LZ78 Substring Compression in CDAWG-compressed Space

Hiroki Shibata[*]and Dominik Köppl[†]

Abstract

The Lempel–Ziv 78 (LZ78) factorization is a well-studied technique for data compression. It and its derivates are used in compression formats such as `compress` or `gif`. While most research focuses on the factorization of plain data, not much research has been conducted on indexing the data for fast LZ78 factorization. Here, we study the LZ78 factorization in the substring compression model, where we are allowed to index the data and have to return the factorization of a substring specified at query time. In that model, we propose an algorithm that works in CDAWG-compressed space, computing the factorization with a logarithmic slowdown compared to the optimal time complexity.

Keywords: lossless data compression, LZ78 factorization, substring compression, CDAWG

1 Introduction

Substring compression is to compute the compressed representation of a substring $T[i..j]$ of a preliminary given text T efficiently. This problem was originally stated for the Lempel–Ziv–77 (LZ77) factorization [11]. Recently, several extensions to other compression schemes such as Lempel–Ziv–78 (LZ78) [13] and its derivates have been studied [6, 7].

We focus on computing substring compression in small memory. This is practically important because the input text T can be very large. Given a text T of length n, some studies have proposed methods to compute substring compression in $\Omega(n)$ words. These approaches rely on additional data structures such as suffix trees (ST) [12], in addition to storing T, so they consume more space (in bits) than storing T as is. To reduce the space requirements, we focus on compressed representations of T. However, if we only store a compressed representation, we cannot compute substring compression efficiently. Therefore, a compressed index structure optimized for substring compression is necessary to achieve both time efficiency and space efficiency.

[*]Kyushu University

[†]University of Yamanashi

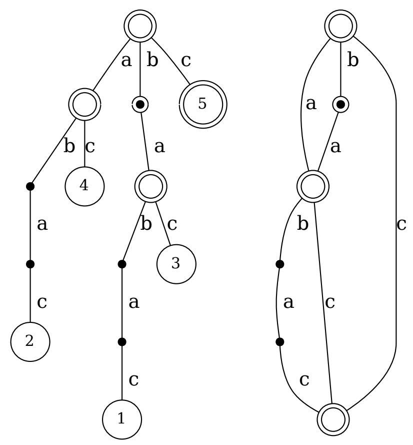

Figure 1: The suffix tree and the CDAWG for T = `babac`. The LZ78 factorization of T is $F_0, F_1, \ldots, F_4 = \epsilon, \mathtt{b}, \mathtt{a}, \mathtt{ba}, \mathtt{c}$. We superimpose the suffix trie and the DAWG on ST and CDAWG, respectively, by drawing implicit nodes with black dots on the edges. We additionally encircle vertices corresponding to LZ78 factors (thus showing explicit nodes as double circles). The CDAWG sink represents the set of strings $\{\mathtt{c}, \mathtt{ac}, \mathtt{bac}, \mathtt{abac}, \mathtt{babac}\}$, which can be read on the root-sink paths. Only a part of these strings are LZ78 factors.

A Compacted Acyclic Word Graph (CDAWG) [3] is a compressed representation that forms a DAG. It is a compact representation of a ST where its isomorphic subtrees are merged. Although suffix trees require $\Theta(n)$ space for all texts, the number of edges e in a CDAWG can become smaller than $\Theta(n)$. While factorizing T, a CDAWG also imitates the ST of T [1]. Specifically, it can restore any position of the text T, a suffix array [8], and an inversed suffix array, in $\mathcal{O}(\log n)$ time.

We propose a memory-efficient algorithm for substring compression of Lempel–Ziv–78 (LZ78) factorization. Our approach works in CDAWG-compressed space and has a multiplicative logarithmic time slowdown compared to the $\mathcal{O}(n)$ time-optimal solution. Furthermore, this method can be easily extended to substring compression of LZ78 derivates such as LZD [5] and LZMW [9].

2 Computing LZ78 in Compressed Space

We first review an *uncompressed* method for LZ78 substring compression proposed in [6]. The process of LZ78 factorization consists of the following two steps: 1) Finding the longest substring $T[l..r]$ that starts from the current position l and matches one of the previously computed factors. 2) Taking a new factor $T[l..r+1]$ by concatinating the previous factor $T[l..r]$ and the character $T[r+1]$. To compute the longest matches efficiently, we superimpose the LZ trie on the ST of T, where the LZ trie is the trie that represents the set of already computed factors. Additionally, we regard the vertex set of the LZ trie as a set of *marks* on the vertices or the edges of the ST. By doing this, computing the longest match can be done by computing the *lowest marked ancestor (LMA)* of a leaf that represents $T[l..n]$ in the ST. Since both computing LMA and marking can be processed in constant time [2, 4], these operations can be performed in constant time for each factor. Thus, the overall time complexity is $\mathcal{O}(z)$ where z is the number of computed factors.

Figure 1 shows an example of the LZ78 factorization with T = `babac`. In the example, we first search the path representing $T = \mathtt{babac}$ on the suffix tree. Since the empty string ϵ is the longest factor on the path, the first factor is $F_1 = \epsilon T[1] = \mathtt{b}$. The second factor is $\mathtt{a}$ because only the empty string is the factor on the path representing $T[2, n] = \mathtt{abac}$. After that, we search the path representing $T[3, n] = \mathtt{bac}$. In this case, the longest factor on the path is $\mathtt{b}$. Therefore, the third factor is $\mathtt{b}T[4] = \mathtt{ba}$. Since the last factor is obviously $\mathtt{c}$, the LZ78 factorization of $\mathtt{babac}$ is $\epsilon, \mathtt{b}, \mathtt{ba}, \mathtt{c}$.

