Compression and Explanation using Hierarchical Grammars

CRAIG G. NEVILL-MANNING AND IAN H. WITTEN

Computer Science Department, University of Waikato, New Zealand
Email: cgn@cs.waikato.ac.nz, ihw@cs.waikato.ac.nz

This paper describes an algorithm, called SEQUITUR, that identifies hierarchical structure in sequences of discrete symbols and uses that information for compression. On many practical sequences it performs well at both compression and structural inference, producing comprehensible descriptions of sequence structure in the form of grammar rules. The algorithm can be stated concisely in the form of two constraints on a context-free grammar. Inference is performed incrementally, the structure faithfully representing the input at all times. It can be implemented efficiently and operates in a time that is approximately linear in sequence length. Despite its simplicity and efficiency, SEQUITUR succeeds in inferring a range of interesting hierarchical structures from naturally occurring sequences.

Received July 17, 1996; revised April 28, 1997

1. INTRODUCTION

Data compression is an eminently pragmatic pursuit: by removing redundancy, storage can be utilized more efficiently. Identifying redundancy also serves a less prosaic purpose—it provides cues for detecting structure. This paper describes an algorithm that excels at both data compression and structural inference. This algorithm is implemented in a system called SEQUITUR that deals efficiently with sequences containing millions of symbols.

SEQUITUR takes a sequence of discrete symbols and produces a set of hierarchical rules that rewrite it as a context-free grammar. Although we refer to it as a 'grammar', the result involves no generalization: it is a hierarchical structure that is capable of generating just one string, namely the original sequence. For relatively unstructured sequences such as natural language text, the top-level rule is very long, perhaps 10–20% as long as the original sequence, whereas all the other rules are fairly short, with only two or three symbols each. This is because the top-level rule contains the non-repetitive residue from the sequence after all of the compressive information has been factored out.

This technique performs well as a data compression scheme. Because it detects and eliminates redundancy, and represents structure hierarchically, it outperforms other dictionary techniques and, on very large or highly structured sequences, also outperforms standard statistical techniques. However, the raw grammar is not itself dramatically smaller than the original input, at least for sequences such as natural language text. To realize its latent compressive potential it is necessary to encode the grammar in a fairly sophisticated manner.

The algorithm is technically interesting for four reasons. First, it can be stated very concisely, in the form of two constraints on a context-free grammar. Second, it operates in a time that is approximately linear with the length of the sequence—a property that we take for granted in compression algorithms but which is rare in grammatical inference. Third, the concise description of the algorithm permits a compact, intuitively appealing, proof of its efficiency. Fourth, inference is performed incrementally, so that the structure faithfully reflects the original at all points during processing.

Despite its simplicity and efficiency, SEQUITUR succeeds in inferring a range of interesting hierarchical structures from naturally occurring sequences, and we describe structural inference from sequences in three different languages. The technique has also been demonstrated to produce instructive explanations of structure in computer programs and recursive grammars, and to optimize graphical rendering. Moreover, it is possible to use the system of rules as a basis for generalization and thereby generate an even more compact generalized grammar which can, for semi-structured sequences, provide even better explanation and compression. This work bears similarity to Wolff's MK10 system [1], but SEQUITUR's computational efficiency allows it to make inferences from significantly longer sequences.

This paper proceeds as follows. We begin by describing the algorithm in terms of the two constraints that underlie it. Because of the dual objectives of the SEQUITUR system—explanation and compression—we evaluate it in two ways: qualitatively by examining the structures that it infers from several sequences, and quantitatively with respect to its compression performance on a standard corpus and on several other structured sequences. Finally, we describe an extended case study of how the rules produced by SEQUITUR can be generalized into a true grammar by detecting discontiguous dependencies. This produces a more

(a)

Sequence	Grammar
S → abcdbc	S → aAdA
	A → bc

(b)

Sequence	Grammar
S → abcdbcabcdbc	S → AA
	A → aBdB
	B → bc

(c)

Sequence	Grammar
S → abcdbcabcdbc	S → AA
	A → abcdbc
	S → CC
	A → bc
	B → aA
	C → BdA

FIGURE 1. Example sequences and grammars that reproduce them: (**a**) a sequence with one repetition; (**b**) a sequence with a nested repetition; (**c**) two grammars that violate the two constraints.

compact structural description that can result in extremely high compression performance. More information about all aspects of SEQUITUR and its application can be found in [2], and the system itself can be used interactively at `www.cs.waikato.ac.nz/sequitur`.

2. FORMING A HIERARCHY OF REPETITIONS

SEQUITUR produces a grammar based on repeated phrases in the input sequence. Each repetition gives rise to a rule in the grammar, and is replaced by a non-terminal symbol, producing a more concise representation of the sequence. It is the pursuit of brevity that drives the algorithm to form and maintain the grammar, and, as a by-product, provide a structural explanation of the sequence.

For example, on the left of Figure 1a is a sequence that contains the repeating string *bc*. Note that the sequence is already a grammar—a trivial one with a single rule. To compress the sequence, a new rule $A \rightarrow bc$ is formed, and both occurrences of *bc* are replaced by *A*. The new grammar is shown on the right of Figure 1a.

The sequence in Figure 1b shows how rules can be reused in longer rules. It is formed by concatenating two copies of the sequence in Figure 1a. Since it represents an exact repetition, compression can be achieved by forming the rule $A \rightarrow abcdbc$ to replace both halves of the sequence. Further gains can be made by forming the rule $B \rightarrow bc$ to compress rule *A*. This demonstrates the advantage of treating the sequence, rule *S*, as part of the grammar—rules may be formed in rule *A* in an analogous way to rules formed from rule *S*. These rules within rules constitute the grammar's hierarchical structure.

The grammars in Figure 1a and b share two properties:

- p_1, no pair of adjacent symbols appears more than once in the grammar, and
- p_2, every rule is used more than once.

Property p_1 can be restated as 'every digram in the grammar is unique', and will be referred to as digram uniqueness. Property p_2 ensures that a rule is useful, so it will be called rule utility. These two constraints exactly characterize the grammars that SEQUITUR generates.

For example, the sequence in Figure 1a contains the repeated digram *bc*. To conform to property p_1, rule *A* is created, so that *bc* occurs only within rule *A*. Every digram in the sequence in Figure 1b appears elsewhere in the sequence: the creation of rule *A* leaves only one repetition, which is taken care of by rule *B*. Property p_2 allows rules longer than two symbols to be formed, as we will describe in Subsection 2.2. To show what happens when these properties are violated, Figure 1c gives two other grammars that represent the sequence in Figure 1b, but lack one of the properties. The first grammar contains two occurrences of *bc*, so p_1 does not hold. In this case, there is redundancy because *bc* appears twice. In the second grammar, *B* is used only once, so p_2 does not hold. If *B* were removed, the grammar would shrink by one rule and one symbol, forming a more concise grammar.

SEQUITUR's operation consists of ensuring that both properties hold. When describing the algorithm, the properties will be referred to as constraints. The algorithm operates by enforcing the constraints on a grammar: when the digram uniqueness constraint is violated, a new rule is formed, and when the rule utility constraint is violated, the useless rule is deleted. The next two sections describe in detail how this is performed.

