UW CSE Technical Report 2007-05-02

Grammar-based Compression of DNA Sequences

Neva Cherniavsky Richard Ladner

May 28, 2004

Abstract

Grammar-based compression algorithms infer context-free grammars to represent the input data. The grammar is then transformed into a symbol stream and finally encoded in binary. We explore the utility of grammar-based compression of DNA sequences. We strive to optimize the three stages of grammar-based compression to work optimally for DNA. DNA is notoriously hard to compress, and ultimately, our algorithm fails to achieve better compression than the best competitor.

1 Introduction

Grammar-based compression methods have shown success for many different types of data. The central idea behind these methods is to use a context-free grammar to represent the input text. Grammars can capture repetitions occurring far apart in the data. This is a limitation on sliding window or block sorting algorithms, such as LZ77 or bzip2.

Compression of DNA sequences is a notoriously hard problem. DNA contains only four symbols, and so can be represented by two bits per symbol. It is very hard to beat the bound of two bits per symbol. However, DNA is also known to have long repetitions that a compressor could hope to capture. Furthermore, DNA has a unique kind of repetition, because it contains both exact matches and *reverse complement* matches.

We explore the effectiveness of grammar-based compression on DNA sequences. We exploit the different types of repetition in DNA by modifying a successful grammar inference algorithm. We then endeavor to maximize our gain in the next two stages of grammar compression: symbol stream encoding and binary encoding. Finally, we return to the grammar inference portion of the algorithm and improve the grammar by quantifying the efficiency of the rules. After striving to optimize each stage of our compression algorithm, we conclude that grammar-based compression does not work well on DNA sequences.

1.1 Grammar compression

Grammar-based compression starts by inferring the context-free grammar to represent the string. The resulting grammar is encoded as a symbol stream, which is then converted to binary (see Figure 1). Each step affects the final size of the compressed file.

Grammar inference takes as input the string to be compressed and infers a grammar to represent that string. The language of the grammar is the one unique input string. Grammar encoding takes as input a grammar and produces a symbol stream. The entropy coder transforms the symbol stream into a bit stream. The decoder then proceeds backwards, first transforming the bit stream into a symbol stream, then converting the symbol stream into the grammar, and finally

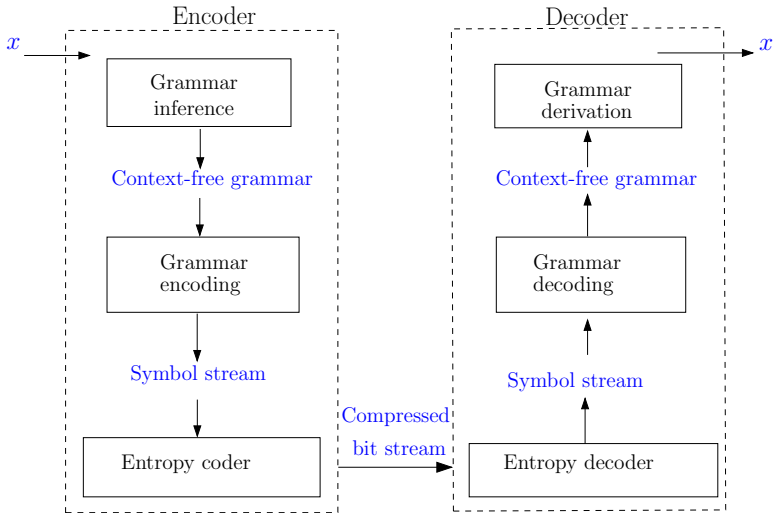

Figure 1: Overview of grammar compression

decoding the grammar by interpreting the right hand side of the start rule to produce the original data.

1.2 Our contributions

We modify a successful grammar inference algorithm, Sequitur [21, 22], to work better on DNA sequences. We then present several different techniques for grammar encoding, ranging from very simple methods to more complicated pointer-based methods. All skew the data in some way to make arithmetic coding more attractive. Lastly, we implement a custom arithmetic coder for the entropy coder, specifically designed to work well with our symbol stream. After evaluating our algorithm, we explore improvements to the grammar to make it more efficient.

2 Background

In this section we describe Sequitur, a compression algorithm that infers a context-free grammar. Sequitur is the starting point for our work, so we devote some time to fully explaining the algorithm. We also discuss other grammar-based algorithms and describe related work in compression of DNA sequences.

2.1 Sequitur

Sequitur, created by Nevill-Manning and Witten in 1997 [21, 22], is an on-line linear-time algorithm that infers a context-free grammar to losslessly represent the input. The language of the grammar consists of one string, the uncompressed text. The grammar represents both local and nonlocal repetition without restrictions on space or time.

Sequitur discovers the grammar incrementally by maintaining two invariants, *digram uniqueness* and *rule utility*. Digram uniqueness requires that no pair of adjacent symbols appears more than once in the grammar. As the algorithm reads in the data, it checks each new digram against a digram table of all current digrams. The first time a digram is repeated, Sequitur forms a new rule with the digram as the right-hand side. Thereafter, when the same digram appears, it is

replaced by a symbol representing the rule. Rule utility requires that each rule is used at least twice (except for the start rule). The algorithm maintains a count of rule usage. If a rule's count falls to 1, which may happen while satisfying digram uniqueness, that rule is eliminated. Its one location is replaced by its right-hand side.

For example, consider the string *acgtcgacgt*, which is the DNA analogue of the string used in the original Sequitur papers [21, 22]. (Usually, DNA is represented with capital letters, but because we need the upper-case symbols to describe the grammar, we will use the lower-case $\{a, c, t, g\}$ throughout this paper.) The algorithm reads in the data, placing the input characters on the right hand side of the start rule, and storing the digrams in the digram table. After the first five symbols, the grammar looks like

$$S \rightarrow acgtc$$

while the table contains the digrams $\{ac, cg, gt, tc\}$. The next symbol, g, forms the digram cg, which is already in the table. Thus Sequitur creates a new rule:

$$\begin{array}{lcl} S \rightarrow a\underline{cg}t\underline{cg} & \Rightarrow & \begin{array}{lcl} S \rightarrow aAtA \\ A \rightarrow cg \end{array} \end{array}$$

Now the digram table contains $\{aA, At, tA, cg\}$. Reading the next two symbols ac causes no invariant violations, so ac is appended to the start rule and the digram table becomes $\{aA$, At, tA, cg, Aa, $ac\}$. The following symbol, g, again forms the digram cg, which is already the right hand side of a rule.

$$\begin{array}{lcl} S \rightarrow aAtAa\underline{cg} \\ A \rightarrow \underline{cg} \end{array} \quad \Rightarrow \quad \begin{array}{lcl} S \rightarrow aAtAaA \\ A \rightarrow cg \end{array}$$

Observe that replacing cg with the rule A causes a violation of digram uniqueness; aA appears twice. Sequitur creates a new rule:

$$\begin{array}{lcl} S \rightarrow \underline{aA}tA\underline{aA} \\ A \rightarrow cg \end{array} \quad \Rightarrow \quad \begin{array}{lcl} S \rightarrow BtAB \\ A \rightarrow cg \\ B \rightarrow aA \end{array}$$

The final symbol t causes another new rule formation:

$$\begin{array}{lcl} S \rightarrow \underline{Bt}A\underline{Bt} \\ A \rightarrow cg \\ B \rightarrow aA \end{array} \quad \Rightarrow \quad \begin{array}{lcl} S \rightarrow CAC \\ A \rightarrow cg \\ B \rightarrow aA \\ C \rightarrow Bt \end{array}$$

Notice that B now appears only once on the right hand side of rules. This is a violation of rule utility, so the algorithm eliminates B.

$$\begin{array}{lcl} S \rightarrow CAC \\ A \rightarrow cg \\ B \rightarrow aA \\ C \rightarrow \underline{B}t \end{array} \quad \Rightarrow \quad \begin{array}{lcl} S \rightarrow CAC \\ A \rightarrow cg \\ C \rightarrow aAt \end{array}$$

There are no more symbols or invariant violations, so this is the final grammar returned by Sequitur.

