Linear-time, incremental hierarchy inference for compression

Craig G. Nevill-Manning
Biochemistry, Stanford University, Stanford, CA 94305-5307
(415) 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

Data compression and learning are, in some sense, two sides of the same coin. If we paraphrase Occam's razor by saying that a small theory is better than a larger theory with the same explanatory power, we can characterize data compression as a preoccupation with *small*, and learning as a preoccupation with *better*. Nevill-Manning *et al.* (1994) presented an algorithm, since dubbed SEQUITUR, that presents both faces of the compression/learning coin. Its performance as a data compression scheme outstrips other dictionary schemes, and the structures that it learns from sequences as diverse as DNA and music are intuitively compelling.

In this paper, we present three new results that characterize SEQUITUR's computational and compression performance. First, we prove that SEQUITUR operates in time linear in n, the length of the input sequence, despite its ability to build a hierarchy as deep as $\log(n)$. Second, we show that a sequence can be compressed incrementally, improving on the non-incremental algorithm that was described by Nevill-Manning *et al.* (1994), and making on-line compression feasible. Third, we present an intriguing result that emerged during benchmarking; whereas PPMC (Moffat, 1990) outperforms SEQUITUR on most files in the Calgary corpus, SEQUITUR regains the lead when tested on multi-megabyte sequences. We make some tentative conclusions about the underlying reasons for this phenomenon, and about the nature of current compression benchmarking.

2 The SEQUITUR algorithm

The SEQUITUR algorithm is described in Nevill-Manning (1996), but it is necessary to briefly review the basic idea here. 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.

	Sequence	Grammar
a	S → abcdbc	S → aAdA A → bc

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

Figure 1 Example sequences and grammars that reproduce them
(a) a sequence with one repetition
(b) a sequence with a nested repetition

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

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.

Figure 2 summarizes the algorithm. Line 1 deals with new observations in the sequence. Lines 2 through 6 enforce the digram utility constraint. Line 3 determines whether the new digram matches an existing rule, or whether a new rule is necessary. Lines 7 and 8 enforce rule utility. Lines 2 and 7 are triggered whenever the constraints are violated. The constraints can be monitored efficiently using a technique that we do not describe here, but can be found in Nevill-Manning (1996).

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

2  Whenever a duplicate digram appears,                                               2
3      if the other occurrence is a complete rule,
4          replace the new digram with the non-terminal that heads the other digram,  3
5      otherwise
6          form a new rule and replace both digrams with the new non-terminal         4

