# Online and offline heuristics for inferring hierarchies of repetitions in sequences

**Craig G. Nevill-Manning**[*] **and Ian H. Witten**[**]

[*]*Department of Computer Science, Rutgers University, USA; nevill@cs.rutgers.edu*

[**]*Department of Computer Science, Waikato University, Hamilton, New Zealand; ihw@waikato.ac.nz*

**Abstract.** *Hierarchical dictionary-based compression schemes form a grammar for a text by replacing each repeated string with a production rule. While such schemes usually operate online, making a replacement as soon as repetition is detected, offline operation permits greater freedom in choosing the order of replacement. In this paper, we compare the online method with three offline heuristics for selecting the next substring to replace: longest string first, most common string first, and the string that minimizes the size of the grammar locally. Surprisingly, two of the offline techniques, like the online method, run in time linear in the size of the input. We evaluate each technique on artificial and natural sequences. In general, the locally-most-compressive heuristic performs best, followed by most frequent, the online technique, and, lagging by some distance, the longest-first technique.*

**Key Words.** dictionary compression, hierarchical compression, grammar inference

## 1 Introduction

Compression can be gained by taking advantage of any structure that can be inferred from an input sequence. Most text compression techniques can be placed in one of two classes: *symbolwise* methods where the structure that is inferred is a set of conditional probability distributions, and *dictionary* methods where the structure is a set of substrings and compression is achieved by replacing fragments of text by an index into a dictionary. This paper adopts the latter approach, but extends it to dictionaries that have non-trivial hierarchical structure. The result is a decomposition of the input string into what is, in effect, a parse tree. The process of coding such a hierarchy is relatively routine (e.g. Cameron, 1988)—although it can become tricky when done adaptively. Rather than reviewing coding schemes for hierarchies, this paper studies algorithms that infer hierarchical structure from an input string.

Dictionary-based compression methods capitalize on repetitions. In simplest form, this involves replacing subsequent occurrences of a substring with references to the first instance. There are several constraints that may affect this process, the most important for our purposes being online vs offline, frequency vs length, and hierarchical vs flat representations.

"Online" algorithms process the input stream in a single pass, and begin to emit compressed output long before they have seen all the input. Historically, virtually all compression algorithms have been online, because main memory has until recently been the principal limiting factor on large-scale application of string processing algorithms for compression. However, hardware developments have begun to relax this constraint. Offline algorithms can examine the input in a more considered fashion, and this raises the question of whether to seek *frequent* repetitions or *long* repetitions—or some intermediate combination of frequency and length. Finally, repetitive structure that is sought in the input may be flat or hierarchical; in the latter case it may be a general hierarchy or restricted to a more trivial tail-recursive one.

This paper brings together recently-developed regimes for inferring hierarchies of repetitions in sequences and compares them on a level footing. We are concerned mainly with algorithms that operate in time that is linear in the length of the input sequence. This is a severe restriction: apart from standard compression algorithms that produce non-hierarchical structure (e.g. Ziv and Lempel, 1977) and tail-recursive hierarchical structure (e.g. Ziv and Lempel, 1978), no linear-time algorithms were known for detecting hierarchical repetition in sequences until recently.

But there are now three linear-time algorithms: an online method that infers a hierarchy of repetitions (Nevill-Manning and Witten, 1997); one that gives preference to frequency by introducing a rewrite rule for the most frequent pair of adjacent symbols first, continuing with the next most frequent and so on (Larsson and Moffat, 1999); and one that gives preference to length by replacing the longest repetition first (Bentley and McIlroy, 1999). Although a straightforward multi-pass implementation of the third algorithm does not operate in linear time, a suffix tree version has recently been developed that does (Nevill-Manning and Farach-Colton, in preparation). A compromise between the extremes of preferring frequency to length and vice versa is to prefer rules that remove the greatest number of symbols from the string, maximizing the product of frequency and length of the substring—though whether linear-time algorithms exist for this is still open.

This paper surveys these methods for hierarchy generation and compares their effectiveness in generating small hierarchies on artificial and natural strings. Section 2 briefly reviews non-hierarchical dictionary compression. Section 3 discusses the hierarchical representation and considers how to define the size of grammars. Section 4 describes SEQUITUR, an on-line technique, and reviews some salient properties: linear-time operation and convergence to the entropy of the source. Then it introduces the three off-line heuristics that will be studied. Section 5 discusses a simple, uniform, implementation of the techniques that was used for the experiments. Section 6 evaluates the four methods, through small illustrative artificial examples, on samples of English text, and on large artificially-generated strings. Detecting hierarchical structures of repetitions is useful in contexts other than data compression, and Section 7 describes three rather different applications of hierarchical inference: hot subpath identification for program optimization, text-to-speech conversion, and phrase hierarchies for browsing large text collections.

## 2 Standard dictionary-based techniques

Two classes of data compression schemes that encompass most popular practical compression programs both derive from seminal work by Ziv and Lempel in the mid-70s: we call these LZ77 and LZ78 schemes. The basic idea of the former class is to replace substrings of the input string with pointers to earlier occurrences of the same substring (Ziv and Lempel, 1977). The basic idea of the latter class is to build a dictionary of phrases that occur in the input string, and replace all phrases by references to dictionary entries, forming each new phrase by adding a single terminal symbol to an existing dictionary entry (Ziv and Lempel, 1978). (There are variants that build long dictionary entries faster by concatenating two dictionary entries to form a new one; see Miller and Wegman, 1985.)

Whereas the structure produced by LZ77 is flat, the dictionary generated by LZ78 is hierarchical: each entry (except for the first few) points to another entry that is one character shorter. The hierarchy of repetitions is a simple one: embedding can only occur on the left, producing a structure that is in effect tail-recursive. Although LZ77 produces a flat encoding, its pointers can denote substrings in the original sequence that overlap; LZ78, on the other hand, parses the string into non-overlapping segments. In this paper we concentrate on non-overlapping hierarchical representations. A hierarchical decomposition of the input string is useful for many applications other than data compression, including keyphrase extraction, full-phrase browsing, computing document similarity, plagiarism detection, and sequence understanding.

