UNIVERSITÀ
DEGLI STUDI
DI UDINE

Università degli studi di Udine

Fast online Lempel-Ziv factorization in compressed space

Original

Availability:
This version is available http://hdl.handle.net/11390/1068535 since 2021-03-24T12:00:38Z

Publisher:
Springer Verlag

Published
DOI:10.1007/978-3-319-23826-5_2

Terms of use:
The institutional repository of the University of Udine (http://air.uniud.it) is provided by ARIC services. The aim is to enable open access to all the world.

Publisher copyright

(Article begins on next page)

25 April 2024

Fast Online Lempel-Ziv Factorization in Compressed Space

Alberto Policriti[1,2] and Nicola Prezza[1] *

[1] Department of Mathematics and Computer Science, University of Udine, Italy
[2] Institute of Applied Genomics, Udine, Italy

Abstract. Let T be a text of length n on an alphabet Σ of size σ, and let H_0 be the zero-order empirical entropy of T. We show that the LZ77 factorization of T can be computed in $nH_0 + o(n \log \sigma) + \mathcal{O}(\sigma \log n)$ bits of working space with an online algorithm running in $\mathcal{O}(n \log n)$ time. Previous space-efficient online solutions either work in compact space and $\mathcal{O}(n \log n)$ time, or in succinct space and $\mathcal{O}(n \log^3 n)$ time.

Keywords: Lempel-Ziv, compression, BWT

1 Introduction and Related Work

Let $T = a_1 a_2 ... a_{n-1}\$$ be a length-n text on an alphabet Σ of size σ, with $\$ \in \Sigma$ being a symbol appearing only at the end of T (in this work we will implicitly assume that the input text ends with \$). The Lempel-Ziv factorization—LZ77 for brevity—of T [18] is a sequence

$$\mathcal{Z} = \langle pos_1, len_1, c_1 \rangle ... \langle pos_i, len_i, c_i \rangle ... \langle pos_z, len_z, c_z \rangle$$

where $0 \leq pos_i, len_i < n$, $c_i \in \Sigma$ for $i = 1, ..., z$, and:

1. $T = \omega_1 c_1 ... \omega_z c_z$, with $\omega_i = \epsilon$ if $len_i = 0$ and $\omega_i = T[pos_i, ..., pos_i + len_i - 1]$ otherwise.
2. For any $i = 1, ..., z$ with $len_i > 0$, it follows that $pos_i < \sum_{j=1}^{i-1} (len_j + 1)$.
3. For any $i = 1, ..., z$, ω_i must be the *longest* prefix of $\omega_i c_i ... \omega_z c_z$ that occurs in a previous position of T.

The Lempel-Ziv factorization is an important tool in text compression, being its size z closely related with the number of repetitions in the processed string. Moreover, by augmenting it with additional (light) structures, one can obtain fast and high-order compressed full-text indexes [9,12]. Structures based on LZ77 have been shown to be competitive in terms of space on repetitive text collections with respect to BWT-based self indexes [9], and a careful combination of the two techniques stands at the basis of some of the most time-and-space efficient repetition-aware indexes [1].

* Corresponding author: prezza.nicola@spes.uniud.it

The Lempel-Ziv factorization can be computed in linear time and $\mathcal{O}(n \log n)$ bits of working space by using suffix trees or suffix arrays [2,3,7]. Recent results—building up on the FM index [4] data structure—reduced space to *compact* ($\mathcal{O}(n \log \sigma)$ bits), while retaining linear running time [13]. The best space bound to date is achieved by the algorithm discussed in [8], which builds the LZ77 factorization of the text in $\mathcal{O}(n \log^{1+\epsilon} n)$ time ($\epsilon > 0$) and $n(H_k+2)+o(n \log \sigma)$ bits of space (although the $\mathcal{O}(n)$ term prevents space from being *fully* compressed).

