An Efficient Preprocessing on the One-dimensional Real-scale Pattern Matching Problem

Yung-Hsing Peng, Chang-Biau Yang†, Chiou-Ting Tseng and Chiou-Yi Hor
Department of Computer Science and Engineering
National Sun Yat-sen University, Kaohsiung, Taiwan 80424
†cbyang@cse.nsysu.edu.tw

Abstract

Given a pattern string P and a text string T, the one-dimensional real-scale pattern matching problem asks for all matched positions in T at which P occurs for some real scales ≥ 1. This problem was first proposed by Amir et al., who also gave an algorithm with $O(n + |P|)$ time for solving it, where $|T| = n$. Recently, Wang et al. proposed a preprocessing on T with $O(n^3)$ time and space, with which for constant-sized alphabets, one can determine all matched positions in $O(|P| + U_r)$ time, where U_r denotes the number of matched positions. For large-sized alphabets, with Wang's preprocessing, it takes $O(|P| + U_r + \log n)$ time to report all matched positions. In addition, Wang et al. also proposed a preprocessing on T with $O(n^2)$ time and space, with which one can answer the decision version of the real-scale matching problem on constant-sized and large-sized alphabets in $O(|P|)$ and $O(|P| + \log n)$ time, respectively. In this paper, we propose an improved preprocessing for this problem. For constant-sized alphabets, our preprocessing takes only $O(n^2)$ time and space, while reporting all matched positions requires $O(|P| + w)$ time, where $w \leq U_r$. For the case of large-sized alphabets, our preprocessing can also be implemented with $O(n^2)$ time and space, and then all matched positions can be determined in $O(|P| + w + \log n)$ time. Compared with Wang's result, our algorithm is more efficient in both preprocessing and searching phases.

1 Introduction

Pattern matching is a classical problem which asks for all positions of a pattern P in a text T. This problem is also called *string matching* when both P and T are one-dimensional strings. According to different perspectives on T, algorithms for string matching can be classified into two main types. In the first perspective, both T and P are input strings, which means any algorithm requires $\Omega(n + |P|)$ time, where n and $|P|$ denote the length of T and P, respectively. Well-known Knuth-Morris-Pratt algorithm[12] and Boyer-Moore algorithm[8] both match this lower bound. In the second perspective, however, T is treated as a database while P is treated as the target string. That is, every algorithm now has its preprocessing phase with T and searching phase with P. In the preprocessing phase, any algorithm should take $\Omega(n)$ time since $|T| = n$. Besides, the searching phase needs $\Omega(|P| + U)$ time to report all positions, where U denotes the number of reported positions. For the second perspective, construction of suffix trees[21] and suffix arrays[1, 14] are both optimal algorithms in preprocessing and searching phases, provided that the size of the alphabet is fixed.

In addition to the original definition, other extended versions of string matching, such as string matching with don't care characters[9] or string matching with the consideration of mismatches[6], are not only more realistic and but also more interesting. Besides, when the problem comes to the case of two-dimensional matching[3, 4], it turns out to be more complicated but more similar to image matching. In this paper, we consider the one-dimensional *real-scale pattern matching (r-matching) problem* in which the pattern P can be scaled with real numbers.

Let $P = p_1^{s_1} p_2^{s_2} \cdots p_u^{s_u}$ be the run-length representation of the pattern string, where $|P| = \sum_{j=1}^{u} s_j$. The α-scaling of P, denoted by $\delta_\alpha(P)$, for any real number $\alpha \geq 1$, represents the string $p_1^{\lfloor \alpha s_1 \rfloor} p_2^{\lfloor \alpha s_2 \rfloor} \cdots p_u^{\lfloor \alpha s_u \rfloor}$. Given a text T and a pattern P, the *r-matching* problem asks one to determine every matched position i in T such that $\delta_\alpha(P)$ occurs, for some real number $\alpha \geq 1$. In addition, the *r-matching decision problem* asks one to

decide whether a given pattern P can be real-scale matched (r-matched) in T. For example, suppose $T = c^4a^5b^7a^4c^3b^4$ and $P = a^2b^3a^2c^1$, one can easily verify that $\delta_{\frac{7}{3}}(P) = a^4b^7a^4c^2$ is a substring of T.

The r-matching problem was first defined by Amir *et al.*[2], who also gave an $O(n + |P|)$ time algorithm by adapting the well known KMP algorithm[12]. In addition, they also proposed the open problem[2] which asks for proper preprocessing on T such that all r-matched positions of P in T can be determined efficiently. Recently, Wang *et al.*[20] gave the first known preprocessing[20]. For constant-sized alphabets, they gave an $O(n^3)$-time and $O(n^3)$-space preprocessing on T, all r-matched positions of P in T can be determined in $O(|P| + U_r)$ time, where U_r denotes the number of r-matched positions[20]. Besides, they also gave an $O(n^2)$-time and $O(n^2)$-space preprocessing on T so that the problem whether P can be r-matched in T can be determined in $O(|P|)$ time[20]. For large-sized alphabets, Wang's preprocessing for the r-matching problem can also be implemented with $O(n^3)$ time and $O(n^3)$ space by using a suffix array[13, 14], which achieves the answering time $O(|P| + U_r + \log n)$[20].

When the scale is confined to natural numbers, the problem is therefore called *discrete-scale pattern matching (d-matching)*. The d-matching problem was also proposed by Amir *et al.*[3], which is now thought to be the origin of the r-matching problem. For this former problem, Amir *et al.*[3] first gave an elegant algorithm with $O(n + |P|)$ time by adapting the KMP algorithm[12]. In addition, Wang *et al.*[20] also proposed the first known preprocessing with $O(n \log n)$ time, which answers a query in $O(|P|+U_r)$ and $O(|P|+U_r+\log n)$ time when the suffix tree and suffix array are used, respectively. Recently, Peng *et al.*[17] further proposed the first known optimal preprocessing for the d-matching problem in both preprocessing and searching phases.

