Efficient Preprocessings on the One-dimensional Discretely Scaled Pattern Matching Problem *

Yung-Hsing Peng, Chang-Biau Yang†, Kuo-Si Huang and Hsing-Yen Ann
Department of Computer Science and Engineering
National Sun Yat-sen University, Kaohsiung, Taiwan 80424
†cbyang@cse.nsysu.edu.tw

Abstract

Let P and T be a pattern and a text strings, respectively. The one-dimensional discretely scaled pattern matching problem is to ask for all valid positions in T that some discrete scales of P occur in these positions. Amir et al. first showed that this problem can be solved in $O(n)$ time by adapting Eilam-Tzoreff and Vishkin's algorithm. Recently, Wang et al. showed that when the size of the alphabet in T is finite, it can also be answered in $O(|P|+U_d)$ time with a preprocessing in $O(n \log n)$ time and $O(n \log n)$ space, where U_d denotes the number of reported positions. For integer alphabets and unbounded alphabets, Wang's preprocessing can also be implemented with $O(n \log n)$ time and $O(n \log n)$ space, achieving $O(|P|+U_d+\log n)$ time to report all valid positions. In this paper, we propose the best known preprocessings for the one-dimensional discretely scaled pattern matching problem. For constant-sized alphabets, we propose an optimal preprocessing, which requires $O(n)$ time and reports all valid positions in $O(|P|+U_d)$ time. Also, our preprocessing can be done in $O(n)$ time for integer alphabets while reporting all valid positions in $O(|P| + U_d + \log n)$ time. For unbounded alphabets, we also show that our preprocessing requires $O(n \log n)$ time and $O(n)$ space, and all valid positions can be reported in $O(|P| + U_d + \log n)$ time.

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 by using Knuth-Morris-Pratt algorithm[13] or Boyer-Moore algorithm[6]. In the case of one-dimensional pattern matching, there also exist efficient data structures used to store T, such that P can be located more efficiently in T[20, 15]. In addition, some extended versions, such as approximate string matching[7] or string matching that allows a limited number of mismatches[4], are also problems of great interest. When it comes to multi-dimensional pattern matching, the problem turns out to be more complicated, but more realistic. Take two-dimensional pattern matching for example, some operations such as rotation[10] and scaling[3] can be adopted to make the problem more similar to image matching. Among all related problems of pattern matching, we focus on the one-dimensional discretely scaled pattern matching problem where the pattern P can be discretely scaled.

Let T be a string over the alphabet Σ. Then, $|T| = n$ and $|\Sigma|$ denote the length of T and the number of symbols in Σ, respectively. 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 $1 \le i \le j \le |T|$. 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 $1 \le i \le m$, $t_j \ne t_{j+1}$, $1 \le j \le m-1$ and r_i denotes the run length of t_i. It is clear that one can map each position i in T' to the position $\sum_{j=1}^{i-1} r_j + 1$ in T.

Let $P = p_1^{s_1} p_2^{s_2} \cdots p_u^{s_u}$ be a pattern string. The α-scaling of P, denoted by $\delta_\alpha(P)$, for any integer $\alpha \ge 1$, 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 pattern matching (d-matching) problem* is to find out every valid position i in T such that $\delta_\alpha(P)$ occurs, for some integer $\alpha \ge 1$. In addition, the *d-matching decision problem* is to decide whether a given pattern P can be discretely scaled

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

matched (d-matched) in T. For example, suppose $T = c^6a^2b^3a^4$ and $P = c^2a^1b^1$, one can easily verify that $\delta_2(P) = c^4a^2b^2$ is a substring of T.

The d-matching problem was first defined by Amir *et al.*[3], who also solved this problem in $O(n)$ time by adapting Eilam-Tzoreff and Vishkin's algorithm[8]. For constant-sized alphabets, Wang *et al.*[19] first showed that with $O(n \log n)$ time and space preprocessing on T by using a suffix tree, all d-matched positions of P in T can be determined efficiently in $O(|P| + U_d)$ time, where U_d denotes the number of d-matched positions. Therefore, it is obvious that the d-matching decision problem can also be answered in $O(|P|)$ time after T has already been preprocessed with Wang's algorithm. For large-sized alphabets, such as Chinese text, Wang's preprocessing can also be implemented by first transforming T into another string over $\{1, 2, \cdots, n\}$, then constructing a suffix array over this integer alphabet[14, 12]. Since the symbols in Σ are, in fact, denoted by integers in computer systems, the transformation of T into numeric form over $\{1, 2, \cdots, n\}$ can be done in $O(R + n)$ time, where R denotes the range between the largest and the smallest integers in computer systems. For example, suppose $\Sigma = \{"A", "G", "C", "T"\}$ is an alphabet represented by ASCII code. Here we have R=20, since characters "A" and "T" are encoded by 65 and 84, respectively. In case R is much greater than n, the transformation can still be accomplished in $O(n \log n)$ time, since one can always sort the symbols. Therefore, the algorithm proposed by Wang *et al.* can also be implemented in $O(n \log n)$ time by using a suffix array, which keeps the answering time for large-sized alphabets in $O(|P| + U_d + \log n)$, rather than $O(|P| \log n + U_d)$.