A line on this research is focused on the *online* computation of the LZ factorization. Okanohara et al. [14] showed that this task can be carried out in $\mathcal{O}(n \log^3 n)$ time using only $(1 + o(1))n \log \sigma + \mathcal{O}(n)$ bits of working space. Starikovskaya in [16] reduced the running time to $\mathcal{O}(n \log^2 n)$, while slightly increasing the working space to $\mathcal{O}(n \log \sigma)$ bits. Finally, Yamamoto et al. in [17] obtained $\mathcal{O}(n \log n)$ running time within $\mathcal{O}(n \log \sigma)$ bits of working space by using Directed Acyclic Word Graphs (DAWGs).

In this paper, we improve upon the space of all the above discussed solutions by describing an *online* algorithm that computes the LZ77 factorization of a length-n string in $\mathcal{O}(n \log n)$ time using only $nH_0 + o(n \log \sigma) + \mathcal{O}(\sigma \log n)$ bits of working space, H_0 being the empirical zero-order entropy of the input text. If one is interested in computing only the phrase boundaries, then running time can be improved to $\mathcal{O}(n \log n / \log \log n)$. Our basic structure is a dynamic FM index over the reversed text, updated by inserting T-characters from the first to the last.

2 Notation

With *T-, L- and F-positions* we will denote positions on the text T and on the L (last) and F (first) column of the BWT matrix, respectively. Indices start from 0, and we will assume that the text length n and the alphabet $\Sigma = \{0, ..., \sigma - 1\}$ are known beforehand. The only restriction we pose on the alphabet size is $\sigma \leq n$ (which is always true after a re-mapping of the symbols). $BWT(T)$ will denote the Burrows-Wheeler transform of string T, and, when clear from the context, we will refer to it simply as BWT. With $\langle l, r \rangle$ we will denote the right-open BWT interval $[l, r)$. $BWT.F(c),\ c \in \Sigma$, will denote the starting F-position of the block corresponding to character c in the BWT matrix. Letting $W \in \Sigma^*$, the *interval of* W will be the interval $[l, r)$ of rows prefixed by W in the BWT matrix ($r = l$ if W does not occur in T). Letting S be a dynamic string representation on the alphabet Σ, $S[i]$ will indicate the i-th character of S, $S.rank(c, i),\ c \in \Sigma,\ 0 \leq i \leq |S|$ the number of characters equal to c in S before position i *excluded*, and $S.insert(c, i),\ c \in \Sigma,\ 0 \leq i \leq |S|$ the insertion of a character c in S at position i.

3 Fast Online LZ-factorization in Compressed Space

Our result builds upon a recent insight by Navarro and Nekrich on the optimal representation of dynamic strings [11]: there exists a data structure that permits

to represent a sequence $S[0, n-1]$ over an alphabet $\Sigma = \{0, ..., \sigma - 1\}$ in $nH_0 + o(n \log \sigma) + \mathcal{O}(\sigma \log n)$ bits of space and that supports queries (access, rank, select) and updates (insertions and deletions) in $\mathcal{O}(\log n / \log \log n)$ time. The bound is worst-case for the queries and amortized for the updates.

We use the optimal sequence representation of Navarro and Nekrich to build a dynamic FM index taking $nH_0 + o(n \log \sigma) + \mathcal{O}(\sigma \log n)$ bits of space that supports (amortized) $\mathcal{O}(\log n / \log \log n)$-time left-extension of the text with an arbitrary character, $\mathcal{O}(\log n / \log \log n)$-time LF function computation, and $\mathcal{O}(\log^2 n / \log \sigma)$-time locate. Our algorithm scans the text from its first to last character, building the dynamic FM index of the reversed text. At each step (i.e. text character), we (1) update the BWT interval of the current LZ phrase and (2) insert a new text character in the index. Each time the BWT interval becomes empty, we have reached the end of the current LZ phrase and we use a locate query to compute the LZ-factor.

