Relative Lempel-Ziv Compression of Suffix Arrays

Puglisi, Simon J.

Puglisi , S J & Zhukova , B 2020 , Relative Lempel-Ziv Compression of Suffix Arrays . in C
Boucher & S V Thankachan (eds) , String Processing and Information Retrieval : 27th
þÿInternational Symposium, SPIRE 2020, Orlando, FL, USA, October 13 1
Proceedings . Lecture Notes in Computer Science (including subseries Lecture Notes in
Artificial Intelligence and Lecture Notes in Bioinformatics) , vol. 12303 LNCS , Springer
Science and Business Media Deutschland GmbH , Cham , pp. 89-96 , International
Symposium on String Processing and Information Retrieval , Orlando , Florida , United
States , 13/10/2020 . https://doi.org/10.1007/978-3-030-59212-7_7

acceptedVersion

Relative Lempel-Ziv Compression of Suffix Arrays[*]

Simon J. Puglisi and Bella Zhukova

Department of Computer Science, University of Helsinki
Helsinki Institute for Information Technology (HIIT)
Helsinki, Finland
{simon.puglisi,bella.zhukova}@helsinki.fi

Abstract. We show that a combination of differential encoding, random sampling, and relative Lempel-Ziv (RLZ) parsing is effective for compressing suffix arrays, while simultaneously allowing very fast decompression of arbitrary suffix array intervals, facilitating pattern matching. The resulting text index, while somewhat larger (5-10x) than the recent r-index of Gagie, Navarro, and Prezza (Proc. SODA '18) — still provides significant compression, and allows pattern location queries to be answered more than two orders of magnitude faster in practice.

1   Introduction

The suffix array [18], $\mathsf{SA}[0..n-1]$, of a text (or string, or sequence) T of length n is an array of integers containing a permutation of $(0 \ldots n-1)$, so that the suffixes of T starting at the consecutive positions indicated in SA are in lexicographical order: $\mathsf{T}[\mathsf{SA}[i]..n] < \mathsf{T}[\mathsf{SA}[i+1]..n]$. Because of the lexicographic ordering, all the suffixes starting with a given substring P of T form an interval $\mathsf{SA}[s..e]$, which can be determined by binary search in $O(|\mathsf{P}| \log n)$ time. The suffix array is thus an efficient data structure for returning all positions in T where a query pattern Q occurs; once s and e are located for $\mathsf{P} = \mathsf{Q}$, it is simple to enumerate the $occ = e - s + 1$ occurrences of Q.

An alternative to binary search is the so-called *backward search* method, which locates the interval of the SA via $2|\mathsf{P}|$ rank queries on the Burrow-Wheeler transform (BWT) of T [6,7]. Backward search is the basis for compressed text indexing, emplified by the FM-index family, which has been widely adopted in practice, for example, in Bioinformatics [17]. The BWT is easily amenable to compression (while still supporting rank queries), and so the challenge then has been to reduce the space required for the SA below its trivial $n \log n$-bit encoding, for which a handful of techniques have emerged in the past two decades. The most longstanding of these is to explicitly store the position of every bth suffix in lexicographical (i.e., SA) order. With these samples in hand, rank queries on BWT (a process called "LF mapping") allow an arbitrary $\mathsf{SA}[i]$ value can

---

be determined in $O(b)$ time, thus allowing all occurrences of a pattern to be obtained in $O(b \cdot occ)$ time, with $O(n/b)$ extra space used for the suffix samples.

Very recently, Gagie, Navarro, and Prezza [8,9], exploiting an ingenious observation, showed how this can be improved to $O(occ \cdot \log \log n)$ time. They call the resulting data structure the r-index. Experiments in [8] show this improvement is not only of theoretical interest: in practice the r-index is around two orders of magnitude faster than indexes that use regular suffix sampling, and always less space consuming. Another recent alternative is the succinct compact acyclic word graph of Belazzougui, Cunial, Gagie, Prezza, and Raffinot [1], which in practice can be significantly faster than the r-index, but is much bigger (albeit much smaller than the $n \log n$ bits required by the plain SA).

