Optimized Relative Lempel-Ziv Compression of Genomes

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

[1] NICTA VRL, Department of Computer Science and Software Engineering
The University of Melbourne, Parkville, Victoria 3010
Email: {kuruppu,jz}@csse.unimelb.edu.au

[2] School of Computer Science and Information Technology
RMIT University, Melbourne, Victoria 3001
Email: simon.puglisi@rmit.edu.au

Abstract

High-throughput sequencing technologies make it possible to rapidly acquire large numbers of individual genomes, which, for a given organism, vary only slightly from one to another. Such repetitive and large sequence collections are a unique challange for compression. In previous work we described the RLZ algorithm, which greedily parses each genome into factors, represented as position and length pairs, which identify the corresponding material in a reference genome. RLZ provides effective compression in a single pass over the collection, and the final compressed representation allows rapid random access to arbitrary substrings. In this paper we explore several improvements to the RLZ algorithm. We find that simple non-greedy parsings can significantly improve compression performance and discover a strong correlation between the starting positions of long factors and their positions in the reference. This property is computationally inexpensive to detect and can be exploited to improve compression by nearly 50% compared to the original RLZ encoding, while simultaneously providing faster decompression.

Keywords: DNA, Compression, Lempel-Ziv, LZ77, Suffix Array, Subsequence matching

1 Introduction

Genetic sequencing of many individuals from the same species is vital for a better understanding of variation between individuals and will ultimately lead to improved medical treatments and evolutionary understanding. With the recent advent of high-throughput sequencing technology, large scale acquisition of genomes is becoming a common exercise. Projects such as the 1000 Genomes project,[1], the proposed Genome 10K vertebrate genome project (Haussler et al. 2009), and many other similar activities are leading to collections of large quantities of highly redundant DNA sequences, creating challenges for efficient storage and retrieval of these sequences.

In previous work, we proposed RLZ (Kuruppu et al. 2010), an algorithm that compresses a collection of genomes or sequences from the same species with respect to the reference sequence for that species using a simple greedy technique, akin to LZ77 parsing (Ziv & Lempel 1977). RLZ parses a genome into factors, or substrings, each of which occurs in the reference sequence. Having parsed a genome up to position i, the next factor is the longest match starting at i, which occurs anywhere in the reference. (We delay a precise description of RLZ until Section 3.) Compared to other DNA compression methods, RLZ is fast, and is able to achieve good compression; only *XMcompress* (Cao et al. 2007) proved significantly more effective, and was orders of magnitude slower.

While the greedy approach of taking the longest match at each point in the parsing is fast and effective, in this paper we show that significantly better compression performance can be obtained by simple non-greedy techniques, and by a more judicious treatment of the factors produced by the parsing. The refined approach is also intended to better handle increasing genetic distance between organisms, by more efficient handling of short factors, which arise in the presence of numerous mutations. As our results show, for our application area – compression of the DNA of related organisms – the size is nearly halved, equalling or bettering all previous methods.

We first review related work on DNA compression. Section 3 details relative Lempel-Ziv parsing, and the basic (greedy) RLZ approach for compressing a collection of related genomes. Then, in Section 4, we describe the integration of non-greedy parsing with RLZ, including the encoding of small factors and an efficient parsing algorithm to keep compression speed fast. Section 5 explores a phenomenon manifest by the RLZ algorithm when compressing a pair of related genomes: a strong correlation between a factor's start point and its position of occurrence in the reference. We show how this phenomenon can be exploited and combined with non-greedy parsing to simultaneously attain handsome improvements in both compression performance and decompression speed. Section 6 details the results of experiments with the new approach, before conclusions and reflections are offered in Section 7.

2 Compression of DNA

DNA sequence compression was introduced by Grumbach and Tahi with the *BioCompress* algorithm (Grumbach & Tahi 1993). While *BioCompress* fol-

This work was supported by the Australian Research Council and by 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.

[1] http://www.1000genomes.org/page.php

lows the principle ideas of LZ77 (Ziv & Lempel 1977), the algorithm is made DNA-specific by simple modifications such as encoding nucleotides with 2 bits per base and detecting reverse complement repeats. BioCompress showed that DNA-specific compression algorithms could outperform general-purpose compression algorithms. Two further variations on the *BioCompress* theme are *Cfact* (Rivals et al. 1996) and *Off-line* (Apostolico & Lonardi 2000).

The above algorithms and their variants produce promising compression results for DNA sequences using exact repeat detection. However, *GenCompress* (Chen et al. 2000) showed that by considering approximate repeats, these results can be improved. Due to SNPs (single nucleotide polymorphisms) and mutations that occur over time, repeated regions between DNA sequences are not necessarily exact, even for sequences from the same species. Since *GenCompress*, most DNA compression algorithms, including *CTW+LZ* (Matsumoto et al. 2000), *DNACompress* (Chen et al. 2002), *DNAPack* (Behzadi & Fessant 2005), *GeNML* (Korodi & Tabus 2007), and *XMcompress* (Cao et al. 2007), have been based on efficient methods of approximate repeat detection.

Early DNA compression algorithms were only able to compress very small files. The first compressor for a large genome was the statistical algorithm *NML* (Korodi & Tabus 2005) and its successor *GeNML* (Korodi & Tabus 2007). These algorithms divide the input sequences into blocks and compress each block using a *Maximum Likelihood Model*. *XMcompress* (Cao et al. 2007) is another statistical compression algorithm, which uses "experts" to determine the probability distribution of each symbol, which in turn is used to encode the symbol. The algorithm is able to compress a human genome using less resources than *GeNML* while producing better compression results, and is the best single sequence compression algorithm we are aware of.

