Smaller RLZ-Compressed Suffix Arrays

Puglisi, Simon J.

IEEE
2021


http://hdl.handle.net/10138/334122
https://doi.org/10.1109/DCC50243.2021.00029

unspecified
acceptedVersion

Downloaded from Helda, University of Helsinki institutional repository.
This is an electronic reprint of the original article.
This reprint may differ from the original in pagination and typographic detail.
Please cite the original version.
Smaller RLZ-Compressed Suffix Arrays*

Simon J. Puglisi and Bella Zhukova

Helsinki Institute for Information Technology (HIIT)
Department of Computer Science, University of Helsinki, Finland

Abstract

Recently it was shown (Puglisi and Zhukova, Proc. SPIRE, 2020) that the suffix array (SA) data structure can be effectively compressed with relative Lempel-Ziv (RLZ) dictionary compression in such a way that arbitrary subarrays can be rapidly decompressed, thus facilitating compressed indexing. In this paper we describe optimizations to RLZ-compressed SAs, including generation of more effective dictionaries and compact encodings of index components, both of which reduce index size without adversely affecting subarray access speeds relative to other compressed indexes. Our experimental analysis also elucidates the relationship between subarray size and per element access time.

1 Introduction

The suffix array [13], SA[0..n − 1], of a string T of length n is an array of integers containing a permutation of (0...n − 1), so that the suffixes of T starting at the consecutive positions indicated in SA are in lexicographical order: T[SA[i]..n] < T[SA[i + 1]..n]. The lexicographic ordering ensures that all the starting positions of the suffixes, starting with a given substring Q of T, form an interval SA[s..e].

The favored technique for determining s and e for a given pattern is the so-called backward search method, which finds the interval of the SA via 2|Q| rank queries on the Burrow-Wheeler transform (BWT) of T [4]. Backward search is the basis for compressed indexing, exemplified by the FM-index family [4], which has been widely adopted in practice, particularly in the field of genomics (see, e.g., [12]). The BWT can be stored compressed while still supporting rank queries, and so the challenge 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 [4, 6, 8].

Recently, Puglisi and Zhukova [15] showed that in practice relative Lempel-Ziv (RLZ) parsing is an effective way to compress the suffix array, and one that supports very fast decompression of intervals. Their starting point was the differentially encoded SA, denoted SA^d, as first introduced by Gonzalez and Navarro [8]. Puglisi and Zhukova [15] then construct a dictionary, R, (usually called the reference sequence [10]), by randomly sampling subarrays from SA^d, and parsed SA^d into phrases relative to R. Supporting random access is then a matter of storing one original SA value for each phrase (to reverse the differential encoding) and storing the phrase starting points in a predecessor data structure in order to determine the phrase overlapping the start of the interval. Decompressing occ consecutive values from SA can

*This work is supported by Academy of Finland grant 319454.
then be performed in time $O(\log \log n + occ + \ell^*)$, where $\ell^*$ is the length of the phrase overlapping the start of the interval to be decompressed. The index is very fast in practice: more than 100 times faster than the r-index [6] and the CDAWG [1], which were previously the fastest published methods for highly-repetitive collections. However, while the RLZ-compressed SA is much smaller than the $n \log n$ bits required by the plain SA, experiments in [15] show that depending on the dataset, it uses 5-15 times more space than the r-index (and always less than the CDAWG).

**Contribution.** This paper makes a number of contributions to the implementation and analysis of the rlzsa index.

- We demonstrate that by selecting segments of the SA for inclusion in the reference string (dictionary) based on the frequency of constituent substrings, references that significant outperform references generated by randomly sampling substrings can be obtained.

- We show that a compact representation of various index components leads to relevant space-time tradeoffs. We also show that further easy-to-exploit reductions in index size are possible on highly repetitive collections.

- The practical relevance of $O(\log \log n + occ + \ell^*)$ extraction time for RLZ-compressed SAs hinges on the relationship of $occ$ and $\ell^*$. We elucidate through experiments a smooth trade-off between interval size and per occurrence extraction time.

