Decompressing Lempel-Ziv Compressed Text

Philip Bille*, Mikko Berggren Ettienne*, Travis Gagie†,
Inge Li Gørtz*, and Nicola Prezza‡

*Technical University of Denmark
2800 Kgs. Lyngby
Copenhagen, Denmark
{phbi,inge}@dtu.dk ,
mikkobe@gmail.com

†Dalhousie University
6050 University Ave.
Halifax, Canada
travis.gagie@dal.ca

‡LUISS University
Viale Romania, 32
Rome, Italy
nicola.prezza@gmail.com

Abstract

We consider the problem of decompressing the Lempel–Ziv 77 representation of a string S of length n using a working space as close as possible to the size z of the input. The folklore solution for the problem runs in $O(n)$ time but requires random access to the whole decompressed text. Another folklore solution is to convert LZ77 into a grammar of size $O(z \log(n/z))$ and then stream S in linear time. In this paper, we show that $O(n)$ time and $O(z)$ working space can be achieved for constant-size alphabets. On general alphabets of size σ, we describe (i) a trade-off achieving $O(n \log^{\delta} \sigma)$ time and $O(z \log^{1-\delta} \sigma)$ space for any $0 \le \delta \le 1$, and (ii) a solution achieving $O(n)$ time and $O(z \log\log(n/z))$ space. The latter solution, in particular, dominates both folklore algorithms for the problem. Our solutions can, more generally, extract any specified subsequence of S with little overheads on top of the linear running time and working space. As an immediate corollary, we show that our techniques yield improved results for pattern matching problems on LZ77-compressed text.

1 Introduction

In this paper we consider the following problem: given an LZ77 representation of a string S of length n, decompress S and output it as a stream in left-to-right order (without storing it explicitly). Our goal is to solve this problem in as little space as possible, i.e. close to the size z of the compressed input string, and as fast as possible. This problem is fundamental in applications dominated by big repetitive data, where information has to be analyzed on-the-fly due to limitations in storage resources.

The folklore solution for the Lempel-Ziv decompression problem achieves linear time, but requires random access to the whole string. A better solution is to convert LZ77 into a straight-line program (i.e. a context-free grammar generating the text) of size $O(z \log(n/z))$. This conversion can be performed in $O(z \lg(n/z))$ space and time [1, 2]. Then, the entire text can be decompressed and streamed in linear time using just the space of the grammar. The problem has also been recently considered in [3] in the context of external-memory algorithms. To the best of our knowledge, the only other work addressing small-space LZ77 decompression is [4], which implements (a practical version of) the ideas described in our paper. In particular, no other theoretical solutions using $O(z)$ space are known.

1.1 Our contributions

The main contribution of this paper is to show that LZ77 decompression can be performed in $O(z)$ space and almost linear time (in the length of the extracted string). We provide two space-time trade-offs which enable us to achieve either linear time *or* linear space *or* both if the alphabet's size is constant. The first trade-off is particularly appealing on small alphabets, while the second dominates the first on large alphabets and the folklore algorithm based on grammars.

Our solution even works for decompressing any specified subsequence of S with little overheads on top of the linear running time (in the extracted substring's length) and working space. As an application, we show that our techniques yield improved results for pattern matching problems on LZ77-compressed text.

We formalize the LZ77 decompression problem as follows. The input consists of an LZ77 representation of a text (we use the version not allowing self references: each phrase cannot overlap its own source) and a list of text substrings encoded as pairs: $(i_1, j_1), \ldots, (i_s, j_s)$. We decompress these substrings and output them (e.g. to a stream or to disk) character-by-character in the order $S[i_1, j_1], \ldots, S[i_s, j_s]$. Since both the input strings and the output can be streamed (for example, from/to disk) we only count the working space used on top of the input and the output. Let the quantity $l = \sum_{k=1}^{s}(j_k - i_k + 1)$ denote the total number of characters to be extracted. Our main results are summarized in the following two theorems. Let S be a string of length n from an alphabet of size σ compressed into an LZ77 representation with z phrases.