Sequitur does not find the smallest context-free grammar to represent the string. That task is NP-hard, as shown by Lehman [17, 18], who closely followed work done by Storer [27]. Nor is the grammar close in size to those returned by the best approximation algorithms. Lehman showed that there is a pathological input on which Sequitur will be an $\Omega(n^{\frac{1}{3}})$ approximation to the smallest grammar. He also showed that on all grammars, Sequitur will do no worse than an $O((n/\log n)^{\frac{3}{4}})$ approximation. This compares unfavorably with Lehman's method, which has worst-case approximation ratio $O(\log^3 n)$, and also is worse than some other more practical implementations. Further work by Charikar et al. [6] found an $O(\log(n/g^*))$ approximation algorithm, where g^* is the size of the smallest grammar. However, as Lehman notes, the focus of data compression algorithms is different than the focus of approximation algorithms; the latter looks for the smallest possible grammar size, regardless of computational cost, while the former tries to find the smallest possible compressed data while using a reasonable amount of computational resources. Approximation algorithms focus on worst-case analysis, often on pathological input, while data compression algorithms are concerned with compressing real data. Furthermore, the smallest grammar size does not necessarily lead to the smallest compressed file.

Kieffer and Yang [13] also showed that Sequitur is not universal, that is, it is not equal to entropy in the limit under some model. This is because it is possible for Sequitur to have two different rules that generate the same string. Consider for example

$$\begin{aligned} A &\rightarrow ac \\ B &\rightarrow gt \\ C &\rightarrow AB \\ D &\rightarrow aEt \\ E &\rightarrow cg \end{aligned}$$

(Assume this is a small piece of the grammar, and the invariants are satisfied elsewhere.) The string generated by the rule C, denoted $\langle C \rangle$, and the string generated by D are the same: $\langle C \rangle = \langle D \rangle = acgt$. In order for Sequitur to be universal, its grammars must be irreducible, and in order for the grammars to be irreducible, this type of repetition must be eliminated. We will refer to this change as the Kieffer-Yang improvement.

2.2 Other grammar-based methods

There are several other grammar-based compression methods. Larsson and Moffat [16] propose an off-line algorithm called Re-Pair that infers a dictionary off-line by recursively creating phrases for the most frequently occurring digrams. It then transmits the dictionary along with the compressed stream for on-line decoding. Another off-line algorithm by Bentley and McIlroy [4] replaces long repetitions with an offset and length. It is not efficient for short string repeats. However, it does have lower main memory requirements than either Sequitur or Re-Pair, because it uses block sizes to control the size of the longest string found.

Another type of grammar-based method for compression uses derivation trees [5, 12]. A derivation tree specifies how a grammar produces a particular string. This method assumes that the grammar is known to both the encoder and decoder, and that the language of the grammar contains all possible input strings to the encoder. The algorithm then encodes the derivation tree with respect to the grammar for the given input string.

Much work has been done on grammar inference [9, 26, 15, 28]. Typically, these methods try to infer a grammar for a set of strings, a problem that has wide applicability in natural language

processing. The strings might be positive or negative examples for the grammar. The metric for a good grammar is one that recognizes strings in the language and rejects those not in the language. The methods are similar to those used by Sequitur, though the aim of the algorithms is clearly very different.

Finally, Kieffer and Yang [13] have developed a framework of provable properties for good compression of context-free grammars, as discussed above. Yang et al. [14] then developed an algorithm for estimating DNA sequence entropy that satisfied the properties and also took advantage of reverse complements. The algorithm runs in linear time through a complicated use of suffix trees. This yielded impressive entropy results, though the grammar was not actually sent through an entropy encoder, and thus the practical compression results remain unknown.

2.3 DNA compression

DNA is notoriously hard to compress [19, 14]. In 1994, Grumbach and Tahi [11] showed that many common compression algorithms do not work well on DNA sequences, and introduced their own method, *biocompress-2*. *Biocompress-2* searches for "palindromes," or reverse complements, within an LZ77-style window [29, 30]. The result is then arithmetically encoded. Grumbach and Tahi achieved only 10.24% compression on average. Their corpus became the standard by which DNA compression algorithms are measured, and we use this corpus in our results section.

Loewenstern and Yianilos [19] created an entropy estimator and compressor called CDNA that tries to capture inexact matches in DNA by creating a set of experts and then weighting these through the Expectation Maximization algorithm. The data are segmented and the predictor is trained on the data minus the segment to be tested. This training is repeated for each segment and the result is the cross-validation of all the iterations. The on-line version of this algorithm would train the experts on everything to the left of the current nucleotide to predict its value. However, this method was never implemented and its mathematical behavior was left to future work. Furthermore, this algorithm takes more than linear time, which is unacceptable for large DNA sequences.

Apostolico and Lonardi [3] developed an off-line method of textual substitution that finds good substrings to compress and represents them via Lempel-Ziv style pointers [29, 30]. They applied their method to DNA sequences [2]. However, in their results section, they compare the compressed result with the original size of the file in bytes, rather than its size when represented with two bits per symbol (which is the minimum necessary to represent DNA without any compression). When this oversight is taken into account, the compression is small, and sometimes the algorithm expands the file beyond two bits per symbol.

In 1999, Chen and Li created GenCompress [7], which uses approximate matching to find both reverse complements and inexact matches. They achieved much better results than *biocompress*−2. The authors further improved their algorithm by creating PatternHunter for homology search [20], which led to DNACompress [8]. PatternHunter finds exact and inexact matches in DNA, with both forward matching and reverse complement matching. DNACompress works by first running PatternHunter on the data, which creates a list of homologous regions sorted from highest to lowest score. The score is based on user-weighted parameters such as base matches and the size of gaps in the homologous region. DNACompress processes items on the list until the score of the top item is less than some pre-defined threshold. Processing consists of replacing the repeated regions with a triple defining the length and starting positions of the repeat, followed by a series of triples defining edit operations (for inexact matches). The algorithm is off-line; PatternHunter is proprietary software; and the asymptotic time complexity is not discussed in either paper.

However, it runs quickly in practice and DNACompress appears to be the best DNA compression algorithm at the present time.

2.4 Adaptive arithmetic coding

We use adaptive arithmetic coding extensively in our implementation of grammars for DNA compression. Arithmetic coding was created independently by Pasco [24] and Rissanen [25] in 1976. Essentially, each symbol is mapped to a unique subinterval in the unit interval $[0, 1)$ using the probability distribution of the data. The width of the interval equals the probability of the string. Thus, a symbol that is likely to occur will have a large interval and a short unique tag indicating that interval, whereas an unlikely symbol will have a small interval and it will take more bits to represent a tag within that interval. Arithmetic coding is most useful when there is a skewed probability distribution, because high probability symbols will map to bigger intervals and can thus be represented by shorter tags. Often arithmetic coding is employed as a last step in data compression, after processing has created the desired skew.

