Rpair: Rescaling RePair with Rsync [⋆]

Travis Gagie[1,2], Tomohiro I[3], Giovanni Manzini[4], Gonzalo Navarro[1,5], Hiroshi Sakamoto[3], and Yoshimasa Takabatake[3]

[1] CeBiB — Center for Biotechnology and Bioengineering, Chile
[2] Faculty of Computer Science, Dalhousie University, Canada
[3] Department of Artificial Intelligence, Kyushu Institute of Technology, Fukuoka, Japan
[4] Department of Science and Technological Innovation, University of Eastern Piedmont, Alessandria, Italy
[5] Department of Computer Science, University of Chile, Santiago, Chile

Abstract. Data compression is a powerful tool for managing massive but repetitive datasets, especially schemes such as grammar-based compression that support computation over the data without decompressing it. In the best case such a scheme takes a dataset so big that it must be stored on disk and shrinks it enough that it can be stored and processed in internal memory. Even then, however, the scheme is essentially useless unless it can be built on the original dataset reasonably quickly while keeping the dataset on disk. In this paper we show how we can preprocess such datasets with context-triggered piecewise hashing such that afterwards we can apply RePair and other grammar-based compressors more easily. We first give our algorithm, then show how a variant of it can be used to approximate the LZ77 parse, then leverage that to prove theoretical bounds on compression, and finally give experimental evidence that our approach is competitive in practice.

1 Introduction

Dictionary compression has proved to be an effective tool to exploit the repetitiveness that most of the fastest-growing datasets feature [24]. Lempel-Ziv (LZ77 for short) [23,33] stands out as the most popular and effective compression method for repetitive texts. Further, it can be run in linear time and even in external memory [18]. LZ77 has the important drawback, however, that accessing random positions of the compressed text requires, essentially, to decompress it from the beginning. Therefore, it is not suitable to be used as a *compressed data structure* that represents the text in little space while simulating direct access to it. Grammar compression [19] is an alternative that offers better guarantees in

[⋆] Partially funded with Basal Funds FB0001, Conicyt, Chile. Partially funded by JST CREST Grant Number JPMJCR1402 (TI, HS, YT), KAKENHI Grant Numbers 19K20213 (TI), 17H01791 (HS), 18K18111 (YT). Partially funded by PRIN Grant Number 2017WR7SHH and by the LSBC_19-21 Project from the University of Eastern Piedmont (GM).

this sense. The aim is to build a small context-free grammar (or Straight-Line Program, SLP) that generates (only) the text. The smallest SLP generating a text is always larger than its LZ77 parse, but only by a logarithmic factor that is rarely reached in practice. With an SLP we can access any text substring with only an additive logarithmic time penalty [3,5], which has led to the development of various self-indexes building on SLPs [4,9,12,13,15,26]. Many other richer queries on sequences have also been supported by associating summary information with the nonterminals of the SLP [1,2,5,7,11,10]. There are applications in which SLPs are preferable to LZ77 for other reasons, as well; see, e.g., [22,25].

Although finding the smallest SLP for a text is NP-complete [8,28], there are several grammar construction algorithms that guarantee at most a logarithmic blowup on the LZ77 parse [8,16,17,28,29]. In practice, however, they are sharply outperformed by RePair [21], a heuristic that runs in linear time and obtains grammars of size very close to that of the LZ77 parse in most cases. This has made RePair the compressor of choice to build grammar-based compressed data structures [1,7,10,11]. A serious problem with RePair, however, is that, despite running in linear time and space, in practice the constant of proportinality is high and it can be built only on inputs that are about one tenth of the available memory. This significantly hampers its applicability on large datasets.

In this paper we introduce a scalable SLP compression algorithm that uses space very close to that of RePair and can be applied on very large inputs. We prove a constant-approximation factor with respect to any SLP construction algorithm to which our technique is applied. Our experimental results show that we can compress a very repetitive 50GB text in less than an hour, using less than 650MB of RAM and obtaining very competitive compression ratios.

