Journal of Discrete Algorithms 10 (2012) 5–9

Contents lists available at ScienceDirect

Journal of Discrete Algorithms

www.elsevier.com/locate/jda

Sub-quadratic time and linear space data structures for permutation matching in binary strings

Tanaeem M. Moosa [1], M. Sohel Rahman [*]

AℓEDA Group, Department of CSE, BUET, Dhaka-1000, Bangladesh

A R T I C L E I N F O

Article history:
Available online 11 August 2011

Keywords:
Data structures
Algorithms
Pattern matching
Permutation matching

A B S T R A C T

Given a pattern P of length n and a text T of length m, the permutation matching problem asks whether any permutation of P occurs in T. Indexing a string for permutation matching seems to be quite hard in spite of the existence of a simple non-indexed solution. In this paper, we devise several $o(n^2)$ time data structures for a binary string capable of answering permutation queries in $O(m)$ time. In particular, we first present two $O(n^2/\log n)$ time data structures and then improve the data structure construction time to $O(n^2/\log^2 n)$. The space complexity of the data structures remains linear.

© 2011 Elsevier B.V. All rights reserved.

1. Introduction

In this paper, we study the Permutation Matching problem [7,2,3], which is an interesting but relatively unexplored variant of the Pattern Matching problem.

Problem 1 *(Permutation Matching).* Given two strings T and P we need to find whether any permutation of P occurs in T as a substring.

The indexing version of the problem is defined as follows.

Problem 2 *(Indexed Permutation Matching).* Given a string T which can be preprocessed, we have to answer queries of the form $Query(T, P)$ where $Query(T, P)$ returns true if and only if some permutation of P occurs in T as a substring.

Apart from being very interesting from a theoretical point of view, Permutation Matching problem has practical motivations as well. For example, Burcsi et al. [2] discussed how this problem can be used to solve different scrabble and table arrangement problems. Furthermore, Permutation Matching problem finds its application in molecular biology, notably in interpretation of mass spectrometry data, alignment, SNP discovery, repeated pattern discovery and gene clusters (please see [2] and the references therein). Furthermore, Permutation Matching is a special case of approximate pattern matching and has been used as a filtering step in the approximate pattern matching algorithms [5].

Interestingly, while the solution for Problem 1, i.e., (non-indexed) Permutation Matching is very easy, that for Problem 2, i.e., Indexed Permutation Matching is very difficult [7]. In fact, the Indexed Permutation Matching problem is very hard even for a binary alphabet. And, in this paper, we focus on the Indexed Permutation Matching problem on a binary alphabet.

[*] Corresponding author.
E-mail address: msrahman@cse.buet.ac.bd (M.S. Rahman).
[1] Google Inc., USA.

1570-8667/$ – see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.jda.2011.08.003

Problem 3 *(Indexed Permutation Matching on a Binary Alphabet).* Suppose we are given a string T on a binary alphabet. T can be preprocessed. Now, we have to answer queries of the form $Query(T, P)$ where $Query(T, P)$ returns true if an only if some permutation of P occurs in T as a substring.

In what follows, we will use the following convention: if a solution to a problem has preprocessing time and space complexity $f(n)$ and $h(n)$ respectively and query time $g(n)$, then we say that the algorithm has complexity $\langle (f(n), h(n)), g(n)\rangle$.

To the best of our knowledge, the first published results in the literature on permutation matching is a very recent work of Cicalese et al. [3]. In [3], the authors proposed a linear space data structure which is constructed in a lazy manner, only computing those entries that are needed for the current query, and storing them for future queries. Once the data structure has been completely constructed, any query can be answered in constant time.[2] During the construction phase however, answering queries may take either $O(1)$ or $O(n)$ time. In [3], the authors argued that the expected number of queries needed to completely construct the data structure, i.e., to achieve the capability for constant time query, was $n\ln n$ and the algorithm would require $O(n^2)$ time to answer these $n\ln n$ queries. In any case, in the conclusion, the authors posed the question of whether there existed any sub quadratic time data structure for this problem. Later, Moosa and Rahman [7] and independently Burcsi et al. [2] answered that question, positively, by devising an $O(n^2/\log n)$ data structure to solve the Indexed Permutation Matching problem for binary alphabets. In this paper, we take a step further towards improving the results of [7] and [2]. In particular, using different techniques, we first present two new data structures with complexity $\langle (O(n^2/\log n), O(n)), O(m)\rangle$. Finally, by combining the above techniques we present an improved data structure with preprocessing time $O(n^2/(\log n)^2)$. Notably, our data structures assume word RAM operations.