One inefficiency in this method is that both the encoder and the decoder need to know the probability distribution. Adaptive arithmetic coding solves this by updating the probability distribution as symbols are sent. When a new symbol occurs that does not exist in the current distribution, an escape symbol is sent, followed by some fixed code known to both the encoder and decoder. The new symbol is then added to the probability distribution. Thus, the escape symbol also has a place in the probability distribution; if there are many new symbols, the cost of encoding the escape symbol will be small.

3 Grammar inference

We created a modified version of Sequitur called *DNASequitur*. Our main contribution is to improve Sequitur to work better on DNA sequences. DNA sequences contain structure and repetitions, so Sequitur might be expected to work well on this data. However, the Nevill-Manning version does not fare well on the standard corpus. One reason might be that there is additional structure in DNA that Sequitur cannot take advantage of, namely the reverse complements.

DNA sequences are made up of four nucleotide bases, a, c, t, and g. These in turn form chemical bonds with each other, so that a bonds with t and c bonds with g. Since DNA consists of two strands linked in together in a double helix, the second strand maps to the first in *reverse complement* order. Here are some properties of the reverse complement:

- The complement of a is t and vice-versa: $a' = t$ and $t' = a$. Similarly, $c' = g$ and $g' = c$.
- The reverse complement of two strings x and y satisfies: $(xy)' = y'x'$.
- The reverse complement of the reverse complement of a string w is the original string: $(w')' = w$.

For example, $(cat)' = atg$.

We added an invariant to Sequitur to exploit this hidden structure of DNA. In addition to digram uniqueness and rule utility, our enhanced version also mandates reverse complement uniqueness and utility. When a digram is stored in the digram table, its reverse complement is also implicitly stored. When either an exact match or a reverse complement match is seen subsequently, a rule is formed; in the first case, the right-hand side is the original digram, and in the second case, the right-hand side is the reverse complement. The digram and its reverse

complement are replaced in the grammar with the nonterminal representing the rule. If the rule is A, the reverse complement is denoted A'. A' is considered to be an instance of rule A for rule utility purposes. A is eliminated when it appears only once on the right hand side of rules in the grammar, either as the original or the reverse complement. If both A and A' appear on the right hand side, then A is used twice and would not be eliminated.

For example, if $A \rightarrow x$, then $A' \rightarrow x'$, and the complement of the string derivation of rule A is equal to the string derivation of the complement: $\langle A \rangle' = \langle A' \rangle$. Concretely, if $A \rightarrow cDt$, then A' is implicitly defined as $A' \rightarrow aD'g$.

Let's look at our original example with this new invariant. We start with the string $acgtcgacgt$. The algorithm reads in the data, placing the input characters on the right hand side of the start rule, and storing both the digrams and their reverse complements in the digram table. Consequently, there is a match much earlier. The first three symbols, acg, result in a digram table of $\{ac, cg\}$ and, implicitly, $\{gt\}$, because $(ac)' = gt$ and cg is its own reverse complement. The next symbol, t, results in a violation of digram uniqueness and a new rule is formed:

$$\begin{array}{lcl} S & \rightarrow & \underline{acgt} \\ \end{array} \Rightarrow \begin{array}{lcl} S & \rightarrow & AA' \\ A & \rightarrow & ac \end{array}$$

Now the digram table (plus implicit complements) contains $\{AA', ac, gt\}$, because AA' is its own reverse complement. Reading the next three symbols cga causes no invariant violations, so cga is appended to the start rule and the digram table and implicit complements become $\{AA'$, ac, gt, $A'c$, gA, cg, ga, $tc\}$. The following symbol, c, again forms the digram ac, which is already the right hand side of a rule.

$$\begin{array}{lcl} S & \rightarrow & AA'cg\underline{ac} \\ A & \rightarrow & \underline{ac} \end{array} \Rightarrow \begin{array}{lcl} S & \rightarrow & AA'cgA \\ A & \rightarrow & ac \end{array}$$

The replacement causes a digram uniqueness violation; the digram gA is implicitly in the table because $(A'c)' = gA$.

$$\begin{array}{lcl} S & \rightarrow & A\underline{A'cgA} \\ A & \rightarrow & ac \end{array} \Rightarrow \begin{array}{lcl} S & \rightarrow & ABB' \\ A & \rightarrow & ac \\ B & \rightarrow & A'c \end{array}$$

The digram table and implicit reverse complements now contain $\{AB, B'A', BB', ac, gt, A'c, gA\}$. The next symbol, g, does not violate any invariants. The final symbol, t, forms the digram gt, which corresponds to A'.

$$\begin{array}{lcl} S & \rightarrow & ABB'\underline{gt} \\ A & \rightarrow & \underline{ac} \\ B & \rightarrow & A'c \end{array} \Rightarrow \begin{array}{lcl} S & \rightarrow & ABB'A' \\ A & \rightarrow & ac \\ B & \rightarrow & A'c \end{array}$$

Notice that $B'A'$ is already implicitly in the digram table as the reverse complement of AB. This is a violation of digram uniqueness, and the grammar becomes:

$$\begin{array}{lcl} S & \rightarrow & \underline{ABB'A'} \\ A & \rightarrow & ac \\ B & \rightarrow & A'c \end{array} \Rightarrow \begin{array}{lcl} S & \rightarrow & CC' \\ A & \rightarrow & ac \\ B & \rightarrow & A'c \\ C & \rightarrow & AB \end{array}$$

B now violates rule utility, since it appears only once on the right hand side. The final grammar is:

$$\begin{array}{lcll} S & \rightarrow & CC' & \Rightarrow \\ A & \rightarrow & ac & \\ B & \rightarrow & A'c & \\ C & \rightarrow & A\underline{B} & \end{array} \quad \begin{array}{lcl} S & \rightarrow & CC' \\ A & \rightarrow & ac \\ C & \rightarrow & AA'c \end{array}$$

A comparison of the grammars inferred by the original Sequitur on DNA data and those inferred by this version shows that this version results in smaller grammar sizes [10].

4 Grammar Encoding

There are several different ways to encode the grammar. We implemented three versions: one which is very simple, and two that take advantage of LZ77 ideas [29, 30] to make the symbol stream smaller and easier to transform into a bit stream.

4.1 Simple symbol stream

A very simple encoding technique is to send the right hand side of each rule, separated by a special symbol.

$$\begin{array}{lcll} S & \rightarrow & CC' & \\ A & \rightarrow & ac & \Rightarrow \quad CC'\#ac\#AA'c \\ C & \rightarrow & AA'c & \end{array}$$

The symbol stream is easy to encode and decode and takes linear time to produce. However, the stream does not result in good compression. The length of the symbol stream is simply too large.

4.2 LZ77-style symbol stream

A better method is to use a pointer to make the stream smaller. The basic idea is to send the right hand side of the start rule. When a rule is encountered for the first time, its right hand side is sent. The second time a rule is encountered, a tuple is sent that indicates where the rule starts, how long it is, and if this occurrence is a complement. Thereafter, when the rule is encountered, the rule number is sent, along with the " ' " symbol if it's a complement.

We implemented two different flavors of encoders. Our first version sends a quadruple the second time a rule is encountered. The quadruple (r, o, l, k) represents the rule number r that the new rule comes from, the offset o within the rule, the length l, and whether this occurrence is a complement k.

