Iterative Dictionary Construction for Compression of Large DNA Data Sets

Shanika Kuruppu, Bryan Beresford-Smith, Thomas Conway, and Justin Zobel

Abstract—Genomic repositories increasingly include individual as well as reference sequences, which tend to share long identical and near-identical strings of nucleotides. However, the sequential processing used by most compression algorithms, and the volumes of data involved, mean that these long-range repetitions are not detected. An order-insensitive, disk-based dictionary construction method can detect this repeated content and use it to compress collections of sequences. We explore a dictionary construction method that improves repeat identification in large DNA data sets. Our adaptation, COMRAD, of an existing disk-based method identifies exact repeated content in collections of sequences with similarities within and across the set of input sequences. COMRAD compresses the data over multiple passes, which is an expensive process, but allows COMRAD to compress large data sets within reasonable time and space. COMRAD allows for random access to individual sequences and subsequences without decompressing the whole data set. COMRAD has no competitor in terms of the size of data sets that it can compress (extending to many hundreds of gigabytes) and, even for smaller data sets, the results are competitive compared to alternatives; as an example, 39 *S. cerevisiae* genomes compressed to 0.25 bits per base.

Index Terms—Dictionary construction, compression, DNA, large data sets.

1 Introduction

The increasing utilization of new sequencing technologies is leading to changes in the kinds of genetic data that are being gathered and stored. The Human Genome Project (HGP) produced a consensus sequence for much of the human genome, while similar work produced reference DNA data for other organisms. Recently, there has been a shift toward producing data that represent the sequences of individuals. In addition to the original HGP genome, there are now sequences for James Watson [1], two men of Nigerian [2] and Chinese [3] descent, and five southern African genomes [4], among many others. The falling cost of high-throughput sequencing is enabling more ambitious activities such as the 1000 Genomes Project,[1] which aims to determine the variations in the human population by analyzing the genomes of at least 1,000 individuals; and the Personal Genomes Project,[2] which aims to improve the understanding of how genetics and the environment affect human traits, beginning a gradual shift toward personalizing medical treatment.[3] Similar projects are proposed for other human variations, such as cancer genomes, while for other organisms, such as bacteria, the fall in the cost of sequencing is making feasible the production of thousands of genomes within the scope of a single research project.

These activities are leading to massive growth in the size of DNA data sets, and are providing opportunities for novel compression techniques that take advantage of the characteristics of these new data. In particular, the high level of similarity between individuals, which is much greater than the similarities between species, presents new opportunities for compression. Our aim is to identify mechanisms for detecting this redundancy and use it in compression of corpora, while preserving the attractive property that individual items can be fetched in any order.

There are, broadly, three kinds of repetition in DNA. First, there are simple repeats found in long sequences of noncoding regions, such as dinucleotide repeats or long sequences of poly-A. Second, there is repetition of material within a genome introduced by copy number variation and mechanisms such as the existence of reverse complements. Third, there are some strings conserved between individuals and between species. These repeats are not necessarily exact, due to biological effects (that is, mutations), and their representation in a corpus may vary due to sequencing artifacts. Our focus in this paper is on the third kind of repetition.

There is a lack of efficient algorithms for detecting the third kind of repetition across large data sets spanning hundreds to thousands of sequences; hence, our aim is to explore a modeling technique for this task. We show that grammar-based techniques are suitable for exploiting long-range repetitions, with the added advantage of allowing for random access into the data. For the task of storage of DNA corpora, compression based on long-range redundancy can be more effective than methods based on local redundancy. We anticipate that such redundancy will increasingly be

1. http://www.1000genomes.org.
2. http://www.personalgenomes.org/.
3. http://www.everygenome.com.

• S. Kuruppu and J. Zobel are with the Department of Computer Science and Software Engineering, The University of Melbourne, 111 Barry Street, Parkville, VIC 3010, Australia.
E-mail: {kuruppu, jz}@csse.unimelb.edu.au.
• B. Beresford-Smith and T. Conway are with the National ICT Australia, The University of Melbourne, Engineering (Building 193), Parkville, VIC 3010, Australia.
E-mail: {bryan.beresford-smith, tom.conway}@nicta.com.au.

Manuscript received 29 Nov. 2010; revised 18 Mar. 2011; accepted 12 Apr. 2011; published online 27 Apr. 2011.
For information on obtaining reprints of this article, please send E-mail to: tcbb@computer.org, and reference IEEECS Log Number TCBB-2010-11-0262.
Digital Object Identifier no. 10.1109/TCBB.2011.82.

prevalent in genomic repositories, as they are used to store sequences from individuals as well as references for species.

To demonstrate the strength of this approach, we present an algorithm, COMpression using RedundAncy of Dna (COMRAD), for compressing highly redundant large DNA databases. Our starting point was an existing general-purpose compression algorithm, RAY [5], which we adapted to DNA using knowledge about alphabet size and the ways in which strings diverge from each other during evolution. COMRAD hierarchically identifies repeated substrings by repeatedly scanning the collection, which allows construction of a corpus-wide dictionary that represents repetitions of up to hundreds to thousands of bases. Such an approach is not expected to be particularly effective on single sequences or small data sets but should excel on larger data volumes. COMRAD makes no assumptions about the species or degree of similarity, or about the source of the data. Like RAY, COMRAD permits random access and independent decompression of individual sequences.

Our experiments demonstrate that COMRAD efficiently produces excellent compression results over large data sets, just by using exact repeats and reverse complements (unlike existing algorithms that use approximate matching [6], [7], [8], [9], [10], [11]). COMRAD was able to compress the (relatively nonredundant) human genome faster than previous methods, while nearly maintaining compression effectiveness; to compress 39 yeast genomes to 0.23 bits per base (bpb); and to compress a highly redundant data set of approximately 64 GB (consisting of 1,023 artificial variants of human chromosome 20) to about 0.04 bpb. In our initial implementation, as reported here, this last data set took around two days on a single-processor machine. Note that this is by far the largest genomic data set compressed in the literature to date. Although COMRAD currently does not provide the best compression results on low-redundancy data, it is still competitive; on high-redundancy data or large volumes of data, COMRAD is the only approach that is practical.

We next discuss COMRAD in the context of existing research in the area of DNA compression algorithms and some general-purpose compression algorithms. The description of our approach (Section 3), the COMRAD algorithm (Section 4), and experimental results (Section 5) are followed by a discussion of the parameters and their effect on compression in Appendix D, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TCBB.2011.82.

2 Background

All compression algorithms exploit redundancy to achieve space savings, but vary greatly in how they do so. LZ77 [12] and PPM [13] are two of the best known families of lossless compression methods. These approaches code strings sequentially, at each step checking to see if the current substring has been seen previously and, if so, encode it by reference to the previous occurrence. A sliding window is maintained over recently observed text; once text has left this window, it cannot be used in encoding.

LZ77 has formed the basis for several of the best known compression utilities, such as `gzip` [14]. These implementations take advantage of localized repetition of data, and produce representations that require the data to be sequentially decompressed. They are highly effective in the presence of proximate repetitions but do not detect the repetitions that occur between instances in a corpus of multiple similar genomes, and thus do not exploit the most significant form of redundancy occurring in genome data. Alternative implementations could detect repeats that are further apart, as is achieved by `7-zip`, but it is impractical to have an unlimited window size and the data must be memory resident. Also, the method does not allow for access to individual sequences in the compressed data.

Since the first and often second kinds of repetition in DNA can be compressed in this manner, variations of the LZ algorithm have been used in DNA compression algorithms. There are two broad kinds of repetition detection for DNA. Exact-repeat-finding algorithms use precisely duplicated regions, as they are relatively easy to detect and code. Approximate-repeat-finding algorithms forgo this efficiency in order to find longer near-duplicated sequences that lead to better compression results.

DNA compression based on exact repeats was introduced by Grumbach and Tahi [15] with the `Biocompress` algorithm, which showed the possibility of compressing DNA better than the existing general-purpose compression algorithms. The algorithm is LZ-like, where the section of the input data set that was already encountered is stored in a 4-ary tree of height h, with the leaves containing the position of the substring from the root to the leaf. When encoding, the tree is checked to see if the current substring was seen before. Due to the storage of the tree structure in main memory, the algorithm is not suitable for large data sets. However, this research led the way to many other DNA compression algorithms, such as `Cfact` [16] and `Off-line` [17].

`Cfact` operates in two phases where in the "*parsing phase*," a suffix tree is used to identify the longest repeated substrings. In the "*encoding phase*," the nonrepetitive and the first occurrence of repetitive substrings are encoded with two bits per base. The remaining repeated substrings are encoded as pointers in the form of *(pos, len)* tuples. While a suffix tree is an elegant method of identifying longest repeats, it won't be possible to store a large data set in a suffix tree format in memory. We could not locate a table of results for the algorithm.

