Phrase hierarchy inference and compression in bounded space

Craig G. Nevill-Manning
Biochemistry, Stanford University, Stanford, CA 94305-5307
(650) 723 5976; cnevill@stanford.edu

Ian H. Witten
Computer Science, University of Waikato, Hamilton, New Zealand
+64 7 838–4246; ihw@waikato.ac.nz

1. Introduction

Text compression by inferring a phrase hierarchy from the input is a recent technique that shows promise both as a compression scheme and as a machine learning method that extracts some comprehensible account of the structure of the input text. Its performance as a data compression scheme outstrips other dictionary schemes, and the structures that it learns from sequences have been put to such eclectic uses as phrase browsing in digital libraries, music analysis, and inferring rules for fractal images.

Originally introduced in basic form in 1994 (Nevill-Manning *et al.*, 1994), phrase hierarchy inference was later extended to provide extremely effective compression of large quantities of semi-structured text (Nevill-Manning *et al.*, 1996). By its very nature, such text presents opportunities for capitalizing on structure inference and using it to advantage for compression; however, the inherently domain-dependent nature of semi-structured text presents little room for the development of general practical or theoretical results that apply to a wide variety of different input. However, the simplicity and elegance of the inference algorithm, which can be described in terms of two very simple constraints, led to a proof that it operates in time linear in the length of the sequence (Nevill-Manning and Witten, 1997a)—a property that is taken for granted in text compression circles but which is unique in grammatical inference. Moreover, one of the cardinal disadvantages of the algorithm for on-line use, namely the fact that, although it forms the grammar incrementally, it has to process the entire input string at once before transmitting any part of it, was removed by the development of an incremental version of the transmission algorithm.

In the present paper we focus attention on the memory requirements of the method. Since the algorithm operates in linear time, the space it consumes is at most linear with input size. The space consumed does in fact grow linearly with the size of the inferred hierarchy, and this makes operation on very large files infeasible. The current implementation consumes 20 bytes per symbol in the hierarchy, and on a large file of text there are about 7 times fewer symbols in the hierarchy than there are symbols in the input. Thus to compress a 100 Mb input file requires around 300 Mb of memory. As with other compression programs, it is not feasible to use a disk cache because of the random nature of the memory access.

Space consumption that is linear with input size is nothing new: it is endemic to high-performance compression programs, at least in their "pure" form. Actual implementations, on the other hand, take a pragmatic approach to space usage: when memory runs out they discard the current model and start afresh, perhaps from scratch, perhaps from a cache of

recent input (e.g. COMPRESS). Some models operate in finite memory by placing limits on the size of internal dictionaries (e.g. LZW) or lookback buffers (e.g. GZIP): by doing so, they inevitably sacrifice any convergence properties that are associated with pure versions of the algorithms. Some compression models operate in finite memory in principle (e.g. PPM, with its fixed maximum context length) but the limit is so large that they too have to discard and reinitialize the model when overflow occurs. Recent schemes such as PPM* make a virtue of maintaining unbounded context. More rational policies of "forgetting" (e.g. least recently-used, least frequently-used) may improve compression over these simplistic techniques, but they require so much extra space for bookkeeping that any improvement is far outweighed by the decrease in compression caused by the much smaller model that can be accommodated.

Drastic space-limitation measures squander the virtue of hierarchy inference. In hierarchy inference, we are often interested just as much in the structure of the model as in the compression obtained from it. To throw away the model just as it gets big enough to become interesting violates common sense. Far better to retain, and reinforce, those parts of the hierarchy that are being used productively, and to selectively eliminate parts that are ineffective predictors. But how? Maintaining extra information expressly for the purpose of accomplishing "forgetting" more effectively is wasteful. Moreover, it seems very hard to arrange, except in a clumsy manner. In hierarchy inference, experience has taught us the value of seeking solutions that are, above all, elegant.