2 Preliminaries

For the sake of brevity, we assume the reader is familiar with SLPs, LZ77, and the links between the two. To prove theoretical bounds for our approach, we consider a variant of LZ77 in which if $S[i..j]$ is a phrase then either $i = j$ and $S[i]$ is the first occurrence of a distinct character, or $S[i..j]$ occurs in $S[1..j-1]$ and $S[i..j+1]$ does not occur in $S[1..j]$. We refer to this variant as LZSS due to its similarity to Storer and Szymanski's version of LZ77 [30], even though they allow substrings to be stored as raw text and we do not.

The best-known algorithm for building SLPs is probably RePair [21], for which there are many implementations (see [14] and references therein). It works by repeatedly finding the most common pair of symbols and replacing them with a new non-terminal. Although it is not known to have a good worst-case approximation ratio with respect to the size of LZ77 parsing, in practice it outperforms other constructions. RePair uses linear time and space but the coefficient in the space bound is quite large and so the standard implementations are practical only on small inputs. A more recent and more space economical alternative to RePair is SOLCA [31] that we will consider in Section 5.

Algorithm 1 Rpair: use Rsync parsing to build an SLP for a given string S

```
1. build an Rsync dictionary and parse for $S$;
2. generate SLPs for the distinct blocks as follows:
   (a) append a unique separator character to each block in the dictionary and then
       concatenate the blocks (in the order of their first appearances in $S$) into a
       string $D$;
   (b) build an SLP for $D$;
   (c) delete from the SLP any non-terminal that occurs only once in the parse tree
       (and any rule including it);
   (d) delete from the SLP the separator characters (and any rules including them);
   (e) list the non-terminals at the roots of the maximal remaining subtrees of the
       parse tree;
   (f) divide the list into sublists such that the concatenation of the expansions of
       the non-terminals in the $i$th sublist is the $i$ block in $D$;
   (g) create a set of rules generating the $i$th sublist from a new non-terminal $X_i$;
3. build an SLP for the parse $P$;
4. replace by $X_i$ each occurrence in $P$ of the terminal for the $i$th block in $D$;
5. combine the SLP for $P$ with the SLPs for the blocks.
```

Context-triggered piecewise hashing (CTPH) is a technique for parsing strings into blocks such that long repeated substrings are parsed the same way (except possibly at the beginning or end of the substrings). The name CTPH seems to be due to Kornblum [20] but the ideas go back to Tridgell's Rsync [32] and Spamsum (`https://www.samba.org/ftp/unpacked/junkcode/spamsum/README`): "The core of the spamsum algorithm is a rolling hash similar to the rolling hash used in 'rsync'. The rolling hash is used to produce a series of 'reset points' in the plaintext that depend only on the immediate context (with a default context width of seven characters) and not on the earlier or later parts of the plaintext."

Specifically, in this paper we choose a rolling hash function and a threshold p, run a sliding window of fixed size w over S and end the current block whenever the window contains a triggering substring, which is a substring of length w whose hash is congruent to 0 modulo p. When we end a block, we shift the window ahead w characters so all the blocks are disjoint and form a parse, which we call the *Rsync parse*. We call the set of distinct blocks the *Rsync dictionary*: if the input text contains many repetitions, we expect the dictionary to be much smaller than the text.

3 Algorithms

Given a string S, we can use Rsync parsing to help build an SLP for S with Algorithm 1 ("Rpair"). The final SLP can be viewed as first generating the parse, then replacing each block ID in the parse by the sublist of non-terminals that generate each block, and finally replacing the sublists by the blocks themselves.

