BIOINFORMATICS ORIGINAL PAPER Vol. 27 no. 21 2011, pages 2979–2986
doi:10.1093/bioinformatics/btr505

Sequence analysis

Advance Access publication September 5, 2011

Robust relative compression of genomes with random access

Sebastian Deorowicz[1,*] and Szymon Grabowski[2]

[1]Institute of Informatics, Silesian University of Technology, 44-100 Gliwice and [2]Department of Computer Engineering, Technical University of Łódź, 90-924 Łódź, Poland

Associate Editor: John Quackenbush

ABSTRACT

Motivation: Storing, transferring and maintaining genomic databases becomes a major challenge because of the rapid technology progress in DNA sequencing and correspondingly growing pace at which the sequencing data are being produced. Efficient compression, with support for extraction of arbitrary snippets of any sequence, is the key to maintaining those huge amounts of data.

Results: We present an LZ77-style compression scheme for relative compression of multiple genomes of the same species. While the solution bears similarity to known algorithms, it offers significantly higher compression ratios at compression speed over an order of magnitude greater. In particular, 69 differentially encoded human genomes are compressed over 400 times at fast compression, or even 1000 times at slower compression (the reference genome itself needs much more space). Adding fast random access to text snippets decreases the ratio to ∼300.

Availability: GDC is available at http://sun.aei.polsl.pl/gdc.

Contact: sebastian.deorowicz@polsl.pl

Supplementary Information: Supplementary data are available at *Bioinformatics* online.

Received on May 17, 2011; revised on August 17, 2011; accepted on August 31, 2011

1 INTRODUCTION

Rapid development in DNA sequencing technologies led to drastic growth of data publicly available in sequence databases, for example, GenBank at NCBI or 1000 Genomes project. The low cost of acquiring an individual human genome opens the door to 'personalized medicine'.

DNA sequences within the same species are both large and highly repetitive. For example, only ∼0.1% of the 3 GB human genome is specific; the rest is common to all humans. This poses interesting challenges for efficient storage and fast access to those data. Most classic data compression techniques fail to recognize this tremendous redundancy, simply because finding matches with e.g. an LZ77 variant with a sliding window would require a multi-gigabyte buffer, not counting the match-finding structures. On the other hand, using a context-based statistical coding (for example, PPM) may require maintaining a high-order statistical model, otherwise the context statistics will be polluted with 'accidental' DNA matches. Using such a model is problematic, due to its enormous memory requirements.

*To whom correspondence should be addressed.

Interestingly, most of the DNA-specialized compressors from the literature are inappropriate to handle modern genomic databases. There are a number of reasons for that: (i) most of them care about compression ratio rather than compression and decompression speed or the memory use during the compression process; for these reasons, they are not practical for sequences larger than, say, several megabytes; (ii) most effort has been focused on succinctly representing a single genome (which is believed to be almost incompressible anyway, hence only tiny improvements were at the stake), not to be particularly efficient in detecting inter-genome redundancy; (iii) extracting a range of symbols from the middle of the compressed stream is a rarely supported feature.

Only since around 2009 can we observe a surge of interest in practical, multi-sequence oriented DNA compressors, usually coupled with random access capabilities and sometimes also offering indexed search. The first algorithms from 2009 (Brandon *et al.*, 2009; Christley *et al.*, 2009) were soon followed by more mature proposals, which will be presented below.

Mäkinen *et al.* (2010) added index functionalities to compressed DNA sequences: *display* (which can also be called the random access functionality) returning the substring specified by its start and end position, *count* telling the number of times the given pattern occurs in the text and *locate* listing the positions of the pattern in the text. The authors noticed that the existing general solutions, paying no attention to long repeats in the input, are not very effective here, and they proposed novel *self-indexes* for the problem.

Other full-text indexes for repetitive sequences were proposed in Claude *et al.* (2010) and Kreft and Navarro (2011). The former work presents two schemes, one using an inverted index on q-grams, tailored to the repetitive nature of the input data, and the other being a grammar-based index. The latter paper introduced a self-index based on the LZ-End (Kreft and Navarro, 2010), which is an algorithm interesting in its own, as it is an LZ77 variant enabling efficient extraction of arbitrary phrases.

Wang and Zhang (2011) presented a scheme for referential compression of genomes, working on the chromosome level. If the pair of respective chromosomes is similar enough, one of them is encoded with respect to the chromosome from the reference sequence. If not, it is split into several pieces for which the best alignments are then found in the reference chromosome. Then, longest common subsequences between matching parts are found and the differences encoded using Huffman coding.

Kuruppu *et al.* (2010) proposed a simple yet quite efficient compression scheme with random access (it is not an index though), dubbed RLZ-std. They choose one of the sequences as the reference sequence and compress it with a self-index (in the cited work, however, a general-purpose compressor, 7z, with no random access

capabilities, was used). The other sequences are greedily parsed to find longest matches to the reference sequence. Some extra data structures are added to provide random access. Unfortunately, the results presented in the cited work are incoherent (which is not clearly admitted in the paper): the given compression ratios refer to sole archives, with no random access support, while the display timings are obtained with partially decompressed (in RAM) data structures, so the actual memory use during query handling is by 50–75% greater than the size of the archive plus the random access helper structure.

The work (Kuruppu *et al.*, 2011b) is a follow-up paper, presenting a stronger LZ77-style algorithm. This time, however, no attempt is made for enabling random access, although it is mentioned how it can be (relatively easily) added to the scheme. As their algorithm is a departure point of our proposal, we describe it more extensively in the next section.

