Lempel-Ziv Compression in a Sliding Window

Philip Bille*[1], Patrick Hagge Cording†[2], Johannes Fischer[3], and Inge Li Gørtz‡[4]

1 **Technical University of Denmark, DTU Compute, Lyngby, Denmark**
phbi@dtu.dk
2 **Technical University of Denmark, DTU Compute, Lyngby, Denmark**
phaco@dtu.dk
3 **Technische Universität Dortmund, Department of Computer Science, Dortmund, Germany**
johannes.fischer@cs.tu-dortmund.de
4 **Technical University of Denmark, DTU Compute, Lyngby, Denmark**
inge@dtu.dk

Abstract

We present new algorithms for the sliding window Lempel-Ziv (LZ77) problem and the approximate rightmost LZ77 parsing problem.

Our main result is a new and surprisingly simple algorithm that computes the sliding window LZ77 parse in $O(w)$ space and either $O(n)$ expected time or $O(n \log\log w + z \log\log \sigma)$ deterministic time. Here, w is the window size, n is the size of the input string, z is the number of phrases in the parse, and σ is the size of the alphabet. This matches the space and time bounds of previous results while removing constant size restrictions on the alphabet size.

To achieve our result, we combine a simple modification and augmentation of the suffix tree with periodicity properties of sliding windows. We also apply this new technique to obtain an algorithm for the approximate rightmost LZ77 problem that uses $O(n(\log z + \log\log n))$ time and $O(n)$ space and produces a $(1+\epsilon)$-approximation of the rightmost parsing (any constant $\epsilon > 0$). While this does not improve the best known time-space trade-offs for exact rightmost parsing, our algorithm is significantly simpler and exposes a direct connection between sliding window parsing and the approximate rightmost matching problem.

1998 ACM Subject Classification E.4 Coding and Information Theory, E.1 Data Structures, F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases Lempel-Ziv parsing, sliding window, rightmost matching

Digital Object Identifier 10.4230/LIPIcs.CPM.2017.15

1 Introduction

The Lempel-Ziv parsing (LZ77) [36] of a string is a key component in data compression, detecting regularities in strings, pattern matching, and string indexing. LZ77 is the basis for several popular compression tools such as `gzip` and `7zip`, and is shown to compress well under certain measures of compressibility [21].

In general terms, given an input string S of length n, the LZ77 parsing divides S into z substrings $f_1 f_2 \dots f_z$, called *phrases*, in a greedy left-to-right order. The ith phrase f_i

* Supported by the Danish Research Council (DFF – 4005-00267, DFF – 1323-00178).
† Supported by the Danish Research Council (DFF – 4005-00267).
‡ Supported by the Danish Research Council (DFF – 4005-00267, DFF – 1323-00178).

© Philip Bille, Patrick Hagge Cording, Johannes Fischer, and Inge Li Gørtz;
licensed under Creative Commons License CC-BY

starting at position p_i is either (a) the first occurrence of a character in S or (b) the longest substring that has at least one occurrence starting to the left of p_i. To compress S, we can then replace each phrase f_i of type (b) with a pair (r_i, l_i) such that r_i is the distance from p_i to the start of the previous occurrence, and l_i is the length of the previous occurrence. (This is actually the LZ77-variant of Storer and Szymanski [33]; the original one [36] adds a character to each phrase so that it outputs *triples* instead of tuples.)

Computing the LZ77 parse is a very well-studied problem. The simplest way to compute the parse is to build an index for the input string, and scan the string left-to-right looking for the longest prefix of the current suffix occurring to the left of the current position. Using a suffix tree to index the string this yields $O(n)$ time and space algorithm. Research on LZ77 parsing algorithms has since branched into practical and space-efficient computation [4, 12, 14, 16, 17, 19, 20, 22, 27, 30], parallel [31] and external computation [18], online parsing [28, 29, 32, 35], approximation of the parse [10], and algorithms that find the rightmost occurrence of a phrase [1, 8].

Almost all of the existing algorithms maintain an index of the entire input string, whereas almost all practical solutions only maintain a short *window* of length w, for some parameter w, of the input string near the current position in the string. This produces a *sliding window LZ77 parse* [2, 33] that has the property that a previous occurrence of a phrase starts no longer than w characters away from the current position. This limits the number of potential longest matches of a phrase to at most w and also reduces the number of bits needed to encode the reference to f_i. Our main result is a new technique for indexing a sliding window. Using the technique we obtain an algorithm for LZ77 sliding window parsing that improves the previous best known time and space bounds for integer alphabets (and matches the known bounds for constant alphabets). The algorithm is surprisingly simple.