Since each separator character appears only once in D and its parse tree, any non-terminal whose expansion includes a separator character also appears

Algorithm 2 Rparse: use Rsync to build an LZSS-like parse for a string S

```
1. build an Rsync dictionary and parse for $S$;
2. append a unique separator character to each block in the dictionary and concatenate the blocks (in the order of their first appearances in $S$) into a string $D$;
3. compute the LZSS parse of $D$;
4. compute the LZSS parse of the parse $P$, treating each block as a meta-character;
5. map $D$'s and $P$'s parses onto $S$:
    (a) discard any separator character $D[j]$ in $D$;
    (b) turn the first occurrence $D[j]$ of any other character in $D$ into the first occurrence $S[j']$ of that character in $S$;
    (c) turn each phrase $D[j..j+\ell-1]$ in block $B$ with source $D[i..i+\ell-1]$ in block $B'$, into a phrase $S[j'..j'+\ell-1]$ with source $S[i'..i'+\ell-1]$, where $S[j']$ and $S[i']$ have the same respective offsets from the beginnings of the first occurrences of $B$ and $B'$ in $S$, as $D[j]$ and $D[i]$ have from the beginnings of $B$ and $B'$ in $D$;
    (d) discard the first occurrence $P[j]$ of each block in $P$;
    (e) turn each phrase $P[j..j+\ell-1]$ with source $P[i..i+\ell-1]$, into a phrase $S[j'..j'+\ell'-1]$ with source $S[i'..i'+\ell'-1]$, where $S[j']$ and $S[i']$ are the first characters in the $j$th and $i$th blocks, respectively, and $\ell'$ is the total length of the $j$th through $(j+\ell-1)$st blocks (and thus also the total length of the $i$th through $(i+\ell-1)$st blocks).
```

only once and is deleted. Since the parse tree of an SLP is binary and each non-terminal we delete appears only once, the number of distinct non-terminals we delete is at least the length of the list of non-terminals at the roots of the maximal remaining subtrees of the parse tree, minus one. Therefore, creating rules to generate the sublists does not cause the number of distinct non-terminals to grow to more than the number in the original SLP for D, plus one.

Algorithm 1 works with any algorithm for building SLPs for D and P. In Section 4 we show that, if we choose an algorithm that builds SLPs for D and P at most an α-factor larger than their LZ77 parses, then we obtain an SLP an $O(\alpha)$-factor larger than the LZ77 parse of S. In the process we will refer to Algorithm 2 ("Rparse"), which produces an LZSS-like parse of S but is intended only to simplify our analysis of Algorithm 1 (not to compete with cutting-edge LZ-based compressors). By "LZSS-like" we mean a parse in which each phrase is either a single character that has not occurred before, or a copy of an earlier substring. We note in passing that, if the parse in Step 3 is still too big for a normal construction, then we can apply Algorithm 1 to it. We will show in the full version of this paper that, if we recurse only a constant number of times, then we worsen our compression bounds by only a constant factor.

4 Analysis

The main advantage of using Rsync parsing to preprocess S is that Rsync parsing is quite easy to parallelize, apply over streamed data, or apply in external memory. The resulting dictionary and parse may be significantly smaller than

S, making it easier to apply grammar-based compression. In the full version of this paper we will analyze how much time and workspace Algorithms 1 and 2 use in terms of the total size of the dictionary and parse, but for now we are mainly concerned with the quality of the compression.

Let b be the number of distinct blocks in the Rsync parse of S, and let z be the number of phrases in the LZ77 parse of S. The first block is obviously the first occurrence of that substring and if $S[i..j]$ is the first occurrence of another block, then $S[i-w..j]$ (i.e., the block extended backward to include the previous triggering substring) is the first occurrence of that substring. Since the first occurrence of any non-empty substring overlaps or ends at a phrase boundary in the LZ77 parse, we can charge $S[i..j]$ to such a boundary in $S[i-w..j]$. Since blocks have length at least w and overlap by only w characters when extended backwards, each boundary has the first occurrences of at most two blocks charged to it, so $b = O(z)$.

In Step 5 of Algorithm 2, we discard $O(b)$ of the phrases in the LZSS parses of D and P when mapping to the phrases in the LZSS-like parse of S. Therefore, by showing that the number of phrases in the LZSS-like parse of S is $O(z)$, we show that the total number of phrases in the LZSS parses of D and P is also $O(z + b) = O(z)$, so the total number of phrases in their LZ77 parses is $O(z)$ as well.