Two recent algorithms have focused on compressing large datasets of DNA sequences from the same species. Christley et al. (2009) uses various encoding techniques to compress variation data (mutations and indels) of human genomes. The compression is relative to the human reference sequence and known variations in a SNP database. While the method produces excellent compression results for human variation data, a large amount of overhead is required to store the reference genome and SNP database. The method does not support compression of assembled genomes nor random access into the compressed data. A similar approach is taken by Brandon et al. (2009).

Mäkinen et al. (2010) proposed algorithms that not only compress sets of related genomes, but also support the following retrieval functionality: *display(i,j,k)*, which returns the substring from position j to k in sequence i; *count(p)*, which returns the number of occurrences of substring p; and *locate(p)*, which returns the positions where p substring occurs in the collection. This family of *self-indexing* techniques achieves good compression results, as well as good search times and is the inspiration behind the RLZ algorithm. Our previous work (Kuruppu et al. 2010) shows that RLZ provides better compression and much faster random access times than the data structures of Mäkinen et al. As it is the basis for our present work, we describe RLZ in finer detail in the next section.

3 The RLZ Algorithm

The input to the RLZ algorithm is a set of sequences, as follows:

Definition 1. *Let $\mathcal{C}$ be a collection of r sequences. Each sequence $T^k \in \mathcal{C}$ has length n, where $1 \le k \le r$ and $N = \sum_{k=1}^{r} |T^k|$.*

The sequence T^1 is called the reference sequence. The RLZ algorithm takes each other sequence in the collection and encodes it as a series of *factors* (substrings) that occur in the reference sequence. The manner in which each sequence is broken into factors is similar to the famous LZ77 parsing (Ziv & Lempel 1977). We now give a precise definition.

Given two strings T and S, the Lempel-Ziv factorisation (or parsing) of T relative to S, denoted $\mathrm{LZ}(T|S)$, is a factorisation $T = w_0w_1w_2\ldots w_z$ 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 $T[|w_0 \ldots w_{i-1}|..|T|]$ which occurs as a substring of S.

For example, if $S = tcttctct$ and $T = ttctgttc$ then in $\mathrm{LZ}(T|S)$ we have $w_1 = ttct$, $w_2 = g$ and $w_3 = ttc$. For the purposes of compression, factors are specified not as strings, but as (p_i, ℓ_i) pairs. Each p_i denotes the starting position in S of an occurrence of factor w_i (or a letter if w_i is generated by rule (a)) and ℓ_i denotes the length of the factor (or is zero if w_i is generated by rule (a)). Thus, in our example above, we have:

$$\mathrm{LZ}(T|S) = (2, 4)(g, 0)(2, 3).$$

For technical convenience, for the remainder of this paper we assume that no factors are generated by rule (a) above; that is, if a symbol c occurs in T then c also occurs in S. For DNA strings, this is not an unreasonable assumption, but for other types of data (even proteins) it may be flawed. However, if S is not so composed we can simply add the at most $\sigma - 1$ missing symbols at the end.

We now define the RLZ encoding of the collection.

Definition 2 (RLZ)**.** *Let T^1 be the reference sequence. Each sequence, T^i for $2 \le i \le r$ is represented with respect to T^1 as*

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

resulting in z factors in total where $z = \sum_{i=2}^{r} z_i$.

Using this representation, the collection $\mathcal{C}$ can be stored in at most $n \log \sigma + z \log n + z \log \frac{N}{z} + \mathrm{O}(z)$ bits. In this bound, the $n \log \sigma$ term is for the reference sequence, which is stored uncompressed; the $z \log n$ term is for the p_i components of each factor, which are pointers into the reference; and the $z \log \frac{N}{z}$ term is for (essentially) the Golomb encoding of the ℓ_i components.

As a baseline in this paper we assume that RLZ encodes each p_i component using $\log n$ bits and a Golomb code ($M = 64$) for each ℓ_i. We refer to this encoding method, combined with greedy parsing, as *standard* RLZ. In this paper we concentrate only on raw compression and not on random access. However, we make some comments on how random access can be acheived in the final section.

4 Non-greedy parsing

In the standard relative LZ parsing algorithm described above, we take, at each point in the parse, the factor which has the longest match in the reference sequence. This is a greedy decision, which does not account for the possibility that a smaller overall encoding may be achieved by taking a shorter match (leading to a different parsing). This section explores such a *non-greedy* approach.

For a non-greedy strategy to be effective, the (p_i, ℓ_i) factor pairs must be coded with a variable-length code: it is well known (Ferragina et al. 2009, Horspool 1995) that if a fixed number of bits is used to code each pair, then the greedy method does as well as any other. The particular variability we focus on is the encoding of *short factors*, which, given the small DNA alphabet, can be encoded much more succinctly as literal strings than as (p_i, ℓ_i) pairs.