In this paper, we propose two efficient algorithms to preprocess T for the d-matching problem, one is alphabet-dependent that preprocesses T in both $O(|\Sigma|n)$ time and space, while the other is alphabet-independent that preprocesses T in $O(n)$ time and space for integer alphabets. In the first algorithm, we show that when $|\Sigma|$ is constant, the d-matching decision problem can be answered in $O(|P|)$ time, while all d-matched positions can also be determined in $O(|P| + U_d)$ time. In addition, we also show that the answering time can be kept in $O(|P| + U_d + \log n)$ when adopting the second algorithm.

The rest of this paper is organized as follows. We will review some important theorems in Section 2, which are crucial to our algorithms. Then, we will describe our main algorithm in Sections 3 and 4. Finally, we draw our conclusions in Section 5.

2 Preliminaries

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 discovered that the suffix tree can be constructed in $O(|\Sigma|n)$ time[20]. Then, several studies focused on how to build a suffix tree more efficiently[16, 18]. Recently, Farach-Colton *et al.*[9] proved that the suffix tree of T over the integer alphabet $\{1, 2, \cdots, n\}$, abbreviated as $T \in [n]^n$, can be constructed in $O(n)$ time. Therefore, they proposed the following theorem:

Theorem 1. [9] *Given a string $T \in [n]^n$, the suffix tree T_S of T can be deterministically constructed in $O(n)$ time and space.*

According to Theorem 1, one can construct the suffix tree of T in $O(n)$ time, if the transformation from T to a string over $[n]$ can be done in $O(n)$ time. That is, the bottleneck becomes the required time for transformation, which means that if the alphabet is unbounded, it will take $O(n \log n)$ time to construct the suffix tree. Note that it is not necessary to transform the suffix tree over $[n]$ back into Σ, since the lexical order still holds.

Similar to a suffix tree, 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[14, 12]. For unbounded alphabets, however, the constructing time of a suffix array increases to $O(n \log n)$[15]. 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 (*lcp*) of suffixes, to search

a given pattern P in T, one can perform the binary search on T_A, which achieves the answering time $O(|P|+U+\log n)$[15]. For large-sized alphabets, suffix arrays are more applicable than suffix trees for pattern matching, because of its efficiency on required space and answering time. Recently, Abouelhoda *et al.*[1] showed that the answering 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 for any given pair of nodes in Tr, their least common ancestor 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[4].

The first amazing result for the LCA problem was proposed by Harel and Tarjan[11], 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[17] then discovered a simpler algorithm for finding LCA, compared with Harel and Tarjan's result. Recently, Bender and Farach-Colton[5] 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 $[i_1, i_2]$, $1 \le i_1 \le i_2 \le n$, 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[5]. 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 2. [5] *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.*

According to Theorem 2, for a given threshold c, one can easily determine all indices i', $i_1 \le i' \le i_2$, such that $A[i'] \le c$. Let $RLEQ_A(i_1, i_2)$ be the function that reports each index i' in the subarray $A[i_1, i_2]$ such that $A[i'] \le c$. $RLEQ_A(i_1, i_2)$ can be recursively written as follows.

Step 1. Find $i' = RMQ_A(i_1, i_2)$.

Step 2. If $A[i'] > c$, stop. Otherwise, report position i'.

Step 3. Recursively perform $RLEQ_A(i_1, i' - 1)$ and $RLEQ_A(i' + 1, i_2)$.

Obviously, $RLEQ_A(i_1, i_2)$ can be done in $O(U_x)$ time, where U_x denotes the number of reported indices. Therefore, we can draw the following conclusion.

Lemma 1. *Given an array A of n numbers and a threshold c, one can preprocess A in $O(n)$ time, such that for any given interval $[i_1, i_2]$, one can determine all indices i' in the subarray $A[i_1, i_2]$ such that $A[i'] \le c$ in $O(U_x)$ time, where U_x is the number of reported indices.*

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.

Lemma 2. *Given an array A of n numbers and a threshold c, one can preprocess A in $O(n)$ time, such that for any given interval $[i_1, i_2]$, one can determine all indices i' in the subarray $A[i_1, i_2]$ such that $A[i'] \ge c$ in $O(U_x)$ time, where U_x is the number of reported indices.*

2.3 Wang's Preprocessing for the d-matching Problem

In Wang's preprocessing[19] for the d-matching problem, all valid patterns are generated by dividing T by each possible discrete scale $g \in N$. We shall describe the main idea briefly as follows.

