Practical Indexing of Repetitive Collections using Relative Lempel-Ziv

Gonzalo Navarro*† and Víctor Sepúlveda*

*CeBiB — Center for Biotechnology and Bioengineering, Chile
† Department of Computer Science, University of Chile, Chile
gnavarro@dcc.uchile.cl, palabragris@gmail.com

Abstract

We introduce a simple and implementable compressed index for highly repetitive sequence collections based on Relative Lempel-Ziv (RLZ). On a collection of total size n compressed into z phrases from a reference string $R[1..r]$ over alphabet $[1..\sigma]$ and with hth order empirical entropy $H_h(R)$, our index uses $rH_h(R) + o(r \log \sigma) + O(r + z \log n)$ bits, and finds the *occ* occurrences of a pattern $P[1..m]$ in time $O((m + occ) \log n)$. This is competitive with the only existing index based on RLZ, yet it is much simpler and easier to implement. On a 1GB collection of 80 yeast genomes, a variant of our index achieves the least space among competing structures (slightly over 0.1 bits per base) while outperforming or matching them in time (1–10 microseconds per occurrence found). Our largest variant (below 0.3 bits per base) offers the best search time (1–3 microseconds per occurrence) among all structures using space below 1 bit per base.

Introduction

Many types of text collections are growing at a pace that outperforms Moore's Law [1]. A key tool to handle this growth is to exploit the high degree of repetitiveness featured by several of the fastest growing collections: genomes of the same species, versioned repositories of documents or software, etc. Lempel-Ziv (LZ) compression [2] is used as the gold standard to decrease the storage requirements of these collections. Reductions of two orders of magnitude have been reported on repositories like the 1000-genomes project [3] (`www.internationalgenome.org`) or Wikipedia (`en.wikipedia.org/wiki/Wikipedia:Size_of_Wikipedia`). To obtain such reductions, one must not compress each document independently, but the collection as a whole. Since accessing individual documents then requires decompression of a significant part of the collection, however, LZ compression can be regarded as an archival-only solution.

Relative Lempel-Ziv (RLZ) [4] is a more recent technique to exploit repetitiveness. It chooses a reference sequence and represents the others as a concatenation of strings appearing in the reference. Those sequences are then encoded with a sequence of pointers to the reference. RLZ obtains compression ratios close to those of LZ, but it is more convenient because it allows direct access to the compressed data.

Other than retrieving the compressed data, the next goal is to efficiently search it. We consider the most fundamental search problem: find the occurrences of a given

Funded with Basal Funds FB0001, Conicyt, Chile. First author also funded by Fondecyt grant 1-170048, Chile.

pattern in the collection. The extremely efficient access provided by RLZ promises to yield much faster compressed indexes than those that build on LZ. Somewhat surprisingly, while there are practical compressed indexes for repetitive collections that build on LZ compression [5], and also on grammar compression [6] and on the Burrows-Wheeler transform [7, 8], the only compressed index based on RLZ [9] is complicated and has not been implemented.

In this paper we present a compressed index based on RLZ that offers theoretical guarantees comparable with those of the existing index [9], but it is much simpler to implement. On a collection $\mathcal{S}$ of total size n compressed into z phrases from a reference string $R[1..r] \in \mathcal{S}$ over alphabet $[1..\sigma]$ and with hth order empirical entropy $H_h(R)$ [10], our index uses $rH_h(R)+o(r\log\sigma)+O(r+z\log n)$ bits, for any $h \leq \alpha \log_\sigma r$ and constant $0 < \alpha < 1$. It finds the *occ* occurrences of a pattern $P[1..m]$ in time $O((m + occ)\log n)$. Three implementations of our index are shown to offer the least space among competing indexes, while matching or outperforming them in search time, or the least times, only outperformed by structures using much more space.

Basic Concepts

Definitions

A *sequence* or *string* $S[1..|S|] = S[1] \cdot S[2] \cdots S[|S|]$ is an array of symbols over an alphabet $[1..\sigma]$, and its substrings are denoted $S[i..j] = S[i] \cdots S[j]$. The concatenation of two strings S and S' is denoted $S \cdot S'$. A *collection* is a set $\mathcal{S} = \{S_1, \ldots, S_t\}$ of sequences, each terminated with a special symbol \$. We identify the set $\mathcal{S}$ with the string $\mathcal{S} = S_1 \cdots S_t$, whose length is $n = |\mathcal{S}| = |S_1| + \cdots + |S_t|$, the *size* of the collection. We say that $\mathcal{S}$ is *repetitive* if any sequence S_k can be obtained by concatenating a few substrings (called *phrases*) from other sequences in $\mathcal{S}$. Such a coverage with phrases is called a *parse* of S_k.