2.1. Digram uniqueness

When a new symbol is observed, it is appended to rule *S*. The last two symbols of rule *S*—the new symbol and its predecessor—form a new digram. If this digram occurs elsewhere in the grammar, the first constraint has been violated. To remedy this, a new rule is formed with the digram on the right-hand side, headed by a new non-terminal. The two original digrams are replaced by this non-terminal.

The algorithm operates incrementally, and in order to illustrate this Figure 2 shows the grammar as new symbols are added in the sequence *abcdbcabcd*. The left-most column states the action that has been taken to modify the grammar—either observing a new symbol and appending it to rule *S*, or enforcing a constraint. The next column shows the sequence observed so far. The third column gives the grammar created from the sequence. The fourth column lists any duplicate digrams, and the final column lists any underused rules.

When the final *c* is added in Figure 2a, the digram *bc* appears twice. The new rule *A* is created, with *bc* as its right-hand side. The two occurrences of *bc* are replaced by *A*. This illustrates the basic procedure for dealing with duplicate digrams.

The appearance of a duplicate digram does not always result in a new rule. If the new digram matches the right-hand side of a rule exactly, no new rule need be created: the digram is replaced by the non-terminal that heads the

	new symbol or action	the string so far	resulting grammar	duplicate digrams	underused rules
(a)	a	a	S → a		
	b	ab	S → ab		
	c	abc	S → abc		
	d	abcd	S → abcd		
	b	abcdb	S → abcdb		
	c	abcdbc	S → abcdbc	bc	
	enforce digram uniqueness		S → aAdA A → bc		
(b)	a	abcdbca	S → aAdAa A → bc		
	b	abcdbcab	S → aAdAab A → bc		
	c	abcdbcabc	S → aAdAabc A → bc	bc	
	enforce digram uniqueness		S → aAdAaA A → bc	aA	
(c)	enforce digram uniqueness	abcdbcabc	S → BdAB A → bc B → aA		
(d)	d	abcdbcabcd	S → BdABd A → bc B → aA	Bd	
	enforce digram uniqueness		S → CAC A → bc B → aA C → Bd		B
	enforce rule utility		S → CAC A → bc C → aAd		

FIGURE 2. Operation of the two grammar constraints: (**a**) enforcing digram uniqueness by creating a new rule; (**b**) re-using an existing rule; (**c**) forming a hierarchical grammar; (**d**) producing a longer rule by enforcing rule utility.

existing rule. Figure 2b demonstrates the changes that occur in the grammar when a third *bc* digram appears: the existing non-terminal *A* is substituted for the third occurrence of *bc*. This results in a new pair of repeating digrams, *Aa*, shown in the last line of Figure 2b. In Figure 2c a new rule *B* is formed accordingly, with *aA* as its right-hand side, and the two occurrences of *aA* are replaced by *B*. The right-hand side of this new rule not only contains terminals, but also non-terminals referring to other rules.

The hierarchy is formed and maintained by an iterative process: the substitution of *A* for *bc* resulted in the new digram *aA*, which was itself replaced by *B*. For larger sequences, these changes ripple through the grammar, forming and matching longer rules higher in the hierarchy.

2.2. Rule utility

Until now, the right-hand sides of rules in the grammar have contained only two symbols. Longer rules are formed by the effect of the rule utility constraint, which ensures that every rule is used more than once. Figure 2d demonstrates the formation of a longer rule. When *d* is appended to

```
As each new input symbol is observed, append it to rule S.

Whenever a duplicate digram appears,
    if the other occurrence is a complete rule,
        replace the new digram with the non-terminal that heads the other digram,
    otherwise
        form a new rule and replace both digrams with the new non-terminal.

Whenever a rule is used only once,
    remove the rule, substituting its contents in place of the non-terminal.
```

FIGURE 3. The entire SEQUITUR algorithm.

rule *S*, the new digram *Bd* causes a new rule, *C*, to be formed. However, the inclusion of this rule leaves only one appearance of rule *B*, violating the second constraint. For this reason, *B* is removed from the grammar, and its right-hand side is substituted in the one place where it occurs. Removing *B* means that rule *C* now contains three symbols. This is the mechanism for creating long rules: form a short rule temporarily, and if subsequent symbols continue the match, allow a new rule to supersede the shorter rule, and delete the shorter rule.

2.3. Efficient implementation

This simple algorithm—two constraints triggering two responses—can be implemented efficiently, and processes sequences at about 2×10^6 symbols/min on a workstation.

The basic operations involved are:

- appending a symbol to rule *S*;
- using an existing rule;
- creating a new rule;
- deleting a rule.

Appending a symbol involves lengthening rule *S*. Using an existing rule involves substituting a non-terminal for a digram of two symbols, thereby shortening the rules containing the digram. Creating a new rule involves making a new non-terminal for the left-hand side, as well as inserting two new symbols as the right-hand side. After creating the rule, substitutions are made as for an existing rule. Deleting a rule involves moving its contents to replace a non-terminal, which lengthens the rule containing the non-terminal. The left-hand side of the rule must then be deleted. Using appropriate doubly linked list data structures, each of these operations can be performed very quickly.

To enforce the rule utility constraint, a usage count is recorded for each rule and a rule is deleted when its count drops to one. To enforce the digram uniqueness constraint, whenever a new digram appears, i.e. as each symbol is appended to rule *S*, the grammar must be searched for any other occurrence of it. This is accomplished by storing all digrams in a hash table, handling collisions using 'open addressing' [3], along with the standard technique of double hashing. Every time a new digram appears in the grammar, it is added to the index. A new digram appears as the result of two pointer assignments linking two symbols together in the doubly linked list. Thus updating the index can be incorporated into the low-level pointer assignments. A digram also disappears from the grammar whenever a pointer assignment is made—the pointer value that is overwritten by the assignment represents a digram that no longer exists.

Figure 3 summarizes the algorithm. Line 1 deals with new observations in the sequence. Lines 2 through 6 enforce the digram utility constraint. Line 3 determines whether the new digram matches an existing rule, or whether a new rule is necessary. Lines 7 and 8 enforce rule utility. Lines 2 and 7 are triggered whenever the constraints are violated.

As well as being efficiently implementable, it can be shown that the algorithm operates in a time which is linear in the length of the input string [4]. The basic idea of the proof is this: the two constraints both have the effect of reducing the number of symbols in the grammar, so the amount of work done when satisfying them is bounded by the compression achieved on the sequence. The saving cannot exceed the original size of the input sequence, so the algorithm is linear in the number of input symbols.[1]

3. QUALITATIVE EVALUATION IN TERMS OF EXPLANATION

Figures 4–6 show three structures discovered from language, music and the output of a biological modelling system respectively. We describe each in turn.

SEQUITUR was used to infer the morphological structure of English, French and German versions of the Bible. The original texts are between 4 and 5×10^6 characters long, and the resulting grammars consist of about 100 000 rules and 600 000 symbols. As noted earlier, the first rule, rule *S*,

[1]A minor caveat is that this result is based on a register model of computation rather than a bitwise one. It assumes that the average lookup time for the hash table of digrams is bounded by a constant; this is only true if hash function operations are register-based. In practice, with a 32-bit architecture the linearity proof remains valid for sequences of up to around 10^9 symbols, and for a 64-bit architecture up to 10^{19} symbols.

