Information Processing Letters 110 (2010) 730–734

Contents lists available at ScienceDirect

Information Processing Letters

www.elsevier.com/locate/ipl

# Efficient indexing algorithms for one-dimensional discretely-scaled strings ☆

Yung-Hsing Peng, Chang-Biau Yang *, Kuo-Si Huang, Hsing-Yen Ann

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

ARTICLE INFO

*Article history:*
Received 26 October 2009
Received in revised form 17 March 2010
Accepted 14 May 2010
Available online 4 June 2010
Communicated by M. Yamashita

*Keywords:*
Algorithm
String matching
Discrete scale

ABSTRACT

The discretely-scaled string indexing problem asks one to preprocess a given text string $T$, so that for a queried pattern $P$, the matched positions in $T$ that $P$ appears with some discrete scale can be reported efficiently. For solving this problem, Wang et al. first show that with an $O(|T|\log|T|)$-time preprocessing on $T$, all matched positions can be reported in $O(|P|+U_d)$ time, where $|T|$, $|P|$, and $U_d$ denote the length of $T$, the length of $P$, and the number of matched positions for discretely-scaled $P$ in $T$, respectively. In this paper, for fixed alphabets we propose the first-known optimal indexing algorithm that takes $O(|T|)$ and $O(|P|+U_d)$ time in its preprocessing and query phases, respectively. For integer and unbounded alphabets, our new algorithm can also be extended to obtain the best-known results.



## 1. Introduction

Pattern matching has been well studied for several decades. Given a pattern $P$ and a text $T$, the pattern matching problem is to ask for all positions in $T$, such that $P$ can be detected. When both $P$ and $T$ are one-dimensional strings, the problem is also called string matching, which can be solved in linear time [11]. Similar to but different from string matching, the *string indexing problem* aims to preprocess $T$, so that $P$ can be detected more efficiently [10,14]. Recently, related problems that involve matching [2–5] or indexing [12,13,15] scaled strings have drawn much attention, because they are considered realistic in the field of computer vision. In this paper, we focus on the indexing problem of *one-dimensional discretely-scaled strings* [15], where the pattern string $P$ can be scaled with some natural scales. Taking $T = a^2c^6a^2b^3a^4$ and $P = c^2a^1b^1$ (run-length encoding) for example, one can see that $c^4a^2b^2$ is a 2-scaling of $P$ that appears in $T$.

The concept of detecting one-dimensional discretely-scaled strings was first proposed by Amir et al. [3], who also gave an $O(|T|+|P|)$-time matching algorithm by adapting Eilam-Tzoreff and Vishkin's algorithm [7]. Afterwards, Wang et al. [15] gave the first-known indexing algorithm for one-dimensional discretely-scaled strings, which takes $O(|T|\log|T|)$ time for preprocessing and $O(|P|+U_d)$ time for query, denoted as $\langle O(|T|\log|T|), O(|P|+U_d)\rangle$, where $U_d$ represents the number of matched positions of discretely-scaled $P$ in $T$. In this paper, we propose an optimal indexing algorithm with complexity $\langle O(|T|), O(|P|+U_d)\rangle$ for fixed alphabets, which is an improvement on Wang's algorithm. Next, we show that this algorithm can be slightly modified to achieve the best-known results for integer alphabets and unbounded alphabets.

The rest of this paper is organized as follows. In Section 2, we explain the required notations and prerequisite knowledge. After that, in Section 3 we propose our indexing algorithms. Finally, in Section 4 we give interesting problems for future study.

☆ This research work was partially supported by the National Science Council of Taiwan under contract NSC-95-2221-E-110-102.