Unlike LZ78, the hierarchies we generate are general ones. A key difference from the schemes we present is that the LZ78 dictionary is formed speculatively: most dictionary entries are never used. This results in unused code space. In contrast, we describe techniques that use every dictionary element. This incurs a coding overhead of specifying which substrings belong in the dictionary, which is unnecessary for LZ78 since it places every qualifying substring into the dictionary.

## 3 Representing and evaluating hierarchical dictionaries

In order to represent a sequence in which repetitions have been identified, we use a rewriting system that resembles a context-free grammar but lacks both recursion and alternative expansions of a given non-terminal. That is, a non-terminal heads at most one rule (no alternative expansions), and does not appear within the body of a rule it heads (no recursion). This implies that the rewriting system can produce only one sequence: it is not a generalization of the sequence, it is simply a re-representation of it. We will nevertheless refer to the rewriting system as a “grammar.” Generally, such grammars involve fewer symbols than the original sequence: they eliminate repetition and are therefore an ideal basis for compression. Moreover, the rewrite rules can also be used for a number of other interesting applications, as we will see.

We are interested in comparing the size of grammars—that is, phrase hierarchies—produced by different methods on the same string. The size should reflect both the number of rules in the grammar, and the number of symbols on the right-hand side of the rules. Although these quantities are easy to compute, it is not clear how to combine them into an overall measure. One possibility is to use a partial order: if one grammar is smaller than another on both counts, it is manifestly superior in terms of size. In some applications it is also desirable to minimize the number of symbols that appear in the top-level string, since these reflect unstructured elements that cannot be absorbed into any rule.

Perhaps the best way of comparing the overall size of grammars would be to come up with an adaptive compression scheme and use their compressed size. However, this would divert us into areas of grammar coding, whereas we want to focus on different ways of constructing grammars. Moreover, while compression may be a reliable measure for large grammars, particular coding decisions would introduce significant bias for small grammars.

Consequently, to provide a definite and simple measure for comparison purposes, we define the size of a grammar as the number of rules plus the number of symbols on their right-hand sides. Note that the labeling of the rule heads is completely arbitrary; it need only be consistent with non-terminals in the rule bodies. By convention, we number or letter non-terminals consecutively, so that rule heads can be recreated if the rules are transmitted in a given order. Summing the number of rules and the number of right-hand-side symbols effectively adds a count of one for each rule, which corresponds to a terminator character (or, equivalently, a length indicator) for each rule. The grammar can certainly be represented in this number of symbols: left open, of course, is the size of each individual symbol. Although terminal symbols are chosen from a predetermined alphabet, the number of non-terminals grows with the grammar. Treating all symbols as equivalent is a convenient simplification which is good enough to allow us to derive interesting and revealing results.

Rather than dealing with both a sequence and a set of rewrite rules, we transform the sequence into a grammar rule as a first step, and then proceed by processing the grammar alone. The transformation is trivial: the string *abcdbc* becomes the grammar $S \rightarrow abcdbc$. The fundamental operation on the grammar is rule creation. This involves creating a new head and body, and replacing all instances of the body in the rest of the grammar with the head. For example, adding a new rule $A \rightarrow bcd$ to the grammar $S \rightarrow abcdebcdfbcd$ results in

$$\begin{aligned} &S \rightarrow aAeAfA \\ &A \rightarrow bcd \end{aligned}$$

Suppose that a substring of length $W$ occurs $N$ times in the input. Creating a new rule for this substring involves replacing $WN$ symbols in the input by one rule of length $W$, and $N$ non-terminals that indicate where in the sequence the substring occurs. According to our convention, the rule costs $W+1$ symbols, so the overall saving is $WN-(W+1+N)$, or

$$(W-1)(N-1)-2. \tag{1}$$

We will evaluate different tradeoffs using this calculation as our fundamental measure of compression.

## 4 Online vs Offline

Compression schemes generally operate online. They process the input stream in a single pass, and begin to emit compressed output before they have seen all of the input. This allows them to operate on arbitrary inputs within a fixed memory bound. However, current memory sizes are beginning to render such restrictions unnecessary: practical considerations of disk fragmentation and backup tend to constrain disk files within a reasonable maximum limit, and it is by no means unknown for a machine to have enough main memory to hold the largest of files. This permits offline operation.

### 4.1 Online techniques

Online operation severely restricts the opportunities for detecting repetitions, for there is no alternative to detecting repetitions in a greedy left-to-right manner. It may be possible to postpone decision-making by retaining a buffer of recent history and using this to improve the quality of the rules generated, but at some point the input must be processed greedily and a commitment made to a particular decomposition—that is inherent in the nature of (single-pass) online processing.

SEQUITUR is an algorithm that creates a hierarchical dictionary for a given string in a greedy left-to-right fashion (Nevill-Manning and Witten, 1997). It builds a hierarchy of phrases by forming a new rule out of existing pairs of symbols, including non-terminal symbols. Rules that become non-productive—in that they do not yield a net space saving—can be deleted, and their head replaced by the symbols that comprise the right-hand side of the deleted rules. This allows rules that concatenate more than two symbols to be formed. For example, the string *abcdbcabcdbc* gives rise to the grammar

$S \rightarrow AA$
$A \rightarrow aBdB$
$B \rightarrow bc$

**4.1.1 Linear-time operation** Surprisingly, SEQUITUR operates in time that is linear in the size of the input (Nevill-Manning and Witten, 1998). We present here a new simplified proof due to Richard Ladner (private communication). This proof also contains an explanation of how the algorithm works.

SEQUITUR operates by reading in a new symbol and processing it by appending it to the top-level string and then examining the last two symbols of that string. It then applies zero or more of the following three transformations until none applies anywhere in the grammar. It then repeats the cycle by reading in a new symbol.

At any given point in time, the algorithm has reached a particular point in the input string, and has generated a certain set of rules. Let $r$ be one less than the number of rules, and $s$ the sum of the number of symbols on the right-hand side of all these rules. Recall that the top-level string $S$, which represents the input read so far, forms one of the rules in the grammar; it begins with a null right-hand side. Initially, $r$ and $s$ are zero.

Here are the three transformations. Only the first two can occur when a new symbol is first processed; the third can only fire if one or more of the others has been applied in this cycle.

1. The digram comprising the last two symbols matches an existing rule in the grammar. Substitute the head of that rule for the digram. $s$ decreases by one; $r$ remains the same.