In this work, we propose another algorithm for effective compression of multiple genomic (DNA) sequences of the same species and show experimental results, suggesting its supremacy over existing solutions. Although the general framework of relative LZ77-style compression is not new in this context, we propose some novel ideas. Our scheme provides fast random access to compressed data, exploiting some space–time trade-offs.

Finally, we note a very recent work from Kuruppu *et al.* (2011c), where they cite our preliminary preprint of this article (Grabowski and Deorowicz, 2011) and focus on the aspect of reference sequence construction. Three dictionary compression algorithms are tested in order to generate relevant reference sequences, and the most successful of them leads to compression ratios comparable to our 'advanced' variant (see Section 3.2) on yeast datasets. Alas, the reference sequence build algorithms are slow and, more importantly, require huge amounts of memory, which makes running them on multiple human genomes problematic.

2 METHODS AND ALGORITHMS

2.1 Differential LZ77 compression

Repetitive data are naturally well handled by compressors from the LZ77 family. The feature common to all those algorithms is to parse the input data into a sequence of matches and literals, usually entropy-encoded, e.g. with Huffman coding. An LZ-match (also called a factor) is a reference to an earlier occurrence of the same subsequence, expressed as the distance (offset) to that earlier subsequence and its length. If, at the current position, there is no (satisfactory) match, a literal is emitted and the compressor moves forward by one symbol. We note that greedy parsing, that is, always choosing the longest match, is usually a suboptimal strategy. Parsing the input into matches and literals is a vital factor for good compression performance, and the problem of optimal parsing is solved only under a simplified assumption of known cost functions for encoding match offsets and match lengths (Ferragina *et al.*, 2009).

RLZ-opt (Kuruppu *et al.*, 2011b) is the current state-of-the-art algorithm for compressing collections of genomes (As mentioned in Section 1, the most recent variants from the same team (Kuruppu *et al.*, 2011c) are significantly better in compression ratio, and also impractically slow in compression and memory-consuming. From those reasons, we do not discuss those variants till the experimental section.). It follows the LZ77 route, but has some features not often met in that family of compressors, typically applied to other kinds of data. First, it is not just used, but *designed* for genomic collections, where random access to an individual sequence (even better, only a small snippet of it) is a welcome feature. For this reason, one of the sequences in the input collection is chosen (and encoded) as the reference sequence, while all the other sequences are encoded with relation to the first one, but without any cross-references to one another. Second, matches are found thanks to a suffix array, which is unusual, since most LZ77 compressors make use of a hash array (or, more rarely, a search tree). Building a suffix array is relatively slow, but this structure facilitates effective non-greedy parsing.

The RLZ-opt algorithm is based on several principles. A lookahead buffer is used for each considered location in the text, which basically means that if the match at position $i+1$ is longer than the match at position i, a literal may be emitted at position i followed by a match. This idea is however extended, not to a fixed-size buffer, but to a buffer whose size changes dynamically, depending on the length of the currently longest match found (details can be found in the cited work). They also used the idea from Manzini and Rastero (2004) of encoding short matches as runs of literals, which gives a fair boost in compression ratio. As a last thing, they notice that long and short (that is, those encoded as literals) matches tend to appear alternatively, and the offset of the long match can usually be predicted quite well from the offset of the previous long match. Those offsets (match positions) usually form long increasing sequences, hence an algorithm for solving the classic longest increasing subsequence (LISS) problem (Gusfield, 1997) is used to detect those matches (called LISS factors), whose offsets are then cheaply encoded. The LISS factors are often followed by single-symbol factors which represent single nucleotide polymorphisms (SNPs) in DNA. We note that prediction of the next factor position is another incarnation of the implicit approximate repeat detection idea, again known from Manzini and Rastero (2004). The parsing products in RLZ-opt, in particular, the match lengths and run-of-literals lengths, are compacted with Golomb encoding.

2.2 Our algorithm at first sight

As said, our algorithm is essentially similar to RLZ-opt (Kuruppu *et al.*, 2011b), and the main differences are as follows:

(1) In addition to the reference sequence, we use extra reference phrases from the other sequences for which matches exist.
(2) Our LZ-parsing is different (details later).
(3) Our LZ match-finding procedure is based on hashing rather than a suffix array, with great benefits for compression speed, and also helping to reduce memory requirements during the compression.
(4) Huffman coding rather than Golomb coding is used in representation of various statistics data.
(5) Compression is performed in blocks (with shared Huffman models), to facilitate random access.
(6) Our (compressed) reference sequence admits random access.
(7) We have found a fast and reliable heuristic to select an appropriate reference sequence.
(8) Optionally we use more than one reference sequence, which can improve the compression significantly, for the price of giving up the random access.

First we describe the scheme from the compression viewpoint, and then discuss how we implemented the random access functionality. Also, we implemented a variant with multiple reference sequences, but this time only for decompressing the whole sequence collection (with increasing the number of reference sequences, the random access would be slower and slower, and also quite cumbersome to implement).

Improved compression ratios of our algorithms can be attributed mostly to the chosen LZ-parsing, which aggressively looks for a certain class of approximate matches, and Huffman encoding, which is more compact than Golomb, and also applied to more byproducts of the compression process. When the genomes are not extremely similar (which is the case of the yeast

collections), then also adding extra reference phrases boosts the compression noticeably.

First, we present the algorithm in the basic form, without the extra reference phrases, and then explain this enhancement.