`Off-line` follows a similar approach to `Cfact`, with a suffix tree used to identify exact repeated substrings but unlike `Cfact`, it encodes most frequent nonoverlapping substrings and uses an augmented suffix tree (reducing the asymptotic cost to $O(n \log^2 n)$ from $O(n^2)$). `Off-line` results are not better than the naïve encoding (1.97 bits per nucleotide) and are worse than algorithms such as `Biocompress`. However, it is a general-purpose compressor and can be applied to many other types of data sets.

Approximate repeat detection algorithms, beginning with `GenCompress` [8], followed by `DNACompress` [9], `DNAPack` [6], and `GeNML` [18], showed that even better results can be achieved by exploiting the approximate nature of the repeated regions in DNA sequences.

GenCompress encodes approximately repeated substrings in an LZ manner using a "relatedness" measure invented by the authors. Using a single pass, if a substring is found that matches an already seen substring, the new substring is encoded with the position and length for the exact repeated regions and edit information for the approximate parts. The results were compared to Biocompress-2 and Cfact, which showed a significant improvement in compression and confirmed the intuition that considering approximate repeats gives better compression results. On the other hand, approximate repeat detection is time consuming and our experiments confirm that the algorithm is infeasible for compressing large data sets (Section 5).

DNACompress finds all approximate repeats in the first pass through the data (using the homology search tool PatternHunter) and, in the second pass, the repeats and nonrepeats are encoded. DNACompress was compared to GenCompress and CTW + LZ and it was able to compress much larger data sets in a faster time without any detriment to the compression rates. Unfortunately, our analysis of the algorithm did not produce such results, as discussed in Section 5.

DNAPack uses a dynamic programming algorithm to find approximately repeated substrings, but with several optimizations so that the runtime is reasonable for long sequences. Compared to Biocompress, GenCompress, CTW + LZ, and DNACompress, the compression performance of DNAPack is better or comparable. Given that dynamic programming can be $O(n^2)$ time, this algorithm has the potential to be very expensive for large data sets, but a nongreedy approach of this kind could lead to better compression rates. However, no runtime results are given in the paper and an implementation could not be found for further analysis.

The GeNML algorithm introduced an alternative approach to compression by encoding blocks of DNA sequences using a *maximum likelihood model*. The algorithm produced better compression results than most of the existing algorithms. It was also the first to compress a large genome such as the *H. sapiens* genome but it is very slow (taking 3.5 hours on a cluster of 12 workstations [18]).

PPM [13], the other major family of lossless sequential compression methods, forms the basis of the DNA compression algorithms CDNA [10], CTW + LZ [11], and XM [7]. The PPM algorithm determines the probability distribution of the next input symbol adaptively, based on all previous symbols observed, potentially considering the context in which those symbols were found.

CDNA is one of the first algorithms to use statistical compression and approximate repeat detection for DNA sequences. The probability distribution of each nucleotide is predicted by approximate partial matching of the current context to previously seen substrings using a Hamming distance to measure the similarity. The results compare well to Biocompress-2.

CTW + LZ uses two methods to compress the input sequences. For long repeats, an LZ77-type encoding is used. For shorter repeats and nonrepeats, a CTW encoding (an algorithm similar to PPM that uses a weighting of multiple models to predict the next symbol probabilities) is used. The algorithm also uses a dynamic programming calculation to search for approximate repeats and approximate reverse complements. This is one of the first algorithms to combine substitutional and statistical compression. However, it is too computationally expensive, as was shown in a later analysis by the authors of DNACompress [9].

XM is based on PPM but with multiple "experts" making recommendations on the symbol probabilities. An example of an expert is an order-k Markov model that gives a probability of a symbol based on the k preceding symbols; another is a copy expert that gives a probability based on whether the next symbol is part of a copied region. XM yielded the best compression results compared to other algorithms and was able to compress a human genome and achieve very good compression results. However, the algorithm is slow for compressing large data sets—in our experience about four days to compress the human genome. A further analysis of XM in comparison to COMRAD is in Section 5.

Given the small size of data sets used for experiments, the papers discussed above have largely neglected memory usage: the largest prior to GeNML is a 4 MB *E. coli* data set used by DNACompress. As the size of available data sets gets large, it is no longer possible to neglect memory usage. Moreover, these previous algorithms do not have promising runtimes; GenCompress required almost half an hour to compress a 4 MB file.

Recent algorithms have been designed for compressing larger data sets of sequences from the same species, where there is a high level of similarity between the sequences. One such algorithm compresses a single genome to the size of an "e-mail attachment" relative to the reference sequence by efficiently representing the differences in each genome compared to the reference, which are in the form of SNPs, insertions and deletions (indels)[4] [19], and is intended for compressing genomes from the 1000 Genomes project.

The "attachment" algorithm compresses a human genome expressed in terms of SNP and indel information, with regard to the reference sequence and a database of known SNPs. The SNPs in the genome that are annotated in the database are encoded using a bit array. Each novel SNP is represented as a position and nucleotide. Each insertion is represented as a pair comprising of a position and a sequence of nucleotides; a deletion as a position and length. Simple encodings (delta codes, Huffman codes) are used for the numbers and nucleotides. The variation data provided by the software for the James Watson genome are 1.97 GB uncompressed and, after compression, the variation data can be represented in 4.2 MB. The method required 4.47 GB of data to be available, comprised of the reference genome and a database based on dbSNP build 129.

The "attachment" method can be effective for compressing variation data for intact human genomes, but is limited to sequences where the dissimilarities are predictable and limited. It would not be effective, for example, for deranged genomes such as those found in cancer, which can contain hundreds of megabases of arbitrarily repeated

4. Single Nucleotide Polymorphisms (SNPs) are single nucleotide changes observed in individual genomes within the same species, which can contribute to the distinguishing features of individuals. Insertions and deletions are substrings of DNA that are present or absent in one genome when compared to another genome, respectively.

or rearranged material. That is, its effectiveness is due to use of a highly specific model of the data that is anticipated, and, in contrast to a typical pattern-based compressor, it is limited to data that fit a predefined template. It also will be ineffective for arbitrary data sets consisting of sequences from multiple species, which may not necessarily be complete genomes, or may even be unassembled contigs, or for single-species data sets where SNP and indel information or a reference sequence is not available. Finally, compressing the variations for a given genome is not the same as compressing the full assembled genome, since mapping the variations to the reference often does not provide the full assembled sequence. While efficient storage of variation data is important, and in the case of human genomes is addressed by Christley et al., our aim in this paper is to compress arbitrary DNA sequences, an entirely different task.

Another method by Brandon et al. [20] uses a similar technique but is general enough to add further data sets and to select a reference sequence (or a set of references) that is suitable for the data set. The results for this method are also promising. However, it is necessary to obtain a list of variations for any arbitrary sequence with respect to a reference, which requires full alignment of the sequence to the reference sequence.

Another recent class of algorithms uses Burrows-Wheeler self-indexing to store and retrieve sequences from repetitive sequence data sets [21], [22], [23]. The method is one of the most general solutions for managing large repetitive sequence data sets, and allows efficient random access and search on the compressed data. However, there are still drawbacks: in particular, the need to store the whole compressed data set and the additional indexing information in memory, which is not scalable to large data sets. It is also not a good representation for data sets containing arbitrary sequences with a significant amount of repetition, as we show later in our comparison of COMRAD to `RLCSA` (one of the implementations described in [21]) in Section 5.

More recently, two grammar-based compression algorithms were proposed by Claude et al. [24] that efficiently return all the positions in the data set containing a given q-gram (q up to 6). The first algorithm uses an LZ77 method to encode a posting list of q-gram positions and uses RE-PAIR [25] to compress the text, while the second algorithm uses a context-free grammar created by RE-PAIR and represents it as a Straight Line Program [26]. While neither achieved the best compression results, the first was competitive for values of q up to 6 and the second works well for highly repetitive data sets.

LZ77-based indexing techniques were explored by Kreft and Navarro [27] in the `LZ-End` algorithm and by Kuruppu et al. [28], [29] in the `RLZ` algorithm. The `LZ-End` algorithm conducts a traditional LZ parsing on the input text but with one exception. Assuming that the input $T_{0,i-1}$ is encoded so far as a series of phrases $Z[0]Z[1]\dots Z[p-1] = T_{0,i-1}$, where each $Z[i]$ for $0 \leq i \leq p-1$ is a substring represented by a phrase i, then the next phrase in the encoding is $Z[p]$, which is the longest prefix in $T_{i,n-1}$ that is also a suffix of one of the existing phrases $Z[i]$, for $0 \leq i \leq p-1$. This restriction ensures that all references to earlier occurrences of a substring end at only certain positions in the text, making random access to substrings in the compressed text efficient. The algorithm is a self-index that supports efficient querying on the compressed text. `RLZ` is also an LZ77 technique but is less general than `LZ-end` in the sense that it is specific for DNA data and the sequences in the data set must have highly similar subsequences (that is, from the same species). `RLZ` compresses each sequence in the data set against a chosen reference sequence using an LZ77 parsing. Efficient random access into the compressed data set is also supported. The algorithm provides good compression and access results while compression and decompression are very fast and memory efficient. COMRAD is compared to `RLZ` in Section 5.