2. The digram comprising the last two symbols occurs elsewhere on the right-hand side of a rule. Create a new rule for it and substitute the head for both its occurrences. $r$ increases by one; $s$ remains the same (it increases by two on account of the new rule, and decreases by two on account of the two substitutions).

3. A rule exists whose head occurs only once in the right-hand sides of all rules. Eliminate this rule, substituting its body for the head. $r$ decreases by one; $s$ decreases by one too (the single occurrence of the rule's head disappears).

To show that this algorithm operates in linear time, we will demonstrate that the total number of rules applied cannot exceed $2n$, where $n$ is the number of input symbols. Consider the quantity $q = s - r/2$.

$q$ is initially 0
$q$ is never negative because $r \leq s$
$q$ increases by 1 for each input symbol processed
$q$ decreases by at least 1/2 for each rule applied (the first transformation decreases it by 1, the second by 1/2, and the third by 1/2).

Since $q$ increases by 1 for each input symbol processed and decreases by at least 1/2 for each rule applied, the number of rules applied is at most twice the number of input symbols. This completes the proof.

**4.1.2 Convergence to entropy** An important question for compression algorithms is whether they are "universal" for the class of stationary information sources over a finite alphabet. Universal codes are guaranteed to converge to the entropy of the source in the limit, as more symbols from the source are observed. Many statistically-based compression methods restrict the maximum length of the context that is taken into account when encoding the upcoming symbol; they are not universal algorithms because they cannot capture correlations that exceed that pre-set maximum length. The LZ77 and LZ78 algorithms are well known to be universal, but converge so slowly that they are roundly outperformed by limited-context statistical prediction methods on input sequences of any conceivable practical length.

The SEQUITUR algorithm that we have described, like the LZ methods, does not suffer from predefined limits on the size of rules and therefore there is no limit on the amount of context that can be taken into account. It also performs well on sequences of practical length. As the algorithm stands, however, experiments on random input show that the size of the grammar increases dramatically as more input is seen, and it is clear that convergence does not occur even with such a simple ergodic source.

Recently, Kieffer and Yang (in press) modified SEQUITUR in a way that admits a proof of universality. Though the proof is too long to sketch here, the modification is simple: it is to check, whenever a new rule is about to be created, whether its expansion is the same as the expansion of any existing rule. If it is, the existing rule is used instead. This dramatically alters the grammar generated by the algorithm on stochastic inputs, for the original algorithm needlessly duplicates many useless rules. For structured input such as natural language text, the change is not so marked. And unfortunately the modification destroys the linear execution-time bound derived above.

### 4.2 Offline techniques

To permit it to work in an online fashion, SEQUITUR processes the symbols in the order in which they appear. The first-occurring repetition is replaced by a rule, then the second-occurring repetition, and so on. If online operation is not required, this policy can be relaxed. This raises the question of whether there exist heuristics for selecting substrings for replacement that yield better compression performance. There are three obvious possibilities: replacing the most frequent digram first, replacing the longest repetition first, and replacing the most compressive repetition first.

**4.2.1 Most frequent first:** FREQUENT The idea of forming a rule for the most frequently-occurring digram, substituting the head of the rule for that digram in the input string, and continuing until some terminating condition is met, was proposed a quarter century ago by Wolff (1975) and has been reinvented many times since then.[1] The most common repeated digram is replaced first, rather than the first one as in SEQUITUR, and the process is repeated until no digram appears more than once. This algorithm operates offline because it must scan the entire string before making the first replacement.

Wolff's algorithm is inefficient: it takes $O(n^2)$ time because it makes multiple passes over the string, recalculating digram frequencies from scratch every time a new rule is created. However, recently Larsson and Moffat (1999) devised a clever algorithm, dubbed RE-PAIR, whose time is linear in the length of the input string, which creates just this structure of rules: a hierarchy generated by giving preference to digrams on the basis of their frequency. They reduce execution time to linear by incrementally updating digram counts as substitutions are made, and using a priority queue to keep track of the most common digrams.

For an example of the FREQUENT heuristic in operation, consider the string *babaabaabaa*. The most frequent digram is *ba*, which occurs four times. Creating a new rule for this yields the grammar

$S \to AAaAaAa$
$A \to ba.$

Replacing *Aa* gives

$S \to ABBB$
$A \to ba$
$B \to Aa,$

a grammar with eleven symbols (including three end of rule symbols). This happens to be the same as the length of the original string (without terminator).

**4.2.2 Longest first:** LONG A second heuristic for choosing the order of replacements is to process the longest repetition first. In the same string *babaabaabaa* the longest repetition is *abaa*, which appears twice. Creating a new rule gives

$S \to bAbaA$
$A \to abaa.$

Replacing *ba* yields

$S \to bABA$
$A \to aBa$
$B \to ba,$

resulting in a grammar with a total of twelve symbols.

Bentley and McIlroy (1999) explored the LONG heuristic for very long repetitions, and removed them using an LZ77 pointer-style approach before invoking *gzip* to compress shorter repetitions. This is not a linear-time solution.

Suffix trees (Gusfield, 1997) provide an efficient mechanism for identifying longest repetitions. A suffix tree is a compacted trie of suffixes in which every suffix of the original string is represented, and internal nodes correspond to repetitions. In the compaction operation, suffixes that share a prefix are merged up to the end of the common prefix. Thus the longest repetition corresponds to the deepest internal node, measured in symbols from the root. Since there is a one-to-one correspondence between leaf nodes and symbols in the string, the number of nodes in the tree is linear in the length of the input. The deepest non-terminal can be found by traversing the tree, which takes time linear in the length of the input.

We are left with two problems: how to find all longest repetitions, and how to update the tree after creating a rule. Farach-Colton and Nevill-Manning (in preparation) show that it is possible to build the tree, and update it after each replacement, in time which is linear overall. The tree can be updated in linear amortized time by making a preliminary pass through it and sorting the depths of the internal nodes. Sorting can be done in linear time using a radix sort, because no repetition will be longer than $n/2$ symbols. The algorithm relies on the fact that the deepest node is modified at each point: it does not generalize to the most-compressive-first heuristic where shorter rules may compress better than the longest repetition.