The *pattern matching (or the search) problem* is, given a string $P[1..m]$ with no \$s, find all the *occ* positions in $\mathcal{S}$ where P occurs. A *self-index* is a data structure that (i) obtains any substring $S_k[i..j]$ from any sequence, and (ii) solves the pattern matching problem efficiently (at least faster than a sequential scan of $\mathcal{S}$).

Our logarithms are to the base 2 by default. The *hth order empirical entropy* of a string R [10], denoted $H_h(R) \leq H_{h-1}(R) \leq \log\sigma$, is a lower bound of the bits-per-symbol achievable by any hth order semistatic statistical compressor on R.

A *Karp-Rabin fingerprint* [11] is a hash for strings computed as $KR(s_1 \ldots s_n) = (\sum_{i=1}^{n} s_i b^{n-i}) \bmod p$, for some b and p. Karp-Rabin fingerprints allow computing $KR(S_1 \cdot S_2)$ in constant time given $KR(S_1)$ and $KR(S_2)$, as well as computing $KR(S_1)$ or $KR(S_2)$ from $KR(S_1 \cdot S_2)$ and $KR(S_2)$ or $KR(S_1)$, respectively.

Some Compressed Data Structures

A bitvector $B[1..n]$ can be represented using $n + o(n)$ bits so that it answers queries $rank_b(B, i)$ (number of bits equal to b in $B[1..i]$) and $select_b(B, j)$ (position of the jth bit equal to b in B) in time $O(1)$ [12]. If B has $m \ll n$ 1s, a compressed representation [13] uses $m\log(n/m) + O(m)$ bits and solves $rank_b$ in time $O(\log(n/m))$, $select_1$ in time $O(1)$, and $select_0$ in time $O(\log m)$.

A Range Maximum Query (RMQ) structure built on array $A[1..z]$ is able to find, given a range $A[i..j]$, the leftmost position of a minimum in $A[i..j]$. Such a structure can use $2z + o(z)$ bits and answer queries in $O(1)$ time without accessing A [14].

A permutation π in $[z]$ can be stored in $z \log z + O(z)$ bits so that we can compute any $\pi(i)$ in $O(1)$ time and any inverse $\pi^{-1}(j)$ in time $O(\log z)$ [15].

A wavelet tree structure [16, 17] stores z points in $[1..z] \times [1..z]$ in $z \log z + o(z \log z)$ bits and returns all the *occ* points in any rectangular range in time $O((occ+1) \log z)$.

An FM-index [18, 19] is built on a string $R[1..r]$ over alphabet $[1..\sigma]$ and uses $rH_h(R) + o(r \log \sigma) + O(r)$ bits, for any $h \leq \alpha \log_\sigma n$ and constant $0 < \alpha < 1$. It can find the *occ* occurrences of any pattern $P[1..m]$ in R in time $O(m + occ \log r)$ and retrieve any substring $R[i..j]$ in time $O(\log r + j - i)$.

Relative Lempel-Ziv Compression

Relative Lempel-Ziv (RLZ) [4] compression builds a *reference* string $R[1..r]$ from $\mathcal{S}$ and represents it in plain form (in our case, we assume R is simply some string $S_k \in \mathcal{S}$). All the sequences of $\mathcal{S}$ are then represented as the concatenation of *phrases* (substrings of R). More formally, if $S_k = R[p_1..p_1 + \ell_1 - 1] \cdots R[p_s..p_s + \ell_s - 1]$, then we represent S_k as the sequence of pairs $(p_1, \ell_1), \ldots, (p_s, \ell_s)$.[1]

The minimum number s_k of phrases to cover S_k with substrings of R can be obtained with an FM-index on R, which finds, at each point, the longest prefix of the remaining text of S_k that appears in R, say in $R[p_i..p_i + \ell_i - 1]$, in time $O(\ell_i + \log r)$. Let $z = s_1 + \cdots + s_t$ be the total number of phrases added over the sequences S_k. Then the whole RLZ compression of $\mathcal{S}$ is carried out in time $O(n + z \log r)$, and the output of the RLZ compressor uses $rH_h(R) + o(r \log \sigma) + O(r + z \log r)$ bits.

With $O(z \log(n/z))$ further bits we can store a compressed bitvector $B[1..n]$ that marks where phrases start in $\mathcal{S}$, so that any substring $S_k[i..j]$ is retrieved in time $O(\log(n/z) + j - i)$ by determining the phrase that contains i and fetching the subsequent characters from R, switching to the next phrase when the current one ends.