There are two important features that are not observed in most existing DNA compression algorithms. First is the ability to detect repetition over a long range to compress large data sets in the gigabyte to terabyte scale. Another is to allow random access into the compressed data set. This would improve decompression times since it eliminates the need to decompress the whole data set in order to access the sequences.

In the following sections, we review `RAY`, on which COMRAD is based, and our modifications that reduce costs for DNA compression; and then give our algorithm in detail and show experimental results. Detailed exploration of other algorithms is also reported in the results.

3 Approach

COMRAD is based on an existing algorithm, `RAY` [5], a general-purpose compression algorithm for arbitrary byte sequences. The `RAY` algorithm [5] allows random access so that specific sections of the compressed data can be individually accessed.

`RAY` makes multiple passes over the input data. These passes incrementally construct a dictionary of symbol sequences that are repeated in the text, with each pass discovering longer sequences. In the first pass (step 1 in Fig. 1), the dictionary records the frequency counts of all adjacent pairs of symbols. The counts are used to identify which pairs of symbols have frequency exceeding the threshold f; these are the only ones eligible to be replaced. During the second pass (step 2 in Fig. 1), symbol pairs that could be replaced are identified and a count of all the replacements that could be made is recorded. This step is necessary to distinguish between the number of occurrences of a substring from the number of times it could be replaced. Then (step 3 in Fig. 1), the symbol pairs with counts of at least two from step 2 are selected to be replaced so they are added to the dictionary in the form of "$\mathrm{A} \leftarrow \mathrm{bc}$," where the symbol pair is now represented by a nonterminal symbol. Then, all occurrences of the selected symbol pairs are replaced with the nonterminals that represent them in another pass through the data. In the final step (step 4 in Fig. 1), the dictionary is updated to be consistent with the compressed string. Steps 2-4 are repeated until a terminating condition is satisfied, which could either be that a predetermined number of passes has been exceeded or that the dictionary no longer contains any

Input
aabcbcaabcabc

Step 1
aa:2 ab:3 bc:4 cb:1 ca:2

Step 2
aabcbcaabcabc (*no candidate*)
a**abc**bcaabcabc (*no candidate*)
aa**bcb**caabcabc (*cand cnt bc: 1*)
aabc**bca**abcabc (*cand cnt bc: 2*)
aabcbc**aab**cabc (*no candidate*)
aabcbca**abc**abc (*no candidate*)
aabcbcaa**bca**bc (*cand cnt bc: 3*)
aabcbcaabc**abc** (*no candidate*)
aabcbcaabca**bc** (*cand cnt bc: 4*)

Step 3
bc $\rightarrow$ A
aaAAaaAaA

Step 4
aa:2 aA:3 AA:1 Aa:2 bc:1
...
aA $\rightarrow$ B
aBAaBB
...
aB $\rightarrow$ C
CACB

Fig. 1. Compression of the string $aabcbcaabcabc$ using RAY. The frequency threshold $f = 2$. **Step 1** Create frequency dictionary of symbol pairs. **Step 2** Determine the symbol pairs that could be replaced (candidates) by scanning through triplets and if the leftmost pair has a higher frequency than the rightmost pair, and the count of the leftmost pair is at least f, then increment candidate count of the leftmost pair by 1. This is to calculate the actual number of replacements that can be made for a pair of symbols with a count of at least f. Increment candidate count of bc by 1. Continue comparison of symbol pairs till the end of the string. **Step 3** bc has the only candidate count above 1 so create a new symbol A to represent it. Replace all occurrences of bc in the input by A to obtain $aaAAaaAaA$. **Step 4** Update the frequency dictionary to be consistent with the new string. Keep bc since it occurs in the rule $bc \rightarrow A$. Continue the algorithm to produce the final compressed string $CACB$.

symbol pairs with a frequency above the threshold f. An example of the algorithm is in Fig. 1.

Once the dictionary has been built, the data and dictionary can be coded using mechanisms such as Huffman coding, which in this application are reasonably efficient due to the fact that most of the occurrences of the most common symbols have typically been removed, reducing their frequency and thus reducing the difference between the 0-order entropy and whole-bit coding that can result from a Huffman process.

SEQUITUR [30] and RE-PAIR [25] are two other algorithms that are similar in nature to RAY, where the focus is on efficiently determining a good dictionary that captures the repetition in the data set to be compressed. SEQUITUR parses the input into a context-free grammar and encodes the top-level rule and hierarchy of rules efficiently. RE-PAIR at each iteration counts the frequency of pairs of symbols, but, unlike RAY, which can replace multiple frequent symbol pairs during an iteration, RE-PAIR only replaces the most frequent symbol pair. The algorithm can compress in $O(n)$ time.

Given that RAY is a general-purpose compression algorithm, it is possible to use RAY for compressing DNA data sets; indeed, the authors of RAY tested their algorithm on an *E. coli* DNA data set. The 4.53 MB data set was "compressed" to 2.12 bpb, not a significant achievement. Due to the unavailability of a RAY implementation, we were unable to test its performance, but it is realistic to assume that it would achieve compression similar to COMRAD, though at significantly greater computational cost.

We modify RAY such that it is efficient for DNA compression. First, we define a symbol to be a substring of, say, 16 consecutive symbols in the first iteration (first execution of steps 1-4 in RAY). The motivation here is that almost all substrings of length 16 or 16-mers occur with reasonable frequency in large DNA collections, and thus naïve RAY would in four iterations discover the vast majority of these 16-mers as dictionary entries. By starting at this length, we save numerous passes through the data. The impact on compression for varying this starting length is explored in Appendix D, which can be found on the Computer Society Digital Library.

Second, in the later iterations, RAY seeks only patterns of the form "ab," where "a" and "b" are any symbols, terminal or nonterminal. Here, again to reduce the number of passes, we recognize patterns such as "$\mathcal{AB}$," "$\mathcal{A}x_1\mathcal{B}$," and "$\mathcal{A}x_1x_2\mathcal{B}$," where $\mathcal{A}$ and $\mathcal{B}$ are nonterminals, and x_i is a terminal symbol. This allows the algorithm to combine already encoded symbol pairs as well as to capture patterns in the neighborhood of existing encoded regions. Note that, in the second and subsequent iterations, it is only necessary to consider sequences that contain a nonterminal; all other repetitions are captured in the first iteration.

Third, we should allow for reverse complements, a property that is specific to DNA. Consideration of reverse complements was first proposed by Grumbach and Tahi [15], and most subsequent DNA compressors also implement reverse complement detection, since it is a common form of repetition found in DNA data sets. We consider reverse complements when populating the frequency dictionaries and when replacing repeated segments of nucleotides with a nonterminal. This gives COMRAD greater compressive power than RAY for DNA.

The above changes and a slight modification to the identification of candidates form the basis for our implementation of COMRAD. We use our implementation to show that this algorithm can provide a good modeling technique to achieve good compression results for DNA sequences, and regard it as not a final result, but a test of whether a RAY-like approach is effective in this context. We now explain COMRAD in detail.

4 Methods

The series of nucleotides making up a DNA sequence can be represented as a string of symbols over the alphabet $\sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$, which is the standard DNA alphabet. Sometimes, there are ambiguities in identifying the nucleotides at certain positions, and therefore a nucleotide could either be a A or a C, or a A or a G and so on. To identify all these possible combinations, the extended DNA alphabet is used where all the 15 possible combinations of the standard four nucleotides are given unique symbols. We identify the extended DNA alphabet as $\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}, \mathtt{M}, \mathtt{R}, \mathtt{W}, \mathtt{S}, \mathtt{Y}, \mathtt{K}, \mathtt{V}, \mathtt{H}, \mathtt{D}, \mathtt{B}, \mathtt{N}\}$.

COMRAD is an iterative algorithm for compressing a set of M DNA sequences, $S_0 = \{S_0^1, S_0^2, \dots, S_0^M\}$, where each S_0^i is a sequence over the extended DNA alphabet Σ. Let n_0^i be the length of the sequence S_0^i, and $S_0^i = s_1 s_2 \dots s_j \dots s_{n_0^i}$ where $1 \leq j \leq n_0^i$ and $s_j \in \Sigma$. Also, let $N = \sum_{i=1}^{M} n_0^i$ be the total length of the original data set S_0. The algorithm requires two parameters: the length L of substrings for the first iteration and the minimum frequency threshold F that each substring needs to satisfy in order to be identified as an efficient replacement. Further discussion of choosing the length and frequency parameters is in Appendix D, which can be found on the Computer Society Digital Library.

Each iteration has two main phases: frequency dictionary creation and substitution. The output of the kth iteration is a frequency dictionary D_k, an alphabet Σ_k, and a set S_k of M compressed sequences over the alphabet Σ_k, where $k = 1, 2, \dots, K$ and K is the number of iterations taken to compress the data set. At each iteration, two passes through the input to that iteration are required, one for frequency dictionary creation and the other to do the substitutions. The number of iterations K can be set to a fixed value but by default, the algorithm terminates when no further compression is observed.