Lemma 1. *If the t-th phrase in the LZSS parse of S is $S[j..j+\ell-1]$ then the $5t$-th phrase resulting from Algorithm 2, if it exists, ends at or after $S[j+\ell-1]$.*

Proof. Our claim is trivially true for $t = 1$, since the first phrases in both parses are the single character $S[1]$, so let t be greater than 1 and assume our claim is true for $t-1$, meaning the $5(t-1)$st phrase in our parse ends at $S[k-1]$ with $k \geq j$. If $k \geq j+\ell$ then our claim is also trivially true for t, so assume $j \leq k < j+\ell$. We must show that our parse divides $S[k..j+\ell-1]$ into at most five phrases, in order to prove our claim for t.

First suppose that $S[k..j+\ell-1]$ does not completely contain a triggering substring, so it overlaps at most two blocks. (It can overlap two blocks without containing a triggering substring if and only if a prefix of length less than w lies in one block and the rest lies in the next block.) Let $S[i..i+\ell-1]$ be $S[j..j+\ell-1]$'s source and let $k' = i+k-j$, so in the LZSS parse $S[k..j+\ell-1]$ is copied from $S[k'..i+\ell-1]$. Since $S[k'..i+\ell-1]$ does not completely contain a triggering substring either, it too overlaps at most two blocks.

Without loss of generality (since the other cases are easier), assume $S[k..j+\ell-1]$ and $S[k'..i+\ell-1]$ each overlap two blocks and they are split differently: $S[k..k+d-1]$ lies in one block and $S[k+d..j+\ell-1]$ lies in the next, and $S[k'..k'+d'-1]$ lies in one block and $S[k'+d'..i+\ell-1]$ in the next, with $d \neq d'$. Assume also that $d < d'$, since the other case is symmetric. Since $S[k..k+d-1]$ is completely contained in a block and occurs earlier completely contained in a block, as $S[k'..k'+d-1]$, our parse does not divide it. Similarly, since $S[k+d..k+d'-1]$ and $S[k+d'..j+\ell-1]$ are each completely contained in a block and occur earlier each completely contained in a block, as $S[k'+d..k'+d'-1]$

and $S[k'+d'..i+\ell-1]$, respectively, our parse does not divide them. Therefore, our parse divides $S[k..j+\ell-1]$ into at most three phrases.

Now suppose the first and last triggering substrings completely contained in $S[k..j+\ell-1]$ are $S[x..x+w-1]$ and $S[y..y+w-1]$ (possibly with $x=y$). By the arguments above, our parse divides $S[k..x+w-1]$ into at most three phrases. Since $S[x+w..y+w-1]$ is a sequence of complete blocks that have occurred earlier (in $S[k'..i+\ell-1]$), our parse does not divide it unless $S[k..x+w-1]$ is a complete block that has occurred before as a complete block, in which case it may divide $S[k..y+w-1]$ once between $S[x+w]$ and $S[y+w-1]$. Since $S[y+w..j+\ell-1]$ is completely contained in a block and occurs earlier completely contained in a block (in $S[k'..i+\ell-1]$), our parse does not divide it. Therefore, our parse divides $S[k..j+\ell-1]$ into at most five phrases. □

We note that we can quite easily can reduce the five in Lemma 1, at the cost of complicating our algorithm slightly. We leave a detailed analysis for the full version of this paper.

Corollary 1. *Algorithm 2 yields an LZSS-like parse of S with at most five times as many phrases as its LZSS parse.*

Proof. If the LZSS parse has t phrases then the t-th phrase ends at $S[n]$ so, by Lemma 1, Algorithm 2 yields a parse with at most $5t$ phrases. □

Theorem 1. *Algorithm 2 yields an LZSS-like parse of S with $O(z)$ phrases.*

Proof. It is well known that the LZSS parse of S has at most twice as many phrases as the its LZ77 parse (since dividing each LZ77 phrase into a prefix with an earlier occurrence and a mismatch character yields an LZSS-like parse with at most twice as many phrases, and the LZSS parse has the fewest phrases of any LZSS-like parse). Therefore, by Corollary 1, Algorithm 2 yields a parse with at most $O(z)$ phrases. □

Corollary 2. *The LZ77 parses of D and P have $O(z)$ phrases.*

Proof. Immediate, from Theorem 1, the fact that the LZ77 parse is no larger than the LZSS parse, and inspection of Algorithm 1. □

Let A be any algorithm that builds an SLP at most an α-factor larger than the LZ77 parse of its input. For example, with Rytter's construction [28] we have $\alpha = O(\log(|S|/z))$.