3.1 Dynamic FM Index

The principal component of our dynamic FM index is a dynamic BWT. There is a simple and well-known algorithm that permits to update the Burrows-Wheeler transform $BWT(S)$ of a sequence $S = s_1 s_2 ... s_{u-1} \#$, $\# \notin \Sigma$ being a character [3] lexicographically smaller than all $s \in \Sigma$, by left-extending S with a character $c \in \Sigma$ (see, for example, section 10.3 of [11]). Letting j be such that $BWT(S)[j] = \#$ and $r = BWT(S).rank(c, j)$, we update $BWT(S)$ by:

1) $BWT(S)[j] \leftarrow c$ and
2) $BWT(S).insert(\#, BWT(S).F(c) + r)$.

Let T^R denote the reversed text. In our algorithm, we index the sequence $S = T^R\#$. By using the dynamic sequence representation of [11], we can build $BWT(T^R\#)$ online in overall $\mathcal{O}(n \log n / \log \log n)$ time and $nH_0 + o(n \log \sigma) + \mathcal{O}(\sigma \log n)$ bits of space by inserting characters in the order $\#, T[0], ..., T[n-1]$ with the above procedure. In the following paragraphs, we will denote with BWT the Burrow-Wheeler transform of the current suffix of $S = T^R\#$.

The second ingredient we need in order to compute the LZ77 factorization of T is a dynamic suffix array sampling to support fast locate. The main challenge is to add such functionality without asymptotically increasing space usage. Let $\gamma > 0$ be the sample rate, and $m = \lceil n/\gamma \rceil$ be the number of stored suffix array pointers. To this end, we employ two structures:

1. A compressed dynamic bitvector B to mark with a "1" sampled F-positions.
2. A dynamic sequence representation $SA[0, m-1]$ over the alphabet $[0, n-1]$ taking compact space ($\mathcal{O}(m \log n)$ bits) and supporting $\mathcal{O}(\log n)$-time access and insert operations.

[3] Note that we use two *different* terminator symbols—$\$ \in \Sigma$ and $\# \notin \Sigma$—to mark the end of the forward (LZ77 algorithm) and reverse (BWT algorithm) text, respectively. Our algorithm will therefore work on texts of the form $\#W\$$, $W \in \Sigma^*$.

We use a sample rate of $\gamma = \log_\sigma n \log\log n$. For component (1), we use again the dynamic sequence representation of Navarro and Nekrich. We remind the reader that the size of a zero-order compressed bitvector B' with b bits set is $nH_0(B') \leq b\log(n/b) + b\log e$. Since B has $m = \lceil n/\gamma \rceil = \lceil \frac{n}{\log_\sigma n \log\log n} \rceil$ bits set, it follows easily that B takes overall $nH_0(B) + o(n) + \mathcal{O}(\log n) = o(n)$ bits of space.

For component (2), we use a simple balanced tree (e.g. a red-black tree or a B-tree with constant fanout) where we store suffix array samples in the leafs and we augment each internal node with the size of the corresponding subtree. Access and insert in position i are then implemented by descending the tree according to the subtree-size counters, accessing/inserting the suffix array pointer in the leafs, and (in the case of insert) updating $\mathcal{O}(\log m)$ subtree-size counters. The tree takes overall $\mathcal{O}(m \log n) = o(n \log \sigma)$ bits of space, and access/insert operations take $\mathcal{O}(\log m) = \mathcal{O}(\log n)$ time. Structures B and SA take overall $o(n \log \sigma)$ bits of space.

Implementing *extend* With $BWT.extend(c) \in \{0, ..., |BWT|\}$, $c \in \Sigma \cup \{\#\}$, we will denote the function that:

1. updates the BWT of the current S suffix by left-extending it with a new character c
2. updates the suffix array samples, and
3. returns the L-position of character # *after* the left-extension has taken place.