The frequency dictionary at each iteration records the frequency counts of selected distinct substrings present in the input and is used to determine which substrings are to be replaced with nonterminal symbols. The algorithm distinguishes between the first iteration, for which the inputs are the original sequences of nucleotides, and subsequent iterations, for which the inputs are sequences containing nucleotides as well as substituted symbols from previous iterations. An outline is shown in Fig. 2.

```
Input:
  1: Set of DNA sequences $S_0$
  2: Iteration 1 substring length $L$
  3: Minimum frequency threshold $F$
  4: Set of patterns $P$
Output:
  1: Compressed DNA sequences $S_K$
  2: Dictionary of symbols $D$
Algorithm:
  1: Create the frequency dictionary $D_1$ of all $L$ length substrings, with frequency of at least $F$, for the input DNA sequences $S_0$
  2: Encode the input sequences $S_0$ to get sequences $S_1$
  3: $k \leftarrow 2$
  4: while the dictionary continues to grow do
  5:     Create the frequency dictionary $D_k$ of all substrings matching patterns in $P$, with frequency of at least $F$, for the input sequences $S_{k-1}$
  6:     Encode the input sequences $S_{k-1}$ to get sequences $S_k$
  7:     $k \leftarrow k + 1$
  8: end while
  9: Cleanup dictionary $D$ to remove infrequent non-terminals and make numbering consecutive
```

Fig. 2. Outline of the COMRAD algorithm.

4.1 First Iteration

4.1.1 Frequency Dictionary Creation

Let D_1 be the frequency dictionary for the substrings of length L that occur in all the sequences S_0^i in S_0. Let $s_{j,j+L}$ be a substring from sequence S_0^i with a length of L, starting at position j and ending at position $j + L - 1$. Also, let $s'_{j,j+L}$ be the reverse complement of the substring $s_{j,j+L}$. The frequency dictionary is constructed as follows: for every substring of length L in each sequence, if the substring or its reverse complement is present in the dictionary, then increment the frequency count. Otherwise, add the new substring to the dictionary with a frequency of 1. Subsequently, all entries in D_1 with a frequency less than F are removed.

4.1.2 Substitution

For each sequence S_0^i in S_0, repeated substrings are substituted with symbols, using the frequency dictionary D_1, independently of the other sequences in S_0, to avoid storing all the sequences in memory simultaneously. Therefore, care must be taken to select a set of replacements for each sequence where the same replacements are likely to be made in the rest of the sequences, preventing the final dictionary from being too large. For this reason, we use the global frequency counts that were determined during dictionary construction.

Beginning from the most frequent substring, if the substring does not overlap with an existing replacement, it is replaced by a unique nonterminal ϕ that represents the substring. If the substring is a reverse complement, an extra symbol r is attached to the end of the unique identifier. Replacements are made until no further substrings with a frequency of F or more remain in the compressed sequences. At the end of the substitution phase, the output is the set of sequences $S_1 = \{S_1^1, S_1^2, \dots, S_1^M\}$ where each S_1^i is a modified DNA sequence that contains a mixture of nucleotides and symbols that identify frequent substrings (nonterminals).

4.2 Second and Subsequent Iterations

From the second iteration and beyond, the algorithm must recognize repeated regions composed of nucleotides and nonterminals. To reduce the number of passes, we have defined the following set of arbitrary patterns P to recognize: $\mathcal{AB}$, $\mathcal{A}x_1\mathcal{B}$, $\mathcal{A}x_1x_2\mathcal{B}$, $\mathcal{A}x_1x_2x_3\mathcal{B}$, $\mathcal{A}x_1x_2x_3x_4\mathcal{B}$, $\mathcal{A}x_1x_2x_3x_4x_5$, and $x_1x_2x_3x_4x_5\mathcal{B}$. In these patterns, $\mathcal{A}$ and $\mathcal{B}$ are nonterminals that represent repeated substrings, and the x_i represent any nucleotide in the extended DNA alphabet. The first pattern allows for pairs of repeated nonterminals to be replaced with a new nonterminal symbol. The next four patterns allow repeated regions that contain short segments of specific nucleotides between nonterminals to be represented. The last two patterns capture any short repeats of length l, where $5 \leq l < L$, that are left over at the right and left ends of the repeat region, since no other patterns can capture these. It is possible to have a customized set of patterns for each iteration, but we use the same set from the second iteration and beyond, since they capture most combinations of terminals and nonterminals. A more detailed discussion of the pattern set is in Appendix D, which can be found on the Computer Society Digital Library.

4.2.1 Frequency Dictionary Creation

The frequency dictionary is created in a similar manner to the first iteration. However, the substrings added to the dictionary follow the patterns defined above. Each substituted input sequence is parsed to detect the above patterns and, if a substring satisfies a pattern, it is stored in the frequency dictionary D_k. Reverse complements are also taken into account as before. Infrequent substrings with a frequency less than F are also removed from D_k.

4.2.2 Substitution

This is similar to the substitution step for the first iteration. Only nonterminals created in the previous iteration are considered in pattern identification.

The algorithm can either be repeated for a predetermined number of iterations or until the frequency dictionary no longer has any patterns with a frequency of at least F. We choose the latter option. The output is a set of compressed DNA sequences $S_K = \{S_K^1, S_K^2, \ldots, S_K^M\}$ and a set of codebooks $D = \{D_1, D_2, \ldots, D_K\}$ that maps nonterminals to substrings.

We have observed that the number of iterations required to construct the dictionary grows slowly with corpus size, with an extra iteration or two for each decimal order of magnitude, but this growth is also determined by the level of redundancy. Coding and decoding costs are linear in collection size.

4.3 Cleanup

Given the interaction between the frequency dictionary creation and the substitution phase, it is possible that some nonterminals with a frequency of less than F remain. As in SEQUITUR [30], any such nonterminals are replaced by their original substrings in both the codebook and the sequences, and the entry for that nonterminal is removed from the codebook.

During this process, some unique IDs may no longer be used leaving gaps in the set of numbers used to identify nonterminals. This means that we would need to store the nonterminal number as well as the substring (right-hand side) of each codebook entry. It would be more space efficient if we could infer the nonterminal identifier from the order in which the substrings are read. For this to be possible, we ensure that all unique IDs are consecutive. A final pass through the codebook and the compressed sequences is undertaken to remap all nonterminal identifiers to be consecutive. An example run of the algorithm is shown in Fig. 1 of Appendix A, which can be found on the Computer Society Digital Library.

4.4 Final Encoding

The COMRAD-compressed sequences and the codebook are canonical Huffman encoded [32] for storage. The sequences and codebook entries consist of nucleotides and nonterminals. A separate tiny dictionary of substrings of length n or n-mers, with n from 1 to 6, was used to aggregate nucleotide sequences into symbol probabilities closer to the probabilities of the nonterminals. This makes the probabilities smaller and hence the Huffman encoding more efficient. The frequencies of the n-mers and the nonterminals are used to determine the code lengths for each symbol (n-mer or nonterminal) to be encoded. The canonical Huffman coding algorithm is used to assign bit representations for each symbol and then the sequences and the codebook are encoded using these bit representations. Further details are explained in Appendix A, which can be found on the Computer Society Digital Library.

The total size of the compressed data set is the size (in bytes) of the encoded codebook, encoded sequence files and the file containing the n-mers, nonterminals, and their codeword lengths.

Huffman encoding was chosen in this initial representation for its speed and simplicity. Experimentally, Huffman encoding is able to give a compressed size that is within 5 percent of the 0-order entropy of the compressed data, due to the fact that most of the symbols have low probability; this result shows that potential further gain is small.

4.5 Compression Cost

The asymptotic cost of the algorithm can be examined by considering the cost at each step of each iteration. In the first iteration, frequency dictionary creation is linear in the length of the original data set since each L-mer needs to be examined to count frequencies. Therefore, the asymptotic cost is $O(N)$. In the substitution step of the first iteration, the asymptotic cost is $O(n_0^i \log n_0^i + 2n_0^i) = O(n_0^i \log n_0^i)$ for each sequence S_0^i of length n_0^i, since two passes through each sequence are required (one to determine the substrings that could be replaced and the other to do the replacements) as well as a sort on the candidate substrings to be replaced.

The analysis is not straightforward for subsequent iterations. The frequency dictionary creation is still linear in the length of the compressed sequences into the iteration and the length is dependent on the number of substitutions made in the previous iterations. Therefore, for each iteration t after the first iteration, the frequency dictionary creation has complexity $O(N_{t-1})$, where $N_{t-1} = \sum_{i=1}^{M} |S_{t-1}^i|$. The substitution step is similarly dependent on the number of substitutions made in the previous iterations, hence has the asymptotic cost $O(n_{t-1}^i \log n_{t-1}^i)$ per sequence.

It is difficult to predict the number of substitutions made at each iteration, since this depends on the repeat properties of the data set to be compressed. Therefore, the space consumption at each step of each iteration is also hard to predict since it is directly proportional to the number of distinct L-mers or pattern substrings in the input to the iteration. These quantities can be predicted by modeling the level of repetition using an appropriate probability distribution but it is unlikely that these assumptions will hold for most real-world data sets. Therefore, we don't explore this path in this paper and simply state that the compression effectiveness and the space consumption of COMRAD depend on the level of repetition and the average length of repeats.