Theorem 1. *For any parameter $0 \leq \delta \leq 1$, we can decompress any s substrings of S with total length l in $O(l \lg^{\delta} \sigma + (s + z)\lg(n/z))$ time using $O(z \lg^{1-\delta} \sigma)$ space.*

Theorem 2. *For any parameter $1 \leq \tau \leq \lg(n/z)$, we can decompress any s substrings of S with total length l in $O\left(\frac{l \lg(n/z)}{\tau} + (s + z)\lg(n/z)\right)$ time using $O(z \lg \tau)$ space.*

Theorems 1 and 2 lead to a series of new and non-trivial bounds on different algorithmic problems on LZ77. For instance, we provide a smooth time-space trade-off for decompressing the whole S in $O(n \lg^{\delta} \sigma)$ time using $O(z \lg^{1-\delta} \sigma)$ space for any constant $0 \leq \delta \leq 1$. By combining Theorem 2 with $\tau = \lg(n/z)$ with grammars, we furthermore show how to decompress S in $O(n)$ time using $O(z \lg \lg(n/z))$ space. Both bounds are strict improvements over the previous best complexity of $O(n)$ time and $O(z \lg(n/z))$ space. See Section 4 and Corollaries 1 and 2 for details.

Our results also imply new trade-offs for the pattern matching and approximate pattern matching problems on LZ77-compressed texts. By showing how our techniques can be combined with existing pattern matching results, in the full version of the paper we show the following:

Theorem 3. *Let S be a string of length n compressed into an LZ77 representation $\mathcal{Z}$ with z phrases, let P be a pattern of length m and let $\mathcal{A}$ be an algorithm that can detect an (approximate) occurrence of P in S (with at most k errors) given P and $\mathcal{Z}$ in $t(z, n, m, k)$ time and $s(z, n, m, k)$ space. Then, we can solve the same task in*

$O(t(z, zm, m, k) + z \lg(n/z))$ *time and* $O(s(z, zm, m, k) + z)$ *space. If* $\mathcal{A}$ *reports all* occ *occurrences using* $t(z, n, m, k)$ *time and* $s(z, n, m, k)$ *space, then we can report all occurrences in* $O(t(z, zm, m, k) + z \lg(n/z) + \mathsf{occ})$ *time and* $O(s(z, zm, m, k) + z + \mathsf{occ})$ *space.*

Theorem 4. *Let* $\mathcal{A}$ *be a streaming algorithm that reports all* occ *(approximate) occurrence of a pattern* $P \in [\sigma]^m$ *(with at most* k *errors) in a stream of length* n *in* $t(n, m, k)$ *time and* $s(n, m, k)$ *space. Then, we can report all occurrences of* P *in the LZ77 representation of a string* $S \in [\sigma]^n$ *in either:*

- $O(t(zm, m, k) + z \lg(n/z))$ *time and* $O(s(zm, m, k) + z \lg\lg(n/z) + \mathsf{occ})$ *space or*
- $O(t(zm, m, k) + z \lg(n/z) + zm \lg^{\delta} \sigma)$ *time and* $O(s(zm, m, k) + z \lg^{1-\delta} \sigma + \mathsf{occ})$ *space.*

The best known algorithm for detecting if pattern P occurs in a string S given P and $\mathcal{Z}$ uses $O(z \lg(n/z) + m)$ time and $O(z \lg n + m)$ space [5]. If we plug this into Theorem 3 we obtain $O(z \lg(n/z) + m)$ time and $O(z \lg m + m)$ space thereby reducing the $\lg n$ factor in the space to $\lg m$ without increasing the time.

We also obtain new trade-offs for reporting all approximate occurrences of P with at most k errors. For example, if we plug in the Landau–Vishkin and Cole–Hariharan [6, 7] algorithms, we can solve the problem in $O(z \lg(n/z) + z \min\{mk, k^4 + m\} + \mathsf{occ})$ time using $O(z + m + \mathsf{occ})$ space for constant-sized alphabets or $O(z \lg\lg(n/z) + m + \mathsf{occ})$ space for general alphabets. The previous best solution has the same time complexity but uses $O(z \lg n + m + \mathsf{occ})$ space [8]. The complete explanation can be found in the full version of the paper.

1.2 Related work

While the LZ77 decompression problem has not been studied much in the literature, the problem of fast LZ77 *compression* in small working space has lately attracted a lot of research in the field of compressed computation [9, 10, 11, 12, 13].

A closely related problem is the *random access problem*, where the aim is to build a data structure taking space as close as possible to $O(z)$ words and supporting efficient access queries to single characters. Existing solutions for the random access problem [1, 2, 14] need $\Omega(z \log(n/z))$ space to achieve $O(\log(n/z))$ access time. Because these data structures can be built efficiently they also solve the LZ77 decompression problem considered in this paper. In particular they can decompress the entire string S given its LZ77 representation in $O(n)$ time using $O(z \lg(n/z))$ working space.