For example, consider our previous grammar:

$$\begin{array}{lcl} S & \rightarrow & CC' \\ A & \rightarrow & ac \\ C & \rightarrow & AA'c \end{array}$$

We start out by sending the right hand side of S: CC'. C is the first appearance of the rule so we send its right hand side:$AA'c$. A is the first appearance of the rule so we send its right hand side: ac. Thus, the first symbols sent are ac. Next, we process A', which is the second appearance of A. This is a quadruple. A came from rule C, but rule C doesn't exist yet. The lowest parent rule that already exists is the start rule, or rule 0. We send $(0, 0, 2, 1)$: the rule is 0, the offset within

the rule is 0, the length of the rule is 2, and this occurrence is a complement. The last symbol in rule C is c, so we send that to the stream.

The symbol stream so far is $ac(0,0,2,1)c$ and we've processed the first appearance of C. The next symbol is the second appearance of rule C. We send a quadruple $(0,0,3,1)$ representing that C starts in rule 0 at offset 0, has length 3, and this occurrence is a complement. The final symbol stream is $ac(0,0,2,1)c(0,0,3,1)$.

The decoder adds symbols to the right hand side of the start rule until it encounters a quadruple; then it creates the rule and continues. In this example, the decoder creates $S \rightarrow ac$. It encounters the quadruple and creates a new rule $A \rightarrow ac$, then replaces the repeat where the rule starts with A and the quadruple with A' (since the quadruple indicated it was a complement). Now the grammar looks like:

$$\begin{array}{lcl} S & \rightarrow & \underline{ac}\underline{(0,0,2,1)} \end{array} \quad \Rightarrow \quad \begin{array}{lcl} S & \rightarrow & AA' \\ A & \rightarrow & ac \end{array}$$

The next symbol is c and is added to the right hand side of the start rule. The final grammar is:

$$\begin{array}{lcl} S & \rightarrow & \underline{AA'c}\underline{(0,0,3,1)} \\ A & \rightarrow & ac \end{array} \quad \Rightarrow \quad \begin{array}{lcl} S & \rightarrow & CC' \\ A & \rightarrow & ac \\ C & \rightarrow & AA'c \end{array}$$

If there were subsequent appearances of A or C, they would be sent with their label (i.e., A, A', C, or C').

Because the parent rule is often the start rule, the size of the offset within the rule grows with the size of the right hand side of the start rule. The actual offset shows no discernible pattern; even when the right hand side of the start rule is big, the offset could be close to the beginning (very small), or it could be close to the end of the right hand side (very big). The size gets as big as tens of thousands of symbols, meaning it can take > 14 bits to encode the offset. This led us to try another method.

4.3 Marker method

In an early paper, Nevill-Manning, Witten, and Maulsby [23] describe in detail a marker method for encoding the symbol stream, which we modified slightly. When a new rule is encountered, a marker symbol is sent, and then the right hand side of the rule is sent. The second time the rule is encountered, a triple (o,l,k) is sent, corresponding to the offset o, the length l, and whether this occurrence is a complement k.

Our symbol stream will look very similar to before: $\#\#ac(1,2,1)c(0,3,1)$. However, the offsets mean something different; they refer to the marker symbols that have not been processed.

The decoder reads the input $\#\#$ and stores these as pointers in the grammar. After reading ac, the decoder's grammar looks like

$$S \rightarrow \#\#ac,$$

though the $\#\#$ symbols represent pointers. The first triple refers to offset 1; for the decoder, this is the index of the pointer in the grammar. There are two pointers currently in the grammar, so 1 refers to the second one.

$$\begin{array}{lcl} S & \rightarrow & \underline{\#\#ac}\underline{(1,2,1)} \end{array} \quad \Rightarrow \quad \begin{array}{lcl} S & \rightarrow & \#AA' \\ A & \rightarrow & ac \end{array}$$

Now there is only one pointer in the grammar. It would not be possible for a subsequent triple to have any offset other than 0. The next symbol transmitted is c, which is appended to the right hand side of the start rule. Finally, the last triple causes another rule formation:

$$\begin{array}{lcl} S & \rightarrow & \underline{\#AA'c}\underline{(0,3,1)} \\ A & \rightarrow & ac \end{array} \quad \Rightarrow \quad \begin{array}{lcl} S & \rightarrow & CC' \\ A & \rightarrow & ac \\ C & \rightarrow & AA'c \end{array}$$

If, after this beginning of the symbol stream, a marker symbol and a triple appeared, the offset of the marker would be 0, because there would be only one pointer represented in the grammar.

The offset is limited in size. Both the encoder and decoder know how many marker symbols have been sent and how many have been already spoken for by other rules. In practice, it can get as big as the number of rules in the grammar, or in the thousands (> 10 bits). When a new rule is encountered, its right hand side is sent and its absolute number is recorded within the rule. The second time the rule is encountered, the offset is calculated based on this absolute number and what marker symbols have already been taken care of.

Although we would expect the second method to have better overall compression, it does not. Perhaps the savings in the offset size is wiped out by the addition of a marker symbol per rule. Since there are thousands of rules, this would be an additional thousands of symbols. The custom arithmetic coders, discussed below, could be expected to deal well with this kind of skewed data. However, the first method has an advantage because of the prevalence of $r = 0$ and $l = 2$ in the quadruple.

5 Binary encoding of symbol streams

In order to maximize compression, we developed a custom adaptive arithmetic coder for the symbol stream. The compressed file has two bit streams embedded within it - the regular symbol stream, entropy encoded with the custom arithmetic coder, and a prefix- and fixed size-based bit stream for encoding the tuples.

5.1 Custom arithmetic coder

In the arithmetic coder, the symbols that start out with frequency are $a, c, t, g, \#$ for the alternate symbol stream, and a few other special symbols, discussed below. The coder is adaptive, in that the special symbols all start out with the same frequency and all rules start out with zero frequency. As symbols are encountered, their frequency is increased by 10. When triples or quadruples are encountered, an escape symbol is sent and they are written to the second bit stream. That rule and its complement are added to the symbol table for the arithmetic coder. Since complements occur with less frequency than regular matches, we add a frequency of 8 for the regular version of the rule and 2 for its complement.

We allow symbols in the arithmetic coder that have zero frequency. That is, many times a rule only appears twice in the grammar, so its label will never appear in the symbol stream. However, it is often still cost-efficient to encode that rule as its right-hand side plus a tuple encoding. This becomes clear below when we discuss the cost measure. There does seem to be an inefficiency in the arithmetic coder, since rules are added to the symbol table with nonzero frequency when their quadruple appears in the stream. We tried sending a special escape symbol so the decoder would know not to add that symbol to the table, but it was more costly than the original version.

5.2 Tuple coder

The triples and quadruples are encoded in a very similar manner. First of all, length 2 and rule 0 appear over 90% of the time. We exploited the power of the arithmetic coder by using different escape symbols for these special tuples. When the tuple is rule 0, length 2, and not a complement, we send "[" as the escape symbol; when it is rule 0, length 2, and a complement we send "$"; and when it is anything else we send "%". Then most of the time, the only thing left to encode is the offset. In both cases, we use a fixed-size code for the offset that is limited by how big the offset can be. For triples, this is ultimately limited by the number of rules in the grammar; for quadruples, it's limited by the size of the right hand side of the start rule.