Cost of grammar-based compression algorithms. The substitution algorithm used in COMRAD is a greedy algorithm. Given that COMRAD is a grammar-based compression algorithm, the aim is to generate the smallest grammar that can represent the input sequences. Finding the smallest grammar for a given sequence is an *NP-hard* problem and Charikar et al. [33] found upper and lower bounds for approximation ratios for algorithms that are very similar to COMRAD such as SEQUITUR and RE-PAIR. We do not know the approximation ratios for COMRAD and acknowledge that the current substitution algorithm may not be the best we can achieve.

4.6 Decompression

Decompression follows two steps. The first step is to Huffman decode the compressed sequences. The second step is to decode the COMRAD-compressed sequences. The COMRAD-compressed sequences are stored in memory for decompressing the whole data set or for random access. There are two methods of decompressing COMRAD-compressed sequences, very similar to decompression of RE-PAIR [25].

The first method begins by loading the codebook into memory, reading each compressed sequence at a time into memory, and while conducting a linear pass through the sequence, recursively decode each nonterminal encountered. The recursive nature of this method stems from the fact that COMRAD nonterminals have a hierarchical structure so, for a given nonterminal, the codebook can be recursively accessed until the base sequence consisting only of nucleotides is reached.

However, many of the nonterminals represent very short sequences of further nonterminals, and so this recursive process is not efficient, though it does result in the codebook being reasonably compact. In this initial implementation, we chose to expand the codebook at loading time, so that each codebook entry consists only of nucleotides. While this consumes more space, the overhead is not excessive, and decompression is faster.

4.7 Random Access

COMRAD supports random access to the compressed data using the codebook and the compressed sequences with additional space proportional to the length of each sequence. Random access is described in terms of *display(i,s,e)*, the substring starting at position s and ending at position $e-1$ is retrieved from sequence i.

Let the compressed set of sequences be $S_c = S_c^1, \dots, S_c^M$ and the dictionary of symbols be D. For details of how D and S_c are stored, refer to "Random Access" in Appendix A, which can be found on the Computer Society Digital Library. D requires $d_L L \log|\Sigma| + d_P\mu$ bits of storage where d_L is the number of L-mers in D and d_P is the number of pattern dictionary entries each with an average length of μ symbols (while most patterns are in the form of patterns in set P, some will be longer due to the replacements made in the Cleanup stage). The compressed sequences require $O(N_c)$ space where $N_c = \sum_{i=1}^{M} |S_c^i|$.

Along with S_c and D, the expanded lengths of the pattern substrings also need to be stored, requiring $d_p \log m$ bits where m is the length of the longest expanded dictionary entry (the lengths of L-mer substrings can be inferred). The lengths are calculated recursively in a similar manner to a dynamic programming calculation, so not all nonterminals need to be expanded fully each time a length needs to be calculated.

For each sequence S_c^i, two bit arrays are also stored: bit array B_{seq}^i is a map of the expanded sequence and contains a 1 at any position where the substring starting at that position is the start of a nonterminal, and bit array B_{cseq}^i is a map of the compressed sequence and contains a 1 at any position where there is a nonterminal. B_{seq}^i enables us to obtain the nonterminal that occurred just before the starting position of the query and B_{cseq}^i enables us to obtain the said nonterminal. The bit arrays are stored in a compact data structure that supports the following operations:

- *rank*(A, j) returns the number of 1 bits in bit array A until position j; and
- *select*(A, j) returns the position of the jth 1 bit in bit array A.

The bit arrays are stored in the "sdarray" format of Okanohara and Sadakane [34]. B_{seq}^i can be stored in $t^i \log\frac{N}{t^i} + O(t^i)$ bits and B_{cseq}^i in $t^i \log\frac{N_c}{t^i} + O(t^i)$ bits, where t^i is the number of nonterminals in sequence S_c^i. *rank()* takes $O(\log\frac{|S_0^i|}{t^i})$ time for B_{seq}^i and $O(\log\frac{|S_c^i|}{t^i})$ for B_{cseq}^i, where $|S_0^i|$ is the length of the original ith sequence and $|S_c^i|$ is the length of the compressed ith sequence. *select()* takes $O(1)$ time.

The overall space usage is $d_L \log|\Sigma| + d_P(\log\mu + \log m) + T\log\frac{N}{T} + T\log\frac{N_c}{T} + O(T) + O(N_c)$ bits where $T = \sum_{i=1}^{M} t^i$.

Then, for a given query in the form of (i, s, e), where i is the sequence number ($1 \leq i \leq M$), s is the start position, and $e-1$ is the end position, random access is as follows: the nonterminal that occurred before or at s is found. If a nonterminal did not occur before s, or the previous nonterminal does not include position s, then the next nonterminal is fetched. Then, while there are more nucleotides to be extracted, follow two steps. First, if there are nucleotides before the next nonterminal, then extract those. Second, extract the necessary nucleotides from the nonterminal itself by recursively expanding the dictionary entry for the nonterminal for as many nucleotides as necessary. Increment to the next nonterminal and continue the two steps until all the nucleotides from position s to $e-1$ are extracted. This algorithm is presented in Appendix B, which can be found on the Computer Society Digital Library. The comparison of our *display()* results to that of RLCSA is in Table 5.

For a given query, the *rank()* operation is only required once to determine the first nonterminal in $O(\log\frac{|S_0^i|}{t^i})$ time. Subsequent nonterminals are accessed by incrementing the rank for each new nonterminal. On average, t' nonterminals need to be accessed and expanded per query, where t' is a function of the query length (the higher the query length, the number of nonterminals that cover the substring $S_0^i[s..e]$ is expected to be higher). For each nonterminal, two *select()* queries are required in $O(1)$ time to obtain the position of the nonterminal in the original sequence and the compressed sequence, respectively. Finally, $e-s$ nucleotides need to be extracted per query, some from S_c^i and the remainder from expanding t' nonterminals, with $O(1)$ complexity per nucleotide. Therefore, the asymptotic cost for a random access query is $O(e - s + t' + \log\frac{|S_0^i|}{t^i})$.

5 Results

Our implementation is in C. To reduce memory for the dictionary, the initial substring length L was restricted to 16 for most of the data sets except for the Bacteria and *H. sapiens* data sets, where due to excessive memory requirements, L was restricted to 15. The details are explained in Appendix A, which can be found on the Computer Society Digital Library. For the F parameter, as a default, we have chosen $F = 4$ since, empirically, the cost of introducing a new symbol to represent a substring with a frequency less than 4 is higher than the cost if the substring was not substituted. These parameters were

TABLE 1
COMRAD Compression Results

Dataset	Sequence count	Original size (Mbase(bpb))	Encoded size (Mbytes(bpb))	Iterations	Compress time (hrs)	Decompress time (secs)	Memory usage (Mbytes)
Influenza	78,041	112.64 (1.97)	6.03 (0.43)	11	0.03	18	75
Hemoglobin	15,199	7.38 (2.07)	1.07 (1.16)	11	<0.01	13	50
Mitochondria	1,521	25.26 (1.95)	5.77 (1.83)	9	0.02	13	277
S. cerevisiae	702	485.87 (2.19)	15.29 (0.25)	15	0.19	13	277
S. paradoxus	576	429.27 (2.12)	18.33 (0.34)	15	0.25	23	554
Bacteria	1,446	2770.54 (2.00)	724.54 (2.26)	15	2.41	377	10413
H. sapiens	96	12066.06 (2.18)	2176.06 (1.44)	22	7.90	1666	10737
63xH. sapiens chr22	63	3129.02 (2.28)	26.87 (0.07)	21	1.44	46	806
127xH. sapiens chr1	127	31411.70 (2.24)	296.29 (0.07)	22	48.96	367	6442
1023xH. sapiens chr20	1,023	63841.82 (2.17)	305.21 (0.04)	19	46.15	1193	2159

The columns show the data set name, total number of sequences, original size and original entropy (bits per base), Huffman encoded size (and bits per base), total number of iterations, total time taken to compress, time taken to decompress, and approximate maximum memory usage, respectively. Above the line: real DNA data. Below the line: variants artificially generated from real data.

determined in preliminary experiments. The data sources for the experiments reported here are available at ww2.cs.mu.oz.au/~kuruppu/comrad/. A detailed analysis of the L and F parameters is in Appendix D, which can be found on the Computer Society Digital Library.

As a first step in determining whether COMRAD is effective, we used an Influenza data set of 78,041 sequences to evaluate COMRAD in the presence of high redundancy. The data set was chosen for its small size (112,640,907 bases) and its high repeat content (78,041 sequences with high commonality). As a preliminary analysis, we created a frequency dictionary of all overlapping substrings of length 20, finding that, out of 111,158,128 substrings, only 3,078,772 are unique, where each is repeated approximately 36 times on average.