Random access data structures can also decompress any set of s substrings with total length l in $O(l + s \lg n)$ time. We provide several new trade-offs for this problem; for instance we can solve it using $O(l + (z + s) \lg(n/z))$ time and $O(z)$ space for constant-sized alphabets or $O(z \lg\lg(n/z))$ space for general alphabets.

In a recent work [4], Puglisi and Rossi implemented the ideas described in our paper. They showed that, even if an implementation of our algorithms is not practical due to the underlying mergeable dictionary, several optimizations can be introduced that drastically improve performance. Their optimized implementation led to new relevant space-time trade-offs on several datasets of practical interest.

2 Preliminaries

We assume a standard unit-cost RAM model with word size $w = \Theta(\lg n)$ and that the input is from an integer alphabet $\Sigma = \{1, 2, \ldots, \sigma\}$ where $\sigma \leq n^{O(1)}$, and we measure space complexity in words unless otherwise specified. A string S of length $n = |S|$ is a sequence $S[1] \ldots S[n]$ of n symbols from an alphabet Σ of size $|\Sigma| = \sigma$. The string $S[i] \ldots S[j]$ denoted $S[i, j]$ is called a *substring* of S. Let ϵ denote the empty string and let $S[i, j] = \epsilon$ when $i > j$. To ease the notation, let $S[i, j] = S[1, j]$ if $i < 1$ and $S[i, n]$ if $j > n$. Let $[u]$ be shorthand for the interval $[1; u] = \{1, 2, \ldots, u\}$ and \$ be a special symbol that never occurs in the input. A straight-line program (SLP) is an acyclic grammar in Chomsky normal form where each non-terminal T has exactly one production rule with T as its left-hand side i.e., a grammar where each non-terminal production rule expands to two other rules and generates one string only.

Lempel-Ziv 77 Algorithm For simplicity of exposition we use the scheme given by Farach & Thorup [15]. Map Σ into $[\sigma]$ and assume that S is prefixed by Σ in the negative positions, i.e. $S[-c] = c$ for $c \in \Sigma$ and $S[0] = \$ \notin \Sigma$.

An *LZ77 representation* [16, 17] of S is a string $\mathcal{Z}$ of the form $(s_1, l_1) \ldots (s_z, l_z) \in ([-\sigma; n] \times [n])^z$. Let $u_1 = 1$ and $u_i = u_{i-1} + l_{i-1}$, for $i > 1$. For $\mathcal{Z}$ to be a valid LZ77 representation of S, we require that $s_i + l_i \leq u_i$ and that $S[u_i, u_i + l_i - 1] = S[s_i, s_i + l_i - 1]$ for $i \in [z]$. This guarantees that $\mathcal{Z}$ *represents* S and clearly S is uniquely defined in terms of $\mathcal{Z}$. We refer to the substring $S[u_i, u_i + l_i - 1]$ as the i^{th} *phrase* of the representation, the substring $S[s_i, s_i + l_i - 1]$ as the *source of the* i^{th} *phrase* and (s_i, l_i) as the i^{th} *member* of $\mathcal{Z}$. We note that the restriction $s_i + l_i \leq u_i$ for all i implies that a source and a phrase cannot overlap and thus we do not handle representations that are self-referential.

By the given definition, the LZ77 representation of a string is not unique, however a minimal *LZ77* representing a text S can be found greedily in $O(n)$ time [18, 19].

Mergeable Dictionary To obtain our results we need mergeable dictionaries with shift operations. The Mergeable Dictionary problem is to maintain a dynamic collection $\mathcal{G}$ of sets $\{G_1, G_2, \ldots\}$ of n elements from an ordered universe $\{1, 2, \ldots, \mathcal{U}\}$ starting from n singleton sets under the operations:

1. $C \leftarrow \texttt{merge}(A, B)$: Remove A and B from $\mathcal{G}$ and insert $C = A \cup B$ instead.
2. $(A, B) \leftarrow \texttt{split}(G, x)$: Split G into two sets $A = \{y \in G \mid y \leq x\}$ and $B = \{y \in G \mid y > x\}$. G is removed from $\mathcal{G}$ while A and B are inserted.
3. $G' \leftarrow \texttt{shift}(G, x)$ for some x such that $y + x \in [\mathcal{U}]$ for each $y \in G$: Create the set $G' = \{y + x \mid y \in G\}$. G is removed from $\mathcal{G}$ while G' is inserted.
4. $\texttt{makeset}(j)$: Insert a new singleton set $G = \{j\}$ in $\mathcal{G}$.