In the infrequent case that the rule is not 0 or the length is not 2, we have to encode the other members of the tuple. For the quadruple, the rule number is usually 0 (even when the length is not 2), so we send 0 for rule 0, and 1 followed by $\lceil \log_2(P-1) \rceil$ bits for any other rule, where P is the number of rules seen so far. (We actually send $\lfloor \log_2(P-1) \rfloor$ if the rule number is in the lower half of the maximum range; this saves us a bit half of the time. We employ this same trick on the offsets as well).

For both the triple and quadruple, we send a γ-code for the length. A γ-code is a special type of prefix code that grows larger with the size of the number it's encoding. To encode n, we calculate $n+1$, then send m 0's, where $m+1$ is the number of bits required to write $n+1$ in binary. We then append the binary representation of $n+1$. The decoder reads m 0's and knows to read the next $m+1$ bits to get n.

n	γ-code
0	1
1	010
2	011
3	00100
4	00101
	⋮

In general, for small numbers the γ-code takes few bits. The vast majority of lengths are < 5; when the length isn't equal to 2, it's usually equal to 3. In the case of the triple, the smallest length is 3, since the special symbols cover the cases when the length is 2. We use the γ-code to describe $n = (\text{length} - 3)$. Thus, in a triple, if the length was 3, the γ-code would send 1; if it was 4, the γ-code would send 010. In a quadruple, it's not known if the length is 2, so we use the γ-code to describe $(\text{length} - 2)$.

6 Basic Results

In this section we present some basic results. For our test files, we use the DNA corpus first presented in [11] and available on the web at [1]. The corpus includes the complete genomes of two mitochondries (MIPACGA, MPOMTCG), two chloroplasts (CHNTXX, MPOCPCG), and the complete sequence of chromosome III of yeast (YSCCHRIII); five human genes (HUMGHCSA, HUMHBB, HUMHDABCD, HUMDYSTROP, HUMHPRTB); and the complete genomes of two viruses (VACCG, HEHCMVCG). In order to conserve space, the tables in this section will focus on MIPACGA, MPOCPCG, HUMHHCSA, HUMDYSTROP, and VACCG. Full results are available in the appendix.

Recall that our DNASequitur algorithm uses reverse complements when inferring the grammar. Table 1 compares the original Sequitur to DNASequitur on the five DNA sequences. The number

of productions is the number of rules formed; the length of the right hand side is the sum of all the right hand sides of the rules; the longest repeat is the longest rule formed; and the maximum number of repeats is the number of repeats for the rule that appeared the most in the grammar. On all files, the number of productions is smaller for DNASequitur, leading to a smaller right-hand side. DNASequitur also creates a larger longest repeat and a higher value for the maximum number of repeats. These two factors explain why DNASequitur produces smaller files than the original Sequitur.

Table 1: Sequitur vs. DNASequitur, best in bold

Sequence	Productions		Length RHS		Longest Repeat		Max Repeats	
	Seq	DNASeq	Seq	DNASeq	Seq	DNASeq	Seq	DNASeq
HUMDYSTROP	1,308	**1,163**	10,413	**10,099**	16	**19**	104	**138**
HUMGHCSA	2,288	**2,200**	13,270	**13,111**	225	**346**	113	**174**
MIPACGA	2,795	**2,616**	23,373	**22,910**	114	**115**	183	**195**
MPOCPCG	3,077	**2,985**	27,153	**26,843**	21	**28**	185	**250**
VACCG	4,480	**4,373**	41,303	**40,654**	560	560	238	**287**

To illustrate the differences between symbol streams, table 2 compares the lengths of the three types of streams before and after binary encoding. The baseline length in the second column is the number of nucleotides in the DNA sequence divided by four. This is the baseline because a DNA sequence can be represented as a fixed code with two bits per symbol. An algorithm achieves compression by producing a file smaller than this baseline. In order to compare fairly across differently designed symbol streams, we ran *bzip*2 on the three streams to produce the binary results. *Bzip*2 is a popular compressor that normally performs quite well on many different types of data.

The simple symbol stream is larger in binary size than either the LZ77 or marker streams. It is so simple that its binary size cannot be improved upon with tricks, and there is a large gap between its size and the baseline. This justifies our implementation of a custom arithmetic coder, and shows why we must be clever in encoding the symbol stream.

Table 2: Comparison of symbol streams, best in bold

Sequence	Baseline	Symbol stream (bytes)			Bzip2 on symbol stream (bytes)		
		Simple	LZ77-style	Marker	Simple	LZ77-style	Marker
HUMDYSTROP	9,693	40,652	41,168	**38,667**	14,321	**13,042**	13,475
HUMGHCSA	16,624	57,052	58,200	**53,704**	20,318	**17,780**	18,491
MIPACGA	25,079	98,834	99,743	**93,820**	34,500	**31,611**	32,769
MPOCPCG	30,256	117,079	117,049	**110,152**	40,687	**37,285**	38,594
VACCG	47,935	183,972	186,055	**175,587**	64,074	**58,798**	60,654

Figure 2 demonstrates why we chose to implement the marker symbol stream, which uses triples as opposed to quadruples. In the LZ77-style symbol stream, the maximum size of the offset grows with the size of the right-hand side of the start rule. However, there is no discernible pattern to the offsets; even close to the final rule number encoded, offsets occur throughout the range of possible values. The bits required to encode the offset thus grow logarithmically with the size of the right hand side of the start rule. In contrast, the size of the offsets in the marker

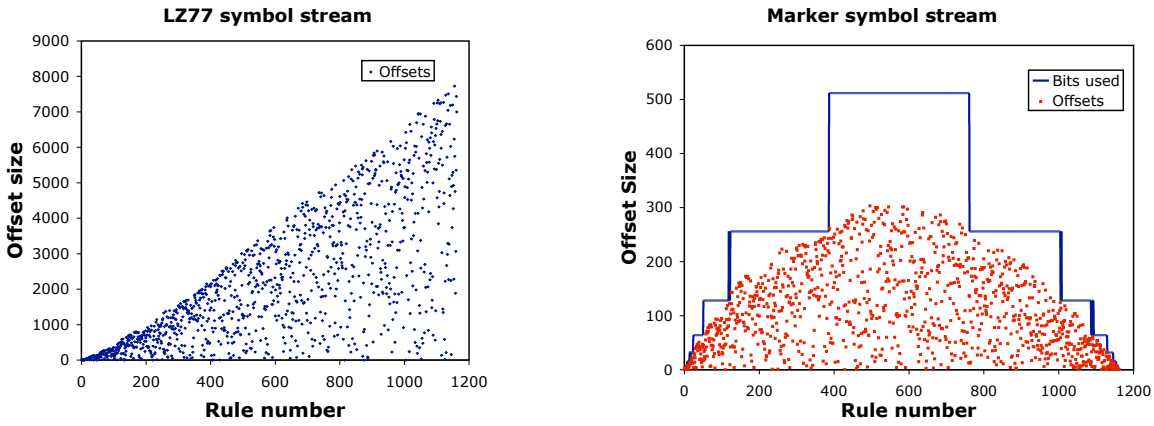

Figure 2: Offsets in the tuple symbol streams for HUMDYSTROP

symbol stream is limited throughout encoding. The figure shows the offsets together with 2^b, where b is the number of bits used to encode the offset. After a peak in the middle of encoding, the maximum size of the offset decreases. This is because the decoder knows how many marker symbols have been sent that do not correspond to a sent rule, and this is the maximum size of the offset; as more rules are sent, more marker symbols are eliminated from the list of open spots. The marker symbol stream appeared to be a good solution to the unchecked growth of offsets in the LZ77-style method.