We then turn our heads to rightmost LZ77 parsing. The greedy LZ77 parse is optimal in terms of the number of phrases [6]. However, if we use variable length encoding of the phrases we may reduce the number of bits needed to encode each phrase. By choosing to reference the rightmost occurrence for each phrase we minimize the number of bits needed to encode the greedy LZ77 parse. Though several efficient algorithms for computing the rightmost parse are known, most require highly non-trivial algorithmic techniques. As an interesting application of our technique for sliding window LZ77 parsing, we obtain a very simple efficient *approximate* rightmost parsing algorithm. Interestingly, this algorithm exposes a direct connection between sliding window parsing and the approximate rightmost matching problem.

In the remainder of this section we will formally state the problems, our results, and discuss previous work.

1.1 Sliding Window Parsing

Given a parameter w, the *sliding window LZ77 parse* (SWLZ77) of a string S is the LZ77 parse with the added requirement that the previous occurrence of a phrase f_i starts within distance at most w from the start of f_i. To limit memory consumption, the SWLZ77 parse is used in most compression tools based on LZ77 in practice.

Fiala and Greene [9] and Larsson [24] show how to efficiently maintain the suffix tree of a sliding window of size w. This immediately leads to an algorithm for computing the SWLZ77 parse in $O(n)$ time and $O(w)$ space. However, the algorithms are based on McCreight's [25] and Ukkonen's [34] suffix algorithms, respectively, and thus assume that the size of the alphabet is constant. (The same restriction on the alphabet size holds for the *w-truncated suffix tree* by Na et al.[26].) Furthermore, the algorithms require non-trivial modifications of

the classic suffix tree algorithms and are thus quite complicated. In practice, a hash table is used for strings in the dictionary, often sacrificing optimality for speed (see e.g. [23] for a survey on this).

In this paper we show the following result.

▶ **Theorem 1.** *Let S be a string of length n over an alphabet of size σ. Given a parameter w, we can compute the sliding window Lempel-Ziv parse in*
(i) *$O(w)$ space and $O(n)$ expected time, or*
(ii) *$O(w)$ space and $O(\frac{n}{w}\text{sort}(w,\sigma) + z\log\log\sigma)$ deterministic worst-case time.*
Here, z is number of phrases in the parsing, and $\text{sort}(w,\sigma)$ is the time for sorting w characters from an alphabet of size σ.

Hence, compared to the previous bounds, Theorem 1(i) matches the previous space bounds while achieving linear expected time and with no restrictions on the alphabet size. If we require a deterministic bound, Theorem 1(ii) incurs a small overhead. Plugging in the currently fastest deterministic sorting algorithm [15], which uses $O(w\log\log w)$ time to sort w characters from an arbitrary alphabet, the bound becomes $O(n\log\log w + z\log\log\sigma)$. We note that the additive overhead of $O(z\,\text{loglog}\,\sigma)$ is $O(n)$ for most combinations of σ, n, and z.

The main technical challenges in the result are restricting the search for a previous occurrence to a dynamic window and supporting searches for self-referential phrases of length $> w$ in $O(w)$ space. To achieve this, we present a simple modification and augmentation of suffix trees, which we call w-sliding window trees, that supports linear time searching within a window and show how to exploit periodicity properties of windows to compactly search for long phrases in $O(w)$ space. However, any text indexing data structure that supports basic suffix tree navigation operation can replace the w-sliding window tree in our solution if different time-space trade-offs are required.

1.2 Approximate Rightmost Parsing

Let $\hat{r}_i$ denote the smallest possible choice of reference r_i, i.e., $\hat{r}_i$ is the distance to rightmost substring matching p_i that begins before p_i in S. If $r_i = \hat{r}_i$, $i = 1,\ldots,z$ the parsing is *rightmost* and if $\hat{r}_i \leq r_i \leq c\cdot\hat{r}_i$ for some $c > 1$ the parsing is *c-rightmost*. The *rightmost parsing problem* is to compute the rightmost parsing, and the *approximate rightmost parsing problem* is to compute a c-rightmost parsing for some $c > 1$.

Ferragina et al. [8] present an algorithm for the rightmost parsing problem with running time $O(n(1+\frac{\log\sigma}{\log\log n}))$ and using $O(n)$ space. Recently, this was improved to $O(n(1+\frac{\log\sigma}{\sqrt{\log n}})$ time and $O(n\log\sigma)$ *bits* of space by Belazzougui and Puglisi [1]. Prior to these results, Crochemore et al. [5] presented, to the best of our knowledge, the only approximate rightmost parsing algorithm. Their algorithm runs in $O(n\log n)$ time and $O(n)$ space and it finds the *rightmost equal-cost position* (REP) for each phrase in the greedy LZ77 parse. The REP for a phrase f_i is some occurrence for which r_i requires the same number of bits to encode as $\hat{r}_i$. If $\hat{r}_i$ is a power of 2 the algorithm finds an occurrence where $r_i \leq 2\hat{r}_i - 1$, i.e., roughly speaking their algorithm is producing a 2-rightmost parsing.