To avoid updating the already inserted suffix array pointers at each text extension, in structure SA we enumerate S-positions starting from the last. In this sense, $S[n] = \#$ corresponds to SA-position 0, and $S[0]$ corresponds to SA-position n (remember that $|S| = |T^R\#| = n + 1$). Suppose we have built the structures for the length-$(i-1)$ suffix of S and that we want to left-extend it with the new character $S[n - i + 1]$. Let j be such that $BWT[j] = \#$, $r = BWT.rank(S[n - i + 1], j)$, and $k = BWT.F(S[n - i + 1]) + r$. Operation $BWT.extend(S[n - i + 1])$ is implemented as follows:

1. We update BWT with the new text character $S[n - i + 1]$ as described at the beginning of this section.
2. If $i \mod \gamma = 0$, then we insert a new suffix array pointer in SA and mark with a "1" the corresponding F-position in B: $SA.insert(i-1, B.rank(1, k))$ and $B.insert(1, k)$.
3. Otherwise ($i \mod \gamma \neq 0$), we mark with a "0" the new suffix F-position in B: $B.insert(0, k)$.

Step (1) takes $\mathcal{O}(\log n / \log\log n)$ amortized time. The insertion of a bit in B takes $\mathcal{O}(\log n / \log\log n)$ time, and the insertion of a suffix array pointer in SA takes $\mathcal{O}(\log n)$ time. Since we update SA every $\log_\sigma n \log\log n$ left-extensions, *extend* takes overall $\mathcal{O}(\log n / \log\log n)$ amortized time.

Implementing *locate* Let BWT be the Burrows-Wheeler transform of the current S suffix. Operation $BWT.locate(i)$ returns the S-position (enumerated from right to left) corresponding to the F-position i. We implement this operation as usual, i.e. by backward-navigating the current S suffix until a sampled F-position or the first suffix position is found:

1. If i is such that $BWT[i] =' \#'$, then we return $|BWT| - 1$.
2. Otherwise:
   (a) If $B[i] = 1$, then we return $SA[B.rank(1, i)]$.
   (b) If $B[i] = 0$, then we return $BWT.locate(i') - 1$, where $i' = BWT.F(c) + BWT.rank(c, i)$ and $c = BWT[i]$.

Since we use a sample rate of $\log_\sigma n \log\log n$ and access and rank operations on BWT take $\mathcal{O}(\log n/\log\log n)$ time, after $\mathcal{O}(\log^2 n/\log\sigma)$ time we find a marked F-position. Then, extracting the suffix array pointer from structure SA takes $\mathcal{O}(\log n)$ time. Since we assume $\sigma \leq n$, *locate* takes overall $\mathcal{O}(\log^2 n/\log\sigma)$ time.

Implementing LF Function With $BWT.LF(\langle l, r\rangle, c), 0 \leq l < |BWT|,\ 0 \leq r \leq |BWT|,\ c \in \Sigma \cup \{\#\}$, we will denote function LF applied to BWT intervals: if $\langle l, r\rangle$ is the interval of a string $W \in \Sigma^*$ in BWT, then $BWT.LF(\langle l, r\rangle, c)$ returns the interval $\langle l', r'\rangle$ of cW in BWT. LF requires a constant number of rank and access operations on BWT, so it takes overall $\mathcal{O}(\log n/\log\log n)$ time.

3.2 Main Algorithm

The extension step of our algorithm is described in Algorithm 1. The algorithm takes as input one T character c, and outputs either the LZ factor ended by c or nothing if c does not end a factor. In Algorithm 1, variables BWT (the dynamic BWT described in section 3.1), $\langle l, r\rangle$ (right-open BWT interval of the current phrase), *len* (length of the current phrase), and i (L-position of character #) are global, and are initialized at the beginning as $BWT \leftarrow' \#'$, $\langle l, r\rangle \leftarrow \langle 0, 1\rangle$, $len \leftarrow 0$, and $i \leftarrow 0$.

First of all, in line 1 we perform one backward-search step using function LF. The new BWT interval $\langle l', r'\rangle$ is nonempty if and only if the current phrase Wc, $W \in \Sigma^*$, does appear previously in the text. If this is the case (lines 16-19), then we increment the current phrase length (line 17), left-extend the current S suffix (line 18), and update the BWT interval of cW^R (line 19) by incrementing its right bound r'. This step is always needed since in line 18 the new S suffix (prefixed by cW^R) falls inside the closed interval $[l', r']$.

