Bioinformatics, 31(21), 2015, 3468–3475
doi: 10.1093/bioinformatics/btv399
Advance Access Publication Date: 2 July 2015
Original Paper

OXFORD

Sequence analysis

ERGC: an efficient referential genome compression algorithm

Subrata Saha and Sanguthevar Rajasekaran*

Department of Computer Science and Engineering, University of Connecticut, Storrs, CT 06269, USA

*To whom correspondence should be addressed.

Associate Editor: John Hancock

Received on February 11, 2015; revised on June 12, 2015; accepted on June 17, 2015

Abstract

Motivation: Genome sequencing has become faster and more affordable. Consequently, the number of available complete genomic sequences is increasing rapidly. As a result, the cost to store, process, analyze and transmit the data is becoming a bottleneck for research and future medical applications. So, the need for devising efficient data compression and data reduction techniques for biological sequencing data is growing by the day. Although there exists a number of standard data compression algorithms, they are not efficient in compressing biological data. These generic algorithms do not exploit some inherent properties of the sequencing data while compressing. To exploit statistical and information-theoretic properties of genomic sequences, we need specialized compression algorithms. Five different next-generation sequencing data compression problems have been identified and studied in the literature. We propose a novel algorithm for one of these problems known as *reference-based genome compression.*

Results: We have done extensive experiments using five real sequencing datasets. The results on real genomes show that our proposed algorithm is indeed competitive and performs better than the best known algorithms for this problem. It achieves compression ratios that are better than those of the currently best performing algorithms. The time to compress and decompress the whole genome is also very promising.

Availability and implementation: The implementations are freely available for non-commercial purposes. They can be downloaded from http://engr.uconn.edu/∼rajasek/ERGC.zip.

Contact: rajasek@engr.uconn.edu

1 Introduction

Next-generation sequencing (NGS) technologies are producing millions to billions of short reads from DNA molecules simultaneously in a single run within a very short time period, leading to a sharp decline in whole genome sequencing costs. As a result, we are observing an explosion of genomic data from various species. Storing these data is an important task that the biologists have to perform on a daily basis. To save space, compression could play an important role. Also, when the size of the data transmitted through the Internet increases, the transmission cost and congestion in the network also increase proportionally. Here again compression could help. Although we can compress the sequencing data through standard general purpose algorithms, these algorithms may not compress the biological sequences effectively, since they do not exploit inherent properties of the biological data. Genomic sequences often contain repetitive elements, e.g. microsatellite sequences. The input sequences might exhibit high levels of similarity. An example will be multiple genome sequences from the same species. Additionally, the statistical and information-theoretic properties of genomic sequences can potentially be exploited. General purpose algorithms do not exploit these properties. In this article, we offer a novel algorithm to compress genomic sequences effectively and efficiently. Our algorithm achieves compression ratios that are better than the currently best performing algorithms in this domain. By compression ratio, we mean the ratio of the uncompressed data size to the compressed data size.

The following five versions of the compression problem have been identified in the literature: (i) *genome compression with a reference*: here we are given many (hopefully very similar) genomic sequences. The goal is to compress all the sequences using one of them as the reference. The idea is to utilize the fact that the sequences are very similar. For every sequence other than the reference, we only have to store the difference between the reference and the sequence itself; (ii) *reference-free genome compression*: this is the same as Problem 1, except that there is no reference sequence. Each sequence has to be compressed independently; (iii) *reference-free reads compression*: it deals with compressing biological reads where there is no clear choice for a reference; (iv) *reference-based reads compression*: in this technique, complete reads data need not be stored but only the variations with respect to a reference genome are stored; and (v) *metadata and quality scores compression*: in this problem, we are required to compress quality sequences associated with the reads and metadata such as read name, platform and project identifiers.

In this article, we focus on Problem 1. We present an algorithm called ERGC (Efficient Referential Genome Compressor) based on a reference genome. It employs a divide and conquer strategy. At first it divides both the target and reference sequences into some parts of equal size and finds one-to-one maps of similar regions from each part. It then outputs identical maps along with dissimilar regions of the target sequence. The rest of this article is organized as follows: Section 2 has a literature survey. Section 3 describes the proposed algorithm and analyses its time complexity. Our experimental platform is explained in Section 4. This section also contains the experimental results and discussions. Section 5 concludes the article.

2 A survey of compression algorithms

We now survey some of the algorithms that have been proposed in the literature to solve Problem 1. In referential genome compression, the goal is to compress a large set S of similar sequences. The core idea of reference-based compression can be described as follows. We first choose the reference sequence R. Then we compress every other sequence $s \in S$ by comparing it with R. The target (i.e. the current sequence to be compressed) is first aligned to the reference. Then mismatches between the target and the reference are identified and encoded. Each record of a mismatch may consist of the position with respect to the reference, the type (e.g. insertion, deletion or substitution) of mismatch and the value.