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

0020-0190/$ – see front matter 
doi:10.1016/j.ipl.2010.05.012

## 2. Preliminary

For any string $S$, let $S[i]$ denote the $i$th 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 $S$. Let $T' = t_1^{r_1} t_2^{r_2} \cdots t_m^{r_m}$ be the run-length encoded (RLE) string 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 $P' = p_1^{s_1} p_2^{s_2} \cdots p_u^{s_u}$ be the RLE string of $P$ with $|P'| = u$. The $\alpha$-scaling of $P$, denoted by $\delta_\alpha(P)$, for any integer $\alpha \geqslant 1$, represents the string $p_1^{\alpha s_1} p_2^{\alpha s_2} \cdots p_u^{\alpha s_u}$. Given a text $T$ and a pattern $P$, the *discretely-scaled string indexing problem* asks one to preprocess $T$, so that one can efficiently report every position for which there exists an $\alpha \geqslant 1$ such that $\delta_\alpha(P)$ matches. In addition, the *discretely-scaled string matching decision problem* is to decide whether some $\delta_\alpha(P)$ appears in $T$.

### 2.1. Suffix tree and suffix array

Given a text $T$ over the alphabet $\Sigma$, the suffix tree [9,14] $T_S$ of $T$ is an $O(|T|)$-space compacted trie of all suffixes in $T$. Given a pattern $P$ along with the suffix tree $T_S$, one can determine all positions of $P$ in $T$ with $O(|P| \log |\Sigma| + U)$ time, where $U$ denotes the number of matched positions. Similar to but different from a suffix tree, a suffix array [10] lexically stores each index (suffix) of $T$. Since one index is sufficient to represent the suffix starting at $T[i]$, the suffix array $T_A$ of $T$ occupies only $O(|T|)$ space, which is independent of $\Sigma$. By keeping the information of the longest common prefix of suffixes, to search $P$ in $T$, one can perform a binary search on $T_A$, which takes $O(|P| + U + \log |T|)$ time to report all matched positions [10]. The details for suffix trees and suffix arrays are omitted here, since they are beyond the scope of this paper. In the following, we focus only on required theorems.

**Theorem 1.** *(See [9].) Given a string $T$ over the integer alphabet $\{1, 2, \ldots, |T|\}$, the suffix tree $T_S$ of $T$ can be constructed in $O(|T|)$ time.*

Based on Theorem 1, one can construct the suffix tree of $T$ in $O(|T| + Sort)$ time, where *Sort* denotes the required time to transform $T$ into a string over $\{1, 2, \ldots, |T|\}$. Therefore, for unbounded alphabets, it takes $\Omega(|T| \log |T|)$ time to construct the suffix tree by sorting. In addition, Theorem 1 also indicates the correctness of Theorem 2, because the suffix array $T_A$ of $T$ can be constructed in $O(|T|)$ time once $T_S$ is given.

**Theorem 2.** *(See [9].) Given a string $T$ over $\{1, 2, \ldots, |T|\}$, the suffix array $T_A$ of $T$ can be constructed in $O(|T|)$ time.*

### 2.2. The range minimum (maximum) query and the three-sided query

Given an array $D$ of $|D|$ numbers, the *range minimum (maximum) query* (*RMQ*) asks for the minimum (maximum) element in the subarray $D[i_1, i_2]$, for any given interval $1 \leqslant i_1 \leqslant i_2 \leqslant |D|$. Bender and Farach-Colton [6] gave an elegant algorithm for preprocessing $D$, so that each RMQ can be answered in $O(1)$ time. Let $RMQ_D(i_1, i_2)$ be the index of the minimum (maximum) element in the subarray $D[i_1, i_2]$. We summarize their result as follows.

**Theorem 3.** *(See [6].) Given an array $D$ of $|D|$ numbers, one can preprocess $D$ in $O(|D|)$ time such that for any given interval $[i_1, i_2]$, one can determine $RMQ_D(i_1, i_2)$ in $O(1)$ time.*

Applying Theorem 3 recursively, one can see the correctness of Lemma 1, which also summarizes the *three-sided query* [1].