All of the above solutions require several highly non-trivial components to achieve their bounds. We show how our solution to the Lempel-Ziv sliding window problem immediately leads to an efficient approximate rightmost parsing algorithm summarized in the following theorem.

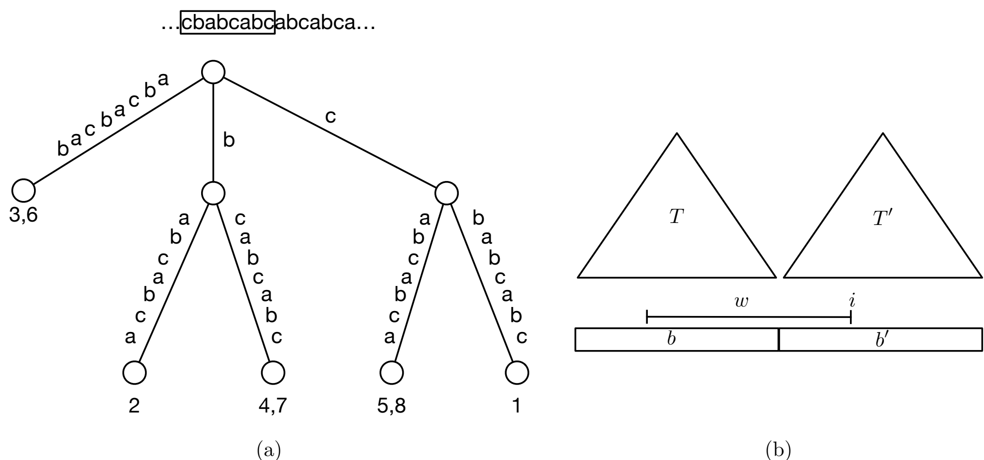

Figure 1 (a) The w-sliding window tree for window size $w = 8$. Text positions at the leaves are relative to the end e of the previous sliding window, implying they must be incremented by e to get absolute positions. (b) Parsing with w-sliding window trees T and T' for blocks b and b'.

▶ **Theorem 2.** *Let S be a string of length n. For any $\varepsilon > 0$ we can compute a $(1+\varepsilon)$-rightmost Lempel-Ziv parsing in $O(n \log z + n \log\log n)$ time and $O(n)$ space, where z is the number of phrases in the parse.*

While our result does not improve the best known trade-offs for rightmost parsing, the algorithm is significantly simpler. It applies our new technique of combining w-sliding window trees and periodicity properties and thereby exposes a direct connection between sliding windows and approximate rightmost matching problems.

2 Lempel-Ziv in a Sliding Window

We now show Theorem 1. Throughout the paper, let S be a string of length n over an alphabet of size σ. We partition S into blocks of size w and parse S from left to right. For these blocks we store a special suffix tree, which we call a *w-sliding window tree.*

▶ **Definition 3** (The w-sliding window tree)**.** The w-sliding window tree of a block is the compact trie of all length w strings starting in the block. Each leaf stores all starting positions of the substring it represents. For each edge e in the w-sliding window trees we store the minimum starting position, $\min(e)$, and the maximum starting position, $\max(e)$, stored in all leaves below it. The w-sliding window tree at position i is the w-sliding window tree for the block starting at position i.

See Figure 1(a) for an example of a w-sliding window tree. We have that the w-sliding window uses $O(w)$ space. Also, given the suffix tree of the string of length $2w$ starting at position i, we can easily build w-sliding window tree starting at position i in $O(w)$ time by truncating all suffixes to length w.

While showing Theorem 1 in the following sections, we will also show the following Lemma that we will need for our approximate rightmost matching algorithm. Given two indices i and j in S, let $\mathrm{lcp}(i,j)$ denote the length of the longest common prefix of $S[i,n]$ and $S[j,n]$.

▶ **Lemma 4.** *Given two w-sliding window trees at position x and $x+w$, respectively, we can find $\ell = \max_{i-w\leq j<i} \mathrm{lcp}(i,j)$ for any $i \in [x+w, x+2w)$ in $O(\ell)$ time (assuming the suffix-trees support constant-time top-down-traversals).*

For simplicity, we first consider the case where the length of each phrase is at most w, and then extend the result to handle arbitrary length phrases.

2.1 Bounded Phrase Length

We first show how to find longest matches if the length of each phrase is at most w. We partition S into blocks of size w and parse S from left-to-right. We only maintain the last two blocks in memory.

2.1.1 Parsing

We implement the sliding window parsing using the w-sliding window trees as follows. See also Figure 1(b). Suppose we have parsed the first $i-1$ characters of S and currently have the w-sliding window trees T and T' for the last two blocks b and b' stored. To compute the phrase starting at position i, we traverse T and T' top-down according to the substring starting at position i. In T, we compare i to $\max(e)$ each time we follow edge e. If $\max(e)$ is within the window (if $\max(e) \geq i - w$) we continue the search and otherwise we stop the search. If we reach a leaf we also stop. When the search stops, we output $\max(e)$ of the previous edge e as the starting position of the longest match in T. In T' we compare with $\min(e)$ in the same fashion. That is, we only continue the search if $\min(e)$ is smaller than i. We output $\min(e)$ of the previous edge e as the starting position of the longest match in T'. We return the maximum of the longest matching path found in T and T' as the longest matching substring within the window.

