Relative Lempel-Ziv Compression of Genomes for Large-Scale Storage and Retrieval[⋆]

Shanika Kuruppu[1], Simon J. Puglisi[2], and Justin Zobel[1]

[1] National ICT Australia
Department of Computer Science & Software Engineering,
University of Melbourne
`{kuruppu,jz}@csse.unimelb.edu.au`
[2] School of Computer Science and Information Technology,
Royal Melbourne Institute of Technology, Australia
`simon.puglisi@rmit.edu.au`

Abstract. *Self-indexes* – data structures that simultaneously provide fast search of and access to compressed text – are promising for genomic data but in their usual form are not able to exploit the high level of replication present in a collection of related genomes. Our 'RLZ' approach is to store a self-index for a base sequence and then compress every other sequence as an LZ77 encoding relative to the base. For a collection of r sequences totaling N bases, with a total of s point mutations from a base sequence of length n, this representation requires just $nH_k(T) + s\log n + s\log\frac{N}{s} + O(s)$ bits. At the cost of negligible extra space, access to ℓ consecutive symbols requires $O(\ell + \log n)$ time. Our experiments show that, for example, RLZ can represent individual human genomes in around 0.1 bits per base while supporting rapid access and using relatively little memory.

1 Introduction

The emergence of high-throughput sequencing technologies, capable of sequencing entire genomes in a single run, has lead to a dramatic change in the number and type of sequencing projects being undertaken. In particular, it is now feasible to acquire and study variations between many individual genomes of organisms from the same species. While the total size of these sets of genomes will be large, individual genomes will not greatly vary, and so these collections represent new challenges for compression and indexing.

In this paper we address the following problem.

Definition 1 ([7]). *Given a collection $\mathcal{C}$ of r sequences $T^k \in \mathcal{C}$ such that $|T^k| = n$ for $1 \le k \le r$ and $\sum_{k=1}^{r} |T^k| = N$, where $T^2, T^3, \ldots, T^r$ are mutated*

[⋆] This work was supported by the Australian Research Council and the NICTA Victorian Research Laboratory. NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Center of Excellence program.

copies of the base sequence T^1 *containing overall s point mutations, the* repetitive collection indexing problem *is to efficiently store* $\mathcal{C}$ *while allowing queries of the form* display(i, j, k) *to efficiently return the substring* $T^k[i..j]$.

We describe a solution to this problem that requires $nH_k(T^1) + s \log n + s \log \frac{N}{s} + O(s)$ bits of space and $O(\ell + \log^{1+\epsilon} n)$ time to return a requested substring of length ℓ for any constant $\epsilon > 0$, assuming a constant alphabet size.

Our approach is to use the base sequence as a dictionary for compression of the other sequences. The collection is parsed into factors in an LZ77 [11] manner, but references to substituted strings are restricted to be in the base sequence only. Arbitrary substrings can then be decoded by reference to the base sequence. This work is inspired by the recent work of Mäkinen et al. [7] who, to our knowledge, were the first to tackle the above problem.

2 Background and Basic Tools

BioCompress [5] was the first compression algorithm to be specific to DNA sequence compression, by simple modifications such as encoding nucleotides with 2 bits per base and detecting reverse complement repeats. Two further variations on the *BioCompress* theme, *Cfact* [10] and *Off-line* [1], both work in rounds, greedily replacing duplicate text with shorter codes. *GenCompress* [3] showed that by considering approximate repeats the results could be improved. Since *GenCompress*, most DNA compression algorithms have been based on efficient methods of approximate repeat detection.

Many of these early DNA compression algorithms are only able to compress small files. Due to the latest advances in DNA sequencing technologies, much larger DNA sequencing projects are underway. As a result, many individual genomes from the same species are being sequenced, creating a large amount of redundancy. Recent algorithms have specifically addressed the issue of compressing DNA data from the same species. Christley et al. [4] compresses variation data from human genomes and encodes the mutations and indels with respect to the human reference sequence and known variations recorded in a SNP database. However, large resources are required to be shared among users of the compressed data (4.2 GB of reference and SNPs in Chirstley et al.'s software). Furthermore, it does not support random access into the sequences.

Before explaining our method we introduce notation and review several pieces of algorithmic machinery on which our results rely.

Strings. A string $T = T[0..n] = T[0]T[1] \ldots T[n]$ is a sequence of $n + 1 = |T|$ symbols. The first n symbols of T are drawn from a constant ordered alphabet, Σ. The final character $T[n]$ is a special "end of string" character, \$, distinct from and lexicographically smaller than all the other characters in Σ. We write $T[i..j]$ to represent the ***substring*** $T[i]T[i+1] \cdots T[j]$ of T that starts at position i and ends at position j. For substring $T[i..j]$, if $j = n$ (resp. $i = 0$) we call the substring a suffix (resp. prefix) of T. With $\hat{T}$ we denote the reverse of T.