In this paper, we focus on the original r-matching problem in which the scale is not confined to natural numbers. We improve previous results by giving an $O(n^2)$-time and $O(n^2)$-space preprocessing on T. With our preprocessing, for constant-sized alphabets, one can determine whether a pattern P is r-matched in T in $O(|P|)$ time, while all r-matched positions can be determined in $O(|P| + w)$ time, where $w \leq U_r$ and w denotes the number of *dominant positions*, which will be further explained in Section 3. With a little modification, we also show how to deal with large-sized alphabets in Section 3.

The rest of this paper is organized as follows. We will review some important theorems in Section 2, which are crucial to our algorithm. Next, we shall propose our main algorithm in Section 3 and apply it to other scaling functions[5] in Section 4. Finally, we give our conclusions along with some future works in Section 5.

2 Preliminaries

In this section, we begin with definitions related to T, which will be used in the following content. Let T be a string over the alphabet Σ, and $|\Sigma|$ denote the number of symbols in Σ. Also, let $T[i]$ denote the ith character in T, and $T[i, j]$ denote the substring ranging from $T[i]$ to $T[j]$, for $1 \leq i \leq j \leq |T|$. Let T' be the *run-length representation* of T, so that $T' = t_1^{r_1}t_2^{r_2}\cdots t_m^{r_m}$ with $|T'| = m$, where $t_i \in \Sigma$, for $1 \leq i \leq m$, $t_j \neq t_{j+1}$, $1 \leq j \leq m-1$ and r_i denotes the run length of t_i. Therefore, one can easily map each character $T'[i]$ in T' to the run $T[\sum_{j=1}^{i-1} r_j + 1, \sum_{j=1}^{i} r_j]$ in T. Now, we are ready to move on to our review.

2.1 Suffix Trees and Suffix Arrays

Given a text T of length n over the alphabet Σ, the suffix tree T_S of T denotes the 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 reported positions. For any alphabet Σ, Weiner[21] discovered that the suffix tree can be constructed in $O(|\Sigma|n)$ time. Then, several studies focused on how to build a suffix tree more efficiently[15, 19].

In addition to suffix trees, a suffix array is also an efficient data structure used in pattern matching. The suffix array T_A of T stores each suffix of T according to their lexical order. Since one index is sufficient to represent the suffix starting at $T[i]$, it can be easily seen that T_A occupies only $O(n)$ space, which is independent of Σ. In general, there are two approaches to construct a suffix array. The straightforward approach requires sorting on all suffixes, which can be done in $O(n)$ time for constant-sized alphabets and integer alphabets[11, 13]. For unbounded alphabets, however, the constructing time of a suffix array increases to $O(n \log n)$[14]. The other approach is to perform the depth-first search on the constructed suffix tree according to the lexical order and then

output the leaf nodes, which takes $O(n)$ time. By keeping the information of the longest common prefix of suffixes, to search a given pattern P in T, one can perform a binary search on T_A, which requires $O(|P| + U + \log n)$ time in the searching phase[14]. For large-sized alphabets, suffix arrays are more applicable than suffix trees for pattern matching, because of its efficiency on required space and searching time. Recently, Abouelhoda *et al.*[1] showed that the searching time can be improved to $O(|P| + U)$, in case the alphabet is constant-sized. This makes suffix arrays superior to suffix trees. However, suffix trees are still suitable for designing algorithms related to trees.

2.2 The Least Common Ancestor and the Range Minimum Query

Finding the *least common ancestor* (LCA) of two nodes in a tree is one of the well-known problems in algorithmic field. In general, the LCA problem asks one to efficiently preprocess a given rooted tree Tr, such that the LCA of any pair of nodes can be determined efficiently. Efficient algorithms for solving the LCA problem are of great interest, because the LCA problem is a crucial factor for solving other algorithmic problems, such as finding the longest common prefix of two suffixes in a suffix tree[6].

The first amazing result for the LCA problem was proposed by Harel and Tarjan[10], who showed that there exists a linear time algorithm to preprocess the given tree, such that the LCA query can be answered in constant time. Schieber and Vishkin[18] then discovered a simpler algorithm for finding LCA, compared with Harel and Tarjan's result. Recently, Bender and Farach-Colton[7] proposed an elegant algorithm for LCA, which is considered by far the easiest one to understand and implement. In addition, they also showed that the *range minimum query (RMQ)*, one of the applications of LCA, can also be optimized in both preprocessing time and query time. More formal description of RMQ is given as follows.

Given an array A of n numbers, the RMQ problem asks to efficiently preprocess A such that for any given interval $[i_1, i_2]$, $1 \leq i_1 \leq i_2 \leq n$, one can efficiently determine the minimum element in the subarray $A[i_1, i_2]$. Bender and Farach-Colton showed that the RMQ problem can be reduced to the LCA problem on the Cartesian tree of array A, which can be built in $O(n)$ time[7]. Let $RMQ_A(i_1, i_2)$ be the index of the minimum element in the subarray $A[i_1, i_2]$, Bender and Farach-Colton proposed the following result.

Theorem 1. [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 constant time.*

Note that Bender and Farach-Colton's idea can also be applied to the case of range maximum query. The following result can be easily drawn.

Theorem 2. *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 the maximum element in $A[i_1, i_2]$ in constant time.*

In addition to Theorems 1 and 2, in the following, we present a related result proposed by Muthukrishnan[16], which can be deemed as an application of RMQ.

Theorem 3. [16] *Given an array A of n integers whose absolute values are bounded by $O(n)$, one can preprocess A in $O(n)$ time such that for any given interval $[i_1, i_2]$, one can determine all distinct integers in the subarray $A[i_1, i_2]$ in $O(U_x)$ time, where U_x is the number of reported indices containing distinct integers.*

With Theorem 3, Muthukrishnan showed that the *document listing problem* (DLP)[16] can be solved optimally in both preprocessing time and searching time. Given d documents $T_1, T_2, \cdots, T_d$ over the alphabet Σ, the DLP asks one to preprocess these documents, such that for a given pattern P, one can efficiently determine which documents contain P. Note that DLP is similar to, but not the same as the original on-line string matching, since it need not list every occurrence of P. Muthukrishnan solved this problem by first building the suffix tree T_{SD} of the concatenated string $T_1\$T_2, \cdots \$T_d$, where \$ denotes the terminate symbol used to concatenate two strings and $\$ \notin \Sigma$. By labeling each leaf nodes in T_{SD} with their document ID, Muthukrishnan reduced the DLP problem to the problem of listing distinct integers in a given subarray. Therefore, assume the overall length of $T_1, T_2, \cdots, T_d$ is n, according to Theorem 3, Muthukrishnan showed that one can preprocess these documents in optimal $O(n)$ time and achieve the searching time $O(|P| + U_d)$, where U_d denotes the number of reported documents. Besides, Muthukrishnan also proposed other valuable results for solving different versions of document retrieval problems[16].