**4.2.3 Most compressive first:** COMPRESSIVE A third heuristic for choosing replacements falls between the extremes of most frequent and longest: it replaces the most

[1] Only last month we received for refereeing a paper claiming discovery of this algorithm.

compressive repetition first. Whether a corresponding linear-time algorithm exists is an open question.

In the string *babaabaabaa*, the most frequent digram is *ba*, which appears four times, and the longest repetition is *abaa*, which appears twice. Replacing either of these yields a total of eleven symbols on the right hand side of the two rules in the grammar. The substring *aba* appears three times: it is neither the longest or the most common. However, replacing it results in ten symbols.

Expression 1 above gives the savings generated by replacing a substring of length $W$ that appears $N$ times. At each point, the substring that maximizes this quantity is replaced. For our example string, $(W, N) = (3, 3)$ gives a saving of two symbols, whereas $(W, N) = (4, 2)$ and $(2, 4)$, the longest and most frequent replacements respectively, save only one symbol.

## 5 Implementation

Although two of the offline heuristics have linear time solutions, we chose to implement all three in a uniform, simple, but less efficient way using a suffix array. A suffix array is a sorted list of all suffixes of a string, and can be constructed by initializing an array of pointers to every character in the string and sorting the array according to the lexicographic ordering of the suffixes denoted by the pointers. Then the array is scanned for a replacement that satisfies the particular heuristic under consideration.

To find the longest repetition, adjacent entries in the suffix array are compared: after scanning the entire array, the longest match is replaced. To find the most compressive repetition, each entry in the array is compared to every following entry until the two only match on the first character; then, for each match, the corresponding saving is calculated from the length of the match and the distance between the entries—which gives the frequency of the repetition. Again, once the entire array has been scanned, the maximally compressive repetition is replaced. A similar procedure is used for the most frequent repetition.

After the first replacement, a new rule is formed. In the case of LONG and COMPRESSIVE, new rules might contain strings that are repeated elsewhere, and so the content of the new rule is added to the suffix array. The process of constructing the suffix array, finding an appropriate repetition, and making a new rule is repeated until no more replacements can be made.

This process can be accelerated in several ways. First, instead of reconstructing the array of pointers from scratch and resorting it each time, the suffix array can be updated. In fact, if newly-created rules are placed in the space occupied by one of the repetitions, there are already pointers to the suffixes in the rule. However, they may be out of order, because they are truncated at the end of the rule. Also, suffixes that precede a repetition that has been replaced have been similarly truncated. To move these to their correct position in the array, we perform a lazy bubble sort: when an entry is found to be out of order it percolates down to its correct position, and scanning resumes from there.

A second optimization can be made by noting that there are often several replacements that yield equivalent savings. For example, near the end of the LONG procedure, there are many matches of length two. This calls for an arbitrary choice, and it is most efficient to choose the first match. Furthermore, it is unnecessary to start from the beginning of the array each time: instead, all repetitions of a particular length can be replaced in a single pass through it. Both optimizations significantly decrease the time taken, although they do not affect the overall time complexity.

## 6 Evaluation

We first demonstrate that there are strings on which some schemes outperform other schemes. For didactic purposes we use very small examples; this is followed by larger-scale analyses of natural and artificial sequences.

### 6.1 Small-scale examples

The first example of Figure 1a shows a case where FREQUENT's replacement of *aa* results in a smaller grammar, in terms of both the symbol count, $s$, and the rule count, $r$, than LONG's replacement of *aaa*. The second example shows where replacing the longest repetition, *abbb*, is better than replacing the most frequent, *bb*.

The third example raises a new issue: how to choose amongst overlapping repetitions. As noted above, we choose to replace the leftmost occurrence. In fact, in this example the FREQUENT heuristic could mimic LONG by replacing the second *aa* in the substring *aaa*, thereby obtaining a better result. However, there is no general way, other than trial and error, of choosing the best overlapping repetition.

Figure 1b compares COMPRESSIVE with the other schemes. By its nature, it can never be worse than the other schemes so far as a single replacement is concerned. The first two examples, based on the same string, show that a string of nine $a$s is better compressed into three triplets than four duplets or two quadruplets.

We will see below that, in practice, LONG is significantly inferior to the other techniques. This is at first surprising because the heuristic only governs the *order* of replacements: LONG will eventually replace substrings of length two. To understand the problem, consider the string $aa|_1aaa|_2aaaa|_3...|_{n-1}a^{n+1}$. Each occurrence of "$|_i$" stands for a different symbol, so that this character acts as a separator and rules cannot be formed across it. LONG forms $n$ rules:

$$\begin{aligned}
&S \rightarrow A\ |_1\ B\ |_2\ C\ |_3\ D\ |_4\ E\ ...\\
&A \rightarrow aa\\
&B \rightarrow Aa\\
&C \rightarrow Ba\\
&D \rightarrow Ca\\
&E \rightarrow ...
\end{aligned}$$

where the rules are formed in reverse order, rule $A$ last.

On this sequence FREQUENT begins by forming $A \rightarrow aa$, then $B \rightarrow AA$, and so on up to a rule that expands to as many $a$s as the largest power of two smaller than $n + 1$. Now each string between the separator characters is expressed as a kind of binary Roman numeral: a string of non-terminals that each stand for $2^k$ $a$s (for some $k$), possibly followed by a final $a$. Next, repeated pairs of non-terminals are replaced. If ties are broken by performing replacements left to right, non-terminals are in order from longest content to shortest. There are $\lfloor \log n \rfloor$ non-terminals, so the number of pairs is $\sum_1^{\lfloor \log n \rfloor - 1} \approx \log^2 n$. These can again be combined, giving a number of rules that is polynomial in $\log n$. Thus the number of rules grows much more slowly for FREQUENT than for LONG.

Figure 2 shows behavior of FREQUENT and LONG on sequences of the kind just discussed for $n < 400$. The number of symbols in rule S grows linearly in $n$, the length of the longest repetition, for both heuristics, with FREQUENT growing faster. The total number of symbols grows faster for LONG, however, and the reason can be seen in Figure 2b. As we expect, the number of rules that LONG produces grows linearly, whereas growth is approximately logarithmic for FREQUENT.