Brandon *et al.* (2009) have used various coders like Golomb (Golomb, 1966), Elias (Peter *et al.*, 1975) and Huffman (Huffman, 1952) to encode the mismatches. Wang and Zhang (2011) have presented a *de novo* compression program, GRS, which obtains variant information by using a modified Unix *diff* program. The algorithm GReEn (Pinho *et al.*, 2012) employs a probabilistic copy model that calculates target base probabilities based on the reference. Given the base probabilities as input, an arithmetic coder was then used to encode the target. Recently, another algorithm, namely, iDoComp (Ochoa *et al.*, 2014) has been proposed which outperforms some of the previously best known algorithms like GRS, GReEn and GDC. GDC (Deorowicz and Grabowski, 2011) is an LZ77-style compression scheme for relative compression of multiple genomes of the same species. In contrast to the algorithms mentioned above, Christley *et al.* (2009) have proposed the DNAzip algorithm that exploits the human population variation database, where a variant can be a single-nucleotide polymorphism (SNP) or an indel (an insertion or a deletion of multiple bases). Some other notable algorithms that employ VCF (Variant Call Format) files to compress genomes have been given by Deorowicz *et al.* (2013) and Pavlichin *et al.* (2013). These algorithms have been used in the 1000 Genomes project to encode SNPs and other structural genetic variants. Next we explain elaborately some of the best known algorithms in this domain.

2.1 GRS

GRS is a reference-based genome compression tool exclusively dependent on the Unix program *diff* as mentioned above. Specifically, the primary step of GRS is to find longest common subsequences in two input strings. The auxiliary Unix program *diff* is employed to calculate a similarity measure between a target genomic sequence and a reference genomic sequence. If the similarity score exceeds a predefined threshold, the difference between the target and reference genomic sequences is encoded using Huffman encoding. If the similarity score is below the threshold, the target and reference sequences are split into smaller blocks and the computation is restarted on each pair of the blocks. It is to be noted that GRS does not require any additional information about the sequences, e.g. a reference SNPs map. If there exists an excessive difference between the target and reference sequences, GRS will not be able to compress the target sequence effectively.

2.2 GDC

GDC is a LZ77-style (Ziv and Lempel, 1977) compression algorithm closely related to RLZopt (Shanika *et al.*, 2011) where GDC performs a non-greedy parsing of the target into the reference by hashing. On the contrary, RLZopt uses a suffix array. The main difference between GDC and the other reference-based compression tools is that it can choose a suitable reference (or, more than one reference) sequence among the set of genomic sequences from the same species using a heuristic and use it to compress the rest. It also introduces a clever trick to encode approximate matches. The algorithm slightly alters the original Lempel-Ziv parsing scheme by considering trade-offs between the length of matches and distance between matches. Compression is performed on input blocks with shared Huffman codes by enabling random access of the reference.

2.3 GReEn

GReEn is also a reference-based genome compression tool. It encodes the target sequence using an arithmetic encoder. At first it generates statistics using the reference sequence and then performs the compression of the target by employing arithmetic coding. Arithmetic encoder uses the previously computed statistics to encode the target. From experimental results, it is evident that GReEn outperforms both GRS and the non-optimized RLZ. Similar to the non-referential compression scheme XM (Cao *et al.*, 2007), GReEn introduces a copy expert model. This model tries to find identical k-mers between the target and reference sequences. Raw characters in the form of arbitrary ASCII characters are encoded with arithmetic encoding. However, there is a special case where target and reference sequences have equal length. Although not justified, GReEn assumes that the sequences are already aligned and can be distinguished by SNPs.

2.4 iDoComp

The basic functioning of iDoComp can be summarized in three main steps: (i) *mapping generation*: in this stage, the target genome is expressed in terms of the reference genome. It uses suffix arrays to parse the target into the reference; (ii) *post-processing*: the post-processing looks for consecutive matches that can be merged together and converted into an approximate match and (iii) *entropy encoding*: entropy encoder compresses the mapping and generates the compressed file.

3 Methods

3.1 Our algorithm

We have developed a reference-based genome compression algorithm called ERGC. It performs better than the best known algorithms existing in the current literature. Our algorithm runs in stages. Each stage is independent of the previous stage(s). In this setting, it can be readily transferred from in-core to out-of-core model and single-core to multi-core environment. We will discuss these enhancements later in this section. Details of our algorithm follow. Assume that R is the reference sequence and T is a target sequence to be compressed. At first ERGC divides the entire reference and target genomes into parts of equal sizes and processes each pair of parts sequentially. If the parts in R and T are $r_1, r_2, \ldots, r_q$ and $t_1, t_2, \ldots, t_q$, respectively, then r_1 and t_1 are processed first, r_2 and t_2 are processed next and so on.

Let (r', t') be the pair processed at some point in the algorithm (where r' comes from the reference genomic sequence R and t' comes from the target genomic sequence T). To find the similarities between r' and t', we need to align t' onto r'. Similar regions between two sequences can be found globally aligning t' onto r' using Needleman–Wunsch algorithm as the sequences in the query set are similar and of roughly equal size. Since the time complexity of the global alignment algorithm is quadratic and thus based on dynamic programming, it is a very time and space intensive procedure especially when the length of the sequences is very large. In this context, we have devised our own *greedy alignment* algorithm to find similar regions between two sequences with high confidence (it is applicable when the sequences of interest are highly similar, e.g. two genomic sequences of the same species). Now we describe our greedy algorithm next.