- We compare RLZ-compressed SAs to other state-of-art indexes for repetitive datasets, including those that attempt to exploit repetitions in SA$^d$, such as the LCSA [3, 9], showing rlzsa to be significantly faster, and usually smaller.

**Roadmap.** The remainder of this paper is organized as follows. Section 2 reviews the RLZ-compressed SA. Section 3 outlines an improved reference generation method. Section 4 explore various ways in which index size can be reduced. Section 5 and Section 6 report on experiments measuring the resource usage of our approach.

## 2 Relative Lempel-Ziv Compressed Suffix Array

In this section we review the RLZ-compressed suffix array as described in [15].

**RLZ parsing.** RLZ parsing is a variant of the classic LZ77 parsing [16]. It compresses a sequence $X$ relative to a second sequence $R$ (the reference) by encoding $X$ as substrings, or phrases, that occur in $R$. The RLZ-compressed SA, denoted rlzsa, is built on a modified form of RLZ parsing that is critical to support efficient access to subarrays of the SA and which we now describe.

In [15], SA is encoded by parsing SA$^d$ into $z$ phrases. Recall, SA$^d[0] = SA[0]$, and then for $i > 0$, SA$^d[i] = SA[i] - SA[i-1]$. Phrases are represented as integer pairs and
either embody literal values from the original SA (literal phrases), or point to sub-
strings 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 \( (i, SA[i]) \). The first phrase is always the 
literal phrase \( (0, SA[0]) \). Parsing begins at position 1 in \( SA^d \) and proceeds according 
to the following rule. If the parsing is up to a position \( i \) in \( SA^d \), then the next phrase 
is either:

- a literal phrase \( (i, SA[i]) \), if the previous phrase was not a literal phrase or \( SA^d[i] \) 
does not occur in \( R \); or

- \( (i, p_i) \), such that \( R[p_i .. p_i + \ell_i], \ell_i \geq 1 \) is longest prefix of \( SA^d[i, n] \) occurring 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 \( \ell_i \) and \( (i - 1, x) \) be the 
preceding literal phrase in the parsing. Then \( SA[i] = SA^d[i] + x = R[p_i] + x, SA[i+1] = 
SA^d[i+1] + SA[i] = R[p_i+1] + SA[i], \ldots, SA[i+\ell_i-1] = R[p_i+\ell_i-1] + SA[i+\ell_i-2] \).

**Data Structure.** The parsing is stored in two arrays, \( S \) and \( P \), both of length \( z \). 
\( S \) contains the starting position in \( SA^d \) of each phrase in ascending order. Elements 
of \( S \) are kept in a predecessor data structure. \( P \) contains either literal SA values or 
positions in \( R \) as output by the parsing algorithm (the second components of each 
pair). The type of each phrase (literal or repeat) is marked using a separate bit string 
of \( z \) bits. The length of the \( i \)th phrase can be determined as \( S[i+1] - S[i] \). An 
illustration of the main components of the data structure is given in Figure 1.

**Decoding a Subarray.** Decoding an arbitrary interval \( SA[s, e] \) is performed as 
follows. An output buffer \( B \) of size \( e - s + 1 \) will contain the decoded elements. At a 
high level, the phrases covering \( SA[s, e] \) are decoded and copied to \( B \) (some parts of 
the first and last phrase may not) until \( B \) is full, at which point we are done. To this 
end, begin finding the index in \( S \) of the predecessor of \( s \). Let \( x \) denote this index. 
If \( P[x] \) is a literal phrase, copy its value to the output buffer. Otherwise (\( P[x] \) is 
non-literal) \( P[x-1] \) is by definition literal, so set \( p = P[x-1] \). The length of the 
phrase is \( \ell = S[x+1] - S[x] \). If \( S[x] = s \), to decode phrase \( x \) we access \( R[P[x]] \), copy 
\( (p + R[P[x]]) \) to the output buffer, and then set \( p = (p + R[P[x]]) \), continuing then to 
copy \( (p + R[P[x] + 1]) \) to \( B \), and so on until either the whole phrase has been decoded, 
or the output buffer is full. And if \( S[x] < s \), first \( (s - S[x]) \) symbols of phrase \( x \) that 
are before position \( s \) must be decoded and discarded (as described). After decoding 
phrase \( x \), if the output buffer is not full, then phrase \( x+1 \) is decoded, and so on, 
until all \( e - s + 1 \) values have been decoded.
Figure 1: An illustration of the main components of the rlzsa index for an example string. In the figure, \( SA^d[0] = SA[0] \) and \( SA^d[i] = SA[i] - SA[i-1] + n \), where \( n \) is added to ensure that all elements of \( SA^d \) are positive, easing presentation. The reference \( R \) consists of substrings of \( SA^d \).

