Working Paper Series
ISSN 1170-487X

Identifying Hierarchical Structure in Sequences: A Linear-Time Algorithm

by: Craig G Nevill-Manning and Ian H Witten

Working Paper 96/25
November 1996

Department of Computer Science
The University of Waikato
Private Bag 3105
Hamilton, New Zealand

IDENTIFYING HIERARCHICAL STRUCTURE IN SEQUENCES: A LINEAR-TIME ALGORITHM

Craig G. Nevill-Manning and Ian H. Witten
Department of Computer Science, University of Waikato,
Hamilton, New Zealand.
{cgn, ihw}@cs.waikato.ac.nz

Abstract

This paper describes an algorithm that infers a hierarchical structure from a sequence of discrete symbols by replacing phrases which appear more than once by a grammatical rule that generates the phrase, and continuing this process recursively . The result is a hierarchical representation of the original sequence. The algorithm works by maintaining two constraints: every digram in the grammar must be unique, and every rule must be used more than once. It breaks new ground by operating incrementally. Moreover, its simple structure permits a proof that it operates in space and time that is linear inthe size of the input. Our implementation can process 10,000 symbols/second and has been applied to an extensive range of sequences encountered in practice.

Key words

Sequence learning, grammatical inference, linear-time learning algorithms

1 Introduction

Many sequences of discrete symbols exhibit natural hierarchical structure. Text is made up of paragraphs, sentences, phrases, and words. Music is composed from major sections, motifs, bars, notes. Records of user interface behavior encode the hierarchical structure of tasks that users perform. Computer programs constitute modules, procedures, statements. Discovering the natural structure that underlies sequences is a challenging and interesting problem that has a wide range of applications, from phrase discovery to music analysis, from programming by demonstration to code optimization.

The search for structure in sequences occurs in many different fields. Adaptive text compression seeks models of sequences that can be used to predict upcoming symbols so that they can be encoded efficiently (Bell *et al.*, 1990). However, text compression models are extremely opaque, and do not illuminate any hierarchical structure in the sequence. Grammatical inference techniques induce hierarchical grammars from a set of example sentences, possibly along with a set of negative examples (Gold, 1967; Angluin, 1982; Berwick and Pilato, 1987). However, it is crucial to their operation that the input is not a continuous stream but is segmented into sentences, which are, in effect, independent examples of the structure being sought. Machine learning methods infer structure from examples, but again several examples are required which are distinguished from one another, and, in addition, the examples do not usually involve sequential information (Langley, 1996). Techniques of Markov modeling and hidden Markov modeling do not make any attempt to abstract information in hierarchical form (Rabiner and Juang, 1986). Sequence learning has played somewhat peripheral roles in areas such as automaton modeling (Gaines, 1976), adaptive systems (Andreae, 1977), programming by demonstration (Cypher, 1993), and human performance studies (Cohen *et al.*, 1990).

We have developed an algorithm that infers a hierarchical structure from a sequence of discrete symbols. The basic insight is that phrases which appear more than once can be replaced by a grammatical rule that generates the phrase, and that this process can be continued recursively. The result is a hierarchical representation of the original sequence. It is not a grammar, for the rules are not generalized and are capable of generating only one string. (It does provide a good basis for going on to infer a grammar, but that is beyond the scope of this paper.) A scheme that resembles the one developed here arose from work in language acquisition (Wolff, 1975, 1977, 1978, 1982). However, it operated in time which is quadratic with respect to the length of the input sequence, whereas the algorithm we

describe takes linear time. This has allowed us to investigate sequences containing several million tokens—in previous work the examples were much smaller, the largest mentioned being several thousand tokens. Another difference, which is of crucial importance in some practical applications, is that the new algorithm works incrementally.

The ability to deal easily with long sequences has greatly extended the range of application of the technique. We have applied it to artificially-generated fractal-like sequences produced by L-systems, and, along with a unification-based rule generalizer, been able to recover the original L-system. Using the same method we have been able to infer relatively compact deterministic, context-free grammars for million-symbol sequences representing biological objects obtained from stochastic, context-sensitive, L-systems; and this has been used to greatly speed up the graphical rendering of such objects. We have applied it to 40 Mbyte segments of a digital library to generate hierarchical phrase indexes for the text, which provides a novel method of browsing. It has been used to compress 3 Mbyte sequences of macromolecular DNA more effectively than other general-purpose compression algorithms. It has been used, with some postprocessing, to elicit structure from a 2 million word extract from a genealogical database, successfully identifying the structure of the database and compressing it much more efficiently than the best known algorithms. All these applications are described by Nevill-Manning (1996); they will not be covered further here. None would have been possible with the earlier quadratic-time algorithm.

This paper describes the algorithm, called SEQUITUR. It works incrementally, modifying the grammar as each new sequence of the symbol is received. Two properties are maintained in the grammar: digram uniqueness and rule utility; these drive the grammar modification process. The next section gives a concise description of the algorithm in terms of these two properties. Section 3 describes the implementation in more detail, with particular emphasis on how efficiency is achieved. Section 4 shows that the run time and storage requirements are linear in the number of input symbols, while Section 5 discusses the algorithm's behavior on extreme input strings. The final section takes a brief look at SEQUITUR's operation on strings of natural-language text to give a feeling for the kind of performance encountered in practice.