Lookahead by h. The non-greedy method we use is a generalization of that used in the `gzip` compressor, as investigated by Horspool (1995). The idea is as follows. Assume the parsing is up to position i. If the lookahead limit $h = 1$ then, instead of encoding the longest match, (p, ℓ), starting at i, the longest match starting at $i+1$, (p', ℓ') is also considered. If $\ell' > \ell$ then the letter at position i is encoded as a single literal followed by the factor (p', ℓ'). This idea can be generalized so that the lookahead can be extended up to position $i+h$, instead of just $i+1$. Horspool explored this approach on English text, testing the gains possible by looking ahead up to $h = 7$ letters. He found that, on English at least, the LZ77 algorithm improved when looking ahead up to 5 to 6 letters, after which no further gains were made.

Longest factor in a region. A slight variation to the simple lookahead by h algorithm is to continue the lookahead until the longest factor within a region is found, and encode the section before the longest factor as a series of shorter factors followed by the longest factor. The longest factor within a region is defined as a factor of position p' and length l', starting from position i' in the current sequence, where no other factor in between positions i and i' has a length $\geq l'$, and no ther factor in between positions i' until $i' + l'$ has a length $\geq l'$.

The algorithm operates in a very similar manner to that of the *lookahead by h* algorithm. The main difference is that instead of the lookahead amount being a constant h, the amount varies depending on the longest factor found so far. First, the longest factor at position i is found of position p and length l. To ensure that this is indeed the longest factor, we set $h = l$ and lookahead by up to l positions to see if there is a longer factor. If there is a longer factor at i' with position p' and length l', then we now set $h = l'$ and continue to see if there's a factor that has a length above l'. This process continues until no factor that is longer than the current longest factor can be found. In the meantime, each longest factor found so far is kept in an array so that when the actual longest factor is found, the series of shorter factors leading up to the actual longest factor can also be encoded. This algorithm essentially finds the local maximum in terms of the factor length.

Efficient non-greedy parsing algorithm. We have described an efficient algorithm for *greedy* RLZ parsing which runs in $O(N)$ time using the suffix tree of T^1 or in $O(N \log n)$ time using the (more compact) suffix array data structure (Kuruppu et al. 2010). It is possible to implement the non-greedy approaches in the same time bounds. In fact, it is possible to compute the factor information for *every* position in the input in $O(N)$ time using a suffix tree. We apply an algorithm for computing so-called *matching statistics*, due to Chang & Lawler (1994) (see also Gusfield (1997), Abouelhoda et al. (2004), and Maaß (2006)). Formally, the matching statistics of string T w.r.t. string S is a table of pairs (p_j, ℓ_j), where $0 \leq j < |T|$ such that:

(a) $T[j..j+\ell_j]$ is the longest prefix of $T[j..|T|]$ which occurs as a substring of S, and

(b) $T[j..j+\ell_j] = S[p_j..p_j+\ell_j]$.

There may be more than one such p_j, and it does not matter which is chosen. Continuing our earlier example, for strings $S = tcttctct$ and $T = ttctgttc$ the matching statistics of T w.r.t. S are given in the following table:

j	0	1	2	3	4	5	6	7
(p_j, ℓ_j)	(2,4)	(0,3)	(1,2)	(0,1)	(0,0)	(2,3)	(0,2)	(1,1)

At position $j = 0$ in T, the longest prefix of $T[0..|T|]$ matching a substring in S is the substring $S[2..6] = ttct$ of length 4 so it is encoded as $(2, 4)$. At position $j = 1$ in T, the longest prefix of $T[1..|T|]$ matching a substring S is the substring $S[0..3] = tct$ of length 3 which is encoded as $(0, 3)$. A special case occurs at position 4 where no prefix of $T[4..|T|] = gttc$ occurs in S. Therefore, we encode it using the start position 0 and a length of 0 to indicate that there was no match.

Clearly any LZ parsing (greedy or non-greedy) of T relative to S can be derived from a subsequence of the matching statistics of T w.r.t. S. To our knowledge, the link between relative Lempel-Ziv parsing and matching statistics computation has never been made in the literature — the two methods appear to have been studied in quite different contexts (classification and approximate pattern matching respectively).

Short factor encoding. The non-greedy parsing algorithm described above is likely to create short factors which can be encoded in a more efficient manner than as position and length pairs. For example, if a factor represents a single nucleotide, then it is cheaper to encode it using 2 bits per nucleotide (spelling out the literal A, C, G or T symbol) rather than using the standard `RLZ` encoding of $\log n + \ell \bmod M + \log M + 1$ bits, where M is the divisor of the Golomb code. With this in mind, we define a *short factor* as any factor which has a literal encoding smaller than its position and length pair encoding.

To encode short factors, first we use a 0 bit to indicate that a short factor is about to be encoded followed by the Golomb encoded length of the short factor (in bases). Then we simply encode each nucleotide from the standard DNA alphabet as 2 bits per base. Any non-short factor is prefixed with a 1 bit to indicate to the decompressor to decode it as position and length pair. We use $M = 8$ for Golomb encoding simply because the datasets used for our experiments show that most short factors have an average length of 12. However, this can be adjusted depending on dataset properties. Finally, we remark that the utility of an efficient short-factor encoding is not limited to non-greedy parsing: short factors are also produced during greedy parsing.

5 Reducing space for position values

One unfortunate aspect of RLZ is the large size of the position components. Because matches are allowed anywhere in the reference sequence, each p_i value can range between 0 and $n-1$ and so $\log n$ bits of storage for each seem to be necessary. That is, although RLZ was developed as a method for coding one sequence relative to another similar sequence, in practice we allowed each factor to come from any location in the reference, so the ordering of factors was not leveraged at encoding time. We now describe a way to reduce the space of the p_i values, exploiting a property particular to the problem of compressing related genomes.