3 Reference Generation

In [15] the reference string was generated by random sampling of substrings from \( SA^d \). While this approach resulted in indexes that achieved both significant compression of the SA and very fast extraction of SA subarrays, the question arises whether more judicious selection of substrings would improve compression while maintaining interval extraction times. In the application of RLZ parsing to compression of text collections, Liao, Petri, Moffat, and Wirth [11] found that superior compression performance could be achieved by selecting substrings for inclusion in the reference according to the abundance of smaller \( k \)-mers (substrings of length \( k \)) contained therein. We investigated the efficacy of a simplified version of their approach to generating a reference for \( SA^d \).

The approach is as follows. We divide \( SA^d \) into \( n/s \) substrings of equal length \( s \) called segments (the last segment may have length < \( s \), and if so is not considered in the process described below). The frequency of each distinct \( k \)-mer in \( SA^d \) is then computed. Each segment is assigned an initial score according to the \( k \)-mers it contains. In particular, let \( f(x) \), for \( k \)-mer \( x \) be the frequency of \( x \) in \( SA^d \). Let \( x \in X_i \) denote that \( k \)-mer \( x \) has at least one occurrence in segment \( X_i \). Then the initial score for segment \( X_i \), \( i \in [0..n/s+1] \) is the \( \ell_p \) norm of the vector of its constituent \( k \)-mers, calculated as:

\[
\text{score}(X_i) = \left( \sum_{x \in X_i} f(x)^p \right)^{1/p}.
\]

The highest scoring \( X_i \) is then selected for inclusion in the reference, and frequencies \( f(.) \) are then reduced for every \( k \)-mer \( x \in X_i \), in particular \( f(x) \) is reduced by the frequency of \( k \)-mer of \( x \) in \( X_i \). Throughout we use \( p = 1/2 \). This process of segment selection and subsequent score adjustment is repeated until the sum of the lengths of the selected segments has reached the target reference length (an input parameter).

Apart from the target reference length, there are three parameters to this process: \( s \), the segment size; \( k \) the \( k \)-mer length; and \( p \) which affects the way in which \( k \)-mer frequencies affect segment scores. Figure 2 (left) provides an exploration of
parameters $s$ and $k$ on a dataset of Wikipedia pages (details of all our test datasets are provided in Section 6). We make two observations. Firstly, reference construction according to the process described above significantly outperforms that achieved by a randomly generated reference, as used in [15] — however, as we shall see in Section 6, on one dataset our constructed references fail to outperform randomly generated ones (but even then do get exceedingly close). Secondly, there is significant variation in performance across $s$ and $k$.

We also experimented with a variant where, instead of dividing SA$^d$ into $n/s$ equal-length segments, segments are allowed to start at any position, meaning $n - s - 1$ segments are in contention for inclusion in the reference. This generally led to marginally improved compression (around 5%), but reference generation time made significantly longer (by roughly a factor of $s$).

As a final remark in this section, we found that not every position in our constructed references was actually covered by any phrase, and removing the symbols at those positions led to around a 5% reduction in reference length on all datasets.

### 4 Representation of Index Components

The implementation of rlzsa described in [15] is completely byte aligned, which is one reason for its speed. In particular, reference $R$ and phrases $P$ are stored as arrays of 32-bit integers of length $r$ and $z$ respectively. The most-significant bit of each integer in $P$ is used to mark with the entry is an absolute SA value or a position in $R$. The maximum phrase length is limited to $2^{16}$, allowing use of the following two-layered predecessor data structure for the phrase starting positions. Every $b$th phrase starting position is sampled and stored in an array of 32-bit integers. A separate array contains a differential encoding of all $z$ starting positions. The aforementioned restriction on phrase lengths means the array of differentially encoded starting positions can be
stored as an array of 16-bit integers. Predecessor search for a position $x$ then begins by first binary searching in the array of samples to find the predecessor sample, say at index $i$ of that array. The differentially encoded array is then accessed starting at index $ib$ and then scanned, summing values until the cumulative sum is greater than $x$, at which point the predecessor is known.