2 The algorithm

SEQUITUR forms a grammar from a sequence based on repeated phrases in it. 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 this pursuit of brevity that drives the algorithm to form and maintain the grammar, and as a by-product, provide a hierarchical structure for the sequence.

At 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 it, a new rule $A \rightarrow bc$ is formed, and both occurrences of *bc* are replaced by *A*. The new grammar is shown at 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 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 Figures 1a and 1b share two properties:

p_1: no pair of adjacent symbols appears more than once in the grammar;

p_2: every rule is used more than once.

p_1 can be restated as 'every digram in the grammar is unique,' and will be referred to as

	Sequence	Grammar		Sequence	Grammar
a	S → abcdbc	S → aAdA A → bc	b	S → abcdbcabcdbc	S → AA A → aBdB B → bc
c	S → abcdbcabcdbc	S → AA A → abcdbc S → CC A → bc B → aA C → BdA	d	S → aabaaab	S → AaA A → aab S → AbAab A → aa

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
(d) two different grammars for the same sequence that obey the constraints.

digram uniqueness. p_2 ensures that each rule is useful, and will be called *rule utility*. These two constraints exactly characterize the grammars that SEQUITUR generates.

Figure 1c shows what happens when these properties are violated. The first grammar contains two occurrences of *bc*, so p_1 does not hold. This introduces redundancy because *bc* appears twice. In the second grammar, *B* is used only once, so p_2 does not hold. If it were removed, the grammar would become more concise.

The grammars in Figures 1a and 1b are the only ones for which both properties hold for each sequence. However, there is not always a unique grammar with these properties. For example, the sequence in Figure 1d can be represented by both of the grammars on its right, and they both obey p_1 and p_2. We deem either grammar to be acceptable.

SEQUITUR's operation consists of ensuring that both properties hold. When describing the algorithm, the properties act 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 how this is performed.

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 it occurs elsewhere in the grammar, the first constraint has been violated. To restore it, 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.

Figure 2 shows the grammar as new symbols are added in the sequence *abcdbcabcd*. The first 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 second column shows the sequence observed so far. The third gives the grammar created from the sequence. The fourth lists any duplicate digrams, while the fifth 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 appears as the right-hand side of an existing rule, then no new rule need be created: the digram is replaced by the non-terminal that heads the existing rule. Figure 2b demonstrates the changes that occur in the grammar when a third *bc* appears. The existing non-terminal *A* is substituted for the third occurrence of *bc*. This results in a new pair of

	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

repeating digrams, aA, shown in the last line of Figure 2b. In Figure 2c, a new rule B is formed accordingly, and the two occurrences of aA are replaced by B.

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.

Rule utility

Up until now, the right-hand sides of rules in the grammar have been only two symbols long. 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 this. When d is appended to rule S, the new digram Bd causes a new rule, C, to be formed. However, forming 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 forming long rules: form a short rule temporarily, and if subsequent symbols continue the match, allow a new rule to supersede the shorter one, and delete the latter.

3 Implementation issues

The SEQUITUR algorithm operates by enforcing the digram uniqueness and rule utility constraints. It is essential that any violation of these constraints be efficiently detected. We now describe mechanisms that accomplish this.

Storing and manipulating the grammar

The choice of an appropriate data structure depends on the kind of operations that need to be performed to modify the grammar. These 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 two symbols, thereby shortening the rules containing the digrams. Creating a new rule involves creating a new non-terminal for the left-hand side,

and inserting two new symbols as the right-hand side. After creating the rule, substitutions are made as for an existing rule, by replacing the two digrams with the new non-terminal. Deleting a rule involves moving the contents of a rule to replace a non-terminal, which lengthens the rule containing the non-terminal. The left-hand side of the rule must then be deleted.

Figure 3 illustrates each of these functions. To ensure that the lengthening and shortening of rules is performed efficiently, a doubly-linked list structure is used whose start and end are connected to a single guard node. The guard node also serves as an attachment point for the left-hand side of the rule, because it remains constant even when the rule contents change. Figure 3c illustrates this arrangement. While the guard node is shown to the left of the rule contents, it could equally be shown at the right. Each non-terminal points to the rule it heads, which is not shown in the figure. With these pointers, no arrays are necessary for accessing rules or symbols, because operations only affect adjacent symbols, or rules headed by a non-terminal symbol. The rules, however, are linked together