Bille et al. [20] show how to extend the mergeable dictionary by Iacono & Özkan [21] to support shifts (Iacono & Özkan [21] write that their data structure can be extended to support the shift operation but do not provide any details):

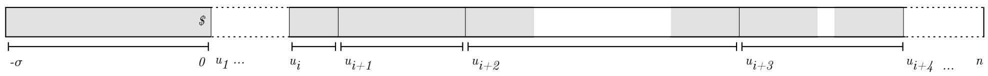

Figure 1: Example of the τ-context of a string. Dashed parts are truncated parts of the string not shown by the figure, grey parts represent substrings in the τ-context and white parts represent substrings not in the τ-context. The first substring in the negative positions $-\sigma$ through 0 is always in the τ-context. Recall that l_i is the length of the i^{th} phrase. In this example, $l_i < \tau, l_{i+1} \leq 2\tau$ and $l_{i+2}, l_{i+3} > 2\tau$.

Theorem 5 ([20])**.** *There exists a mergeable dictionary with shifts supporting all operations in* $\lg \mathcal{U}$ *amortized time using linear space. For a set* G*, let* $\mathcal{U}_G = \max(G) - \min(G)$*. The split operations take* $O(\lg \mathcal{U}_G)$ *worst-case and amortized time and the makeset and shift operations take* $O(1)$ *worst-case and amortized time. The amortized time of the merge operation is* $O(\lg \mathcal{U}_G)$*, where* G *is the set output by the operation.*

3 LZ77 Induced Context

In this section we present the centerpiece of our algorithm. It builds on the fundamental property of LZ77 compression that any substring of a phrase also occurs in the source of that phrase. Our technique is to store a short substring, which we call *context*, extracted around phrase borders. The contexts are stored in a compressed form that allows faster substring extraction than that of LZ77. We then take advantage of this property when extracting a substring of S by splitting it into short chunks which in turn are extracted by repeatedly mapping them to the source of the phrase they are part of. Eventually, they will end up as substrings of the contexts from where they can be efficiently extracted.

The technique resembles what Farach & Thorup refer to as *winding* in [15]. We show new applications of the technique and obtain better time complexity by using the mergeable dictionary of Section 2.

Recall that we assume S is prefixed by the alphabet in the negative positions, that $S[0] = \$$, and that u_k is the starting position in S of the k^{th} phrase.

Definition 1. *Let* τ *be a positive integer. The* τ-context *of a string* S *(induced by an LZ77 representation* $\mathcal{Z}$ *of* S*) is the set of positions* j *where either* $j \leq 0$ *or there is some* k *such that* $u_k - \tau < j < u_k + \tau$*. If positions* i *through* j *are in the* τ*-context of* S*, then we simply say "*$S[i, j]$ *is in the* τ*-context of* S*".*

Definition 2. *Let* τ *be a positive integer. The* τ-context string *of* S*, denoted* S^τ*, is the subsequence of* S *that includes* $S[j]$ *if and only if* j *is in the* τ*-context of* S*. We denote with* $\pi^\tau(j)$ *the unique position in* S^τ *where such a position* j *is mapped to (i.e.* $S^\tau[\pi^\tau(j)] = S[j]$*).*

It is easy to show how to map positions from S to S^τ (see the full version for a proof)

Lemma 1. *Let* $\mathcal{Z}$ *be an LZ77 representation of a string* S *of length* n *with* z *phrases and let* τ *be a positive integer. Given* $t = O(z)$ *sorted positions,* $p_1 \leq \ldots \leq p_t \in [n]$ *in the* τ*-context of* S *we can compute* $\pi^\tau(p_1), \ldots, \pi^\tau(p_t)$ *in* $O(z)$ *time and space.*

We use π as shorthand for π^τ whenever τ is clear from context. The following properties follow from the definitions and Lemma 1 but will come in handy later on:

Property 1. *If a, a' are positions in the τ-context of S and $a < a'$ then $\pi(a) < \pi(a')$.*

Property 2. *If $S[a, b]$ is in the τ-context of S then $S^\tau[\pi(a), \pi(b)] = S[a, b]$*

We now consider the following problem: given a substring $S[i, j]$ of length at most τ, find a pair of integers (i', j') such that $i' \leq i$, $S[i, j] = S[i', j']$ and $S[i', j']$ is in the τ-context of S.