**Lemma 1.** *(See [1].) Given an array $D$ of $|D|$ numbers and a threshold $c$, one can preprocess $D$ in $O(|D|)$ 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 $D[i'] \geqslant c$, where $U_x$ is the number of reported indices.*

### 2.3. The inverse dictionary

To efficiently find $\delta_\alpha(P)$ in $T$, Wang et al. [15] construct a set of strings that considers all possible scales for $T$, which we call the *inverse dictionary* of $T$. Let $r_{\max} = \max\{r_1, r_2, \ldots, r_m\}$ and $G = \{1, 2, \ldots, r_{\max}\}$. For each $g \in G$, the $g$-inverse string $T_g$ of $T$ is constructed by replacing characters in $T$ according to $\frac{r_i}{g}$, for $1 \leqslant i \leqslant m$. If $g$ is a divisor of $r_i$, then $t_i^{r_i}$ will be replaced by $t_i^{\frac{r_i}{g}}$. Otherwise, $t_i^{r_i}$ will be replaced by $t_i^{\lfloor \frac{r_i}{g} \rfloor} \$ t_i^{\lfloor \frac{r_i}{g} \rfloor}$, where \$ denotes the terminate symbol used to concatenate strings and $\$ \notin \Sigma$. Taking $T = a^2 c^6 a^2 b^3 a^4$ and $g = 2$ for example, we have $T_2 = a^1 c^3 a^1 b^1 \$ b^1 a^2$. Let $T_G = T_1 \$ T_2 \$ \cdots \$ T_{r_{\max}}$ be a plain string that stores all $g$-inverse strings. In the following we describe important properties derived by Wang et al., which is an inspiration for our improvement.

**Lemma 2.** *(See [15].) Given $P$ and $T$ with $|P'| \geqslant 2$, the set of positions that $\delta_\alpha(P)$ occurs in $T$ is bijective to the set of positions that $P$ occurs in $T_G$.*

Fig. 1 is a simple example for illustrating Lemma 2. In Fig. 1, one can see that the substring $c^2 a^1 b^1$ in $T_2$ maps to the substring $c^4 a^2 b^2$, which is also a 2-scaling of $P = c^2 a^1 b^1$. For $|P'| = 1$, the problem reduces to indexing $P$ in $T$ [15], which can be solved with the suffix tree of $T$.

Proving that the length of $T_G$ is bounded by $O(|T| \times \log |T|)$ [15], Wang et al. derived an $\langle O(|T| \log |T|), O(|P| + U_d) \rangle$ indexing algorithm by the suffix tree of $T_G$. Note that other than building the inverse dictionary $T_G$, one may consider the approach of indexing quotient strings [3]: to find $p_2^{\frac{s_2}{s_1}} p_3^{\frac{s_3}{s_2}} \cdots p_u^{\frac{s_u}{s_{u-1}}}$ in the suffix tree of $t_2^{\frac{r_2}{r_1}} t_3^{\frac{r_3}{r_2}} \cdots t_m^{\frac{r_m}{r_{m-1}}}$. Up to now, the best-known result for indexing quotient strings over fixed alphabets is of complexity $\langle O(|T|), O(|P| + U_p + \log m) \rangle$ [13], where $U_p$ denotes the number of matched positions for proportionally-scaled $P$ in $T$. However, note that $U_p \geqslant U_d$ because the scales considered