2.1.2 Correctness

We argue that the algorithm correctly finds a longest match. A longest match within the window must start in one of the two blocks b or b'. Since we only continue the search in T as long as $\max(e)$ is in the window, the match that we found starts at a position in the window. Similarly for T'.

2.2 Unbounded Phrase Length

We now consider the general case of unbounded phrase length and show how to extend the solution from the previous section to handle this case by exploiting a periodicity property of the sliding window [3].

Given a w-sliding window tree we now might reach a leaf, from where we need to continue the matching further. If there is only one substring stored at the leaf we can simply continue matching the corresponding substrings in S until the phrase ends. Unfortunately, we may have multiple strings stored at a leaf and thus we cannot afford naively matching against these.

We modify searching for longest match starting at position i as follows. We match in the w-sliding window trees T and T' just as before. If we reach a leaf in T we pick the maximum starting position x stored in the leaf ($x = \max(e)$ if e is the incoming edge) and continue matching from position $i + w$ and $x + w$ until we get a mismatch. If we reach a leaf in T' we pick the minimum starting position stored in the leaf (using $\min(e)$) and continue matching in the same fashion.

2.2.1 Correctness and Analysis

To show that the modification works correctly we will show the following. Let f_i be a phrase of length $> w$ starting at position p_i. If there are multiple strings in the w-sliding window tree that start in the window $S[p_i - w, p_i - 1]$ and match the first w characters of f_i, then *all* of these strings can be extended to longest matches with f_i. In particular, since the algorithm chooses one of these string to compare against (the maximum or minimum such string) this implies that the algorithm is correct.

We need the following lemma.

▶ **Lemma 5** (Breslauer and Galil [3], Lemma 3.1)**.** *Let P and S be strings such that S contains at least three occurrences of P. Let $t_1 < t_2 < \cdots < t_h$ be the locations of all occurrences of P in S and assume that $t_{i+2} - t_i \leq |P|$, for $i = 1, \ldots, h-2$ and $h \geq 3$. Then, this sequence forms an arithmetic progression with difference $d = t_{i+1} - t_i$, for $i = 1, \ldots, h-1$, that is equal to the period length of P.*

Using Lemma 5 we show the following result.

▶ **Lemma 6.** *Assume we reach a leaf in the w-sliding window tree T (or T') when matching the phrase starting at p_i, and that there are $k \geq 2$ strings that are associated with the leaf and start in the window $S[p_i - w, p_i - 1]$. Let $t_1 < \cdots < t_k$ be the starting positions of the strings. Then $\mathrm{lcp}(p_i, t_j) = l_i$ for all $j = 1, \ldots, k$.*

Proof. The starting positions $t_1, \ldots, t_k$ correspond to starting positions of occurrences of the w-length substring $S[p_i, p_i + w - 1]$. Since they all start in the window $S[p_i - w, p_i - 1]$ we have $p_i - w \leq t_j < p_i$ for all $j = 1, \ldots, k$. Therefore $t_{j+2} - t_j \leq w$ for all $1 \leq j \leq k-2$ and also $p_i - t_{k-1} \leq w$, and it follows from Lemma 5 that the sequence $t_1, t_2, \ldots, t_k, p_i$ forms an arithmetic progression, i.e., the substring $S[p_i, p_i + w - 1]$ is periodic with period length $d = p_i - t_k$. The suffix $S[p_i, n]$ starts with $r \geq 1$ whole repetitions of the period followed by possibly a prefix of the period of length $r' < d$. Let $l_i = rd + r'$. All the suffixes $S[t_1, n], \ldots S[t_k, n]$ start with strictly more than r repetitions of the period. Therefore, they all match with $S[p_i, n]$ up to position $p_i + l_i - 1$. Thus, continuing matching from any of these when we reach the leaf in T will give us the correct answer. The proof for the case where we reach a leaf in T' is similar. ◀

In summary, this proves that the algorithm finds the longest match and thus correctly computes the SWLZ77 parse.

2.3 Implementation and Analysis

The total space for the two w-sliding window trees stored at any time is $O(w)$. Building the w-sliding window trees requires building a suffix tree of size $2w$. If the alphabet size is polynomial ($\sigma = n^{O(1)}$) we can build all $\lceil n/w \rceil$ suffix trees in total $O(n)$ worst-case time [7]. If the alphabet size is larger we first hash to a polynomial sized alphabet and then build the suffix trees. This takes $O(n)$ expected time. Given the suffix tree for a $2w$ length substring we construct the w-sliding window tree in $O(w)$ time and use perfect hashing at each node [13] to index the first characters of outgoing edges and thus enable linear time matching (building the perfect hash tables takes expected $O(w)$ time). This concludes the proof of Lemma 4.