We first give an informal overview of how the algorithm works. Recall that if a substring of S is contained within a phrase in the LZ77 parse of S, then the substring also occurs in the source of that phrase. The idea is to repeat this process of finding an identical substring in the source until the found string is in the τ-context of S, which happens after at most z steps. To do this efficiently for multiple strings, we use the mergeable dictionary structure to maintain the relevant positions. This allows us to process all strings inside a phrase simultaneously because they all need to be moved to the same source. By processing the phrases in right-to-left order we can bound the number of dictionary operations by the number of phrases.

The following algorithm gives the details of how to solve the problem for a set of z substrings using $O(z)$ space and $O(z \lg n)$ time (which we later improve).

Algorithm 1. *Let $\mathcal{Z}$ be an LZ77 representation of a string S of length n with z phrases and let τ be a positive integer. The input is $t = O(z)$ substrings of S given as pairs of integers denoting start and end positions: $(a_1, b_1), \ldots, (a_t, b_t)$ where $b_i - a_i < \tau$ for all $i \in [t]$. Let $\mathcal{G}$ be a mergeable dictionary as given by Theorem 5. For each of the pairs (a_i, b_i) create a singleton set G_i with element x_i at position a_i and finally merge all these elements into a single set G. Each element x_i has associated its rank i among the input pairs as satellite information.*

We now consider the members of $\mathcal{Z}$ one by one in reverse order. Member (s_i, l_i) is processed as follows:

1. *If $l_i \leq \tau$ skip to the next member.*
2. *Otherwise let*
   - *(a)* $(A, B) \leftarrow \texttt{split}(G, u_i + l_i - \tau)$
   - *(b)* $(A', B') \leftarrow \texttt{split}(A, u_i - 1)$,
   - *(c)* $B'' \leftarrow \texttt{shift}(B', s_i - u_i)$
   - *(d)* $G \leftarrow \texttt{merge}(A', B'')$.

In step 1, we skip a phrase if it is no longer than τ because any string of length τ or shorter starting in that phrase is already in the τ-context of S. In step 2a, we split the set such that all strings that start in the last τ positions of the phrase are not shifted, because these already are in the τ-context of S. In 2a-d we split the set

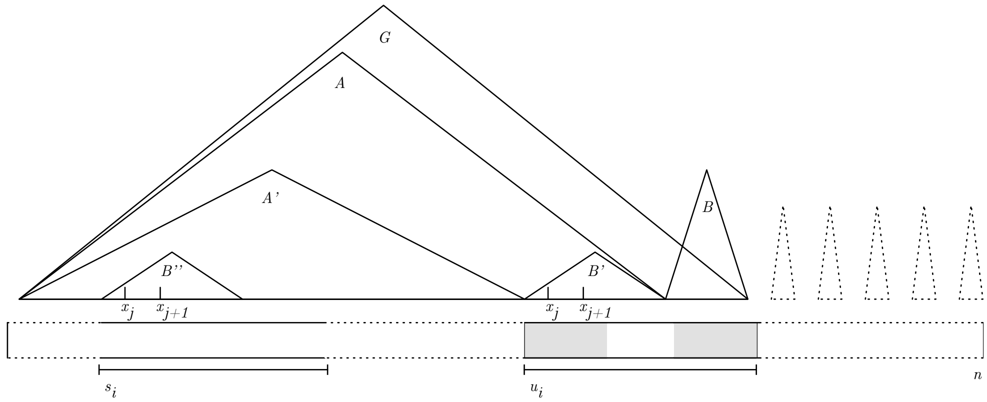

Figure 2: Example of the dictionaries created during an iteration. The dashed parts of the string are truncated parts not relevant to the example. The dotted triangles represent the B-sets from earlier iterations. The grey parts show the τ-context of the string inside the i^{th} and $(i-1)^{th}$ phrase. Note that the set B'' is the set B' after the shift operation. As exemplified by the elements x_j and x_{j+1}, the relative order and position inside the set is unaffected by the shift. Let p and p' be the position of x_j before and after the shift, respectively. Observe also that $S[p, p + (b_j - a_j) - 1] = S[p', p' + (b_j - a_j) - 1]$, so the shift does not affect the substring represented by x_j. An iteration starts from the set G, obtaining A by cutting off B. The new G (not shown in the figure) is then obtained by shifting all elements in the range of B', which are all contained in the i^{th} phrase to the same relative position in the source of the phrase.

to obtain the set of strings B'' that starts in the i^{th} phrase excluding those starting in the last τ positions, as they are already in the τ-context of S. These strings are then shifted to the source and will be considered again in later iterations.