This paper describes two elegant ways of curtailing the space complexity of hierarchy inference, one of which yields a bounded-space algorithm. We begin with a brief review of the hierarchy inference procedure that is embodied in the SEQUITUR program. Then we consider its performance on quite large files, and show how compression performance improves as file size increases. We recognize that hierarchy inference will never define the state of the art in general-purpose text compression, not in practice (compared with PPM variants), chiefly because of its fundamentally non-statistical nature, nor in principle (compared with LZ methods), because it is easy to see that it fails to perform well for random input.[1] However, compression performance does improve markedly as file size increases, and it is interesting to see how. In Section 4 we look at one way of reducing the space occupied: adopt a more conservative policy towards the formation of rewrite rules. This has a dramatic effect on the number of rules produced, and presumably (though we have not yet studied it) on their comprehensibility and applicability. It can also yield a very slight improvement in compression. Section 5 examines a second way: a bounded-space version of the algorithm that transmits old information to make room for new, and garbage collects rules that are no longer useful.

2. Hierarchy inference

It is necessary to briefly review the basic algorithm that we have developed for inferring a grammatical hierarchy from a text sequence and using it for compression. A fuller description can be found in Nevill-Manning (1996) and Nevill-Manning and Witten (1997a), while a similar but less efficient system is described by Wolff (1980). The grammar is based on repeated phrases in the sequence. Each repetition gives rise to a grammar rule, 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, generate a hierarchical structure for the sequence.

---

[1] We do believe, however, that it will define the state of the art in compression of "structured" text, although the concept seems too vague to allow such a claim to be defended in any general way.

The following two constraints are maintained in the grammar:

> p_1: no pair of adjacent symbols appears more than once in the grammar;
> p_2: every rule is used more than once.

Property p_1 requires that every digram in the grammar be unique, and will be referred to as *digram uniqueness*. Property p_2 ensures that each rule is useful, and will be called *rule utility*. These two constraints exactly characterize the grammars that SEQUITUR generates. In fact, Sequitur's operation consists chiefly of 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 grammar starts out with a single rule, which we shall *S*. Rule *S* expands to reproduce the entire sequence; we shall return to this point later. When SEQUITUR observes a new symbol, it appends it to rule *S*. The last two symbols of rule *S*—the new symbol and its predecessor—form a new digram. If this digram occurs elsewhere in the grammar, the first constraint has been violated. To restore it, a new rule is formed with the digram on the right-hand side, headed by a new non-terminal symbol. The two original digrams are replaced by this non-terminal symbol. 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 non-terminal symbol that heads the existing rule replaces the digram.

Rules longer than two symbols are formed by the effect of the rule utility constraint, which ensures that every rule is used more than once. When a longer repetition that subsumes an existing rule is observed, the new rule factors out both occurrences of the original non-terminal. Because this non-terminal only appears once in the grammar, the original rule is superfluous, and can be deleted, substituting its contents in place of the lone non-terminal. 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.

The resulting grammar is illustrated in Figure 1. For realistically-sized sequences, rule *S* becomes long, while the other rules tend to remain short. The grammar can be decomposed into the rules other than rule *S*, which correspond to repetitions in the text, and rule *S*, which restates the input sequence in terms of the other rules. From a 60 Mb sample of computer science technical reports, SEQUITUR inferred a grammar with one million rules and nine million symbols. Over half of the symbols in the grammar were contributed by rule *S*. More structured or repetitive sequences will tend to have a shorter rule *S*.

3. Performance on large files

It is axiomatic in computing that, due to advances in hardware, computational power and capacity grows exponentially over time. Unfortunately, the quantities of data upon which algorithms are required to operate grow in a similar fashion. This is clearly illustrated in the evolution of data compression benchmarks. When the Calgary corpus (Bell *et al.* 1990) appeared, all files were under a megabyte in size, yet were considered large at the time. Arnold and Bell (1997) introduced a replacement for the Calgary corpus, the Canterbury corpus, that includes a set of large files ranging from two to four Mb in size. The TREC competition (Harman, 1992) and the New Zealand Digital Library (Witten, Cunningham and Apperley, 1996) routinely deal with multi-gigabyte corpora. In this inflationary