In squeezing of our data structure, the most obvious starting point is to represent the reference sequence $R$ in $|R|\lceil \log n \rceil$ bits. Also, although it is fast, the predecessor data structure described above wastes space in general. Our first optimization then is to replace it with an Elias-Fano predecessor data structure, as implemented in the SDSL library [7]. Elias-Fano supports predecessor queries in $O(\log(\frac{n}{z}))$ time (using $2z+z\lceil \log(\frac{n}{z}) \rceil$ bits of space), making interval extraction time $O(\log(\frac{n}{z}) + occ + \ell^*)$.

**Representing phrase sources.** Another simple space saving is yielded by storing phrase source positions using $\lceil \log(|R|) \rceil$ bits, instead of the 32 (or 64) bits consumed when using a plain integer type to store each of those components.

\begin{center}
\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure3.png}
\caption{Illustration of skewness in phrase $p$ values (left) and $\ell$ values (right). Points are arranged in order of decreasing frequency rank (i.e., leftmost point is the most common value). On some datasets (such as boost this skewness may allow for index size to be further reduced via variable-length encoding). This, however, is not always the case, with the distribution of values being much flatter in the other three datasets.}
\end{figure}
\end{center}

Further space savings are possible, however. In particular, we found that the number of distinct phrase source positions, $r'$, was always significantly lower than $|R|$ on all datasets. Simply by storing the distinct source positions in a lookup table of (total) size $r'\lceil \log |R| \rceil$ and then using $\lceil \log(r') \rceil$ bits for each source position allows from 5 bits (DNA) to 9 bits (boost) to be saved per phrase (in our experiments, $\lceil \log |R| \rceil$ was in the range 17 to 24). On extremely repetitive datasets it seems possible to do better still. As Figure 3 illustrates, the most frequent source position in the boost dataset occurs more than $10^5$ times (corresponding to more than 25% of the phrases), suggesting significant space reductions should be achievable via variable-length encodings that support random access, such as [2, 5]. The curve is much flatter for other datasets.

While this analysis is very encouraging, we recoiled from implementing these mea-
sures, and plan to include them in the full version of this article.

5 Affect of Interval Size on Extraction Time

As mentioned earlier, decompressing $occ$ consecutive values from SA can then be performed in time $O(\log \log n + occ + \ell^*)$, where $\ell^*$ is the length of the phrase overlapping the start of the interval to be decompressed. This suggests that while for long intervals (high $occ$) the cost of $\ell^*$ will be easily amortized, there may be a significant overhead for smaller intervals ($occ < \ell^*$).

To elucidate the affect of interval size on per occurrence retrieval time, we measured locate time per occurrence of all indexes on 1000 patterns of length 8 randomly extracted from two datasets (see Section 6 for details of datasets). Results are shown in Figure 4 for two data sets. The figure shows a hefty overhead for small intervals due to a combination of predecessor query and redundant phrase decoding. Both these costs are gradually amortized over the occurrences returned as interval size increases.

![Figure 4: Scatter plot showing time per occurrence. Each point in the plot corresponds to a query. The horizontal axis shows interval size. The vertical axis shows time for interval extraction divided by interval size. Note the doubly logarithmic scales.](image)

6 Performance Comparison

In this section we report on the practical performance of our data structure, both in plain and compact form, in comparison to other state-of-the art solutions.

**Test Machine.** All our experiments were conducted on 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 18.04.5 LTS) running kernel 5.4.0-58-generic. Programs were compiled using gcc version 7.5.0. All given runtimes were recorded with the C++11 `high_resolution_clock` time measurement facility.
Datasets and Patterns. Mirroring the experiments in [6], we used the following four repetitive datasets in our experiments:

- **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;

**DNA**, **boost**, and **einstein**, are all $\sim$600MB in size, and **world** is $\sim$47MB. We measured memory usage and locate times per occurrence of all indexes on 1000 patterns of length 8 extracted from each dataset. The average number of occurrences per pattern was 89453 (**boost**), 607750 (**DNA**), 31788 (**einstein**), 29781 (**world**).