Otherwise, if Wc does not occur previously and $len = |W| > 0$ (lines 2-8), then Wc is a new LZ factor and interval $\langle l, r\rangle$ holds all occurrences of W^R seen until now in the *reversed* text. Notice, however, that $\langle l, r\rangle$ holds also the *current* occurrence of W^R (i.e. $i \in [l, r)$) in addition to at least one previous occurrence (i.e. $r - l \geq 2$). We must therefore be careful to output a *previous* occurrence of W^R: in lines 4-8 we locate either l or $r-1$, depending on which one is different than i. Moreover, we must subtract *len* from the located text position

since *locate* returns an occurrence of W^R in the *reversed* text, and position 0 is reserved for the terminator character #. After locating the occurrence, we can extend the BWT with character c (line 12), reset the BWT interval to the full range $\langle 0, |BWT| \rangle$ (line 13), reset phrase length to zero (line 14), and return the factor.

The last case to consider is when Wc does not occur previously and $len = |W| = 0$ (lines 9 and 10). Then, this is the first occurrence of c in the text and we simply output a factor $\langle null, 0, c \rangle$ after extending the BWT with character c and resetting the global variables as described above (lines 13-14).

Algorithm 1: add_character(c)

input : Character $c \in \Sigma$ (right-extending current T prefix)
output: A factor $\langle pos, len, c \rangle$ if c ends a factor. Nothing otherwise.

```
1  ⟨l', r'⟩ ← BWT.LF(⟨l, r⟩, c);                    /* backward search step */
2  if l' ≥ r' then
3      if len > 0 then
4          if i = l then
5              occ ← r − 1;
6          else
7              occ ← l;
8          P ← BWT.locate(occ) − len;    /* locate a previous occurrence */
9      else
10         P ← null;                              /* first occurrence of c */
11     L ← len;                  /* length of current phrase (c excluded) */
12     BWT.extend(c);                  /* insert character c in the BWT */
13     ⟨l, r⟩ ← ⟨0, |BWT|⟩;                                /* reset interval */
14     len ← 0;                                    /* reset phrase length */
15     return ⟨P, L, c⟩;                                  /* return LZ factor */
16 else
17     len ← len + 1;                 /* increase current phrase length */
18     i ← BWT.extend(c);              /* insert character c in the BWT */
19     ⟨l, r⟩ ← ⟨l', r' + 1⟩;          /* new suffix falls inside [l', r') */
```

From the analysis carried out in section 3.1 it is clear that, excluding *locate*, all steps in Algorithm 1 take (amortized) $\mathcal{O}(\log n / \log\log n)$ time. Notice that we call *locate* once per phrase. It is known that the number z of LZ77 phrases satisfies $z \in \mathcal{O}(n/\log_\sigma n)$ [10]. Since the cost of a single *locate* query is $\mathcal{O}(\log^2 n / \log \sigma)$, in Algorithm 1 *locate* takes $\mathcal{O}(\log n)$ amortized time. We can state our final result:

Theorem 1. *Let $T \in \Sigma^n$. By calling Algorithm 1 on $T[0], ..., T[n-1]$, we build the LZ77 factorization of T online in $nH_0 + o(n \log \sigma) + \mathcal{O}(\sigma \log n)$ bits of working space and $\mathcal{O}(n \log n)$ time.*

Notice that, if we wish to compute only the LZ phrase boundaries, then we do not need *locate*, and the LZ factorization can be built using a simplified version of Algorithm 1 in $\mathcal{O}(n \log n / \log \log n)$ time.

4 Conclusions

In this paper, we presented an online algorithm for computing the LZ77 factorization of a text in $nH_0 + o(n \log \sigma) + \mathcal{O}(\sigma \log n)$ bits of working space and $\mathcal{O}(n \log n)$ time. To our knowledge, ours is the first solution of this problem reaching *fully compressed* working space. Moreover, we obtain this result while being as fast as the fastest online LZ77-construction algorithms described in literature.