Self-Indexes

There are various self-indexes appropriate for repetitive collections. For lack of space, we mention only those that have been implemented and proved to be most successful in practice, referring the reader to a recent work [8, extended version] for the rest. The LZ Index [5] is a self-index based on the LZ parsing [2]: If $\mathcal{S}$ is parsed into lz phrases with parsing depth hz, then the LZ Index uses $O(lz)$ space and searches in time $O(m^2\, hz + (m + occ) \log n)$; we will compare with its tuned version [20]. The r-index [8] builds on the Burrows-Wheeler Transform (BWT) [21] of $\mathcal{S}$: if the BWT has b equal-letter runs, then the r-index uses $O(b)$ space and searches in time $O((m + occ) \log \log_w n)$ on a w-bit RAM machine. Finally, the Hybrid Index [22, 23] is a practical LZ-based index that does not provide worst-case time guarantees but performs very well.

[1] An exception occurs if the next symbol of S_k does not appear in R. To avoid this case, we (virtually) append the symbols $[1..\sigma]$ at the end of R, so this case is handled as a phrase of length 1.

As said, there exists only one self-index building on RLZ [9]. It uses $O(r+z)$ space (more precisely, $(2+\frac{1}{\epsilon})rH_h(R) + o(r\log\sigma) + O(r + z\log r)$ bits) and searches in time $O(m(\log^\epsilon r + \frac{\log\sigma}{\log\log r}) + occ(\log^\epsilon_\sigma r + \log_r z + \log\sigma))$. With $O(z\log r\log\log r)$ further bits, it searches in time $O(m(\log\log r + \frac{\log\sigma}{\log\log r}) + occ(\log^\epsilon_\sigma r + \log_r z + \log\sigma))$, for any constant $\epsilon > 0$. These times are slightly lower than ours, but they use more space associated with the reference R. This index has not been implemented.

Our Index

We follow the basic design of the previous RLZ-based index [9] (which is, in turn, analogous to that of most LZ-based indexes). We divide the occurrences into three sets: within the reference R, inside a phrase of the RLZ parse, and crossing RLZ phrases. We discuss how to handle each kind of occurrence.

Occurrences within the Reference

We search for P using the FM-index of R. Since we assume $R \in \mathcal{S}$, all those occurrences are reported. The total time of this step is $O(m + occ\log r)$.

Occurrences within Phrases

For each occurrence $R[i..i+m-1]$ reported in the previous step, we find all the phrases $S_k[p_u..p_u+\ell_u-1] = R[t_u..t_u+\ell_u-1]$ in the RLZ parse whose source covers $R[i..i+m-1]$, that is, $[i..i+m-1] \subseteq [t_u..t_u+\ell_u-1]$. Then we report an occurrence starting at $S_k[p_u+i-t_u]$.

The structure to find the relevant sources is as follows (cf. [5]). We sort all the sources of phrases by their starting point and store a plain bitvector $S[1..z+r]$ where we traverse R left to right and for each position $R[i]$ we append a 1 for each source starting at $t_u = i$, and then a 0. We also store a permutation π in $[z]$ that maps target order to source order in phrases. We also build an RMQ data structure on the (virtual) array $E[1..z]$ of the ending points $t_u + \ell_u - 1$ of the sources. All these structures add up to $O(r + z\log r)$ bits.

We know that the sources that start at or before $R[i]$ are those referred to in $E[1..s]$, with $s = select_0(S,i) - i$. If the maximum $E[t]$ in $E[1..s]$ is $E[t] < i+m-1$, then no source covers $R[i..i+m-1]$. Otherwise, the source corresponding to $E[t]$ covers the occurrence and we must report its corresponding target, and recurse in $E[1..t-1]$ and $E[t+1..s]$. This technique [24] ensures that we perform $O(1)$ steps per occurrence reported. The RMQs do not access E, but we need to determine if $E[t] < i+m-1$, that is, if $t_u + \ell_u < i+m$, where $E[t]$ represents the target $[t_u, t_u+\ell_u-1]$. We find out $t_u = select_1(S,t) - t + 1$, $p_u = select_1(B,\pi(t))$, and $\ell_u = select_1(B,\pi(t)+1) - p_u + 1$, all in $O(1)$ time.

Therefore, from every occurrence found within R we find each of its occurrences copied inside RLZ phrases in $O(1)$ time. All the occurrences within phrases are then found in time $O(occ)$.

The structures B, S, and π can replace the sequence of pointers used in our basic RLZ description. To extract a substring $S_k[i..j]$ we must now use B, π^{-1}, S, and the FM-index on R, which gives worst-case time $O((j-i+1)\log n)$.