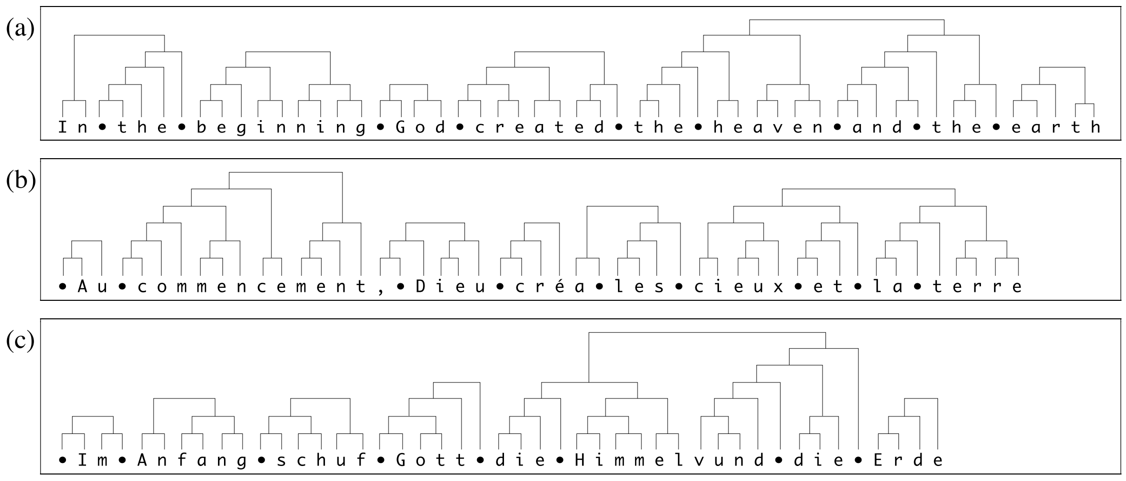

(a)
I n • t h e • b e g i n n i n g • G o d • c r e a t e d • t h e • h e a v e n • a n d • t h e • e a r t h
(b)
• A u • c o m m e n c e m e n t , • D i e u • c r é a • l e s • c i e u x • e t • l a • t e r r e
(c)
• I m • A n f a n g • s c h u f • G o t t • d i e • H i m m e l v u n d • d i e • E r d e

FIGURE 4. Hierarchies for Genesis 1:1 in (**a**) English, (**b**) French and (**c**) German.

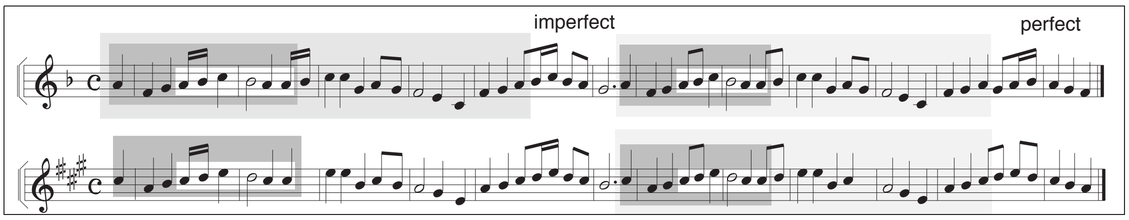


FIGURE 5. Illustration of matches within and between two chorales: for chorales *O Welt, sieh hier dein leben* and *O Welt, Ich muss Dich lassen* by J. S. Bach.

is very long, whereas all the others are fairly short. In these grammars, rule *S* contains about 400 000 symbols, accounting for two thirds of the total size of the grammar.

Figure 4 shows parts of the hierarchies inferred from the three Bibles. Each branch in the tree represents a rule in the grammar, with its children representing the right-hand side of the rule. Spaces are made explicit as bullets. At the top level, rule *S*, of the English version, the verse is parsed into six subsequences: *In the, beginning, God, created, the heaven and the* and *earth*. Four of these are words, and the other two are groups of words. *In the* is broken up, at the next level down, into *In* and *the*. The other phrase, *the heaven and the* is split into *the heaven* and *and the*. The words *beginning* and *created* consist of roots and affixes: *beginning* is split into *begin* and *ning*, while created is split into *creat* and *ed*. The root of *beginning* is *begin*, but the normal form of the suffix is *ing* rather than *ning*. SEQUITUR has no way of learning the consonant doubling rule that English follows to create suffixes, so other occurrences of words ending in *ning* cause this rule to be formed. Similarly, whereas *create* is the infinitive of the verb, *ed* is usually the affix for past tense, so the division makes lexical, if not linguistic, sense. For the most part, then, SEQUITUR segments English plausibly.

SEQUITUR is language independent—the two constraints on the grammar are not designed to favour English. The German version of the same verse is split into words and phrases, and eventually into words. In fact, the hierarchy for the phrase *die Himmel und die Erde* is very similar to the hierarchy for the English equivalent. The French version is split correctly, with *commencement* being broken into the root *commence* and the suffix *ment*, analogously to *beginning*.

Figure 5 shows two chorales, *O Welt, sieh hier dein leben* and *O Welt, Ich muss Dich lassen*, harmonized by J. S. Bach [5]. Shaded boxes represent rules that SEQUITUR infers after processing a corpus of chorales (which included these two), and the nesting of the boxes indicates the hierarchy of rules. The light grey boxes highlight the similarity of the first and second half of the first chorale, and the second half of the second chorale. In fact, these chorales are harmonized from the same original melody, and have been transposed and rhythmically altered for the lyrics. The four darker boxes show the common parts of all four halves, and the white box indicates a shorter motif that is employed in other chorales. SEQUITUR also forms rules for imperfect and perfect cadences by comparison between chorales.

In order to model the topology and growth patterns of living things, Aristid Lindenmayer created a class of rewriting systems called L-systems [6] which excel at capturing fractal graphical structure. The L-system in Figure 6a evaluates to the sequence in Figure 6b, which, when interpreted as LOGO commands, draws the plant in Figure 6d. From this sequence

(a)
```
S → f
f → f[+f]f[—f]f
```

(b)
```
f[+f]f[-f]f[+f[+f]f[-
f]f]f[+f]f[-f]f[-f[+f
]f[-f]f]f[+f]f[-f]f[+
ff[+f]f[-f]f[+f[+f]f[-
-f]f]f[+f]f[-f]f[-f[+
f]f[-f]f]f[+f]f[-f]f]
f[+f]f[-f]f[+f[+f]f[-
f]f]f[+f]f[-f]f[-f[+f
]f[-f]f]f[+f]f[-f]f[-
f[+f]f[-f]f[+f[+f]f[-
f]f]f[+f]f[-f]f[-f[+f
]f[-f]f]f[+f]f[-f]f]f
[+f]f[-f]f[+f[+f]f[-f
]f]f[+f]f[-f]f[-f[+f]
f[-f]f]f[+f]f[-f]f
```

(c)
```
S → BFAGA
A → B]B
B → DFCGC
C → D]D
D → fFEGE
E → f]f
F → [+
G → [-
```

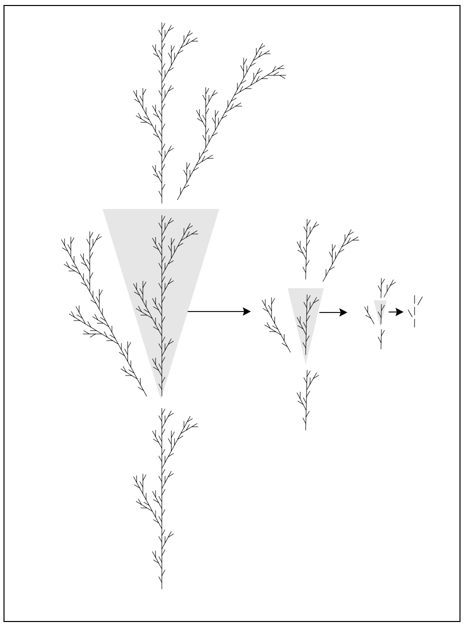
(d)


FIGURE 6. Inference of grammars from L-system output: (**a**) an L-system; (**b**) a string produced by the L-system; (**c**) non-recursive grammar inferred by SEQUITUR; (**d**) graphical interpretation of (b).