Let T be a string over an alphabet Σ, where $T = t_1^{r_1} t_2^{r_2} \cdots t_m^{r_m}$ and $\sum_{i=1}^{m} r_i = n$. Let $r_{max} = \max\{r_1, r_2, \cdots, r_m\}$ and $G = \{1, 2, \cdots, r_{max}\}$. For each $g \in G$, one can transform T into the set of valid patterns T_g, by computing the value r_i/g,

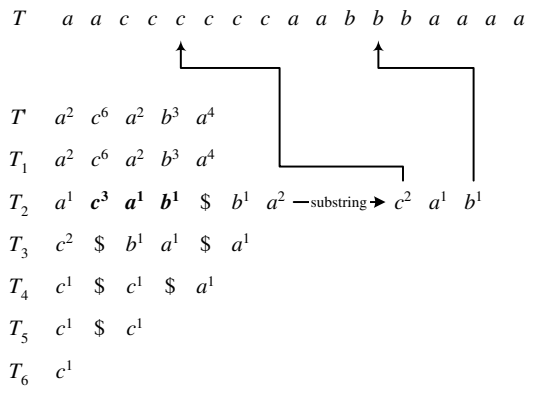

Figure 1: An example for illustrating Wang's algorithm.

$1 \leq i \leq m$. If g is a divisor of r_i, then $t_i^{r_i}$ will be replaced by $t_i^{r_i/g}$. Otherwise, let q be the quotient of r_i/g, $t_i^{r_i}$ will be replaced by $t_i^q\$t_i^q$, where \$ denotes the terminate symbol used to concatenate two valid patterns and $\$ \notin \Sigma$. For example, suppose $T = a^2c^6a^2b^3a^4$, for $g = 2$ we have $T_2 = a^1c^3a^1b^1\$b^1a^2$. To avoid redundant computation, one can associate a list $Life$ of size m on T, which keeps the life span for each $t_i^{r_i}$. Each life span $Life[i]$ in the list is initially set by r_i, which decreases by 1 after finishing one transformation for $t_i^{r_i}$.

With the information of life span, each T_g can be constructed one by one in the order $g = 1, g = 2, \cdots g = r_{max}$, skipping all $t_i^{r_i}$ in T that $r_i < g$. Therefore, the task of building the concatenated string $T_G = T_1\$T_2\$\cdots\$T_{r_{max}}$ under its run-length representation T'_G can be done in $O(n)$ time and space. In addition, it is easy to see that each position in T'_G can map back to a position in T', because T'_G is produced from T'. Take Figure 1 for example, where $T = a^2c^6a^2b^3a^4$, one can see that the substring $c^2a^1b^1$ in T_2 maps to the substring $c^4a^2b^2$, which is a 2-scaled string for $c^2a^1b^1$ in T.

For finding all d-matched positions, Wang *et al.*[19] gave the definition and related results for both *trivial* and *non-trivial patterns* as follows.

Definition 1. [19] *A plain pattern $P = p_1p_2\cdots p_u$ is a trivial pattern if $p_1 = p_2 \cdots = p_u$, that is, $P = p_1^u$. Otherwise, P is a non-trivial pattern.*

Lemma 3. [19] *Let P be a trivial pattern. The set of positions where $\delta_\alpha(P)$ occurs in T is equal to the set of positions where P occurs in T.*

Theorem 3. [19] *Given a non-trivial pattern P and a text T, locating the discretely scaled strings $\delta_\alpha(P)$ in T can be achieved by locating P in T_G.*

According to Lemma 3, for any trivial pattern, one can easily determine all d-matched positions of P in T with $O(|P|)$ time by using the suffix tree of T. For efficiently searching non-trivial pattern P in T_G, Wang *et al.* store T_G in its plain text representation and build the suffix tree of T_G. If T_G is represented in plain text, the total length of valid patterns was proved to be $O(n \log n)$[19]. Therefore, both time and space complexities in their preprocessing would be $O(n \log n)$ since $O(|T_G|) = O(n \log n)$.

In the following sections, we shall show that based on Theorem 1 and Lemma 2, Wang's preprocessing can further be improved.

3 Efficient Preprocessing Dependent on Σ

To reduce the required space, we store T_G with its run-length represented string T'_G, which costs only $O(n)$ space. Given a non-trivial pattern P, whose run-length representation is P', our main task is to design an algorithm such that all positions of $\delta_\alpha(P)$ in T, for some $\alpha \in N$, can be determined by efficiently searching P' in T'_G. Note that both P' and T'_G are run-length represented strings, the matching relationship between run-length represented strings is defined as follows.

