X we have $Y = 5$ and $X = 3$, respectively.

Algorithm 1 Transform $\hat{T}'$ into a string over $\{1, 2, \ldots, m-1\}$.

Split $\hat{T}'$ to obtain each σ_i^Y and σ_i^X, for $1 \leqslant i \leqslant |\Sigma|$.
Set $h = 0$.
for $i = 1$ to $i = |\Sigma|$ **do**
  Compute $D(\sigma_i^Y)$ with bucket sort.
  Compute $D(\sigma_i^X)$ with bucket sort.
  Map $D(\sigma_i^Y)$ to $\{h+1, h+2, \ldots, h+|D(\sigma_i^Y)|\}$.
  Map 1 to $h + |D(\sigma_i^Y)| + 1$.
  Map $D(\sigma_i^X)$ to $\{h + |D(\sigma_i^Y)| + 2, h + |D(\sigma_i^Y)| + 3, \ldots, h + |D(\sigma_i^Y)| + |D(\sigma_i^X)| + 1\}$.
  Transform σ_i^Y and σ_i^X into sequences over $\{h+1, h+2, \ldots, h + |D(\sigma_i^Y)| + |D(\sigma_i^X)| + 1\}$.
  Set $h = h + |D(\sigma_i^Y)| + |D(\sigma_i^X)| + 1$.
end for
Assemble the transformed σ_i^Y and σ_i^X to obtain the transformed $\hat{T}'$, for $1 \leqslant i \leqslant |\Sigma|$.

Table 1
Indexing algorithms for scaled patterns.

Algorithm	Scale	Property	Complexity
Peng et al. [14]	integer scale with $\alpha \geqslant 1$	proportional	$\langle O(\|T\|), O(\|P\| + U_d)\rangle$
Peng et al. [15]	real scale with $\alpha \geqslant 1$	general	$\langle O(\|T\|^2), O(\|P\| + U_r)\rangle$
This paper	real scale with $\alpha > 0$	proportional	$\langle O(\|T\|), O(\|P\| + U_p + \log m)\rangle$

Lemma 3. *Algorithm* 1 *can be implemented with $O(|T|)$ time, provided that Σ is a fixed alphabet or an integer alphabet.*

Proof. First, one can see that splitting and assembling of $\hat{T}'$ can be done in $O(|T|\log|\Sigma|)$ time, which reduces to $O(|T|)$ time if Σ is a fixed alphabet or an integer alphabet. In the following we show that the loop also takes $O(|T|)$ time, by which the lemma holds.

To determine $D(\sigma_i^Y)$, one can separate σ_i^Y into Y lists, where the jth list stores the fractions with denominator j, for $1 \leqslant j \leqslant Y$. In T', let $K_{\sigma_i} = \sum_{t_{k+1}=\sigma_i} r_k$ be the summation of the run length for each $T'[k] = t_k^{r_k}$ with its next character $T'[k+1] = t_{k+1}^{r_{k+1}} = \sigma_i^{r_{k+1}}$. By applying bucket sort to each nonempty list, $D(\sigma_i^Y)$ can be determined in $O(K_{\sigma_i})$ time. Similarly, $D(\sigma_i^X)$ can be determined in $O(|\sigma_i|)$ time by considering the numerator with X lists. With the rule of counting rational numbers, such as $\frac{1}{1}, \frac{1}{2}, \frac{2}{1}, \frac{1}{3}, \frac{2}{2}, \frac{3}{1}, \ldots$, one can verify that $|D(\sigma_i^Y)|$ and $|D(\sigma_i^X)|$ are bounded by $O(K_{\sigma_i}^{2/3})$ and $O(|\sigma_i|^{2/3})$, respectively. Therefore, it takes $O(K_{\sigma_i}^{2/3}\log K_{\sigma_i} + |\sigma_i|^{2/3}\log|\sigma_i|) = O(K_{\sigma_i} + |\sigma_i|)$ time to map $D(\sigma_i^Y)$ and $D(\sigma_i^X)$ into integers. Note that for the jth list used in computing $D(\sigma_i^Y)$ ($D(\sigma_i^X)$), one can associate an array of size j to store the mapping if the list is nonempty. In this way, the overall size of these arrays is bounded by $O(K_{\sigma_i} + |\sigma_i|)$. By looking up these arrays, the transformation for σ_i^Y and σ_i^X can be done in $O(|\sigma_i^Y| + |\sigma_i^X|) = O(|\sigma_i|)$ time. Note that $O(\sum_{i=1}^{|\Sigma|} K_{\sigma_i} + \sum_{i=1}^{|\Sigma|}|\sigma_i|) = O(|T|)$. That is, the loop in this algorithm takes $O(|T|)$ time, by which the lemma holds. □

With Theorem 1 and Lemma 3, the suffix array $\hat{T}'_A$ of $\hat{T}'$ can be constructed in $O(|T|)$ time for any T over fixed alphabets or integer alphabets.