SEQUITUR produces the grammar in Figure 6c, which reflects the self-similarity of the sequence at different levels in the similarity of rules *S*, *B* and *D*. This grammar can be used to infer the original recursive L-system by unifying similar rules. It can also be used to form a graphical hierarchy that can significantly optimize graphical rendering.

4. QUANTITATIVE EVALUATION IN TERMS OF COMPRESSION

The two constraints—digram uniqueness and rule utility—ensure that redundancy due to repetition is eliminated from a sequence. Digram uniqueness eliminates a repetition of two symbols by forming a rule that both occurrences can reference. Rule utility eliminates superfluous rules when repetitions continue for longer than two symbols.

The novel *Far from the Madding Crowd* by Thomas Hardy is a benchmark sequence for data compression schemes. As expressed in the file `book1` as part of the Calgary corpus [7], it is 768 771 bytes long. The grammar that SEQUITUR forms from it has 27 365 rules, with right-hand sides which have an average length of 1.97 symbols (excluding the first rule *S*). Rule *S* contains 131 416 symbols, and there are 185 253 symbols in the entire grammar (not counting the symbols that introduce each rule, as these can be reconstructed from the sequence in which the grammar is written). Thus by forming rules from repetitions, SEQUITUR reduces the number of symbols to 25% of its original value.

Unfortunately, the alphabet from which symbols are drawn is greatly expanded, since names for 27 000 rules must be added. Simply numbering rules with codes like '#42' (for the 42nd rule) turns a reduction to 25% in symbol count into an expansion to 150%. Of course, the distribution of symbols will be highly skewed, and symbol counts are an extremely poor indicator of compressibility.

We will describe two methods for encoding SEQUITUR's grammar. The first is straightforward but performs poorly, while the second outperforms a good macro compressor, `gzip`, and rivals a good statistical compressor, PPMC.

4.1. Simple encoding method

To encode the grammar, we dispense with the textual representation and consider transmitting it as a sequence of symbols from a large alphabet, encoding each according to its probability of occurrence. A symbol that occurs with probability p can be encoded in $\log_2 p$ bits, and arithmetic coding is a practical method that can approach this bound arbitrarily closely in the limit [8]. Statistical methods such as PPM that use arithmetic coding condition the probability of a symbol on the preceding symbols [9]. Because no digram appears twice in the grammar produced by SEQUITUR, this approach yields no gain. For that reason, a single distribution based on the frequency of each symbol—in other words, an order-zero model—is used to encode the grammar in the first instance.

Forming a grammar for the novel and encoding it using an order-zero frequency model results in a compression rate of 3.49 bits per character (b.p.c.)—a rather disappointing figure when compared with standard compression methods. For example, UNIX `compress` achieves a compression rate of 3.46 b.p.c.; `gzip` achieves 3.25 b.p.c. and a good general-purpose compression scheme, PPMC, reduces the novel to 2.52 b.p.c.

In principle, SEQUITUR should be able to rival the best dictionary schemes, because forming rules is similar to forming a dictionary. Furthermore, since SEQUITUR stores its dictionary as a hierarchy it should be capable of outperforming other dictionary techniques. We now describe how this can be achieved by sending the grammar implicitly.

4.2. Implicit encoding method

Rather than sending a list of rules, it is better to adopt an implicit encoding technique that sends the sequence, and whenever a rule is used, transmits sufficient information to the decoder for it to reconstruct the rule. Because rule S represents the entire sequence, this is tantamount to sending rule S and transmitting other rules as they appear. When a non-terminal is encountered in rule S, it is treated in three different ways depending on how many times it has been seen. The first time it occurs, its contents are sent. At this point, the decoder is unaware that the symbols will eventually become a rule. On its second occurrence, a pointer is sent that identifies the contents of the rule that was sent earlier. The pointer consists of an offset from the beginning of rule S and the length of the match, similar to the pointers used in LZ77 [10]. At the decoding end, this pointer is interpreted as an instruction to form a new rule, with the target of the pointer comprising the contents of the rule. The decoder numbers rules in the order in which they are received, and the encoder keeps track of this numbering. On the third and subsequent occurrences of the non-terminal, this number is used to identify the non-terminal.

The advantage of this approach is that the first two times a rule is used, the non-terminal that heads it need not be sent. For example, under the previous scheme, a rule that is used twice is transmitted by sending two non-terminals, the rule contents and an end-of-rule marker. Under the new scheme, only the contents and a pointer are necessary. Furthermore, rules are sent only when they are needed. If the grammar were processed rule by rule, starting with rule S, rules would either be encountered before they were referenced, in which case a code would be reserved unnecessarily, or referenced before they were sent, in which case the decoder's reconstruction of the sequence would be delayed.

Sending the grammar for `book1` implicitly yields a compression rate of 2.82 b.p.c. This is better than the other dictionary techniques and only 12% worse than PPMC.

The sequence in Figure 1b is transmitted using this scheme as $abcd(1,2)(0,4)$. Because both rules A and B only occur twice, no non-terminals appear in the encoding. The sequence is sent by transmitting rule S, which consists of two instances of rule 2. The first time rule 2 appears, its contents are transmitted. This consists of $a①d①$. The first symbol, a, is encoded normally. The first time rule 1 appears, its contents, bc, are sent. The next symbol, d, is sent as normal. Now rule 1 appears for the second time, and the pointer $(1,2)$ is sent. The first element of the pointer is the distance from the start of the sequence to the start of the first occurrence of bc, in this case 1. The second element of the pointer is the length of the repetition: 2. Now the decoder forms a rule $1 \rightarrow bc$, and replaces both instances of bc in the sequence with ①. Having transmitted the first instance of rule 2 in its entirety, the encoder returns to rule S to transmit the second occurrence of rule 2. The repetition starts at the beginning of the sequence, at distance 0, and continues for 4 symbols. The length refers to the 4-symbol compressed sequence, rather than the uncompressed repetition, which is 6 symbols long. A more detailed description can be found in [11].

4.3. Comparison with macro schemes

The phrases that are discovered improve on the dictionaries of macro-based compression schemes in four ways. First, the dictionary is stored hierarchically, using shorter dictionary entries as part of longer ones. Second, there is no window to reduce searching time and memory usage at the expense of forgetting useful repetitions. Third, the length of dictionary entries is not limited. Fourth, there are no unnecessary phrases in the dictionary. Each of these advantages is expanded below.