Contribution. The contribution of this short paper is to show that, in practice (at least), *relative Lempel-Ziv parsing* is an effective way to compress the suffix array, and one that supports decompression of intervals especially fast. Our starting point is the differentially encoded SA, denoted SA^d, as first introduced by Gonzalez and Navarro [11]. We then derive an RLZ dictionary, R, (usually called the *reference sequence* [14]), by randomly sampling subarrays from SA^d, and parse SA^d into phrases relative to R. Supporting random access is then a matter of storing one original SA value for each phrase (to undo the differential encoding) and storing the phrase starting points in a predecessor data structure. Decompressing *occ* consecutive values from SA can then be performed in essentially $\mathrm{O}(\log \log n + occ)$ time, and is very fast in practice: more than 100 times faster than the r-index [8] and the CDAWG [1], which are the fastest published methods. Depending on the dataset, our index uses 5-15 times more space than the r-index, and less than the CDAWG.

We acknowledge our approach is uncomplicated, and is essentially a new combination of known techniques: as noted above, dictionary compression of differentially encoded SAs has been explored previously by Gonzalez and Navarro [11], where they used the RePair grammar compressor [15] rather than RLZ (which was undiscovered at the time). Furthermore, RLZ is widely known to support fast random access to its underlying data, but to date has only been applied to textual data, be it natural language [13,3,16] or genomic [14,3]. However, as our experiments show, this combination turns out to be extremely effective, representing a new point on the pareto curve, and seems to simply have been overlooked to date. Another piece of related work is the relative suffix tree of Farruggia et al. [5], in which one or more suffix arrays are compressed *relative to another suffix array*, and pattern matching is supported on each individual SA. That work is different to ours in that we deal with compression of a single SA.

Our own interest in SA compression comes from our recent work developing fast indexes for gapped matching [2]. These indexes rely for their efficiency on fast scans of suffix array intervals, which is easy on an uncompressed SA, but lose significant throughput when current compressed SA implementations are used. The RLZ-compressed suffix array we describe in this paper allows us to derive

compressed forms of our gapped-matching indexes that use much less space but operate at comparable speed to uncompressed ones.

Roadmap. In the following section we review the differentially encoded SA of Navarro and Gonzalez [11,12] and the way it induces a sequences containing repetitions, which can then be exploited by a dictionary compressor. We also review relative Lempel-Ziv parsing [14], before describing our data structure and the way in which it supports fast subarray access. We then report on an experimental comparison of a prototype of our index, dubbed rlzsa, with the r-index and the CDAWG [1] — which represent, to our knowledge, the current state of the art. Conclusions and reflections are then offered.

2   New Locate Index

SA contains a permutation of the integers $(0 \ldots n-1)$ and so is not directly amenable to dictionary compression in the same way that, say, the text T would be — it contains no repeated elements. SA does contain repetitions of a different nature, however. In particular, because of the lexicographical order on the suffixes in SA if an interval of suffixes $\mathsf{SA}[x,y]$ are all preceded by the same symbol c, then there must exist another interval $\mathsf{SA}[x', x' + (y-x) + 1]$ for which $\mathsf{SA}[x] = \mathsf{SA}[x']+1, \mathsf{SA}[x+1] = \mathsf{SA}[x'+1]+1, \ldots, \mathsf{SA}[y] = \mathsf{SA}[x'x'+(y-x)+1]+1$. Navarro and Gonzalez [11] observed that these so-called *self repetitions* can be turned into actual repetitions if one differentially encodes the suffix array as $\mathsf{SA}^d[0] = \mathsf{SA}[0]$ and $\mathsf{SA}^d[i] = (\mathsf{SA}[i] - \mathsf{SA}[i-1] + n)$ for $i \geq 1$. Note that the "$+n$" is for technical convenience, so that all values in SA^d are positive.