4.2. Two-phase searching with RMQ

Recall that a trivial checking for Lemma 2, which examines all positions of $\hat{P}'[2, u-1]$ in $\hat{T}'$, still takes $O(|T|)$ time in the worse case. In the following, we show how to improve the searching time from $O(|T|)$ to $O(|P| + U_p + \log m)$, by which we complete our indexing algorithm. Briefly, our searching strategy can be realized as a two-phase searching with RMQ. Given $\hat{P}' = p_2^{s_2/s_1} p_3^{s_3/s_2} \ldots p_u^{s_u/s_{u-1}}$, in the first phase we search for two strings $p_3^{s_3/s_2} p_4^{s_4/s_3} \ldots p_u^{s_u/s_{u-1}}$ and $p_3^{s_3/s_2} p_4^{s_4/s_3} \ldots p_u^{|T|}$ with the suffix array $\hat{T}'_A$. Therefore, it takes $O(|P| + \log m)$ time to obtain the interval $[i_1, i_2]$ such that for any $i_1 \leqslant j \leqslant i_2$, $\hat{T}'_A[j]$ is a position satisfying conditions (1) and (3) in Lemma 2. Note that though the string $p_3^{s_3/s_2} p_4^{s_4/s_3} \ldots p_u^{|T|}$ does not exist in $\hat{T}'$, the searching can still locate i_2. In the preprocessing phase, we associate a 3-tuple $(t_{\hat{T}'_A[j]-1}, t_{\hat{T}'_A[j]}, \frac{r_{\hat{T}'_A[j]}}{r_{\hat{T}'_A[j]-1}})$ to each element $\hat{T}'_A[j]$ in $\hat{T}'_A$, for $1 \leqslant j \leqslant m-1$. For $\hat{T}'_A[j] = 1$ (the first position of $\hat{T}'$), set its tuple to $(-\infty, -\infty, -\infty)$. Clearly, the second condition in Lemma 2 can be checked by searching for tuples of the form $(p_1, p_2, \frac{r_{\hat{T}'_A[j]}}{r_{\hat{T}'_A[j]-1}} \leqslant \frac{s_2}{s_1})$, for $i_1 \leqslant j \leqslant i_2$. One can see that an element $\hat{T}'_A[j]$ with the tuple $(p_1, p_2, \frac{r_{\hat{T}'_A[j]}}{r_{\hat{T}'_A[j]-1}} \leqslant \frac{s_2}{s_1})$ indicates the proportionally matched position $\hat{T}'_A[j]-1$ in T'.

For efficiency, we separate these 3-tuples into smaller arrays with respect to $t_{\hat{T}'_A[j]-1}$ and $t_{\hat{T}'_A[j]}$. According to Lemma 1, one can preprocess these arrays in $O(m)$ time, so that the three-sided query can be answered in $O(1)$ time, for any given subarray of a small array. Because it takes $O(\log m)$ time to locate the required subarray, the overall answering time would be $O(|P| + U_p + \log m)$.

One can verify that the separation of tuples can be done in $O(|T|\log|\Sigma|)$ time and $O(|T|)$ time for fixed alphabets and integer alphabets, respectively. Hint: The separation for integer alphabets can be done by applying radix sort with $(t_{\hat{T}'_A[j]-1}, t_{\hat{T}'_A[j]}, j)$ as the key and $\frac{r_{\hat{T}'_A[j]}}{r_{\hat{T}'_A[j]-1}}$ as the value. Finally, note that the special case for $|P'| \leqslant 2$ (see Section 3) can be answered in $O(|P| + U_p + \log m)$ time by using the suffix array of $t_1 t_2 \ldots t_m$. Therefore, we complete our $\langle O(|T|), O(|P| + U_p + \log m)\rangle$-algorithm.

Table 1 summarizes our result with other indexing algorithms for scaled patterns. For simplicity, we assume the alphabet of T is fixed. In Table 1, U_d and U_r denote the number of matched positions of P in T, for integer and real scale with $\alpha \geqslant 1$, respectively. Note that for any integer scale $\alpha \geqslant 1$, $\delta_\alpha(P)$ is a proportionally-scaled pattern. Therefore, against the integer scale, our indexing algorithm is more suitable for detecting scaled patterns. Our algorithm is also the first known indexing algorithm that considers both enlarged and shrunk patterns. For indexing real scaled patterns, our algorithm is a good alternative because of its efficiency.