2.3 Basic variant

Let us assume that T^1 is the reference sequence, and T^i for $2 \le i \le r$ are the following sequences that will be encoded relatively to T^1. The reference sequence cannot be compressed as effectively as the other ones and actually constitutes a substantial part of the final archive. We first divide the reference sequence into blocks of size 8192 symbols and store explicitly start positions of each block in the compressed (output) form of this sequence. It is possible that the start positions for i-th and $(i-1)$-th block are equal, which means that the whole block contains only N or n symbols (this phenomenon is quite frequent especially on the available human genomes). Using blocks also enables fast access to data (only local decompression of the reference sequence is needed). Then, within each block except for those N-only blocks, we divide the reference sequence into q-grams over the actual alphabet. More precisely, if the number of distinct symbols in the given dataset is σ, then q is the largest number such that $\sigma^q \le 4096$ (housing an even larger alphabet of q-grams may be too costly for the representation of the statistical model). The resulting stream is Huffman-encoded, mostly to prevent inefficiencies on middle-sized runs of N (or n) symbols, which are not rare. We also hash overlapping subsequences of length M_1 from the reference sequence (that is, $T^1[1 \ldots M_1]$, $T^1[2 \ldots M_1+1]$, etc.), to enable further match searches.

In the next phase, we process the sequences T^i, $2 \le i \le r$, one by one, scanning them from left to right and looking for matches in the reference sequence. (For clarity of exposition, we assume that the data in T^i sequences, $2 \le i \le r$, are not partitioned into blocks.) Our parsing strategy is non-greedy, but does not mimic the lookahead approach known from Kuruppu *et al.* (2011b). Instead, we make use of the following simple observations. One is that shorter matches are better than moderately longer ones if their offsets are significantly cheaper to encode. The other is that matches with (one or more) single-character mismatches inside are frequent in genomes, and it is worth encoding them efficiently. (We will call those approximate matches, to distinguish from exact matches.) Note that the latter idea, although expressed in different terms, roughly corresponds to predicting the positions for LISS factors in Kuruppu *et al.* (2011b). Approximate matches have a mismatch count limit $k = 29$ (choosing this value was convenient for technical reasons, but extending it to, e.g. 128 does not improve compression ratios noticeably).

We denote the minimum match length of the first (contiguous) piece of an approximate match with M_1 and the minimum match extension, that is, the length of each next piece of an approximate match, with M_2.

LZ-matches are traditionally represented as a pair: reference offset and match length. We encode offsets as differences between the sequence position in the current genome and the matched-to sequence position in the reference genome. This tends to produce relatively small numbers (both negative and non-negative), but the extra step, differential encoding of those values, makes the resulting stream even flatter.

As said, our parsing scheme may prefer a shorter match if its offset encoding is cheaper. The chosen heuristics basically distinguishes between near match offsets (encoded on a single byte) and distant match offsets (encoded on 5 bytes), and sets a penalty for switching from match to match between those kinds of offsets. Also a penalty is inflicted for each mismatch inside a match. Moreover, the penalties for offset kind switching are larger if the average match length is larger in a given file. This prevents from greedy looking for the longest match 'anywhere' on human genomes, where it is more beneficial to find approximate copies of possibly large areas. Of course, those approximate region copies are found implicitly in our scheme, that is, as a sequence of exact or approximate matches, possibly separated by short snippets of mismatches. On the other hand, on less repetitive collections (yeast datasets) finding possibly long matches to diverse regions of the reference sequence is often a good strategy, and hence the penalties for switching offset kinds (and thus their encoding) are less severe.

Handling long runs of N or n symbols deserves special care. We encode runs of N (or n) symbols of length at least M_1 as a pseudo-match (with the run's actual length and an artificial unique offset).

There are four conceptual streams in our solution: match offsets, match lengths, literals (those symbols that do not belong to any match) and match / literal flags. The flags are not binary since they also tell the number of mismatches in a match. In this way, an approximate match is represented by a single offset (but the number of encoded match lengths is equal to the number of its pieces). Variable-length byte coding is used for match offsets and match lengths. The separate byte positions imply separate order-0 Huffman models, which are responsible for the final compression stage. For example, the first byte in a match offset has 161 values for offset differences from -80 to 80, one value telling that the offset delta is less than -80 (followed by 4 extra bytes), one value telling that the offset delta is >80, one value denoting an N-run and one value for signaling a match to a string from the concatenated extra reference phrases (see later). Finally, about 80 values are reserved for switching the reference sequence (used only in the 'ultra' variant of our algorithm, see Section 2.7). Also, long runs of offsets equal to zero are run-length encoded.

Literals are processed like the reference sequence (only without dividing them into blocks), with packing several of them into supersymbols and applying Huffman. Finally, match / literal flags are also packed and submitted to yet another Huffman model. How many literals / match offsets / flags should be clumped together before entering the Huffman coding phase is not fixed but estimated for the given data, according to the length of the appropriate stream. For greater compression efficiency, we apply exclusions for the literals within approximate matches. As an example, let us take a match whose (say, first) mismatching pair is A–C. In our solution, the decompressor 'knows' that all symbols here are possible except for C.