Definition 2. *Given two run-length represented strings $P' = p_1^{s_1}p_2^{s_2}\cdots p_u^{s_u}$ and $T' = t_1^{r_1}t_2^{r_2}\cdots t_m^{r_m}$, we say that P' matches T' at position i, if the following two situations hold. (1)$p_j = t_{i+j-1}$, for $1 \leq j \leq u$. (2)$r_i \geq s_1, r_{i+u-1} \geq s_u$, and $s_j = r_{i+j-1}$, for $2 \leq j \leq u-1$.*

Theorem 4. *Given a non-trivial pattern P and a text T, along with their run-length represented strings P' and T', the d-matching problem of P in T can be solved by locating P' in T'_G, where T'_G denotes the concatenated string $T'_1\$T'_2\$\cdots\$T'_{r_{max}}$, which is also run-length represented.*

Proof: According to Definition 2, any run-length represented string P', $|P'| > 1$, occurs at position i in T', if and only if there exists a non-trivial pattern P, which is the plain representation of P', occurring at the position $(\sum_{j=1}^{i} r_j) - s_1 + 1$ in T. Therefore, we conclude that for any non-trivial pattern P, there exists a run-length represented pattern $\delta_\alpha(P') = p_1^{\alpha s_1}p_2^{\alpha s_2}\cdots p_u^{\alpha s_u}$ occurring at

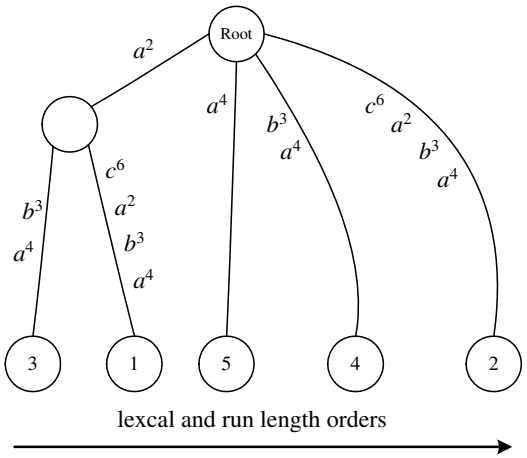

Figure 2: The suffix tree of the run-length represented string $T = a^2c^6a^2b^3a^4$.

position i in T', if and only if the plain representation $\delta_\alpha(P)$ occurs at position $(\sum_{j=1}^{i} r_j)-\alpha s_1+1$ in T. Since T'_G is produced by T', one can easily verify that locating P' in T'_G leads to locating $\delta_\alpha(P')$ in T', which further leads to locating $\delta_\alpha(P)$ in T. Therefore, the d-matching problem of non-trivial P in T can be solved by locating P' in T'_G. □

To efficiently determine all positions of P' in T'_G, we build the suffix tree of T'_G. Note that the suffix tree of a run-length represented string is different from that of a plain one. Here we give an example in Figure 2 to show the suffix tree constructed from a run-length represented string. In this figure, we have $T = a^2c^6a^2b^3a^4$ and the suffix tree of T' is constructed according to both lexical order and run-length order.

For any alphabet Σ, we show that the suffix tree of T'_G can be built in $O(n \log |\Sigma|)$ time.

Lemma 4. *Given an alphabet Σ, the suffix tree of T'_G can be built in $O(n \log |\Sigma|)$ time.*

Proof: Let $\Sigma = \{\sigma_1, \sigma_2, \cdots, \sigma_{|\Sigma|}\}$. For each $\sigma \in \Sigma$, one can find its maximum run length in T by using an array $A_{r_{max}}$ of size $|\Sigma|$. It requires $O(n \log |\Sigma|)$ time to find the maxima of all characters in Σ. Let $M(\sigma)$ denote the maximum run length of symbol σ in T, we can transform the run-length represented characters in T'_G into integers according to the following renaming rule. For each character σ_i^r in T'_G, $1 \le r \le M(\sigma_i)$, it is transformed into the integer $(\sum_{j=1}^{i-1} M(\sigma_j)) + r$. Obviously, this transformation keeps not only the lexical order of Σ, but also the run-length order for characters over a same symbol. That is, for any two run-length characters over a same symbol, written as $\sigma_i^{r_1}$ and $\sigma_i^{r_2}$, we have $\sigma_i^{r_1} \le \sigma_i^{r_2}$ if $r_1 \le r_2$. Also, we have $\sum_{j=1}^{|\Sigma|} M(\sigma_j) \le n$, which means this transformation produces bounded integers over $\{1, 2, \cdots, n\}$. The total time required for transforming T'_G into $[n]$ is $O(n \log |\Sigma|)$, since $O(|T'_G|) = O(n)$ and it takes $O(\log |\Sigma|)$ time to transform a character.