Our *greedy alignment* algorithm is based on hashing. At first, the algorithm generates all the k-mers from r' and hashes them into a hash table H (for some suitable values of k). It then generates k-mers from t' one at a time and hashes them to H until one of these k-mers collides with an entry in H. If none of the k–mers collides with an entry in H, the algorithm generates another set of k'-mers (where $k' < k$) from r' and hashes them into a hash table H'. In a similar way, it then generates k'-mers from t' one at a time and hashes them to H' until one of these k'-mers collides with an entry in H'. We employ a predefined set of values for k and try these values one at a time until a collision happens. The reason for taking a range of k–mer values is that the occurrences of substitutions, insertions and/or deletions can be more frequent in some parts of r' and t' than in the others. A range of values for k ensures that for at least one value a collision will occur. If there is no such collision, it is not possible to align t' onto r'. If none of the values of k from this set results in a collision, then the algorithm extends the length of r' on both sides and a similar scheme is followed as described above. If all of the above mentioned techniques fail, then t' is saved as a raw sequence. Otherwise we align r' and t' with the k-mer that causes a collision as the anchor and extend the alignment beyond that position until there is a mismatch between r' and t'. We record the matching length and the starting position of this stretch of matching in the reference genome. At this point, we delete the matching sequences from r' and t' and align the rest using our greedy algorithm as described above until the length of r' or t' becomes zero or there can not be any further alignment possible. This is how the algorithm proceeds iteratively. Next we describe how ERGC takes care of unmatched sequences.

As there can be substitutions, insertions and/or deletions in the reference and target genomes, some portions of the genomes between two alignments will not be matched perfectly. In this case, we align those sequences using edit scripting. If the edit distance is large enough to exceed the cost to store the unmatched sequence of the target genome, we discard the edit script and store the raw sequence. The information generated to compress the target sequence is stored in an ASCII formatted file. After having processed all the parts of r' and the corresponding parts in t', we compress the starting positions and matching length using delta encoding. The resulting file is further compressed by using PPMd lossless data compression algorithm. It is a variant of prediction by partial matching algorithm and an adaptive statistical data compression technique based on context modeling and prediction. For more details, the reader is referred to Moffat *et al.* (1990). Some recent implementations of PPMd are effective in compressing text files containing natural language text. The seven-zip open-source compression utility provides several compression options including the PPMD algorithm. Details of the algorithm are shown as Algorithm 1.

Values of parameters such as set K, q and τ have been optimized to get the best results. In our experiments, we have used default values of K and q where τ is chosen dynamically. The set K contains two fixed values, i.e. $K = \{21, 9\}$. At first ERGC tries to align using 21-mers (i.e. $k = 21$). If it fails, $k = 9$ is picked to align the parts. The value of q is chosen in such a way that each part is composed of 20 000 nucleotides approximately. If the unaligned substrings from the reference and target are approximately equal and the memory needed to store the cost of edit distance information exceeds the memory needed to store the raw sequence, ERGC discards the edit distance information and stores the raw sequence as an ASCII formatted text file. Again, to speed up the proposed algorithm we have used several techniques.

Algorithm 1: ERGC

Input: Reference sequence R, target sequence T, a threshold τ, a set K of values for k. The set K contains two fixed values $K = \{21, 9\}$. τ is the memory needed to store the raw sequence of interest

Output: Compressed sequence T_C

begin

1 Divide R and T into q equal parts. Let these be $r_1, r_2, \ldots, r_q$ and $t_1, t_2, \ldots, t_q$, respectively;
2 **for** $i := 1$ **to** q **do**
3 Hash the k-mers (for a suitable value of k from K) of r_i into a hash table H;
4 Generate one k-mer at a time from t_i and hash it into H;
5 If there is no collision try different values of k from K and repeat lines 3 and 4;
6 If all the different k-mers have been tried with no collision, extend the length of r_i and go to line 3;
7 When a collision occurs in H, align r_i and t_i with this common k-mer as the anchor;
8 Extend the alignment beyond the common k-mer until there is a mismatch;
9 Record the matching length and the starting position of this match in the reference genome R;
10 Compute the edit distance between unmatched subsequences (one each from r_i and t_i);
11 If the edit distance d_i is $\geq \tau$, store the raw (unmatched) subsequence of t_i;
12 Otherwise store the edit script information;
13 Compress the stored information using delta encoding;
14 Encode the stored information using PPMD encoder;
15 Return the compressed sequence T_C

3.2 An illustrative example

Let us illustrate our algorithm with a suitable example in brief. Please see Figure 1 for visual details. Suppose (r', t') is the pair processed at some point in the algorithm. As described above, r' comes from the reference genomic sequence R, and t' comes from the target genomic sequence T. Let $a_1a_2 \ldots a_{|r'|}$ and $b_1b_2 \ldots b_{|t'|}$ be nucleotide positions in r' and t', respectively. At first, we hash the k-mers (for a suitable value of k from K) of r' into a hash table H. In this procedure, similar k-mers fall into the same bucket in the hash table H. We record the position of occurrence of each k-mer. Next we generate one k-mer at a time from t' and hash it into H. Let a specific k'-mer starting at position b_p in t' collide with an entry of H. As a number of identical k'-mers can be found across the genomic sequence, the bucket can have multiple positions of identical k'-mers. Next we retrieve the k'-mer from the bucket which has the least position among all the identical k'-mers in the same bucket. Let the position be a_i in r'. The proposed method then aligns t' onto r' using b_p and a_i and extend the alignment beyond the common k'-mers until a mismatch is found. While extending, the first mismatch occurs in a_{j+1} and b_{q+1} of r' and t', respectively. So, $b_p \ldots b_q$ can be represented with respect to r' by recording the position of a_i and the length of $b_p \ldots b_q$. The same procedure is repeated again. The unaligned substring $b_{q+1} \ldots b_{r-1}$ is aligned with $a_{j+1} \ldots a_{k-1}$ by considering some heuristics. At this point, three cases are possible.