2.4 Extra reference phrases

We have noticed that good LZ-compressors can be very efficient on our data if no restriction to match references is set. Still, unrestricted set of reference positions to LZ-matches prevents random access, since a referenced substring can also be represented as one or more matches to earlier substrings, and the resulting tree of references can grow with hardly any constraint. Some compromise has to be found then. Our solution is to take long enough runs of successive *literals* in T^i, $2 \le i \le r$, and append them to the reference sequence. They act as a reservoir for extra matches. The offset of such a match, as mentioned earlier, has a unique 1-byte prefix, and what follows is the match position from the beginning of the area of extra reference phrases (no delta coding used here). The minimal length of a literal run is $M_3 = 48$. Note that we detect the literal runs on the fly and attach at the end of the extended reference sequence, hence this idea does not require an extra pass over data. In a single pass we cannot be sure, which extra phrases will give a match for some future sequence, so the value of M_3 is chosen quite arbitrarily, but it cannot be too small; we assume that the literal run should be longer than the minimal match length to increase the probability of finding a match to it. It is also possible in an extra pass over the compressed data to remove the unsuccessfully added extra phrases, but due to the additional time we decided not to implement this feature in the current version.

2.5 Random access

An important functionality of our application is extraction of arbitrary contiguous snippets from a pointed sequence and a pointed chromosome. To this end, we divide each chromosome into blocks of approximately equal size, about b_d symbols. The LZ-parsing in the random access supporting compression mode disallows creation of phrases crossing block boundaries.

We consider two space / access time trade-offs, using different variants for encoding the reference sequence symbols: (i) packed into 2 byte words, corresponding to the actual alphabet of the dataset (if, for example, there are five DNA symbols in the dataset, they are packed in sextuples, since $5^6 \le 2^{16} < 5^7$); (ii) packed into supersymbols and then Huffman-compressed.

We start with variant (i). The query asks for a range $[\ell, r]$ from chromosome id_{chr} in collection id_{col}. (We assume that id_{col} is not the reference sequence. Handling the opposite case is easier.) All the chromosomes of a single sequence (chromosome collection) are merged before compression, hence the original query range must be translated (in constant time) into the range $[\ell', r']$ in id_{col}. Then, two binary searches are used to find the first and the last block at least partially covered by the obtained range. Binary search, rather than constant-time lookup, is needed, since the (decompressed) blocks are of varying sizes, only approximately equal, as the LZ-matches are not artificially truncated. All the found blocks are partially decompressed, which includes the streams of flags, match lengths, match offsets and literals. The streams of flags and match lengths are scanned, to skip over the prefix of the first block corresponding to the collection offsets less than ℓ'. Similarly, the last block is processed only until the current position exceeds r'. Any reference to a supersymbol (stored on a 2 byte word) requires decoding it letter-by-letter. Some care must be taken if the start position ℓ' is inside an approximate match.

Handling variant (ii) is similar, but whole Huffman-compressed blocks must be extracted from the reference sequence when there are some LZ-matches in the currently extracted snippet. This makes the procedure significantly slower than in the previous variant. There are two tricks implemented to mitigate this overhead. First, Huffman coding is applied in the reference sequence over equal-size original blocks, of $b_r = 8192$ symbols. Second, several most recently extracted Huffman blocks from the reference sequence are cached (during a query), since successive LZ-matches may reuse them.

2.6 Selecting the reference sequence

The choice of the reference sequence has significant impact on the overall compression results. Of course, if the reference sequence is totally irrelevant (from another species), the compressed results are horrible, but even a wrong choice from the given set of same species genomes leads to a noticeable loss. We found out that choosing the sequence that maximizes the number of (possibly repeating) subsequences of length M_1 (which is set to 13); not containing the symbol N in it works correctly most of the time, in our experiments failing only in case of one human chromosome. We use this procedure in the variant with 'R' suffix (see Section 3.2).

2.7 Multiple reference sequences

An alternative to the idea of extra reference phrases is using two, or more, reference sequences. In our implementation, choosing h reference sequences means that the first h sequences of the dataset become reference ones. The first of them is compressed as in the standard variant. The following ones may have matches to the previously compressed reference sequences, and the match finding is (again) based on hashing. During the compression (and decompression) process, the reference sequences are stored uncompressed in memory.

LZ-parsing is again similar to the standard variant, but the search space is now broadened, since the parser has to also decide constantly which reference sequence it should choose. Our strategy is based on two simple assumptions: (i) matches with small (differential) offsets are cheaper to encode than matches with large (differential) offsets, provided that both come from the same reference sequence; (ii) if both previous and current matches have large (differential) offsets then it is better to use the reference sequence of the previous match. Simply speaking, switching the reference sequence is rarely a good deal and we do it (from a match to a match) sparingly, based on a heuristic. Match encoding is also similar to the variant with a single reference sequence, but the repertoire of flags has been extended; in particular, there are flags to signal a changed (compared to the previous match) reference sequence.

Now we present a theoretical (and not implemented) variant with two reference sequences. Of course, it is crucial to encode the latter sequence with reference to the former one; otherwise on the available datasets, with relatively few sequences, a compression loss would be inevitable. We believe fast random access is possible using the classic rank/select operations. To give a flavor of this idea, let us assume we have two reference sequences and the second, T^2, of length n, is LZ-encoded relative to the first one, and we also assume exact matches for clarity. We create a bit-vector $B[1, n]$, with 1s exactly in the positions where LZ-matches start and just after they end. Now, $rank(B, k)$ is odd iff position k in T^2 is inside some match and then $k - select(B, rank(B, k))$ tells where exactly within this match k is. We also need to perform *select* on the match offsets and lengths, and have some extra structures to handle prefix sums. To reduce the overhead of B, in reality we should use its compressed representation [see, e.g. Claude and Navarro (2008)].