### 6.2 Natural language text

Figure **??** shows grammar sizes for a 50 Kb excerpt from Thomas Hardy's novel *Far from the Madding Crowd*, a test file from the Calgary compression corpus (Bell *et al.*, 1990). The height of the bars represent the total number of symbols in the grammar, while the solid parts represent the number of symbols in the top-level rule of the grammar, rule $S$. COMPRESSIVE outperforms FREQUENT by a small margin. SEQUITUR is next, with LONG lagging behind.

As foreshadowed above, LONG performs significantly worse than the other techniques. The most compressive repetitions in English text are common short rules, not infrequent long ones. By Expression 1, a string of length two that appears $n$ times saves the same number of symbols as a string of length $n$ that appears twice. In *Far from the Madding Crowd*, the most common pair of symbols is *e_*, which appears 14097 times, whereas the longest repetition is only 105 characters long.[2] Indeed, apart from *_the*, the 15 most compressive replacements are only two symbols long. FREQUENT is clearly a better approximation to COMPRESSIVE than is LONG.

Note, though, that LONG has fewer symbols in rule $S$ and therefore accounts for more structure at the surface level, which is an interesting aspect of the compressibility of the string. Minimizing the length of rule $S$ is appropriate if the objective is not overall compression, but rather segmentation into a small number of repeating elements to gain insight into its structure. For example, it might be useful to segment a piece of English text into the fewest "words" to analyze its lexical structure.

### 6.3 L-systems

To highlight the difference between the four heuristics, we performed experiments on longer artificial sequences. These were produced by L-systems, which are used to model the growth of plants (Prusinkiewicz and Hanan, 1989) and other natural objects. For example, the grammar of Figure **??** gives rise to the plant-like shape shown below after five applications of the rewriting rule when the five symbols $f$, [, ], $+$ and $-$ are interpreted as appropriate graphical commands in the Logo language (Abelson and diSessa, 1980) – step forward, save position and orientation, restore position and orientation, turn right, and turn left respectively. In L-systems, all rewrites are done in parallel. Because they produce a hierarchy of repetitions, the result of each heuristic can be assessed according to how close the inferred grammar is to the original.

Four input sequences were generated by the L-systems detailed in Table 1, which shows each grammar, the number of derivation steps, and the size of the artificial string produced. Figure 5 illustrates how well the heuristics compress these artificial strings. In each case, the original L-system size is included on the left. Note that the grammars in Table 1 involve recursion, yet the hierarchical rules that are inferred are non-recursive and capture the particular input string exactly. For this reason, the original L-system size in Figure 5 is calculated as the size of the equivalent non-recursive L-system corresponding to the number of derivation steps used, a non-recursive expansion of the structures in Table 1.

As before, the height of the bars in Figure 5 represent the total number of symbols in the inferred grammar and the solid parts show the size of rule $S$. In each case, COMPRESSIVE performs best overall—even outperforming the original L-systems, because they include repeated elements like *f-* in Table 1a. While LONG is always worst, in all but the second case it produces the shortest rule $S$. The online SEQUITUR algorithm performs less well than COMPRESSIVE and FREQUENT on all but the third example.

Qualitatively, COMPRESSIVE is the only method that consistently arrives at a structure similar to the original L-system. In order to infer the rewrite rules, the repetition boundaries must be detected very precisely. Otherwise higher-level rules must account for extra or missing symbols, which destroys their ability to mimic the structure of the original rules. For example, Table **??** shows the grammars inferred by COMPRESSIVE and FREQUENT for L-system. COMPRESSIVE's grammar is much smaller, and if rules A through G are expanded, the grammar consists of five versions of the original recursive rule, one for each application. In contrast, the grammar inferred by FREQUENT reflects none of the structure of the source grammar.

## 7 Applications

Efficient methods for inferring hierarchical structure from sequences, which arose in the field of data compression, have numerous applications elsewhere. The three sketched below all use SEQUITUR, because it was the first linear-time algorithm for hierarchical structure detection, but the other heuristics are likely to be equally suitable—perhaps more so.

### 7.1 Hot subpath identification

The sequence of machine instructions executed by a program is highly repetitive. Whereas repetition is used in data com-

[2] This is a copying error in the text. The longest repetition in the corrected text is *which was to be found nowhere in*, which Hardy repeats to good effect in *Suddenly an unexpected series of sounds began to be heard in this place up against the sky. They had a clearness which was to be found nowhere in the wind, and a sequence which was to be found nowhere in nature. They were the notes of Farmer Oak's flute.*

pression to save space and bandwidth, it can be used by optimization techniques to save computation. The first step in program optimization identifies frequently-executed sequences of instructions: these are the paths that will yield the greatest improvement if optimized. The granularity with which program traces are analyzed is often based on a static analysis of the program code: for example, counting the number of times an entire function or a loop is executed. However, the program's dynamic behavior might contribute additional regularities, and the techniques described above provide a non-parametric way of decomposing a stream of instructions that allows unexpected structures to be discovered.

Larus (1999) shows how SEQUITUR can be used to infer "whole program paths," which are decompositions of an instruction stream that could be used as the basis for further optimization. The paths identified in the SPECINT95 commercial benchmark promise a significant opportunity for optimization. The hierarchical decomposition may also be useful for dynamic optimization, improving instruction scheduling and cache loads within a microprocessor.

### 7.2 Segmentation for text-to-speech conversion

Hierarchical grammar inference has been used to improve speech synthesis. Because words do not usually correspond to phonetic units, Martin (1999) used SEQUITUR to segment the input hierarchically; they attach phonemes to rules at the appropriate level. The techniques described here segment text into common substrings, which often correspond to phonetic units. This is not surprising: although English orthography is not strictly phonetic, there is nevertheless a high degree of correlation between certain substrings and phonemes, and the modularity of phonemes is reflected in the modularity of corresponding substrings.

Martin (1999) developed a client-server scheme for speech synthesis in which the server uses a grammar derived automatically from a large corpus of text to segment natural-language strings sent to it by a client. The individual segments, corresponding to individual grammar rules, are translated to phonetic strings individually and returned to the client for speaking. If users wish to improve the rendition, they edit the phonetic strings interactively, whereupon the strings are transmitted back to the server and replace the original phonetic transcriptions of those segments. Thus the system improves gradually as users interact with it.