Using a hierarchical representation for the dictionary means that rules can be transmitted more efficiently. The saving comes from the smaller pointer needed to specify both the start of the repetition and its length. Because rules properly contain other symbols, i.e. they do not overlap other non-terminals, the number of places a rule can start is reduced to the number of symbols currently in the grammar. Furthermore, the length of the repetition is expressed in terms of the number of terminal and non-terminal symbols that it spans, rather than the number of original terminals. This means that the length will usually be shorter than the corresponding length specified relative to the original sequence. This corresponds to Storer and Szymanski's compressed pointer macro classification [12].

The lack of a finite window for pointer targets has several ramifications. First, it undoes some of the improvement achieved by using a hierarchical dictionary, because it allows a greater number of targets for pointers. Second, the lack of windowing usually means that memory usage and search time both grow. SEQUITUR's memory usage is linear in the length of the input string: this is unavoidable given the basic design of the algorithm. However, SEQUITUR's linked-list data structures and digram indexing scheme mean that the average search time is bounded. The advantage of the lack of a window is that all repetitions can be detected, no matter how far they are separated.

LZ78 techniques (described in [13]) add items to the dictionary speculatively—a particular entry is not guaranteed to be used. This saves the cost of specifying which phrases should be included in the dictionary, but means that the codes assigned to unused entries are wasted. SEQUITUR only forms a rule when repetitions occur, combining the LZ77 policy of specifying a repetition only when needed with the LZ78 technique of maintaining a dictionary. Furthermore, LZ78 techniques grow the dictionary slowly, whereas SEQUITUR can send a rule of any length in one step. This does not always result in superior compression, as the results in the next section illustrate.

TABLE 1. Performance of various compression schemes (bits/character)

Name	Description	Size	`Compress`	`Gzip`	SEQUITUR	PPMC
`bib`	Bibliography	111 261	3.35	2.51	2.48	**2.12**
`book1`	Fiction book	768 771	3.46	3.25	2.82	**2.52**
`book2`	Non-fiction book	610 856	3.28	2.70	2.46	**2.28**
`geo`	Geophysical data	102 400	6.08	5.34	**4.74**	5.01
`news`	USENET	377 109	3.86	3.06	2.85	**2.77**
`obj1`	Object code	21 504	5.23	3.84	3.88	**3.68**
`obj2`	Object code	246 814	4.17	2.63	2.68	**2.59**
`paper1`	Technical paper	53 161	3.77	2.79	2.89	**2.48**
`paper2`	Technical paper	82 199	3.52	2.89	2.87	**2.46**
`pic`	Bi-level image	513 216	0.97	**0.82**	0.90	0.98
`progc`	C program	39 611	3.87	2.68	2.83	**2.49**
`progl`	Lisp program	71 646	3.03	**1.80**	1.95	1.87
`progp`	Pascal program	49 379	3.11	**1.81**	1.87	1.82
`trans`	Shell transcript	93 695	3.27	**1.61**	1.69	1.75
Average			3.64	2.69	2.64	**2.49**
L-systems		908 670	0.38	0.07	**0.01**	0.32
Amino acids		1 586 289	4.52	4.08	**3.28**	3.54
Bible	King James version	4 047 392	2.77	2.32	**1.84**	1.92

4.4. Compression performance

The results for four compression methods—`compress`, `gzip`, the standard implementation of PPMC and SEQUITUR—for the Calgary corpus are shown in Table 1. The best figure for each row is shown in bold. Overall, SEQUITUR outperforms all schemes other than PPMC, which is 6% better. Although it beats `gzip` on average, it is worse on 8 out of the 14 individual files. SEQUITUR beats PPMC on the geophysical data, the picture and the transcript. While PPMC is better able to detect subtle probabilistic relationships between symbols that typically appear in highly variable sequences such as text, SEQUITUR excels at capturing exact, long repetitions that occur in highly structured files—although direct comparison is somewhat unfair to PPMC because it was run with standard parameters that restrict memory usage to well below that consumed by SEQUITUR. While SEQUITUR does not perform as well as PPMC on text such as `book1` and `book2`, it outperforms it on longer text such as the King James version of the Bible, shown in the last row of Table 1.

For highly structured sequences such as the output from L-systems, SEQUITUR performs very well indeed. The row labelled L-systems in Table 1 shows the compression performance of various schemes on the output of context-sensitive, stochastic L-systems. PPMC performs the worst because it fails to capitalize on the very long repetitions that exist. `Gzip` performs 2–4 times better than PPMC. Surprisingly, the textual version of SEQUITUR's grammar is 13 times smaller than PPMC's output and 3 times smaller than `gzip`'s output. The encoded version of SEQUITUR's grammar is 1300 times smaller than the original.

The second to last row of Table 1 shows compression performance on a sequence of amino acids specified by parts of human DNA. SEQUITUR outperforms the next best scheme, PPMC, by 7%. This implies that SEQUITUR captures structure ignored by the other schemes. The biological significance of this structure is a matter for future work, in collaboration with biochemists.

5. DISCONTIGUOUS DEPENDENCIES IN SEMI-STRUCTURED TEXT

Thus far, we have discussed how the repetition of contiguous subsequences of symbols can be detected and exploited. In realistic settings, relationships occur between symbols, and between subsequences, that are not adjacent to each other. We introduce here a particular kind of input, semi-structured text, which is ubiquitous and generally exhibits this kind of dependency quite strongly.

We define semi-structured text as data that is both readable by humans and suitable for automatic processing by computers [14]. A widespread example of this is the hypertext markup language HTML. This consists of free text interspersed with structural information specified by markup tags. Tags are drawn from a limited set of reserved words, and the sequence of tags throughout a document is intended to conform to a prescribed grammar. Two kinds of sequence therefore coexist: at one level there is relatively unpredictable free text, whereas at another the markup is highly structured.

The situation is even more pronounced in some semi-structured databases. In the remainder of this section we will study two genealogical databases maintained by the Latter Day Saints Church: the International Genealogical Index, which stores birth, death and marriage records, and the Ancestral File, which stores linked pedigrees for families all over the world. The former contains the records of over

```
0 @26DS-KX@ INDI
1 AFN 26DS-KX
1 NAME Dan Reed /OLSEN/
1 SEX M
1 BIRT
2 DATE 22 JUN 1953
2 PLAC Idaho Falls,Bonneville,Idaho
1 FAMC @00206642@
0 @00206642@ FAM
1 HUSB @NO48-3F@
1 WIFE @93GB-DD@
1 CHIL @26DS-KX@
1 CHIL @21B7-WR@
0 @26DN-7N@ INDI
1 NAME Mary Ann /BERNARD
```

FIGURE 7. An excerpt from the GEDCOM genealogical database [15], with structure and content distinguished.

265 million people, while the latter contains records for over 21 million; they are growing at a rate of 10–20% per year.