Suffix Arrays and Self-Indexes. The ***suffix array*** of T, denoted SA_T or just SA, when the context is clear, is an array $\mathrm{SA}[0..n]$ that contains a permutation of the integers $0..n$ such that $T[\mathrm{SA}[0]..n] < T[\mathrm{SA}[1]..n] < \cdots < T[\mathrm{SA}[n]..n]$. In other words, $\mathrm{SA}[j] = i$ iff $T[i..n]$ is the jth suffix of T in ascending lexicographical order. All the positions of occurrence in T of a given pattern, $P[1..m]$, lie in a contiguous range of the suffix array $\mathrm{SA}[sp..ep]$. In recent years, successful attempts have been made to compress suffix arrays, and data structures call *self-indexes* have emerged [8]. A self-index of a text T allows the following functionality, all in compressed space: (i) extract (display) any substring $T[s..e]$, (ii) find the range $sp..ep$ for a pattern P, and (iii) return $\mathrm{SA}[i]$ for any i.

Relative Lempel-Ziv Factorization. Given two strings T and S, the Lempel-Ziv factorization (or parsing) of T relative to S, denoted $\mathrm{LZ}(T|S)$, is a factorization $T = w_0 w_1 w_2 \ldots w_z$ where w_0 is the empty string and for $i > 0$ each factor (string) w_i is either: (a) a letter which does not occur in S; or otherwise (b) the longest prefix of $T[|w_0 \ldots w_{i-1}|..|T|]$ that occurs as a substring of S. For example, if $S = abaababa$ and $T = aabacaab$ then in $\mathrm{LZ}(T|S)$ we have $w_1 = aaba$, $w_2 = c$ and $w_3 = aab$. It is convenient to specify the factors not as strings, but as (p_i, ℓ_i) pairs, where p_i denotes the starting position in S of an occurrence[1] of factor w_i (or a letter if w_i is by rule (a)) and ℓ_i denotes the length of the factor (or is zero if w_i is by rule (a)). Thus, in our example: $\mathrm{LZ}(T|S) = (3,4)(c,0)(2,3)$. For convenience, we assume no factors are generated by rule (a) above; that is, if c occurs in T_i for $i \geq 2$ then c also occurs in T_1. If T_1 is not so composed we can simply add the at most $\sigma - 1$ missing symbols to the end of it.

Compressed Integer Sets. We make use of a compressed set representation due to Okanohara and Sadakane [9] (called "sdarray" in their paper). Given a set S of m integers over a universe u this data structure supports the operations: $\mathrm{rank}(S, i)$, returning the number of item in S less than or equal to i; and $\mathrm{select}(S, i)$, returning the value of the ith item in S. The data structure requires $m \log \frac{u}{m} + \mathrm{O}(m)$ bits and $\mathrm{O}(1)$ time for select and $\mathrm{O}(\log \frac{u}{m})$ time for rank.

3 Storing and Accessing Related Genomes

We now describe how to store a collection of related sequences, while still allowing efficient access to arbitrary substrings of any of the constituent sequences.

Lemma 2. *Given a collection $\mathcal{C}$ of r sequences $T^k \in \mathcal{C}$ such that $|T^k| = n$ for $1 \leq k \leq r$ and $\sum_{k=1}^{r} |T^k| = r \cdot n = N$, with*

$$\mathrm{LZ}(T^i|T^1) = (p_1, \ell_1), (p_2, \ell_2), \ldots, (p_{z_i}, \ell_{z_i}),$$

for $2 \leq i \leq r$ and $z = \sum_{i=2}^{r} z_i$, we can store $\mathcal{C}$ in at most $n \log \sigma + z \log n + z \log \frac{N}{z} + \mathrm{O}(z)$ bits such that any substring $T^k[s..e]$ can be output in $\mathrm{O}(e - s + \log \frac{N}{z})$ time.

[1] There may be more than one occurrence; for our purposes here it does not matter to which one p_i refers.

Proof. T^1, the base sequence, is stored in $n \log \sigma$ bits (in the obvious way).