2.3 Wang's Preprocessing for the r-matching Decision Problem

For the r-matching problem, Wang *et al.* gave a preprocessing algorithm with $O(n^3)$ time and space, which answers one query in $O(|P| + U_r)$ time[20]. Besides, they also proposed a preprocessing algorithm for the decision version of the r-matching problem in $O(n^2)$ time and space [20]. Here we only give an introduction to their preprocessing for the r-matching decision problem, which plays an important role in our algorithm. The main idea of Wang's preprocessing is to generate the finite set $\Gamma(T)$ of real scales for the text T, then use these scales to generate all valid patterns and store them in a suffix tree. In the following, we present Wang's results but give the proof with slight modification from the original one, which provides better understanding for our algorithm in the next section.

Lemma 1. [20] *Suppose $T = t_1^{r_1}t_2^{r_2}\cdots t_m^{r_m}$, then we have $\Gamma(T) = \cup_{i=1}^{m}\cup_{j=1}^{r_i}\frac{r_i}{j}$ such that for any given pattern P, P can be real-scale matched in T if and only if there exists any $\alpha \in \Gamma(T)$ such that $\delta_\alpha(P)$ occurs in T.*

Proof: Assume $P = p_1^{s_1}p_2^{s_2}\cdots p_u^{s_u}$ is r-matched at the ith position in T, which maps to the i'th position in T'. That is, the matched substring in T' is $t_{i'}^{r_{i'}}t_{i'+1}^{r_{i'+1}}\cdots t_{i'+u-1}^{r_{i'+u-1}}$. Under this situation, the valid scale of P can be written as $\cap_{j=2}^{u-1}[\frac{r_{i'+j-1}}{s_j}, \frac{r_{i'+j-1}+1}{s_j}) \cap [1, \frac{r_{i'}+1}{s_1}) \cap [1, \frac{r_{i'+u-1}+1}{s_u}) = [\alpha_{low}, \alpha_{up})$, where α_{low} and α_{up} denote the lower and upper bounds of the valid scale, respectively. If $\alpha_{low} = 1$, then it is clear that $\alpha_{low} \in \Gamma(T)$, since $\frac{r_1}{r_1} = 1$ is an element in $\Gamma(T)$. If $\alpha_{low} > 1$, then we have $\alpha_{low} = Max_{j=2}^{u-1}\frac{r_{i'+j-1}}{s_j}$, which is again an element in $\Gamma(T)$ since $s_j \le r_{i'+j-1}$ and $\frac{r_{i'+j-1}}{s_j} \in \Gamma(T)$, for $2 \le j \le u-1$.

Assume P cannot be r-matched at any position in T, then there exists no $\alpha \in [1, \infty)$ such that $\delta_\alpha(P)$ occurs in T. This means there exists no scale $\alpha \in \Gamma(T)$ such that $\delta_\alpha(P)$ occurs in T, since each scale in $\Gamma(T)$ is in the range $[1, \infty)$. Therefore, the lemma holds. □

Based on Lemma 1, Wang *et al.* constructed $|\Gamma(T)|$ strings $T_{\alpha_1}, T_{\alpha_2}, \cdots, T_{\alpha_{|\Gamma(T)|}}$, where each T_{α_k} corresponds to scale α_k in $\Gamma(T)$, for $1 \le k \le |\Gamma(T)|$. Given $T = t_1^{r_1}t_2^{r_2}\cdots t_m^{r_m}$, each T_{α_k} can be generated from T' by doing replacement as follows. Let $f(r_j, \alpha_k) = \beta$ be the largest integer such that $\lfloor \beta\alpha_k \rfloor \le r_j$. For each $t_j^{r_j}$ in T', $1 \le j \le m$, it is replaced by $t_j^{f(r_j,\alpha_k)}$ if $\lfloor f(r_j, \alpha_k)\alpha_k \rfloor = r_j$. Otherwise, $t_j^{r_j}$ is replaced by $t_j^{f(r_j,\alpha_k)}\$t_j^{f(r_j,\alpha_k)}$, where \$ denotes the terminate symbol which does not appear in T. Since $r_j \in Z^+$ and $\alpha_k \ge 1$, $f(r_j, \alpha_k)$ can be computed in constant time by checking two integers $\lceil \frac{r_j}{\alpha_k} \rceil$ and $\lfloor \frac{r_j}{\alpha_k} \rfloor$. Therefore, each T_{α_k} can be constructed in $O(m)$ time. In the following, we keep explaining Wang's idea.

Lemma 2. [20] *For any given pattern P, $\delta_{\alpha_k}(P)$ occurs in T if and only if P occurs in T_{α_k}.*

Proof: Let $P = p_1^{s_1}p_2^{s_2}\cdots p_u^{s_u}$ and $\delta_{\alpha_k}(P) = p_1^{\lfloor s_1\alpha_k \rfloor}p_2^{\lfloor s_2\alpha_k \rfloor}\cdots p_u^{\lfloor s_u\alpha_k \rfloor}$. Assume $\delta_{\alpha_k}(P)$ occurs in T, then there exists at least one substring $T'[i, i+u-1] = t_i^{r_i}t_{i+1}^{r_{i+1}}\cdots t_{i+u-1}^{r_{i+u-1}}$ in T' such that (1)$t_{i+j-1} = p_j$, for $1 \le j \le u$; and (2)$r_i \ge \lfloor s_1\alpha_k \rfloor$, $r_{i+u-1} \ge \lfloor s_u\alpha_k \rfloor$ and $r_{i+j-1} = \lfloor s_j\alpha_k \rfloor$, for $2 \le j \le u-1$. According to the replacement rule, to generate T_{α_k} we have $T'[i, i+u-1]$ be replaced by the substring $p_1^x p_1^{s_1} p_2^{s_2}\cdots p_u^{s_u}p_u^y = p_1^x P p_u^y$, for some integers $x, y \ge 0$. Since $p_1^x P p_u^y$ is a substring in T_{α_k}, we have that P is also a substring in T_{α_k}.