SEQUITUR was chosen for this application because of its ability to process a large amount of training text very quickly. Based on the results in the present paper, we expect COMPRESSIVE and FREQUENT to perform even better on this task.

### 7.3 Browsing large text collections

Hierarchical phrase structures suggest a new way of approaching the problem of familiarizing oneself with the contents of a large collection of electronic text. Nevill-Manning *et al.* (1999) presented the hierarchical structure inferred by SEQUITUR interactively to the user. Users can select any word from the lexicon of the collection, see which phrases it appears in, select one of them and see the larger phrases in which it appears, and so on. Although reminiscent of the permuted title or keyword-in-context (KWIK) indexes of days gone by, there are two crucial differences. First, the new interface presents a *hierarchical* structure of phrases. This greatly reduces the size of the index and allows the user to home in on useful information in logarithmic time. Second, phrases are restricted to those that occur more than a predetermined number of times—usually twice or more. This shifts attention from individual items towards the content of the collection as a whole.

Figure 6 shows a screen display of a hierarchy based on the complete text of the *Computists' Communique*, an online AI research news magazine. SEQUITUR has been used in word mode, with complete words as tokens rather than individual characters. Punctuation is removed and words are folded to lower case. The vocabulary of the collection is displayed alphabetically in the leftmost column. Users can select a word (either with the mouse or by typing), and all phrases in which it appears are displayed, along with the number of times each phrase occurs.

In Figure 6 the user has selected *index* from the vocabulary and the phrases it appears in are listed in the next column to the left. For example, *index htm* appears six times. Note that this particular phrase appears as an artifact of word parsing: it emanates from the filename *index.htm*—as of course does *index html*, further down the list. It is encouraging that these junk entries consume far less space in the list than they would in a conventional query for the term *index*. Each phrase can be selected and expanded in turn. The user has selected *indexing and retrieval*, which also appears six times in the corpus. In this particular case, each of these six phrases occurs exactly once and cannot be expanded any further.

The user can traverse the grammar, extending and hence specializing the query term. Every word is the root of a tree structure whose leaves are the occurrences of that word in the top-level rule. Occurrences in other rules are internal nodes corresponding to phrases that contain the word. Those phrases are themselves used elsewhere in the grammar, either in the top-level rule of in other rules for longer phrases. It is possible to stop at any internal node and use that phrase as a query term, or continue down the tree to a leaf and retrieve the corresponding document.

## 8 Conclusion

This paper has examined the tradeoffs inherent in different heuristics for data compression using hierarchical structure inference. The work was stimulated by the existence of an efficient, linear-time, online algorithm, SEQUITUR, and the recent discovery that offline techniques can also work in linear time. While an online algorithm is forced to identify rules soon after they first occur, offline algorithms offer more scope for different preference heuristics. We have studied three: most frequent first, longest first, and most compressive first. Throughout we measure compression in terms of the total number of symbols in the grammar.

Not surprisingly, COMPRESSIVE is the best method. In large-scale tests with regular L-system structures it consistently finds grammars that are smaller than a non-recursive version of the generating grammar. Note, however, that like all these heuristics it strives only for *local* optimality, and so could conceivably be outperformed by other methods—though they would likely be far more costly. At the other extreme, LONG produces consistently poor results—much to our initial surprise. However, it does tend to minimize the size of the top-level rule $S$, a kind of "surface" representation of the input, that could be useful in some applications.

Both FREQUENT and LONG, like SEQUITUR, take time linear in the length of the input sequence. Again this is a surprising result, and the linear-time algorithms are quite intricate. Although such an algorithm has not been demonstrated for COMPRESSIVE, neither has a tight lower bound: this is clearly an area for future investigation.

For practical data compression using hierarchy inference, if online operation is necessary SEQUITUR is the only reasonable choice. If offline operation is acceptable, FREQUENT is the recommended technique, for experiments show that it performs almost as well as COMPRESSIVE and yet works in linear time. However, if one is concerned to infer the best structure for a sequence, or the exact structure of an artificial sequence, COMPRESSIVE gives better results in exchange for additional execution time.

Finally, there are many applications of hierarchical structure inference techniques outside the data compression areas. The existence of new linear-time algorithms broadens the choice for application developers.

### Acknowledgements

We appreciate numerous enlightening discussions with Martin Farach-Colton about various algorithmic issues.

### References

Abelson, H. and diSessa, A. (1980) *Turtle geometry.* MIT Press, Cambridge, MA.

Bentley, J. and McIlroy, D. (1999) "Data compression using long common strings." *Proc Data Compression Conference*, pp. 287–295. IEEE Press, Los Alamitos, CA.

Cameron, R.D. (1988) "Source encoding using syntactic information source models." *IEEE Trans Information Theory*, Vol. 34, No. 4; pp. 843–850; July.

Gusfield, D. (1997) *Algorithms on strings, trees, and sequences.* Cambridge University Press, Cambridge, UK.

Kieffer, J.C. and Yang. E.-H. (2000) "Grammar based codes: a new class of universal lossless source codes." *IEEE Trans on Information Theory*, 46, 737–754.

Larsson, N.J. and Moffat, A. (1999) "Offline dictionary-based compression." *Proc Data Compression Conference*, pp. 296–305. IEEE Press, Los Alamitos, CA.

Martin, A.R. (1999) "Intelligent Speech Synthesis Using the Sequitur Algorithm and Graphical Training: Server Software," M.S. Thesis, Engineering Science, University of Toronto.

Miller, V.S. and Wegman, M.N. (1985) "Variations on a theme by Ziv and Lempel." In *Combinatorial algorithms on words*, edited by A. Apostolico and Z. Galil, pp. 131–140. NATO ASI Series, Vol. F12. Springer-Verlag, Berlin.

Nevill-Manning, C.G. and Witten, I.H. (1997) "Identifying hierarchical structure in sequences: a linear-time algorithm." *J Artificial Intelligence Research*, Vol. 7, pp. 67-82.

Nevill-Manning, C.G., Witten, I.H. and Paynter, G.W. (1999) "Lexically-generated subject hierarchies for browsing large collections." *International Journal of Digital Libraries*, Vol. 2, No. 2/3, pp. 111-123.