3.2.1 Case 1: mutations in r' and t'

In this case, the unaligned substring of t' is aligned with r' employing edit distance calculations. In our example, this case arises while we attempt to align unmatched substring $b_{q+1} \ldots b_{r-1}$ onto $a_{j+1} \ldots a_{k-1}$. If the lengths of $b_{q+1} \ldots b_{r-1}$ and $a_{j+1} \ldots a_{k-1}$ are approximately equal and the memory needed to store the edit distance information between them is less than for the raw sequence $b_{q+1} \ldots b_{r-1}$, we store $b_{q+1} \ldots b_{r-1}$ with respect to $a_{j+1} \ldots a_{k-1}$ by recording the starting position of a_{j+1} and edit distance information. Otherwise we store the raw sequence $b_{q+1} \ldots b_{r-1}$.

3.2.2 Case 2: insertions in r'

In this case, nucleotides are inserted in positions $a_{l+1} \ldots a_{m-1}$ of r'. We align b_{s+1} onto a_m and extend it until we find any mismatch using a similar procedure and record the positions of a_m and $|b_{s+1} \ldots b_t|$ as described above.

3.2.3 Case 3: insertions in t'

Case 3 arises when nucleotides are inserted in positions $b_{t+1} \ldots b_{u-1}$ of t'. In this special case, we store the raw sequence $b_{t+1} \ldots b_{u-1}$ and the starting position of b_{t+1} as there is no other choice left.

3.3 Time complexity analysis

Consider a pair of parts r and t (where r comes from the reference and t comes from the target). Let $|r| = |t| = \ell$. We can generate k-mers from r and hash them in $O(\ell k)$ time. The same amount of time is spent, in the worst case, to generate and hash the k-mers of t. The number of different k-values that we try is a small constant and hence the total time spent in all the hashing that we employ is $O(\ell k)$. If a collision occurs, then the alignment we perform is greedy and takes only $O(\ell)$ time. After the alignment recording the difference and subsequent encoding also takes linear (in ℓ) time.

If no collision occurs for any of the k-values tried, t is stored as such and hence the time is linear in ℓ. Put together, the run time for processing r and t is $O(\ell k)$. Extending this analysis to the entire target sequence, we infer that the run time to compress any target sequence T of length n is $O(nk)$ where k is the largest value used in hashing.

It is easy to see that our algorithm can be implemented in a single pass through the data and thus can be employed in an out-of-core setting by using an appropriate value for the length of r and t. This can be ensured by choosing the length of r and t to be $\Theta(M)$ where M is the size of the core memory. The performance of any out-of-core algorithm is measured in terms of the number of I/O operations performed. A *single pass through the data* refers to the number of I/O operations needed to bring each data item exactly once into the core memory. As a result, ERGC is optimal in terms of out-of-core computing.

4 Results and discussion

4.1 Experimental environment

We have compared our algorithm with the best known algorithms existing in the referential genome compression domain. In this section, we summarize the results. All the experiments were done on an Intel Westmere compute node with 12 Intel Xeon X5650 Westmere cores and 48 GB of RAM. The operating system running was Red Hat Enterprise Linux Server release 5.7 (Tikanga). ERGC

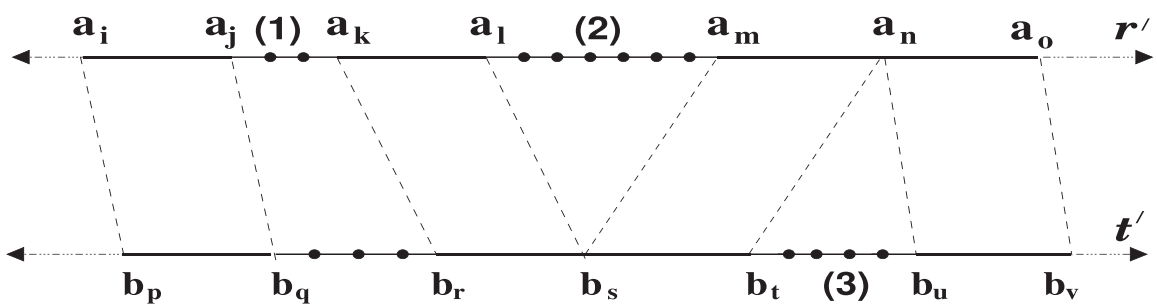


Fig. 1. An illustrative example of ERGC. Here (r', t') is the pair processed at some point in the algorithm. As $b_p \ldots b_q$ is aligned onto $a_i \ldots a_j$ using greedy alignment algorithm, we only need to store the position of a_i and the length of the matching alignment, i.e. $|a_i \ldots a_j|$. The next alignment begins at a_k and b_r from r' and t', respectively. The unmatched sequences in between are $a_{j+1} \ldots a_{k-1}$ and $b_{q+1} \ldots b_{r-1}$. $b_{q+1} \ldots b_{r-1}$ can be saved as a raw sequence or using edit scripting. In a similar fashion, the sequence information of $b_r \ldots b_s$, $b_{s+1} \ldots b_t$ and $b_u \ldots b_v$ are stored using $a_k \ldots a_l$, $a_m \ldots a_n$ and $a_{n+1} \ldots a_o$, respectively. Since $b_{t+1} \ldots b_{u-1}$ could not be aligned, it is stored as a raw sequence

compression and decompression algorithms are written in standard Java programming language. Java source code is compiled and run by Java Virtual Machine (JVM) 1.6.0.