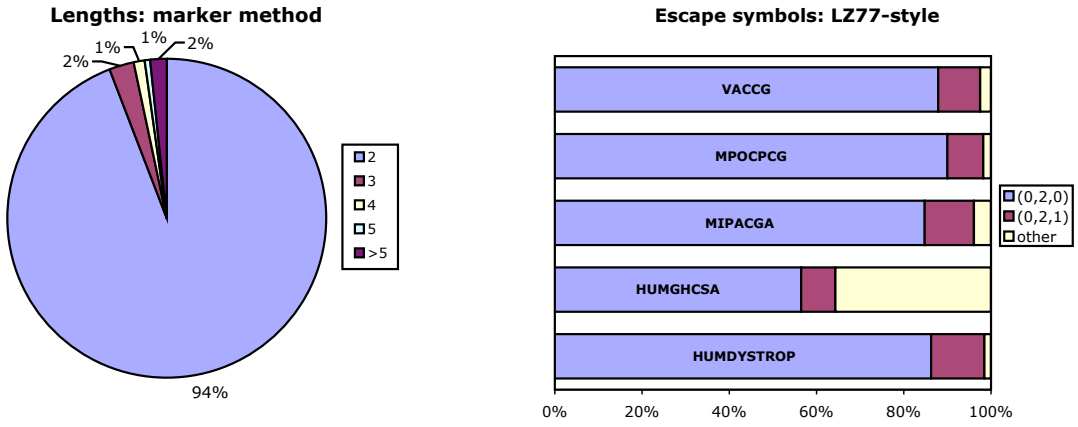

Figure 3: Frequency of lengths and frequency of the special escape symbols

To justify our use of escape symbols, figure 3 shows the frequency of length 2 in the marker method and the frequency of the special escape symbols for the LZ77-style method. The marker method graph is the aggregate lengths of the five test files. Special symbols are used when the length is 2, which happens over 94% of the time. The LZ77-style graph shows the percentage frequency of the three escape symbols for the test files. $(0, 2, 0)$ and $(0, 2, 1)$ refer to rule 0, length 2, and complement. Together, these two represent over 90% of the cases on average. Arithmetic coders perform well on skewed data in general, and the escape symbols help skew the frequency

of symbols in the stream further.

Finally, table 3 shows the sizes of the compressed files for different algorithms. For this measure we look at bits per symbol. The baseline for compression is two bits per symbol. Our methods (labeled Marker and LZ77-style) beat Sequitur on all the files; however, we don't compress on four out of the five files. It is worth noting that *bzip*2 also doesn't do well on the raw DNA; it too only compresses one out of the five files. The pure arithmetic coder achieves compression on four out of five files. DNACompress, the best known competitor, compresses all files, and does particularly well on HUMGHCSA. When that outlier is excluded, DNACompress performs close to, though better than, pure arithmetic coding on the DNA.

Table 3: Comparison of symbol streams (bits/symbol), best in bold

Sequence	Sequitur	Marker	LZ77-style	bzip2	arith	DNACompress
HUMDYSTROP	2.34	2.20	2.20	2.18	1.95	**1.91**
HUMGHCSA	1.86	1.74	1.77	1.73	2.00	**1.03**
MIPACGA	2.16	2.10	2.10	2.12	1.88	**1.86**
MPOCPCG	2.13	2.07	2.07	2.12	1.87	**1.67**
VACCG	2.11	2.06	2.06	2.09	1.92	**1.76**

7 Grammar improvement

Given the sub-optimal basic results, we explored ways in which to improve our algorithm. After exhausting methods to optimize the grammar encoding and the binary encoding of the symbol streams, we looked for ways in which to improve the efficiency of the grammar itself.

7.1 Kieffer-Yang improvement

Kieffer and Yang suggested an improvement to Sequitur to make it universal [13]. As described at the end of section 2.1, the improvement does not allow rules to have the same derivation. We implemented the improvement for both exact matches and reverse complement matches. The algorithm is the same as before, except when a rule is created, its right hand side derivation string is put into a table. When two digrams match and a new rule is about to be created, the would-be new rule's derivation and its reverse complement are checked for membership in the table. If there is a match, the new rule is not created, and the digrams are replaced with the appropriate old rule.

This leads to some surprising results. First of all, the customized encoders assume that a rule always appears before its complemented version. With the Kieffer-Yang improvement, this is no longer the case. To fix it, we implement a post-pass routine that looks for complements appearing before their original versions. Once found, the complement becomes the original; that is, the right hand side of the rule is changed to be its complement, and all appearances of the rule are switched to their complement. This allows the encoders to work, but results in more complements appearing in the grammar, which in turn increases the entropy of the data stream.

Furthermore, even when we use the simple encoding method, Kieffer-Yang still does not show much improvement over the traditional version. Table 4 shows the size of the simple symbol stream on DNASequitur with and without the Kieffer-Yang improvement. It also compares the sizes of the binary files after encoding with the LZ77-style symbol stream. Even without the

Table 4: Grammar Comparison, best in bold

Sequence	Simple symbol stream		Binary (w/LZ77-style stream)	
	Original	Kieffer-Yang	Original	Kieffer-Yang
HUMDYSTROP	40,652	**40,483**	10,658	**10,637**
HUMGHCSA	**57,052**	57,645	**14,674**	15,092
MIPACGA	98,834	**98,386**	26,311	**26,115**
MPOCPCG	117,079	**116,475**	**31,259**	31,431
VACCG	**183,972**	184,244	**49,307**	49,784

post-pass routine, the size of the Kieffer-Yang grammar is bigger on two out of five files. After encoding, Kieffer-Yang is larger on three out of five files. Though theoretically sound, in practice the improvement does not lead to greater compression on DNA sequences.

7.2 Cost measure improvement

We also explored an improvement of our own. We recognized that sometimes DNASequitur creates rules that are more expensive to encode than just sending the right hand side of the rule as is. We developed a cost measure to quantify this difference and remove rules that are inefficient. For example, one might expect a short rule that appears infrequently would be less efficient than simply sending the right hand side in place.

The cost measure compares the cost of using the rule with the cost of sending the right hand side instead. In the latter case, we're measuring the cost of replacing each instance of rule A with its right-hand side, and adding that to the cost of replacing each instance of A' with the complement of the right-hand side. Both costs are dependent on the bits per symbol of each symbol; we use an information measure to estimate this. The information also depends on whether the symbol is a complement.

Let $I(s)$ represent the information of symbol s, $N(s)$ be the number of times symbol s appears in the symbol stream, and T be the total number of symbols in the stream. Let $A \rightarrow x$, where x represents the right hand side of rule A, and $x = x_1 \cdots x_n$. $I(x)$ is an estimate of how many bits it will take to encode string x:

$$\begin{aligned} I(s) &= -\log_2(N(s)/T) \\ I(x) &= \sum_{i=1}^{n} I(x_i) \end{aligned}$$

Recalling the properties of the reverse complement from section 3, the cost of replacing a rule with its right hand side is

$$R(A) = N(A)I(x) + N(A')I(x')$$

The cost of using the rule is the cost of sending the right hand side once, plus the cost of encoding the tuple prefix code (including the rule marker symbol "#" and the escape symbol), plus the bits per symbol of using the rule thereafter. Let C be the cost of the tuple prefix code.

$$U(A) = I(x) + C + (N(A) - 2)I(A) + N(A')I(A')$$

or

$$U(A) = I(x) + C + (N(A) - 1)I(A) + (N(A') - 1)I(A'),$$

depending if the second appearance of A is a complement. The rule only appears in the symbol stream $(N(A) - 2 + N(A'))$ times. The first appearance in the grammar becomes the right hand side, and the second is taken up by the tuple.