Figure 7 shows an example of an individual record, a family record and the beginning of another individual record. The first gives name, gender, birth date, birthplace and a pointer to the individual's family. The family record, which follows directly, gives pointers to four individuals: husband, wife and two children—one of which is the individual himself. This example, however, gives an impression of regularity which is slightly misleading. For most of the information-bearing fields such as NAME, DATE and PLACE, there are records that contain free text rather than structured information. For example, the last line of Figure 7 shows a name given with an alternative. The DATE field might be 'Abt 1767' or 'Will dated 14 Sep 1803'. There is a NOTE field (with a continuation line code) that frequently contains a brief essay on family history. Nevertheless, the tags such as NAME and DATE and the level numbers at the beginning of each line are strongly related, and certain template structures do recur. The purpose of this section is to show how such regularities can be exploited.

5.1. Learning the structure of the genealogical database

The data was presented to SEQUITUR as a sequence of words where words were viewed as sequences of characters delimited by a single space (thus words sometimes began with spaces). The dictionary was encoded separately from the word sequence, which was represented as a sequence of numeric dictionary indexes. The input comprised 1.8×10^6 words, and the dictionary contained 148 000 unique entries. The grammar that SEQUITUR formed had 71 000 rules and 648 000 symbols, 443 000 of which were in the top-level rule. The average length of a rule (excluding the top-level one) was nearly three words.

Examination of SEQUITUR's output reveals that significant improvements could be made quite easily by making small changes to the organization of the input file. We first describe how this was done manually, by using human insight to detect regularities in SEQUITUR's output; next, we show how the grammar can be interpreted and finally, we show how the process of identifying such situations can be automated.

5.2. Manual generalization

Of the dictionary entries, 94% were codes used to relate records together for various familial relationships. Two types of code are used in the database: individual identifiers such as @26DS-KX@, and family identifiers such as @00206642@. These codes obscure template structures in the database—the uniqueness of each code means that no phrases can be formed that involve them. For example, the line

0 @26DS-KX@ INDI

in Figure 7 occurs only once, as do all other INDI lines in the file, and so the fact that 0 and INDI always occur together is obscured: SEQUITUR cannot take advantage of it. In fact, the token INDI occurs 33 000 times in the rules of the grammar, and in every case it could have been predicted with 100% accuracy by noting that 0 occurs two symbols previously, and that the code is in the individual identifier format.

This prediction can be implemented by replacing each code with the generic token *family* or *individual* and specifying the actual codes that occur in a separate stream. Replacing the code in the example above with the token *individual* yields the sequence

0 individual INDI,

which recurs many thousands of times in the file and therefore causes a grammar rule to be created. In this grammar, INDI occurs in a rule that covers the phrase

↲0 individual INDI ↲1 AFN individual ↲1 NAME.

This is now the only place that INDI occurs in the grammar.

Overall, this strategy halves the number of rules in the grammar, the length of the top-level rule, and the total number of symbols.

5.3. Interpreting the grammar

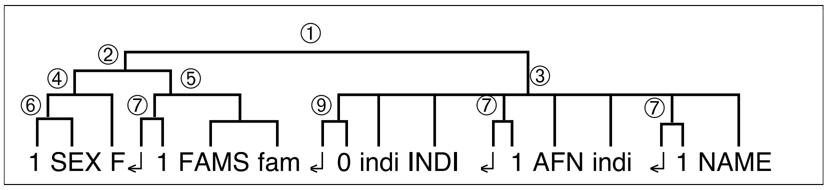
Figure 8 shows nine of the 35 000 rules in SEQUITUR's grammar for the sequence with generalized codes, renumbered for clarity. Rule ① is the second most widely used rule in the grammar: it appears in 261 other rules.[2] The other eight rules are all those that are referred to, directly

[2]The most widely used rule is 2 PLAC, which occurs 358 times, indicating that the text surrounding the place tag is highly variable. However, the structure of the rule itself is uninteresting.

(a)

```
① → ②③
② → ④⑤
③ → ⑨individual INDI⑦AFN individual⑦NAME
④ → ⑥ F
⑤ → ⑦⑧
⑥ → 1 SEX
⑦ → ↵1
⑧ → FAMS family
⑨ → ↵0
```

(b)



(c)

... ① Sybil Louise /MCGHIE/ (① Eliza) Jane /LOSEE/ (① Margaret) /SIMMONS/
(① Marie) Elizabeth /BERREY/ ① Athena Augusta /ROGERS/ (① William) Henry /WINN/ ...

FIGURE 8. A phrase from the genealogical database: (**a**) hierarchical decomposition; (**b**) graphical representation; (**c**) examples of use.

or indirectly, by rule ①: Figure 8b shows the hierarchy graphically. The topmost line in Figure 8b represents rule ①. The two branches are rules ② and ③, the contents of rule ①. The hierarchy continues in this way until all of the rules have been expanded.

Rule ① represents the end of one record and the beginning of the next. Rule ⑨ is effectively a record separator (recall that each new record starts with a line at level 0), and this occurs in the middle of rule ①. Although grouping parts of two records together achieves compression, it violates the structure of the database, in which records are integral. However, the two parts are split apart at the second level of the rule hierarchy, with one rule, ②, for the end of one record, and another, ③, for the start of the next. The short rules ⑦ and ⑨ capture the fact that every line begins with a nesting level number. There is also a rule for the entire SEX field indicating the person is female, which decomposes into the fixed part 1 SEX and the value F on the end, so that the first part can also combine with M to form the other version of the SEX field. There is a similar hierarchy for the end of a male record, which occurs 259 times.

As for the usage of this rule, Figure 8c shows part of rule S. Here, rules have been expanded for clarity: parentheses are used to indicate a string which is generated by a rule. This part of the sequence consists mainly of rule ① in combination with different names. Separate rules have been formed for rule ① in combination with common first names.

5.4. Automatic generalization

In order to automate the process of identifying situations where generalization is beneficial, it is first necessary to define the precise conditions that give rise to possible savings. In the case described above, the rule

INDI ↵1 AFN

occurred many times in the grammar, and accounted for a significant portion of the compressed file. Conditioning this phrase on a prior occurrence of ↵0 greatly increases its predictability. The problem is that other symbols may be interposed between the two. One heuristic for identifying potential savings is to scan the grammar for pairs of phrases where the cost of specifying the distances of the second relative to the first (predictive coding) is less than the cost of coding the second phrase by itself (explicit coding).

Figure 9 gives two illustrations of the tradeoff. In the top half of Figure 9, using normal coding, the cost of coding the Bs is 3 times the cost of coding an individual B: $\log_2$(frequency of B/total symbols in grammar) bits. For predictive coding, the statement 'A predicts B' must be encoded once at the beginning of the sequence. Reducing this statement to a pair of symbols, AB, the cost is just the sum of encoding A and B independently. Each time that A occurs, it is necessary to specify the number of intervening symbols before B occurs. In the example, $A\langle 3\rangle$ signifies that the next B occurs after three intervening symbols. These distances are encoded using an adaptive order-0 model with escapes to introduce new distances.