4.2 Datasets

To measure the effectiveness of our proposed algorithm, we have done a number of experiment using real datasets. We have used hg18 release from the UCSC Genome Browser, the Korean genomes *KOREF_*20090131 (KOR131 for short) and *KOREF_*20090224 (KOR224 for short) (Ahn *et al.*, 2009) and the genome of a Han Chinese known as YH (Levy *et al.*, 2008). Since to compress a genomic sequence we need a reference genome, we have randomly chosen five pairs of target-reference sequences from the above benchmark datasets. We have taken chromosomes 1–22, X and Y chromosomes (i.e. a total of 24 chromosomes) for comparison purposes. Please see Table 1 for details about the datasets we have used.

4.3 Discussion

Next we present details on the performance evaluation of our proposed algorithm ERGC with respect to both compression and running time. We have compared ERGC with two of the three best performing algorithms namely GDC and iDoComp using several standard benchmark datasets. GReEn is another state-of-the-art algorithm existing in the literature. But we could not compare it with our algorithm, as the site containing the code was down at the time of experiments. GDC, GReEn and iDoComp are highly specialized algorithms designed to compress genomic sequences with the help of a reference genome. These are the best performing algorithms in this area as of now.

Given a reference sequence, our algorithm compresses the target sequence by exploiting the reference. So, it needs the reference sequence at the time of decompression also. We use the target and reference pairs of sequences illustrated in Table 1 to assess the effectiveness of the algorithm. Some notable algorithms such as Pavlichin *et al.* (2013), Deorowicz *et al.* (2013) and Christley *et al.* (2009) exploit SNPs/indels variation files and achieve high compression ratios. But it may not be possible to find variation files for every species and these algorithms will not work without variation files. Our algorithm does not employ variation files and so it can compress any genomic sequence given a reference. As a result, we feel that algorithms that employ variation files form a separate class of algorithms and are not comparable to our algorithm. Again our proposed algorithm is devised in such a way that it is able to work with any alphabet used in the genomic sequence. Every other algorithm works only with valid alphabets intended for genomic sequence e.g. $\sum = \{A, a, C, c, G, g, T, t, N, n\}$. The characters most commonly seen in sequences are in $\sum$ but there are several other valid characters that are used in clones to indicate ambiguity about the identity of certain bases in the sequence. It is not uncommon to see these 'wobble' codes at polymorphic positions in DNA sequences. It also differentiates between lower-case and upper-case letters. GDC, GReEn and iDoComp can differentiate between upper-case and lower-case letters specified in $\sum$ but previous algorithms like GRS or RLZ-opt only work with A, C, G, T and N in the alphabet. iDoComp replaces the character in the genomic sequence that does not belong to $\sum$ with *N*. Specifically, ERGC will compress the target genome file regardless of the alphabets used and decompress the compressed file which is exactly identical to the target file. This is the case for GDC and iDoComp also but GReEn does not include the metadata information and output the sequence as a single line instead of multiple lines.

Effectiveness of various algorithms including ERGC is measured using several performance metrics such as compression size, compression time, decompression time, etc. Gain measures the percentage improvement over the compression achieved by ERGC with respect to GDC and iDoComp. Comparison results are shown in Table 2. Clearly, our proposed algorithm is competitive and performs better than all the best known algorithms. In Tables 3 and 4, we show a comparison between compressed size (from different algorithms) and the actual size of individual chromosomes for some datasets. Memory consumption is also very low in our algorithm as it only processes one and only one part from the target and reference sequences at any time. Please note that we did not report the performance evaluation of GDC for every dataset, as it ran at least 1 h but did not able to compress a single chromosome for some datasets.

As stated above, ERGC differentiates upper-case and lower-case characters. It compresses target file containing the genomic sequence to be compressed and metadata if any with the help of a reference. The decompression procedure produces exactly the same file as the input. It does not depend on the alphabets and is universal in this sense. Consider dataset D_1 where the target and reference sequences/chromosomes are from YH and hg18, respectively (Table 1). In this setting, GDC runs indefinitely. iDoComp

Table 1. Genomic sequence datasets used for the referential compression evaluation

Dataset	Species	Chr.	Sequence	Taken from
D_1	*Homo sapiens*	24	Target: YH	yh.genomics.org.cn
			Reference: hg18	ncbi.nlm.nih.gov
D_2	*Homo sapiens*	24	Target: YH	yh.genomics.org.cn
			Reference: KO224	koreangenome.org
D_3	*Homo sapiens*	24	Target: YH	yh.genomics.org.cn
			Reference: KO131	koreangenome.org
D_4	*Homo sapiens*	24	Target: KO224	koreangenome.org
			Reference: KO131	koreangenome.org
D_5	*Homo sapiens*	24	Target: hg18	ncbi.nlm.nih.gov
			Reference: KO131	koreangenome.org

Table 2. Performance evaluation of different algorithms using various metrics

		GDC			iDoComp			ERGC			Gain	
Dataset	A.size	R.size	C.time	D.time	R.size	C.time	D.time	R.size	C.time	D.time	GDC	iDoComp
D_1	2987	NA	NA	NA	65 708.47	3157.00	812.02	**7704.75**	**606.12**	**129.69**	NA	88.27%
D_2	2987	31 832.76	5980.40	**40.70**	29 723.53	2192.00	268.97	**9016.41**	**840.56**	124.15	71.68%	69.67%
D_3	2987	37 154.19	6785.40	**40.93**	33 131.26	1857.00	276.47	**9200.24**	**875.30**	118.13	75.24%	72.23%
D_4	2938	11 851.95	4829.10	**50.09**	7043.82	2534.00	137.45	**5073.44**	**624.48**	225.22	57.19%	27.97%
D_5	2996	NA	NA	NA	209 380.79	3040.00	953.06	**19 396.40**	**988.12**	**148.32**	NA	90.73%

Best values are shown in bold face. A.size and R.size refer to actual size in MB and reduced size in KB, respectively. C.time and D.time refer to the compression time and decompression time in seconds, respectively.