Notably, despite the name resemblance, the problem studied by Bose et al. [1] and Ibarra [4] is different from our problem. In particular, the above two papers studied the problem of matching two permutations while we want to find whether some permutation of a string is a substring of another string. The rest of the paper is organized as follows. Section 2 presents a few notations and definitions used in this paper while Section 3 discusses the main results. Finally, we briefly conclude in Section 4.

2. Preliminaries

In this section, we present the necessary notions and notations required to present our solution. Let S be a string. We use $|S|$ to represent the length of S. We write $S[i], 0 \leqslant i < \ell = |S|$ to represent the element of S at position i. Note that we use 0 based indexing. If S is a string, then we use $S(a, \ell_1)$ to represent the substring $S[a] \ldots S[a + \ell_1 - 1]$, that is the substring starting from position a and having length ℓ_1. Now, assume that the string S is taken from a binary alphabet $\Sigma = \{0, 1\}$. We define $one(S)$ to be the number of 1's in the string S. Clearly, $one(S)$ is sufficient to represent any permutation of S. In what follows, whenever we refer to a binary alphabet Σ, we will safely assume that $\Sigma = \{0, 1\}$. We will use $maxOne(S(\ell))$ and $minOne(S(\ell))$ to denote, respectively, the maximum and minimum number of 1's in any ℓ-length substring of S.

The $(min, +)$ convolution of two sequences A and B is another sequence C, such that $C_i = min_{1\leqslant j<i}(A_j + B_{i-j})$. The readers may want to compare the definition with that of the usual convolution, also known as $(+, .)$ convolution, which is defined as $C_i = \sum_{1\leqslant j<i}(A_j \times B_{i-j})$.

3. Main results

In this section, we present our main results. Firstly, in Section 3.1, we present a very brief review of the Moosa–Rahman algorithm presented in [7]. Then, in Sections 3.2 and 3.3 we present two new data structures having $O(n^2/\log n)$ preprocessing time. Finally, combining the techniques of Sections 3.2 and 3.3, we present an improved data structure in Section 3.4, which can be constructed in $O(n^2/\log^2 n)$ time. All the data structures presented here require linear space.

3.1. Review of the Moosa–Rahman algorithm of [7]

In this section, we briefly review the work of [7]. Notably, the algorithm of [2] also uses a similar technique. The Moosa–Rahman algorithm [7] utilizes the following lemma from [3].

Lemma 4 *(Interpolation Lemma [3]). If S_1 and S_2 are two substrings of a string S on a binary alphabet Σ such that $\ell = |S_1| = |S_2|$, $i = one(S_1)$, $j = one(S_2)$, $j > i + 1$, then, there exists another substring S_3 such that $\ell = |S_3|$ and $i < one(S_3) < j$.*

The main idea of the Moosa–Rahman algorithm is as follows. For a particular window size ℓ, i.e., for all possible ℓ-length substrings of T, we can compute $maxOne(T(\ell))$ and $minOne(T(\ell))$ in $O(n)$ using the idea of the folklore algorithm [7]. After we have calculated the above for all possible values of ℓ, answering a query is simple using Lemma 4 as follows: first, calculate $p = one(P)$ and then, check whether $minOne(S(|P|)) \leqslant p \leqslant maxOne(S(|P|))$. Since, we need to compute the length and $one(P)$, this requires $O(m)$. As there are $O(n)$ different values of ℓ, total preprocessing time is still $O(n^2)$. Notably

[2] The constant query time assumes that we give the number of 1's as a query instead of the pattern.

however, the size of the data structure is now $O(n)$. To achieve the much desired $o(n^2)$ preprocessing time, Moosa–Rahman algorithm cleverly reduces the above idea to the $(min, +)$ convolution problem. For details, the readers are referred to [7].

3.2. Four-Russian trick

In this section, we describe a new $O(n^2/\log n)$ time data structure employing a different technique. This solution uses the four-Russian trick and requires word RAM operation. Notably the Moosa–Rahman algorithm was free from the above assumption. Here, the basic idea is to calculate $maxOne(T(\ell))$ and $minOne(T(\ell))$ for a specific ℓ in $O(n/\log n)$ time and thereby achieving the $O(n^2/\log n)$ time. The data structure is described below.