The cost C of the tuple is the sum of cost of encoding the escape symbol and possibly the marker symbol and the cost of encoding the prefix code. The prefix code cost is fixed, depending only on rule number, length, complement, and most importantly, offset size. Thus this is calculated during an earlier pre-pass, before grammar improvement. The cost of encoding the escape symbol and marker symbol depends on the estimate of bits per symbol, which in turn depends on the information.

The grammar improvement phase of the algorithm calculates cost from bottom-up and eliminates rules if $U(A) > R(A)$. As rules are eliminated, the number of times symbols appear changes. We keep track of these changes in order to ensure accurate cost measurement at every point. Once the grammar has been completely checked, it is checked again until no more rules are eliminated. This repetition happens a constant number of times in practice.

7.3 Cost measure results, best in bold

Table 5: Original vs. Cost measure improved

Sequence	Productions		Length RHS		Longest Repeat		Max Repeats	
	Org	Cost	Org	Cost	Org	Cost	Org	Cost
HUMDYSTROP	1,163	**623**	**10,099**	10,314	19	19	138	**219**
HUMGHCSA	2,200	**1,713**	**13,111**	13,342	346	346	174	**209**
MIPACGA	2,616	**1,673**	**22,910**	23,358	115	115	195	**264**
MPOCPCG	2,985	**1,677**	**26,843**	27,554	28	28	250	**393**
VACCG	4,373	**1,452**	**40,654**	42,887	**560**	420	287	**585**

Table 5 shows the differences between the original (DNASequitur) grammars and the cost measure grammars. In some cases, close to half of the rules are removed. Only one sequences had the longest rule removed, and all sequences increased the maximum number of repeats for any given rule. The tradeoff shows in the increased size of the right hand side.

To evaluate our measure, we compare the percentage compression of the files on settings near our baseline of $U(A) > R(A)$. Specifically, we look at $U(A) + k > R(A) : -100 \leq k \leq 150$. Figure 4 shows the results. The relative flatness of the curves before $k = 0$ is due to the fact that negative k values amount to a penalty on removing rules. After a certain point, the cost measure will remove no rules. On the other end of the spectrum, there are diminishing returns after $k = 100$.

If our cost measure was precisely correct, we would see a maximum of percentage compression at 0. However, this only seems to be the case for the file with the most repetition, HUMGHCSA. The rest seem to reach maximum around $k = 35$. This includes VACCG, which actually shows compression under the cost measure, a big improvement over the 3% expansion it showed originally.

In the end, our cost measure does result in smaller files on all test sequences. It's not clear why the maximum compression is reached at $k = 35$, but it is promising that the maximum is at a consistent k across most sequences. Perhaps there is some inherent skew built in because we're using information, which is an optimistic estimate of bits per symbol.

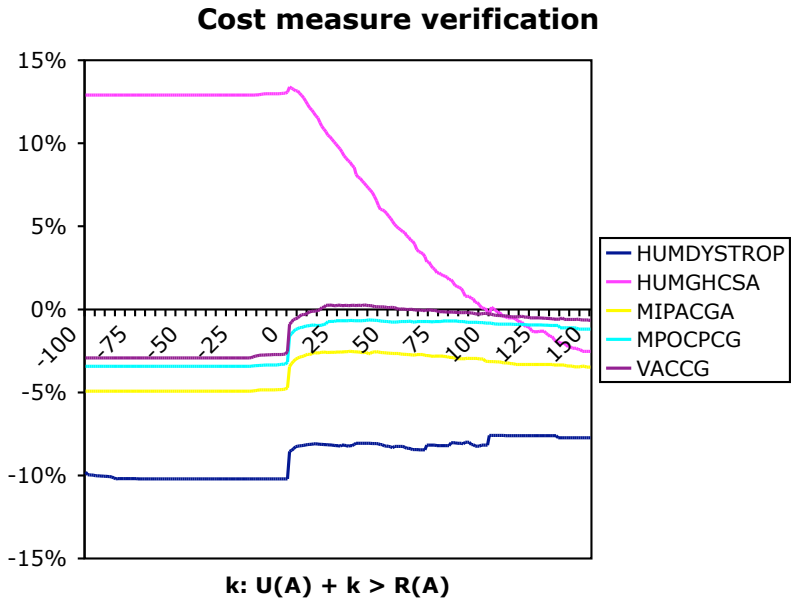

Figure 4: Evaluation of the cost measure

8 Conclusion and Future Work

We have thoroughly investigated using grammar compression for DNA sequences. Though grammars are good at capturing long repetitions that occur far apart in the data, DNA does not seem to have enough exact matches for the grammars to exploit. Even when both exact matches and reverse complement matches are considered, there is simply not enough repetition to make up for the expansion that using a grammar causes.

The best method for compressing DNA sequences appears to be DNACompress, which captures exact and inexact matches. Alhough DNACompress is not as elegant as Sequitur and does not seem to have reported asymptotic running time results, it executes quickly in practice and compresses DNA about 13 % on average.

One direction for future work is to explore the use of edit grammars for DNA sequences. Edit grammars allow rules like $A \to X[editop]$, where *editop* is an edit operation on X, such as insert a character, delete a character, or replace a character with another [6]. Sequitur could be modified to form an edit grammar while still using the same invariants of digram uniqueness and rule utility. If it were successful, this modification would combine an elegant and theoretically sound algorithm with good practical results, a great goal for any algorithms researcher.

References

[1] National center for biotechnology information: Entrez cross-database search. `http://www.ncbi.nlm.nih.gov/Entrez/`.

[2] Apostolico, A., and Lonardi, S. Compression of biological sequences by greedy off-line textual substitution. In *ProceedingsData Compression Conference* (Snowbird, UT, 2000),

J. A. Storer and M. Cohn, Eds., IEEE Computer Society Press, pp. 143–152.

[3] Apostolico, A., and Lonardi, S. Off-line compression by greedy textual substitution. *Proceedings of the IEEE 88*, 11 (2000), 1733–1744.

[4] Bentley, J., and McIlroy, D. Data compression using long common strings. In *DCC: Data Compression Conference* (1999), IEEE Computer Society TCC, pp. 287–295.

[5] Cameron, R. D. Source encoding using syntactic models. *IEEE Transactions on Information Theory 34*, 4 (1988), 843–850.

[6] Charikar, M., Lehman, E., Liu, D., Panigrahy, R., Prabhakaran, M., Rasala, A., Sahai, A., and Shelat, A. Approximating the smallest grammar: Kolmogorov complexity in natural models. In *Proceedings of the Thiry-Fourth Annual ACM Symposium on Theory of Computing, Montréal, Québec, Canada, May 19–21, 2002* (New York, NY, USA, 2002), ACM, Ed., ACM Press, pp. 792–801.

[7] Chen, X., Kwong, S., and Li, M. A compression algorithm for DNA sequences and its applications in genome comparison. In *Proceedings of the 10th Workshop on Genome Informatics (GIW-99)* (Dec. 1999), pp. 52–61.

[8] Chen, X., Li, M., Ma, B., and Tromp, J. DNACompress: Fast and effective DNA sequence compression. *Bioinformatics 18*, 12 (Dec. 2002), 1696–1698.