Navarro and Gonzalez [11] (see also their later journal paper [12] apply a grammar compressor to SA^d, augmenting the grammar with additional pointers to facilitate random access to values in SA^d, and storing original SA values at regular intervals so that the differential encoding can be reversed.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
T	a	c	t	a	g	a	c	t	a	g	a	c	t	a	g	a	c	t	a	g	a	c	t	a	g	a	c	t	a	g	a	$
SA	31	30	25	20	15	10	5	0	28	23	18	13	8	3	26	21	16	11	6	1	29	24	19	14	9	4	27	22	17	12	7	2
SA^d	31	31	27	27	27	27	27	27	60	27	27	27	27	27	55	27	27	27	27	27	60	27	27	27	27	27	55	27	27	27	27	27

reference	27	27	27	27	27	27	55	27	27	27	27	27	27	27	27	27

phrases	31	30	10	28	1	29	1	-	0	1	2	8	9	20	21	32

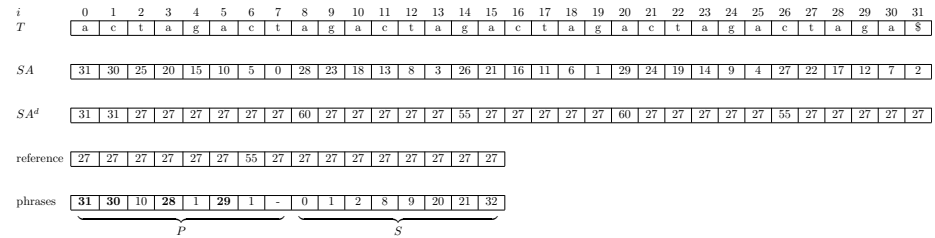(P: first 8 phrase entries; S: last 8 phrase entries)

Fig. 1: An example illustrating components of our data structure.

Figure 1 shows a small example illustrating the different components of our data structure and the intermediate stages in their construction.

RLZ parsing. A variant of the classic LZ77 parsing [21], RLZ parsing compresses a sequence X relative to a second sequence R (the reference) by encoding X as

a sequence of substrings, or *phrases*, that occur in R. Our data structure is built on an atypical form of RLZ parsing that is critical to support efficient access to subarrays of the SA and which we now describe.

We derive our reference string R by randomly sampling substrings from SA^d. In Section 3 will return to the implementation details such as the number of samples and the size of each sample, but for the time being let us assume R is in hand. References built by random sampling have been shown to work well in practice for compressing web corpora [13] and non-trival bounds on their size have also since been proved [10].

We encode SA by parsing SA^d into phrases — represented as integer pairs — that either represent literal values from the original SA (*literal phrases*), or point to substrings that occur in the reference sequence R (*repeat phrases*). The first component of the pair is always the starting position in SA^d (equivalently SA) of the phrase. A literal phrase at position i is represented as $(i, \mathsf{SA}[i])$. The first phrase is always the literal phrase $(0, \mathsf{SA}[0])$. Parsing begins at position 1 in SA^d and proceeds according to the following rule. If the parsing is up to position i in SA^d, then the next phrase is either:

- a literal phrase $(i, \mathsf{SA}[i])$, if the previous phrase was not a literal phrase or $\mathsf{SA}^d[i]$ does not occur in R; or
- the longest prefix of $\mathsf{SA}^d[i, n]$ that occurs in R.

Observe that the parsing rule ensures that every repeat phrase is preceded by a literal phrase. This allows us to easily recover the portion of the SA that is covered by a repeat phrase. Let (i, p_i) be a repeat phrase of length ℓ_i and $(i-1, x)$ be the preceding literal phrase in the parsing. Then $\mathsf{SA}[i] = \mathsf{SA}^d[i] + x = R[p_i] + x, \mathsf{SA}[i+1] = \mathsf{SA}^d[i+1] + \mathsf{SA}[i] = R[p_i+1] + \mathsf{SA}[i], \ldots, \mathsf{SA}[i+\ell_i-1] = R[p_i+\ell_i-1] + \mathsf{SA}[i+\ell_i-2]$.

