Theoretical Computer Science 609 (2016) 129–142

Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

Compressed parameterized pattern matching [☆]

Richard Beal [*], Donald Adjeroh

West Virginia University, Lane Department of Computer Science and Electrical Engineering, Morgantown, WV 26506, United States

ARTICLE INFO

Article history:
Received 29 January 2015
Received in revised form 31 July 2015
Accepted 16 September 2015
Available online 25 September 2015
Communicated by R. Giancarlo

Keywords:
Parameterized matching
Compressed pattern matching
Parameterized string
Lossless compression
Parameterized arithmetic coding
Parameterized border
p-match
p-string
pAC
p-border
Catenate
Tunstall codes
Huffman codes
LZSS

ABSTRACT

Pattern matching between traditional strings is well-defined for both uncompressed and compressed sequences. Prior to this work, parameterized pattern matching (p-matching) was defined predominately by the matching between uncompressed parameterized strings (p-strings) from the constant alphabet Σ and the parameter alphabet Π. In this work, we define the compressed parameterized pattern matching (compressed p-matching) problem to find all of the p-matches between a pattern P and text T, using only P and the compressed text T_c. Initially, we present parameterized compression (p-compression) as a new way to losslessly compress data. Experimentally, we show that p-compression is competitive with various other standard compression schemes. Subsequently, we provide the compression and decompression algorithms. Next, two different approaches are developed to address the compressed p-matching problem: (1) using the recently proposed parameterized arithmetic codes (*pAC*) and (2) using the parameterized *border* array (*p-border*). Our general solution is independent of the underlying compression scheme. The results are further examined for catenate, Tunstall codes, Huffman codes, and LZSS.

1. Introduction and background

Consider a pattern P and a text T. In applications where we wish to detect similar biological sequences [41,13] or identify plagiarism [8,51] between P and T, we can solve the problem with parameterized pattern matching (p-matching) [8]. The current research to support p-matching between a compressed P or T is quite limited. Currently, the work of Apostolico et al. [6,5] introduces the p-match between P and T compressed via run-length encodings. As an application of the parameterized-border array, we also look at a specific case of run-length encoding and p-matching in [16]. In [12], we support p-matching between an uncompressed P and a compressed T for arbitrary compression schemes and show how to apply our results for the example of Tunstall coding; this paper extends the results of [12]. After our original article [12], Garg et al. [22] studied p-matching via a compressed text and compressed pattern using the Word Based Tagged Code [25].

The p-match problem was introduced by Baker [8] as a special form of inexact matching not correctly and efficiently supported by approximate pattern matching schemes. More specifically, the p-match compares strings considering both

[☆] A shorter version of this paper was presented at the Data Compression Conference, Snowbird, UT, 2013 (Beal and Adjeroh [12]). This work was partly supported by grants from the National Historical Publications & Records Commission, WV-EPSCoR RCG, and National Science Foundation (#IIS-1236983).

[*] Corresponding author.
E-mail addresses: r.beal@computer.org (R. Beal), don@csee.wvu.edu (D. Adjeroh).

http://dx.doi.org/10.1016/j.tcs.2015.09.015
0304-3975/© 2015 Elsevier B.V. All rights reserved.

exact symbols from the constant alphabet Σ and potentially inexact symbols from the parameter alphabet Π. The first p-match solutions revolved around the parameterized suffix tree (p-suffix tree) [8], analogous to the traditional suffix tree [42,24]. The p-suffix tree construction time was improved by Baker in [9]. Other contributions to the p-suffix tree include the improved construction in [34] and the randomized algorithms in [18,35,36]. Due to the large memory footprint of the p-suffix tree implementation, the p-match was also solved using extensions of traditional pattern matching schemes by Baker [10] and Amir et al. [4]. The multiple p-match problem is addressed by Idury et al. [29]; the two-dimensional p-match is studied in [40,19]. The parameterized suffix array (p-suffix array), analogous to the traditional suffix array [2,39], was introduced in [27] as a lightweight alternative to the p-suffix tree. We offer theoretical improvements to p-suffix array construction in [15] and study other p-match data structures in [14,17,16].

In this work, we focus on lossless compression. Even though data storage is now readily available in large capacities, Xie et al. [47] highlight the importance of lossless compression beyond saving space, also contributing to storage system performance, such as energy efficiency and access speed. This motivates the need for compressed pattern matching. Denote the length of T_c, the compressed form of T, as n_c and denote the number of occurrences of the m-length P in the n-length T as n_{occ}. Consider the LZ [52,53] family, which is a type of dictionary compression. Navarro et al. [37] perform compressed exact matching in $O(\min\{n, mn_c\} + n_{occ})$ time for LZ77 whereas Kärkkäinen et al. [30] address the compressed approximate matching problem in $O(mkn_c + n_{occ})$ time for LZ78, where k is the permitted number of edits, insertions, and deletions. Matching in LZW compressed text is studied in [23,44]. Other compression schemes have been studied on similar grounds.