We will use two pre-computed tables A of size $2^s \times 2^s$ and B of size 2^s. The table B keeps the number of 1's in a bit pattern of length s. Given a window W of size ℓ, suppose its first s bits are denoted by F and the s bits just following W are denoted by G. Let us denote by W^{+q} the window that we get after shifting W by q positions. Then, the set $\mathcal{W}$ is defined by $\mathcal{W} = \{W^{+i} | 0 \leqslant i < s\}$. Now, let $One(W)$ denote the number of 1's in the window W and $maxOne(\mathcal{W})$ denote the maximum number of 1's in the windows in the set $\mathcal{W}$. Then we assign $A[F, G] = k$ if an only if $maxOne(\mathcal{W}) = One(W^{+0}) + k \equiv One(W) + k$.

In this way we can shift the window by s, and we can update the *maxOne* using table A. The number of one in the current window is calculated using B. By appending s zeros in front of our text, we can calculate $maxOne(T(\ell))$ in $O(n/s)$ for an specific ℓ. Similarly, we can calculate $minOne(T(\ell))$. Since, each entry of the pre-computed tables can be calculated in $O(s)$ time, the total data structure construction time is $O(n^2/s + s4^s)$.

Lemma 5. *The above data structure can be constructed in $O(n^2/\log n)$ time using $O(n)$ space.*

Proof. We can get the desired running time and space complexity by choosing the value of s wisely. If we choose $s = \frac{\log n}{6}$, the running time becomes:

$$\begin{aligned} O\left(n^2/s + s4^s\right) &= O\left(n^2/\log n + (\log n)4^{\frac{\log n}{6}}\right) \\ &= O\left(n^2/\log n + (\log n)2^{\frac{\log n}{3}}\right) \\ &= O\left(n^2/\log n + (\log n)\left(2^{\log n}\right)^{1/3}\right) \\ &= O\left(n^2/\log n + (\log n)(n)^{1/3}\right) \\ &= O\left(n^2/\log n\right) \end{aligned}$$

Similarly, the space complexity becomes:

$$\begin{aligned} O\left(n + 4^s\right) &= O\left(n + 4^{\frac{\log n}{6}}\right) \\ &= O\left(n + 2^{\frac{\log n}{3}}\right) \\ &= O\left(n + \left(2^{\log n}\right)^{1/3}\right) \\ &= O\left(n + n^{1/3}\right) \\ &= O(n) \qquad \square \end{aligned}$$

3.3. Table lookup

In this section, we describe another $O(n^2/\log n)$ time data structure using table lookup. The basic idea is to calculate $maxOne(T(\ell))$ and $minOne(T(\ell))$ for $\ell \in [a \ldots a + \Omega(\log n)]$ in $O(n)$ time and thereby achieving the $O(n^2/\log n)$ running time. We will focus on calculating *maxOne* in this section as *minOne* can be calculated similarly. We will need the following lemmas.

Lemma 6. *The value of $maxOne(T(\ell + 1)) - maxOne(T(\ell))$ is either* 0 *or* 1.

Proof. It is obvious that $maxOne(T(\ell + 1)) - maxOne(T(\ell)) \geqslant 0$. Now we only need to realize that the removal of a character from either side of a window of length $\ell + 1$, can decrease the number of 1's by at most one. $\square$

The following lemma is an immediate consequence of Lemma 6.

Lemma 7. *Any consecutive s entries of maxOne can be represented by the maximum number of $1'$s for the initial position and an s bit number representing the increment.*

Note that Lemma 7 is also true when we have processed the text partially. Now we will show how to preprocess for length $\ell = a, a+1, \ldots, a+s$ in $O(n)$ time. We are moving $s+1$ windows of length $a, a+1, \ldots, a+s$ respectively. The maximum number of $1'$s in these windows can be represented by that in the windows of size a and an s bit number; we will call it the s bit signature. When we shift the windows by one position, the signature of the new windows can be calculated from the old signature in $O(1)$ time using right shift instruction.