With the $L = 16$ and $F = 4$ parameters, after 11 iterations, beyond which there was no further compression, COMRAD had compressed the data set to 0.43 bpb, which is the final entropy calculated by dividing the total number of bits required for the compressed sequences (excluding the sequence names) and the codebook, by the total number of bases in the data set. The original entropy of the data set, which is the 0-order entropy of the data set calculated using the probability distribution of the nucleotides in the data set, was 1.97 bpb. The final file size was around 6 MB, which is around 4 percent of the original file size of 113 MB. This result was evidence that COMRAD was successful at detecting redundancy and achieving reasonable compression, as well as providing decompression speeds of around 6×10^6 bases per second. This result is reported in the first line in Table 1.

To assess the improvement in compression rates over each additional iteration, the Influenza data set was analyzed at each of its iterations to obtain the 0-order entropy. Most of the gain is in the first five to six iterations, with only small gains observed thereafter. The first iteration is also the most computationally expensive, taking around 135 seconds, compared to around 30 seconds for the second iteration, 20 seconds for the third, and continuing to markedly decrease thereafter. Note that the number of iterations required to compress a data set is higher for data sets containing long individual sequences (human chromosomes require approximately 20 iterations) compared to data sets containing shorter individual sequences (Mitochondria and Bacteria sequences require less than 10 iterations).

The next few lines in the upper block of Table 1 show the effectiveness of COMRAD on a variety of freely available genomic data sets. Our focus was to develop a method that was suitable for corpora that contain many variants of the same sequence, and thus we chose sets that were of this form. One was 15,199 Hemoglobin sequences, occupying 7.38 MB, which compressed to 1.07 MB or 1.16 bpb. Similarly, we tested sets of mitochondrial DNA sequences, Yeast, and Bacteria. These yielded varying results, from effectively no compression (Bacteria) to a very promising 0.25 bpb (*S. cerevisiae*). For the Bacteria and Mitochondria data sets, the compression is ineffective due to the disproportionately large codebook compared to the compressed sequences. These data sets have many short repeats, and, although COMRAD detects them, the large number of small repeats in the codebook leads to the larger final compressed data set size. Algorithms that don't have an external dictionary such as `7-zip` will be able to compress such data sets much more effectively, whereas data sets such as yeast are better for COMRAD-type algorithms since they contain longer repeats, resulting in a smaller codebook overhead.

The final line of the upper block of Table 1 shows results of compressing four human genomes, consisting of the reference genome (NCBI build 37 release on March 2nd, 2009), Craig Venter genome [35], Chinese genome [3], and the Korean genome [36] that add up to 12,066,063,708 bases. The total compressed file size was 2,176 MB giving a compression rate of 1.44 bpb. The total time taken to compress the genomes was approximately 8 hours, and decompression is around 7 megabases per second. This is not much of an improvement for COMRAD compared to compressing a single human genome (1.82 bpb), due to the codebook being very large (1.69 GB). We believe that with the addition of further human genomes to the data set, longer repeats can be found and the cost of the dictionary will be shared among a larger data set improving the results.

There are not yet many published variants of larger genomes. Given that one of the main aims of COMRAD is to compress data sets in the gigabyte scale, in order to explore the potential performance of COMRAD, we artificially generated variants using a model loosely based on observed differences between human individuals. While such a model does not capture true sequence-to-sequence variation, it does indicate whether our approach is capable of

TABLE 2
Performance of COMRAD on Data Sets Containing Varying Mutation Rates

Dataset Name	Mutation Rate (bpb)	Indel Rate (bpb)	Original Entropy	Final Entropy
32xHsChr22 set 1	1 in 100	1 in 100,000	2.2846	1.2538
32xHsChr22 set 2	1 in 1,000	1 in 1,000,000	2.2820	0.4205
32xHsChr22 set 3	1 in 10,000	1 in 10,000,000	2.2815	0.1272
32xHsChr22 set 4	1 in 100,000	1 in 100,000,000	2.2815	0.0784

Each data set contains 32 leaf sequences of H. sapiens *chromosome 22 generated from the evolution model described with the parameters specified in the Mutation Rate and Indel Rate columns. The columns are the data set name, the single point mutation rate, the indel rate (an indel has a length of 10,000), the 0-order entropy of the 32 sequences before and after compression (both expressed in terms of bits per base), respectively. Note that the rates are a parameter to the model of evolution and the actual level of difference between each of the sequences in a data set is dependent on the evolutionary relationship between a pair of sequences.*

capturing overall sequence similarity and is able to efficiently compress data sets of a size that has not been attempted previously by a DNA compression algorithm.

The variant generation takes a genome as input and creates genomic variations to create two child sequences. Those two child sequences also undergo further variation to create two children each and so forth. This method was used to create several generations of sequences for the *H. sapiens* chromosome 1 (127 variants), chromosome 20 (1,023 variants), and chromosome 22 (63 variants). The model used for the experiments assumed a single point mutation probability of 1 in 10,000 bases, to a randomly chosen base, and an indel probability of 1 in 10,000,000 with an indel length of 10,000 bases.

For the *H. sapiens* chromosome 1 data set, the 31.41 GB of data were compressed to 0.29 GB (0.07 bpb). Compression took approximately 49 hours; decompression ran at about 94×10^6 bases per second. Similar results were observed on the other artificial data sets, with a trend of compression converging to a limit of around a few bytes per mutation. The larger data sets needed more iterations than did the Influenza data set to yield stable compression, with typically most of the gain in the first 9-10 iterations and up to 20 or so iterations to achieve the final result.

In order to analyze the performance of COMRAD when the level of difference between the sequences in the data set changes, further experiments were performed using the artificial variant-generation method. The *H. sapiens* chromosome 22 sequence was taken and five data sets of 63 variants were generated as before but with each data set containing a different mutation and indel rate. Unlike before, only the 32 leaf sequences were kept in each of the data sets to ensure that each sequence is distantly related to most other sequence (sequences 1 and 2 have a common parent, sequences 1 and 3 have a common grandparent, sequences 1 and 5 have a common great grandparent, and so on). The results are shown in Table 2. As expected, the lower the mutation rate, the better the compression results, with the results improving significantly when the rates are reduced 10-fold from the starting rates (from 1.25 bpb down to 0.42 bpb) and the level of improvement diminishing for every subsequent 10-fold reduction in the rates. Even with a 1 in 100 mutation rate, COMRAD was able to almost halve the data set size. (This is a very high mutation rate that would not typically be detected in a data set containing the same species. Even with a mutation rate of 1 in 1,000, if a randomly selected pair of leaf sequences is taken then, they would differ in approximately 1 in 80 bases.)

To better understand incremental compression, we first compressed randomly chosen 22 sequences out of the 32 leaf sequences of the *H. sapiens* chr 22 data set. Then, we continued to add one sequence at a time to the existing data set, recording the compression performance at each addition. The first 22 sequences compressed to 27 MB, the addition of one sequence led to a 285 KB increase, then the addition of another sequence led to a 3.5 MB decrease. In a similar way, addition of further sequences sometimes led to an increase and other times to a decrease, and finally when the 10th sequence was added, the overall size was 25 MB. The addition of certain sequences made the overall compressed size larger, since they contained substrings that weren't present in the original data set. However, most of the time, adding more sequences led to some other substrings getting a high enough frequency and being added to the codebook, hence making the overall compressed size smaller.

Next, we compared COMRAD with the two other algorithms we are aware of that have similar design goals, which is to compress large sets of similar sequences, while supporting random access on the compressed data. RLCSA is a self-index implementation described by Mäkinen et al. [22] and is the only publicly available implementation that we found. RLZ [28] is a DNA compression algorithm where each sequence in the data set is compressed with respect to a chosen reference sequence. We compare the compression results of COMRAD, RLZ (standard algorithm described in [28]), and RLCSA for our test data sets and later in the section, we compare the random access performance of the three algorithms. The compression and decompression results are in Table 3.

For the data sets such as Hemoglobin, Mitochondria, and Bacteria where the sequences in the data set have very little similarity, RLCSA does not perform well, often resulting in the compressed size being larger than the original data set. In this case, COMRAD has an advantage since it makes no assumptions about the data set and, if there are exact repeats in the data, COMRAD will detect them. For the *S. cerevisiae* and *S. paradoxus* data sets, where all the sequences are from the same species, RLCSA performs well but still not as well as COMRAD. COMRAD outperforms RLCSA in terms of compression (except for the Influenza data set), and the speed at which the data are compressed and decompressed, but RLCSA is much more memory efficient than COMRAD. On the other hand, COMRAD does not support *count* and *locate*, and it is difficult to predict how much extra space would be required to support this functionality. RLCSA

TABLE 3
Compression Results of RLCSA, RLZ, and COMRAD

Dataset	RLCSA				RLZ				COMRAD			
	Size (bpb)	Cmp. (sec)	Decmp. (sec)	Mem (Mbyte)	Size (bpb)	Cmp. (sec)	Decmp. (sec)	Mem (Mbyte)	Size (bpb)	Cmp. (sec)	Decmp. (sec)	Mem (Mbyte)
Hemoglobin	2.16	1862	162	6	4.13	5	1	<1	1.16	14	13	50
Influenza	0.43	17042	1977	19	3.10	67	11	<1	0.43	111	18	75
Mitochondria	3.89	948	82	17	2.95	11	1	<1	1.83	60	13	277
S. cerevisiae	0.57	781	312	100	0.29	213	12	46	0.25	669	13	277
S. paradoxus	0.88	740	295	98	0.44	260	10	44	0.34	885	23	554
Bacteria	5.13	120323	8905	1850	2.90	3072	96	9	2.26	8661	377	10413
H. sapiens	2.54	34525	14538	1888	0.50	9874	172	966	1.44	28442	1666	10737

Each algorithm shows the compressed size (in bits per base), time taken to compress and decompress (in seconds), and the approximate maximum memory usage (in MB) during the experiments.

already supports these functions with a small amount of additional space.