Data Structure. We store the parsing in two arrays, S and P, both of length z. S contains the starting position in SA^d of each phrase in ascending order. We build and store a predecessor data structure for S. P contains either literal SA values or positions in R as output by the parsing algorithm (the second components of each pair). The length of the ith phrase can be determined as $S[i+1] - S[i]$.

Decoding a Subarray. We now describe how to decode an arbitrary interval $\mathsf{SA}[s, e]$ using our data structure. The decoded subarray will be materialized in an output buffer B of size $e - s + 1$. At a high level, we will decode the phrases covering $\mathsf{SA}[s, e]$ and copy the decoded values that belong in $\mathsf{SA}[s, e]$ (some parts of the first and last phrase may not) into B until it is full, at which point we are done. To this end, we begin by finding the index in S of the predecessor of s. Let x denote this index. If $P[x]$ is a literal phrase, we copy its value to the output buffer. Otherwise ($P[x]$ is non-literal) $P[x-1]$ is by definition literal and we set $p = P[x-1]$. The length of the phrase is $\ell = S[x+1] - S[x]$. Assuming for the moment $S[x] = s$, to decode phrase x we access $\mathsf{R}[P[x]]$, copy $(p + \mathsf{R}[P[x]] - n)$ to the output buffer, and then set $p = (p + \mathsf{R}[P[x]] - n)$, continuing then to copy $(p + \mathsf{R}[P[x] + 1] - n)$ to B, and so on until either the whole phrase has been

decoded, or the output buffer is full. Note that if $S[x] < s$, then we first decode (as described) and *discard* the $(s - P[x])$ symbols of phrase x that are before position s. After decoding phrase x, if the output buffer is not full, we continue to decode phrase $x + 1$, and so on, until all $e - s + 1$ values have been decoded.

Implementation Details. In our practical implementation, P is an array of 32-bit integers. We also limit the maximum phrase length to 2^{16}. For the predecessor data structure, we use the following two-layered approach. We sample every bth phrase starting position and store these in an array. In a separate array we store a differential encoding of all starting positions. Because of the aforementioned phrase length restriction, the array of differentially encoded starting positions takes 16 bits per entry. Predecessor search for a position x proceeds by first binary searching in the sampled array to find the predecessor sample at index i of that array. We then access the differentially encoded array starting at index ib and scan, summing values until the cummulative sum is greater than x, at which point we know the predecessor.

3   Experimental Evaluation

In this section we compare the practical performance of our rlzsa index to other leading compressed indexes, in particular the r-index of Gagie et al. [8] and the cdawg of Belazzougui et al. [1][1]. These indexes were selected because they are the best current approaches for locate queries according to experiments in [8][2]. We provide results for two variants of rlzsa, which are labelled rlzsa-rand and rlzsa-lz in the plots. The rlzsa-rand variant uses a reference constructed via random sampling substrings from the datasets (parameters below). The rlzsa-lz variant selects substrings for the reference based on a length-limited form of LZ77 parsing, which we describe in the full version of this paper.

Mirroring the experiments in [8], we measured memory usage and locate times per occurrence of all indexes on 1000 patterns of length 8 extracted from four repetitive datasets:

- DNA, an artificial dataset of 629145 copies of a DNA sequence of length 1000 (Human genome) where each character was mutated with probability 10^{-3};
- boost, a dataset of concatenated versions of the GitHub's boost library;
- einstein, a dataset of concatenated versions of Wikipedia's Einstein page;
- world, a collection of all pdf files of CIA World Leaders from January 2003 to December 2009 downloaded from the Pizza&Chili corpus.

The average number of occurrences per pattern was 89453 (boost), 607750 (DNA), 31788 (einstein), 29781 (world).

---

[1] The only implementation of cdawg works only for strings on {a,c,g,t}.

[2] We also tried unsuccessfully to include the Locally Compressed Suffix Array (LCSA) of Gonzalez, Navarro, and Farrada [12], which is based on differential encoding of the SA and RePair grammar compression. After expending significant effort attempting to get their code to work we discovered — in communication with the authors [4] — that our failure was due to known bugs in the (dated) LCSA codebase.