After transforming T'_G into $[n]$, we can construct the suffix tree T'_{GS} of T'_G in $O(n)$ time by using Theorem 1. Therefore, the overall constructing time is $O(n \log |\Sigma|) + O(n) = O(n \log |\Sigma|)$ and the lemma holds. □

For any T'_{GS}, we have the following lemma.

Lemma 5. *For any symbol $\sigma \in \Sigma$, every internal node in T'_{GS} has at most $O(\sqrt{n \log n})$ outgoing edges whose first character is over the symbol σ.*

Proof: For any text T of length n, it can be written in the run-length represented form $T' = t_1^{r_1} t_2^{r_2} \cdots t_m^{r_m}$, where $\sum_{i=1}^{m} r_i = n$. For any subsequence X' in T' carrying the same symbol, written as $X' = x_1^{y_1} x_2^{y_2} \cdots x_h^{y_h}$, where $x_1 = x_2 = \cdots = x_h$ and h denotes the length of X', we have $\sum_{i=1}^{h} y_i \le n$. To maximize the number of different run lengths, we have $y_i \ne y_j$, $i \ne j$. Therefore, the longest sequence for y in ascending order is $1, 2, \cdots, h$, which leads to $1+2+\cdots+h \le n$. Therefore, we conclude that $h = O(\sqrt{n})$, which means for any symbol $\sigma \in \Sigma$, there are at most $O(\sqrt{n})$ different kinds of run lengths over σ in T. Applying this result on T_G, whose length has been known to be $O(n \log n)$[19], we get that for any symbol $\sigma \in \Sigma$ in T_G, there are at most $O(\sqrt{n \log n})$ different run lengths over σ. Hence, the lemma holds. □

Since the definition of a match between P' and T'_G is different from that of P and T_G, directly searching P' in the suffix tree T'_{GS} may not be a good idea. Suppose we have $P' = p_1^{s_1} p_2^{s_2} \cdots p_u^{s_u}$, beginning from the root of T'_{GS}, a direct search will explore all outgoing edges e_i whose first character is of the form p_1^r, $r \ge s_1$. According to Lemma 5 and Definition 2, such a search will take $O(|P|\sqrt{n \log n} + U_d)$ time in worst case. Rather than the direct search, in the following, we shall propose another two preprocessing methods on T'_{GS}, with which we can search P' in T'_{GS} more efficiently.

According to Definition 2, to search P' in T'_{GS}, one can avoid exhaustively searching $p_1^{s_1}$ from the root, if some extra information can be properly stored on the leaf nodes in T'_{GS}. Note that each suffix of T'_G has its position labeled on a leaf node

of T'_{GS}. Let $L = \{l_1, l_2, \cdots, l_k\}$ be the ordered set of leaf nodes in T'_{GS}, where L is sorted by the lexical order of their representative suffixes. Let $Pos(l_i)$ denote the position stored in the leaf l_i, for $1 \leq i \leq k$. For each leaf node (suffix) $l \in L$, we attach its previous character $T'_G[Pos(l) - 1]$, denoted as $Prev(l)$. For a leaf node l without previous character, $Prev(l)$ is simply set by the terminate symbol \$. This preprocessing can be done in $O(n)$ time, because the number of leaf nodes in T'_{GS} is bounded by $O(n)$ and each attachment can be done in constant time.

With the information of previous characters, we can temporarily skip $p_1^{s_1}$ and search for the *remaining pattern* $p_2^{s_2}p_3^{s_3}\cdots p_u^{s_u}$ based on Definition 2. Among all leaf nodes obtained by this search, the leaf nodes carrying information p_1^r with $r \geq s_1$ can be used to locate P' in T'_G. The following three lemmas give more detailed explanation.

Lemma 6. *The searching for the remaining pattern $p_2^{s_2}p_3^{s_3}\cdots p_u^{s_u}$ in T'_{GS} can be done in $O(|P|\log|\Sigma| - s_1)$ time.*

Proof: Except for the final character $p_u^{s_u}$, each character can be located by traversing exactly one outgoing edge. When locating $p_i^{s_i}$ in T'_{GS}, for $2 \leq i < u-1$, two situations need to be considered. The first situation is to locate $p_i^{s_i}$ on an edge, which can be done by comparing $p_i^{s_i}$ with the next character on this edge. The second situation is to locate $p_i^{s_i}$ on an internal node, where all outgoing edges are partitioned into $|\Sigma|$ groups based on the first characters on these edges. This can be done in $O(\log|\Sigma| + \log s_i)$ time, since locating the group of symbol p_i requires $O(\log|\Sigma|)$ time, and then $O(\log s_i)$ time is required for finding the run length s_i with binary search. As for locating the final character $p_u^{s_u}$, it can be done by first identifying the group of p_u in $O(\log|\Sigma|)$ time, and then finding the minimum value of r in p_u^r, $r \geq s_u$, in $O(\log s_u)$ time with binary search. Consequently, the searching time required for the remaining pattern $p_2^{s_2}p_3^{s_3}\cdots p_u^{s_u}$ in T'_{GS} is $O(|P'|\log|\Sigma| + \log s_2 + \log s_3 + \cdots + \log s_u) \leq O(|P'|\log|\Sigma| + s_2 + s_3 + \cdots + s_u) = O(|P'|\log|\Sigma| + |P| - s_1) = O(|P|\log|\Sigma| - s_1)$. Therefore, the lemma holds. □

