Linear-Time Off-Line Text Compression by Longest-First Substitution

Shunsuke Inenaga\(^1\), Takashi Funamoto\(^1\), Masayuki Takeda\(^1,2\), and Ayumi Shinohara\(^1,2\)

\(^1\) Department of Informatics, Kyushu University 33, Fukuoka 812-8581, Japan
\{s-ine, t-funa, ayumi, takeda\}@i.kyushu-u.ac.jp
\(^2\) PRESTO, Japan Science and Technology Corporation (JST)

Abstract. Given a text, grammar-based compression is to construct a grammar that generates the text. There are many kinds of text compression techniques of this type. Each compression scheme is categorized as being either off-line or on-line, according to how a text is processed. One representative tactics for off-line compression is to substitute the longest repeated factors of a text with a production rule. In this paper, we present an algorithm that compresses a text basing on this longest-first principle, in linear time. The algorithm employs a suitable index structure for a text, and involves technically efficient operations on the structure.

1 Introduction

Text compression is one of the main stream in the area of string processing \(^4\). The aim of compression is to reduce the size of a given text by efficiently removing the redundancy of the text. Compressing a text enables us to save not only memory space for storage, but also time for transferring the text since its compressed size is now smaller. It is ideal to compress the text as much as possible, but compression in reality has to be done in the trade-off between time and space, i.e. text compression algorithms are also required to have fast performance.

One major scheme of text compression is grammar-based text compression, where a grammar that produces the text is generated. Many attempts to generate a smaller grammar have been made so far, such as in the well-known LZ78 algorithm \(^20\) and the SEQUITUR algorithm \(^14,15\). These two algorithms both process an input text on-line, namely, they read the text in a single pass, and begin to emit compressed output (production rules for a grammar) before they have seen all of the input. Actually, the history of text compression algorithms began with processing texts on-line, since limitation of available memory space has until recently been a big concern. On-line algorithms run on relatively small space by employing the idea of a sliding window, but they only generate a grammar based on replacing the repeating factors in the window that is of bounded size. Therefore some possibilities to compress texts into smaller sizes would remain.
Due to recent hardware developments, we are now allowed to dedicate more memory space to text compression. This gives us opportunities to design off-line algorithms that more efficiently process an input text and give us better compression. Two strategies for seeking for repeating factors in the whole input text are possible; the most-frequent-first and longest-first strategies.

Text compression by the most-frequent-first substitution was first considered by Wolff [19]. His algorithm is, given a text, to recursively replace the most frequently occurring digram (factor of length two) with a new character, which results in a production rule corresponding the digram. Though Wolff’s algorithm takes \( O(n^2) \) time for an input text of length \( n \), Larsson and Moffat [12] devised a clever algorithm, named Re-Pair, that runs in \( O(n) \) time and compresses the text by recursively substituting new characters for the most frequent digram.

In this paper we consider the other one, text compression by the longest first substitution, where we generate a grammar by substituting new characters for the longest repeating factors of a given text of length more than one. For example, from string \( \text{abcacaabaaabcabababcaccabacabcac} \) of length 35 we obtain the following grammar

\[
S \rightarrow AaBaAbBbAcBcA \\
A \rightarrow abcac \\
B \rightarrow aba.
\]

of size 24. Bentley and McIlroy [5] gave an algorithm for this compression scheme, but Nevill-Manning and Witten [16] stated that it does not run in linear time. They also claimed the algorithm by Bentley and McIlroy can be improved so as to run in linear time, but they only noted a too short sketch for how, which is unlikely to give a shape to the idea of the whole algorithm. This paper, therefore, introduces the first explicit, and complete, linear-time algorithm for text compression with the longest-first substitution. The core of our algorithm is the use of suffix trees [18], for they are quite useful for finding the longest repeating factors as is mentioned in [10]. Our algorithm, which is really combinatorial, involves highly technical but necessary update operations on suffix trees towards upcoming substitutions. We give a precise analysis for the time complexity of our algorithm, which results in being linear in the length of an input text string.

2 Preliminaries

2.1 Notations on Strings

Let \( \Sigma \) be a finite alphabet. An element of \( \Sigma^* \) is called a string. Strings \( x, y, \) and \( z \) are said to be a prefix, factor, and suffix of string \( w = xyz \), respectively. The sets of all prefixes, factors, and suffixes of a string \( w \) are denoted by \( \text{Prefix}(w) \), \( \text{Factor}(w) \), and \( \text{Suffix}(w) \), respectively.

The length of a string \( w \) is denoted by \( |w| \). The empty string is denoted by \( \varepsilon \), that is, \( |\varepsilon| = 0 \). Let \( \Sigma^+ = \Sigma^* - \{\varepsilon\} \). The \( i \)-th character of a string \( w \) is denoted...