Test Machine and Environment. We used a 2.10 GHz Intel Xeon E7-4830 v3 CPU equipped with 30 MiB L3 cache and 1.5 TiB of main memory. The machine had no other significant CPU tasks running and only a single thread of execution was used. The OS was Linux (Ubuntu 16.04, 64bit) running kernel 4.10.0-38-generic. Programs were compiled using `g++` version 5.4.0. All given runtimes were recorded with the C++11 `high_resolution_clock` time measurement facility.

Results. The results of our experiments appear in Figure 2. On all datasets, both variants of our new rlzsa index are clearly the fastest, providing a newly relevant point on the space-time curve. We locate occurrences always at least two orders of magnitude faster than all other indexes: compared to r-index, from a minimum of 120 times on `world` to a maximum of 160 times on `DNA`. On DNA we are 100 times faster than cdawg, which is the next fastest index, and is more than twice the size of the rlzsa variants. The r-index is always the smallest index, from 5 times (world) to 14 times (DNA) smaller than rlzsa-rand.

We remark that in preliminary experiments, we observed rlzsa times to be extremely stable, and quite invariant to reference size. In the plots the rlzsa-rand variant used references size $|R|$ of 106496 (`boost`), 28597248 (`DNA`), 6417408 `einstein`, 2760704 (`world`), with the reference sequence made up of substrings of length 4096 (`boost`, `world`) or 3072 (`DNA`, `einstein`). Finally, the rlzsa-lz index is noticeably smaller than the rlzsa-rand one on the `boost` dataset, but otherwise the two rlzsa indexes are very close in size.
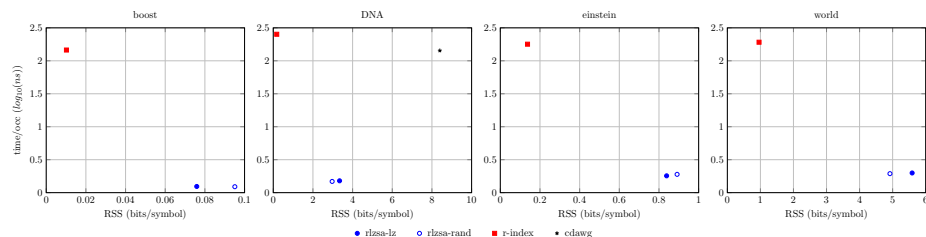

Fig. 2: Locate time per occurrence and working space (in bits per symbol) of the indexes. The vertical axis shows nanoseconds per reported occurrence and is logarithmic.

4   Concluding Remarks

We have described and tested a compressed data structure — rlzsa — that represents the suffix array and allows fast decompression of arbitrary subarrays, facilitating indexed pattern matching. The speed of interval access comes from the cache-friendly nature of RLZ decompression: after an initial predecessor query, all subarray values are obtained by a (usually small) number of cache-friendly copies from the reference sequence. Our index is also easy to construct.

There a numerous avenues for future work. Firstly, although we may never reach the impressively small size of the r-index, we believe the space usage of

the rlzsa can be significantly further reduced in practice by both simple representational techniques (e.g., bit packing position values, using Elias-Fano for the predecessor structure) and by adapting improved reference construction schemes that work well for RLZ when compressing text [13,19,20,16]. Secondly, is there a way to derive a hybrid of the rlzsa and r-index approaches that is smaller than the former and faster than the latter? Finally, it may be possible to derive space bounds for the rlzsa by combining the analysis of Gagie et al. [10], which relates the size of RLZ under random sampling to grammar compression of T, with the analysis of Gonzalez and Navarro [11], which relates grammar compression of the differentially encoded SA to the kth order empirical entropy of T.