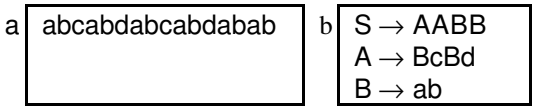

Figure 1 (a) a sequence and (b) the grammar inferred from it

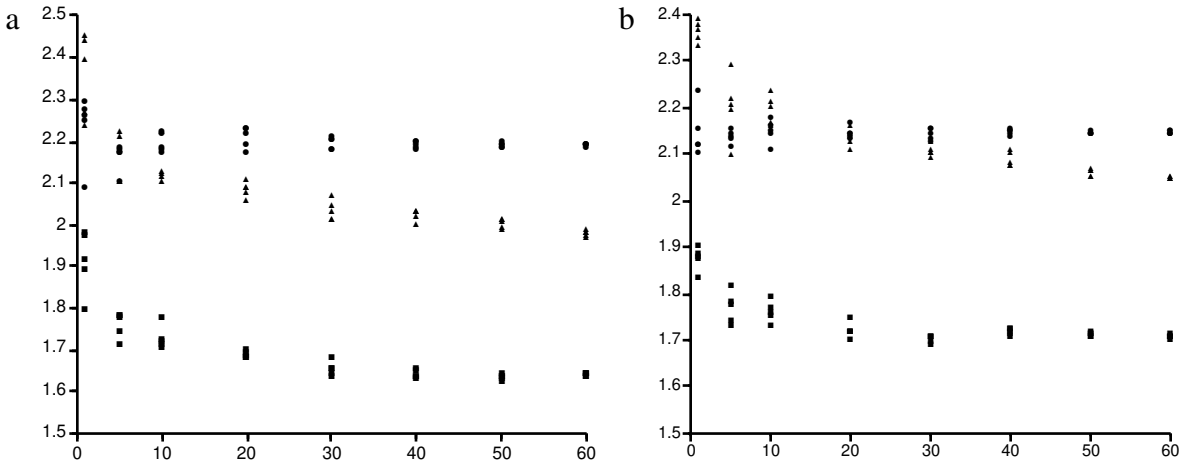

Figure 2 Compression rate against size for (a) Humanities collection, (b) Computer Science Technical Reports, for restricted PPMC (●), PPMD (■) and SEQUITUR (▲)

environment, it is instructive to revisit data compression schemes periodically to examine the effects of larger files on existing algorithms.

One striking effect of size on adaptive compression algorithms is the improvement in compression performance as more data is seen. Nevill-Manning and Witten (1997b) observed, for example, that SEQUITUR performed worse than PPMC on the (now) small files in the Calgary corpus, but better on two of the files from the large Canterbury corpus. The experiment reported here improves on our earlier experiment in three ways: we use larger input sequences, a range of input sizes created by random sampling from a large corpus, and more appropriate compression schemes for comparison.

Figure 2 plots for two large corpuses of English text. We sampled text from two of the collections in the New Zealand Digital Library. The text used for Figure 2a was extracted from the *Humanity Development Library*, a collection put together by the Global Help Project to provide ideas, experiences, and solutions to workers in developing countries. That used for Figure 2b was text sampled from the 40,000 technical reports in the *Computer Science Technical Report* collection, text that has been extracted automatically from original PostScript files. In each case we took samples of a given size from the much larger corpus, obtained the overall compression figure in bits/character, and repeated the sampling five times to get an idea of the variance in compression. The procedure was repeated for sample sizes from five to sixty Mb.

Three compression schemes were used: SEQUITUR, the now-standard implementation of PPMC with a 3-character context but limited memory (Moffat, 1990), and a modified version of PPMC using escape method D, a 7-character context, and unlimited model size[2]. Unfortunately we were unable to obtain a LZ compressor that could take advantage of large amounts of memory. The comparison with PPM in Nevill-Manning and Witten (1997b) was based just on the PPMC implementation, which is restricted to a 3-character context so that it runs in a reasonable amount of memory.

It is apparent from all three graphs that the longer-context PPM model performs much better than the shorter one. Even for PPMD, performance flattens out for large files, and it is possible that an 8-character context model would perform better still. In both plots SEQUITUR's performance is intermediate between the four- and five-character context versions of PPM. This underscores our belief that SEQUITUR will probably never compete, in

