# Relative Lempel-Ziv with Constant-Time Random Access

Héctor Ferrada[1,⋆], Travis Gagie[2,⋆⋆], Simon Gog[3,⋆⋆⋆], and Simon J. Puglisi[2,†]

[1] Department of Computer Science
University of Chile, Chile
[2] Department of Computer Science
University of Helsinki, Finland
[3] Institute of Theoretical Informatics
Karlsruhe Institute of Technology, Germany

**Abstract.** Relative Lempel-Ziv (RLZ) is a variant of LZ77 that can compress well collections of similar genomes while still allowing fast random access to them. In theory, at the cost of using sublinear extra space, accessing an arbitrary character takes constant time. We show that even in practice this works quite well: e.g., we can compress 36 *S. cerevisiae* genomes from a total of 464 MB to 11 MB and still support random access to them in under 50 nanoseconds per character, even when the accessed substrings are short. Our theoretical contribution is an optimized representation of RLZ's pointers.

## 1 Introduction

Advances in DNA sequencing have led to the creation of massive genomic databases. In many cases these databases hold collections of genomes from individuals of the same species or closely related species. Such genomes tend to be very similar, so referential compression schemes such as Ziv and Lempel's LZ77 [14] perform very well on them (see, e.g., [1,13]). Supporting fast random access to LZ77-compressed texts is problematic [12] but several authors have proposed variants of LZ77 on which random access is easier: e.g., Kreft and Navarro's LZ-End [7], Kuruppu, Puglisi and Zobel's Relative Lempel-Ziv (RLZ) [8,9] and Deorowicz and Grabowski's GDC [2].

In theory RLZ implemented with compressed bitvectors offers constant-time random access, which is faster than LZ-End, GDC or schemes such as block graphs [4] or FOLCA [10] that are not based on LZ77. The main disadvantage of using compressed bitvectors is their redundancy, which is nevertheless sublinear

⋆ Supported by Fondecyt 1-140796, Chile.
⋆⋆ Supported by Academy of Finland grant 268324.
⋆⋆⋆ This work was carried out while the third author was employed at the University of Melbourne, supported by ARC Grant DP110101743.
† Supported by Academy of Finland grant 258308.

E. Moura and M. Crochemore (Eds.): SPIRE 2014, LNCS 8799, pp. 13–17, 2014.

in the length of the original file. As far as we are aware, such an implementation has never been tried in practice. In this paper we describe an implementation with which we can, e.g., compress 36 *S. cerevisiae* genomes from 464 MB to 11 MB and still support random access to them in under 50 nanoseconds per character, even when the accessed substrings are short. Our theoretical contribution is a compressed representation of RLZ's pointers that is optimized for genomic databases.

## 2 Relative Lempel-Ziv

Given a collection of similar genomes, RLZ works by selecting or generating a reference genome $R$, which it leaves uncompressed (or only entropy-compressed), then compressing each of the remaining genomes relative to $R$. To compress another genome $S[0..n-1]$ relative to $R$, our implementation of RLZ greedily parses $S$ into phrases such that each phrase consists of a substring of $R$ followed by a single character, called a mismatch character. When the alphabet size is constant this can be done in $\mathcal{O}(n)$ time using, e.g., an FM-index [3] for $R$ (see, e.g., [6]).

Kuruppu et al. originally defined RLZ such that each phrase is either a substring of $R$ or a single character. Deorowicz and Grabowski pointed out, however, that with this definition, single-nucleotide polymorphisms (SNPs) — the most common kind of differences between individuals' genomes — tend to cause two phrase breaks each, instead of only one.

If we want only to compress $S$, we need only store a sequence $(\ell_0, p_0, c_0), \ldots, (\ell_{z-1}, p_{z-1}, c_{z-1})$ of triples, where $z$ is the number of phrases. Each triple $(\ell_r, p_r, c_r)$ indicates that the corresponding phrase is $R[p_r..p_r + \ell_r - 1]\, c_r$, or just $c_r$ if $\ell_r = 0$ (in which case $p_r$ is irrelevant). To decompress $S$ later, we simply replace each triple by its phrase.

### 2.1 Compressed Bitvectors

A bitvector $B$ is a binary string that supports $\mathsf{access}$, $\mathsf{rank}$ and $\mathsf{select}$ queries: $B.\mathsf{access}(i)$ returns $B[i]$ (and is often written simply $B[i]$); $B.\mathsf{rank}(i)$ returns the number of 1s in $B[0..i]$; and $B.\mathsf{select}(i)$ returns the position of the $i$th 1 in $B$. The best theoretical bound is due to Pătraşcu [11], who showed how $B$ can be stored in $|B|H_0(B) + \mathcal{O}(|B|/\log^a |B|)$ bits, where $H_0(B)$ is the 0th-order empirical entropy of $B$ and $a$ is any constant, such that all three kinds of queries can be answered in $\mathcal{O}(1)$ time.