Nevill-Manning, C.G. and Witten, I.H. (1998) "Phrase hierarchy inference and compression in bounded space," *Proc. Data Compression Conference*, J.A. Storer and M. Cohn (Eds.), Los Alamitos, CA: IEEE Press. 179–188.

Prusinkiewicz, P. and Hanan, J. (1989) *Lindenmayer systems, fractals, and plants.* Springer-Verlag, New York.

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

Ziv, J. and Lempel, A. (1977) "A universal algorithm for sequential data compression." *IEEE Trans Information Theory*, Vol. IT-23, No. 3, pp. 337–343; May.

Ziv, J. and Lempel, A. (1978) "Compression of individual sequences via variable-rate coding." *IEEE Trans Information Theory*, Vol. IT-24, No. 5, pp. 530–536; September.

| | string | FREQUENT | LONG |
|---|---|---|---|
| a | aabaaaaaa | S → AbAAA | S → BbAA |
| | | A → aa | A → Ba |
| | | | B → aa |
| | | ($s$=7, $r$=2) | ($s$=8, $r$=3) |
| | abbbbabbb | S → BABb | S → AbA |
| | | A → bb | A → abbb |
| | | B → aA | |
| | | ($s$=8, $r$=3) | ($s$=7, $r$=2) |
| | aabbaaabb | S → ABAaB | S → AaA |
| | | A → aa | A → aabb |
| | | B → bb | |
| | | ($s$=9, $r$=3) | ($s$=7, $r$=2) |

| | string | COMPRESSIVE | LONG |
|---|---|---|---|
| b | aaaaaaaaa | S → AAA | S → AAa |
| | | A → aaa | A → BB |
| | | | B → aa |
| | | ($s$=6, $r$=2) | ($s$=7, $r$=3) |

| string | COMPRESSIVE | FREQUENT |
|---|---|---|
| aaaaaaaaa | S → AAA | S → BBa |
| | A → aaa | A → aa |
| | | B → AA |
| | ($s$=6, $r$=2) | ($s$=7, $r$=3) |

Figure 1 Comparisons: (a) FREQUENT *vs* LONG; (b) COMPRESSIVE *vs* LONG and FREQUENT

| | grammar | steps | size |
|---|---|---|---|
| a | `S → f-f-f-f`<br>`f → f+f-f-ff+f+f-f` | 4 | 30427 |
| b | `S → f`<br>`f → f[-f]f[+f]f` | 5 | 7811 |
| c | `S → f++f++f`<br>`f → f-f++f-f` | 5 | 7168 |
| d | `S → f-f-f-f`<br>`f → f-F+ff-f-ff-fF-ffF -ff+f+ff+fF+fff` | 3 | 43911 |

Table 1 L-systems, the number of derivations and size of derived test strings

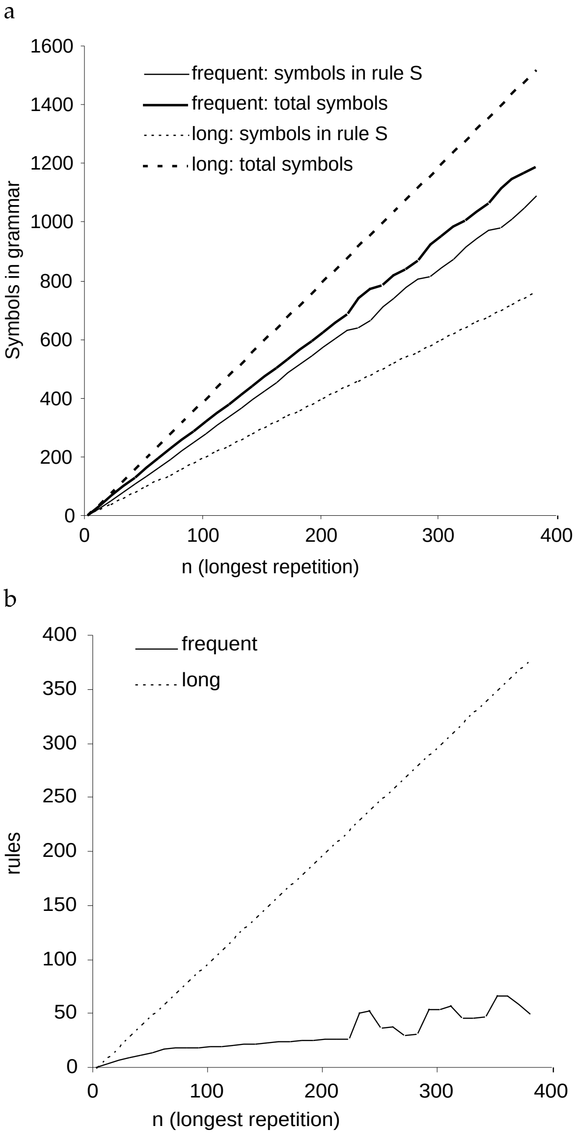


Figure 2 Performance of FREQUENT and LONG on the sequence *aa* $|_1$ *aaa* $|_2$ *aaaa* $|_3$ ... $|_{n-1}$ $a^{n+1}$. $n+1$ is displayed on the horizontal axes (a) symbols in rule S and overall, (b) rules