References

1. Belazzougui, D., Cunial, F., Gagie, T., Prezza, N., Raffinot, M.: Composite repetition-aware data structures. In: Proc. 26th Annual Symposium on Combinatorial Pattern Matching (CPM). pp. 26–39 (2015)
2. Cáceres, M., Puglisi, S.J., Zhukova, B.: Fast indexes for gapped pattern matching. In: Proc. 46th International Conference on Current Trends in Theory and Practice of Informatics (SOFSEM). pp. 493–504. LNCS 12011, Springer (2020)
3. Deorowicz, S., Grabowski, S.: Robust relative compression of genomes with random access. Bioinformatics 27(21), 2979–2986 (2011)
4. Farrada, H.: Personal Communication
5. Farruggia, A., Gagie, T., Navarro, G., Puglisi, S.J., Sirén, J.: Relative suffix trees. Comput. J. 61(5), 773–788 (2018)
6. Ferragina, P., Manzini, G.: Opportunistic data structures with applications. In: 41st Annual Symposium on Foundations of Computer Science, FOCS 2000, 12-14 November 2000, Redondo Beach, California, USA. pp. 390–398. IEEE Computer Society (2000)
7. Ferragina, P., Manzini, G.: Indexing compressed text. J. ACM 52(4), 552–581 (2005)
8. Gagie, T., Navarro, G., Prezza, N.: Optimal-time text indexing in BWT-runs bounded space. In: Proc. SODA. pp. 1459–1477. ACM-SIAM (2018)
9. Gagie, T., Navarro, G., Prezza, N.: Fully functional suffix trees and optimal text searching in bwt-runs bounded space. J. ACM 67(1), 2:1–2:54 (2020)
10. Gagie, T., Puglisi, S.J., Valenzuela, D.: Analyzing relative lempel-ziv reference construction. In: String Processing and Information Retrieval - 23rd International Symposium, SPIRE 2016, Beppu, Japan, October 18-20, 2016, Proceedings. pp. 160–165. LNCS 9954 (2016)
11. González, R., Navarro, G.: Compressed text indexes with fast locate. In: Proc. 18th Annual Symposium on Combinatorial Pattern Matching (CPM). pp. 216–227. LNCS 4580 (2007)
12. González, R., Navarro, G., Ferrada, H.: Locally compressed suffix arrays. ACM Journal of Experimental Algorithmics 19(1), article 1 (2014)
13. Hoobin, C., Puglisi, S.J., Zobel, J.: Relative Lempel-Ziv factorization for efficient storage and retrieval of web collections. Proceedings of the VLDB Endowment 5(3), 265–273 (2011)

14. Kuruppu, S., Puglisi, S.J., Zobel, J.: Relative Lempel-Ziv compression of genomes for large-scale storage and retrieval. In: Proceedings of the 17th Symposium on String Processing and Information Retrieval (SPIRE). pp. 201–206. LNCS 6393 (2010)
15. Larsson, N.J., Moffat, A.: Offline dictionary-based compression. Proc. IEEE 88(11), 1722–1732 (2000)
16. Liao, K., Petri, M., Moffat, A., Wirth, A.: Effective construction of relative lempel-ziv dictionaries. In: Proc. 25th International Conference on the World Wide Web (WWW). pp. 807–816 (2016)
17. Mäkinen, V., Belazzougui, D., Cunial, F., Tomescu, A.I.: Genome-Scale Algorithm Design: Biological Sequence Analysis in the Era of High-Throughput Sequencing. Cambridge University Press (2015)
18. Manber, U., Myers, G.: Suffix arrays: a new method for on-line string searches. SIAM J. Computing 22(5), 935–948 (1993)
19. Tong, J., Wirth, A., Zobel, J.: Compact auxiliary dictionaries for incremental compression of large repositories. In: Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management, CIKM 2014, Shanghai, China, November 3-7, 2014. pp. 1629–1638. ACM (2014)
20. Tong, J., Wirth, A., Zobel, J.: Principled dictionary pruning for low-memory corpus compression. In: The 37th International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR '14, Gold Coast , QLD, Australia - July 06 - 11, 2014. pp. 283–292. ACM (2014)
21. Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Transactions on Information Theory 23(3), 337–343 (1977)