Now, to update the s bit *maxOne* signature, all we need is to "merge" the old s bit *maxOne* signature with the signature of the current windows. To merge two signatures a_1, b_1 and a_2, b_2, the significant information is $a_1 - a_2$,b_1 and b_2. Now the value of $a_1 - a_2$ will be greater than or equal to -1 and if it is larger than s the old signature will not change; otherwise we will use a pre-computed table C of size $(s+2) \times 2^s \times 2^s$ to find the new signature.

In this way, we can calculate *maxOne* for windows having s consecutive lengths in $O(n)$. As we need to preprocess for $O(n/s)$ different lengths, the time complexity will be $O(n^2/s + s^2 4^s)$.

Lemma 8. *The above data structure can be constructed in $O(n^2/\log n)$ time and $O(n)$ memory.*

Proof. We can get the desired running time and space complexity by choosing the value of s wisely. By choosing $s = \frac{\log n}{6}$ the running time becomes:

$$\begin{aligned} O\left(n^2/s + s4^s\right) &= O\left(n^2/\log n + (\log n)^2 4^{\frac{\log n}{6}}\right) \\ &= O\left(n^2/\log n + (\log n)^2 2^{\frac{\log n}{3}}\right) \\ &= O\left(n^2/\log n + (\log n)^2 \left(2^{\log n}\right)^{1/3}\right) \\ &= O\left(n^2/\log n + (\log n)^2 (n)^{1/3}\right) \\ &= O\left(n^2/\log n\right) \end{aligned}$$

Similarly the space requirement is:

$$\begin{aligned} O\left(n + s4^s\right) &= O\left(n + (\log n) 4^{\frac{\log n}{6}}\right) \\ &= O\left(n + (\log n) 2^{\frac{\log n}{3}}\right) \\ &= O\left(n + (\log n)\left(2^{\log n}\right)^{1/3}\right) \\ &= O\left(n + (\log n) n^{1/3}\right) \\ &= O(n) \qquad \square \end{aligned}$$

3.4. $O(n^2/\log^2 n)$ preprocessing

In this section, we improve the preprocessing time to $O(n^2/\log^2 n)$ by combining the ideas of Section 3.2 and Section 3.3. We will calculate *maxOne*$(T(\ell))$ for $\ell = a, a+1, \ldots, a+s$ in $O(n/s)$ time and thereby achieving $O((n/s)^2)$ time overall.

More specifically, we now want to shift the $s+1$ windows in s steps. We will show how to update the signatures of windows and *maxOne* in $O(1)$. The signatures of windows can be updated in $O(1)$ time using the table B as described in Section 3.2. Now if we can calculate the signature of maximum number of 1's among these s bit shifts, we can use the table C of Section 3.3 to merge this with the old *maxOne* signature. To calculate the *maxOne* signature of these s bit shifts we can make a table like A of Section 3.2. We will take s bits (denoted by F_1) before the start of a window, s bits (denoted by F_2) from the window and s bits (denoted by G) after the window. We will use a pre-computed table D of size $2^s \times 2^s \times 2^s$, whose $D[F_1, F_2, G]$ entry is a, b if an only if maximum number of 1's is a more than the initial value and b is the corresponding s bit signature. It is easy to see that each entry of the precomputed table can be calculated using a trivial $O(s)$ algorithm. So we can preprocess the whole text in $O((n/s)^2 + s^2 8^s)$ time.

Lemma 9. *The above data structure can be constructed in $O(n^2/\log^2 n)$ time in $O(n)$ space.*

Proof. Again, the desired running time can be achieved by choosing the value of s wisely. By choosing $s = \frac{\log n}{9}$ the running time becomes:

$$\begin{aligned} O\left((n/s)^2 + s^2 8^s\right) &= O\left((n/\log n)^2 + (\log n)^2 8^{\frac{\log n}{9}}\right) \\ &= O\left(n^2/\log^2 n + \log^2 n 2^{\frac{\log n}{3}}\right) \\ &= O\left(n^2/\log^2 n + \log^2 n \left(2^{\log n}\right)^{1/3}\right) \\ &= O\left(n^2/\log^2 n + \log^2 n n^{1/3}\right) \\ &= O\left(n^2/\log^2 n\right) \end{aligned}$$

Similarly space requirement is:

$$\begin{aligned} O\left(n+8^s\right) &= O\left(n+8^{\frac{\log n}{9}}\right) \\ &= O\left(n+2^{\frac{\log n}{3}}\right) \\ &= O\left(n+\left(2^{\log n}\right)^{1/3}\right) \\ &= O\left(n+n^{1/3}\right) \\ &= O(n) \qquad \square \end{aligned}$$

3.5. Query

It is easy to realize from Lemma 4 that, any query can be answered in $O(m)$ time. Notably, the $O(m)$ time complexity arises from the fact that we need to count the length and the number of 1's of the given query pattern. Hence, if the query is given in the form ⟨length, number of 1's⟩, then it can be answered in constant time. This may turn out to be an additional advantage in some applications.