Among all leaf nodes located by the remaining pattern $p_2^{s_2}p_3^{s_3}\cdots p_u^{s_u}$, we still have to determine which nodes contain the pattern $p_1^{s_1}p_2^{s_2}p_3^{s_3}\cdots p_u^{s_u}$. Before doing so, we adopt an efficient way to represent the located leaf nodes. For a node v in T'_{GS}, let $T'_{GS}(v)$ denote the descendant subtree rooted at v. It is clear that the leaf nodes of $T'_{GS}(v)$ can be represented as $\{l_i, l_{i+1}, \cdots l_j\}$, for some $1 \leq i \leq j \leq k$. Based on this property, each node in T'_{GS} can store all of its leaf nodes by simply using one pair of integers $[i, j]$. With the traditional bottom-up dynamic programming, it takes $O(n)$ time to compute and store the leaf node range for every node in T'_{GS}.

Also, for an outgoing edge e on v, let $Child_e(v)$ denote the child node of v which can be reached by traversing the edge e. To represent all *suspected* leaf nodes located by the remaining pattern $p_2^{s_2}p_3^{s_3}\cdots p_u^{s_u}$, the following two situations need to be considered.

Case 1. $p_u^{s_u}$ is located in exactly one edge e. Let v be the node which has the outgoing edge e, it is obvious that the leaf nodes of $Child_e(v)$ are exactly the leaf nodes located by $p_2^{s_2}p_3^{s_3}\cdots p_u^{s_u}$, which can be represented by using the pair of integers stored in $Child_e(v)$.

Case 2. $p_u^{s_u}$ is located in more than one edge. This happens when we try to locate $p_u^{s_u}$ in an internal node v. Let e_1 and e_2 be the located edges of the least and greatest lexical order, respectively. Let the range stored in $Child_{e_1}(v)$ and $Child_{e_2}(v)$ be $[i_1, j_1]$ and $[i_2, j_2]$, respectively. In this case, the leaf node located by $p_2^{s_2}p_3^{s_3}\cdots p_u^{s_u}$ can be represented by using the pair of integers $[i_1, j_2]$, since l_{i_1} and l_{j_2} are the located leaf nodes with the least and greatest lexical order, respectively.

Therefore, the leaf nodes located by $p_2^{s_2}p_3^{s_3}\cdots p_u^{s_u}$ can be represented by using one pair of integers, which can be computed in constant time once the searching for $p_2^{s_2}p_3^{s_3}\cdots p_u^{s_u}$ is finished. Here we show that the set of *contributive leaf nodes*, which contain the given pattern P', can be determined by using the technique of range maximum queries.

Lemma 7. *Given non-trivial P' and T'_{GS}, with additional $O(|\Sigma|n)$ time preprocessing on T'_{GS}, one can determine all positions of P' in T'_{GS} with $O(|P|\log|\Sigma| + U_d)$ time.*

Proof: Since the range of suspected leaf nodes can be determined by locating the pattern $p_2^{s_2}p_3^{s_3}\cdots p_u^{s_u}$ in T'_{GS}, the following task is to report all contributive leaf nodes located in this suspected range. That is, we desire to find out the leaf nodes whose previous character is of the form p_1^r, $r \geq s_1$. This problem can be solved by using the technique of range maximum queries as follows. We first partitions all leaf nodes of T'_{GS} into $|\Sigma|$

arrays, each of size $O(n)$, according to their previous characters. In other words, if $Prev(l_i) = \sigma^r$, then set $A_\sigma[i] = r$, where A_σ is the array storing the leaf nodes with previous character σ. If $Prev(l_i)$ is not σ, then set $A_\sigma[i] = 0$. Obviously, such a partition can be done with a linear scan in $O(|\Sigma|n)$ time.

According to Theorem 2, one can preprocess each array in $O(n)$ time such that each range maximum query can be answered in constant time. Since there are $|\Sigma|$ arrays, we conclude that the total time complexity of this preprocessing is $O(|\Sigma|n)$. Given $p_1^{s_1}$ and the suspected range $[i_1, i_2]$ located by $p_2^{s_2}p_3^{s_3}\cdots p_u^{s_u}$, we can first locate A_{p_1}, then find out all contributive leaf nodes by using Lemma 2, with $c = s_1$. Obviously, the reporting time would be $O(U_d)$.