---

2 PPMD used a maximum of 250 Mb of memory during these experiments; SEQUITUR used 180Mb.

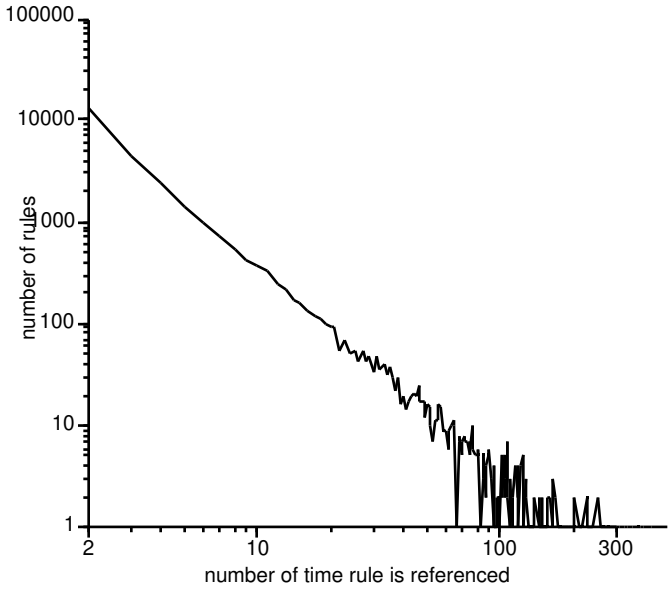

Figure 3 Histogram of rule usage, for *book1* of the Calgary corpus

terms of raw compression, with context-based statistical methods. Nevertheless we still believe that it is an interesting compression scheme, partly because of the explicit nature of the model it forms and partly because of its superior ability to take advantage of semi-structured text.

4. Inhibiting rule formation

A typical grammar formed by SEQUITUR contains a predominance of rules that are used a rather small number of times. Figure 3 shows the number of rules that appear twice, three times, etc. in the grammar inferred from *book1* of the Calgary corpus. Out of 27,000 rules, nearly half appear only twice in the grammar. Of the remaining rules, nearly a third appear only three times. It seems intuitively obvious that for larger texts, one can afford to be far more conservative about rule formation—perhaps even tailoring the minimum number of occurrences that are required before a rule is created to the amount of text that has been processed. In order to investigate this, we decided to reduce the number of rules that are created by delaying rule formation until a predetermined number of instances of a digram have been encountered. In effect, this alters the first constraint above to

$p_1^{(k)}$ No pair of adjacent symbols appears more than k times in the grammar.

This modification does indeed have the effect of dramatically reducing the number of rules. Figure 4a shows the number of rules generated from a particular input file (*book1* in the Calgary corpus) for values of k from two (the normal setting for SEQUITUR) to eight. Rules reduce by a factor of five, from 26,000 to 5,000. The effect on compression, shown in Figure 4b, is interesting: there is a rather sharp minimum at a value of k=5. Note the vertical scale, however: the minimum represents a compression gain of only about 0.05 bit/char, or under 2%. Nevertheless it is interesting that improvement occurs, and also that none is apparent at k=3. (It is ironic that we briefly investigated k=3 some years ago and, finding that compression deteriorated, abandoned this line of work.) At any rate, the fact that optimal compression is achieved with one-third as many rules (around 8,000 when k=5) is encouraging, particularly for applications where rule comprehension is an issue.

Unfortunately the implications for space economy are disappointing. Reducing the number of rules yields a corresponding increase in the number of symbols in Rule *S*. The trade-off