After processing all members, scan each set in $\mathcal{G}$ to retrieve all the elements. Let $p(x_i)$ denote the new position in $\mathcal{G}$ of element x_i. We then output the pairs $(p(x_1), e_1), \ldots, (p(x_t), e_t)$ in order of their rank i where $e_i = p(x_i) + b_i - a_i$.

The proof of correctness and the analysis of the time complexity can be found in the full version of the paper.

Bucketing To decrease the running time from $O(z \lg n)$ to $O(z \lg(n/z))$, we apply a bucketing argument similar to the one given by Farach & Thorup [15]. The overall idea, described in detail in the full version of the paper, is to divide the universe of text positions $[n]$ into z buckets of size $\lceil n/z \rceil$ each, and keep a separate mergeable dictionary for each of the buckets. With a constant additional number of dictionary operations per phrase, we are then able to simulate the dictionary used by Algorithm 1. Since the universe size of each dictionary is now reduced to $O(n/z)$, dictionary operations cost just $O(\lg(n/z))$ time (amortized) and we obtain:

Lemma 2. *Let $\mathcal{Z}$ be an LZ77 representation of a string S of length n with z phrases and let τ be a positive integer. Given $t \in O(z)$ substrings of S as pairs of integers $(a_1, b_1), \ldots, (a_t, b_t)$ where $b_i - a_i < \tau$ we can find t pairs of integers $(a'_1, b'_1), \ldots, (a'_t, b'_t)$ such that $S[a'_i, b'_i] = S[a_i, b_i]$, $a'_i \leq a_i$ and $S[a'_i, b'_i]$ is in the τ-context of S using $O(z)$ space and $O(z \lg(n/z))$ time.*

3.1 LZ77 Compressed Context

It is possible to obtain an LZ77 representation $\mathcal{Z}^\tau$ of the string S^τ directly from an LZ77 representation $\mathcal{Z}$ of S. Informally, the idea is to split every phrase of $\mathcal{Z}$ into two new phrases consisting of respectively the first and last $O(\tau)$ characters of the phrase. In order to find a source for these phrases, we use Algorithm 1 which finds an identical string that also occurs in S^τ.

We now describe the algorithm sketched above that constructs an LZ77 representation $\mathcal{Z}^\tau$ of S^τ given the LZ77 parse $\mathcal{Z}$ of S.

Algorithm 2. *First we construct $O(z)$* relevant pairs *of integers representing substrings of S by considering the members of $\mathcal{Z}$ one by one in order. Member (s_i, l_i) is processed as follows:*

1. *If $l_i \le \tau$: Let $(u_i, u_i + l_i - 1)$ be a relevant pair.*
2. *If $\tau < l_i < 2\tau$: Let $(u_i, u_i + \tau - 1)$ and $(u_i + \tau, u_i + l_i - 1)$ be relevant pairs.*
3. *Otherwise $l_i \ge 2\tau$: Let $(u_i, u_i + \tau - 1)$ and $(u_i + l_i - \tau + 1, u_i + l_i - 1)$ be relevant pairs.*

Each of the relevant pairs represents a prefix or a suffix of a phrase. The concatenation of these phrase prefixes and suffixes in left-to-right order is exactly the string S^τ. Let (a, b) be a relevant pair created when considering the i^{th} member of $\mathcal{Z}$. Then we say that $(a', b') = (a - u_i + s_i, b - u_i + s_i)$ is the related source pair *and clearly $S[a, b] = S[a', b']$.*

Note that the related source pairs might not be in the τ-context. We now use Algorithm 1 to find a pair of integers (a'', b'') for each related source pair (a', b') such that $S[a'', b''] = S[a', b']$, $a'' \le a'$ and $S[a'', b'']$ is in the τ-context of S. We give the pairs in order of creation and this order is preserved by Algorithm 1. If (a'', b'') is the i^{th} output of Algorithm 1 then $(\pi^\tau(a''), l)$ is the i^{th} member of Z^τ where $l = b - a + 1$ and $\pi^\tau(a'')$ is computed using Lemma 1.