Let $Z^i = \mathrm{LZ}(T^i|T^1) = (p_1, \ell_1), (p_2, \ell_2), \ldots, (p_z, \ell_{z_i})$ for $i > 1$ be the parsing of text T^i relative to T^1. We store Z^i in two pieces. The position components of each factor, $p_1, \ldots, p_{z_i}$ are stored in a table $P_i[1..z_i]$ taking $z_i \log n$ bits. P_i is simply a concatenation of the bits representing $p_1, \ldots, p_z$. Each entry in P_i is $\log n$ bits long, allowing access to any entry $p_j = P_i[j]$ in O(1) time. The length components are stored in a compressed integer set, L_i, containing the values $j = \sum_{k=1}^{u} \ell_k$ for all $u \in 1..z_i$. In other words, the values in L_i are the starting positions of factors in T_i. Via a select query, L_i allows us to access a given p_j as $P_i[\mathrm{select}(L_i, j)]$; and the length of the jth factor, ℓ_j, is simply $\mathrm{select}(L_i, j+1) - \mathrm{select}(L_i, j)$ for $j \in 1..z-1$ (because of the terminating sentinel we always have $\ell_z = 1$). Furthermore, the factor that $T^i[j]$ falls in is given by $\mathrm{rank}(L_i, j)$. L_i is stored in the “sdarray” representation [9], which requires $z \log \frac{N}{z} + \mathrm{O}(z)$ bits and allows rank in $\mathrm{O}(\log \frac{N}{z})$ time and select in O(1).

As described, P_i and L_i allow, given j, fast access to p_j and ℓ_j: simply a select query on L_i to get ℓ_j and a lookup on P_i to retrieve p_j. Given a position k in sequence T^i, we can determine the factor in which k lies by issuing a rank query on L_i, in particular $\mathrm{rank}(L_i, k)$. To find a series of consecutive factors we only need to use rank in obtaining the first factor. For the others, the ℓ values can be retrieved using repeated select queries on L_i and the p values by accessing consecutive fields in P_i, both in constant time per factor. This observation allows us to extract any substring $T^i[s..e]$ in $\mathrm{O}(e - s + \log \frac{N}{z})$ time overall. □

We can reduce the $n \log \sigma$ term in the size of the above data structure to nH_k bits by storing T^1 as a self-index instead of in plain form. This increases the cost to access a substring of length ℓ to $\mathrm{O}(\ell \log^{1+\epsilon} n + \log \frac{N}{z})$ worst-case time. It is possible to build our data structure in $\mathrm{O}(n + N \log n)$ time and $n \log \sigma + n \log n$ bits of extra space. The basic idea is to build the suffix array for T_1 and “stream” every other sequence against it to generate the $\mathrm{LZ}(T^i|T^1)$ parsings.

4 Experimental Results

The compression performance of our algorithm, which we call RLZ, is compared to the algorithms `XM` [2], which is known to currently be the best single sequence DNA compression algorithm, Comrad [6], which specialises in compression of large related DNA datasets, and RLCSA, an implementation of a self-index from Mäkinen et al. [7]. The RLZ *display()* function is also compared to RLCSA [7].

Table 1 compares RLZ with RLCSA, Comrad and `XM` by compressing datasets containing many real biological sequences that slightly vary from each other. The datasets are *S. coronavirus* with 141 sequences, *S. cerevisiae* with 39 genomes, *S. paradoxus* with 36 genomes, and *H. sapien* with 4 genomes.

The compression performance of RLZ is varied (Table 2). Unsurprisingly, for the smaller datasets, `XM` has the best compression results. For the larger dataset, RLZ is produces better compression results than Comrad while using less memory (≈45 Mbyte for the yeast sets). RLCSA doesn’t perform as well as the other algorithms, but it uses less memory (≈100 Mbyte for yeast) and

Table 1. Compression results for four repetitive collections. The first row is the original size for all datasets (Size in Megabases rather than Mbytes), the remaining rows are the compression performance of RLZ, RLCSA, Comrad and XM algorithms. The two columns per dataset show the size in Mbytes and the 0-order entropy (in bits per base).

Dataset	S. coronavirus		S. cerevisiae		S. paradoxus		H. sapien	
	Size (Mbyte)	Ent. (bpb)	Size (Mbyte)	Ent. (bpb)	Size (Mbyte)	Ent. (bpb)	Size (Mbyte)	Ent. (bpb)
Original	4.19	1.98	485.87	2.18	429.27	2.12	12066.06	2.18
RLZ	0.08	0.15	17.89	0.29	23.38	0.44	754.43	**0.50**
RLCSA	0.22	0.43	41.39	0.57	47.35	0.88	3834.82	2.54
Comrad	0.09	0.18	15.29	**0.25**	18.33	0.34	2176	1.44
XM	0.03	**0.06**	74.53	1.26	13.17	**0.25**	—	—

Table 2. Compression and decompression times (in seconds)

Dataset	S. coronavirus		S. cerevisiae		S. paradoxus		H. sapien	
	Comp. (sec)	Decom. (sec)	Comp. (sec)	Decom. (sec)	Comp. (sec)	Decom. (sec)	Comp. (sec)	Decom. (sec)
RLZ	**2**	**1**	**213**	**12**	**260**	**10**	**9874**	**172**
RLCSA	6	2	781	312	740	295	34525	14538
Comrad	10	16	1070	45	1068	50	28442	1666
XM	43	62	18990	17926	30580	28920	—	—

Table 3. *display()* times for RLZ and RLCSA for varying query lengths. Query datasets contain 1000 queries each with the same length. The times for each algorithm are in microseconds per character extracted. The times are an average of 5 consecutive runs per dataset. RLZ used 28.71 MByte of memory and RLCSA used 47.35 MByte.