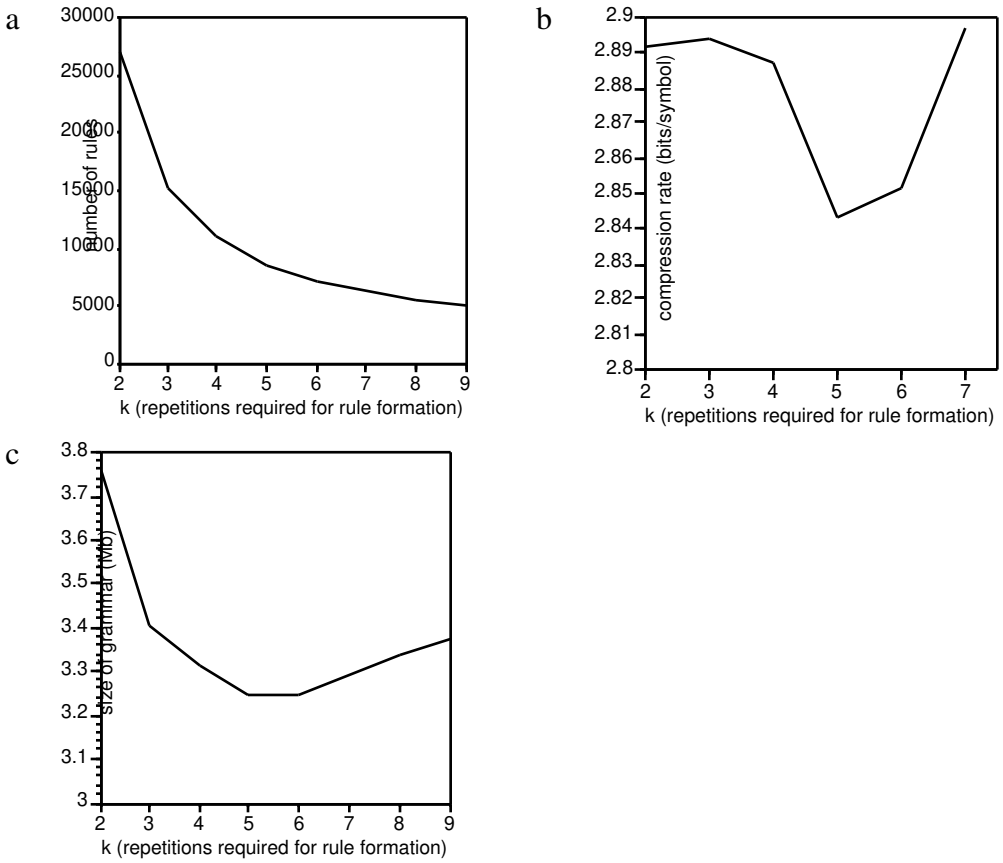

Figure 4 Effect of the number of repetitions required to form a rule. (a) number of rules in grammar (b) compression rate and (c) size of entire grammar in memory. All graphs are for *book1* from the Calgary corpus.

is positive, but not dramatically so. Figure 4c shows how the total in-memory storage required for the grammar varies with k. Although inhibiting rule formation in this way may yield grammars that are more concise and comprehensible, and—assuming that only modest values are used for the parameter k—hardly affects compression at all, we have to conclude that this is not a productive direction to explore for a bounded-memory implementation of hierarchical inference and compression.

5. Bounded-memory strategy

A more direct way of bounding the space consumed by SEQUITUR is to transmit symbols from the beginning of rule S when memory is full, and delete them. Although this sounds simple, it was not immediately apparent that it would be easy to achieve. Terminal symbols can of course be transmitted directly, and the only effect of deleting them is that they might have participated in future rules that will not now be formed. Non-terminal symbols pose more of a problem, for the decoder cannot process them without knowing the contents of the corresponding rule. Moreover, deleting symbols from rule S will merely postpone the problem of memory overflow, not solve it entirely, because the number of other rules will continue to grow linearly with the size of the input.

In fact, however, these problems are not serious at all. As often appears to be the case in hierarchy inference, the simplest solution works well. Whatever kind of symbol rule S begins with—terminal or non-terminal—transmit it. If it is a non-terminal, transmit the rule that it heads, along with any rules headed by symbols within that rule, and so on

```
main
        read symbol
        append symbol to rule S
        enforce constraints
        If memory exhausted then
                forget(first symbol in rule S)

forget(symbol)
        if symbol is terminal then
                transmit symbol
        else
                if symbol's rule has not been transmitted then
                        transmit(symbol's rule)
                else
                        transmit symbol
                        if this is the last instance of symbol's rule then
                                delete the rule and its contents
                                free up the codespace
                                transmit message 'delete rule referred to by previous symbol'

transmit(rule)
        transmit message 'make new rule'
        transmit length of rule
        for each symbol in rule
                if symbol is non-terminal and rule hasn't been transmitted then
                        transmit(symbol's rule)
                else
                        transmit symbol
```

Figure 5 Algorithm for bounded-memory implementation of hierarchy compression

recursively. If any of those rules are not used elsewhere in the grammar, delete them and free up not only the space that they occupy, but the codespace they occupy—which involves informing the decoder. Mark rules as having been transmitted: if a rule has already been transmitted, there is no need to re-transmit it. The details of this procedure are shown in the algorithm in Figure 5.