By Corollary 2, applying A to D — Step 2b in Algorithm 1 — yields an SLP for D with $O(\alpha z)$ rules. As explained in Section 3, Steps 2c to 2g then increase the number of rules by at most one while modifying the SLP such that, for each block in the dictionary, there is a non-terminal whose expansion is that block.

Similarly, applying A to P — Step 3 — yields an SLP for P with $O(z)$ rules. Replacing the terminals in the SLP by the non-terminals generating the blocks and then combining the two SLPs — Steps 4 and 5 — yields an SLP for S with $O(\alpha z)$ rules. This gives us our main result of this section:

Theorem 2. *Using A in Steps 2b and 3 of Algorithm 1 yields an SLP for S with $O(\alpha z)$ rules.*

5 Experiments

We use two genome collections in our experiments: cN consists of N concatenated variants of the human chromosome chr19, of about 59MB each; sN consists of N concatenated variants of salmonella genomes, of widely different sizes.

The chr19 collection was downloaded from the 1000 Genomes Project. Each chr19 sequence was derived by using the bcftools consensus tool to combine the haplotype-specific (maternal or paternal) variant calls for an individual with the chr19 sequence in the GRCH37 human reference. The salmonella genomes were downloaded from NCBI (BioProject PRJNA183844) and preprocessed by assembling each individual sample with IDBA-UD [27] setting kMaxShortSequence to 1024 per public advice from the author to accommodate the longer paired end reads that modern sequencers produce. More details of the collections are available in previous work [6, Sec. 4].

We compare two grammar compressors: RePair [21] produces the best known compression ratios but uses a lot of main memory space, whereas SOLCA [31] aims at optimizing main memory usage. Their versions combined with parallelized CTPH parsing are BigRepair and BigSOLCA. RePair could be run only on the smaller collections. Our experiments ran on a Intel(R) I7-4770 @ 3.40 GHz machine with 32 GB memory using 8 threads; currently only the CTPH parsing takes advantage of the multiple threads.

For RePair we use Navarro's implementation for large files, at `http://www.dcc.uchile.cl/gnavarro/software/repair.tgz`, letting it use 10GB of main memory, whereas the implementation of SOLCA is at `https://github.com/tkbtkysms/solca`. To measure their compression ratios in a uniform way, we consider the following encodings of their output: if RePair produces r (binary) rules and an initial rule of length c, we account $2r$ bits to encode the topology of the pruned parse tree (where the nonterminal ids become the preorder of their internal node in this tree) and $(r+c)\lceil\log_2 r\rceil$ bits to encode the leaves of the tree and the initial rule. SOLCA is similar, with $c = 1$. Our code is available at `https://gitlab.com/manzai/bigrepair`.

Table 1 shows the results in terms of compression ratio, time, and space in RAM. On the more repetitive chr19 genomes, BigRePair is clearly the best choice for large files. It loses to RePair in compression ratio, but RePair takes 11 hours just to process 5.5GB, so it is not a choice for larger files. Instead, BigRepair processes 55GB in about 20 minutes and 6.5GB. Similarly, SOLCA obtains better compression but more compression time than BigSOLCA, though the latter uses more space. The comparison between the two compressors shows that BigRepair performs better than both SOLCA and BigSOLCA in both compression ratio (reaching nearly half the compressed size of SOLCA on the largest files) and time (half the time of BigSOLCA). Still SOLCA uses much less space: it compresses 55GB in 3.6 hours, but using less than 750MB.