Table 3. Chromosome-wise performance evaluation of different algorithms using various metrics on dataset D_2

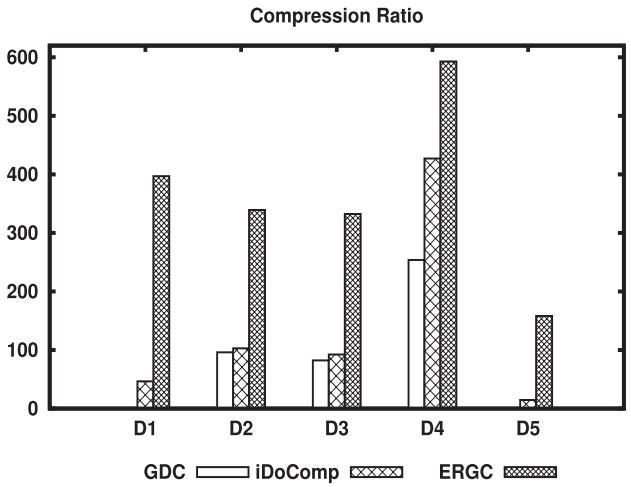
		GDC			iDoComp			ERGC		
Chromosome	A.size	R.size	C.time	D.time	R.size	C.time	D.time	R.size	C.time	D.time
C_1	251 370 554	2 391 974	720.8	**3.19**	2 366 447	200.00	23.09	**700 785**	**57.81**	8.78
C_2	247 000 341	2 345 540	755.90	**3.29**	1 822 538	175.00	14.62	**685 102**	**49.02**	8.51
C_3	202 826 864	1 266 255	368.20	**2.72**	1 124 827	164.00	11.11	**612 271**	**38.30**	8.18
C_4	194 460 954	1 485 973	281.50	**2.66**	1 305 098	140.00	11.12	**626 786**	**44.44**	7.03
C_5	183 872 170	1 575 619	353.60	**2.45**	1 364 733	117.00	10.56	**545 591**	**36.56**	7.41
C_6	173 748 332	1 260 446	327.00	**2.33**	1 087 335	117.00	9.76	**555 871**	**38.97**	6.82
C_7	161 468 454	1 893 681	413.40	**2.14**	1 495 373	108.00	10.27	**512 645**	**30.61**	6.79
C_8	148 712 746	1 224 754	198.50	**1.91**	1 056 722	96.00	8.93	**468 036**	**30.25**	5.82
C_9	142 611 146	2 258 374	232.40	**1.76**	2 606 197	82.00	19.51	**421 259**	**28.04**	5.66
C_{10}	137 630 990	1 427 667	304.40	**1.83**	1 295 147	106.00	9.18	**413 975**	**28.16**	5.66
C_{11}	136 693 265	1 127 511	218.40	**1.79**	935 087	96.00	7.84	**418 994**	**28.18**	5.91
C_{12}	134 555 367	901 759	223.20	**1.79**	760 267	87.00	6.99	**406 557**	**27.36**	5.74
C_{13}	116 045 370	682 679	63.30	**1.40**	858 431	86.00	13.92	**302 716**	**22.28**	4.61
C_{14}	108 141 402	808 036	79.10	**1.29**	931 770	70.00	13.78	**293 895**	**19.69**	4.30
C_{15}	102 011 238	1 130 386	96.00	**1.17**	1 275 730	61.00	14.75	**251 420**	**18.92**	4.31
C_{16}	90 307 716	1 260 031	175.20	**1.13**	1 157 411	60.00	10.08	**284 209**	**17.95**	3.58
C_{17}	80 087 662	870 827	286.20	**1.10**	677 073	51.00	4.58	**245 816**	**16.21**	3.43
C_{18}	77 385 780	560 056	55.00	**1.03**	447 788	47.00	4.27	**236 401**	**16.15**	3.15
C_{19}	64 875 186	546 966	128.30	**0.84**	577 941	39.00	7.00	**276 688**	**12.87**	2.66
C_{20}	63 476 571	414 322	50.20	**0.84**	384 499	52.00	4.38	**199 830**	**12.92**	2.44
C_{21}	47 726 736	318 750	12.60	**0.50**	445 831	46.00	8.44	**141 432**	**13.73**	2.10
C_{22}	50 519 630	573 318	28.00	**0.56**	685 738	36.00	9.88	**149 413**	**9.13**	2.10
C_X	157 495 656	3 628 432	584.60	**2.35**	2 824 069	123.00	12.76	**351 252**	**107.16**	6.85
C_Y	58 735 843	2 643 399	**24.60**	**0.63**	2 950 851	33.00	22.15	**131 863**	135.85	2.32

A.size and R.size refer to actual size and reduced size in bytes, respectively. C.time and D.time refer to the compression time and decompression time in seconds, respectively. Best results are shown in bold.