### 2.2 Absolute Pointers

If we want to support random access, then we can store a bitvector $B_1[0..n]$ with 1s marking where phrases start in $S$; an array $P[0..z-1] = [p_0, \ldots, p_{z-1}]$; and an array $C[0..z-1] = [c_0, \ldots, c_{z-1}]$. We set $B_1[0] = 0$ so that $B_1.\mathsf{rank}(i)$ is the index of the phrase containing $S[i]$, and we set $B_1[n] = 1$ so that $S[i]$

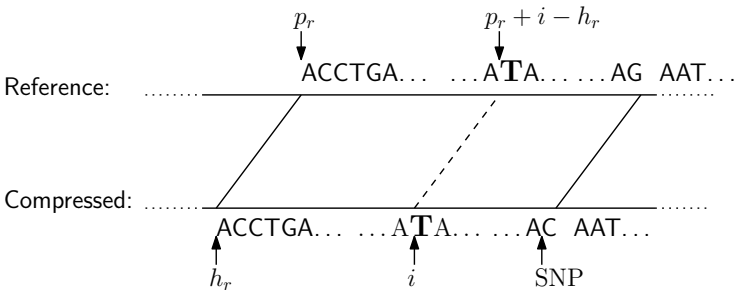


**Fig. 1.** We can use the relative pointer $P'[r] = p_r - h_r$ instead of the absolute pointer $P[r] = p_r$ because $P'[r] + i = p_r + i - h_r = P[r] + i - h_r$

is a mismatch character if and only if $B_1[i+1] = 1$. For simplicity, we assume $B_1.\mathsf{select}(0) = 0$.

To access $S[i]$, we

1. find the index $r = B_1.\mathsf{rank}(i)$ of the phrase containing $S[i]$;
2. check whether $B_1[i+1] = 1$, to see if $S[i]$ is a mismatch character;
3. if so, return $C[r]$;
4. if not, find the starting position $h_r = B_1.\mathsf{select}(r)$ of the $r$th phrase;
5. return $R[P[r] + i - h_r]$.

We say this approach uses *absolute* pointers because each cell of $P$ contains a direct pointer to the starting position of the appropriate substring of $R$.

### 2.3 Relative Pointers

We can simplify our access procedure somewhat if we store *relative* pointers instead of absolute pointers. That is, we store an array $P'[0..z-1] = [p_0 - h_0, \ldots, p_{z-1} - h_{z-1}]$, where $h_r$ is again the starting position in $S$ of phrase $r$. Notice that $P'[r] + i = p_r + i - h_r = P[r] + i - h_r$ for any $i$ — see Figure 1 — so we no longer need $\mathsf{select}$ to access $S$. Specifically, to access $S[i]$ we now

1. find the index $r = B_1.\mathsf{rank}(i)$ of the phrase containing $S[i]$;
2. check whether $B_1[i+1] = 1$, to see if $S[i]$ is a mismatch character;
3. if so, return $C[r]$;
4. if not, return $R[P'[r] + i]$.

### 2.4 Compressed Pointers

Another benefit of relative pointers is that we can compress $P'$ more easily than $P$. For example, Kuruppu et al. noted that the difference between phrases' absolute pointers is often the total length of the phrases between them, and used

**Table 1.** Random access time (for extraction lengths $m = 8, 64, 512, 4096$ characters) for RLZ and GDC for a collection of 36 *S. cerevisiae* genomes, totalling 464 MB before compression. The two GDC rows correspond to different settings for the R-block size and D-block size (see [2]). Times are given in nanoseconds per extracted character averaged over 10 million extractions.

| Method | Size (MB) | $m = 8$ | $m = 64$ | $m = 512$ | $m = 4096$ |
|---|---|---|---|---|---|
| RLZ | 11 | 50 | 7 | 3 | 2 |
| GDC-ra-$2^8$-$2^8$ | 14 | 500 | 68 | 15 | 8 |
| GDC-ra-$2^{12}$-$2^{12}$ | 10 | 750 | 102 | 19 | 8 |

that observation to achieve better compression by discarding pointers that can be computed from earlier pointers and phrases' lengths. They did not support fast random access with that implementation, but they pointed out that it could be reintroduced by sampling the missing pointers, creating a tradeoff between compression and access time.

The most likely explanation for Kuruppu et al.'s observation is that, if phrase $r$ breaks because of an SNP, then usually $p_{r+1} = p_r + \ell_r + 1$, in which case $P'[r+1] = P'[r]$. To take advantage of this, we run-length compress $P'$ as follows: we store a bitvector $B_2[0..z-1]$ with 1s marking the relevant pointers in $P'$ that differ from the preceding relevant pointers; we store an array $P''$ containing the pointers in $B_2$; and then we discard $P'$. (Recall that a pointer is irrelevant if its phrase is only a mismatch character, and relevant otherwise.)