Occurrences Crossing Phrases

To report the occurrences of P that cross phrases [5], we take every phrase beginning, $S_k[p_u]$, and add the reverse of the prefix $S_k[1..p_u-1]$, $S_k[1..p_u-1]^{rev}$, to a set $\mathcal{X}$, and the suffix $S_k[p_u+1..|S_k|]$ to a set $\mathcal{Y}$. Then, every occurrence of P corresponds to a prefix $P[1..i]$ that matches a suffix of some $S_k[1..p_u-1]$ and a suffix $P[i+1..m]$ that matches a prefix of the corresponding $S_k[p_u+1..|S_k|]$.

By lexicographically sorting $\mathcal{X}$ and $\mathcal{Y}$ to form $X_1,\ldots,X_z$ and $Y_1,\ldots,Y_z$, the strings of interest form a range in each set. We store a wavelet tree on the points (x,y) such that X_x corresponds to Y_y. We can then search for $P[1..i]^{rev}$ in $\mathcal{X}$ and for $P[i+1..m]$ in $\mathcal{Y}$, obtaining ranges $[x_1..x_2]$ and $[y_1..y_2]$. Therefore, all the wavelet tree points within $[x_1..x_2]\times[y_1..y_2]$ are occurrences. We store an array $T[1..z]$ such that Y_y is the beginning of phrase number $T[y]$. Thus, if the wavelet tree reports point (x,y), we can report an occurrence at position $select_1(B,T[y])-i+1$. Array T is also useful to access any desired X_x or Y_y when searching for a prefix/suffix of P.

All these structures use $O(z\log z)$ bits and find all the *occ* occurrences crossing phrases in time $O((m+occ)\log z)$, once the ranges $[x_1,x_2]$ and $[y_1,y_2]$ for the m partitions of P are given.

Previous indexes required $\Omega(m^2)$ time to find all the ranges [5, 25]. We use a more recent result [8, and references therein] that reduces this time to $O(m\log n)$.

Lemma 1 ([8]) *Let $\mathcal{X}$ be a set of strings and assume we have some data structure supporting extraction of any length-ℓ substring of strings in $\mathcal{X}$ in time $f_e(\ell)$ and computation of the Karp-Rabin fingerprint of any substring of strings in $\mathcal{X}$ in time f_h. We can then build a data structure of $O(|\mathcal{X}|)$ words such that, later, we can solve the following problem in $O(m\log(\sigma)/w+t(f_h+\log m)+f_e(m))$ time: given a pattern $P[1..m]$ and $t>0$ suffixes $Q_1,\ldots,Q_t$ of P, discover the ranges of strings in (the lexicographically-sorted) $\mathcal{X}$ prefixed by $Q_1,\ldots,Q_t$.*

With the lemma, we find the $t=m-1$ ranges of $\mathcal{X}$ prefixed by $Q_1,\ldots,Q_t = P[1..1]^{rev},\ldots,P[1..m-1]^{rev}$. We use the lemma analogously on $\mathcal{Y}$, to find the ranges of $Q_1,\ldots,Q_t = P[2..m],\ldots,P[m..m]$. The data structure then uses $O(z\log n)$ bits. Since we support extraction of any substring of length ℓ in time $O(\ell\log n)$, we find the $O(m)$ ranges on $\mathcal{X}$ and $\mathcal{Y}$ in time $O(m\log n+m\,f_h)$. We now show how to compute fingerprints in time $O(\log n)$ as well.

We use fingerprints of $O(\log n)$ bits. Let $\mathcal{S}'$ be the string $\mathcal{S}$ without R, and let $p_1,\ldots,p_z$ be the starting points of the RLZ phrases. Then we store an array $K_S[1..z]$ with the Karp-Rabin fingerprints of prefixes of $\mathcal{S}'$: $K_S[s]$ is the fingerprint of $\mathcal{S}'[1..p_{s+1}-1]$. Then we can compute the fingerprint of a sequence of phrases, from the sth to the tth, by operating $K_S[s-1]$ and $K_S[t]$ in constant time.

In addition, we store an array $K_R[1..r/\log n]$, with the fingerprints of the prefixes of R whose length is a multiple of $\log n$. We can therefore compute the fingerprint of any substring of R in time $O(\log n)$, by operating two cells of K_R and adding $O(\log n)$ individual symbols to the signatures.

Arrays K_S and K_R use $O(z\log n+r)$ bits, and allow computing any signature on $\mathcal{S}$ in time $f_h=O(\log n)$: We identify with bitvector B the phrases involved. The

signature for the whole phrases contained in the substring is computed with K_S, whereas the signature of the remaining phrase prefix and suffix is computed with K_R by going to the source of the phrases in R, found in $O(\log z)$ time with π^{-1} and S.