With RLZ, it was nontrivial to determine what a reference sequence should be for the data sets containing arbitrary sets of sequences like the Hemoglobin, Influenza, Mitochondria, and Bacteria data sets. Therefore, a random sequence was chosen from the data set to be the reference. For the data sets *S. cerevisiae*, *S. paradoxus*, and *H. sapiens*, a reference sequence was available in each data set. COMRAD outperformed RLZ on all the data sets except for the human genomes. RLZ performed particularly badly on the data sets where an appropriate reference sequence was not available. In terms of compression and decompression times, and memory usage, RLZ was much more efficient than either RLCSA or COMRAD. As shown by the results, for data sets where the sequences have a large amount of similarity and a reference sequence is readily available, RLZ is more suitable than COMRAD. However, COMRAD is far more suited for more general compression.

We also compare COMRAD with some publicly available existing DNA compression algorithms, dna-x [31], GenCompress [8], DNACompress [9], CDNA [10], and XM [7] (Version 2.1). The algorithms and the results are further discussed in Appendix C, which can be found on the Computer Society Digital Library. In summary, dna-x and XM performed better than COMRAD for all data sets except in terms of runtime where both dna-x and XM were many orders of magnitude slower than COMRAD for data sets in the megabase range. In our experiments, DNACompress did not perform well.

Since XM consistently has the best compression results, we also compressed the reference human genome (NCBI Build 36) using XM as a comparison to COMRAD. For a single sequence, COMRAD does not compress as well as XM (with 500 experts, XM produces a result of 1.64 bpb on the human genome, compared to 1.82 bpb with our method). One reason is that COMRAD does not consider approximate repeats. However, COMRAD was able to compress the genome more than 10 times faster than XM (8 hours versus 94 hours). We also attempted to use XM to compress the four human genomes by compressing each chromosome against the respective reference chromosome. However, a single chromosome 1 sequence took around 6 hours to compress so we did not continue this experiment. In this analysis, we note that the aims of XM are somewhat different: to discover redundancy within a sequence, or by comparison to another sequence, and then adaptively compress. For this task, XM appears to be highly effective (as was shown by the human genome compression), but, due to its high memory consumption, it is less suited to the task of corpus compression.

For comparison with a general-purpose compression algorithm, we chose gzip since it is widely available and is often used to compress downloadable biological data. However, gzip also cannot be compared directly to COMRAD since gzip only uses a limited amount of main memory for compression, and provides adaptive compression. We did not restrict gzip to individual sequences, and thus for the smallest data sets, multiple sequences might be compressed with reference to each other. While gzip provides an obvious point of reference, we note that the purpose of the comparison is to set our work in a relevant context rather than assert that one method or the other is superior: the methods are designed for different tasks.

Results with gzip are shown in Table 4. For all but the smallest data sets, gzip has, unsurprisingly, been entirely ineffective. Models of a local region are unable to capture much useful redundancy for typical stored sequences, and little compression is achieved. This is a dramatic demonstration of the need for long-range or global models for achievement of maximum compression [15], [16]. In terms of compression time, gzip is faster than COMRAD. With another variant of an LZ algorithm that is able to use more main memory, 7-Zip, we observed better compression results, in similar time. For small data sets, 7-Zip has better compression results than COMRAD. However, as the data sets become larger, 7-Zip is no longer able to compress the data to the same level as COMRAD since even 7-zip has a maximum dictionary size limit of 32 MB. This is clearly evident for the *H. sapiens* data set where 7-zip was invoked for each chromosome file where the respective chromosomes from each genome were concatenated into a single file and given the proximity of similar sequences, 7-zip was still unable to compress as well as COMRAD.

While many of the existing algorithms such as dna-2, XM, and 7-zip perform well, none of these algorithms are able to provide random access into the compressed data. COMRAD on the other hand has random access support and also has the potential to support other database queries such as *count*, and *locate*. The only other work that we know of that provides this feature is the set of self-indexing algorithms by Mäkinen et al. [22], which includes *display* as well as *count* and *locate* operations. COMRAD does not implement *count* and *locate* yet; hence, we compare the RLCSA algorithm to COMRAD with the *locate* feature turned off.

We now compare the *display* results of RLCSA, RLZ, and COMRAD in terms of the speed and memory usage for the

TABLE 4
Comparison of COMRAD to Existing General-Purpose Compression Algorithms

Dataset	gzip			7-Zip			COMRAD		
	size (bpb)	comp (hrs)	decomp (secs)	size (bpb)	comp (hrs)	decomp (secs)	size (bpb)	comp (hrs)	decomp (secs)
Influenza set	0.56	<0.01	1	0.12	0.02	1	0.43	0.03	18
Hemoglobin	0.92	<0.01	1	0.69	<0.01	1	1.16	<0.01	13
Mitochondria set	2.00	<0.01	1	1.05	0.01	1	1.83	0.02	13
S. cerevisiae set	2.10	0.16	12	0.10	0.25	5	0.25	0.19	13
S. paradoxus set	2.16	0.15	3	0.16	0.27	3	0.34	0.25	23
Bacteria set	2.19	1.17	110	1.83	2.58	88	2.26	2.41	377
H. sapiens	2.03	5.57	327	1.70	8.49	488	1.44	7.90	1666
63xH. sapiens chr22	1.49	0.77	124	1.24	1.55	77	0.07	1.44	46
127xH. sapiens chr1	1.95	10.57	1067	1.63	24.25	1483	0.07	48.96	367
1023xH. sapiens chr20	2.05	28.75	949	1.75	45.55	3444	0.04	46.15	1193

The columns show the data set name, the compressed size (in bits per base), and time to compress for `gzip` using the -9 option, `7-Zip` using the "ultra" options, and COMRAD, respectively.

S. paradoxus data set. The results are in Table 5. For all data sets, COMRAD was many magnitudes faster than RLCSA. We assume this is due to the lack of memory locality of the BWT representation of RLCSA. COMRAD also lacks locality when accessing dictionary elements but only a few nonterminals need to be accessed per query. However, RLZ was also many magnitudes faster than COMRAD since the number of memory accesses required is limited to the number of factors that cover the region being extracted. In terms of memory usage, RLCSA uses just over half as much memory compared to COMRAD, since the COMRAD dictionary is an overhead of the algorithm. However, it is still cheaper than holding the uncompressed data in memory and, for very large data sets, is potentially faster than retrieving data from disk. RLZ uses just over a third of the memory used by COMRAD. While RLZ has the best random access performance overall, note that RLZ is only applicable for data sets with highly similar sequences and where there is an obvious choice of reference sequence, whereas COMRAD has no such restrictions.

System specification. All algorithms (except DNACompress) were run on a machine with four 2.6 GHz Dual-Core AMD Opteron processors and 32 GB of RAM running Ubuntu 8.04.3. DNACompress experiments were run on a machine with two 2.66 GHz Intel(R) Core(TM)2 Duo processors and 4 GB of RAM, running Windows XP. Only a single processor was used except for XM. All algorithms, except gzip and 7-zip, were executed with default parameter settings.

6 Conclusion

We have shown that a new method of dictionary construction for compression has been very effective in long-range repetition detection making it ideal for compressing very large DNA data sets. The results have shown that COMRAD is able to compress much larger data sets than existing DNA compression algorithms and able to detect repetition over a much longer range than existing general-purpose dictionary-based compression algorithms. While the algorithm is very memory intense and requires more theoretical grounding, we have shown experimentally that the method is effective and many improvements can be made to the general idea to create a very powerful and extendible compression algorithm for compressing and searching in very large data sets.

TABLE 5
display() Times for COMRAD, RLZ, and RLCSA for Varying Query Lengths on the *S. paradoxus* Data Set

Query Length	10	100	1000	10000	100000
RLCSA (μsec/char)	18.00	2.400	0.850	0.7100	0.7100
RLZ (μsec/char)	**0.23**	**0.025**	**0.0046**	**0.0025**	**0.0022**
COMRAD (μsec/char)	4.11	0.450	0.077	0.0380	0.0350

Data sets contain 1,000 randomly generated queries, each with the same length. The times for each algorithm are in microseconds per character extracted and are an average of five consecutive runs per data set. COMRAD used 76.35 MB of memory, RLCSA used 47.35 MB, and RLZ used 28.71 MB.