While inspecting the RLZ factors we noticed the factor sequence consisted, with few exceptions, of alternating short and long factors. Moreover, the p component of the ith long factor always seemed to be less than the p component of the $(i+1)$th long factor, forming a long subsequence of increasing p components in the factor sequence. An example of this behaviour for the *S. cerevisiae* genome 273614N is as follows:

```
  ...
  10030697  10
* 16287     23
  10086342  13
  8689589   13
* 16336     48
  3831041   11
* 16395     28
  9166835   12
  11588317  13
* 16448     84
  787019    13
  ...
```

In this table, the left-hand values are the position in the reference sequence and the right-hand values are the factor length. The factors marked with * are long factors (relative to the remaining shorter factors) and their position (left-hand) values form an increasing subsequence. On closer inspection, we found that the *longest increasing subsequence* (LISS) was comprised of roughly half the factors, and these factors tended to be long, as Figure 1 illustrates this distribution. These factors can be identified in $O(z \log z)$ time using the algorithm by Schensted (1961). From here on we identify the factors participating in the LISS as *LISS factors* and the remaining factors as *interleaving factors*.

Such long LISSs are present in the RLZ factors because of the close genetic relationship of the sequence being compressed to the reference sequence. The dataset contains sequences that are evolutionarily related to the reference sequence but with mutations, insertions, deltions and rearrangements scattered across the sequence. When a sequence is factorised relative to the reference, similarity is captured by very long factors that form an alignment between the two sequences. The shorter factors in between correspond to areas of genetic mutation and rearrangment, which characterize the difference between individuals of the same species. Indeed, one of

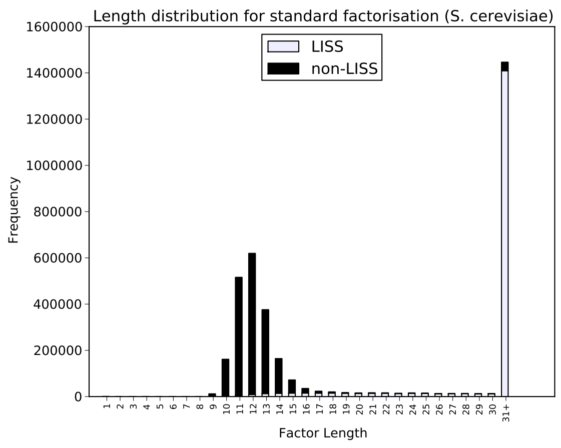


Figure 1: The factor length distribution for *S. cerevisiae* when the standard factorisation algorithm is used.

the most common mutations is replacement of a single nucleotide, thus converting, for example, one long factor into two shorter factors separated by a single base.

The presence of such LISS in the RLZ factors suggests position components (of factors that participate in the LISS) can be encoded more efficiently than $\log n$ bits each, by encoding differences, rather than absolute positions. More precisely, we store a bit vector of z bits total, where the ith bit is set to 1 if and only if the factor is an LISS factor and is 0 otherwise. Each interleaving factor is encoded in the standard manner. The first LISS factor is also encoded in the standard way, using $\log n$ bits per position and Golomb encoding the length. The remaining LISS factor positions are encoded by Golomb encoding the difference between the current position and the previous LISS factor position, and the lengths are Golomb encoded.

We found encoding differences between LISS factor positions as above led to a significant boost in compression performance. However, a futher saving is possible. Let factor (p_i, ℓ_i) be a LISS factor, (p_{i+1}, ℓ_{i+1}) be an interleaving factor, and (p_{i+2}, ℓ_{i+2}) be another LISS factor. Then, given the likelihood that the interleaving factors encode mutations, factor $i+2$ will begin at or close to $p_i + l_i + l_{i+1}$. In other words, after encoding a mutation, the algorithm will go back to detecting factors from the position where the reference and current sequence align. This position can be predicted from the last position where the two sequences aligned and the length of the mutation. Using this observation, it is unnecessary to encode differences since the positions can be well predicted from the cumulative length of the last LISS factor and the lengths of the interleaving factors.

With this in mind, we use the following technique to encode the positions for LISS factors. The first LISS factor is still encoded in the same manner as an interleaving factor. Any subsequent LISS factor position is encoded by first checking if the factor position is exactly at the predicted position using the previous LISS factor position and the cumulative length since the previous LISS factor position. If this condition is satisfied then a 0 bit is output. Otherwise, a 1 bit is output and a further 0 or 1 bit is output depending on if the actual position is to the left or to the right of the predicted position, respectively. Then the difference between the actual and expected position is Golomb encoded.

6 Results

We use three datasets to analyse the compression performance of the new techniques. The first two datasets are two different species of yeast; *S. cerevisiae* with 39 genomes and *S. paradoxus* with 36 genomes.[2] The third dataset is 33 strains of *E. coli* sequences.[3]

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.

First, we discuss the compression performance for the various combinations of factorisation and encoding algorithm. There are four main combinations:

- Lookahead factorisation (`lookahead`)
- Lookahead factorisation with short-factor encoding (`lookahead+shortfac`)
- Lookahead factorisation with LISS encoding (`lookahead+liss`)
- Lookahead factorisation with LISS encoding and short-factor encoding (`lookahead+liss+shortfac`)

For the lookahead factorisation algorithm, we experiment with lookahead limits ranging from 0 to 30 followed by the longest factorisation algorithm, described in Section 4. A lookahead limit of zero equates to the standard greedy factorisation algorithm.