2.8 Implementation details

Our tool, Genome Differential Compressor (GDC), supports all the ambiguity codes (not only the standard N symbol) and can losslessly handle both lowercase and uppercase DNA notation. Also, it handles headers and End-Of-Line (EOL) symbols in the input FASTA data. According to our experience, this is not common in existing implementations; for example, we found out from the authors of Kuruppu *et al.* (2011b) that in their experiments those rare symbols different to A, C, G, T or N were converted to N, and also their software requires eliminating all EOLs before compression. More details concerning those issues are given in the Supplementary Material.

GDC was implemented in C++, using g++ 4.1.2 compiler.

3 RESULTS AND DISCUSSION

We have run experiments to evaluate the performance of our algorithm. In this section, the main results are presented and discussed. Additional results and more details on the used datasets and our test methodology can be found in the Supplementary Material.

3.1 Test methodology

The test machine was a 2.4 GHz Quad-Core AMD Opteron CPU with 64 GB RAM running Red Hat 4.1.2-46, a single core of the CPU was used.

The test collections include several datasets previously used in the literature, and a large collection of human genomes taken from Complete Genomics Inc. The former group comprises two yeast datasets and a dataset of four human genomes, all publicly available and used earlier in Kuruppu *et al.* (2011b), and also two human genome sequences and the plant genomes, *Arabidopsis thaliana* and rice, used in Wang and Zhang (2011); in the latter test, we measured only relative compression on the datasets, for compatibility with the cited work. The largest and probably most interesting collection is, however, a dataset of 70 human genomes totaling almost 220 GB; to our knowledge, no one has yet made genomic compression experiments on such a scale.

We tested our algorithm in several variants. The *fast* one, compared with the *normal* variant, resigns from Huffman encoding of the reference sequence, in order to significantly speed up random access to data. The *advanced* variant includes the idea of extra reference phrases, not used in *normal*. The 'R' suffix means 'with reference sequence finding'. Finally, *ultra* is the variant with multiple reference sequences, boasting the highest compression but with no random access functionality. We stress that in the non-ultra GDC modes are tested for compression only in Tables 1–4 and their archives do not contain extra data facilitating random access. Only in Table 5 (and the Supplementary Material), we explore this issue, presenting obtained space / access time trade-offs. As it will be

Table 1. Compression results for *S.cerevisiae* (39 genomes)

Method	Total		Avg. per seq.		c-sp	d-sp
	Size (MB)	Ratio	Size	Ratio	(MB/s)	(MB/s)
original	493.98	–	–	–	–	–
gzip*	132.00	3.7	3.385	3.7	3.65	91.92
bzip2*	122.66	4.0	3.145	4.0	6.22	16.08
7z*	4.97	99.3	0.052	244.7	0.54	163.03
RLCSA	40.13	12.3	–	–	0.57	1.75
Comrad	16.50	29.9	–	–	0.70	12.05
RLZ-opt*	9.33	52.9	0.167	76.1	1.60	189.99
RLZ-RePair*	6.48	76.2	–	–	0.27	160.91
GDC-fast	6.95	71.1	0.103	123.5	33.15	160.91
GDC-normal	6.88	71.8	0.103	123.5	33.15	161.43
GDC-advanced	6.66	74.2	0.097	131.0	38.90	161.43
GDC-advanced-R	6.66	74.2	0.097	131.0	29.06	161.43
GDC-ultra	4.57	108.2	0.042	303.5	2.53	161.43

The '*' character after compressor name means that EOLs were initially removed.

Table 2. Compression results for *S.paradoxus* (36 genomes)

Method	Total		Avg. per seq.		c-sp	d-sp
	Size (MB)	Ratio	Size	Ratio	(MB/s)	(MB/s)
original	436.43	–	–	–	–	–
gzip*	120.43	3.6	3.345	3.6	3.53	91.86
bzip2*	112.24	3.9	3.118	3.9	6.08	15.61
7z*	5.34	81.7	0.069	176.6	0.49	156.93
RLCSA	46.48	9.4	–	–	0.54	1.33
Comrad	19.69	22.2	–	–	0.64	10.41
RLZ-opt*	13.44	32.5	0.300	40.4	1.44	170.48
RLZ-RePair*	7.70	55.7	–	–	0.26	144.54
GDC-fast	10.23	42.7	0.182	66.8	17.80	130.67
GDC-normal	9.20	47.4	0.182	66.8	17.66	124.34
GDC-advanced	8.73	50.0	0.169	71.9	20.49	128.36
GDC-advanced-R	8.73	50.0	0.169	71.9	16.99	128.36
GDC-ultra	5.01	87.2	0.062	196.2	3.03	151.54

The '*' character after compressor name means that EOLs were initially removed.

explained soon, however, also our specialized competitors, Comrad, RLZ-optand RLZ-RePair, cannot really support random access, at least within the memory use close to their respective archive sizes given in Tables 1–4. All the tested compressors (including all GDC variants) use as the reference sequence the sequence pointed as such in a given dataset description.