7  Whenever a rule is used only once,
8      remove the rule, substituting its contents in place of the non-terminal        5
```

Figure 2 The SEQUITUR algorithm

2 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 O($\sqrt{n}$) previous 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.

The numbers at the right of Figure 2 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 new pair of symbols is formed. Action 3 corresponds to using an existing rule, action 4 to forming a new rule, and action 5 to removing a rule.

Figure 3 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	before	after	saving
Matching existing rule	3	...ab... $A \rightarrow ab$	...A... $A \rightarrow ab$	1
Creating new rule	4	...ab...ab...	...A...A... $A \rightarrow ab$	0
Deleting a rule	5	...A... $A \rightarrow ab$	...ab...	1

Figure 3 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,

$$n - o = a_3 + a_5 \qquad (1)$$

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, 1973). 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 digram appears. Digrams 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

yzxyzwxyzvwxy	$S \rightarrow ABwBvwxy$
	$A \rightarrow yz$
	$B \rightarrow xA$

Figure 4 A sequence that postpones rule formation by extending matches from right to left

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.

3 Incremental compression

A method of efficiently encoding the grammar produced by SEQUITUR is described in Nevill-Manning *et al.* (1994). However, this technique required the entire grammar to be formed before transmitting the compressed form. Incrementality is paramount in on-line situations such as the compression of communications traffic. The grammar is formed incrementally, so the material for transmission is available. The problem with incremental transmission is that at many points in the sequence, it is possible that a longer match will result when subsequent symbols are seen. For example, Figure 4 shows a sequence that builds up repetitions from right to left, so that rules are only formed when the final symbol in the repetition appears. It would be wasteful to transmit the *wxy* of the last repetition, in case the last symbol was *z*, and the whole subsequence could be replaced by a pointer to *wB*. Unfortunately, it is possible to construct a situation, such as the one in Figure 4, in which transmission could be postponed for an arbitrarily long period. Of course, such sequences are unlikely to occur in practice, so in most cases points will occur regularly when the sequence can be transmitted without danger of wasting symbols. Indeed, as we will show below, the longest postponement of transmission is $O(\sqrt{n})$ symbols, where n is the total number of symbols seen so far. The problem is in detecting situations in which a match can be ruled out.

Let α and β stand for the second to last and last symbols in rule S respectively. Figure 5 gives examples where either α or β can be safely transmitted. If β only occurs once in the grammar, it can be safely transmitted, because it cannot match any existing sequence in the grammar, as shown in Figure 5a. Similarly, if the only other places that β occurs are at the end of rules, so that it does not appear as the left symbol in any digrams, no digram consisting of β and the next symbol to appear can match. This situation is shown in Figure 5b. Apart from these two situations (which are exceptional), it is impossible to determine when β can be safely transmitted: whatever follows β elsewhere in the grammar might occur next.

Therefore, a decision can only be made about transmitting a symbol when it is the second-last symbol in rule S. If β is terminal, $\alpha\beta$ evidently does not occur elsewhere in

a	$S \rightarrow \ldots \alpha\beta$ ...	β can be transmitted if it only occurs once in the grammar
b	$S \rightarrow \ldots \alpha\beta$ $A \rightarrow \ldots C$ $B \rightarrow \ldots\beta$ ...	β can be transmitted if it only occurs at the end of rules
c	$S \rightarrow \ldots \alpha\beta$ $A \rightarrow \ldots \alpha C$ $C \rightarrow x\ldots$ ...	α can be transmitted if β is terminal, and α is not followed elsewhere by a symbol whose prefix is β
d	$S \rightarrow \ldots \alpha\alpha\beta$ $\alpha \rightarrow \beta x$ ...	α can be transmitted if β is terminal, and where α is followed by a symbol whose prefix is β, it cannot be extended