Although we use fingerprints, the resulting ranges are deterministically correct. Karp-Rabin parameters are found in $O(n)$ time so that no two substrings of $\mathcal{S}$ have the same signature with high probability. Otherwise, a Las Vegas procedure using $O(n \log n)$ expected time ensures that no collisions arise [26].

Implementation

We use several structures from *sdsl* library [27] (`github.com/simongog/sdsl-lite`) to implement the different components of our indexes. The sparse bitvector $B[1..n]$ that marks the beginning of each phrase in the collection is stored using structure *sd_vector*, which is optimized for very sparse bitvectors [28, 13]. For the FM-index of the reference R, we use the the *csa_wt* structure, which stores the BWT [21] as a wavelet tree [16] with a sample density of 16 for the suffix array and 32 for its inverse.

The bitvector $S[1..z+r]$ used to find the secondary occurrences is stored in plain form (*bit_vector*) since it is short. To speed up searches, we represent both permutations π and π^{-1} explicitly. For the RMQs we use structure *rmq_succinct_sct* [14].

Also to speed up searches, we explicitly store the lexicographically sorted arrays $\mathcal{X}$ and $\mathcal{Y}$ (to access the text in combination with B), instead of array T. The two-dimensional search structure is implemented with a wavelet tree variant (structure *wt_int* supported by a plain *bit_vector*).

Using these structures as a base, we implemented three versions of our index.

RLZ Basic Index. The first version does not use Lemma 1, resorting instead to a quadratic-time procedure that binary searches for all the prefixes and suffixes of P. The characters of $\mathcal{X}$ and $\mathcal{Y}$ needed for the string comparisons are extracted from the FM-index of R. Since each substring access in the FM-index has an additive overhead related to the sampling of the inverse suffix array, we do not extract each symbol separately, but rather extract the whole substring we may need for the comparison (up to m symbols). The substring read is nevertheless limited by the length of the phrases: reading a string of $\mathcal{X}$ or $\mathcal{Y}$ may require accessing various consecutive phrases of some S_k, each of which will incur in the described additive penalty when extracted from R. This is the slowest version but it uses the least space.

RLZ Ref Index. This index adds to the previous one an explicit and plain representation of R, using $r \log \sigma$ further bits (and it removes the sampling of the inverse suffix array, as it is not used). In exchange for this extra space, the binary search for prefixes and suffixes of P is considerably sped up, since we can directly access the text of any arbitrary phrase $S_k[p_u..p_u + \ell_u - 1] = R[t_u..t_u + \ell_u - 1]$ by using B, π^{-1} and S, as described at the end of the section on occurrences within phrases. In this case we extract the symbols one by one, as we compare them with the appropriate symbols of P.

RLZ Hash Index. The third version of our index is the fastest and most space-consuming. In addition to the explicit reference R, it includes a simplified implemen-

tation of the structure of Lemma 1. Essentially, we implement the $m-1$ searches in $\mathcal{X}$ and $\mathcal{Y}$ on an $O(z)$-word Patricia Tree [29] that is enhanced with the Karp-Rabin fingerprints of the edge labels, in order to reduce the chance of false positives (which in our scenario are relatively expensive to verify).

We use one of these structures for the array $\mathcal{X}$ of the reverse phrases and another for the array $\mathcal{Y}$ of the suffixes. On each node of this tree we store the position of the rightmost leaf descendant in the arrays $\mathcal{X}$ or $\mathcal{Y}$ ($\lceil \lg z \rceil$ bits); the leftmost descendant is inferred from the traversal. We also store the first letter ($\lceil \lg \sigma \rceil$), the length ($\lceil \lg \ell_{max} \rceil$ given the maximum length in bits ℓ_{max}), and the Karp-Rabin fingerprint of the text associated with each child node (24 bits to have a low probability of collisions). The pointers to other nodes add up to $2\lceil \lg(4z) \rceil$ bits per node because there are $4z$ nodes.

Before all the prefix and suffix searches, we compute the Karp-Rabin fingerprint of each prefix of P, in a single pass of time $O(m)$. We can then combine two elements of this array to find the fingerprint of any substring of P in constant time. Like any Patricia tree, we must check one text occurrence after each search, but the chance of error is extremely small (much smaller than on a standard Patricia tree search) because the Karp-Rabin hashes of the pattern substrings and the tree edges match, at least up to the last edge traversed, if the search finishes in the middle of that edge.