The other tested algorithms include three widely used general-purpose compressors, gzip, bzip2 and 7z, and specialized compressors, Comrad 0.2.1 (Kuruppu *et al.*, 2011a), RLZ-opt 0.1.1 and RLCSA (Mäkinen *et al.*, 2010) (the last-named being an index, hence we did not expect its compression ratios to be comparable to the rest). Comrad finds repeats over multiple passes through data, extending already compressed regions. The archives it produces are potentially searchable, but as far as we know neither the search (locate) nor display functionalities have been implemented. In

Table 3. Compression results for *H.sapiens* taken from Complete Genomics Inc. (70 genomes)

Method	Total		Avg. per seq.		c-sp	d-sp
	Size (MB)	Ratio	Size	Ratio	(MB/s)	(MB/s)
original	218 961.98	–	–	–	–	–
7z*	1 131.27	193.5	7.00	446.8	0.38	155.12
RLZ-opt*	1 731.31	126.5	15.26	204.9	1.34	130.38
GDC-fast	1 462.66	149.7	7.34	426.3	35.23	144.62
GDC-normal	1 201.15	182.3	7.34	426.3	35.90	146.16
GDC-advanced	1 201.25	182.3	7.34	426.3	36.36	147.05
GDC-advanced-R	1 201.25	182.3	7.34	426.3	26.71	146.65
GDC-ultra	910.13	240.6	3.12	1003.0	4.17	150.02

The '*' character after compressor name means that EOLs were initially removed.

Table 4. Compression results for *Arabidopsis thaliana* (TAIR), rice (TIGR) and Korean *H.sapiens*

Dataset	Raw	GRS			GDC		
	size (MB)	Ratio	c-sp (MB/s)	d-sp (MB/s)	Ratio	c-sp (MB/s)	d-sp (MB/s)
TAIR	120.65	18 160.0	2.52	2.99	21 615.0	3.23	39.56
TIGR	378.52	67.8	0.30	0.97	2760.2	2.63	48.28
Homo sapiens	3131.78	160.2	1.70	2.20	261.6	3.64	35.97

Two genomes in each collection, and only relative compression measured, in agreement with the methodology from Wang and Zhang (2011).

Table 4, we compare our GDC against GRS from Wang and Zhang (2011).

We have not tested the older DNA compressors, XM (Cao *et al.*, 2007) and dna2 (Manzini and Rastero, 2004), because their available implementations handle only the four-symbol alphabet (we note also that XM is extremely slow). In spite of our attempts, we were unable to get a fully working version of the LZ-End compressor (Kreft and Navarro, 2010) from the authors; the program we had in our hands sends to the output only semi-compressed data (not interesting from the compression point), and thus also compression time and especially decompression time cannot be honestly measured. For these reasons, we gave up benchmarking it.

The results in Tables 1–3 show compression ratios and compression / decompression speeds (MB stands for 10^6 bytes). For each run, the ratio is specified as a pair of numbers: overall (in the column 'Total') as the ratio between the original dataset size and the compressed dataset size, and differential (in the column 'Avg. per seq.') as a similar ratio but with the reference sequence sizes subtracted from both terms. In Table 4, only the differential ratios are given.

3.2 Main results

GDC improves compression ratio and speed: on the yeast collections (Tables 1 and 2), the main competitors of GDC are RLZ-opt and RLZ-RePair. When compression only matters, the GDC-ultra variant clearly wins in compression ratio with RLZ-RePair,

Table 5. Random access time for 70 (top five rows) and 4 (bottom six rows) genomes of *H.sapiens*, for varying pattern length, m

	File size	RAM usage	$m=10$	$m=100$	$m=1000$	$m=10\,000$
GDC-fast/normal	1465.2 / 1204.2	–	–	–	–	–
GDC-ra-2^{11}-2^{13}	1761.1 / 1499.7	2124 / 1863	1180 / 3800	129 / 398	23 / 57	12 / 23
GDC-ra-2^{11}-2^{15}	1578.9 / 1317.4	1739 / 1477	1710 / 4390	188 / 457	58 / 64	12 / 23
GDC-ra-2^{11}-2^{17}	1501.6 / 1240.1	1582 / 1320	4040 / 6720	408 / 681	50 / 85	14 / 25
GDC-ra-2^{11}-2^{19}	1479.0 / 1217.5	1537 / 1275	11 750 / 14 580	1190 / 1474	129 / 165	22 / 33
RLCSA	5429.0	5685	33830	4698	1651	1282
GDC-fast/normal	980.4 / 719.4	–	–	–	–	–
GDC-ra-2^{11}-2^{13}	997.1 / 736.6	1051 / 789	830 / 3360	90 / 351	18 / 52	10 / 21
GDC-ra-2^{11}-2^{15}	990.6 / 729.1	1038 / 774	1300 / 3810	140 / 404	22 / 56	11 / 21
GDC-ra-2^{11}-2^{17}	987.2 / 725.8	1031 / 770	3170 / 5770	320 / 593	41 / 75	12 / 23
GDC-ra-2^{11}-2^{19}	986.2 / 724.7	1027 / 766	10070 / 12830	1017 / 1280	108 / 143	19 / 30

All times are in nanosecond per character. File and memory sizes are in Megabytes. The complete collections are of size: 218 962 and 12 253 MB.

being even about twice faster than RLZ-opt. RLZ-RePair is better by 3–12% in compression ratio than GDC-advanced, but cannot scale: its compression speed is two orders of magnitude worse than GDC-advanced, and memory consumption is huge (∼12 GB). RLZ-opt is faster and uses less memory, but for the price of compression loss; the total, i.e. including also the reference sequence, GDC-advanced archives are smaller by 29 or 35%, respectively. Both RLZ variants (their existing implementations) cannot handle random access queries without a significant inflation of its data structures in memory; in particular, their 7z-compressed reference sequence must be decompressed before packing into bytes, which is a memory-demanding operation, especially for RLZ-RePair, where the artificial reference sequences may consist of a large fraction of the input dataset.