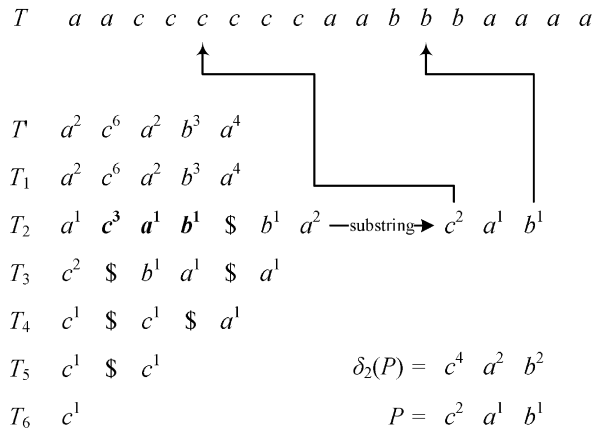


**Fig. 1.** An example for illustrating the relationship between $T$, $T_g$, $P$, and $\delta_g(P)$.

in quotient strings are not confined to natural numbers. Taking $T = c^6a^4b^3c^3a^5$ and $P = a^2b^2c^2$ for example, we have $U_d = 0$ but $U_p = 1$ (with the non-discrete scale $\frac{3}{2}$). Therefore, we have to examine each of the $U_p$ proportionally matched positions to see if its corresponding scale is discrete or not. Clearly, the best known indexing algorithm on quotient strings [13] is not optimal for indexing one-dimensional discretely-scaled strings. In the next section, for fixed alphabets we propose the first-known algorithm of the optimal complexity $\langle O(|T|), O(|P| + U_d)\rangle$.

## 3. Improved indexing algorithms

Briefly, our indexing algorithm consists of two main parts. First, we propose an $O(|T|)$-time strategy that stores the inverse dictionary in RLE strings. Next, we give an optimal $O(|P| + U_d)$-time searching strategy with the three-sided query.

### 3.1. Building the RLE dictionary

To reduce the required space, we store $T_G$ implicitly by using its RLE string $T'_G$. Note that each character $t_i^{r_i}$ in $T'$ generates no more than $O(r_i)$ characters in $T'_G$, and we can represent consecutive terminate symbols by one single \$. This means $T'_G$ takes only $O(|T|)$ space. To construct $T'_G$ in $O(|T|)$ time, we create a list *Life* of length $m$, in which each element is a pair of integers $(life, pos)$. The $i$th element in the list is initially set to $(r_i, i)$, which means the life of $T'[i]$ is initially set to $r_i$. In our construction, we decrease the life of $T'[i]$ by 1 after finishing one replacement for $t_i^{r_i}$. In addition, an element $(life_i, pos_i)$ is removed from *Life* if $life_i = 0$. With the information of *Life*, each $T_g$ can be constructed one by one in the order of $g = 1$, $g = 2, \ldots, g = r_{\max}$, skipping all $t_i^{r_i}$ in $T$ that $r_i < g$. When a skipping occurs (this happens when $pos_{i+1} \neq pos_i + 1$), we simply insert a single \$ as a separation. In this way, $T'_G$ can be constructed in $O(|T|)$ time. In addition, it is easy to see that each position in $T'_G$ maps to one position in $T'$, because $T'_G$ is produced from $T'$ (see Fig. 1). With this observation, we derive the following property.

**Lemma 3.** *Given $P$ and $T$ with $|P'| \geqslant 2$, the set of positions that $\delta_\alpha(P)$ occurs in $T$ is bijective to the set of positions that $P'$ occurs in $T'_G$.*

**Proof.** Assume that $P'$ occurs at $T'_g[i]$ in $T'_G$, where $T'_g$ is the RLE string of $T_g$, for some natural scales $g \leqslant r_{\max}$. Also, assume that $T'_g[i]$ maps to $T'[i']$ in $T'$. Since $T'$ is the RLE string of $T$, this implies that $\delta_g(P)$ occurs at position $((\sum_{j=1}^{i'} r_j) - g \times s_1 + 1)$ in $T$. If $P'$ does not occur at any $T'_g[i]$, then $P$ does not occur at any $T_g[i]$, which means $\delta_g(P)$ does not occur in $T$, for any natural scale $g$. Hence, the lemma holds. □

With an $O(|T|)$-time preprocessing on $T$, for any $g$ the value $((\sum_{j=1}^{i'} r_j) - g \times s_1 + 1)$ can be determined in $O(1)$ time. Therefore, the problem now reduces to indexing $P'$ in $T'_G$. For indexing $P'$ in $T'_G$, we construct the suffix tree $T'_{GS}$ of $T'_G$. Note that the alphabet of $T'_G$ is not fixed, which means a straightforward construction of $T'_{GS}$ will take $O(|T| \log |T|)$ time. Taking $\Sigma = \{a, b\}$ (for plain text $T$) and $T'_G = a^2b^3\$a^4b^5a^1\$a^1$ for example, $a^2$, $a^4$, and $a^1$ are distinct symbols even if they are of the same character '$a$'. In this example, $T'_G$ contains 6 distinct symbols, including the terminate symbol \$. Though we notice that the SBC-tree (string B-tree for compressed sequences) [8] is a suffix-tree like data structure for indexing RLE strings, for $T$ over fixed alphabets it is an $\langle O(|T| \log |T|), O(|P| + U_d + \log |T|)\rangle$ algorithm. In the following, we explain how to construct $T'_{GS}$ in $O(|T|)$ time, and then give an $O(|P| + U_d)$-time strategy for searching $P'$ in $T'_{GS}$.