A recent report provides a detailed survey on methods for compressed pattern matching for text and images [1]. The relationship between the BWT and pattern matching is detailed in [2]. See [11,48,7] for methods relating to improved BWT compression with variations [38]. Dictionary-based compression is the focus of compressed matching in [32,33]. Variable-to-fixed length coding is studied in [33,49,31,50,46,45]. Compressed pattern matching in multidimensional objects is addressed in [3]. In contrast to dictionary compression schemes, Crochemore et al. [20,21] observe compression via antidictionaries, i.e. collections of substrings that either never or rarely occur in a text. Apostolico et al. [6] address the p-match in terms of fully compressed run-length encodings in $O(n + (r_P \times r_T)\alpha(r_T)\log(r_T))$ time, where α is the inverse of Ackermann's function and r_T and r_P respectively denote the number of runs in the encodings for T and P. In [5], Apostolico et al. provide an alternative solution in $O(r_P \times r_T)$ time. The recent research of Garg et al. [22] focuses on p-matching between texts and patterns compressed with a word-based compression scheme [25]; the time complexity is not analyzed in the paper.

Main contributions: We formally define the compressed parameterized pattern matching (compressed p-matching) problem to find all of the p-matches between a pattern P and text T, using only the uncompressed P and the compressed text T_c. Initially, we introduce parameterized compression (p-compression) as a new way to losslessly compress a text. Experimentally, it is shown that p-compression is competitive with standard lossless compression schemes.

Two different approaches are then developed to address the compressed p-matching problem. Our solutions to the compressed p-match use a partial decompression function ∂, which gives our approaches the flexibility to address solutions where the text T is compressed into T_c using any compression scheme. Each call to the ∂ function will use T_c to sequentially retrieve the next symbol in T, executing in $O(\mu)$ time with $O(\upsilon)$ extra space. First, we apply the recently proposed parameterized arithmetic coding (*pAC*) [15] scheme, i.e. arithmetic coding techniques for prefixes of the dynamic parameterized suffixes, to the compressed p-match problem. This solution however is limited by the practical considerations of the *pAC* [15]. This motivates us to propose a different approach to the compressed p-match via the parameterized border (*p-border*) array, yielding our main result formalized in the following.

Theorem. *Given T_c, the compressed form of the n-length text T, and P, an m-length pattern, compressed p-matching via the p-border can be performed in $O(n\mu)$ time with $O(\max\{m, \upsilon\})$ extra space, where the partial decompression function ∂ executes in $O(\mu)$ time with $O(\upsilon)$ extra space.*

With the recent interest in variable-to-fixed length (VF) coding [33,49,31,50,46], we show how to define the aforementioned partial decompression function ∂ for Tunstall codes, in addition to catenate, Huffman coding, and LZSS. Note that data structures created during compression are not considered extra space for our ∂ algorithm. In the case of Tunstall coding, we show that $\mu, \upsilon \in O(1)$. For Huffman coding, where b-length words are coded and h_W is bth order entropy of W (where W is the n-length encoding for our n-length T), we have $\mu \in O(\frac{h_W}{b})$ and $\upsilon \in O(1)$. We show that LZSS, with a window size (displacement) of Δ, can be implemented such that $\mu \in O(1)$ and $\upsilon \in O(\Delta)$. Also, we show that traditional symbol-based p-matching [10,4] can be achieved with ∂ as catenate, where $\mu, \upsilon \in O(1)$.

Unlike in [6,5] where p-matching via compressed strings is studied for the run-length encodings of T and P, our work differs in that (1) p-matching is performed on an uncompressed P and T_c, a compressed form of T, (2) T_c is a compressed transformation, and (3) our results are applicable for any compression scheme with a defined ∂ function.

2. Preliminaries

A string on an alphabet Σ is a production $T = T[1]T[2]\dots T[n]$ from Σ^n with $n = |T|$ the length of T. In this work, we will assume the use of indexed alphabets, where for an alphabet, say $\mathcal{A} = \{a_1, a_2, \dots, a_{|\mathcal{A}|}\}$, the symbol a_i can be used to index into the ith element of an array. We will use the following string notations: $T[i]$ refers to the ith symbol of string T, $T[i \dots j]$ refers to the substring $T[i]T[i+1]\dots T[j]$, and $T[i \dots n]$ refers to the ith suffix of T: $T[i]T[i+1]\dots T[n]$. The