The results start similarly on the less compressible salmonella collection, but, as the size of the input grows, there are significant differences. The time of BigRePair on chr19 was stable around 2GBs per minute, but on salmonella it is not: When moving from 10GB to 20GB of input data, the time per processed

File	Size	RePair			BigRePair			SOLCA			BigSOLCA		
		Ratio	Time	Spc	Ratio	Time	Spc	Ratio	Time	Spc	Ratio	Time	Spc
c50	2.75	0.80%	1832	3842	0.91%	29.30	454.7	1.35%	244.1	107.4	1.54%	66.47	183.4
c100	5.51	0.30%	7311	3155	0.48%	25.05	246.4	0.77%	236.4	53.67	0.86%	56.96	130.4
c250	13.8				0.23%	22.10	119.8	0.40%	239.0	29.78	0.44%	48.55	95.00
c500	27.5				0.14%	22.31	118.0	0.28%	237.4	17.05	0.30%	47.46	84.72
c1000	55.1				0.10%	22.61	117.3	0.22%	237.3	13.56	0.23%	47.79	78.82
s815	3.75	1.72%	8478	3726	1.93%	51.70	2254	3.01%	317.7	161.0	3.50%	104.1	291.4
s2073	9.72				2.01%	55.48	1055	3.01%	370.9	153.1	3.53%	116.9	285.9
s4570	22.0				2.61%	201.1	534.2	3.57%	480.6	154.4	4.24%	142.8	335.1
s11264	53.1				1.51%	2560	294.2	2.20%	620.2	92.60	2.61%	113.1	206.7

Table 1. Performance of the compressors. File sizes are expressed in GB, compression ratios in percentage of compressed file over uncompressed file, compression times in seconds per input GB, and compression main memory usage in MBs per input GB.

GB of BigRePair jumps by a factor of 3.6, and when moving from 20GB to 50GB it jumps by more than 10. To process the largest 53GB file, BigRePair requires more than 37 hours and over 15 GB of RAM. SOLCA, instead, handles this file in nearly 9 hours and less than 5 GB, and BigSOLCA in less than 2 hours and 11 GB, being the fastest. What happens is that, being less compressible, the output of the CTPH parse is still too large for RePair, and thus it slows down drastically as soon as it cannot fit its structures in main memory. The much lower memory footprint of SOLCA, instead, pays off on these large and less compressible files, though its compression ratio is worse than that of BigRePair. In the full version of this paper we will investigate applying BigRePair and BigSOLCA recursively, following the strategy mentioned at the end of Section 3.

References

1. A. Abeliuk, R. Cánovas, and G. Navarro. Practical compressed suffix trees. *Algorithms*, 6(2):319–351, 2013.
2. H. Bannai, T. Gagie, and T. I. Online LZ77 parsing and matching statistics with RLBWTs. In *CPM*, pages 7:1–7:12, 2018.
3. D. Belazzougui, P. H. Cording, S. J. Puglisi, and Y. Tabei. Access, rank, and select in grammar-compressed strings. In *ESA*, pages 142–154, 2015.
4. P. Bille, M. B. Ettienne, I. L. Gørtz, and H. W. Vildhøj. Time-space trade-offs for Lempel-Ziv compressed indexing. In *CPM*, pages 16:1–16:17, 2017.
5. P. Bille, G. M. Landau, R. Raman, K. Sadakane, S. S. Rao, and O. Weimann. Random access to grammar-compressed strings and trees. *SIAM J. Comput.*, 44(3):513–539, 2015.
6. C. Boucher, T. Gagie, A. Kuhnle, and G. Manzini. Prefix-free parsing for building big BWTs. In *WABI*, pages 2:1–2:16, 2018.
7. N. Brisaboa, A. Gómez-Brandón, G. Navarro, and J. Paramá. Gract: A grammar-based compressed index for trajectory data. *Inf. Sci.*, 483:106–135, 2019.