Figure 5 Conditions under which a symbol can be transmitted

the grammar, or it would have been replaced by a non-terminal. However, it is necessary to check that β is not the start of a non-terminal that follows α elsewhere in the grammar. This can be checked by examining the prefixes of all non-terminals that follow the other occurrences of α. If none of the non-terminals start with β, it is safe to transmit α. For example, in Figure 4, when the final *xy* has been seen, *x* is followed by *A* elsewhere in the grammar, and rule *A* begins with *y*. This indicates that *y* could be the beginning of rule *A*, and so sending *x* may eliminate the possibility of simply sending a pointer later on. This situation is shown in Figure 5c.

There is one further constraint on when a symbol can eventually become part of a longer rule. Consider the sequence $\alpha\alpha\beta$, where $\alpha \rightarrow \beta x$, as shown in Figure 5d. The rule just described indicates that the second α cannot be transmitted, because elsewhere in the grammar α is followed by a non-terminal that starts with β. However, extending α in this way would mean that rules overlap, which is prohibited. Because the second α cannot be extended, it can safely be sent.

The criterion for deciding when another chunk of rule *S* can be transmitted, where *S* ends with $\alpha\beta$, is therefore: β is terminal, and no instance of α is followed by a non-terminal whose contents start with β, unless that non-terminal is another α. Nevill-Manning (1996) shows that this is guaranteed to happen within an interval of length $O(\sqrt{n})$, where n is the length of the sequence. In the novel "Far from the madding crowd," encoding can be performed on average every 14 symbols: the minimum interval is one, which is guaranteed to be the case after the very first symbol has been received, and the maximum interval is 161.

4 Compression in the large

The performance of SEQUITUR on the Calgary Corpus (Bell *et al.*, 1990) was measured in 1994 as 2.70 bits/character (Nevill-Manning *et al.*, 1994), a little better than that of LZFG (Fiala and Green, 1989), the best macro compressor at the time the corpus was created, which rates 2.95 bits/character. Since then the performance on the corpus of the newer macro compressor *gzip* has been measured at 2.69 bits/character (see Witten *et al.*, 1994, for a description of *gzip*). Minor improvements to SEQUITUR have increased its

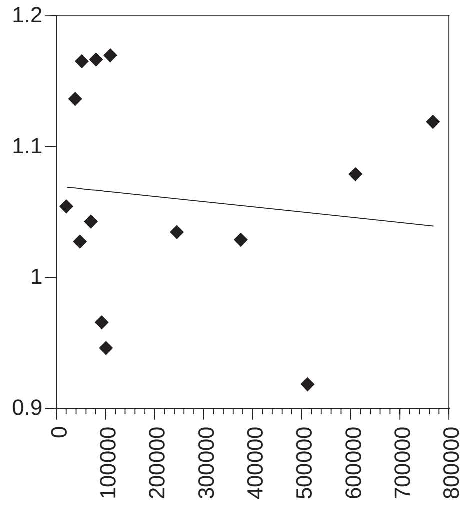

Figure 6 Ratio of SEQUITUR's compression rate to PPMC's compression rate as a function of file size, along with linear regression line

performance to 2.64 bits/character, so it is still ahead—albeit only insignificantly ahead—of macro compressors. However, PPMC yields much better performance at 2.49 bits/character on the corpus.

However, these results depend rather critically on the size of the corpus. To illustrate this, Figure 6 plots the ratio of SEQUITUR's compression to PPMC's compression, against file size. In this graph, a number greater than one means that PPMC compresses more effectively than SEQUITUR. The spread of the points is large, because of the different characteristics of the files, but fitting a least-squares error line to the points results in the downward-trending line shown. This trend implies that SEQUITUR's relative performance improves as files become longer.

To test this hypothesis, the compression on the file so far was measured at intervals of 10,000 symbols. Figure 7 shows the results for three files: the CIA world fact book (world91b.txt from the large Canterbury corpus, Arnold and Bell, 1996), the King James Bible, and an Excel spreadsheet (kennedy.xls from the large Canterbury corpus). Figure 7a shows that while PPMC's performance outstrips SEQUITUR's for the initial 400,000 symbols, PPMC plateaus, while SEQUITUR continues to improve. In this case, the file contains descriptions of 247 nations and dependent areas, all in a consistent format. SEQUITUR continues to adapt to the format and extract structure from it, whereas PPMC, presumably because of its finite-context model, fails to exhibit any significant improvement after the first 400,000 symbols. Incidentally, we use PPMC not because it is the best compression technique available, but because it is widely used as a benchmark, and we wish to demonstrate this general principle: that sequence size is an important factor in evaluating the relative merits of compression techniques. The relative performance of the two techniques on the King James' Bible is shown in Figure 7b. This is a more closely-fought race; SEQUITUR edges into the lead after 1,500,000 symbols, loses it temporarily, then regains it for good after 2,500,000 symbols. In this competition, evaluation after a mere megabyte is clearly premature.