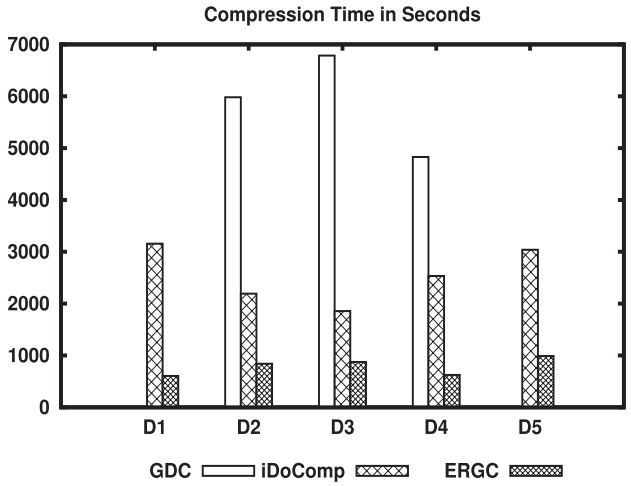
Table 4. Chromosome-wise performance evaluation of different algorithms using various metrics on dataset D_3

		GDC			iDoComp			ERGC		
Chromosome	A.size	R.size	C.time	D.time	R.size	C.time	D.time	R.size	C.time	D.time
C_1	251 370 554	2 721 204	762.70	**3.20**	2 586 013	154.00	23.78	**697 595**	**57.75**	8.74
C_2	247 000 341	2 696 425	780.80	**3.24**	2 051 129	127.00	15.28	**691 698**	**48.14**	8.58
C_3	202 826 864	1 534 808	386.50	**2.74**	1 294 797	118.00	11.61	**618 221**	**37.80**	7.66
C_4	194 460 954	1 782 402	338.20	**2.69**	1 493 618	112.00	11.42	**645 704**	**43.58**	6.90
C_5	183 872 170	1 826 898	421.50	**2.49**	1 536 335	119.00	11.40	**544 582**	**37.17**	6.83
C_6	173 748 332	1 496 369	368.60	**2.34**	1 235 131	109.00	10.02	**557 506**	**37.74**	6.66
C_7	161 468 454	2 175 785	541.60	**2.15**	1 685 493	83.00	10.84	**516 224**	**30.36**	6.53
C_8	148 712 746	1 420 008	232.90	**1.99**	1 187 419	75.00	9.12	**465 201**	**29.79**	5.70
C_9	142 611 146	2 435 967	295.80	**1.76**	2 801 794	76.00	19.91	**427 900**	**27.46**	5.26
C_{10}	137 630 990	1 615 075	338.10	**1.81**	1 432 409	92.00	9.42	**424 362**	**27.56**	5.23
C_{11}	136 693 265	1 329 807	225.60	**1.85**	1 059 181	85.00	8.19	**429 965**	**28.16**	5.21
C_{12}	134 555 367	1 086 044	288.90	**1.76**	870 113	101.00	7.20	**414 576**	**26.93**	4.97
C_{13}	116 045 370	820 625	72.10	**1.42**	948 002	59.00	14.24	**305 009**	**21.84**	4.29
C_{14}	108 141 402	926 154	89.20	**1.26**	1 007 829	76.00	13.81	**308 622**	**19.97**	4.02
C_{15}	102 011 238	1 242 272	131.50	**1.19**	1 362 715	57.00	14.71	**272 473**	**19.22**	3.92
C_{16}	90 307 716	1 393 463	232.50	**1.13**	1 253 762	51.00	10.15	**286 410**	**17.51**	3.53
C_{17}	80 087 662	1 003 297	258.00	**1.10**	757 520	63.00	4.78	**250 363**	**15.98**	3.15
C_{18}	77 385 780	670 979	71.90	**1.05**	512 016	53.00	4.35	**235 380**	**15.80**	3.11
C_{19}	64 875 186	655 576	143.70	**0.83**	644 870	29.00	7.07	**274 639**	**12.95**	2.56
C_{20}	63 476 571	486 193	62.10	**0.83**	429 085	37.00	4.36	**202 068**	**12.73**	2.42
C_{21}	47 726 736	376 979	13.70	**0.50**	475 821	22.00	8.41	**142 895**	**13.84**	2.19
C_{22}	50 519 630	632 996	28.10	**0.56**	730 299	34.00	9.93	**167 700**	**9.32**	2.09
C_X	157 495 656	4 862 027	683.40	**2.40**	3 503 378	**108.00**	14.22	**402 636**	120.93	6.23
C_Y	58 735 843	2 854 541	**18.00**	**0.64**	3 067 689	17.00	22.25	**139 320**	162.77	2.35

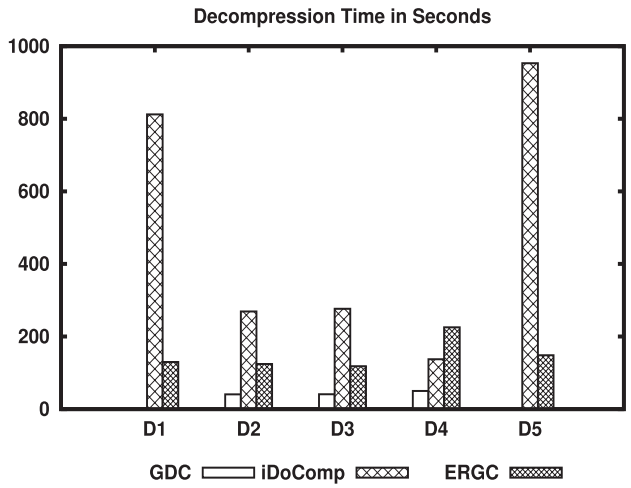
A.size and R.size refer to actual size and reduced size in bytes, respectively. C.time and D.time refer to the compression time and decompression time in seconds, respectively. Best results are shown in bold.