Assume P occurs in T_{α_k}, then $p_1^{s_1}p_2^{s_2}\cdots p_u^{s_u}$ is a substring in T_{α_k}. That is, in T'_{α_k}, which denotes the run-length represented string of T_{α_k}, there exists at least one substring $\hat{P} = p_1^{s_1+x}p_2^{s_2}\cdots p_u^{s_u+y}$, for some integers $x, y \ge 0$. Therefore, we have $\delta_{\alpha_k}(\hat{P}) = p_1^{\lfloor (s_1+x)\alpha_k \rfloor}p_2^{\lfloor s_2\alpha_k \rfloor}\cdots p_u^{\lfloor (s_u+y)\alpha_k \rfloor}$. With the replacement from T' to T_{α_k}, in T', there exists at least one substring $T'[i, i+u-1] = t_i^{r_i}t_{i+1}^{r_{i+1}}\cdots t_{i+u-1}^{r_{i+u-1}}$ such that (1)$t_{i+j-1} = p_j$, for $1 \le j \le u$; and (2)$r_i \ge \lfloor (s_1+x)\alpha_k \rfloor$, $r_{i+u-1} \ge \lfloor (s_u+y)\alpha_k \rfloor$ and $r_{i+j-1} = \lfloor s_j\alpha_k \rfloor$, for $2 \le j \le u-1$. Hence, $\delta_{\alpha_k}(\hat{P})$ is a substring in T, which indicates that $\delta_{\alpha_k}(P)$ is also a substring in T. Therefore, the lemma holds. □

According to Lemmas 1 and 2, one can easily conclude that P can be r-matched in T, if and only if P occurs in the concatenated string $T_R = T_{\alpha_1}\$T_{\alpha_2}\$\cdots\$T_{\alpha_{|\Gamma(T)|}}$. Since $|\Gamma(T)|$ is bounded by n and the construction of each T_{α_k} takes $O(n)$ time and space, the overall time and space for generating the concatenated string T_R would be $O(n^2)$. Therefore, Wang *et al.* stored T_R by using a suffix tree and proposed the following result.

Theorem 4. [20] *Given a pattern P and a text T with $|T| = n$, one can preprocess T with $O(n^2)$ time and space, so that whether $\delta_\alpha(P)$ occurs in T for some real scale $\alpha \ge 1$ can be determined in $O(|P|)$ time.*

Wang's preprocessing for the r-matching decision problem, however, can not be directly applied

to report all matched positions in T, which is resulted from two main flaws. The first flaw happens in the replacement rule, which only enables one to locate matched positions in T', rather than T. The second flaw is the redundancy of reported positions. In other words, each matched position in T' may be reported more than once. This can be easily verified that there are $O(n^2)$ leaf nodes in the suffix tree of T_R, but there are only m mapping positions, where $m \leq n$. To get over these flaws, Wang *et al.* proposed another algorithm, which takes $O(n^3)$ time and space for preprocessing and requires $O(|P|+U_r)$ time for reporting all matched positions in T.

In the next section, we will propose an improved preprocessing on T with $O(n^2)$ time and space, which requires $O(|P|+w)$ time for reporting all matched positions in T, where $w \leq U_r$ denotes the number of dominant positions.

3 An Improved Preprocessing with $O(n^2)$ Time

The main idea of our preprocessing is to combine Wang's preprocessing with the RMQ technique. We start by explaining the idea of *dominant positions*, which enables us to report all matched positions efficiently. The definition of a dominant position in the r-matching problem is given as follows.

Definition 1. *In the r-matching problem, a position i in T is called a dominant position if it is a matched position to r-match P while position $i+1$ is not.*

For a given pattern P, let $DP = \{dp_1, dp_2, \cdots, dp_z\}$ denote the set of dominant positions in T. Since each position in T has its own corresponding position in T', we have $DP' = \{dp'_1, dp'_2, \cdots, dp'_w\}$, $1 \leq w \leq z$, which denotes the corresponding set of DP in T'. The following lemma shows that the mapping between DP and DP' is one-to-one.

Lemma 3. *Given two dominant positions dp_i and dp_j in T, whose corresponding positions in T' are dp'_i and dp'_j, respectively. We have $dp_i = dp_j$ if and only if $dp'_i = dp'_j$.*

Proof: Suppose $P = p_1^{s_1} p_2^{s_2} \cdots p_u^{s_u}$ and $T = t_1^{r_1} t_2^{r_2} \cdots t_m^{r_m}$. Since each position in T' maps to a unique run in T, we shall prove that each run in T contains at most one dominant position. Assume P is r-matched at position i in T', where the corresponding substring in T' can be written as $t_i^{r_i} t_{i+1}^{r_{i+1}} \cdots t_{i+u-1}^{r_{i+u-1}}$. Obviously, the valid scale of P can be written as $\cap_{j=2}^{u-1}[\frac{r_{i+j-1}}{s_j}, \frac{r_{i+j-1}+1}{s_j}) \cap [1, \frac{r_i+1}{s_1}) \cap [1, \frac{r_{i+u-1}+1}{s_u}) = [\alpha_{low}, \alpha_{up})$, where α_{low} and α_{up} denote the lower and upper bounds of the valid scale, respectively.