In total we get $O(n)$ expected time for constructing all w-sliding window trees, and $O(n)$ time for searching for all phrases. This sums to $O(n)$ expected time as desired. This proves Theorem 1(i).

To get deterministic bounds of Theorem 1(ii), we can instead build the suffix trees using Fischer and Gawrychowski [11]. These build suffix trees in sorting time complexity and support searches for a pattern of length m in $O(m + \log\log\sigma)$ time. We search for z phrases of total length n, and hence in total we use $O(\frac{n}{w}\text{sort}(w,\sigma) + z\log\log\sigma)$ time. In summary, this proves Theorem 1.

3 Approximate Rightmost Matching

We now show Theorem 2.

3.1 Algorithm

We assume that we have the leftmost LZ77 parse (defined analogously to the rightmost parse, see beginning of Section 1.2) of the input string. If not we can easily compute it within the bounds of Theorem 2. Our algorithm updates the references of the phrases in a left-to-right order.

For *levels* $i = 1, \ldots, \log_{(1+\epsilon)} n$ we build the w-sliding window trees for $S[j(1+\epsilon)^i, (j+1)(1+\epsilon)^i]$ for $j = 0, \ldots, \frac{n}{(1+\epsilon)^i} - 1$. That is, for a fixed i, we compute all the w-sliding window trees of size $(1+\epsilon)^i$ spaced by $(1+\epsilon)^i$ characters (remember $\epsilon > 0$ is an arbitrary constant).

▶ **Definition 7** (Covering w-sliding window tree). Given a position k in S such that $j(1+\epsilon)^i \le k \le (j+1)(1+\epsilon)^i$ for some i and j, we say that the w-sliding window tree of the substring $S[j(1+\epsilon)^i, (j+1)(1+\epsilon)^i]$ is covering k on level i. We denote this tree by $T_{i,k}$.

We maintain references to the w-sliding window trees such that given k and i we can find the w-sliding window tree on level i covering k in constant time.

To update a reference of a phrase f_l beginning at position p_l to be the $(1+\epsilon)$-rightmost, we search the w-sliding window trees $T_{i,p_l-(1+\epsilon)^i}$ and T_{i,p_l}, $i = 1, \ldots, \log_{(1+\epsilon)} n$, for the occurrence of f_l closest to (but not after) p_l. We then update the phrase f_l. The search is done as described in Lemma 4.

The order in which the w-sliding window trees are searched is a binary search over the levels. If an occurrence is found at level i we continue the search on the smaller levels. If not, we continue the search on the bigger levels.

We assumed that all w-sliding window trees were built as the first step of the algorithm, but we can restrict the algorithm to only build the w-sliding window trees that are in fact needed and then discard them again when the algorithm progresses to a position not covered by it. In the proof of Theorem 2 we show that this improves the total time used to construct the w-sliding window trees from $O(n\log n)$ to $O(n\log z)$ and the space usage to $O(n)$.

3.2 Analysis

In this section we prove Theorem 2. We start by showing the approximation guarantees of our algorithm and then we analyze its space and time complexity.

3.2.1 Approximation

Here we prove that our algorithm produces a $(1+\epsilon)$-rightmost parsing.

▶ **Lemma 8.** *Let $f_1, \ldots, f_z = (r_1, l_1), \ldots, (r_z, l_z)$ be the parsing produced by our algorithm. For $k = 1, \ldots, z$ we have that $r_k \le (1+\epsilon)\hat{r}_k$, i.e., our algorithm produces a $(1+\epsilon)$-rightmost parsing.*

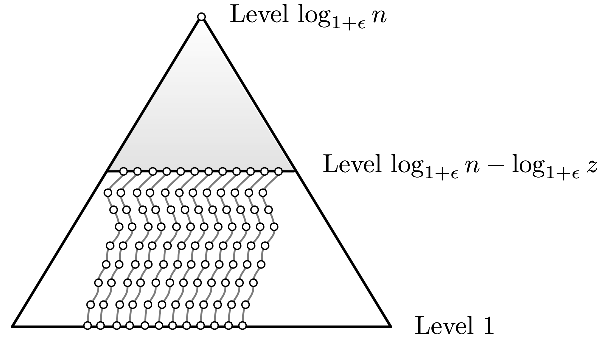

Figure 2 Suppose the hierarchy of w-sliding window trees is represented by a $(1+\epsilon)$-ary tree as shown in this figure. Consider the case where all phrases are exactly of size $\frac{n}{z}$. In this case all w-sliding window trees of size $(1+\epsilon)^{\log_{1+\epsilon} n - \log_{1+\epsilon} z} = \frac{n}{z}$ (represented by the nodes on level $\log_{1+\epsilon} n - \log_{1+\epsilon} z$) have to be built. Furthermore, all trees represented by ancestor nodes (in the shaded part of the tree) are also built. The total time to do this is $O(n \log z)$. Now suppose that all w-sliding window trees built on the levels from 1 to $\log_{1+\epsilon} n - \log_{1+\epsilon} z - 1$ form disjoint paths in the tree as shown in the figure. We then have to build each tree represented by each node, but the sizes of these are exponentially decreasing as the levels decrease, and the total work therefore sums to $O(n)$.