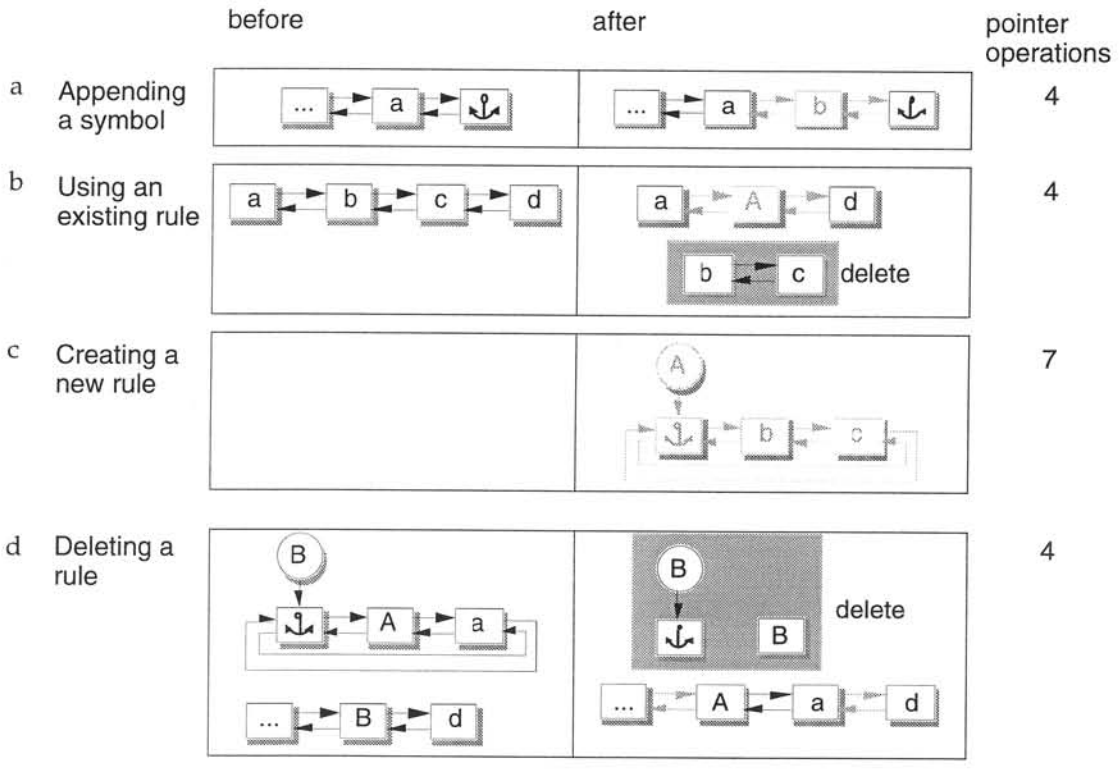

Figure 3 The four operations required to build the grammar:
(a) appending a symbol,
(b) using an existing rule,
(c) creating a new rule, and
(d) deleting a rule

so that they can be traversed for printing.

Creating a new symbol involves allocating space in memory and initializing it. Changing a link in the diagram involves one pointer operation. Appending a symbol to a rule involves creating one symbol and making four pointer assignments, as shown in Figure 3a, where the symbol b is appended to a rule ending in a. Using an existing rule involves deleting two symbols (the digram to be replaced), creating one symbol for the new non-terminal, and making four pointer assignments; Figure 3b shows the substitution of A in place of bc. Creating a new rule involves creating a new left-hand side, two symbols and a guard node, and making seven pointer assignments as shown in Figure 3c. Deleting a rule requires deleting the left-hand side of the rule, deleting the guard node and terminal symbol, and making four pointer assignments to reposition the contents of the rule, as shown in Figure 3d. As for garbage collection, because the grammar always grows as more of the sequence appears, no symbol ever need be deallocated. Instead, symbols are kept in a 'spare' rule, where they are inserted or extracted as necessary.

Maintaining rule utility

The rule utility constraint demands that a rule is deleted if it is referred to only once. One way to enforce this constraint is to scan the entire grammar at each step, counting non-terminal frequencies. However, it is more efficient to record rule frequencies and update them at the same time as symbols are created and deleted.

Every time a non-terminal symbol is created (either allocated on the heap or reused from the list of spare symbols), the frequency for that rule is incremented, and every time a non-terminal is deleted, the frequency is decremented. When the frequency of a rule falls to one, the rule is automatically deleted. The frequency of the rule is recorded within the rule data structure, to which the non-terminal symbol has a pointer, obviating the need for an array to index the appropriate rule.

Maintaining digram uniqueness

The second constraint is more difficult to enforce. When a new digram appears, the grammar must be searched for any other occurrence of it. One simple solution would be to scan the entire grammar each time looking for a match, but this is inefficient and leads to a quadratic-time algorithm. A better solution requires an index that is efficient to search.

The data structure for storing the digram index must permit fast access and efficient addition and deletion of entries. A hash table provides constant-time access, and adding and deleting entries requires little extra work. Because no digram appears more than once, the table need only contain a pointer to the first symbol in the single matching digram. The hash table consists of a simple array of pointers. Collisions are handled by *open addressing* to avoid the allocation of memory that chaining requires (Knuth, 1968).

Every time a new digram appears in the grammar, it is added to the index. A new digram appears as a result of two pointer assignments linking two symbols together in the doubly-linked list (one forward pointer and one back pointer). Thus updating the index can be incorporated into the low-level pointer assignments. A digram also disappears from the grammar when a pointer assignment is made—the pointer value that is overwritten by the assignment represents a digram that no longer exists.

To demonstrate the mechanism for updating the hash table when a new rule is created, Figure 4 shows the example at the end of Figure 1a, with the addition of the contents of the digram index. When the second c is appended to rule S, the digram table shows that bc already exists in the grammar, so the rule $A \rightarrow bc$ is created. Creating the link between b and c in the right-hand side for rule A updates the entry in the index for bc to point to its new location—the hash table now contains a pointer to the symbol b at the start of rule A. Next, the first bc is removed. This breaks the link between the b in the digram and the preceding symbol a, so ab is removed from the index. It also breaks the link between c and the following d, so cd is removed from the index. Next, A is inserted in place of bc, creating links between a and A, as well as between A and d, adding these digrams to the index. This process continues, resulting in a correct index of digram pointers, but costing just one indexing operation per two pointer operations.