Figure 2 shows the compression performance (in Mbytes) for the various algorithmic combinations. The baseline standard algorithm is indicated with a horizontal dashed line. Unsurprisingly, for all datasets, using lookahead with the standard (p, ℓ) factor encoding leads to worse compression. Using the `lookahead+shortfac` combination encodes short factors efficiently and improves compression. The `lookahead+liss` combination also reduces the compressed size for the yeast datasets but not for the *E. coli* dataset. While LISS encoding is sometimes able to reduce compressed size, it does not encode shorter factors efficiently. For all three datasets, the combination of lookahead factorisation along with LISS encoding and short-factor encoding `lookahead+liss+shortfac`, provides clearly superior compression to the other methods.

To further analyse the LISS encoding, we use the factor length distribution for the *S. cerevisiae* dataset. Figure 1 shows the length distribution of factors when the standard factorisation algorithm is used. Each bar shows the split between LISS and non-LISS factors for a given factor length. Most short factors have a length ranging from 10–15 while the majority of LISS factors have length at least 31. This is why, even when using the standard greedy factorisation, it is beneficial to use specialized short-factor encoding.

Figure 3 shows the length distribution of factors when the longest factorisation algorithm is used. There is a remarkable difference between this distribution and that in Figure 1 (produced by the greedy parsing). When a lookahead factorisation algorithm is used, a large proportion of the interleaving (non-LISS) factors end up having a length of one. An example of this behaviour for the *S. cerevisiae* genome `2736147N` is as follows:

[2]`ftp://ftp.sanger.ac.uk/pub/dmc/yeast/latest`

[3]`ftp://ftp.ensemblgenomes.org/pub/bacteria/release-5/fasta/`

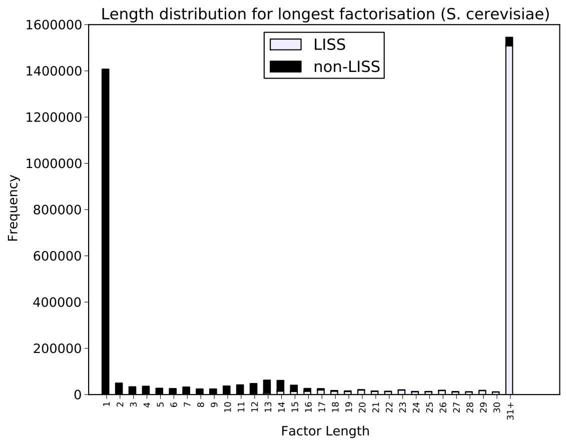


Figure 3: The factor length distribution for *S. cerevisiae* when the longest factorisation algorithm is used.

```
...
* 25359 203
2927486 1
* 25563 58
2533639 1
* 25622 97
4585768 1
* 25720 156
11302741 1
* 25877 230
...
```

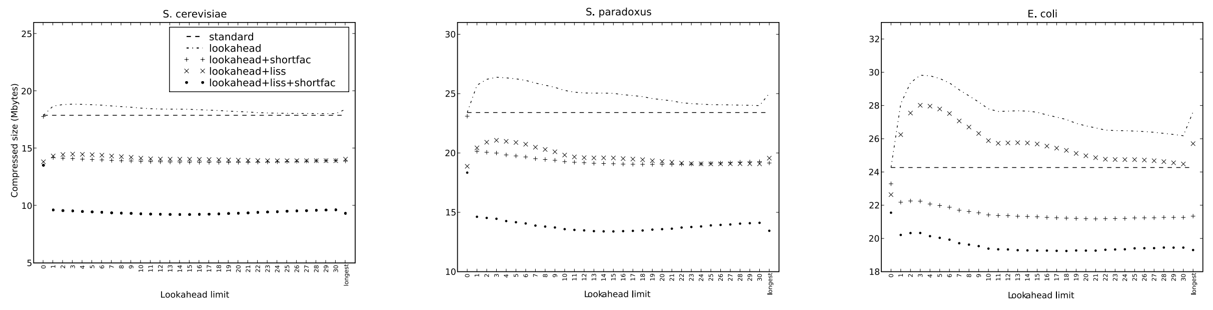
A LISS factors is frequently followed by a non-LISS factor of length one. We hypothesise that these single-symbol factors are single nucleotide polymorphisms (SNPs), or point mutations. When the longest factor at a given position is found and it is an alignment to the reference sequence, then the alignment stops when a nucleotide is reached that is not shared between the sequence and its reference. Without the lookahead algorithm, a relatively short factor is found from the next position to be encoded and then the aligning continues. With the lookahead algorithm, by looking ahead by just one position, a single point mutation is skipped and the alignment to the reference can continue just by encoding the mutation as a factor of length one. We plan to more closely analyse this SNP hypothesis in the future.

Figure 4 and Figure 5 show, respectively, compression and decompression[4] times for a range of lookahead limits. Both LISS encoding and short-factor encoding was enabled for this experiment. In general, compression is faster when lookahead is used. This is explained by the use of the efficient parsing algorithm described in Section 4. The longest factor can be found in $O(N)$ time (or $O(N \log n)$ time using a suffix array) on average, and a longer region of the sequence is covered each time a longer factor is found by looking ahead. However, when the longest factorisation algorithm is used, the compression is slower. In order to find the longest factor within a region, a lot more comparisons are required as the lookahead may happen up to hundreds to thousands of positions.