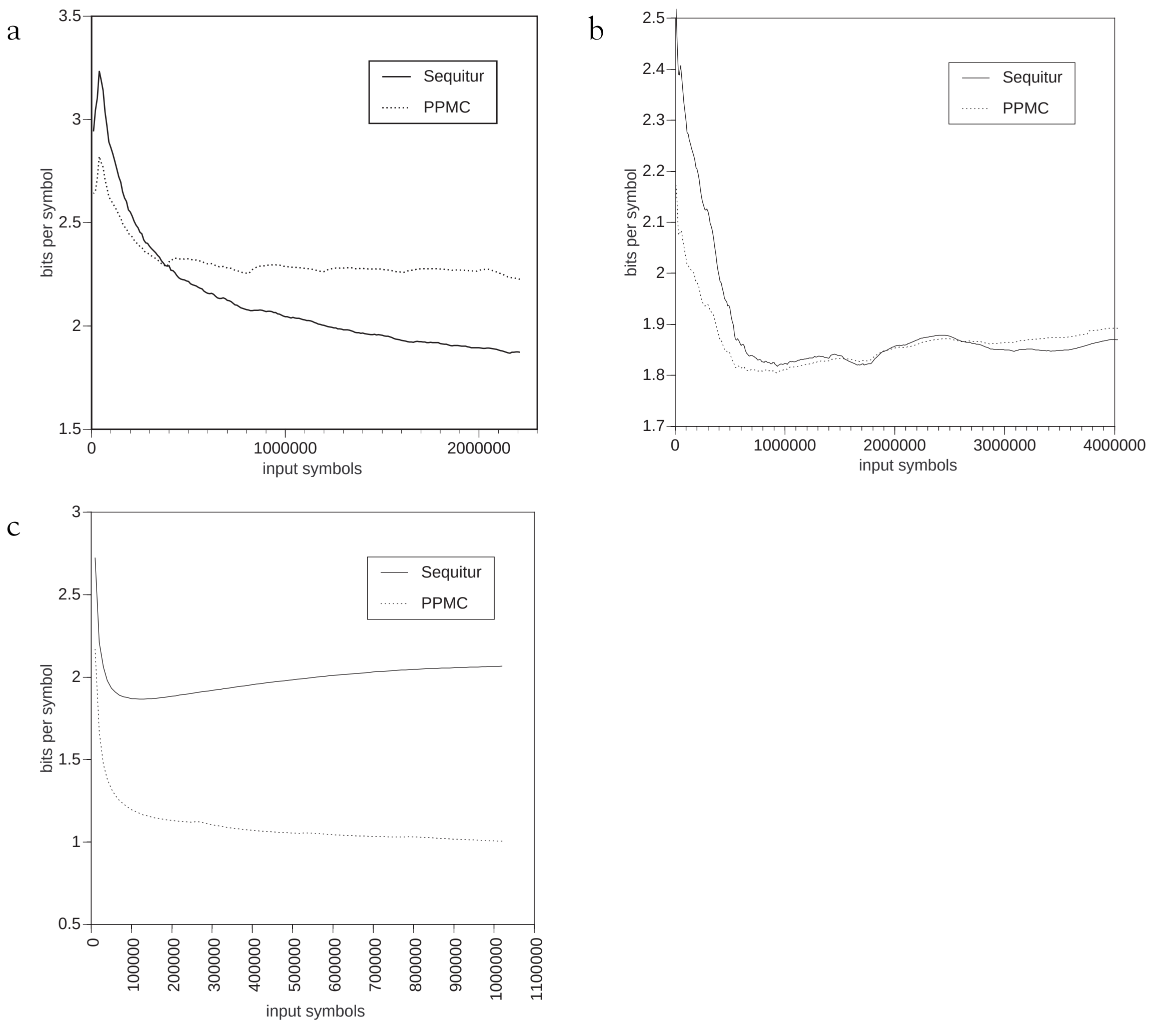

Figure 7 Compression for the file so far plotted at 10,000 symbol intervals
(a) the CIA world fact book (world91b.txt from the large Canterbury corpus)
(b) the King James Bible
(c) Excel spreadsheet (kennedy.xls from the large Canterbury corpus)

Length does not always benefit SEQUITUR: when compressing an Excel spreadsheet, SEQUITUR performs best after 130,000 symbols, thereafter worsening for the entire file while PPMC continues to improve. This comparison is shown in Figure 7c. The file contains a long sequence of binary 32-bit numbers with little exact repetition. PPMC takes advantage of the skewed distribution of bytes (ASCII NUL and TAB are over-represented) but SEQUITUR can only form phrases from repeated subsequences.

We draw two conclusions from these experiments. First, when repetitive structure is present in the sequence, SEQUITUR excels at capturing that structure, especially when it applies over long sequences. Second, using files less than a megabyte in size disadvantages SEQUITUR, and it is important that corpuses exercise data compression methods on a range of file sizes.

5 Conclusion

The SEQUITUR algorithm, introduced in Nevill-Manning *et al.* (1994), is unique in that it combines good compression performance with a comprehensible representation that actually yields insight into the structure of the input sequence. The original paper on SEQUITUR concluded that the compression obtained exceeded that of the best macro compression schemes, but fell somewhat short of that obtained by PPMC. However, it turns out that this conclusion is unduly pessimistic. Results presented in this paper indicate that for natural language texts, SEQUITUR performance begins to prove superior to PPMC's when the text size exceeds a few million characters. Moreover, for a particular natural language text containing several parts in the same consistent format, SEQUITUR continues to improve well beyond the point where PPMC reaches a asymptote.

Any scheme that works by inferring a grammar from the input runs the risk of operating very slowly, because grammatical inference normally involves extensive searching. Moreover, one normally expects the time complexity of such procedures to be much worse than linear in the size of the input. Surprisingly, we have been able to show in this paper that the time complexity of SEQUITUR is linear in the input size. Moreover, the implementation is very fast: when inferring a grammar, SEQUITUR processes input text at the rate of approximately one megabyte per minute on a SUN workstation.

Finally, although it is fundamental to SEQUITUR's operation that it infers the grammar incrementally, the previously-reported implementation had to wait until the entire grammar was in place before beginning to transmit the sequence. The present paper has shown how it is possible to transmit the information incrementally, a distinctly non-obvious feat that was previously thought to be impossible, and moreover has bounded the coding delay at the square root of the number of symbols seen so far.

Acknowledgments

Thanks to Ross Arnold and Tim Bell for providing their large Canterbury corpus.

References

Arnold, R. and Bell, T.C. (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.

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

Fiala, E.R., Greene, D.H. (1989) "Data compression with finite windows," *Communications of the ACM*, 32(4), 490–505.

Knuth, D.E. (1973) *The art of computer programming 4: searching and sorting.* Addison-Wesley.

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

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.

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

Witten, I.H., Moffat, A., and Bell, T.C. (1994) *Managing Gigabytes: compressing and indexing documents and images*. New York: Van Nostrand Reinhold.