We set $P''[0]$ to be the first relevant pointer in $P'$ but we do not mark it in $B_2$, so that $B_2.\mathsf{rank}(r)$ is the index of the run in $P$ containing the pointer for phrase $r$. The last step in our access procedure now becomes "if not, return $R[P''[B_2.\mathsf{rank}(r)] + i]$".

## 3 Experiments

We implemented the above scheme in `C++` using the Succinct Data Structure Library (SDSL) [5] version 2.0.1 (available at `https://github.com/simongog/sdsl-lite`) for the bitvectors. As a baseline we also tested Deorowicz and Grabowski's GDC data structure, which is the best RLZ variant supporting random access of which we are aware.

*Setup.* We performed experiments on a machine equipped with a 3.16GHz Intel Core 2 Duo CPU with 6144KiB L2 cache and 4GiB 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 12.04, 64bit) running kernel 3.2.0. All programs were compiled using `g++` version 4.8 with `-O3 -static -DNDEBUG` options. All reported runtimes are recorded with the C `clock` function.

*Data.* We tested our implementation on a collection of 36 *S. cerevisiae* (yeast) genomes, each about 12 MB long, and 464 MB in total.

*Mismatch Characters.* The genomes were over the alphabet {A, C, G, T, N} but there were relatively few Ns in the array $C$ of mismatch characters. Because of this, we stored in a hash table the positions of all the Ns in $C$; replaced the Ns by As; and packed the mismatch characters into two bits each. We considered $C$ blocks of 20 mismatch characters and stored a binary string in which the $i$th bit indicated whether the $i$th block originally contained any Ns. Whenever we read an A from $C$, we checked the binary string to see if the block containing that A originally contained any Ns and, if so, checked the hash table to see if that particular A was originally an N.

*Results.* Sizes and times for random access are shown in Table 1. Our implementation of RLZ had very fast extraction times for short substrings. For example, it was over 10 times faster than GDC for substrings of length 8. For longer substrings the gap between the speed of the two approaches narrowed, but even for 4KB substrings RLZ was still more than 4 times faster than GDC.

## References


1. Deorowicz, S., Danek, A., Grabowski, S.: Genome compression: A novel approach for large collections. Bioinformatics 29(20), 2572–2578 (2013)
2. Deorowicz, S., Grabowski, S.: Robust relative compression of genomes with random access. Bioinformatics 27(21), 2979–2986 (2011)
3. Ferragina, P., Manzini, G.: Indexing compressed text. J. ACM 52(4), 552–581 (2005)
4. Gagie, T., Gawrychowski, P., Puglisi, S.J.: Faster approximate pattern matching in compressed repetitive texts. In: Asano, T., Nakano, S.-i., Okamoto, Y., Watanabe, O. (eds.) ISAAC 2011. LNCS, vol. 7074, pp. 653–662. Springer, Heidelberg (2011)
5. Gog, S., Beller, T., Moffat, A., Petri, M.: From theory to practice: Plug and play with succinct data structures. In: Gudmundsson, J., Katajainen, J. (eds.) SEA 2014. LNCS, vol. 8504, pp. 326–337. Springer, Heidelberg (2014)
6. Kärkkäinen, J., Kempa, D., Puglisi, S.J.: Lightweight Lempel-Ziv parsing. In: Bonifaci, V., Demetrescu, C., Marchetti-Spaccamela, A. (eds.) SEA 2013. LNCS, vol. 7933, pp. 139–150. Springer, Heidelberg (2013)
7. Kreft, S., Navarro, G.: On compressing and indexing repetitive sequences. Theor. Comp. Sci. 483, 115–133 (2013)
8. Kuruppu, S., Puglisi, S.J., Zobel, J.: Relative Lempel-Ziv compression of genomes for large-scale storage and retrieval. In: Chavez, E., Lonardi, S. (eds.) SPIRE 2010. LNCS, vol. 6393, pp. 201–206. Springer, Heidelberg (2010)
9. Kuruppu, S., Puglisi, S.J., Zobel, J.: Optimized relative Lempel-Ziv compression of genomes. In: Proc. ACSC, pp. 91–98 (2011)
10. Maruyama, S., Tabei, Y., Sakamoto, H., Sadakane, K.: Fully-online grammar compression. In: Kurland, O., Lewenstein, M., Porat, E. (eds.) SPIRE 2013. LNCS, vol. 8214, pp. 218–229. Springer, Heidelberg (2013)
11. Pătraşcu, M.: Succincter. In: Proc. FOCS, pp. 305–313 (2008)
12. Verbin, E., Yu, W.: Data structure lower bounds on random access to grammar-compressed strings. In: Fischer, J., Sanders, P. (eds.) CPM 2013. LNCS, vol. 7922, pp. 247–258. Springer, Heidelberg (2013)
13. Wandelt, S., Leser, U.: FRESCO: Referential compression of highly-similar sequences. IEEE Trans. Comp. Bio. Bioinf. 10(5), 1275–1288 (2013)
14. Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Trans. Inf. Theor. 23(3), 337–343 (1977)