Indexes Measured. For comparison we used r-index implementation of Gagie at. al. [6], LCSA implementation of Cáceres and Navarro [3], and rlzsa-rand which denotes the best result in [15]. These indexes were selected because r-index is the smallest CSA, rlzsa-rand is the fastest CSA according to experiments [15], and LCSA also uses a differentially encoded SA combined with Re-Pair, instead of RLZ.

We provide results for two variants of the rlzsa approach, which are labelled rlzsa and rlzsa-sdsl in the plots. The rlzsa variant uses byte-aligned arrays and two-tiered predecessor data structure. The rlzsa-sdsl variant implements Section 4, i.e., packed integer vectors for the reference $R$ and phrases $P$ and an Elias-Fano data structure over phrases’ starting positions $S$, used both for finding the phrase covering the start of a query interval and lengths of the phrases. Both variants use the same data structure as the r-index to find the interval to be decoded (i.e. backward search) and the space of this structure and time for the backward search process is always included in the measurements reported.

In the plots the **DNA** and **einstein** used reference constructed with segments selection, **boost** and **world** — with floating segments selection, segment size $s$ and $k$-mers length $k$ for both rlzsa and rlzsa-sdsl were the same: $s = 1024$ for **world** and 4096 for the other datasets, $k = 1024$ for **boost**, 16 for **DNA** and **world**, and 8 for **einstein**. Reference size $|R|$ for rlzsa was 110592 (**boost**), 16793600 (**DNA**), 3063808 (**einstein**), and 1831936 (**world**), for rlzsa-sdsl: 57344 (**boost**), 12185600 (**DNA**), 2867200 (**einstein**), and 1038336 (**world**).

Results. The results of our experiments appear in Figure 5. Approaches based on RLZ are significantly faster than both r-index and LCSA. LCSA is only marginally faster than r-index and requires 1.5-24× more space on these datasets, whereas rlzsa is from almost 116 (on **world**) up to 178 (on **DNA**) times faster than r-index. The rlzsa-sdsl variant is, as expected, slower than the byte-aligned rlzsa index, showing a

---

1Available here: [https://github.com/nicolaprezza/r-index/tree/master/datasets](https://github.com/nicolaprezza/r-index/tree/master/datasets).
speedup of 30-43 times compared to r-index. On the other hand, rlzsa requires 3-12\times more space than r-index, whereas for rlzsa-sdsl the increase is just 2-7\times.

For the DNA smarter choice of the reference showed not only significant decrease in space (more than 26%), but also almost 19% less time. For the other datasets time stayed the same with the space usage decreasing on 20-33%, except on boost where rlzsa even with reference constructed using floating segments approach did not beat the rlzsa-rand space usage.

7 Concluding Remarks

We have shown that RLZ-compressed suffix arrays can be made even more effective by judicious construction of reference sequences (as opposed to construction by random substring sampling) and by the use of succinct encodings for various index components. The new indexes extend the Pareto frontier for compressed text indexing.

We believe this frontier can be further extended by combining the rlzsa and r-index, for example, by using the r-index to obtain (at query time) the absolute SA values needed for rlzsa interval extraction.
Another important issue is that of parameter selection. We have not yet attempted to optimise our index construction pipeline, but it is already reasonably fast (8 mins and 37 secs on the 2GB einstein data set, for example). This allows the user some freedom to try different settings of $k$ and $s$ in order to optimise index size, but an automatic method that first analyses the input SA would be preferable.

We also remark that RLZ parsing can also be expected to be effective on the so-called document array, $D$, which is central to document retrieval queries (see, e.g., Navarro’s extensive survey [14]).

Acknowledgements. We are very grateful to the anonymous reviewers whose detailed comments materially improved our manuscript, and to Ariel Cáceres for providing an implementation of the LCSA optimized for extracting intervals.

References