Action	grammar	change in digrams	digram index
observe symbol 'c'	S → abcdbc		{ab, bc, cd, db}
make new rule A	S → abcdbc A → bc	bc updated	{ab, bc, cd, db}
substitute A for bc	S → aAdbc A → bc	ab, cd removed, aA, Ad added	{bc, db, aA, Ad}
substitute A for bc	S → aAdA A → bc	db removed, dA added	{bc, dA, aA, Ad}

Figure 4 Updating the digram index as links are made and broken

Having an efficiently maintained index of digrams is of little use if it is checked unnecessarily often. Rechecking the entire grammar whenever a symbol is added is infeasible, and inefficient if large portions of the grammar are unchanged since the last check. In fact, the only parts that need checking are those where links have been made or broken. That is, when any of the actions that affect the maintenance of the digram table are performed, the newly created digrams should be checked in the index. Of course, every time a link is created, the digram is *entered* into the index, and this is the very time to check for a duplicate. So if while attempting to add a new digram to the index an entry is found to be already present, a duplicate digram has been detected and the appropriate actions should be performed. This is the only time when it is necessary to consult the digram index. For example, when a new symbol is appended to rule S, the digram thus created must be entered in the table. If an entry already exists for this digram, the uniqueness constraint has been violated, and the appropriate response is triggered.

Transient rules

In Figure 2d, the rule $B \rightarrow aA$ is formed, but when the next symbol, d, appears, rule B is subsumed into a longer rule, $C \rightarrow aAd$. It seems inefficient to create a new rule and then to destroy it when the next symbol is observed. One modification to SEQUITUR could postpone the creation of a rule until it is clear that it is necessary. Another possibility is to extend rules like B, rather than creating a new rule and deleting the existing one. We, will demonstrate why these two options offer no improvement over the algorithm described so far.

First, let us examine the postponement of rule creation. The idea is to recognize repetition in the grammar, but to put off creating a rule until it is clear that the repetition cannot grow any longer. This occurs when a symbol is observed that does not continue the match. At this point, the rule can be created knowing that it will not be superseded by a longer rule. This means that every rule in the grammar is permanent—there are no transient rules.

However, tracking partial matches is difficult. Figure 5 shows a situation where two matches are in progress. There is a repeated subsequence aAe, which awaits another A to complete rule B. There is also a repeated bc, which expects a d to complete rule A. If the expected d arrives, A is exactly matched, which allows the match with B to complete. This is illustrated on the left-hand side of Figure 5. If some other symbol arrives, however, the match with A fails, and a rule C must be created to account for the successful match so far. Because the match with A failed, the match with B cannot succeed, so another rule E is

created to account for the matched portion. In a large grammar the tracking and recovery from partial matches becomes even more complicated.

Transient rules can be seen as performing this tracking within the grammar. No partial match recovery is necessary because the rules represent the worst case, while allowing longer matches to succeed by disappearing if necessary. Furthermore, at any point the grammar obeys the two constraints, so if the sequence were to end, no further transformations would be necessary. This is useful when using the grammar to predict or explain the sequence: all that is known about the sequence is contained within the grammar, rather than in the partial match tracking data structures. This is the essence of the incrementality of the algorithm. The fact that the state information is completely contained in the grammar rather than in other data structures, and that the grammar is efficiently indexed, means that the algorithm is simple, and that it stores and indexes partial matches efficiently.

The second solution to transient rules is to extend rules instead of replacing them. This introduces a new rule to the algorithm: 'if a digram appears more than once, and its first symbol is a non-terminal, and that symbol appears only twice in the grammar, then append the second symbol in the digram to the rule headed by the first symbol in the digram.' For example, when the digram *Bd* appears, because *B* only appears twice, *d* is appended to rule *B*.

The reason for the stipulation that the rule must appear only twice is this: if the rule is

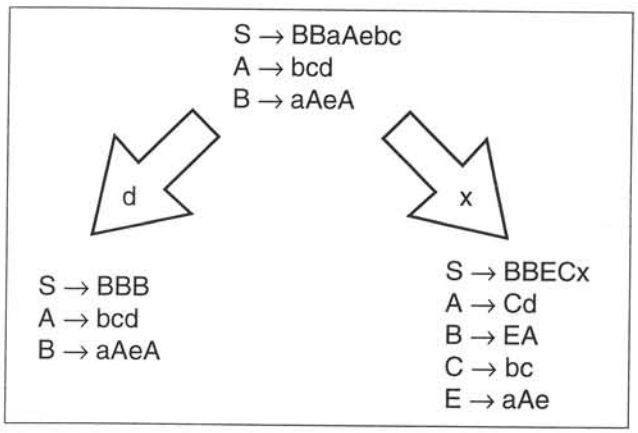

Figure 5 Two outcomes for the rule creation postponement approach, after seeing the string *abcdebcdabcdebc*
(a) the grammar with match *bc* awaiting a longer match
(b) the grammar if the awaited *d* occurs
(c) the grammar if *d* does not occur next

extended, it affects all instances of that rule. For the transformation to preserve the sequence, all instances of the rule must be followed by the same symbol. Because all digrams are unique apart from the duplicate in question, any other occurrence of the non-terminal apart from the two in the duplicate digrams must be followed by some other symbol. So if the non-terminal occurs more than twice, the rule it heads cannot be extended. The introduction of this new case in the algorithm does not result in the removal of the digram utility rule. Allowing rule extension may increase efficiency by saving one rule creation, but it complicates the algorithm rules unnecessarily—it is superfluous, because its effect is achieved by the combination of the other rules.