In decompression, however, RLZ-RePair is up to 16% faster than GDC variants (the difference under the same conditions is actually by about 5% smaller, because of the removed EOL symbols before RLZ compression). Still, both compressors achieve well over 100 MB/s decompression speed, which is at least at the I/O subsystem speed level, that is, no hindrance in most real scenarios.

Interestingly, in those tests the general-purpose 7z appears to be a strong competitor. GDC-ultra is better by <10% in compression ratio, which can even be improved a little if the reference sequence is 7z-compressed, but 5–6 times faster in compression (decompression speeds are comparable). On the other hand, 7z clearly wins with RLZ-RePair. Part of the success of 7z comes from the fact that it uses a huge, 1 GB LZ-buffer. The respective buffers/blocks for gzip and bzip2 are 32 kB and 900 kB, much too small to exploit inter-sequence redundancies, making them totally useless for this compression scenario (for this reason, they are not run in the successive experiments).

On 70 human genomes (Table 3), the overall picture is similar, yet some differences should be pointed out. First, we performed the matching at the chromosome level rather than on whole sequences (this concerns all the tested algorithms). We believe this is a sound approach from the biological point and also clearly beneficial for the compression speed and memory use requirements. Second, despite reordering the data by chromosomes, this 220 GB compression ordeal was too hard for some of the competitors, namely Comrad and RLZ-RePair. The former would need more than a week to finish its work, while the latter (even slower) would require much more memory than was available. RLZ-opt and 7z did manage to compress this huge collection, but the time they required was 45 and 160 h, respectively. On the other hand, most GDC variants (except for GDC-ultra) needed <2 h. Decompression was in all cases fast, again largely exceeding 100 MB/s. The total archive sizes were relatively close (ranging from 910 MB for GDC-ultra to 1731 MB for RLZ-opt), but the differential ratios varied more, with GDC-ultra achieving the factor 1003. In other words, we showed that a single human genome, in a large enough collection, may be compressed to ∼3 MB, using no 'external' knowledge.

We also checked if matching at the chromosome level may miss many large cross-chromosome matches. It does not seem so; the resulting GDC-normal compression ratios (total and differential) were by <1% worse than in the previous experiment. Compression and decompression speed suffered more; they dropped to 18.5 and 96.1 MB/s, respectively. It may be interesting to report the memory use during the compression process in this mode (i.e. when whole genomes are matched). GDC variants need ∼12 GB of RAM on the human collections (similar results for both the 4 and the 70 genomes). RLCSA is more demanding; its peak memory use was 25 GB on the four genomes while on the 70 genomes it required too much memory to be run on our machine. RLZ-opt cannot be run in this test, due to its filesize limitation to 2 GB.

The results of the experiment involving four human genomes are shown in the Supplementary Material. Basically, the observations are similar, with the difference that 7z wins over GDC-ultra by a few per cent in compression ratio (but differentially encoded genomes are about 1.7 times smaller with GDC-ultra), for the price of being 29 times slower in compression.

GDC outperforms GRS in relative compression: next (Table 4), we compared our tool against GRS from Wang and Zhang (2011), on the datasets used in the cited work, in the same scenario. This means that both compared algorithms are based on relative compression and only the differential archive sizes are measured. We tried to run GRS also on the datasets from our previous experiments. On the two yeast datasets it refused to proceed, claiming that the sequences are too dissimilar to one another. On the four human genomes, it worked extremely slowly, and preliminary results (on a single

chromosome) showed that its compression ratio was worse than gzip's. We conclude that GRS cannot be considered a usable tool, sometimes producing very unstable results (cf. Table 4, TIGR dataset compression ratios).

GDC provides fast random access: finally, we measured the random access times (Table 5) for our algorithm, for the two human genome collections (70 and 4 genomes), varying the displayed pattern size and the block size parameters. The two numbers in each GDC description (one per row, e.g. 2^{11}-2^{15}) correspond to the reference sequence block size and any other sequence block size, in symbols. The two values in GDC-related table cells correspond to the fast and the normal mode. Clearly, there is a space–time trade-off here. Using the fast mode is an obvious choice for large collections, but when the number of genomes is small, then the difference in compression between it and the normal mode becomes more significant. RLCSA answered the queries significantly (often over an order of magnitude) slower than GDC. As for RLZ, it was unable to perform random access queries on the human collections, due to the (mentioned earlier) filesize limitation to 2 GB (its random access timings on the two smaller collections are given in the Supplementary Material). It would also make sense to compare GDC against LZ-End (Kreft and Navarro, 2010) in the random access tests. Unfortunately, we were not able to obtain a fully operational copy of this program from its authors, despite our attempts.

3.3 Parameter tuning

Human genome sequences are not only large, but also much more similar to each other than the yeast datasets. An adverse side-effect of this phenomenon is that there are lots of collisions in hashing which make the compression slow. To mitigate this effect, we took a couple of precautions. On the yeast datasets, the minimum match parameters (described in the previous section) are set as follows: $M_1 = 13$, $M_2 = 4$, while on the human genomes they are set to: $M_1 = 20$, $M_2 = 4$. This, for example, means that the hashed sequences are longer on the human genomes and collisions are rarer.