**Lemma 4.** *The suffix tree $T'_{GS}$ of $T'_G$ can be built in $O(|T|)$ time for $T$ over fixed alphabets.*

**Proof.** Let $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_{|\Sigma|}\}$ be the given fixed alphabet. For each $\sigma \in \Sigma$, we record its maximum run length in $T$, which can be done in $O(|T|)$ time for fixed $|\Sigma|$. Let $M(\sigma)$ denote the maximum run length of symbol $\sigma$ in $T$. For each RLE character $\sigma_i^r$ in $T'_G$, $1 \leqslant r \leqslant M(\sigma_i)$, we transform it into the integer $(\sum_{j=1}^{i-1} M(\sigma_j)) + r$. That is, for any two RLE characters carrying the same symbol, written as $\sigma_i^{r_1}$ and $\sigma_i^{r_2}$, we have $\sigma_i^{r_1} \leqslant \sigma_i^{r_2}$ if $r_1 \leqslant r_2$. Also, we have $\sum_{j=1}^{|\Sigma|} M(\sigma_j) \leqslant |T|$, which means the RLE characters in $T'_G$ are transformed into integers in $\{1, 2, \ldots, |T|\}$. Since for fixed $|\Sigma|$ the transformation can be done in $O(|T|)$ time, by Theorem 1, the lemma holds. □

Note that each $t_i^{r_i}$ in $T'$ may generate $O(\sqrt{r_i})$ distinct run lengths in $T'_G$. Therefore, one can further verify that each $t_i$ in $T'_G$ may have $O(\sqrt{|T|})$ distinct run lengths. To find $P' = p_1^{s_1}p_2^{s_2}\cdots p_u^{s_u}$ in $T'_G$, from the root of $T'_{GS}$ we have to explore every edge of symbol $p_1$ whose run length is greater than or equal to $s_1$. In the worst case, there will be $O(\sqrt{|T|})$ such explorations. That is, a naive approach that searches $P'$ in $T'_{GS}$ for $O(\sqrt{|T|})$ times would take $O(|P|\sqrt{|T|})$ time. In the following, we show how to overcome this bottleneck by the three-sided query.

**Table 1**
Indexing algorithms for one-dimensional discretely-scaled strings.

| Algorithm | Complexity | Fixed alphabets | Integer alphabets | Unbounded alphabets |
|---|---|---|---|---|
| Wang et al. [15] | Time | $O(\|T\|\log\|T\|)$ | $O(\|T\|\log\|T\|)$ | $O(\|T\|\log\|T\|)$ |
| | Space | $O(\|T\|\log\|T\|)$ | $O(\|T\|\log\|T\|)$ | $O(\|T\|\log\|T\|)$ |
| | Decision | $O(\|P\|)$ | $O(\|P\|+\log\|T\|)$ | $O(\|P\|+\log\|T\|)$ |
| | Position | $O(\|P\|+U_d)$ | $O(\|P\|+U_d+\log\|T\|)$ | $O(\|P\|+U_d+\log\|T\|)$ |
| This paper | Time | $O(\|T\|)$ | $O(\|T\|)$ | $O(\|T\|\log\|T\|)$ |
| | Space | $O(\|T\|)$ | $O(\|T\|)$ | $O(\|T\|)$ |
| | Decision | $O(\|P\|)$ | $O(\|P\|+\log\|T\|)$ | $O(\|P\|+\log\|T\|)$ |
| | Position | $O(\|P\|+U_d)$ | $O(\|P\|+U_d+\log\|T\|)$ | $O(\|P\|+U_d+\log\|T\|)$ |