Proof. Consider a phrase f_k starting at position p_k. Suppose the search for an occurrence of f_k terminates on level i. This means that there is an occurrence in T_{i,p_k}, but not in T_{i-1,p_k}. Since the algorithm disregards matches starting before $p_k - (1+\epsilon)^i$ we therefore have that $(1+\epsilon)^{i-1} < \hat{r}_k \leq r_k \leq (1+\epsilon)^i$ from which it follows that $r_k \leq (1+\epsilon)\hat{r}_k$. ◀

3.2.2 Space

The space used by our algorithm is dominated by the w-sliding window trees we construct. Once we are done using a w-sliding window tree we can discard it. When processing f_l we only need the w-sliding window trees covering p_l and $p_l - (1+\epsilon)^i$ for every level i. So at any point in time the total size of the maintained w-sliding window trees is bounded by $\sum_{i=1}^{\log_{1+\epsilon} n} O((1+\epsilon)^i) = O(n)$, hence the space usage by our algorithm is $O(n)$.

3.2.3 Time

First we analyze the time required for constructing the w-sliding window trees. Recall that a suffix tree is only constructed if we actually need to access it. For each phrase beginning we may have to access $\log \log_{1+\epsilon} n$ w-sliding window trees, however some of these may be reused. Our algorithm parses the string left to right, so if a w-sliding window tree is covering both p_l and p_{l+1} we only need to construct it once since we process f_{l+1} immediately after f_l.

In the worst case, we may be required to build all w-sliding window trees on level $\log_{1+\epsilon} n - \log_{1+\epsilon} z$, meaning that all possible w-sliding window trees on level $1 + \log_{1+\epsilon} n - \log_{1+\epsilon} z$ to $\log_{1+\epsilon} n$ will also have to be built. This requires $O(n \log z)$ time since the total size of the w-sliding window trees on any level is $O(n)$ and the number of levels is $\log_{1+\epsilon} z$. In the worst case the w-sliding window trees on the remaining levels are not subject to reuse. On level $\log_{1+\epsilon} n - \log_{1+\epsilon} z$ the total size of the w-sliding window trees is $O(n)$. On the previous level we also need to build at most z w-sliding window trees but the total size of these will be $O(n/(1+\epsilon))$. Therefore the total size of the w-sliding window trees on the

lower levels is at most $\sum_{i=1}^{\log_{1+\epsilon} n - \log_{1+\epsilon} z} n/(1+\epsilon)^i = O(n)$. The total time for building the the w-sliding window trees is thus $O(n \log z + n) = O(n \log z)$ time. See also Figure 2.

We now look at the time it takes to search the w-sliding window trees. Consider phrase f_l and assume that we have all the w-sliding window trees covering p_l. We binary search for the w-sliding window tree having an occurrence of f_l as close to p_l as possible. Since there are $\log_{1+\epsilon} n$ levels this takes $O(|f_l| \log\log n)$ time, resulting in a total of $\sum_{i=1}^{z} |f_i| \log\log n = O(n \log\log n)$ time for this step. In total our algorithm uses $O(n(\log z + \log\log n))$ time. This concludes the proof of Theorem 2.