The matched positions in T derived from $T'[i]$, therefore, are successive indices ranging from $(\sum_{j=1}^{i} r_j - \lceil \alpha_{up} s_1 \rceil + 1)$ to $(\sum_{j=1}^{i} r_j - \lfloor \alpha_{low} s_1 \rfloor)$. According to Definition 1, only one dominant position $(\sum_{j=1}^{i} r_j - \lfloor \alpha_{low} s_1 \rfloor)$ is contained in the run corresponding to $T'[i]$. Hence, the lemma holds. □

From Lemma 3 and its proof, we propose the following lemma, which is a key idea in our algorithm.

Lemma 4. *Given $DP' = \{dp'_1, dp'_2, \cdots, dp'_w\}$ and the corresponding range of valid scales $SB = \{[low_1, up_1), [low_2, up_2), \cdots, [low_w, up_w)\}$, where $[low_i, up_i)$ denotes the range for the valid scale of dp'_i, all matched positions can be determined in $O(w)$ time, in case that a preprocessing with $O(n)$ time on T is allowed.*

Proof: According to the proof of Lemma 3, the set of valid positions can be written as $W = \cup_{i=1}^{w} [\sum_{j=1}^{dp'_i} r_j - \lceil up_i s_1 \rceil + 1, \sum_{j=1}^{dp'_i} r_j - \lfloor low_i s_1 \rfloor]$. With a preprocessing of $O(n)$ time on T, the value of $\sum_{j=1}^{dp'_i} r_j$ can be determined in constant time for any given dp'_i. That is, it takes $O(w)$ time to accomplish the union W. Regardless of the number of matched positions U_r, one can always determine all matched positions by deciding the set W in $O(w)$ time. Therefore, the lemma holds. □

Next, we will propose our preprocessing on T with $O(n^2)$ time, which can be used to determine DP' and SB in $O(|P|+w)$ time. Let T_{SR} be the suffix tree of the concatenated string $T_R = T_{\alpha_1} \$ T_{\alpha_2} \$ \cdots \$ T_{\alpha_{|\Gamma(T)|}}$. Let $L = \{l_1, l_2, \cdots, l_k\}$ be the ordered set of leaf nodes in T_{SR} traversed in lexical order. For each leaf node l_i in L, we *label* it with its corresponding position in T'. In Section 2.3, we know that T_R is directly generated from T', which means each position in T_R can be directly mapped into a position in T'. Therefore, the total time used for labeling would be $O(n^2)$. Let LAB_P be the set of labels located by searching the pattern P in T_{SR}. In the following, we show that $DP' = LAB_P$.

Lemma 5. *Let LAB_P be the set of labels located by searching the pattern P in T_{SR}, we have $DP' = LAB_P$.*

Proof: Suppose $P = p_1^{s_1}p_2^{s_2}\cdots p_u^{s_u}$ and $T = t_1^{r_1}t_2^{r_2}\cdots t_m^{r_m}$. Assume there exists a position $i \in DP'$ such that $i \notin LAB_P$. Since $i \in DP'$, in T' there exists a substring $T'[i, i+u-1] = t_i^{r_i}t_{i+1}^{r_{i+1}}\cdots t_{i+u-1}^{r_{i+u-1}}$ such that $(1)t_{i+j-1} = p_j$, for $1 \le j \le u$. $(2)r_i \ge \lfloor s_1\alpha \rfloor$, $r_{i+u-1} \ge \lfloor s_u\alpha \rfloor$ and $r_{i+j-1} = \lfloor s_j\alpha \rfloor$, for $2 \le j \le u-1$ and some real scale $\alpha \in \Gamma(T)$ (see Lemma 1). According to the replacement rule in Section 2.3, $T'[i, i+u-1]$ would be replaced by the substring $\hat{P} = p_1^{s_1+x}p_2^{s_2}\cdots p_u^{s_u+y}$ for some integers $x, y \ge 0$. Therefore, $\hat{P}$ results in $s_1 + x$ leaf nodes (or suffixes) in T_{SR} which are labeled with i. Since the $(x+1)$-th suffix begins with $P = p_1^{s_1}p_2^{s_2}\cdots p_u^{s_u}$, we have $i \in LAB_P$, which is contradictory to our assumption. Hence, we have $DP' \subseteq LAB_P$.

Similarly, for any given label $i \in LAB_P$, it always corresponds to the substring $T'[i, i+u-1] = t_i^{r_i}t_{i+1}^{r_{i+1}}\cdots t_{i+u-1}^{r_{i+u-1}}$ in T' such that $(1)t_{i+j-1} = p_j$, for $1 \le j \le u$. $(2)r_i \ge \lfloor s_1\alpha \rfloor$, $r_{i+u-1} \ge \lfloor s_u\alpha \rfloor$ and $r_{i+j-1} = \lfloor s_j\alpha \rfloor$, for $2 \le j \le u-1$ and some real scale $\alpha \in \Gamma(T)$. Therefore, we have $i \in DP'$, which means $LAB_P \subseteq DP'$.

Since $DP' \subseteq LAB_P$ and $LAB_P \subseteq DP'$, we obtain that $DP' = LAB_P$ and the lemma holds. □

However, this labeled T_{SR} remains disabled from determining DP' in $O(|P| + w)$ time, since there exist some redundant leaf nodes. To overcome the flaw of redundancy, we show that with an $O(n^2)$-time preprocessing on T_{SR}, one can determine DP' in $O(|P| + w)$ time.

Lemma 6. *With an $O(n^2)$-time preprocessing on T_{SR}, for any given pattern P, one can determine DP' in $O(|P| + w)$ time.*

Proof: Since $L = \{l_1, l_2, \cdots, l_k\}$ is sorted in lexical order, one can store these sorted leaves with an array A_{LAB} of size k, where the set of leaf nodes located by searching P in T_{SR} always corresponds to a subarray $A_{LAB}[i, j]$ in A_{LAB}, for some $1 \le i \le j \le k$. The corresponding pair (i, j) can be easily determined in constant time, in case that T_{SR} has been preprocessed with a traditional bottom-up dynamic programming in linear time. Since each labeled position is an integer bounded by n, based on Theorem 3, one can preprocess A_{LAB} in $O(n^2)$ time, so that all distinct labels within any subarray of A_{LAB} can be determined in optimal $O(U_x)$ time. Because LAB_P denotes the set of labels located by searching P in T_{SR}, it can be seen that all distinct labels in $A_{LAB}[i, j]$ form LAB_P. According to Lemma 5, we have $U_x = |LAB_P| = |DP'| = w$.