(a) Compression ratios offered by different algorithms.



(b) Compression times taken by different algorithms.



(c) Decompression times taken by different algorithms.

Fig. 2. Comparative statistics of compression ratios, compression and decompression times observed in our experiments. Compression ratio is the ratio of the uncompressed data size to the compressed data size. Compression and decompression times shown are in seconds

compresses 2987 MB of data (i.e. the total size of 24 chromosomes and header information/metadata) roughly to 64 MB where ERGC reduces it to roughly 7 MB of data. The per cent improvement ERGC achieves with respect to iDoComp is 88.27%. Specifically, ERGC compresses $9\times$ better than iDoComp for this particular dataset. ERGC is also faster than iDoComp in terms of both compression ($5\times$) and decompression ($6\times$) times (see Table 2 for details).

Now consider dataset D_2 where the target and reference sequences are from YH and KO224, respectively. The compressions achieved by GDC and iDoComp are roughly equal, whereas ERGC is about $3\times$ better than them. GDC's compression time is longer than both of iDoComp and ERGC, but it decompresses the sequences very quickly. ERGC's compression is approximately $2.5\times$ and $7.5\times$ faster than iDoComp and GDC, respectively. Next consider D_5. GDC runs indefinitely for this dataset. The percentage improvement ERGC achieves with respect to iDoComp is 90.73%. Specifically, ERGC takes $11\times$ fewer disk space compared to iDoComp for this particular dataset. ERGC is also faster than iDoComp in terms of both compression ($2\times$) and decompression ($6\times$) times. Figure 2 shows a comparative study of different algorithms including ERGC with respect to compression ratio, compression and decompression time.

In brief, the minimum and maximum improvements observed from datasets $D_1 - D_5$ were 27.97% and 90.73% with respect to iDoComp, respectively. The minimum and maximum improvements over GDC observed were 57.9% and 75.24%, respectively. ERGC compresses at least $2.12\times$ and at most $5.21\times$ faster than iDoComp. Although it is better than iDoComp and GDC in compression time for every dataset, it is slower than GDC with respect to decompression for datasets $D_2 - D_4$.

5 Conclusions

Data compression is a very important problem in biology especially for NGS data. Five different NGS data compression problems have been identified and studied. In this article, we have presented a novel algorithm for one of these problems, namely, reference-based genome compression to effectively and efficiently compress genomic sequences. From the experimental results, it is evident that our algorithm indeed achieves compression ratios that are better than those of the currently best known algorithms. The compression time is also better than that of state-of-the-art algorithms in this domain. Although GDC is better than ERGC in terms of decompression time, the time ERGC takes to decompress the genomic sequences is also very promising.

Funding

This research has been supported in part by the NIH grant R01-LM010101 and the NSF grant 1447711.

Conflict of Interest: none declared.

References

Ahn,S.-M. *et al.* (2009) The first Korean genome sequence and analysis: full genome sequencing for a socio-ethnic group. *Genome Res.*, **19**, 1622–1629.

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 2007 IEEE Data Compression Conference (DCC 07), IEEE, pp. 43–52.

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

Deorowicz,S. and Grabowski,S. (2011) Robust relative compression of genomes with random access. *Bioinformatics*, **27**, 2979–2986.

Deorowicz,S. *et al*. (2013) Genome compression: a novel approach for large collections. *Bioinformatics*, **29**, 1–7.

Fritz,M.H.-Y. *et al*. (2011) Efficient storage of high throughput DNA sequencing data using reference-based compression. *Genome Res.*, **21**, 734–740.

Golomb,S.W. (1966) Run-length encodings. *IEEE Trans. Inf. Theory*, **12**, 399–401.

Huffman,D. (1952) A method for the construction of minimum-redundancy codes. In Proceedings of the Institute of Radio Engineers, pp. 1098–1101.

Levy,S. *et al*. (2008) The diploid genome sequence of an Asian individual. *Nature*, **456**, 60–66.

Moffat,A. (1990) Implementing the PPM data compression scheme. *IEEE Trans. Commun.*, **38**, 1917–1921.

Ochoa,I. *et al*. (2014) iDoComp: a compression scheme for assembled genomes. *Bioinformatics*, **31**, 626–633.

Pavlichin,D. *et al*. (2013) The human genome contracts again. *Bioinformatics*, **29**, 2199–2202.

Peter,E. (1975) Universal codeword sets and representations of the integers. *IEEE Trans. Inf. Theory*, **21**, 194–203.

Pinho,A.J. *et al*. (2012) GReEn: a tool for efficient compression of genome resequencing data. *Nucleic Acids Res.*, **40**, e27.

Shanika,K. *et al*. (2011) Optimized relative lempel-ziv compression of genomes. In: Reynolds,M. (ed.) 34th *Australasian Computer Science Conference*, Perth, Australia, **Vol. 113**, pp. 91–98.

Wang,C. and Zhang,D. (2011) A novel compression tool for efficient storage of genome resequencing data. *Nucleic Acids Res.*, **39**, e45.

Ziv,J. and Lempel,A. (1977) A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory*, **23**, 337–343.