Query Length	10	100	1000	10000	100000
RLCSA (μsec/char)	18.00	2.40	0.85	0.71	0.71
RLZ (μsec/char)	**0.2300**	**0.0250**	**0.0046**	**0.0025**	**0.0022**

is faster than Comrad and XM. Overall, RLZ compress and decompress much faster, and uses drastically less memory, than RLCSA, Comrad and XM.

For the *H. sapien* dataset, each chromosome of each dataset was compressed against the respective reference genome chromosomes for RLZ and RLCSA. We do not report XM results since it took nearly 6 hours for a single chromosome 1 sequence to compress. RLZ performed very well on this dataset compared to RLCSA and Comrad (RLZ used ≈1 Gbyte of memory while RLCSA and Comrad used ≈2 and ≈16 Gbyte respectively). With the inclusion of the 7-zipped human reference genome, the total dataset of three human genome sequences (summing to 12 Gbase) can be represented in just under 755 MByte with RLZ; of this, 643 Mbyte is the overhead for the base sequence, and we anticipate that roughly 27,000 human genomes could be stored in a terabyte, a massive increase on the 1500 or so that could be stored using methods such as `7-zip`.

Results for the *display(i,s,e)* (retrieve substring $T^i[s..e]$) function are in Table 3. RLZ displays substrings significantly faster than RLCSA for small query lengths and continues its good performance for even larger query lengths, with approximately 0.022 (μsec/c) for RLZ compared to 0.77 (μsec/c) for RLCSA.

Tests were conducted on a 2.6 GHz Dual-Core AMD Opteron CPU with 32Gb RAM and 512K cache running Ubuntu 8.04 OS. The compiler was gcc v4.2.4 with the -O9 option.

5 Discussion

A key issue is choice of a reference sequence. While selecting the reference genome is a simple matter for the *yeast* datasets, it may not be a good representation of the other individual sequences. To observe the compression performance for different reference sequence choices, we compressed the dataset by selecting each genome in turn as the reference. Selecting a genome such as *DBVPG6765* leads to 16.5 MByte as opposed to selecting *UWOPS05_227_2*, which led to 24.5 MByte. We also explored the effect on compression when a reference sequence that is unrelated to the original dataset is selected. 35 genomes of *S. paradoxus* (excluding *REF* genome) was compressed against the *S. cerevisiae REF* genome. The compressed size was 112.09 MByte compared to the result of 2.04 MByte when the *S. paradoxus REF* genome was used. RLZ's effectiveness is at its best when compressing related sequences and care must be taken when selecting a reference sequence.

References

1. Apostolico, A., Lonardi, S.: Compression of biological sequences by greedy off-line textual substitution. In: Proc. IEEE DCC, pp. 143–152 (2000)
2. Cao, M.D., Dix, T., Allison, L., Mears, C.: A simple statistical algorithm for biological sequence compression. In: Proc. IEEE DCC, pp. 43–52 (2007)
3. Chen, X., Kwong, S., Li, M.: A compression algorithm for DNA sequences and its applications in genome comparison. In: Proc. RECOMB, p. 107. ACM, New York (2000)
4. Christley, S., Lu, Y., Li, C., Xie, X.: Human genomes as email attachments. Bioinformatics 25(2), 274–275 (2009)
5. Grumbach, S., Tahi, F.: Compression of DNA sequences. In: Proc. IEEE DCC, pp. 340–350 (1993)
6. Kuruppu, S., Beresford-Smith, B., Conway, T., Zobel, J.: Repetition-based compression of large DNA datasets. In: Poster at RECOMB (2009)
7. Mäkinen, V., Navarro, G., Sirén, J., Välimäki, N.: Storage and retrieval of highly repetitive sequence collections. J. Computational Biology 17(3), 281–308 (2010)
8. Navarro, G., Mäkinen, V.: Compressed full text indexes. ACM Computing Surveys 39(1) (2007)
9. Okanohara, D., Sadakane, K.: Practical entropy-compressed rank/select dictionary. In: Proc. ALENEX. SIAM, Philadelphia (2007)
10. Rivals, E., Delahaye, J., Dauchet, M., Delgrange, O.: A guaranteed compression scheme for repetitive DNA sequences. In: Proc. IEEE DCC, p. 453 (1996)
11. Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Transactions on Information Theory 23(3), 337–343 (1977)