References

1. Djamal Belazzougui and Simon J. Puglisi. Range predecessor and Lempel-Ziv parsing. In Robert Krauthgamer, editor, *Proceedings of the 27th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2016)*, pages 2053–2071. SIAM, 2016. doi:10.1137/1.9781611974331.ch143.
2. Timothy C. Bell. Better OPM/L text compression. *IEEE Trans. Commun.*, 34(12):1176–1182, 1986. doi:10.1109/TCOM.1986.1096485.
3. Dany Breslauer and Zvi Galil. Real-time streaming string-matching. *ACM Trans. Algorithms*, 10(4):22:1–22:12, 2014. doi:10.1145/2635814.
4. Maxime Crochemore, Lucian Ilie, and William F. Smyth. A simple algorithm for computing the Lempel Ziv factorization. In James A. Storer and Michael W. Marcellin, editors, *Proceedings of the 2008 Data Compression Conference (DCC 2008)*, pages 482–488. IEEE Computer Society, 2008. doi:10.1109/DCC.2008.36.
5. Maxime Crochemore, Alessio Langiu, and Filippo Mignosi. The rightmost equal-cost position problem. In Ali Bilgin, Michael W. Marcellin, Joan Serra-Sagristà, and James A. Storer, editors, *Proceedings of the 2013 Data Compression Conference (DCC 2013)*, pages 421–430. IEEE, 2013. doi:10.1109/DCC.2013.50.
6. Maxime Crochemore, Alessio Langiu, and Filippo Mignosi. Note on the greedy parsing optimality for dictionary-based text compression. *Theor. Comput. Sci.*, 525:55–59, 2014. doi:10.1016/j.tcs.2014.01.013.
7. Martin Farach-Colton, Paolo Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *J. ACM*, 47(6):987–1011, 2000. doi:10.1145/355541.355547.
8. Paolo Ferragina, Igor Nitto, and Rossano Venturini. On the bit-complexity of Lempel-Ziv compression. *SIAM J. Comput.*, 42(4):1521–1541, 2013. doi:10.1137/120869511.
9. Edward R. Fiala and Daniel H. Greene. Data compression with finite windows. *Commun. ACM*, 32(4):490–505, 1989. doi:10.1145/63334.63341.
10. Johannes Fischer, Travis Gagie, Paweł Gawrychowski, and Tomasz Kociumaka. Approximating LZ77 via small-space multiple-pattern matching. In Nikhil Bansal and Irene Finocchi, editors, *Proc. of the 23rd Annual European Symposium on Algorithms (ESA 2015)*, volume 9294 of *LNCS*, pages 533–544. Springer, 2015. doi:10.1007/978-3-662-48350-3_45.
11. Johannes Fischer and Paweł Gawrychowski. Alphabet-dependent string searching with wexponential search trees. In Ferdinando Cicalese, Ely Porat, and Ugo Vaccaro, editors, *Proc. of the 26th Annual Symp. on Combinatorial Pattern Matching (CPM 2015)*, volume 9133 of *LNCS*, pages 160–171. Springer, 2015. doi:10.1007/978-3-319-19929-0_14.
12. Johannes Fischer, Tomohiro I, and Dominik Köppl. Lempel-Ziv computation in small space (LZ-CISS). In Ferdinando Cicalese, Ely Porat, and Ugo Vaccaro, editors, *Proc. of the 26th Annual Symposium on Combinatorial Pattern Matching (CPM 2015)*, volume 9133 of *LNCS*, pages 172–184. Springer, 2015. doi:10.1007/978-3-319-19929-0_15.
13. Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *J. ACM*, 31(3):538–544, 1984. doi:10.1145/828.1884.

14 Keisuke Goto and Hideo Bannai. Space efficient linear time Lempel-Ziv factorization for small alphabets. In Ali Bilgin, Michael W. Marcellin, Joan Serra-Sagristà, and James A. Storer, editors, *Proceedings of the 2014 Data Compression Conference (DCC 2014)*, pages 163–172. IEEE, 2014. doi:10.1109/DCC.2014.62.

15 Yijie Han. Deterministic sorting in $O(n \log\log n)$ time and linear space. In John H. Reif, editor, *Proceedings of the 34th Annual ACM Symposium on Theory of Computing (STOC 2002)*, pages 602–608. ACM, 2002. doi:10.1145/509907.509993.

16 Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. Lightweight Lempel-Ziv parsing. In Vincenzo Bonifaci, Camil Demetrescu, and Alberto Marchetti-Spaccamela, editors, *Proc. of the 12th International Symposium on Experimental Algorithms (SEA 2013)*, volume 7933 of *LNCS*, pages 139–150. Springer, 2013. doi:10.1007/978-3-642-38527-8_14.

17 Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. Linear time Lempel-Ziv factorization: Simple, fast, small. In Johannes Fischer and Peter Sanders, editors, *Proceedings of the 24th Annual Symposium on Combinatorial Pattern Matching (CPM 2013)*, volume 7922 of *LNCS*, pages 189–200. Springer, 2013. doi:10.1007/978-3-642-38905-4_19.

18 Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. Lempel-Ziv parsing in external memory. In Ali Bilgin, Michael W. Marcellin, Joan Serra-Sagristà, and James A. Storer, editors, *Proceedings of the 2014 Data Compression Conference (DCC 2014)*, pages 153–162. IEEE, 2014. doi:10.1109/DCC.2014.78.

19 Dominik Kempa and Simon J. Puglisi. Lempel-Ziv factorization: Simple, fast, practical. In Peter Sanders and Norbert Zeh, editors, *Proceedings of the 15th Meeting on Algorithm Engineering and Experiments (ALENEX 2013)*, pages 103–112. SIAM, 2013. doi:10.1137/1.9781611972931.9.

20 Dominik Köppl and Kunihiko Sadakane. Lempel-Ziv computation in compressed space (LZ-CICS). In Ali Bilgin, Michael W. Marcellin, Joan Serra-Sagristà, and James A. Storer, editors, *Proceedings of the 2016 Data Compression Conference (DCC 2016)*, pages 3–12. IEEE, 2016. doi:10.1109/DCC.2016.38.