The bottom half of Figure 9 shows a more complex example, where two As appear with no intervening B, and a B occurs with no preceding A. The first situation is flagged by a distance of ∞, and the second is handled by encoding B using explicit coding.

		explicit	predictive
to encode the Bs in	encode	... B ... B ... B ...	‘A predicts B’, A<3> ... A<4> ... A<4>
AabcB ... AdefgB ... AhijkB	cost	3 × cost(B)	cost(A) + cost(B) + cost(3, 4, 4)

		explicit	predictive
to encode the Bs in	encode	... B ... B ... B	‘A predicts B’, A<3> ... A<∞> ... A<4> ... B
AabcB ... A ... AhijkB ... B	cost	3 × cost(B)	cost(A) + cost(B) + cost(3, ∞, 4) + cost(B)

FIGURE 9. Examples of two ways of coding symbol *B*.

TABLE 2. Predictions based on part of the GEDCOM database [15]

	Prediction	Normal (bits/symbol)	Predicted (bits/symbol)	Saving (total bits)
1	↵0 … INDI ↵1 AFN	3.12	0.01	2298
2	INDI ↵1 AFN … ↵1 NAME	3.12	0.01	2298
3	FAMS … ↵0	2.25	0.81	638
4	↵1 SEX … ↵2	0.96	0.06	656
5	BAPL … ↵1 ENDL	1.57	0.76	509
6	FAMC … ↵1 FAMS	2.57	1.47	427
7	↵1 BIRT↵2 DATE … ↵2 PLAC	1.88	1.11	382
8	↵1 NAME … ↵1 SEX	1.25	0.82	315
9	ENDL … ↵1 SLGC	2.15	1.58	266

5.5. Algorithm for automatic generalization

The procedure for identifying useful generalizations considers every pair of symbols as a candidate for generalization, where symbols include both terminals and non-terminals; thus the algorithm is quadratic in the number of rules. It works as follows.

First, as a preliminary step, a list is made for every symbol (whether terminal or non-terminal) of all positions where that symbol occurs in the original sequence. For each rule (other than rule *S*) in which the symbol appears, all positions in which the rule occurs are added to the list for the symbol itself, so that the symbol effectively inherits the positions of rules in which it appears.

Next, for each pair of unique symbols, one symbol is chosen as the predictor and the other as the predicted symbol. The gaps between the two are calculated as in Figure 9, and the total number of bits required to encode the predicted symbol using both explicit and predictive coding is calculated. Then the predictive/predicted roles are reversed, and the savings are recorded in the corresponding cell in a matrix of symbol–symbol pairs. Once this calculation has been performed for all symbol pairs, those with greatest savings are turned into generalizations.

Table 2 shows the top ranking pairs of phrases, the number of bits required for the predicted symbol with the explicit coding the number of bits required using predictive coding, and the total savings. The ellipsis between the two phrases in the first column represents the variable content between the keywords. At the top of the list is the prediction

↵0 … INDI ↵1 AFN,

which is the relationship that we exploited by hand—the intervening symbol represented by the ellipsis is an individual code. Predictions 2, 3 and 6 indicate that the codes should be generalized after the AFN, FAMS and FAMC tags respectively. Predictions 5, 7 and 9 indicate that dates should be generalized. Prediction 4 indicates that the SEX field can be generalized by replacing the two possible tags, F and M. Finally, prediction 8 indicates that names should be generalized. The generic tokens are equivalent to non-terminals that can have all possible codes etc. as their bodies.

Figure 7 shows the content in grey and the identified structure in black. This demonstrates that the structure and content have been successfully distinguished. By identifying significant phrases and predictive relationships between them, the database template structure has been discovered. This is possible because of the reliable statistics that a large sample provides. In the next section, we will see that these inferences significantly increase compression performance.

5.6. Compressing semi-structured text

The genealogical databases are currently stored using a scheme that performs special-purpose compression designed specifically for this particular format, and compresses a typical excerpt to 16% of its original size. (Because we do

TABLE 3. Compression rates of various schemes on the genealogical data

	Scheme	Dictionary	Code indexes	Word indexes	Total size (Mb)	Compression
Original	–				9.18	100.0%
Byte-oriented	`compress`				2.55	27.8%
	`gzip`				1.77	19.3%
	PPMC				1.42	15.5%
Word-oriented	WORD-0	–	–	–	3.20	34.8%
	WORD-1	–	–	–	2.21	24.1%
	MG	0.14	–	2.87	3.01	32.8%
SEQUITUR		0.11	–	1.07	1.18	12.9%
SEQUITUR with	Codes	0.11	0.40	0.60	1.11	12.1%
generalization of	Dates	0.11	0.64	0.31	1.06	11.5%
	Gender	0.11	0.64	0.30	1.05	11.4%
	Names	0.11	0.76	0.17	1.04	11.3%

not have access to the compression system, exact figures are not available for this excerpt.) Table 3 shows the results of several compression programs on the 9 MB sample. With the sole exception of MG [16], these compression programs do not support random access to records of the database, and are not suitable for use in practice because random access is always a *sine qua non* for information collections of this size.

The first block of Table 3 summarizes the performance of several byte-oriented schemes. UNIX `compress` provides a benchmark bound on the possible compression, while `gzip` achieves substantially better compression. PPMC performs extremely well on the data, giving a compression rate of over 6:1. For all these schemes, compression rates are about twice as great as they are on `book1` from the Calgary corpus, which indicates the high regularity of this database relative to normal English text.

The next block of Table 3 summarizes the performance of some word-oriented compression schemes. These schemes split the input into an alternating sequence of words and non-words—the latter comprising white space and punctuation. WORD uses a Markov model that predicts words based on the previous word and non-words based on the previous non-word, resorting to character-level coding whenever a new word or non-word is encountered [17]. We used both a zero-order context (WORD-0) and a first-order one (WORD-1). MG is designed for full-text retrieval and uses a semi-static zero-order word-based model, along with a separate dictionary [16]. In this scheme, as in WORD-0, the code for a word is determined solely by its frequency, and does not depend on any preceding words. This proves rather ineffective on the genealogical database, indicating the importance of inter-word relationships. WORD-1 achieves a compression rate that falls between that of `compress` and `gzip`. The relatively poor performance of this scheme is rather surprising, indicating the importance of sequences of two or more words as well perhaps as the need to condition inter-word gaps on the preceding word and vice versa. None of these standard compression schemes perform as well as the original *ad hoc* special-purpose compression program, except, marginally, PPMC.

To compress the database using SEQUITUR, the sequence of word numbers is compressed using the encoding described in Subsection 4.1. The dictionary was compressed in two stages: front coding followed by PPMC. Front coding [18] involves sorting the dictionary, and whenever an entry shares a prefix with the preceding entry, replacing the prefix by its length. For example, the word *baptized* would be encoded as 7*d* if it were preceded by *baptize*, since the two have a prefix of length 7 in common. A more principled dictionary encoding was also implemented, but failed to outperform this simple approach.

The grammar, when encoded using the method described above, was 1.07 Mb in size. The dictionary compressed to 0.11 Mb, giving a total size for the whole text of 1.18 Mb, as recorded at the top of the bottom block of Table 3. This represents almost 8:1 compression, some 20% improvement over the nearest rival, PPMC.

Generalizing the codes, as described in Subsection 5.2, halves the number of rules in the grammar, the length of the top-level rule and the total number of symbols. The compressed size of the grammar falls from 1.07 Mb to 0.60 Mb. However, the codes need to be encoded separately. To do this, a dictionary of codes is constructed and compressed in the same way as the dictionary for the main text. Each code can be transmitted in a number of bits given by the logarithm of the number of entries in the code dictionary. The total size of the two files specifying the individual and family codes in this way is 0.40 Mb, bringing the total for the word indexes to 1.0 Mb, a 7% reduction over the version with codes contained in the text. Including the dictionaries gives a total of 1.11 Mb to recreate the original file.