4 Computational complexity

This section shows that the algorithm is linear in space and time. The complexity proof is an amortized one—it does not put a bound on the time required to process one symbol, but bounds the time taken for the whole sequence. The processing time for one symbol can in fact be as large as $O(\sqrt{n})$ where n is the number of input symbols so far. However, the pathological sequence that produces this worst case requires that the preceding $O(\sqrt{n})$ symbols involve no formation or matching of rules.

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 satisfying the constraints 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.

Figure 6 gives a full description of the SEQUITUR algorithm. Line 1 deals with new observations in the sequence. Lines 2 through 9 enforce the digram utility constraint. Line 3 checks whether a new link should be inserted by consulting the digram table, while 8 and 9 record a new diagram. Line 4 determines whether the new digram matches an existing rule, or whether a new one is necessary. Line 10 through 12 enforce rule utility. Lines 3 and 11 are triggered whenever the constraints are violated.

The numbers at the right identify the main sections of the algorithm, and the proof will demonstrate bounds on the number of times that each of them are executed. Action 1 appends symbols to rules S, and is performed exactly n times, once for every symbol in the input. Action 2 is performed when a link is created. Action 3 corresponds to using an existing rule, action 4 to forming a new rule, and action 5 to removing a rule.

Figure 7 shows examples of actions 3, 4 and 5, and the savings in grammar size associated with each one. The savings are calculated by counting the number of symbols in the grammar before and after the action. The non-terminals that head rules are not counted, because they can be recreated based on the order in which the rules occur. Actions 3 and 5 are the only actions performed on the grammar that reduce the number of symbols. There are no actions that increase the size of the grammar, so the difference between the size of the input and the size of the grammar must equal the number of times that both these actions have been taken.

More formally, let

n be the size of the input string,
o be the size of the final grammar,
r be the number of rules in the final grammar,
a_1 be the number of times new symbol is seen (action 1),
a_2 be the number of times a new digram is seen (action 2),
a_3 be the number of times an existing rule is used (action 3),
a_4 be the number of times a new rule is formed (action 4), and

```
                                                                          action
1   As each new input symbol is observed, append it to rule S.              1

2   Each time a link is made between two symbols                             2
3       if the new digram is repeated elsewhere,
4           if the other occurrence is a complete rule,
5               replace the new digram with the non-terminal that heads the rule,  3
6           otherwise,
7               form a new rule and replace both digrams with the new non-terminal  4
8       otherwise,
9           insert the digram into the index

10  Each time a digram is replaced by a non-terminal
11      if either symbol is a non-terminal that only occurs once elsewhere,
12          remove the rule, substituting its contents in place of the other non-terminal  5
```

Figure 6 The SEQUITUR algorithm

	action	before	after	saving
Matching existing rule	3	...ab... A → ab	...A... A → ab	1
Creating new rule	4	...ab...ab...	...A...A... A → ab	0
Deleting a rule	5	...A... A → ab	...ab...	1

Figure 7 Reduction in grammar size for the three grammar operations

a_5 be the number of times a rule is removed (action 5).

According to the reasoning above, the reduction in the size of the grammar is the number of times actions 3 and 5 are executed. That is,

$$n - o = a_3 + a_5 \tag{1}$$

Next, the number of times a new rule is created (action 4) must be bounded. The two actions that affect the number of rules are 4, which creates rules, and 5, which deletes them. The number of rules in the final grammar must be the difference between the frequencies of these actions:

$$r = a_4 - a_5$$

In this equation, r is known, and a_5 is bounded by equation (1), but a_4 is unknown. Noting that a_1, the number of times a new symbol is seen, is equal to n, the total work is

$$a_1 + a_2 + a_3 + a_4 + a_5 = n + a_2 + (n - o) + (r + a_5)$$

To bound this expression, note that the number of rules must be less than the number of symbols in the final grammar, because each rule contains at least two symbols, so

$$r < o$$

Also, from (1):

$$a_5 = n - o - a_3 < n$$

Consequently,

$$a_1 + a_2 + a_3 + a_4 + a_5 = 2n + (r - o) + a_5 + a_2 < 3n + a_2$$

The final operation to bound is action 2, which checks for duplicate digrams. Searching the grammar is performed by hash table lookup. Assuming an occupancy less than, say, 80% gives an average lookup time bounded by a constant (Knuth, 1967). This occupancy can be assured if the size of the sequence is known in advance, or by enlarging the table and recreating the entries whenever occupancy exceeds 80%. The number of entries in the table is just the number of digrams in the grammar, which is the number of symbols in the grammar minus the number of rules in the grammar, because symbols at the end of a rule do not form the left hand side of any digram. So the size of the hash table is less than the size of the grammar, which is bounded by the size of the input. This means that the memory requirements of the algorithm are linear.

As for the number of times that action 2 is performed, a digram is only checked when a new link is created. Links are only created by actions 1, 3, 4 and 5, which have already been shown to bounded by $3n$, so the time required for action 2 is also $O(n)$.

Thus we have shown that the algorithm is linear in space and time. However, this claim must be qualified: it is based on a register model of computation rather than a bitwise one. We have assumed that the average lookup time for the hash table of digrams is bounded by a constant. However, as the length of the input increases, the number of rules increases without bound, and for unstructured (e.g., random) input, the digram table will grow without bound. Thus the time required to execute the hash function and perform addressing will not be constant, but will increase logarithmically with the input size. Our proof ignores this effect: it assumes that hash function operations are register-based and therefore constant time. 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.