Experimental Evaluation

Our experiments ran on an Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz with 32GB of RAM, running version 4.15.0-30-generic x86_64 of the linux kernel. We compare our index variants with the most competitive representatives of previous work for repetitive text collections: the LZ Index (`github.com/migumar2/uiHRDC`), the r-index (`github.com/nicolaprezza/r-index`), and the Hybrid Index with default parameters (`github.com/hferrada/HydridSelfIndex`). As our repetitive collection, we use a set of 80 fully assembled yeast genomes, with a total size of 1GB, from which we choose one random genome as the RLZ reference. To obtain each search time, we average over 10,000 searches for patterns of lengths 10, 20, 40 and 80 extracted at random text positions. The index spaces are given in bits per symbol (bps), whereas the (user) times are given in microseconds per pattern occurrence. Our index and testbed are available at `https://github.com/vsepulve/relz_search`.

Figure 1 shows the breakdown of the space of our index. The RLZ Basic Index uses 0.089 bps, the RLZ Ref Index uses 0.104 bps, and the RLZ Hash Index uses 0.277 bps. The total collection has $n = 1{,}013{,}782{,}646$ symbols over an alphabet of size $\sigma = 4$, the reference is of length $r = 12{,}069{,}408$ and we obtain $z = 605{,}505$ phrases (the average phrase length is $\approx 1{,}654$). The space of the RLZ Basic Index is not far from the 0.077 bps obtained with GDC2 [30], a state-of-the-art RLZ compressor.

Figure 2 (top left) shows the average number of occurrences of each type for each pattern length. Patterns of length 10 appear a few tens of times in the reference and crossing phrases, and thousands of times inside phrases. Longer patterns appear once in the reference, a hundred of times inside phrases, and almost never crossing phrases. We report those occurrences inside phrases in constant time, using B, S, E, and π.

Figure 2 (bottom) gives the average time per query, classified into the time to find

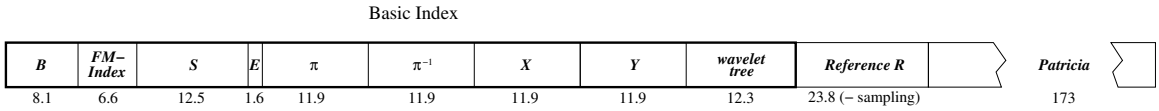

Figure 1: Space breakdown of our index, showing bits per 1000 symbols of the components of the Basic Index, the Ref Index (which adds the explicit reference R), and the Hash Index (which adds the Patricia tree).

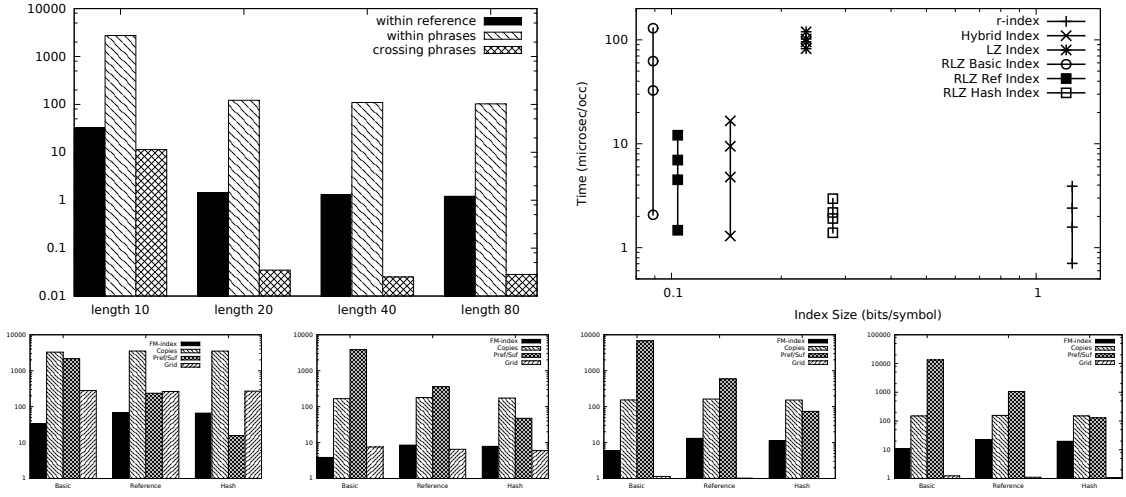

Figure 2: Searching patterns of lengths 10, 20, 40, and 80. Top left: number of occurrences of each type. Top right: Search time per occurrence versus index space (higher times for longer patterns). Bottom: total microseconds per search phase, for each length. Logscale.

the occurrences in the reference (FM-index), to find those inside phrases (Copies), to search for suffixes and reversed prefixes of P (Pref/Suf), and to extract from the grid the occurrences that cross phrases (Grid). The time for Pref/Suf decreases sharply as we move from the Basic to the Reference to the Hash version, as expected. It is almost irrelevant for length 10 (except for the Basic version), but it grows in importance until being dominant; only the Hash version keeps its time under control. The Grid time, instead, loses importance as fewer occurrences crossing phrases are found.