Therefore, with an $O(n^2)$-time preprocessing on T_{SR}, one can first search P in T_{SR} with $O(|P|)$ time, then locate $A_{LAB}[i, j]$ in constant time, finally determine DP' in $O(w)$ time. Hence, the searching time is $O(|P|+w)$ and the lemma holds. □

Since DP' can be efficiently determined, our following task is to obtain the set SB in $O(w)$ time, which can be done by using RMQ. For each T_{α_k} in $T_{\alpha_1}\$T_{\alpha_2}\$\cdots\$T_{\alpha_{|\Gamma(T)|}}$, let T'_{α_k} be its run-length represented string. Let $LOW_R = LOW_{\alpha_1}\phi LOW_{\alpha_2}\phi\cdots\phi LOW_{\alpha_{|\Gamma(T)|}}$ and $UP_R = UP_{\alpha_1}\phi UP_{\alpha_2}\phi\cdots\phi UP_{\alpha_{|\Gamma(T)|}}$ be two concatenated arrays of real numbers, where each LOW_{α_k} and UP_{α_k}, $1 \le k \le |\Gamma(T)|$, are of size $|T'_{\alpha_k}|$ and ϕ denotes the real number zero used for concatenation. According to the replacement rule, we know that except for the symbol \$, each character $T'_{\alpha_k}[j] = t_{\alpha_k,j}^{r_{\alpha_k,j}}$ in T'_{α_k} corresponds to a unique character $T'[j'] = t_{j'}^{r_{j'}}$ in T', for $1 \le j \le |T'_{\alpha_k}|$, $1 \le j' \le m$. We assign the elements in LOW_{α_k} and UP_{α_k} as follows. For $1 \le j \le |T'_{\alpha_k}|$, if $t_{\alpha_k,j} \ne \$$, then we assign $LOW_{\alpha_k}[j] = \frac{r_{j'}}{r_{\alpha_k,j}}$ and $UP_{\alpha_k}[j] = \frac{r_{j'}+1}{r_{\alpha_k,j}}$. Otherwise, both $LOW_{\alpha_k}[j]$ and $UP_{\alpha_k}[j]$ are set to zero. The mapping from each $T'_{\alpha_k}[j]$ to $T'[j']$ can be determined in constant time, thus, $O(mn)$ time is required to construct LOW_R and UP_R. Next, we preprocess LOW_R for the range maximum query, while UP_R is preprocessed for the range minimum query. According to Theorems 1 and 2, these preprocesses can also be accomplished in $O(mn)$ time. Finally, for the concatenated string $T_R = T_{\alpha_1}\$T_{\alpha_2}\cdots\$T_{\alpha_{\Gamma(T)}}$, we also preprocess it by using a linear scan with $O(n^2)$ time, which guarantees that each position in T_R can be mapped to a position of its run-length represented string T'_R in constant time.

For any leaf node l_i in T_{SR}, let $POS'(l_i)$ be the position of its corresponding suffix in the run-length represented string T'_R. Also, let $LAB(l_i)$ denote its label corresponding to T'. After searching P in T_{SR}, based on the proof of Lemma 6, in $O(w)$ time one can obtain a set of leaf nodes $\{v_1, v_2, \cdots, v_w\} \subseteq L$ with distinct labels that $LAB(v_i) = dp'_i$, $1 \le i \le w$. Note that in L, there may exist more than one leaf node with the same label dp'_i, for $1 \le i \le w$. In the following, we shall show that for any leaf node labeled with dp'_i and located by searching P in T_{SR}, it can be used to obtain $[low_i, up_i) \in SB$ in constant time. In other words, to obtain SB, it is not necessary to consider the leaf nodes other than $\{v_1, v_2, \cdots, v_w\}$.

Lemma 7. *With a proper $O(n^2)$-time preprocessing, given any leaf node v located by searching P in T_{SR} that $LAB(v) = dp'_i \in DP'$, one can obtain $[low_i, up_i) \in SB$ in constant time.*

Proof: Assume v is an arbitrary leaf node located by searching P in T_{SR} that $LAB(v) = dp'_i \in DP'$. There exists a substring $T'[dp'_i, dp'_i + u - 1]$ in T', such that $(1) t_{dp'_i+j-1} = p_j$, for $1 \le j \le u$. $(2) r_{dp'_i} \ge \lfloor s_1\alpha \rfloor$, $r_{dp'_i+u-1} \ge \lfloor s_u\alpha \rfloor$ and $r_{dp'_i+j-1} = \lfloor s_j\alpha \rfloor$, for $2 \le j \le u-1$ and some real scale $\alpha \in \Gamma(T)$. Therefore, the valid range of real scale for P at position dp'_i in T' can be written as $[low_i, up_i) = \cap_{j=2}^{u-1}[\frac{r_{dp'_i+j-1}}{s_j}, \frac{r_{dp'_i+j-1}+1}{s_j}) \cap [1, \frac{r_{dp'_i}+1}{s_1}) \cap [1, \frac{r_{dp'_i+u-1}+1}{s_u})$
$= [Max_{j=2}^{u-1}\frac{r_{dp'_i+j-1}}{s_j}, Min_{j=2}^{u-1}\frac{r_{dp'_i+j-1}+1}{s_j}) \cap [1, \frac{r_{dp'_i}+1}{s_1}) \cap [1, \frac{r_{dp'_i+u-1}+1}{s_u})$.