[9] Cook, C. M., Rosenfeld, A., and Aronson, A. Grammatical inference by hill climbing. *Informational Sciences 10* (1976), 59–80.

[10] Earl, E., and Ladner, R. Enhanced sequitur for finding structure in data. In *Proceedings of the 2003 Data Compression Conference (DCC 2003)* (Mar. 2003), p. 425.

[11] Grumbach, S., and Tahi, F. A new challenge for compression algorithms: genetic sequences. *Information Processing and Management 30*, 6 (1994), 875–886.

[12] Kawaguchi, E., and Endo, T. On a method of binary-picture representation and its application to data compression. *IEEE Transactions on Pattern Analysis and Machine Intelligence 2* (1980), 27–35.

[13] Kieffer, J. C., and hui Yang, E. Grammar-based codes: A new class of universal lossless source codes. *IEEETIT: IEEE Transactions on Information Theory 46* (2000), 737–754.

[14] Lanctot, J. K., Li, M., and hui Yang, E. Estimating dna sequence entropy. In *Proceedings of the eleventh annual ACM-SIAM symposium on Discrete algorithms* (2000), Society for Industrial and Applied Mathematics, pp. 409–418.

[15] Langley, P. Simplicity and representation change in grammar induction. Tech. rep., Stanford University, 1995.

[16] Larsson, N. J., and Moffat, A. Offline dictionary-based compression. In *DCC: Data Compression Conference* (1999), IEEE Computer Society TCC, pp. 296–305.

[17] Lehman, E. *Approximation Algorithms for Grammar-based Data Compression*. PhD thesis, Massachusetts Institute of Technology, 2002.

[18] Lehman, E., and Shelat, A. Approximation algorithms for grammar-based compression. In *Proceedings of the 13th Annual ACM-SIAM Symposium On Discrete Mathematics (SODA-02)* (New York, Jan. 6–8 2002), ACM Press, pp. 205–212.

[19] Loewenstern, D., and Yianilos, P. N. Significantly lower entropy estimates for natural DNA sequences. Tech. Rep. 96-51, DIMACS, Dec. 2 1996.

[20] Ma, B., Tromp, J., and Li, M. PatternHunter: faster and more sensitive homology search. *Bioinformatics 18*, 3 (Mar. 2002), 440–445.

[21] Nevill-Manning, C. G., and Witten, I. H. Compression and explanation using hierarchical grammars. *The Computer Journal 40*, 2/3 (1997), 103–116.

[22] Nevill-Manning, C. G., and Witten, I. H. Identifying hierarchical structure in sequences: A linear-time algorithm. *Journal of Artificial Intelligence Research 7* (Sept. 01 1997), 67–82.

[23] Nevill-Manning, C. G., Witten, I. H., and Maulsby, D. L. Compression by induction of hierarchical grammars. In *Proceedings of the 1994 Data Compression Conference (DCC 94)* (Snowbird, Utah, Mar. 1994), J. A. Storer and M. Cohn, Eds., IEEE Computer Society Press, Los Alamitos, California, pp. 244–253.

[24] Pasco, R. *Source Coding Algorithms for Fast Data Compression.* PhD thesis, Stanford University, 1976.

[25] Rissanen, J. J. Generalized kraft inequality and arithmetic coding. *IBM Journal of Research and Development 20* (May 1976), 198–202.

[26] Stolcke, A., and Omohundro, S. Inducing probabilistic grammars by bayesian model merging. In *International Conference on Grammatical Inference* (Alicante, Spain, 1994), Springer-Verlag, pp. 106–118.

[27] Storer, J. A., and Szymanski, T. G. Data compression via textual substitution. *Journal of the ACM 29*, 4 (Oct. 1982), 928–951.

[28] Wolff, J. G. An algorithm for the segmentation of an artificial language analogue. *British Journal of Psychology 66* (1975), 79–90.

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

[30] Ziv, J., and Lempel, A. Compression of individual sequences via variable rate coding. *IEEE Transactions on Information Theory IT-24*, 5 (Sept. 1978), 530–536.

Appendix

Sequence	Size	Sequitur	Marker	LZ77-style
CHNTXX	155,939	2.24	2.17	2.17
HEHCMVCG	229,354	2.22	2.18	2.17
HUMDYSTROP	38,770	2.34	2.20	2.20
HUMGHCSA	66,495	1.86	1.74	1.77
HUMHBB	73,308	2.20	2.11	2.11
HUMHDABCD	58,864	2.26	2.15	2.15
HUMHPRTB	56,737	2.22	2.16	2.16
MIPACGA	100,314	2.16	2.10	2.10
MPOCPCG	121,024	2.13	2.07	2.07
MPOMTCG	186,609	2.21	2.16	2.16
VACCG	191,737	2.11	2.06	2.06
YSCCHRIII	316,613	2.17	2.12	2.11

Table 6: Our results on the entire corpus: Original Sequitur, DNASequitur with the marker method symbol stream, and DNASequitur with the LZ77-style symbol stream. Size is the number of nucleotides in the original file; all other measurements are in bits per symbol (bps), where 2 bps is the baseline for compression. MPOCPCG is also referred to as CHMPXX; MIPACGA is also referred to as PANMTPACGA

Sequence	M + KY	L + KY	M + Cost	L + Cost	M + KY + Cost	L + KY + Cost
CHNTXX	2.19	2.19	2.15	2.12	2.19	2.17
HEHCMVCG	2.21	2.21	2.13	2.12	2.20	2.18
HUMDYSTROP	2.20	2.19	2.18	2.16	2.17	2.16
HUMGHCSA	1.79	1.82	1.76	1.75	1.81	1.81
HUMHBB	2.14	2.14	2.06	2.05	2.12	2.10
HUMHDABCD	2.17	2.18	2.14	2.12	2.18	2.16
HUMHPRTB	2.17	2.17	2.15	2.14	2.17	2.15
MIPACGA	2.09	2.08	2.08	2.06	2.07	2.06
MPOCPCG	2.08	2.08	2.05	2.02	2.07	2.05
MPOMTCG	2.22	2.22	2.14	2.12	2.22	2.20
VACCG	2.08	2.08	2.02	2.01	2.07	2.05
YSCCHRIII	2.14	2.14	2.09	2.08	2.12	2.11

Table 7: More results on the entire corpus, all use DNASequitur for the grammar inference step. "M" stands for marker method symbol stream, "L" stands for LZ77-style symbol stream, "KY" stands for Kieffer-Yang improvement, and "Cost" stands for cost measure improvement

Sequence	Our best	bzip2	arith	DNACompress
CHNTXX	2.12	2.18	1.96	1.61
HEHCMVCG	2.12	2.17	1.99	1.85
HUMDYSTROP	2.16	2.18	1.95	1.91
HUMGHCSA	1.75	1.73	2.00	1.03
HUMHBB	2.05	2.14	1.97	1.79
HUMHDABCD	2.12	2.07	2.00	1.80
HUMHPRTB	2.14	2.09	1.97	1.82
MIPACGA	2.06	2.12	1.88	1.86
MPOCPCG	2.02	2.12	1.87	1.67
MPOMTCG	2.12	2.17	1.98	1.89
VACCG	2.01	2.09	1.92	1.76
YSCCHRIII	2.08	2.16	1.96	-

Table 8: Results on the entire corpus for some competing compressors. For comparison, our best version, which is the LZ77 stream after a cost measure improvement, is included. DNACompress did not provide results for YSCCHRIII.