5 Exploring the extremes

Having described SEQUITUR algorithmically, we now characterize its performance on a variety of data. This section explores how large or small a grammar can be for a given sequence length, as well as determining the minimum and maximum amount of work the algorithm can perform, and the amount of work required to process one symbol. Figure 8 summarizes these extreme cases, giving part of an example sequence and the grammar that results. Bounds are given in terms of n, the number of symbols in the input.

The deepest hierarchy that can be formed has depth $O(\sqrt{n})$, and an example of a sequence that forms such a hierarchy is shown in Figure 8a. In the deepest hierarchy, each rule (except the one at the lowest level) must contain a non-terminal, so that the hierarchy deepens at each rule. Furthermore, it is unnecessary for any rule to be longer than two symbols. Therefore, to produce a deep hierarchy from a short string, each rule should be one terminal symbol longer than the one on which it builds. In order to create these rules, the string represented must appear in two different contexts, otherwise the rule will be incorporated into a longer rule. One context is the tallest hierarchy, which it must participate in. The other context should not be in any hierarchy, to reduce the size of the input string, so it should appear in rule S. Note that every rule in Figure 8a appears both in the hierarchy and in rule S. At each repetition of the sequence, one terminal symbol is appended, producing a new level in the hierarchy. There is no point in including a repetition of length

one, so the m^{th} repetition has length $m+1$. This repetition gives rise to the m^{th} rule (counting rule S). The total length of the sequence for a hierarchy of depth m is therefore

$$n = 2 + 3 + 4 + \ldots + (m+1) = O(m^2)$$

and the deepest hierarchy has depth $m = O(\sqrt{n})$.

At the other end of the spectrum, the grammar with the shallowest hierarchy is shown in Figure 8b. It has no rules apart from rule S. It is also the largest grammar for a sequence of a given length, precisely because no rules can be formed from it. The sequence that gives rise to it is one in which no digram ever recurs. Of course, in a sequence with an alphabet of size $|\Sigma|$, there are only $O(|\Sigma|^2)$ different digrams, which bounds the length of such a sequence. This kind of sequence produces the worst case compression: there are no repetitions, and therefore no structure is detected by SEQUITUR.

Turning from the largest grammar to the smallest, Figure 8c depicts the grammar formed from the most ordered sequence possible—one consisting entirely of the same

		bound	example sequence	example grammar
a	deepest hierarchy	$O(\sqrt{n})$	ababcabcdabcdeabcdef	S → ABCDDf A → ab B → Ac C → Bd D → Ce
b	largest grammar; shallowest hierarchy	n	aabacadae...bbcbdbe...	S → aabacadae...
c	smallest grammar	O(log *n*)	aaaaaaaaaaaaaa...	S → DD A → aa B → AA C → BB D → CC
d	largest number of rules	*n*/4	aaaaabababacacadad...	S → AABBCCDD A → aa B → ab C → ac D → ad
e	maximum processing for one symbol	$O(\sqrt{n})$	yzxyzwxyzvwxy	S → ABwBvwxy A → yz B → xA
f	greatest number of rule creations and deletions	*n* new rules *n* deleted rules	abcde abcde abcde...	S → AAA... A → abcde•

Figure 8 Some extreme cases for the algorithm

symbol. When four contiguous symbols appear, such as *aaaa*, a rule $B \rightarrow aa$ is formed. When another four appear, rule S contains $BBBB$, forming a new rule $C \rightarrow BB$. Every time the number of symbols doubles, a new rule is created. The hierarchy is thus O(log n) deep, and the grammar is O(log n) in size. This represents the greatest data compression. It is not necessary to have a sequence of only one symbol to achieve this logarithmic lower bound—any recursive structure will do.

To produce the grammar with the largest number of rules, each rule should only include terminal symbols, because building a hierarchy will reduce the number of rules required to cover a sequence of a given size. Furthermore, no rule should be longer than two symbols or occur more than twice. Therefore each rule requires 4 symbols for its creation, so the maximum number of rules for a sequence of length n is $n/4$, as shown in Figure 8d.

Having discussed the size of grammars, we now move to the effort involved in maintaining them. The upper bound for processing a sequence has been discussed, and shown to be linear. However, it is still useful to characterize the amount of processing involved for each new symbol. Figure 8e shows a sequence where the repetition is built up as yz, then xyz, then $wxyz$, etc. Just before the second occurrence of $wxyz$ is completed, no matches have been possible for the w, x, and y. When z appears, yz matches rule A, then xA matches rule B. Finally, wB forms a new rule. This cascading effect can be arbitrarily large if the repetitions continue to be built up in this right-to-left fashion. The amount of processing required to deal with the last z is proportional to the depth of the deepest hierarchy, as the matching cascades up the hierarchy. The maximum time to process one symbol is therefore $O(\sqrt{n})$. The fact that w, x, and y failed to match means that they required little time to process, ensuring that the linear time bound overall is preserved.

