Information Processing Letters 111 (2011) 318–322

Contents lists available at ScienceDirect

Information Processing Letters

www.elsevier.com/locate/ipl

The indexing for one-dimensional proportionally-scaled strings ☆

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

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

ARTICLE INFO

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

Keywords:
Design of algorithms
String matching
Scale
Proportional

ABSTRACT

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

© 2010 Elsevier B.V. All rights reserved.

1. Introduction

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

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

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

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

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

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

0020-0190/$ – see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.ipl.2010.12.001

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

2. Required techniques

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

2.1. Suffix arrays for integer alphabets

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

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

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

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

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

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

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

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

3. Matching proportionally-scaled patterns

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

3.1. Definition

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

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

Definition 1. Given

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

is a proportionally-scaled pattern of P if

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

Download English Version:

https://daneshyari.com/en/article/428644

Download Persian Version:

https://daneshyari.com/article/428644

Daneshyari.com