Acknowledgments

This work was supported by the Australian Research Council, and by the NICTA Victorian Research Laboratory. NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Center of Excellence program. The authors thank Simon Puglisi for his assistance with this work.

References

[1] D. Wheeler et al., "The Complete Genome of an Individual by Massively Parallel DNA Sequencing," *Nature*, vol. 452, no. 7189, pp. 872-876, 2008.

[2] D. Bentley et al., "Accurate Whole Human Genome Sequencing Using Reversible Terminator Chemistry," *Nature*, vol. 456, no. 7218, pp. 53-59, 2008.

[3] J. Wang et al., "The Diploid Genome Sequence of an Asian Individual," *Nature*, vol. 456, no. 7218, pp. 60-65, 2008.

[4] S. Schuster et al., "Complete Khoisan and Bantu Genomes from Southern Africa," *Nature*, vol. 463, no. 7283, pp. 943-947, 2010.

[5] A. Cannane and H. Williams, "General-Purpose Compression for Efficient Retrieval," *J. Am. Soc. for Information Science and Technology*, vol. 52, no. 5, pp. 430-437, 2001.

[6] B. Behzadi and F.L. Fessant, "DNA Compression Challenge Revisited: A Dynamic Programming Approach," *CPM '05: Proc. 16th Ann. Symp. Combinatorial Pattern Matching*, pp. 190-200, 2005.

[7] M.D. Cao, T. Dix, L. Allison, and C. Mears, "A Simple Statistical Algorithm for Biological Sequence Compression," *DCC '07: Proc. Data Compression Conf.*, pp. 43-52, 2007.

[8] X. Chen, S. Kwong, and M. Li, "A Compression Algorithm for DNA Sequences and Its Applications in Genome Comparison," *RECOMB '00: Proc. Fourth Ann. Int'l Conf. Research in Computational Molecular Biology*, pp. 107-117, 2000.

[9] X. Chen, M. Li, B. Ma, and J. Tromp, "DNACompress: Fast and Effective DNA Sequence Compression," *Bioinformatics,* vol. 18, no. 12, pp. 1696-1698, 2002.

[10] D. Loewenstern and P. Yianilos, "Significantly Lower Entropy Estimates for Natural DNA Sequences," *DCC '97: Proc. Data Compression Conf.,* p. 151, 1997.

[11] T. Matsumoto, K. Sadakane, and H. Imai, "Biological Sequence Compression Algorithms," *Genome Informatics,* vol. 11, pp. 43-52, 2000.

[12] J. Ziv and A. Lempel, "A Universal Algorithm for Sequential Data Compression," *IEEE Trans. Information Theory,* vol. IT-23, no. 3, pp. 337-343, May 1977.

[13] J. Cleary and I. Witten, "Data Compression Using Adaptive Coding and Partial String Matching," *IEEE Trans. Comm.,* vol. COM-32, no. 4, pp. 396-402, Apr. 1984.

[14] P. Deutsch, "Gzip File Format Specification Version 4.3," 1996.

[15] S. Grumbach and F. Tahi, "Compression of DNA Sequences," *DCC '93: Proc. Data Compression Conf.,* pp. 340-350, 1993.

[16] E. Rivals, J. Delahaye, M. Dauchet, and O. Delgrange, "A Guaranteed Compression Scheme for Repetitive DNA Sequences," *DCC '96: Proc. Data Compression Conf.,* p. 453, 1996.

[17] A. Apostolico and S. Lonardi, "Compression of Biological Sequences by Greedy Off-Line Textual Substitution," *DCC '00: Proc. Data Compression Conf.,* pp. 143-152, 2000.

[18] G. Korodi and I. Tabus, "An Efficient Normalized Maximum Likelihood Algorithm for DNA Sequence Compression," *ACM Trans. Information Systems,* vol. 23, no. 1, pp. 3-34, 2005.

[19] S. Christley, Y. Lu, C. Li, and X. Xie, "Human Genomes as Email Attachments," *Bioinformatics,* vol. 25, no. 2, pp. 274-275, 2009.

[20] M. Brandon, D. Wallace, and P. Baldi, "Data Structures and Compression Algorithms for Genomic Sequence Data," *Bioinformatics,* vol. 25, no. 14, pp. 1731-1738, 2009.

[21] J. Sirén, N. Välimäki, V. Mäkinen, and G. Navarro, "Run-Length Compressed Indexes Are Superior for Highly Repetitive Sequence Collections," *SPIRE '08: Proc. 15th Int'l Symp. String Processing and Information Retrieval,* pp. 164-175, 2009.

[22] V. Mäkinen, G. Navarro, J. Sirén, and N. Välimäki, "Storage and Retrieval of Individual Genomes," *RECOMB '09: Proc. 13th Ann. Int'l Conf. Research in Computational Molecular Biology,* pp. 121-137, 2009.

[23] V. Mäkinen, G. Navarro, J. Sirén, and N. Välimäki, "Storage and Retrieval of Highly Repetitive Sequence Collections," *J. Computational Biology,* vol. 17, no. 3, pp. 281-308, 2010.

[24] F. Claude, A. Fariña, M. Martínez-Prieto, and G. Navarro, "Compressed *q*-Gram Indexing for Highly Repetitive Biological Sequences," *Proc. 10th IEEE Conf. Bioinformatics and Bioeng.,* pp. 86-91, 2010.

[25] N.J. Larsson and A. Moffat, "Offline Dictionary-Based Compression," *DCC '99: Proc. Data Compression Conf.,* pp. 296-305, 1999.

[26] F. Claude and G. Navarro, "Self-Indexed Text Compression Using Straight-Line Programs," *MFCS '09: Proc. 34th Int'l Symp. Math. Foundations of Computer Science,* pp. 235-246, 2009.

[27] S. Kreft and G. Navarro, "LZ77-Like Compression with Fast Random Access," *DCC '10: Proc. 20th Data Compression Conf.,* pp. 239-248, 2010.

[28] S. Kuruppu, S.J. Puglisi, and J. Zobel, "Relative Lempel-Ziv Compression of Genomes for Large-Scale Storage and Retrieval," *SPIRE '10: Proc. 16th Int'l Symp. String Processing and Information Retrieval,* E. Chavez and S. Lonardi, eds., pp. 201-206, 2010.

[29] S. Kuruppu, S.J. Puglisi, and J. Zobel, "Optimized Relative Lempel-Ziv Compression of Genomes," *ACSC '11: Proc. 34th Australasian Computer Science Conf.,* M. Reynolds, ed., pp. 91-98, 2011.

[30] C. Neville-Manning and I. Witten, "Compression and Explanation Using Hierarchical Grammars," *The Computer J.,* vol. 40, nos. 2/3, pp. 103-116, 1997.

[31] G. Manzini and M. Rastero, "A Simple and Fast DNA Compressor," *Software—Practice and Experience,* vol. 34, pp. 1397-1411, 2004.

[32] S. Hirschberg and D. Lelewer, "Efficient Decoding of Prefix Coding," *Comm. ACM,* vol. 33, no. 4, pp. 449-459, 1990.

[33] M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat, "The Smallest Grammar Problem," *IEEE Trans. Information Theory,* vol. 51, no. 7, pp. 2554-2576, July 2005.

[34] D. Okanohara and K. Sadakane, "Practical Entropy-Compressed Rank/Select Dictionary," *ALENEX '07: Proc. Workshop Algorithm Eng. and Experiments,* 2007.

[35] S. Levy et al., "The Diploid Genome Sequence of an Individual Human," *PLoS Biology,* vol. 5, no. 10, p. e254, 2007.

[36] S.-M. Ahn et al., "The First Korean Genome Sequence and Analysis: Full Genome Sequencing for a Socio-Ethnic Group," *Genome Research,* vol. 19, no. 9, pp. 1622-1629, 2009.

Shanika Kuruppu is working toward the PhD degree at the University of Melbourne undertaking a research project titled "Efficient Algorithms for Specialised Search in Biomedical Data." Her research interests include implementing compression algorithms to store large DNA data sets efficiently and using succinct data structures to implement fast access and search for compressed data sets.

Bryan Beresford-Smith received the PhD degree in applied mathematics and worked as a senior lecturer in computer science before moving to private industry and then NICTA. He is a senior researcher at NICTA. His research interests include colloid science, parallel algorithms and architectures, wireless networks, and more recently bioinformatics and genomics.

Thomas Conway received the PhD degree in 2002 from the Department of Computer Science and Software Engineering, University of Melbourne, in the area of logic programming. He then worked for several years in the commercial sector on text search and information retrieval problems. In 2007, he joined NICTA and got involved in the bioinformatics activities there. His research at NICTA has centered on ways to apply insights from theoretical computer science to help address practical problems in the processing and interpretation of second generation sequencing data.

Justin Zobel is a professor at the University of Melbourne. Since completion of the PhD degree in 1991, he has been working at RMIT and NICTA. He is best known for his contributions to indexing and query evaluation mechanisms for text search, and has also made significant contributions in fundamental algorithms, compression, bioinformatics, and research methods, and for his text on writing skills for computer scientists.