Decompression time is more variable but always

[4]The compression time only includes the time to generate and encode the factors and does not include the time required to generate the suffix array and its other associated data structures, nor the time to compress the reference sequence. These times are included in the results presented later in Table 1. Similarly, the decompression time only includes the time to decode the factors and does not include the time taken to decode the reference sequence. These times are also included in Table 1.

Figure 2: The variation in Compressed size (in Mbytes) of the *S. cerevisiae*, *S. paradoxus*, and *E. coli* datasets for changes in the lookahead limit, using the four combinations of encoding techniques.



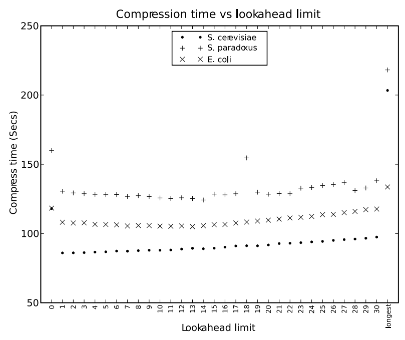


Figure 4: The variation in the time taken to compress the data (in seconds) when the lookahead limit is varied for *S. cerevisiae*, *S. paradoxus* and *E. coli* datasets. The time only includes the time taken to discover the factors and encode them.

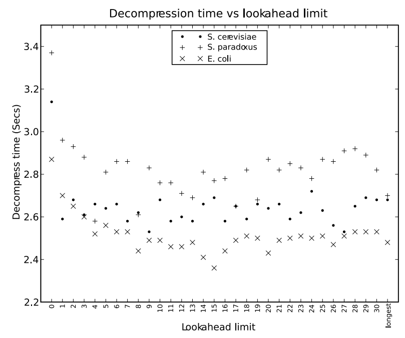


Figure 5: The variation in the time taken to decompress the data (in seconds) when the lookahead limit is varied for *S. cerevisiae*, *S. paradoxus* and *E. coli* datasets. The time only includes the time taken to decode the factors.

faster when some amount of lookahead is performed. The main reason for this is reduced cache misses. The most expensive operation during decompression is the access to the reference sequence that is required to copy the symbols for (that is, decode) each factor. With lookahead, we increase the chance that a factor will be considered short, and encoded as a literal — decoding literals requires no access to the reference. Moreover, lookahead produces longer non-short factors, which increase the number of symbols decoded per access to the reference, and also reduces cache misses.

Figure 6 illustrates the tradeoff between space and throughput achieved by the variants. The `lookahead+liss+shortfac` combination provides the best results across all datasets tested.

Finally, we directly compare the results of the standard RLZ algorithm (`RLZ-std`) to the results of the optimized RLZ algorithm (`RLZ-opt`) (`lookahead+liss+shortfac`) in Table 1. For this, we used the two yeast datasets and a dataset of four human genomes, consisting of the reference human genome[5] , the Craig Venter genome[6], the Korean genome[7] and the Chinese genome[8]. We also compare the optimized RLZ algorithm to three other DNA compression algorithms. Comrad (Kuruppu et al. 2009) is a dictionary compression algorithm that is also suitable for compressing datasets of DNA sequences from the same or similar species. RLCSA (Mäkinen et al. 2010) is one of the self-index implementations that supports queries such as *display()*, *count()* and *locate()* (we turned off support for *count()* and *locate()* for this experiment to make the algorithms comparable). Finally, `XM` (Cao et al. 2007) is a statistical DNA compressor that is also able to compress DNA sequences with respect to a reference sequence and is the DNA compression algorithm with the best compression results that we known of.

For the *S. cerevisiae* dataset, prior to `RLZ-opt`, Comrad had the best compression results. However, `RLZ-opt` was able to compress the dataset to a lower size of 0.15 bpb and this is almost half of the compressed size achieved by `RLZ-std`. For the *S. paradoxus* results, `XM` had the best results compared to `RLZ-std`, but `RLZ-opt` was able to achieve an equivalent result to `XM`. For the *H. sapien* results, the non-RLZ algorithms were not able to compress the dataset very well. `RLZ-opt` was also unable to achieve a much better result compared to `RLZ-std`. However, most of the 753.90 Mbytes with `RLZ-std` (or the 707.15 Mbytes with `RLZ-opt`) consists of the compressed reference sequence, which has a size of 645.34 Mbytes. The compressed size of the three other genomes was 111.56 Mbytes (0.10 bpb) using `RLZ-std` and 64.81 Mbytes (0.06 bpb) using `RLZ-opt`, almost a halving of the compressed size. The overall compressed result would also improve when more genomes are available to be added to the dataset. The compression and decompression times for RLZ are much lower compared to the other algorithms. `RLZ-opt` takes slightly longer to compress compared to `RLZ-std` but `RLZ-opt` is faster to decompress as was discussed previously.