Separating the dictionaries represents the use of some domain knowledge to aid compression, so comparisons with general-purpose compression schemes are unfair. For this reason, PPMC was applied to the same parts as SEQUITUR, to determine what real advantage SEQUITUR provides.

PPMC was first applied in a byte-oriented manner to the sequence of word indexes. It compressed these to 1.07 Mb, far worse than SEQUITUR's 0.60 Mb. In an attempt to improve the result, PPMC was run on the original file with generic tokens for codes, yielding a file size of 0.85 Mb—still much worse than SEQUITUR's result. Note that this approach does outperform running PPMC on the unmodified file. Finally, the WORD-1 scheme was applied to the sequence of generalized codes, but the result was worse still. Note, however, that the performance of PPMC and WORD could no doubt be improved by utilizing models of higher order.

Table 2 ranks possible generalizations in order of their usefulness. Predictions 1, 2, 3 and 6 encourage generalization of codes, which has been performed. Predictions 5, 7 and 9 indicate that dates should be generalized. Extracting dates in a way analogous to the extraction of codes from the main text reduces the grammar from 0.60 Mb to 0.31 Mb, and adds 0.24 Mb to specify the dates separately. This represents a net gain of 0.05 Mb, or 5%. Prediction 4 indicates that the SEX field can be generalized by replacing the two possible tags, F and M. Acting on this reduces the size by a further 1%. Finally, prediction 8 indicates that names should be generalized, resulting in a final compressed size of 1.04 Mb, or a ratio of almost 9:1. The final block of Table 3 summarizes these improvements.

6. CONCLUSION

It has often been argued that compression and learning are closely related. Over the centuries, Occam's razor has been invoked as informal justification for shaving philosophical hairs off otherwise baroque theories. In recent years, formal compressive metrics are increasingly being adopted to evaluate the results of machine learning. However, there are few, if any, general-purpose systems that combine realistic compression with useful insight into the structure of what is being compressed. Neither statistical nor macro-based compression schemes generate insightful structural descriptions of the sequences being compressed.

The SEQUITUR system presented here has a foot in both camps. It achieves respectable compression on ordinary sequences such as natural language text. It also produces structural descriptions that reveal interesting aspects of the sequences being compressed. The basic algorithm represents sequences in terms of a hierarchical structure of rewrite rules. These rules identify the phrases that occur in the string, where phrases are not defined by any statistical measure but merely correspond to sequences that occur more than once.

The algorithm is remarkable in that it operates (i) incrementally, (ii) in a time which is linear with sequence length, and (iii) efficiently enough to be used on large sequences containing many millions of symbols. These advantages stem from its underlying elegance, based on the maintenance of two simple constraints. To give an idea of the resources it consumes, on multi-Mbyte files it processes input at the rate of 2 Mbyte/min on a current workstation, and the memory required is about twice that needed to store the input.

The basic SEQUITUR algorithm provides good compression, when its output is suitably transformed by coding rules implicitly, and useful insight into the structure of sequences. More importantly, because the representation of sequences in terms of rewrite rules is so natural and transparent, it is particularly easy to modify the basic algorithm to enhance both compression and comprehensibility for certain genres of sequence. For example, we noted in Section 3 that, given the noise-free output of an L-system, the original recursive hierarchy can be inferred by unifying similar rules, combining phenomenal compression with exact reconstruction of the source. And we saw in Section 5 that given a large, real-world, noisy, sample of semi-structured text, inferences can be made automatically about the source that are both insightful and provide compression significantly superior to rival methods.

ACKNOWLEDGEMENTS

David Maulsby, from Stanford University, suggested representing structure as a hierarchical grammar. Dan Olsen from Brigham Young University collaborated with us on the genealogical database project. The phrase 'discontiguous dependencies' is inspired by Gerry Wolff from the University of Wales.

REFERENCES

[1] Wolff, J. G. (1980) Language acquisition and the discovery of phrase structure. *Language Speech*, **23**, 255–269.

[2] Nevill-Manning, C. G. (1996) *Inferring Sequential Structure*, Ph.D. Thesis, Department of Computer Science, University of Waikato, Hamilton, New Zealand.

[3] Knuth, D. E. (1973) *The Art of Computer Programming 3: Searching and Sorting*. Addison-Wesley, Reading, MA.

[4] Nevill-Manning, C. G. and Witten, I. H. (1997) Linear-time, incremental hierarchy inference for compression. *Proc. Data Compression Conf.*, Snowbird, UT, pp. 3–11.

[5] Mainous, F. D. and Ottman, R. W. (1966) *The 371 Chorales of Johann Sebastian Bach*. Holt, Rinehart and Winston, Inc., New York.

[6] Lindenmayer, A. (1968) Mathematical models for cellular interaction in development, Parts I and II. *J. Theor. Biol.*, **18**, 280–315.

[7] Bell, T. C., Cleary, J. G. and Witten, I. H. (1990) *Text Compression*. Prentice-Hall, Englewood Cliffs, NJ.

[8] Moffat, A., Neal, R. and Witten, I. H. (1995) Arithmetic coding revisited. *Proc. Data Compression Conf.*, Snowbird, UT, pp. 202–211.

[9] Cleary, J. G. and Witten, I. H. (1984) Data compression using adaptive coding and partial string matching. *IEEE Trans. Commun.*, **COM-32**, 396–402.

[10] Ziv, J. and Lempel, A. (1977) A universal algorithm for sequential data compression. *IEEE Trans. Inform. Theory*, **IT-23**, 337–343.

[11] Nevill-Manning, C. G., Witten, I. H. and Maulsby, D. L. (1994) Compression by induction of hierarchical grammars. *Proc. Data Compression Conf.*, Snowbird, UT, pp. 244–253.

[12] Storer, J. A. and Szymanski, T. G. (1982) Data compression via textual substitution. *J. Assoc. Comput. Mach.*, **29**, 928–951.

[13] Ziv, J. and Lempel, A. (1978) Compression of individual sequences via variable-rate coding. *IEEE Trans. Inform. Theory*, **IT-24**, 530–536.

[14] Nevill-Manning, C. G., Witten, I. H. and Olsen, D. R. (1996) Compressing semi-structured text using hierarchical phrase identification. *Proc. Data Compression Conf.*, Snowbird, UT, pp. 53–72.

[15] GEDCOM Standard, Draft release 5.4, Family History Department, The Church of Jesus Christ of Latter-day Saints, Salt Lake City, UT.

[16] Witten, I. H., Moffat, A. and Bell, T. C. (1994) *Managing Gigabytes: Compressing and Indexing Documents and Images*. Van Nostrand Reinhold, New York.

[17] Moffat, A. (1989) Word-based text compression. *J. Software—Practice Experience*, **19**, 185–198.

[18] Gottlieb, D., Hagerth, S. A., Lehot, P. G. H. and Rabinowitz, H. S. (1975) A classification of compression methods and their usefulness for a large data processing center. *Proc. National Computer Conf.*, pp. 453–458.