Because P occurs at $POS'(v)$ in T'_R, we know that $T'_R[POS'(v), POS'(v)+u-1]$ is a run-length represented substring $\hat{P} = p_1^{s_1+x}p_2^{s_2}\cdots p_u^{s_u+y}$, for some integers $x, y \ge 0$. It is clear that $T'_R[POS'(v)+1, POS'(v)+u-2] = p_2^{s_2}p_3^{s_3}\cdots p_{u-1}^{s_{u-1}}$. Since $LAB(v) = dp'_i$, in LOW_R and UP_R we have $LOW_R[j] = \frac{r_{dp'_i+j-1}}{s_j}$ and $UP_R[j] = \frac{r_{dp'_i+j-1}+1}{s_j}$, for $POS'(v)+1 \le j \le POS'(v)+u-2$. In other words, the range $[Max_{j=2}^{u-1}\frac{r_{dp'_i+j-1}}{s_j}, Min_{j=2}^{u-1}\frac{r_{dp'_i+j-1}+1}{s_j})$ can be obtained by querying the maximum and minimum elements in the subarrays $LOW_R[POS'(v)+1, POS'(v)+u-2]$ and $UP_R[POS'(v)+1, POS'(v)+u-2]$, respectively. By Theorems 1 and 2, the range $[Max_{j=2}^{u-1}\frac{r_{dp'_i+j-1}}{s_j}, Min_{j=2}^{u-1}\frac{r_{dp'_i+j-1}+1}{s_j})$ can always be obtained in constant time. In addition, for any given dp'_i, both $[1, \frac{r_{dp'_i}+1}{s_1})$ and $[1, \frac{r_{dp'_i+u-1}+1}{s_u})$ can also be retrieved in constant time. Therefore, we conclude that $[low_i, up_i)$ can be determined in constant time. □

By applying Lemma 7 to each leaf node $v \in \{v_1, v_2, \cdots, v_w\}$, it is clear that one can obtain SB in $O(w)$ time as follows.

Lemma 8. *With a proper $O(n^2)$-time preprocessing, after searching P in T_{SR} with $O(|P|)$ time, one can obtain SB in $O(w)$ time.*

Therefore, by Lemmas 4, 6 and 8, our main theorem is given as follows.

Theorem 5. *Given a pattern P and a text T with $|T| = n$, one can preprocess T with $O(n^2)$ time and $O(n^2)$ space, so that all matched positions at which $\delta_\alpha(P)$ occurs in T for some real scale $\alpha \ge 1$ can be determined in $O(|P| + w)$ time, where $w \le U_r$ denotes the number of dominant positions and U_r denotes the number of matched positions.*

For large-sized alphabets, which means the number of symbols in Σ is not of constant size, our algorithm can also be implemented by replacing T_{SR} with T_{AR}, where T_{AR} denotes the suffix array of T_R. After an $O(n \log n)$-time conversion of T into a string over the integer alphabet $\{1, 2, \cdots, n\}$, based on the related techniques of suffix array[11, 13], one can construct T_{AR} in $O(n^2)$ time. As a result, for large-sized alphabets both preprocessing time and space can still be kept in $O(n^2)$, while it takes $O(|P| + w + \log n)$ time to determine all matched positions in T for any given pattern P.

To summarize related results, we list various algorithms on the r-matching problem 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. Also, the term "Decision" denotes the required time for the decision problem whether P can be r-matched in T. Finally, we use the term "Position" to represent the time spent on finding all positions where P can be r-matched in T.

4 Implementation for Other Scaling Function

In this section, we shall consider two other important scaling functions, by which we have $\delta_\alpha(P) = p_1^{\lceil \alpha s_1 \rceil}p_2^{\lceil \alpha s_2 \rceil}\cdots p_u^{\lceil \alpha s_u \rceil}$ and $p_1^{\lfloor \alpha s_1 + 0.5 \rfloor}p_2^{\lfloor \alpha s_2 + 0.5 \rfloor}\cdots p_u^{\lfloor \alpha s_u + 0.5 \rfloor}$, respectively. One can easily verify that our preprocessing still works by adapting some of its content. Therefore, in the following, we only give our adaptation, but omit detailed proofs.

For $\delta_\alpha(P) = p_1^{\lceil \alpha s_1 \rceil}p_2^{\lceil \alpha s_2 \rceil}\cdots p_u^{\lceil \alpha s_u \rceil}$, the valid scale that P occurs at the ith position in T' can again be written as $\cap_{j=2}^{u-1}(\frac{r_{i+j-1}-1}{s_j}, \frac{r_{i+j-1}}{s_j}] \cap [1, \frac{r_i}{s_1}] \cap [1, \frac{r_{i+u-1}+1}{s_u}] = (Max_{j=2}^{u-1}\frac{r_{i+j-1}-1}{s_j}, Min_{j=2}^{u-1}\frac{r_{i+j-1}}{s_j}] \cap [1, \frac{r_i}{s_1}] \cap [1, \frac{r_{i+u-1}+1}{s_u}]$. Based on this change, the function $f(r_j, \alpha_k) = \beta$ should also be redefined as the largest integer such that $\lceil \beta\alpha_k \rceil \le r_j$.

In the case $\delta_\alpha(P) = p_1^{\lfloor \alpha s_1 + 0.5 \rfloor}p_2^{\lfloor \alpha s_2 + 0.5 \rfloor}\cdots p_u^{\lfloor \alpha s_u + 0.5 \rfloor}$, for P occurring at the ith position in T', one

Table 1: Preprocessing and searching time on the r-matching problem.

Algorithm	Complexity	Constant-sized Alphabets	Large-sized Alphabets
Amir *et al.*[2]	Time	$O(n+\|P\|)$	$O(n+\|P\|)$
	Space	$O(n+\|P\|)$	$O(n+\|P\|)$
	Decision	$O(n+\|P\|)$	$O(n+\|P\|)$
	Position	$O(n+\|P\|)$	$O(n+\|P\|)$
Wang's method 1[20]	Time	$O(n^3)$	$O(n^3)$
	Space	$O(n^3)$	$O(n^3)$
	Decision	$O(\|P\|)$	$O(\|P\|+\log n)$
	Position	$O(\|P\|+U_r)$	$O(\|P\|+U_r+\log n)$
Wang's method 2[20]	Time	$O(n^2)$	$O(n^2)$
	Space	$O(n^2)$	$O(n^2)$
	Decision	$O(\|P\|)$	$O(\|P\|+\log n)$
	Position	Disabled	Disabled
This paper	Time	$O(n^2)$	$O(n^2)$
	Space	$O(n^2)$	$O(n^2)$
	Decision	$O(\|P\|)$	$O(\|P\|+\log n)$
	Position	$O(\|P\|+w)$	$O(\|P\|+w+\log n)$