We leave the proof of correctness and the analysis of the time complexity to the full version of the paper. We obtain the following lemma:

Lemma 3. *Let $\mathcal{Z}$ be an LZ77 representation of a string S of length n with z phrases. We can construct an LZ77 representation $\mathcal{Z}^\tau$ of S^τ with $O(z)$ phrases in $O(z \lg(n/z))$ time and $O(z)$ space.*

3.2 Packed and SLP-Compressed Context

In this section we consider how to store in two different (packed/compressed) representations a τ-context string of S. Our representations can be built quickly and support fast random access.

Our first solution uses word packing. First, we construct the LZ77 representation of S^τ using Algorithm 2 and decompress it naively. Constructing the representation takes time $O(z \lg(n/z))$ while decompressing it takes linear time in its length, $O(z\tau)$. A string of length $z\tau$ can be stored in $O(z\tau \lg \sigma / \lg n)$ words using word packing.

Lemma 4. *Let S be a string S of length n from an alphabet of size σ compressed into an LZ77 representation with z phrases, and let τ be a positive integer. We can build and store S^τ in $O(z(\lg(n/z)+\tau))$ time and $O(z\tau\lg\sigma/\lg n)$ space.*

As an alternative solution, we show how to store the context string as an SLP supporting fast random access. The following lemma follows easily from Charikar et al. [2] and Rytter [1] (for a proof, see the full version of the paper):

Lemma 5. *Let S be a string of length n compressed into an LZ77 representation with z phrases, and let τ be a positive integer. We can build a balanced SLP of size $O(z\lg\tau)$ for S^τ in $O(z\lg(n/z))$ time and $O(z\lg\tau)$ space. Furthermore, the SLP supports extraction of any length-ℓ substring of S^τ in $O(\ell+\log\tau)$ time.*

4 LZ77 Decompression

We now describe how to apply the techniques described in the previous section to extract arbitrary substrings of S. Let S be a string of length n compressed into an LZ77 representation with z phrases and let $\tau\le\log(n/z)$ be a positive integer that we will fix later. We show how to extract s substrings of total length l.

Split each substring into consecutive blocks of length τ (except, possibly, the last for each substring), obtaining at most $l/\tau+s$ blocks. Process a batch of z blocks at a time in left-to-right order. There are at most $O(1+l/(\tau z)+s/z)$ batches, each containing z blocks. A batch is processed in $O(z\log(n/z))$ time using Lemma 2 thereby finding a substring s' in the τ-context of S for every block s in the batch.

Using Lemma 5, we first build the SLP in $O(z\lg(n/z))$ time and $O(z\lg\tau)$ space. After that, the z substrings in each batch can be extracted in $O(z\log\tau+z\tau)=O(z\tau)$ time. Summing up, the time to build the SLP (once) and extract and output all batches is $O(z\lg(n/z)+(1+\frac{l}{\tau z}+s/z)(z\lg(n/z)+z\tau))=O((s+z)\lg(n/z)+\frac{l\lg(n/z)}{\tau})$. The total space is $O(z\lg\tau)$. This proves Theorem 2. If we instead use Lemma 4, we spend $O(z(\lg(n/z)+\tau))$ time and $O(z\tau\lg\sigma/\lg n)$ space to build S^τ. After that, the z substrings in each batch can be extracted in $O(z\tau)$ time. Summing up, the time to build S^τ (once) and extract and output all batches is $O(z(\lg(n/z)+\tau)+(1+\frac{l}{\tau z}+s/z)(z\lg(n/z)+z\tau))=O((s+z)\lg(n/z)+\frac{l\lg(n/z)}{\tau})$ while the space becomes $O(z\tau\lg\sigma/\lg n)$ (on top of the input). To prove Theorem 1, we fix $\tau=\lg(n/z)/\lg^\delta\sigma$ for any constant $0\le\delta\le 1$.