As a note on memory usage, in order to support the lookahead functionality, memory usage is three times that of the usage when standard factorisation is used, but this is still a small proportion of overall collection size. When standard factorisation is

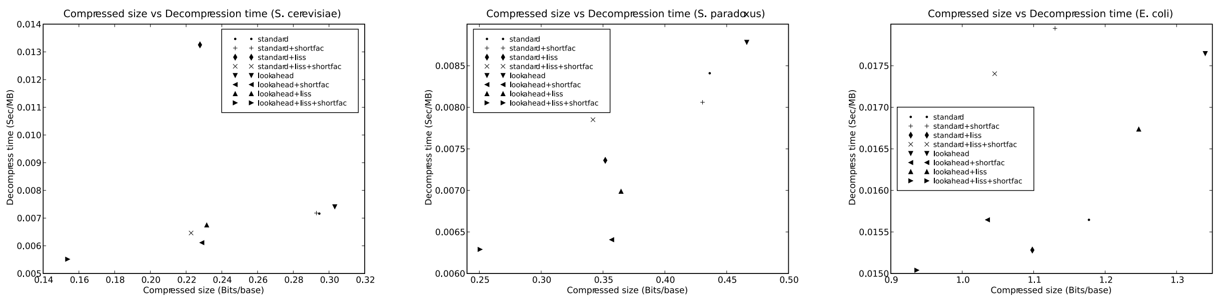
[5]ftp://ftp.ncbi.nih.gov/genomes/H_sapiens/Assembled_chromosomes/hs_ref_GRC37_chr*.fa.gz

[6]ftp://ftp.ncbi.nih.gov/genomes/H_sapiens/Assembled_chromosomes/hs_alt_HuRef_chr*.fa.gz

[7]ftp://ftp.kobic.kr/pub/KOBIC-KoreanGenome/fasta/chromosome_*.fa.gz

[8]ftp://public.genomics.org.cn/BGI/yanhuang/fa/chr*.fa.gz

Figure 6: Space-decompression time tradeoff for the `RLZ` variants.



Dataset	S. cerevisiae				S. paradoxus				H. sapien			
	Size (Mbytes)	Ent. (bpb)	Comp. (sec)	Dec. (sec)	Size (Mbytes)	Ent. (bpb)	Comp. (sec)	Dec. (sec)	Size (Mbytes)	Ent. (bpb)	Comp. (sec)	Dec. (sec)
Original	485.87	2.18	—	—	429.27	2.12	—	—	11831.71	2.18	—	—
RLCSA	41.39	0.57	781	312	47.35	0.88	740	295	3834.82	2.54	34525	14538
Comrad	15.29	0.25	1070	45	18.33	0.34	1068	50	2176.00	1.44	28442	1666
`XM`	74.53	1.26	18990	17926	13.17	**0.25**	30580	28920	—	—	—	—
RLZ-std	17.89	0.29	**143**	9	23.38	0.44	**182**	**6**	753.90	0.51	**15451**	573
RLZ-opt	**9.33**	**0.15**	233	**8**	**13.44**	**0.25**	241	**6**	**707.15**	**0.48**	17861	**526**

Table 1: Compression results for two repetitive yeast collections and a set of four human genomes. The first row is the original size for all datasets (size in megabases), the remaining rows are the compression performance of RLCSA, Comrad and XM algorithms followed by the standard `RLZ` algorithm and the improved `RLZ` algorithm using `lookahead+liss+shortfac` algorithms. The four columns per dataset show the size in Mbytes, the 0-order entropy (in bits per base), time taken to compress (in seconds) and time taken to decompress (in seconds), respectively. The time taken to compress includes the time taken to generate the suffix arrays and other associated data structures, and the time to compress the reference sequence. The time taken to decompress includes the time taken to decompress the reference sequence.

used, the memory usage for *S. cerevisiae*, *S. paradoxus* and *E. coli* are 45.78 Mbyte, 43.58 Mbyte and 17.22 Mbyte, respectively. With lookahead factorisation, 118.75 Mbyte, 113.0 Mbyte and 43.89 Mbyte, respectively. This is due to the increased space ($2n \log n$ bits) required to store the inverse suffix array SA^{-1} (satisfying the property $SA^{-1}[SA[i]] = i$) and longest common prefix (LCP) array SA_{LCP} ($SA_{LCP}[i]$ contains the length of the longest common prefix between $SA[i-1]$ and $SA[i]$) to implement a variation of the efficient non-greedy parsing algorithm described in Section 4.

7 Concluding Remarks

As described in our previous work (Kuruppu et al. 2010) `RLZ` allows fast access to arbitrary parts of the collection with a very slight space overhead. For the non-greedy parsings we have considered in this paper, fast access can be achieved by applying the same data structures. For the approach which separates long and short factors (using the LISS), more care is required to acheive random access as the position components of the long factors are differentially encoded. The key idea to enabling random access is to store absolute position values periodically in the sequence of differences.

Our next aim is to augment the `RLZ` representation with succinct data structures that allow fast indexed search over the compressed collection. We believe this functionality can be added to our approach, using yet unpublished techniques, not dissimilar to earlier work (Navarro 2004, 2008).

Finally, while we were able to drastically improve compression via the presence of a LISS in the factors, the best way to exploit this phenomenon remains unclear and warrants further analysis. Use of the LISS turns `RLZ` from a one pass to a two pass algorithm. An online LISS finding algorithm, such as that of Liben-Nowell et al. (2006), may be effective in our setting as the LISS is long (roughly half the sequence) and tends to manifest early in the factor set.

With the cost of acquiring a single human genome falling below $10,000, the volume of genomic data will grow dramatically; with our methods, collections of related genomes can be stored with great efficiency. Approaches such as ours are key to management of the volumes of biological data now being produced.