3.6. Lazy construction

Our algorithm can also be modified so that the data structure is constructed in a lazy manner like the lazy algorithm of [3]. More specifically, our data structure can be constructed such that it only computes the entries relevant to current query and storing them for future references. Initially all entries of *maxOne* will be marked as not calculated. When answering a query, if *maxOne* for the current query length is already calculated then the query will take $O(1)$ time. Otherwise we can use the idea of Section 3.4 as follows. Let the length of the current query is a. Then, we will calculate *maxOne* for $\ell = \lfloor a/s \rfloor, \ldots, \lfloor a/s \rfloor + s - 1$ in $O(n/s)$ where $s = \Theta(\log n)$. If we assume that the query length is uniformly distributed then it will take $O(n)$ queries to compute all entries of the table and it will take $O(n^2/\log n)$ time, which is better than the result of [3].

4. Conclusion

In this paper, we have presented several data structures to solve the Indexed Permutation Matching problem for binary alphabet efficiently. We have first presented two data structures with complexity $\langle (O(n^2/\log n), O(n)), O(m) \rangle$ matching the results of [7,2]. Combining the ideas of the above two data structures, we have then devised a data structure with complexity $\langle (O(n^2/\log^2 n), O(n)), O(m) \rangle$. Notably, our data structures assume word RAM operations.

It is an interesting question whether there exists an $\langle (o(n^2/\log^2 n), O(n)), O(m) \rangle$ data structure for permutation matching for binary alphabet. We believe that an $o(n^2/\log^2 n)$ solution to this problem will require significantly different approach than ours.[3]

A difficult challenge is to find an $o(n^2)$ indexing scheme for general alphabet. Notably, the interpolation lemma (Lemma 4) can be extended for general alphabet. However, the extended lemma does not give us the adequate weapon to tackle the general problem. Hence, for general alphabet, it seems to be more challenging to achieve an $o(n^2)$ construction time. Another open problem is to equip the data structure with the capability of finding the occurrences of the patterns in the text.

Acknowledgements

The authors would like to acknowledge the fruitful comments of the anonymous reviewers.

References

[1] Prosenjit Bose, Jonathan F. Buss, Anna Lubiw, Pattern matching for permutations, Inf. Process. Lett. 65 (5) (1998) 277–283.
[2] Peter Burcsi, Ferdinando Cicalese, Gabriele Fici, Zsuzsanna Lipták, On table arrangements, scrabble freaks, and jumbled pattern matching, in: Paolo Boldi, Luisa Gargano (Eds.), FUN, in: Lecture Notes in Computer Science, vol. 6099, Springer, 2010, pp. 89–101.
[3] Ferdinando Cicalese, Gabriele Fici, Zsuzsanna Lipták, Searching for jumbled patterns in strings, in: Jan Holub, Jan Žďárek (Eds.), Proceedings of the Prague Stringology Conference 2009, Czech Technical University in Prague, Czech Republic, 2009, pp. 105–117.
[4] Louis Ibarra, Finding pattern matchings for permutations, Inf. Process. Lett. 61 (6) (1997) 293–295.
[5] Petteri Jokinen, Jorma Tarhio, Esko Ukkonen, A comparison of approximate string matching algorithms, Softw. Pract. Exper. 26 (12) (1996) 1439–1458.
[6] Gad M. Landau, personal communication, 2011.
[7] Tanaeem M. Moosa, M. Sohel Rahman, Indexing permutations for binary strings, Inf. Process. Lett. 110 (18–19) (2010) 795–798.

[3] Notably, during the publication process of this manuscript we were notified about a very recent claim of a data structure with a construction time of $O(n^2/\log^3 n)$ [6]. We are still awaiting the news of its formal publication to study the details.