Solving this task in small space is of great importance in areas such as LZ-based self-indexing, where computing the LZ77 parse of the text is a spatial bottleneck during index construction. Ideally, it would be desirable being able to solve the problem in $\mathcal{O}(z)$ words of working space (result easily reachable with LZ78), considering that for repetitive text collections z can be *exponentially* smaller than n. One first improvement over our approach could be to obtain high-order compressed space, e.g. by using techniques similar to those employed in [6,11,15]. However, this strategy would still not perform well over highly repetitive text collections—being H_k not sensitive to long repetitions—and being entropy-based techniques usually affected by an $o(n)$ spatial term that could be exponentially larger than z. Alternatively, one could consider using a run-length compressed BWT. Yet, this approach would also require a more sparse SA sampling, which in the most efficient implementations [1,5] is based on the LZ parse itself.

References

1. Belazzougui, D., Cunial, F., Gagie, T., Prezza, N., Raffinot, M.: Composite repetition-aware data structures. In: Proc. CPM. pp. 26–39 (2015)
2. Crochemore, M., Ilie, L.: Computing longest previous factor in linear time and applications. Information Processing Letters 106(2), 75–80 (2008)
3. Crochemore, M., Ilie, L., Smyth, W.F.: A simple algorithm for computing the Lempel-Ziv factorization. In: 18th Data Compression Conference (DCC'08). pp. 482–488. IEEE Computer Society Press, Los Alamitos, CA (2008)
4. Ferragina, P., Manzini, G.: Opportunistic data structures with applications. In: Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on. pp. 390–398. IEEE (2000)
5. Ferragina, P., Manzini, G.: Indexing compressed text. Journal of the ACM (JACM) 52(4), 552–581 (2005)

6. Ferragina, P., Manzini, G., Mäkinen, V., Navarro, G.: An alphabet-friendly fm-index. In: String Processing and Information Retrieval. pp. 150–160. Springer (2004)
7. Kärkkäinen, J., Kempa, D., Puglisi, S.J.: Linear time Lempel-Ziv factorization: Simple, fast, small. In: Combinatorial Pattern Matching. pp. 189–200. Springer (2013)
8. Kreft, S., Navarro, G.: Self-index based on LZ77 (Ph.D. thesis). arXiv preprint arXiv:1112.4578 (2011)
9. Kreft, S., Navarro, G.: Self-indexing based on LZ77. In: Combinatorial Pattern Matching. pp. 41–54. Springer (2011)
10. Lempel, A., Ziv, J.: On the complexity of finite sequences. Information Theory, IEEE Transactions on 22(1), 75–81 (1976)
11. Navarro, G., Nekrich, Y.: Optimal dynamic sequence representations. SIAM Journal on Computing 43(5), 1781–1806 (2014)
12. Navarro, G., Raffinot, M.: Practical and flexible pattern matching over Ziv–Lempel compressed text. Journal of Discrete Algorithms 2(3), 347–371 (2004)
13. Ohlebusch, E., Gog, S.: Lempel-Ziv factorization revisited. In: Combinatorial Pattern Matching. pp. 15–26. Springer (2011)
14. Okanohara, D., Sadakane, K.: An online algorithm for finding the longest previous factors. In: Algorithms-ESA 2008, pp. 696–707. Springer (2008)
15. Policriti, A., Gigante, N., Prezza, N.: Average linear time and compressed space construction of the Burrows-Wheeler transform. In: Language and Automata Theory and Applications, Lecture Notes in Computer Science, vol. 8977, pp. 587–598. Springer International Publishing (2015)
16. Starikovskaya, T.: Computing Lempel-Ziv factorization online. In: Mathematical Foundations of Computer Science 2012, pp. 789–799. Springer (2012)
17. Yamamoto, J., I, T., Bannai, H., Inenaga, S., Takeda, M.: Faster Compact On-Line Lempel-Ziv Factorization. In: 31st International Symposium on Theoretical Aspects of Computer Science (STACS 2014). Leibniz International Proceedings in Informatics (LIPIcs), vol. 25, pp. 675–686. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2014)
18. Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Transactions on information theory 23(3), 337–343 (1977)