References

Abouelhoda, M. I., Kurtz, S. & Ohlebusch, E. (2004), 'Replacing suffix trees with enhanced suffix arrays', *Journal of Discrete Algorithms* **2**(1), 53–86.

Apostolico, A. & Lonardi, S. (2000), Compression of biological sequences by greedy off-line textual substitution, *in* 'DCC '00: Proceedings of the Conference on Data Compression', pp. 143–152.

Behzadi, B. & Fessant, F. L. (2005), 'DNA compression challenge revisited: A dynamic programming approach', *Symposium on Combinatorial Pattern Matching* **3537**, 190–200.

Brandon, M., Wallace, D. & Baldi, P. (2009), 'Data structures and compression algorithms for genomic sequence data', *Bioinformatics* **25**(14), 1731–1738.

Cao, M. D., Dix, T., Allison, L. & Mears, C. (2007), A simple statistical algorithm for biological sequence compression, *in* 'DCC '07: Proceedings of the Conference on Data Compression', pp. 43–52.

Chang, W. I. & Lawler, E. L. (1994), 'Sublinear approximate string matching and biological applications', *Algorithmica* **12**(4–5), 327–344.

Chen, X., Kwong, S. & Li, M. (2000), A compression algorithm for DNA sequences and its applications in genome comparison, *in* 'RECOMB '00: Proceedings of the fourth annual international conference on Computational molecular biology', ACM, p. 107.

Chen, X., Li, M., Ma, B. & Tromp, J. (2002), 'DNACompress: fast and effective DNA sequence compression', *Bioinformatics* **18**(12), 1696–1698.

Christley, S., Lu, Y., Li, C. & Xie, X. (2009), 'Human genomes as email attachments', *Bioinformatics* **25**(2), 274–275.

Ferragina, P., Nitto, I. & Venturini, R. (2009), On the bit-complexity of Lempel-Ziv compression, *in* 'SODA '09: Proceedings of the twentieth Annual ACM-SIAM Symposium on Discrete Algorithms', Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, pp. 768–777.

Grumbach, S. & Tahi, F. (1993), Compression of DNA sequences, *in* 'DCC '93: Proceedings of the Conference on Data Compression', pp. 340–350.

Gusfield, D. (1997), *Algorithms on Strings, Trees, and Sequences : Computer Science and Computational Biology*, Cambridge University Press, Cambridge, UK.

Haussler, D. et al. (2009), 'Genome 10k: A proposal to obtain whole-genome sequence for 10 000 vertebrate species', *Journal of Heredity* **100**(6), esp086–674.

Horspool, R. N. (1995), The effect of non-greedy parsing in Ziv-Lempel compression methods, *in* 'DCC '95: Proceedings of the Conference on Data Compression', IEEE Computer Society, Washington, DC, USA, p. 302.

Korodi, G. & Tabus, I. (2005), 'An efficient normalized maximum likelihood algorithm for DNA sequence compression', *ACM Trans. Inf. Syst.* **23**(1), 3–34.

Korodi, G. & Tabus, I. (2007), Normalized maximum likelihood model of order-1 for the compression of DNA sequences, *in* 'Data Compression Conference, 2007. DCC '07', pp. 33–42.

Kuruppu, S., Beresford-Smith, B., Conway, T. & Zobel, J. (2009), 'Repetition-based compression of large DNA datasets', Poster at: 13th Annual International Conference on Research in Computational Molecular Biology (RECOMB09).

Kuruppu, S., Puglisi, S. J. & Zobel, J. (2010), Relative Lempel-Ziv compression of genomes for large-scale storage and retrieval, *in* 'Proc. 17th International Symposium on String Processing and Information Retrieval (SPIRE)', LNCS, Springer. To appear.

Liben-Nowell, D., Vee, E. & Zhu, A. (2006), 'Finding longest increasing and common subsequences in streaming data', *Journal of Combinatorial Optimization* **11**(2), 155–175.

Maaß, M. G. (2006), 'Matching statistics: efficient computation and a new practical algorithm for the multiple common substring problem', *Software, Practice and Experience* **36**(3), 305–331.

Mäkinen, V., Navarro, G., Sirén, J. & Välimäki, N. (2010), 'Storage and retrieval of highly repetitive sequence collections', *Journal of Computational Biology* **17**(3), 281–308.

Matsumoto, T., Sadakane, K. & Imai, H. (2000), 'Biological sequence compression algorithms', *Genome Informatics* **11**, 43–52.

Navarro, G. (2004), 'Indexing text using the Lempel-Ziv trie', *Journal of Discrete Algorithms* **2**(1), 87–114.

Navarro, G. (2008), 'Indexing LZ77: the next step in self-indexing', Keynote talk at 3rd Workshop on Compression, Text and Algorithms. Slides: http://spire2008.csse.unimelb.edu.au/talks/gn08-wcta.pdf.

Rivals, E., Delahaye, J., Dauchet, M. & Delgrange, O. (1996), A guaranteed compression scheme for repetitive DNA sequences, *in* 'DCC '96: Proceedings of the Conference on Data Compression', p. 453.

Schensted, C. (1961), 'Longest increasing and decreasing subsequences', *Canad. J. Math* (13).

Ziv, J. & Lempel, A. (1977), 'A universal algorithm for sequential data compression', *IEEE Transactions on Information Theory* **23**(3), 337–343.