The parameters M_1, M_2, M_3 were chosen experimentally but without severe tuning. Still, it might be interesting to know how their selection affects the efficiency of our algorithms. Recall that M_3 has a meaning only in GDC-advanced mode and corresponds to the minimum length of a literal run treated as an extra reference phrase, that is, setting it to a very high value practically turns the advanced into the normal variant (both in compression ratio and speed), which, we believe, is still very competitive. The parameter M_2, on the other hand, concerns approximate matches only. Setting M_2 to a much too high value (instead of 4 in all our experiments) results in compression loss across all datasets, from a few to about 10%. The compression speed may also drop moderately. Finally, M_1 cannot be too small, since encoding short matches and the triple (offset, length, flag) is costlier than encoding them as individual DNA letters. It seems that the optimum is around $M_1 = 13$ for all datasets. For human genomes, however, which are much more repetitive than other collections of sequences, this threshold had to be raised, otherwise the compression speed dropped several times. According to our preliminary experiments, also for human sequences using $M_1 = 13$ gives (slightly) better compression, but the price is too high. We admit that manual setting of different values to M_1 is a weakness of our algorithm, and for compressing a large collection of genomes it may be recommended to try out a few parameter values on a small fraction of the given input, to get an idea about the compression ratio / speed trade-offs.

The interplay between the parameters M_1, M_2, M_3, compression speed and compression (overall or differential) ratio was carefully examined on the individual datasets, and the plots are included in the Supplementary Material.

4 CONCLUSION

We presented a specialized compressor, GDC, for storing DNA sequences (genomes) from many individuals of the same species. Its functionalities include also random access to arbitrary snippets of the compressed data. Experimental comparison with the predecessor of our solution, RLZ-opt, shows that we are more than an order of magnitude faster in compression with significantly higher compression ratios. The advantage in compression ratio gets particularly striking when only differential results are taken into account; here our tool is better than RLZ-opt close to three times on human genomes. Those differential compression ratios, that is, the fraction of a full genome its compressed representation occupies, when represented with reference to another genome of the same species, will get more and more important with the growth of genomic databases, a trend which is already clearly visible with the rapid progress in DNA sequencing technology.

The key new ideas of our solution are augmenting the reference sequence with extra reference phrases taken from the other sequences (without compromising compression or decompression speed), specific LZ-parsing that implicitly detects some class of approximate repeats and more compact encoding of the various byproducts of the scheme.

ACKNOWLEDGEMENT

We thank Shanika Kuruppu and Simon Puglisi for providing us with their software and tips on how to use it.

Funding: Polish Ministry of Science and Higher Education under the project (N N516 441938); European Community from the European Social Fund, in part.

Conflict of Interest: none declared.

REFERENCES

Brandon,M.C. *et al.* (2009) Data structures and compression algorithms for genomic sequence data. *Bioinformatics*, **25**, 1731–1738.

Cao,M.D. *et al.* (2007) A simple statistical algorithm for biological sequence compression. In *Proceedings of the DCC*. IEEE Computer Society Press, Washington, DC, USA, pp. 43–52.

Christley,S. *et al.* (2009) Human genomes as email attachments. *Bioinformatics*, **25**, 274–275.

Claude,F. and Navarro,G. (2008) Practical rank/select queries over arbitrary sequences. *Lect. Notes Comput. Sci.* **5280**, 176–187.

Claude,F. *et al.* (2010) Compressed q-gram indexing for highly repetitive biological sequences. In *Proceedings of the International Conference on Bioinformatics Bioengineering*. IEEE Computer Society Press, Washington, DC, USA, pp. 86–91.

Ferragina,P. *et al.* (2009) On the bit-complexity of Lempel–Ziv compression. In *Proceedings of the SODA*. SIAM, Philadelphia, PA, USA, pp. 768–777.

Grabowski,S. and Deorowicz,S. (2011) Engineering relative compression of genomes. *CoRR* , abs/1103.2351, 1–12.

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

Kreft,S. and Navarro,G. (2010) LZ77-like compression with fast random access. In *Proceedings of the DCC*. IEEE Computer Society Press, Washington, DC, USA, pp. 239–248.

Kreft,S. and Navarro,G. (2011) Self-Indexing based on LZ77. *Lect. Notes Comput. Sci.* **6661**, 41–54.

Kuruppu,S. *et al.* (2010) Relative Lempel–Ziv compression of genomes for large-scale storage and retrieval. *Lect. Notes Comput. Sci.* **6393**, 201–206.

Kuruppu,S. *et al.* (2011a) Iterative dictionary construction for compression of large DNA datasets. *IEEE ACM Trans. Comput. Biol. Bioinformatics*, **99**, [Epub ahead of print, doi: 10.1109/TCBB.2011.82, April 27, 2011].

Kuruppu,S. *et al.* (2011b) Optimized relative Lempel–Ziv compression of genomes. In *Proceedings of the ACSC*. Australian Computer Society, Inc., Sydney, Australia, pp. 91–98.

Kuruppu,S. *et al.* (2011c) Reference sequence construction for relative compression of genomes. In *Proceedings of the SPIRE*, (In press).

Larsson,N.J. and Moffat,A. (2000) Off-line dictionary-based compression. *Proc. IEEE*, **88**, 1722–1732.

Mäkinen,V. *et al.* (2010) Storage and retrieval of highly repetitive sequence collections. *J. Comput. Biol.*, **17**, 281–308.

Manzini,G. and Rastero,M. (2004) A simple and fast DNA compressor. *Software Pract. Exper.*, **34**, 1397–1411.

Wang,C. and Zhang,D. (2011) A novel compression tool for efficient storage of genome resequencing data. *Nucleic Acids Res.*, **39**, 25.[Epub ahead of print, doi:10.1093/nar/gkr009, January 25, 2011].