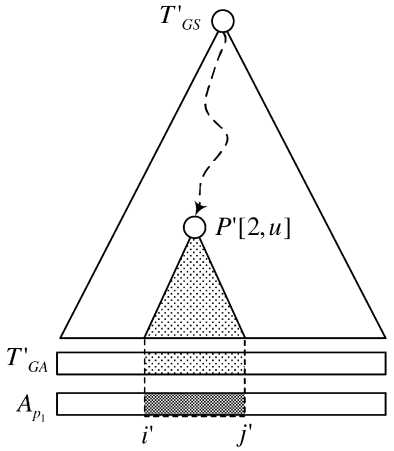


**Fig. 2.** An illustration for our searching strategy.

### 3.2. Searching with the three-sided query

Our strategy consists of two steps. First, we skip the character $p_1^{s_1}$ but locate the remaining pattern $P'[2,u]$ in $T'_G$. Then, for each located $T'_G[i]$, we check if $T'_G[i-1]$ can match $p_1^{s_1}$. Fig. 2 shows an illustration for our searching strategy, whose details are described as follows. Let $T'_{GA}$ be the suffix array that lexically stores the leaf nodes in $T'_{GS}$. For each $\sigma \in \Sigma$, we create an array $A_\sigma$ of size $|T'_G|$, where $A_\sigma[i]$ stores the run length of $T'_G[T'_{GA}[i]-1]$ if $T'_G[T'_{GA}[i]-1]$ carries the symbol $\sigma$. For each $T'_G[T'_{GA}[i]-1]=\sigma'_j$ that $\sigma_j \neq \sigma$, we set $A_\sigma[i]=0$. Note that the set of positions of $P'[2,u]$ in $T'_G$ maps to a subarray $T'_{GA}[i',j']$ of $T'_{GA}$. Therefore, the checking step for $p_1^{s_1}$ reduces to a three-sided query on $A_{p_1}[i',j']$ with threshold $c=s_1$, which can be done in optimal $O(U_d)$ time by Lemma 1. To complete our algorithm, in the following we show that with an $O(|T'_{GS}|)$-time preprocessing on $T'_{GS}$, the set of positions for $P'[2,u]$ in $T'_G$ can be determined in $O(|P|)$ time.

**Lemma 5.** *One can preprocess $T'_{GS}$ in $O(|T'_{GS}|)$ time, so that in $O(|P|)$ time, the set of positions of $P'[2,u]$ in $T'_G$ can be determined (for fixed alphabets).*

**Proof.** We accomplish the proof by discussing two parts encountered in the searching progress. The first part is for the middle pattern $p_2^{\alpha s_2}p_3^{\alpha s_3}\cdots p_{u-1}^{\alpha s_{u-1}}$. To search $p_i^{s_i}$, we can first locate the set of outgoing edges of $p_i$ in $O(1)$ time. Note that each internal node of $T'_{GS}$ has no more than $|\Sigma|$ such sets, and $|\Sigma|$ is fixed. We also label each edge with the run length of its first character. By doing so, in the located set the binary search for $s_i$ can always be done in $O(\log s_i)$ time, because we need only search the first $s_i$ edges. Therefore, it requires $O(\sum_{i=2}^{u-1}\log s_i)$ time for searching $p_2^{\alpha s_2}p_3^{\alpha s_3}\cdots p_{u-1}^{\alpha s_{u-1}}$.