The extraordinary thing about this algorithm is that it does not affect the operation of SEQUITUR at all. SEQUITUR will continue to work with the rules that have been transmitted (provided they have not been deleted), and indeed it will continue to modify those rules. But it turns out that although it may modify the form in which a rule is expressed, it does not modify the string of terminal symbols into which it expands. Thus the fact that the rule has already been transmitted is irrelevant.

It turns out that no additional mechanism is required in the implementation to reclaim space. SEQUITUR already operates with linked lists of symbols, for other reasons; and so no extra

```
main
        while not end of file
                symbol = receive_symbol()
                if symbol = 'delete rule referred to by previous symbol' then
                        delete previous symbol's rule and its contents
                        free up codespace
                else if symbol is terminal then
                        print symbol
                else
                        print contents of symbol

receive_symbol()
        symbol = receive()
        if symbol = 'make new rule' then
                decode length
                make new rule
                do length times:
                        symbol = receive_symbol()
                        append symbol to rule
                return non-terminal referring to new rule
        else
                return symbol
```

Figure 6 Decompression algorithm

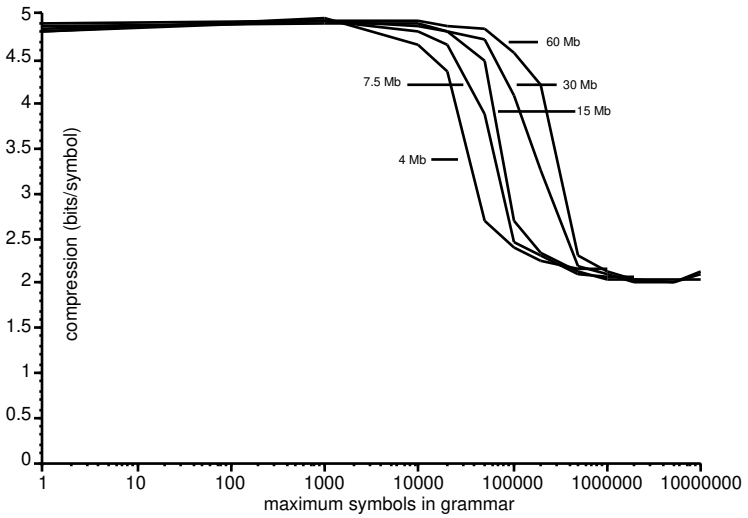

Figure 7 Compression for samples of various sizes against maximum number of symbols allowed in the grammar.

space overhead is incurred. Moreover, the memory bound can be reduced to one symbol with no effect on correctness. The resulting compression rate is just the zero-order entropy of the source.

In this scheme, the decoder is particularly simple, as it need not maintain complicated data structures for enforcing grammar constraints or modifying rules. In fact it resembles a macro decoder, and so its speed is similar to the popular LZ-based schemes (except that arithmetic coding is used for low-level transmission). The decoding algorithm is sketched in Figure 6.

Because new symbols are added to the grammar at the end of rule *S*, and symbols are transmitted from the beginning of rule *S*, there is a latency between reading and transmitting a symbol. The amount of latency depends on the memory bound. A bound of one symbol gives zero latency, but, naturally, only zero-order entropy compression. At the other extreme, unlimited memory gives a latency equal to the length of the input. As the latency is expressed in terms of input symbols, and the memory bound is expressed in terms of symbols in the grammar, the latency on a particular file depends on its compressibility; that is, how input symbols translate into grammar sizes. Higher compression rates mean longer delay if the memory bound is held constant.

Figure 7 shows how compression improves as the amount of memory available increases. Graphs are shown for four different samples of text from the Computer Science Technical Reports, ranging in size from 4 Mb to 60 Mb. The amount of memory varies along the horizontal axis from enough for one symbol only to enough for 10,000,000 symbols, that is, 200 Mb. The vertical axis shows compressed size in bits/char. The intercept at 4.8 or so bits/char corresponds to the zero-order entropy of the texts, and the small amount of splay is caused by inter-sample variation. Models with memory size up to 1000 or so symbols have negligible compression capability, except that—interestingly—they seem to provide some reduction in the variation between samples.