When searching P' in T'_{GS}, based on Lemma 6, it takes $O(|P|\log|\Sigma| - s_1)$ time to locate the suspected range $[i_1, i_2]$, $O(\log|\Sigma|)$ time to locate A_{p_1} and $O(U_d)$ time to report the positions. Therefore, the total time required for searching is $O(|P|\log|\Sigma| + U_d)$ and the lemma holds. □

Based on Theorem 4, Lemmas 4 and 7, our main result is given as follows.

Theorem 5. *Let T be a text of length n over an alphabet Σ. One can preprocess T in $O(|\Sigma|n)$ time with $O(|\Sigma|n)$ storage such that all positions at which P is d-matched in T can be reported in $O(|P|\log|\Sigma| + U_d)$ time, where U_d is the number of reported positions.*

By Theorem 5, this preprocessing is optimal in both preprocessing time and searching time, if Σ is of constant size. However, for alphabets of large size, such as the integer alphabet, this preprocessing is not feasible since the preprocessing time and searching time will increase to $O(n^2)$ and $O(|P|\log n + U_d)$, respectively. In the next section, we shall show that the case of large-sized alphabets can be overcome by using a suffix array, which is independent of Σ.

4 Efficient Preprocessing Independent of Σ

In general, there are two typical cases for large-sized alphabets. The first one is the integer alphabet, which is bounded in the form $\{1, 2, \cdots, n\}$, where n denotes the length of the given text T. The second case is the unbounded alphabet whose size is not bounded by $O(n)$. In this section, we shall discuss each of these two cases, which can be overcome by using a suffix array.

For the case of integer alphabet, an array of size n can be used to achieve the transformation as mentioned in the proof of Lemma 4. The transformation can be done in $O(n)$ time, because each symbol can be located in constant time rather than $O(\log|\Sigma|)$ time. Since our transformation of T'_G yields a string over $[n]$, it is obvious that the suffix array of T'_G can be constructed in $O(n)$ time with $O(n)$ space[14, 12]. To search the pattern $p_1^{s_1}p_2^{s_2}\cdots p_u^{s_u}$ in such an array, we can also temporarily skip $p_1^{s_1}$, since each element in this suffix array can also be regarded as a leaf node in the suffix tree T'_{GS}. For locating the pattern $p_2^{s_2}p_3^{s_3}\cdots p_u^{s_u}$, we turn to search for two patterns $p_2^{s_2}p_3^{s_3}\cdots p_u^{s_u}$ and $p_2^{s_2}p_3^{s_3}\cdots p_u^{\infty}$. By doing so, the range of leaf nodes mentioned above can now be determined by using a binary search in the suffix array. In addition, the suffix array can be further partitioned into n arrays for locating $p_1^{s_1}$, where A_{p_1} can be located in constant time. To keep the overall size of these arrays in $O(n)$, we eliminate the zero elements mentioned in Lemma 7. This leads to additional $O(\log n)$ time for locating the suspected range $[i_1, i_2]$ in A_{p_1}. Note that the size of these arrays is bounded by $O(n)$, the total preprocessing time over these arrays for answering range maximal query is also bounded by $O(n)$.

Therefore, this preprocessing can be done in $O(n)$ time with $O(n)$ space. Since it takes $O(|P| + \log n)$ time for searching $p_2^{s_2}p_3^{s_3}\cdots p_u^{s_u}$ in a suffix array, constant time for locating A_{p_1}, $O(\log n)$ time for locating the suspected range $[i_1, i_2]$ in A_{p_1} and finally $O(U_d)$ time to report the answers, our result for the integer alphabet is given as follows.

Theorem 6. *Let T be a text of length n over an integer alphabet $[n]$. One can preprocess T in $O(n)$ time with $O(n)$ storage such that all positions at which P is d-matched in T can be reported in $O(|P| + U_d + \log n)$ time, where U_d is the number of reported positions.*

For unbounded alphabets, recall that the transformation of T into a string over $[n]$ can be done in $O(n\log n)$ time. In this case, it takes $O(\log n)$ time to locate A_{p_1}. Therefore, the searching time of P' in T'_G is still $O(|P| + U_d + \log n)$. The following result for unbounded alphabets can also be easily obtained.

Theorem 7. *Let T be a text of length n over an unbounded alphabet Σ. One can preprocess T in $O(n\log n)$ time with $O(n)$ storage such that all po-*

sitions at which P is d-matched in T can be reported in $O(|P|+U_d+\log n)$ time, where U_d is the number of reported positions.