5. Future work

Based on our results, there are two interesting topics worth future study. The first is to improve our algorithm so that the cost $\log m$ in the answering phase can be removed, for any T over fixed alphabets. That is, we would like to know if there exists any optimal indexing algorithm for fixed alphabets. The second topic is to devise efficient algorithms for finding the shrunk pattern $\delta_\alpha(P) = p_1^{\lceil \alpha s_1\rceil} p_2^{\lceil \alpha s_2\rceil} \ldots p_u^{\lceil \alpha s_u\rceil}$, for any $0 < \alpha \leqslant 1$. Amir et

al. [2] proposed a novel $O(|T| + |P|)$-time algorithm that reports all matched position of $p_1^{\lfloor \alpha s_1 \rfloor} p_2^{\lfloor \alpha s_2 \rfloor} \dots p_u^{\lfloor \alpha s_u \rfloor}$ in T, for $\alpha \geqslant 1$. Their algorithm can be applied to match the scaled pattern $p_1^{\lceil \alpha s_1 \rceil} p_2^{\lceil \alpha s_2 \rceil} \dots p_u^{\lceil \alpha s_u \rceil}$ in $O(|T| + |P|)$ time, for $\alpha \geqslant 1$. However, it cannot be directly applied to the case for $0 < \alpha \leqslant 1$. Therefore, to study the case for such shrunk patterns would make the research of scaled matching more complete.

References

[1] S. Alstrup, G.S. Brodal, T. Rauhe, New data structures for orthogonal range searching, in: IEEE Symposium on Foundations of Computer Science, Redondo Beach, California, USA, 2000, pp. 198–207.

[2] A. Amir, A. Butman, M. Lewenstein, Real scaled matching, Information Processing Letters 70 (4) (1999) 185–190.

[3] A. Amir, G.M. Landau, U. Vishkin, Efficient pattern matching with scaling, Journal of Algorithms 13 (1992) 2–32.

[4] A. Amir, A. Butman, M. Lewenstein, E. Porat, D. Tsur, Efficient one dimensional real scaled matching, Journal of Discrete Algorithms 5 (2) (2007) 205–211.

[5] A. Amir, G. Călinescu, Alphabet-independent and scaled dictionary matching, Journal of Algorithms 36 (1) (2000) 34–62.

[6] A. Amir, M. Lewenstein, E. Porat, Faster algorithms for string matching with k mismatches, Journal of Algorithms 50 (2) (2004) 257–275.

[7] M.A. Bender, M. Farach-Colton, The LCA problem revisited, in: Proceedings of Latin American Theoretical Informatics, Punta del Este, Uruguay, 2000, pp. 88–94.

[8] P. Clifford, R. Clifford, Simple deterministic wildcard matching, Information Processing Letters 101 (2007) 53–54.

[9] T. Eilam-Tzoreff, U. Vishkin, Matching patterns in a string subject to multi-linear transformation, Theoretical Computer Science 60 (1988) 231–254.

[10] M. Farach-Colton, P. Ferragina, S. Muthukrishnan, On the sorting-complexity of suffix tree construction, Journal of the ACM 47 (6) (2000) 987–1011.

[11] D.K. Kim, J.S. Sim, H. Park, K. Park, Constructing suffix arrays in linear time, Journal of Discrete Algorithms 3 (2–4) (2005) 126–142.

[12] D. Knuth, J. Morris, V. Pratt, Fast pattern matching in strings, SIAM Journal on Computing 6 (1) (1977) 323–350.

[13] U. Manber, G. Myers, Suffix arrays: A new method for on-line string searches, SIAM Journal on Computing 22 (5) (1993) 935–948.

[14] Y.-H. Peng, C.-B. Yang, K.-S. Huang, H.-Y. Ann, Efficient indexing algorithms for one-dimensional discretely-scaled strings, Information Processing Letters 110 (2010) 730–734.

[15] Y.-H. Peng, C.-B. Yang, C.-T. Tseng, C.-Y. Hor, A new efficient indexing algorithm for one-dimensional real scaled patterns, Journal of Computer and System Sciences (2010), in press.

[16] E. Ukkonen, On-line construction of suffix trees, Algorithmica 14 (3) (1995) 249–260.

[17] B.-F. Wang, J.-J. Lin, S.-C. Ku, Efficient algorithms for the scaled indexing problem, Journal of Algorithms 52 (1) (2004) 82–100.