8. M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat. The smallest grammar problem. *IEEE Trans. Inf. Theory*, 51(7):2554–2576, 2005.
9. A. R. Christiansen and M. B. Ettienne. Compressed indexing with signature grammars. In *LATIN*, pages 331–345, 2018.
10. F. Claude, A. Fariña, M. Martínez-Prieto, and G. Navarro. Universal indexes for highly repetitive document collections. *Inf. Sys.*, 61:1–23, 2016.
11. F. Claude and J. I. Munro. Document listing on versioned documents. In *SPIRE*, pages 72–83, 2013.
12. F. Claude and G. Navarro. Self-indexed grammar-based compression. *Fund. Inf.*, 111(3):313–337, 2010.
13. F. Claude and G. Navarro. Improved grammar-based compressed indexes. In *SPIRE*, pages 180–192, 2012.
14. I. Furuya, T. Takagi, Y. Nakashima, S. Inenaga, H. Bannai, and T. Kida. MR-RePair: Grammar compression based on maximal repeats. In *DCC*, pages 508–517, 2019.
15. T. Gagie, P Gawrychowski, J. Kärkkäinen, Y. Nekrich, and S. J. Puglisi. LZ77-based self-indexing with faster pattern matching. In *LATIN*, pages 731–742, 2014.
16. A. Jeż. Approximation of grammar-based compression via recompression. *Theor. Comp. Sci.*, 592:115–134, 2015.
17. A. Jeż. A really simple approximation of smallest grammar. *Theor. Comp. Sci.*, 616:141–150, 2016.
18. J. Kärkkäinen, D. Kempa, and S. J. Puglisi. Lempel-Ziv parsing in external memory. In *DCC*, pages 153–162, 2014.
19. J. C. Kieffer and E.-H. Yang. Grammar-based codes: A new class of universal lossless source codes. *IEEE Trans. Inf. Theory*, 46(3):737–754, 2000.
20. J. D. Kornblum. Identifying almost identical files using context triggered piecewise hashing. *Digital Investigation*, 3(Supplement-1):91–97, 2006.
21. J. Larsson and A. Moffat. Off-line dictionary-based compression. *Proceedings of the IEEE*, 88(11):1722–1732, 2000.
22. R. Lasch, I. Oukid, R. Dementiev, N. May, S. S. Demirsoy, and K.-U. Sattler. Fast & strong: The case of compressed string dictionaries on modern CPUs. In *DaMoN*, pages 4:1–4:10, 2019.
23. A. Lempel and J. Ziv. On the complexity of finite sequences. *IEEE Trans. Inf. Theory*, 22(1):75–81, 1976.
24. G. Navarro. Indexing highly repetitive collections. In *IWOCA*, pages 274–279, 2012.
25. C. G. Nevill-Manning and I. H. Witten. Identifying hierarchical structure in sequences: A linear-time algorithm. *J. Artif. Intell. Res.*, 7:67–82, 1997.
26. T. Nishimoto, T. I, S. Inenaga, H. Bannai, and M. Takeda. Dynamic index, LZ factorization, and LCE queries in compressed space. *CoRR*, abs/1504.06954, 2015.
27. Y. Peng, H. C. M. Leung, S. M. Yiu, and F. Y. L. Chin. IDBA-UD: a de novo assembler for single-cell and metagenomic sequencing data with highly uneven depth. *Bioinformatics*, 28(11):1420–1428, 2012.
28. W. Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theor. Comp. Sci.*, 302(1-3):211–222, 2003.
29. H. Sakamoto. A fully linear-time approximation algorithm for grammar-based compression. *J. Discr. Alg.*, 3(24):416–430, 2005.
30. J. A. Storer and T. G. Szymanski. Data compression via textual substitution. *J. ACM*, 29(4):928–951, 1982.

31. Y. Takabatake, T. I, and H. Sakamoto. A space-optimal grammar compression. In *ESA*, pages 67:1–67:15, 2017.
32. A. Tridgell. *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, The Australian National University, 1999.
33. J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. on Inf. Theory*, IT-23(3):337–349, 1977.