While the linear bound has been given in order notation, sequences certainly differ in the proportion of work to sequence length. This ratio is minimized by the sequence in Figure 8b, where no repetitions exist and no grammar is formed. It is maximized by the sequence in Figure 8f, which consists of multiple repetitions of a multi-symbol sequence. Each time the repetition appears, there are several rule deletions and creations as the match lengthens. In fact, every symbol except a incurs a rule creation, and a subsequent deletion, so there are O(n) creations and deletions. If m is the length of the repetition, the proportion of symbols that do not incur this work is $1/m$, which tends toward zero as the repetition length approaches infinity.

6 Performance in practice

To give an idea of how SEQUITUR performs on realistic sequences, we turn from artificial cases to a sequence of English text. Figure 9a plots the number of rules in the grammar against the number of input symbols, for a 760,000 character English novel, and shows that the increase is approximately linear. Figure 9b shows the approximately linear growth of the total number of symbols in the grammar. The growth of the number of unique words in the text, shown in Figure 9c, is high at the start and drops off toward the end. It has been observed in much larger samples (Zobel, *et al.*, 1995) that—surprisingly—new words continue to appear at a fairly constant rate, corresponding not just to neologisms but to names, acronyms, and typographical errors as well. In this example, the number of rules grows linearly because once words have been recognized, multi-word phrases are

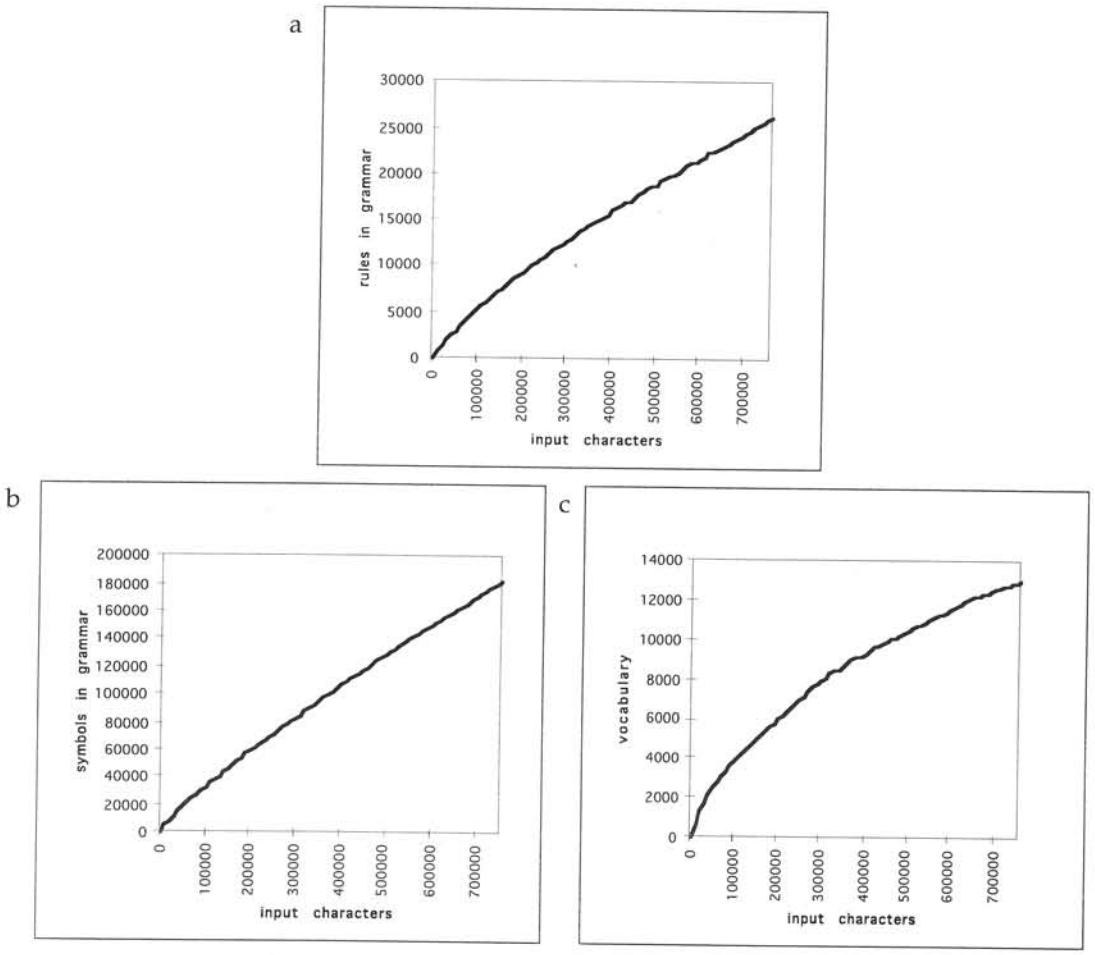

Figure 9 Growth rates on English text
(a) rules in the grammar
(b) symbols in the grammar
(c) vocabulary size in the input

constructed, and the number of such phrases is unbounded.

SEQUITUR operates very quickly. Experiments have been conducted on a variety of sequences, from text in various natural languages, semi-structured databases, L-system output, and DNA sequences. Symbols are either words or characters. Sequence lengths of up to 6.5 million words have been evaluated, corresponding to files of up to 42 Mbyte of information. On long sequences, SEQUITUR operates at a rate of around 10,000 symbol/sec (on a fast SPARCstation).