Virtually all improvement in compression takes place in the range between 1000 and 500,000 symbols—the range is rather squashed because of the logarithmic horizontal axis. Thus it appears that a model of a certain size is required to capture the structure of this particular text—a smaller model approximates the order-0 entropy of the text, which a larger one affords little improvement, despite a twenty-fold increase in the model size. The

point at which this transition occurs is likely to depend heavily on the nature of the input—a sequence generated by a very simple source is likely to have an earlier transition, whereas a more complex source would require a larger model.

6. Conclusions

The space requirements of phrase hierarchy inference are linear in the size of the input, as are those of other good compression techniques, and it is necessary to find a way of artificially restricting the memory consumed in order to achieve a practical compression method that will work for all input files, no matter how large. As with other compression techniques, it is not easy to find a way of limiting the size of the model, except by crude, draconian, methods like deleting the model entirely and starting again.

Two potential memory-restriction techniques have been proposed and investigated. The first involves inhibiting rule formation until a specified number of occurrences of the phrase have been encountered. This is very successful at reducing the number of rules, and has only a small—and, as it turns out, beneficial—effect on compression performance. However, although it seems to shows great potential (as yet uninvestigated) for applications that require the rules to be interpreted, it fails to fulfill the goal of reducing space requirements.

The second memory-restriction technique is quite different. It involves forcing transmission of the beginning of the sequence, and of relevant parts of the model, when memory becomes full, so that these elements can be deleted to make room for new work. Despite first appearances, this is extremely easy to implement and leads to a rather elegant scheme for bounded-memory phrase hierarchy compression that involves a particularly simple decoder. The algorithm can work with any quantity of available memory, right down to one symbol, and the effect on compression rate is just as it should be.

The simplicity of the decoder and its similarity to other macro schemes suggests an interesting possibility. Just as standards for image compression are defined by specifying only the decoder, thereby allowing for arbitrarily sophisticated encoders, it would be possible to define a standard macro decoder for text, where the details of detecting phrases are left to the designer of the encoder. The very same decoder would work for simple methods like COMPRESS through to sophisticated systems like SEQUITUR; and the amount of space available for decoding could be negotiated as part of the protocol.

Acknowledgments

This research has been supported by the New Zealand Foundation for Research, Science and Technology. We are overwhelmingly grateful to Alistair Moffat for providing both PPM compressors used for comparison, and to the New Zealand Digital Library project for supplying the samples of text used in our experiments.

References

Arnold, R. and Bell, T. (1997) "A corpus for the evaluation of lossless compression algorithms," *Proc. Data Compression Conference*, J.A. Storer and M. Cohn (Eds.), Los Alamitos, CA: IEEE Press, 201–210.

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

Harman, D.K. (1992) "Overview of the first text retrieval conference." In *Proc. TREC Text Retrieval Conference*, edited by D.K. Harman. National Institute of Standards,. 1–20.

Moffat, A. (1990) "Implementing the PPM data compression scheme." IEEE Trans on Communications 38(11): 1917-1921.

Nevill-Manning (1996) "Inferring Sequential Structure," Doctoral dissertation, University of Waikato, Hamilton, New Zealand.

Nevill-Manning, C.G. and Witten, I.H. (1997a) "Compression and explanation using hierarchical grammars," *Computer Journal, in press*.

Nevill-Manning, C.G. and Witten, I.H. (1997b) "Linear-time, incremental hierarchy inference for compression," *Proc. Data Compression Conference*, J.A. Storer and M. Cohn (Eds.), Los Alamitos, CA: IEEE Press, 3–11.

Nevill-Manning, C.G., Witten, I.H. & Maulsby, D.L. (1994), "Compression by induction of hierarchical grammars," *Proc. Data Compression Conference*, J.A. Storer and M. Cohn (Eds.), Los Alamitos, CA: IEEE Press. 244-253.

Nevill-Manning, C.G., Witten, I.H. & Olsen, D.R. (1996), "Compressing semi-structured text using hierarchical phrase identification," *Proc. Data Compression Conference*, J.A. Storer and M. Cohn (Eds.), Los Alamitos, CA: IEEE Press.

Witten, I.H., Cunningham, S.J., and Apperley, M.D. (1996) "The New Zealand Digital Library project" *New Zealand Libraries*, 48(8) 146–152.

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