The second part is to search $p_u^{s_u}$, which can be done by first locating the set of edges of $p_u$ in $O(1)$ time, and then reporting all edges whose run lengths are greater than or equal to $s_u$. Note that, in fact, only two such edges need be reported (the first and the last). Among the edges of $p_u$, it takes $O(\log s_u)$ time to find the first edge. Because the last edge can be located in $O(1)$ time, the searching time for $p_u^{s_u}$ is $O(\log s_u)$. Therefore, the overall searching time for $p_2^{\alpha s_2}p_3^{\alpha s_3}\cdots p_u^{\alpha s_u}$ is $O(\sum_{i=2}^{u}\log s_i) \leqslant O(\sum_{i=2}^{u} s_i) = O(|P|-s_1) \leqslant O(|P|)$. □

Since $O(|T'_{GS}|)=O(|T|)$, for fixed alphabets our indexing algorithm achieves the optimal time complexity $\langle O(|T|), O(|P|+U_d)\rangle$. For integer alphabets, we can replace the suffix tree $T'_{GS}$ with the suffix array $T'_{GA}$. To search $p_2^{\alpha s_2}p_3^{\alpha s_3}\cdots p_u^{\alpha s_u}$ in $T'_{GA}$, we turn to search two patterns $p_2^{\alpha s_2}p_3^{\alpha s_3}\cdots p_u^{\alpha s_u}$ and $p_2^{\alpha s_2}p_3^{\alpha s_3}\cdots p_u^{\infty}$. In addition, in $A_\sigma$ we only store non-zero elements. Therefore, the size of all $A_\sigma$ arrays is bounded in $O(|T|)$. Note that it takes additional $O(\log|T|)$ time to locate the interval on small arrays for the three-sided query. Therefore, for integer alphabets we obtain an $\langle O(|T|), O(|P|+U_d+\log|T|)\rangle$ algorithm. For unbounded alphabets, one can easily derive an $\langle O(|T|\log|T|), O(|P|+U_d+\log|T|)\rangle$ result, because it needs $O(|T|\log|T|)$ time to transform an unbounded alphabet into an integer alphabet.

To summarize, we list various indexing algorithms for one-dimensional discretely-scaled strings in Table 1, including previous and our results. In this table, the terms "Time" and "Space" denote the required time and space for preprocessing, respectively. Next, the term "Decision" denotes the required time for the decision problem whether $\delta_\alpha(P)$ can be matched in $T$. Finally, we use the term "Position" to represent the time spent on finding all positions where $\delta_\alpha(P)$ occurs in $T$.

## 4. Future work

Other than discretely-scaled strings, the indexing of real-scaled strings, where the scale $\alpha$ is a real number, remains worthy of study. For $\alpha \geqslant 1$, Amir et al. [2] defined

the first real-scaling function and proposed an $O(|T|)$-time algorithm for solving the matching problem on this function. Afterwards, Wang et al. [15] proposed the first known indexing algorithm, which takes $O(|T|^3)$ time and space to achieve the searching time $O(|P| + U_r)$, where $U_r$ denotes the number of positions of $\delta_\alpha(P)$ in $T$, for some real $\alpha \geqslant 1$. Recently, Peng et al. [12] further gave an improved indexing algorithm that takes $O(|T|^2)$ time and space to achieve the searching time $O(|P| + w)$, where $w \leqslant U_r$. We notice that Amir et al. also proposed a new scaling function [4,5], which is more natural to computer vision. As we know, no optimal preprocessing has been proposed yet, either for the first or the second real-scaling function. Therefore, to devise efficient indexing algorithms on real-scaled strings remains an interesting challenge.

## 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, Real two-dimensional scaled matching, Algorithmica 53 (3) (2009) 314–336.

[5] 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.

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

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

[8] M.Y. Eltabakh, W.-K. Hon, R. Shah, W.G. Aref, J.S. Vitter, The SBC-tree: an index for run-length compressed sequences, in: Proceedings of the 11th International Conference on Extending Database Technology: Advances in Database Technology, Nantes, France, 2008, pp. 523–534.

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

[10] 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.

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

[12] 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, in press.

[13] Y.-H. Peng, C.-B. Yang, C.-T. Tseng, C.-Y. Hor, Efficient indexing for one-dimensional proportionally-scaled patterns, in: Proceedings of the 26th Workshop on Combinatorial Mathematics and Computation Theory, Chiayi, Taiwan, 2009, pp. 21–26.

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

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