We have intentionally avoided becoming drawn into a discussion of the structural hierarchies that SEQUITUR produces, because they need careful interpretation in the context of actual applications. However, to give a taste for the phrases that are generated, Figure 10 shows parts of three hierarchies inferred from the text of the Bible in English, French, and German. The hierarchies are formed without any knowledge of the preferred structure of words and phrases, but nevertheless capture many meaningful regularities. In Figure 10a, the word *beginning* is split into *begin* and *ning*—a root word and a suffix. Many words and word groups appear as distinct parts in the hierarchy (spaces have been made explicit by replacing them with bullets). The same algorithm produces the French version in Figure 10b, where *commencement* is split in an analogous way to *beginning*—into the root *commence* and the suffix *ment*. Again, words such as *Au*, *Dieu* and *cieux* are distinct units in the hierarchy. The German version in Figure 10c correctly identifies all words, as well as the phrase *die Himmel und die*. In fact, the hierarchy for *the heaven and the* in Figure 10a bears some similarity to the German equivalent. A more comprehensive evaluation of results in a wide range of domains is given by Nevill-Manning (1996).

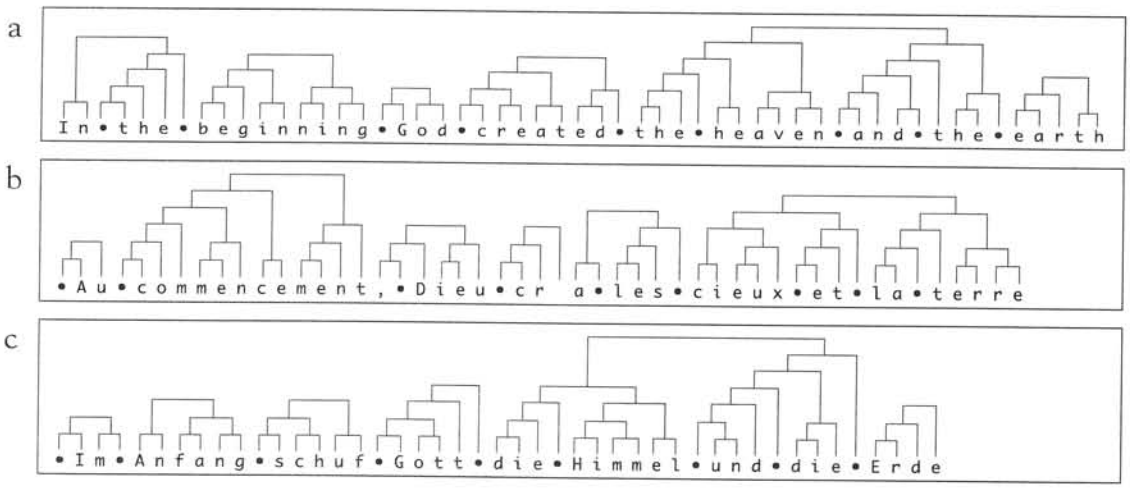

Figure 10 Hierarchies for Genesis 1:1 in (a) English, (b) French, and (c) German

7 Conclusion

An algorithm has been presented for the identification of hierarchical structure in sequences, based on the idea of abstracting subsequences that occur more than once into rules, and continuing this operation recursively. The algorithm works by maintaining two constraints: every digram in the grammar must be unique, and every rule must be used more than once. It operates incrementally and, subject to a caveat about the register model of computation used, in linear space and time. This efficiency has permitted its application to extremely long sequences in many different domains.

References

Andreae, J.H. (1977) *Thinking with the teachable machine.* London: Academic Press.

Angluin, D. (1982) "Inference of reversible languages," *Journal of the Association for Computing Machinery, 29*, 741–765.

Bell, T.C., Cleary, J.G., and Witten, I.H. (1990) *Text compression.* Englewood Cliffs, NJ: Prentice-Hall.

Berwick, R.C., and Pilato, S. (1987) "Learning syntax by automata induction," *Machine Learning, 2*, 9–38.

Cohen, A., Ivry, R.I., and Keele, S.W. (1990) "Attention and structure in sequence learning," *Journal of Experimental Psychology, 16*(1), 17–30.

Cypher, A., editor (1993) *Watch what I do: programming by demonstration,* Cambridge, Massachusetts: MIT Press.

Gaines, B.R. (1976) "Behaviour/structure transformations under uncertainty," *International Journal of Man-Machine Studies, 8*, 337–365.

Gold, M. (1967) "Language identification in the limit," *Information and Control, 10*, 447–474.

Knuth, D.E. (1968) *The art of computer programming 1: fundamental algorithms.* Addison-Wesley.

Langley, P. (1996) *Elements of Machine Learning.* San Francisco: Morgan Kaufmann.

Nevill-Manning, C.G. (1996) "Inferring sequential structure," D.Phil. thesis, Computer Science Department, University of Waikato, New Zealand.

Rabiner, L.R. and Juang, B.H. (1986) "An introduction to hidden Markov models," *IEEE ASSP Magazine, 3*(1), 4–16.

Wolff, J.G. (1975) "An algorithm for the segmentation of an artificial language analogue," *British Journal of Psychology, 66*(1), 79–90.

Wolff, J.G. (1977) "The discovery of segments in natural language," *British Journal of Psychology, 68*, 97–106.

Wolff, J.G. (1980) "Language acquisition and the discovery of phrase structure," *Language and Speech, 23*(3), 255–269.

Wolff, J.G. (1982) "Language acquisition, data compression and generalization," *Language and Communication, 2*(1), 57–89.