can obtain the valid scale with the formula $\cap_{j=2}^{u-1}\lfloor\frac{r_{i+j-1}-0.5}{s_j},\frac{r_{i+j-1}+0.5}{s_j})\cap[1,\frac{r_i+0.5}{s_1})\cap[1,\frac{r_{i+u-1}+0.5}{s_u})=[Max_{j=2}^{u-1}\frac{r_{i+j-1}-0.5}{s_j},Min_{j=2}^{u-1}\frac{r_{i+j-1}+0.5}{s_j})\cap[1,\frac{r_i+0.5}{s_1})\cap[1,\frac{r_{i+u-1}+0.5}{s_u})$. Similarly, the function $f(r_j,\alpha_k)=\beta$ should be redefined as the largest integer such that $\lfloor\beta\alpha_k+0.5\rfloor\le r_j$. Finally, to ensure the correctness of Lemma 1, this time we have $\Gamma(T)=(\cup_{i=1}^{m}\cup_{j=1}^{r_i}\frac{r_i-0.5}{j})\cap[1,\infty]$.

Recently, Amir *et al.*[5] also proposed an $O(n\log|P|+\sqrt{n\log|P|}|P|^{\frac{3}{2}})$-time and $O(n+|P|)$-space algorithm for directly solving the r-matching problem with scaling function $\delta_\alpha(P)=p_1^{\lfloor\alpha s_1+0.5\rfloor}p_2^{\lfloor\alpha s_2+0.5\rfloor}\cdots p_u^{\lfloor\alpha s_u+0.5\rfloor}$, which is considered more precise and realistic in the field of computer vision. In the worst case $O(|P|)=O(n)$, nonetheless, their algorithm takes $O(n^2\sqrt{\log n})$ time, which still reveals the necessity of our $O(n^2)$-time preprocessing.

5 Concluding Remark

In Table 1, one can see that our preprocessing is more efficient than the previous results proposed by Wang *et al.* Our preprocessing is more applicable for longer strings, especially in terms of required space. We also show that the searching time of all matched positions can be improved from $O(|P|+U_r)$ to $O(|P|+w)$, where $w\le U_r$. Therefore, our result for the r-matching problem is more efficient in both preprocessing and searching phases. For other scaling functions, such as rounding to the nearest integer[5], our algorithm can also be applicable.

Compared with the one dimensional real-scale matching problem, the two dimensional real-scale matching (2D r-matching) problem, which is more valuable for image matching, remains worthy of further study. For $\alpha\ge 1$, Amir *et al.*[4] first proposed an algorithm with $O(np^3+n^2p\log p)$ time for solving the 2D r-matching problem, where the text and the pattern are represented by an $n\times n$ matrix and a $p\times p$ matrix, respectively. Up to now, no efficient preprocessing for the 2D r-matching problem has been proposed yet. Therefore, how to extend our preprocessing to the two dimensional case would be an interesting challenge. In addition, further study for the one dimensional real-scale matching would still be necessary. Note that our preprocessing requires $O(n^2)$ space, thus a straightforward implementation could be infeasible for very long texts, such as DNA sequences with longer than 100000 bases. Hence, to devise a space-efficient preprocessing for the one dimensional real scaled matching problem remains an important work in the future.

References

[1] M. I. Abouelhoda, E. Ohlebusch, and S. Kurtz, "Optimal exact string matching based on suffix arrays," *Proceedings of the 9th International Symposium on String Processing and Information Retrieval*, Lisbon, Portugal, pp. 31–43, 2002.

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

[3] A. Amir, G. M. Landau, and U. Vishkin, "Ef-

ficient pattern matching with scaling," *Journal of Algorithms*, Vol. 13, pp. 2–32, 1992.

[4] A. Amir, A. Butman, M. Lewenstein, and E. Porat, "Real two dimensional scaled matching," *Proceedings of the 8th Workshop on Algorithms and Data Structures*, Ottawa, Ontario, Canada, pp. 353–364, 2003.

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

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

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

[8] R. Boyer and J. Moore, "A fast string searching algorithm," *Communications of the ACM*, Vol. 20(10), pp. 762–772, 1977.

[9] P. Clifford and R. Clifford, "Simple deterministic wildcard matching," *Information Processing Letters*, Vol. 101, pp. 53–54, 2007.

[10] D. Harel and R. E. Tarjan, "Fast algorithms for finding nearest common ancestors," *SIAM Journal on Computing*, Vol. 13(2), pp. 338–355, 1984.

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

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

[13] P. Ko and S. Aluru, "Space efficient linear time construction of suffix arrays," *Journal of Discrete Algorithms*, Vol. 3(2-4), pp. 143–156, 2005.

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

[15] E. M. McCreight, "A space-economical suffix tree construction algorithm," *Journal of the ACM*, Vol. 23(2), pp. 262–272, 1976.

[16] S. Muthukrishnan, "Efficient algorithms for document retrieval problems," *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, San Francisco, CA, pp. 657–666, 2002.

[17] Y.-H. Peng, C.-B. Yang, K.-S. Huang, and H.-Y. Ann, "Efficient preprocessings on the one-dimensional discretely scaled pattern matching problem," *Proceedings of the 24th Workshop on Combinatorial Mathematics and Computation Theory*, Puli, Nantou, Taiwan, pp. 35–43, 2007.

[18] B. Schieber and U. Vishkin, "On finding lowest common ancestors: Simplification and parallelization," *SIAM Journal on Computing*, Vol. 17, pp. 1253–1262, 1988.

[19] E. Ukkonen, "On-line construction of suffix trees," *Algorithmica*, Vol. 14, No. 3, pp. 249–260, 1995.

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

[21] P. Weiner, "Linear pattern matching algorithms," *Proceedings of the 14th IEEE Symposium on Switching and Automata Theory*, University of Iowa, pp. 1–11, 1973.