Figure 2 (top right) compares index sizes versus average time per occurrence found. The RLZ Basic Index is always the smallest, using less than 0.1 bps, but it is also slow (2–100 microseconds per occurrence). For example, it uses just 61% of the space of the next smallest index, the Hybrid Index, but it is up to an order of magnitude slower. On patterns of length up to 40, the RLZ Basic Index outperforms the LZ Index in space and time (using 38% of its space), but its need to extract the text from the FM-index makes it eventually reach the LZ Index time for longer patterns.

Storing the text explicitly makes the RLZ Ref Index much faster than the RLZ Basic Index (1–10 microseconds per occurrence), but it still depends sharply on the pattern length. Using slightly over 0.1 bps, this index is still smaller than the Hybrid Index (71% of its space), and about as fast (faster for lengths over 10). The RLZ Ref Index also uses 44% of the LZ Index space, and is an order of magnitude faster.

The dependence on the pattern length is smoothed out in the RLZ Hash Index

(1–3 microseconds per occurrence), at the price of increasing the space up to 0.3 bps. This index is way faster than all the others for lengths over 10, but it uses almost twice the space of the Hybrid Index (and it is also slightly slower on length 10).

The r-index is the fastest index for patterns of length up to 20, but it loses to the RLZ Hash Index on longer patterns. Further, in this collection it is more than 4.5 times larger than the RLZ Hash Index (the BWT has almost 13.5 million runs).

Conclusions

We have introduced a self-index for highly repetitive text collections based on Relative Lempel-Ziv [4]. Our design is a simplification of a previous proposal [9] that has not been implemented. The search complexity of our index, $O((m + occ) \log n)$ for a pattern of length m occurring occ times in a collection of length n, is close to that of the previous version. We reach this complexity with a much simpler design by using a new data structure based on Patricia trees and hashing [31].

Our implementation further simplifies this design and contains an $O(m^2)$ additive penalty in the worst-case search time, but uses small space and locates each pattern occurrence within a few microseconds. We develop other variants that do not use Patricia trees and thus have higher search times (up to hundreds of microseconds per occurrence), but in exchange are about 3 times smaller, outperforming every other structure in terms of space within competitive search time. Compared to previous work, the variant RLZ Ref Index uses half the space of the LZ Index [5] and is an order of magnitude faster. It is also 30% smaller than the Hybrid Index [23] and about as fast (while providing worst-case time guarantees). Our larger RLZ Hash Index is still 4.5 times smaller than the r-index [8] and faster on all but short patterns.

Our use of hashing reduces the false positives of the Patricia trees to zero in our experiments, and this helps decrease search times by up to 17%. Our future work plan includes a faithful implementation of Lemma 1, using a recent practical variant [32, Lem. 5 in extended version], so as to remove the $O(m^2)$ worst-case additive term.

References

[1] Z. D. Sthephens, S. Y. Lee, F. Faghri, R. H. Campbell, Z. Chenxiang, M. J. Efron, R. Iyer, S. Sinha, and G. E. Robinson, "Big data: Astronomical or genomical?" *PLoS Biol.*, vol. 17, no. 7, p. e1002195, 2015.

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

[3] M. H.-Y. Fritz, R. Leinonen, G. Cochrane, and E. Birney, "Efficient storage of high throughput DNA sequencing data using reference-based compression," *Genome Res.*, pp. 734–740, 2011.

[4] S. Kuruppu, S. J. Puglisi, and J. Zobel, "Relative Lempel-Ziv compression of genomes for large-scale storage and retrieval," in *Proc. 17th SPIRE*, 2010, pp. 201–206.

[5] S. Kreft and G. Navarro, "On compressing and indexing repetitive sequences," *Theor. Comp. Sci.*, vol. 483, pp. 115–133, 2013.

[6] F. Claude and G. Navarro, "Self-indexed grammar-based compression," *Fund. Inf.*, vol. 111, no. 3, pp. 313–337, 2010.

[7] V. Mäkinen, G. Navarro, J. Sirén, and N. Välimäki, "Storage and retrieval of highly repetitive sequence collections," *J. Comp. Biol.*, vol. 17, no. 3, pp. 281–308, 2010.