However, in fact, the symbols can also be sorted in $O(R+n)$ time and space by using an array of size R, where R denotes the range of symbols in T. By using this array, we can also locate A_{p_1} in constant time. Therefore, our preprocessing can also be done in $O(R+n)$ time with $O(R+n)$ space.

Theorem 8. *Let T be a text of length n that the range of symbols in T is R. One can preprocess T in $O(R+n)$ time with $O(R+n)$ storage such that all positions at which P is d-matched in T can be reported in $O(|P|+U_d+\log n)$ time, where U_d is the number of reported positions.*

According to Theorem 8, our preprocessing can still be done in linear time when $R = O(n)$. In addition, for $O(n) < R < O(n \log n)$, the constructing time can be less than $O(n \log n)$, by increasing the required space to $O(R)$.

Note that it takes only $O(n)$ and $O(|P|)$ time to transform T and P into their run-length representation T' and P', respectively. Therefore, Theorems 5, 6, 7 and 8 still hold, even when T' and P' are not directly provided. To summarize our results, we list various algorithms on the d-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 decision time of the problem whether P can be d-matched in T. Finally, we use the term "Position" to represent the time spent on finding all positions where P is d-matched in T.

5 Conclusion

In Table 1, one can see that the data structure in our preprocessing can not only be constructed efficiently, but also achieve efficient time for searching. For constant-sized alphabets, we propose the first-known optimal preprocessing on the d-matching problem. In addition, we also achieve the best-known results for the case of integer alphabets and unbounded alphabets.

Compared with the d-matching problem, the preprocessing of the real scaled string matching problem (r-matching problem), where the scale α is a real number, remains worthy of study. For $\alpha \geq 1$, Amir *et al.*[2] first proposed an $O(n)$ time algorithm for solving the r-matching problem by adapting the KMP algorithm[13]. Recently, Wang *et al.*[19] gave the first preprocessing on the r-matching problem for $\alpha \geq 1$, which takes $O(n^3)$ time and $O(n^3)$ space to achieve the searching time $O(|P|+U_r)$, where U_r denotes the number of positions that P can be r-matched in T. Owing to its high space complexity, Wang's preprocessing on the r-matching problem can only be implemented for short strings. Therefore, to devise an efficient preprocessing on the r-matching problem remains an interesting challenge.

References

[1] M. I. Abouelhoda, E. Ohlebusch, and S. Kurtz, "Optimal exact string matching based on suffix arrays," *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, "Efficient pattern matching with scaling," *Journal of Algorithms*, Vol. 13, pp. 2–32, 1992.

[4] 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.

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

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

[7] R. Cole and R. Hariharan, "Approximate string matching: a simpler faster algorithm," *SIAM Journal on Computing*, Vol. 31(6), pp. 1761–1782, 2002.

[8] T. Eilam-Tzoreff and U. Vishkin, "Matching patterns in a string subject to multi-linear transformation," *Theoretical Computer Science*, Vol. 60, pp. 231–254, 1988.

[9] M. Farach-Colton, P. Ferragina, and S. Muthukrishnan, "On the sorting-complexity of suffix tree construction,"

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

Algorithm	Complexity	Constant Alphabets	Integer Alphabets	Unbounded Alphabets
Amir *et al.*[3]	Time	$O(n)$	$O(n)$	$O(n)$
	Space	$O(n)$	$O(n)$	$O(n)$
	Decision	$O(n)$	$O(n)$	$O(n)$
	Position	$O(n + U_d)$	$O(n + U_d)$	$O(n + U_d)$
Wang *et al.*[19]	Time	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
	Space	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
	Decision	$O(\|P\|)$	$O(\|P\| + \log n)$	$O(\|P\| + \log n)$
	Position	$O(\|P\| + U_d)$	$O(\|P\| + U_d + \log n)$	$O(\|P\| + U_d + \log n)$
This paper	Time	$O(n)$	$O(n)$	$O(n \log n)$ / $O(R + n)$
	Space	$O(n)$	$O(n)$	$O(n)$ / $O(R + n)$
	Decision	$O(\|P\|)$	$O(\|P\| + \log n)$	$O(\|P\| + \log n)$
	Position	$O(\|P\| + U_d)$	$O(\|P\| + U_d + \log n)$	$O(\|P\| + U_d + \log n)$

Journal of the ACM, Vol. 47(6), pp. 987–1011, 2000.

[10] K. Fredriksson and E. Ukkonen, "A rotation invariant filter for two-dimensional string matching," *Proceedings of the 9th Annual Symposium on Combinatorial Pattern Matching (CPM'98), LNCS 1448*, Piscataway, New Jersey, USA, pp. 118–125, 1998.

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

[12] 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.

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

[14] 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.

[15] 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.

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

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

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

[19] 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.

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