The main idea of computing LZ78 and its derivates by a CDAWG is almost the same as the above method. However, we cannot simply extend it for CDAWGs because a vertex/an edge of a CDAWG may correspond to the multiple vertices/edges in the corresponding ST, respectively (see Figure 1). Therefore, we use another data structure instead of representing the LZ trie with the CDAWG.

We focus on the fact that the LMA query in the above procedure only queries the suffix tree leaves, while internal vertices of the suffix tree are never checked. Let $\mathsf{LMA}[i]$ be the depth of the LMA of the leave that represents $T[i..n]$. By using LMA, we can determine r by $r = l + \mathsf{LMA}[l] - 1$. In the proposed method, we store LMA and update it each time a new factor is computed. Consider a new factor $F = T[l..r + 1]$ of length ℓ that has been computed. In this case, we should update all values $\mathsf{LMA}[i]$ for $T[i..n]$ having F as a prefix to $\max\{\ell, \mathsf{LMA}[i]\}$. In the above condition, the set of such i's forms a range in the suffix array SA. We can compute this range in $\mathcal{O}(\log n)$ time and $\mathcal{O}(e)$ space by using the method of [1] with the following change: Consider a permuted LMA array LMA' that is defined as $\mathsf{LMA}'[i] = \mathsf{LMA}[\mathsf{SA}[i]]$ instead of just LMA. By using LMA', the LMA queries are converted into the following two types of queries: 1) Given $1 \leq a \leq b \leq n$ and ℓ, update $\mathsf{LMA}'[i]$ to $\max\{\mathsf{LMA}'[i], \ell\}$ for all $i \in [a, b]$. 2) Given $1 \leq i \leq n$, return $\mathsf{LMA}'[i]$. This problem is known as *stabbing-max problem*, and it can be solved in $\mathcal{O}(\log n)$ time per query and in $\mathcal{O}(q)$ space where q is the number of queries [10]. Since the time complexity of computing the range in SA is $\mathcal{O}(\log n)$ and computing and updating LMA is $\mathcal{O}(\log n)$, the algorithm takes $\mathcal{O}(\log n)$ time per factor.

The overall time complexity of computing LZ78 substring compression is $\mathcal{O}(z \log n)$ time. On the other hand, the space complexity is $\mathcal{O}(e + z)$ words because CDAWG takes $\mathcal{O}(e)$ words and stabbing-max data structure takes $\mathcal{O}(z)$ words. Our approach can easily be extended for LZD and LZMW compression in the same time and space complexity.

References

[1] D. Belazzougui and F. Cunial. Representing the suffix tree with the CDAWG. In *Proc. CPM*, volume 78 of *LIPIcs*, pages 7:1–7:13, 2017. doi: 10.4230/LIPIcs.CPM.2017.7.

[2] D. Belazzougui, D. Kosolobov, S. J. Puglisi, and R. Raman. Weighted ancestors in suffix trees revisited. In *Proc. CPM*, volume 191 of *LIPIcs*, pages 8:1–8:15, 2021. doi: 10.4230/LIPIcs.CPM.2021.8.

[3] A. Blumer, J. Blumer, D. Haussler, R. M. McConnell, and A. Ehrenfeucht. Complete

inverted files for efficient text retrieval and analysis. *J. ACM*, 34(3):578–595, 1987. doi: 10.1145/28869.28873.

[4] R. Cole and R. Hariharan. Dynamic LCA queries on trees. *SIAM J. Comput.*, 34(4): 894–923, 2005. doi: 10.1137/S0097539700370539.

[5] K. Goto, H. Bannai, S. Inenaga, and M. Takeda. LZD factorization: Simple and practical online grammar compression with variable-to-fixed encoding. In *Proc. CPM*, volume 9133 of *LNCS*, pages 219–230, 2015. doi: 10.1007/978-3-319-19929-0_19.

[6] D. Köppl. Non-overlapping LZ77 factorization and LZ78 substring compression queries with suffix trees. *Algorithms*, 14(2)(44):1–21, 2021. doi: 10.3390/a14020044.

[7] D. Köppl. Computing LZ78-derivates with suffix trees. In *Proc. DCC*, pages 133–142, 2024.

[8] U. Manber and E. W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993. doi: 10.1137/0222058.

[9] V. S. Miller and M. N. Wegman. Variations on a theme by Ziv and Lempel. In *Combinatorial Algorithms on Words*, pages 131–140, Berlin, Heidelberg, 1985.

[10] Y. Nekrich. A dynamic stabbing-max data structure with sub-logarithmic query time. In *Proc. ISAAC*, volume 7074 of *LNCS*, pages 170–179, 2011. doi: 10.1007/978-3-642-25591-5_19.

[11] J. A. Storer and T. G. Szymanski. The macro model for data compression (extended abstract). In *Proc. STOC*, pages 30–39, 1978. doi: 10.1145/800133.804329.

[12] P. Weiner. Linear pattern matching algorithms. In *Proc. SWAT*, pages 1–11, 1973. doi: 10.1109/SWAT.1973.13.

[13] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Trans. Information Theory*, 24(5):530–536, 1978. doi: 10.1109/TIT.1978.1055934.