21 S. Rao Kosaraju and Giovanni Manzini. Compression of low entropy strings with Lempel-Ziv algorithms. *SIAM J. Comput.*, 29(3):893–911, 1999. doi:10.1137/S0097539797331105.

22 Dmitry Kosolobov. Faster lightweight Lempel-Ziv parsing. In Giuseppe F. Italiano, Giovanni Pighizzini, and Donald Sannella, editors, *Proceedings of the 40th International Symposium on Mathematical Foundations of Computer Science (MFCS 2015)*, volume 9235 of *LNCS*, pages 432–444. Springer, 2015. doi:10.1007/978-3-662-48054-0_36.

23 Alessio Langiu. On parsing optimality for dictionary-based text compression – the Zip case. *J. Discrete Algorithms*, 20:65–70, 2013. doi:10.1016/j.jda.2013.04.001.

24 N. Jesper Larsson. Extended application of suffix trees to data compression. In James A. Storer and Martin Cohn, editors, *Proceedings of the 1996 Data Compression Conference (DCC 1996)*, pages 190–199. IEEE Computer Society, 1996. doi:10.1109/DCC.1996.488324.

25 Edward M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2):262–272, 1976. doi:10.1145/321941.321946.

26 Joong Chae Na, Alberto Apostolico, Costas S. Iliopoulos, and Kunsoo Park. Truncated suffix trees and their application to data compression. *Theor. Comput. Sci.*, 1-3(304):87–101, 2003. doi:10.1016/S0304-3975(03)00053-7.

27 Enno Ohlebusch and Simon Gog. Lempel-Ziv factorization revisited. In Raffaele Giancarlo and Giovanni Manzini, editors, *Proceedings of the 22nd Annual Symposium on Combinatorial Pattern Matching (CPM 2011)*, volume 6661 of *LNCS*, pages 15–26. Springer, 2011. doi:10.1007/978-3-642-21458-5_4.

28 Daisuke Okanohara and Kunihiko Sadakane. An online algorithm for finding the longest previous factors. In Dan Halperin and Kurt Mehlhorn, editors, *Proceedings of the 16th Annual European Symposium on Algorithms (ESA 2008)*, volume 5193 of *LNCS*, pages 696–707. Springer, 2008. doi:10.1007/978-3-540-87744-8_58.

29 Alberto Policriti and Nicola Prezza. Fast online Lempel-Ziv factorization in compressed space. In Costas S. Iliopoulos, Simon J. Puglisi, and Emine Yilmaz, editors, *Proceedings of the 22nd International Symposium on String Processing and Information Retrieval (SPIRE 2015)*, volume 9309 of *LNCS*, pages 13–20. Springer, 2015. doi:10.1007/978-3-319-23826-5_2.

30 Alberto Policriti and Nicola Prezza. Computing LZ77 in run-compressed space. In Ali Bilgin, Michael W. Marcellin, Joan Serra-Sagristà, and James A. Storer, editors, *Proceedings of the 2016 Data Compression Conference (DCC 2016)*, pages 23–32. IEEE, 2016. doi:10.1109/DCC.2016.30.

31 Julian Shun and Fuyao Zhao. Practical parallel Lempel-Ziv factorization. In Ali Bilgin, Michael W. Marcellin, Joan Serra-Sagristà, and James A. Storer, editors, *Proceedings of the 2013 Data Compression Conference (DCC 2013)*, pages 123–132. IEEE, 2013. doi:10.1109/DCC.2013.20.

32 Tatiana Starikovskaya. Computing Lempel-Ziv factorization online. In Branislav Rovan, Vladimiro Sassone, and Peter Widmayer, editors, *Proceedings of the 37th International Symposium on Mathematical Foundations of Computer Science (MFCS 2012)*, volume 7464 of *LNCS*, pages 789–799. Springer, 2012. doi:10.1007/978-3-642-32589-2_68.

33 James A. Storer and Thomas G. Szymanski. Data compression via textural substitution. *J. ACM*, 29(4):928–951, 1982. doi:10.1145/322344.322346.

34 Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995. doi:10.1007/BF01206331.

35 Jun-ichi Yamamoto, Tomohiro I, Hideo Bannai, Shunsuke Inenaga, and Masayuki Takeda. Faster compact on-line Lempel-Ziv factorization. In Ernst W. Mayr and Natacha Portier, editors, *Proceedings of the 31st International Symposium on Theoretical Aspects of Computer Science (STACS 2014)*, volume 25 of *LIPIcs*, pages 675–686. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2014. doi:10.4230/LIPIcs.STACS.2014.675.

36 Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory*, 23(3):337–343, 1977. doi:10.1109/TIT.1977.1055714.