[8] T. Gagie, G. Navarro, and N. Prezza, "Optimal-time text indexing in BWT-runs bounded space," in *Proc. 29th SODA*, 2018, pp. 1459–1477, extended version in `arxiv.org/abs/1809.02792`.
[9] H. H. Do, J. Jansson, K. Sadakane, and W.-K. Sung, "Fast relative Lempel-Ziv self-index for similar sequences," *Theor. Comp. Sci.*, vol. 532, pp. 14–30, 2014.
[10] G. Manzini, "An analysis of the Burrows-Wheeler transform," *J. ACM*, vol. 48, no. 3, pp. 407–430, 2001.
[11] R. M. Karp and M. O. Rabin, "Efficient randomized pattern-matching algorithms," *IBM J. Res. Devel.*, vol. 2, pp. 249–260, 1987.
[12] D. R. Clark, "Compact PAT trees," Ph.D. dissertation, U. Waterloo, Canada, 1996.
[13] D. Okanohara and K. Sadakane, "Practical entropy-compressed rank/select dictionary," in *Proc. 9th ALENEX*, 2007, pp. 60–70.
[14] J. Fischer and V. Heun, "Space-efficient preprocessing schemes for range minimum queries on static arrays," *SIAM J. Comp.*, vol. 40, no. 2, pp. 465–492, 2011.
[15] J. I. Munro, R. Raman, V. Raman, and S. S. Rao, "Succinct representations of permutations and functions," *Theor. Comp. Sci.*, vol. 438, pp. 74–88, 2012.
[16] R. Grossi, A. Gupta, and J. S. Vitter, "High-order entropy-compressed text indexes," in *Proc. 14th SODA*, 2003, pp. 841–850.
[17] G. Navarro, "Wavelet trees for all," *J. Discr. Alg.*, vol. 25, pp. 2–20, 2014.
[18] P. Ferragina and G. Manzini, "Indexing compressed texts," *J. ACM*, vol. 52, no. 4, pp. 552–581, 2005.
[19] D. Belazzougui and G. Navarro, "Alphabet-independent compressed text indexing," *ACM Trans. Alg.*, vol. 10, no. 4, p. article 23, 2014.
[20] F. Claude, A. Fariña, M. Martínez-Prieto, and G. Navarro, "Universal indexes for highly repetitive document collections," *Inf. Sys.*, vol. 61, pp. 1–23, 2016.
[21] M. Burrows and D. Wheeler, "A block sorting lossless data compression algorithm," Digital Equipment Corporation, Tech. Rep. 124, 1994.
[22] H. Ferrada, T. Gagie, T. Hirvola, and S. J. Puglisi, "Hybrid indexes for repetitive datasets," *CoRR*, vol. abs/1306.4037, 2013.
[23] H. Ferrada, D. Kempa, and S. J. Puglisi, "Hybrid indexing revisited," in *Proc. 20th ALENEX*, 2018, pp. 1–8.
[24] S. Muthukrishnan, "Efficient algorithms for document retrieval problems," in *Proc. 13th SODA*, 2002, pp. 657–666.
[25] F. Claude and G. Navarro, "Improved grammar-based compressed indexes," in *Proc. 19th SPIRE*, 2012, pp. 180–192.
[26] P. Bille, I. L. Gørtz, B. Sach, and H. W. Vildhøj, "Time–space trade-offs for longest common extensions," *J. Discr. Alg.*, vol. 25, pp. 42–50, 2014.
[27] S. Gog, T. Beller, A. Moffat, and M. Petri, "From theory to practice: Plug and play with succinct data structures," in *Proc. 13th SEA*, 2014, pp. 326–337.
[28] P. Elias, "Efficient storage and retrieval by content and address of static files," *J. ACM*, vol. 21, no. 2, pp. 246–260, 1974.
[29] D. R. Morrison, "Patricia – practical algorithm to retrieve information coded in alphanumeric," *J. ACM*, vol. 15, no. 4, pp. 514–534, 1968.
[30] S. Deorowicz, A. Danek, and M. Niemiec, "GDC 2: Compression of large collections of genomes," *CoRR*, vol. abs/1503.01624, 2015.
[31] T. Gagie, P. Gawrychowski, J. Kärkkäinen, Y. Nekrich, and S. J. Puglisi, "LZ77-based self-indexing with faster pattern matching," in *Proc. 11th LATIN*, 2014, pp. 731–742.
[32] D. Kempa and D. Kosolobov, "LZ-End parsing in compressed space," in *Proc. 27th DCC*, 2017, pp. 350–359, extended version in `arxiv.org/pdf/1611.01769.pdf`.