Theorems 1 and 2 immediately yield the following two corollaries on the complexity of decompressing the entire string S (notice that $z\lg(n/z)=O(n)$):

Corollary 1. *For any parameter $0\le\delta\le 1$, we can decompress S in $O(n\lg^\delta\sigma)$ time using $O(z\lg^{1-\delta}\sigma)$ space.*

On large alphabets, we can further improve upon this result by plugging $\tau=\log(n/z)$ into Theorem 2:

Corollary 2. *We can decompress S in $O(n)$ time using $O(z\lg\lg(n/z))$ space.*

5 References

[1] W. Rytter, "Application of Lempel-Ziv factorization to the approximation of grammar-based compression," *TCS*, vol. 302, no. 1-3, pp. 211–222, 2003.

[2] M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat, "The smallest grammar problem," *IEEE Trans. Info. Theory*, vol. 51, no. 7, pp. 2554–2576, 2005.

[3] D. Belazzougui, J. Kärkkäinen, D. Kempa, and S. J. Puglisi, "Lempel-Ziv Decoding in External Memory," in *Proc. SEA*, 2016, pp. 63–74.

[4] S. J. Puglisi and M. Rossi, "On Lempel-Ziv Decompression in Small Space," in *Proc. DCC*, 2019, pp. 221–230.

[5] Paweł Gawrychowski, "Pattern matching in lempel-ziv compressed strings: fast, simple, and deterministic," in *European Symposium on Algorithms*, 2011, pp. 421–432.

[6] Gad M Landau and Uzi Vishkin, "Fast parallel and serial approximate string matching," *JDA*, vol. 10, no. 2, pp. 157–169, 1989.

[7] R. Cole and R. Hariharan, "Approximate string matching: A simpler faster algorithm," *SIAM JComp.*, vol. 31, no. 6, pp. 1761–1782, 2002.

[8] T. Gagie, P. Gawrychowski, and S. J. Puglisi, "Approximate pattern matching in LZ77-compressed texts," *JDA*, vol. 32, pp. 64–68, 2015.

[9] J. Fischer, T. Gagie, P. Gawrychowski, and T. Kociumaka, "Approximating LZ77 via small-space multiple-pattern matching," in *Proc. ESA*, 2015, pp. 533–544.

[10] J. Fischer, T. I, and D. Köppl, "Lempel Ziv Computation In Small Space (LZ-CISS)," in *Proc. CPM*, 2015, pp. 172–184.

[11] A. Policriti and N. Prezza, "Fast online Lempel-Ziv factorization in compressed space," in *Proc. SPIRE*, 2015, pp. 13–20.

[12] A. Policriti and N. Prezza, "LZ77 computation based on the run-length encoded BWT," *Algorithmica*, vol. 79, pp. 1–26, 2017.

[13] T. Nishimoto, T. I, S. Inenaga, H. Bannai, and M. Takeda, "Dynamic index and lz factorization in compressed space," in *Proc. PSC*, 2016, pp. 158–170.

[14] P. Bille, M. B. Ettienne, I. L. Gørtz, and H. W. Vildhøj, "Time-space trade-offs for lempel-ziv compressed indexing," *TCS*, vol. 713, pp. 66 – 77, 2018.

[15] M. Farach and M. Thorup, "String Matching in Lempel-Ziv Compressed Strings," *Algorithmica*, vol. 20, no. 4, pp. 388–404, 1998.

[16] A. Lempel and J. Ziv, "On the complexity of finite sequences," *IEEE Trans. Info. Theory*, vol. 22, no. 1, pp. 75–81, 1976.

[17] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Trans. Info. Theory*, vol. 23, no. 3, pp. 337–343, May 1977.

[18] M. Crochemore, L. Ilie, and W. F. Smyth, "A simple algorithm for computing the Lempel Ziv factorization," in *Proc. DCC*, 2008, pp. 482–488.

[19] J. Kärkkäinen, D. Kempa, and S. J. Puglisi, "Linear time Lempel-Ziv factorization: Simple, fast, small," in *Proc. CPM*, 2013, pp. 189–200.

[20] Philip Bille, Mikko Berggren Etienne, and Inge Li Gørtz, "Mergeable dictionaries with shifts," *arXiv preprint arXiv:1901.00718*, 2019.

[21] John Iacono and Özgür Özkan, "Mergeable dictionaries," in *Proc. ICALP*, 2010, pp. 164–175.

[22] J. Kärkkäinen and E. Ukkonen, "Lempel-Ziv parsing and sublinear-size index structures for string matching," in *Proc. WSP*, 1996, pp. 141–155.