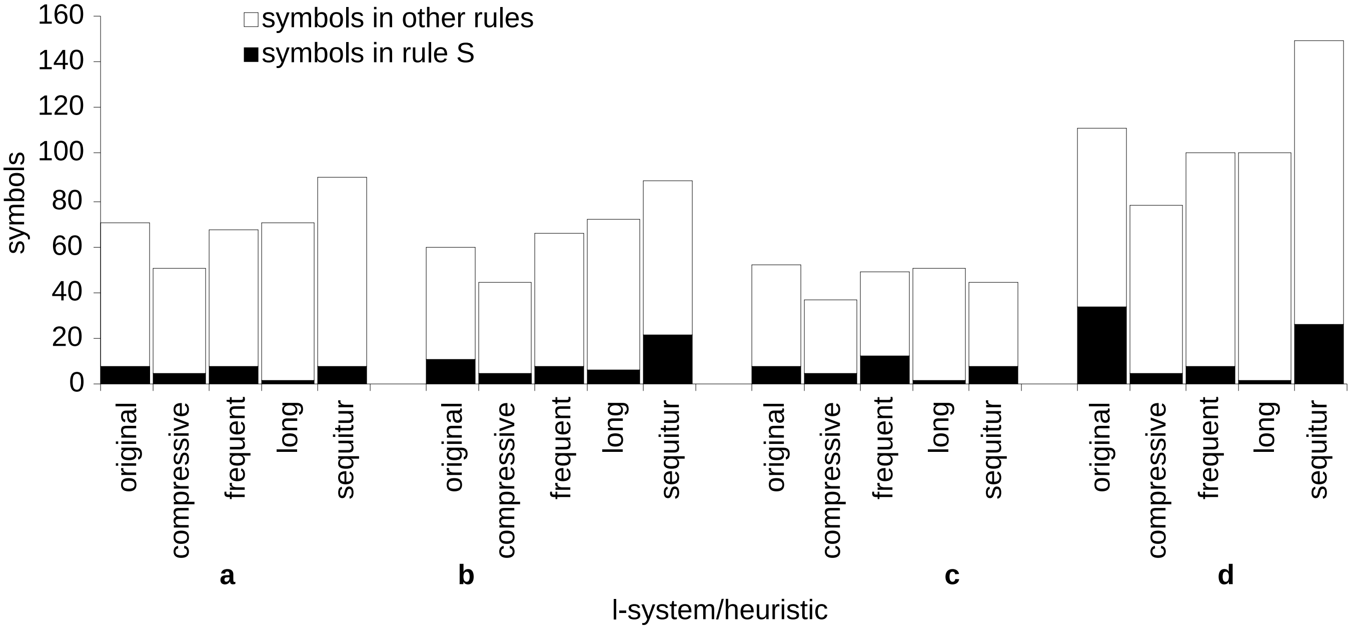
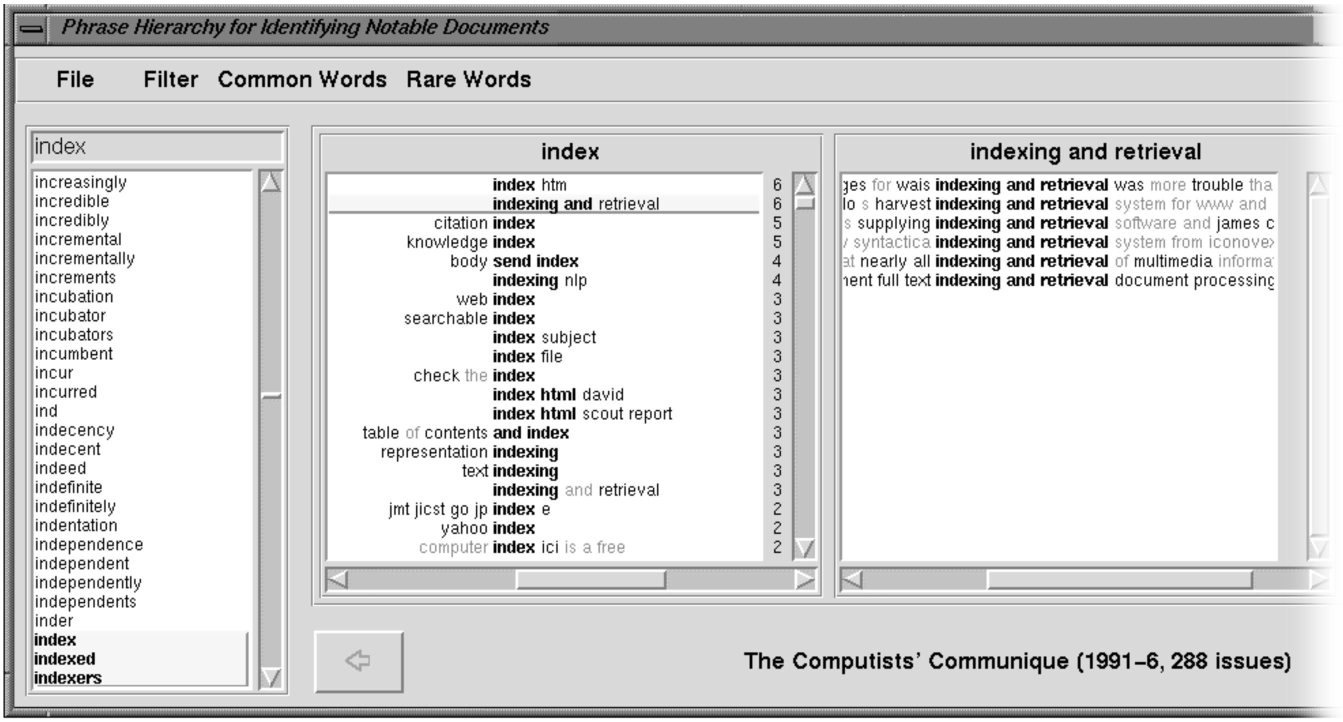


Figure 5 Size of grammar produced for the four L-systems by the four different methods



index

increasingly
incredible
incredibly
incremental
incrementally
increments
incubation
incubator
incubators
incumbent
incur
incurred
ind
indecency
indecent
indeed
indefinite
indefinitely
indentation
independence
independent
independently
independents
inder
**index**
**indexed**
**indexers**

index

**index** htm 6
**indexing and** retrieval 6
citation **index** 5
knowledge **index** 5
body **send index** 4
**indexing** nlp 4
web **index** 3
searchable **index** 3
**index** subject 3
**index** file 3
check the **index** 3
**index html** david 3
**index html** scout report 3
table of contents **and index** 3
representation **indexing** 3
text **indexing** 3
**indexing** and retrieval 3
jmt jicst go jp **index** e 2
yahoo **index** 2
computer **index** ici is a free 2

indexing and retrieval

ges for wais **indexing and retrieval** was more trouble tha
lo s harvest **indexing and retrieval** system for www and
s supplying **indexing and retrieval** software and james c
/ syntactica **indexing and retrieval** system from iconove
at nearly all **indexing and retrieval** of multimedia informa
nent full text **indexing and retrieval** document processing

The Computists' Communique (1991–6, 288 issues)

Figure 6 Hierarchically browsing the *Computists' Communique*