Off-Line Dictionary-Based Compression

N. JESPER LARSSON AND ALISTAIR MOFFAT

Dictionary-based modeling is a mechanism used in many practical compression schemes. In most implementations of dictionary-based compression the encoder operates on-line, incrementally inferring its dictionary of available phrases from previous parts of the message. An alternative approach is to use the full message to infer a complete dictionary in advance, and include an explicit representation of the dictionary as part of the compressed message. In this investigation, we develop a compression scheme that is a combination of a simple but powerful phrase derivation method and a compact dictionary encoding. The scheme is highly efficient, particularly in decompression, and has characteristics that make it a favorable choice when compressed data is to be searched directly. We describe data structures and algorithms that allow our mechanism to operate in linear time and space.

Keywords**—***Dictionary-based modeling, hierarchical modeling, phrase-based compression, text compression.*

I. Introduction

Dictionary-based modeling is a mechanism used in many practical compression schemes. For example, the various members of the two Lempel–Ziv (LZ) families parse the input message into a sequence of phrases selected from a dictionary and obtain their compression by virtue of the fact that a reference to the phrase can be more compact than the phrase itself. Despite the inherent disadvantage in prediction capability compared to symbol-based methods—the conditioning context used to guide probability predictions is, in essence, reset to the empty string at the start of each phrase—the paradigm is attractive because of the elegant balance it achieves between speed, memory usage, simplicity, and compression ratio.

In most implementations of dictionary-based compression, the encoder operates *on-line*, incrementally inferring its dictionary of available phrases from previous parts of the message and adjusting its dictionary after the transmission of each phrase. Doing so allows the dictionary to be transmitted implicitly, since the decoder simultaneously makes similar adjustments to its dictionary after receiving each phrase.

An alternative approach—the topic explored in this article—is to use the full message (or a large block of it) to infer a complete dictionary in advance and include an explicit representation of the dictionary as part of the compressed message. Intuitively, the advantage of this *off-line* approach to dictionary-based compression is that with the benefit of having access to all of the message, it should be possible to optimize the choice of phrases so as to maximize compression performance. Indeed, we demonstrate that, particularly on large files, very good compression can be attained by an off-line method without compromising the fast decoding that is a distinguishing characteristic of dictionary-based techniques.

Several nontrivial sources of overhead, in terms of both computation resources required to perform the compression and bits generated into the compressed message, have to be managed as part of the off-line process. In this investigation, we develop a compression scheme *Re-Pair*, which is a combination of a simple but powerful phrase derivation method and a compact dictionary encoding. The scheme is efficient, particularly in decompression, and has characteristics that make it a favorable choice when compressed data is to be searched directly.

It should also be noted that while off-line compression involves the disadvantage of having to store a large part of the message in memory for processing, the difference between doing this and storing the growing dictionary of an on-line compressor is largely illusory. Indeed, incremental dictionary-based algorithms maintain an equally large part of the message in memory as part of the dictionary; similarly, on-line predictive symbol-based context models occupy space that may be linear in the size of that part of the message on which prediction is based.

Our scheme is off-line only while inferring the dictionary, and, during decompression, bits are read and phrases written in an interleaved manner. Moreover, during decoding, only a relatively compact representation of the dictionary is stored. Thus, during decompression, *Re-Pair* has a space advantage over both incremental dictionary-based schemes and context-based source models.

The goal of dictionary-based modeling is to derive a set of *phrases* (normally, but not always, substrings of the message

Manuscript received February 15, 2000; revised August 14, 2000. This work was supported by the Australian Research Council and the Swedish Research Council for Engineering Sciences. A preliminary presentation of the work described in this paper was made at the 1999 IEEE Data Compression Conference, Snowbird, Utah, March 1999.

N. J. Larsson was with the Department of Computer Science, Lund University, Lund, Sweden. He is now with Apptus Technologies, Lund, Sweden (e-mail: jesper.larsson@apptus.se).

A. Moffat is with the Department of Computer Science and Software Engineering, The University of Melbourne, Victoria 3010, Australia (e-mail: alistair@cs.mu.oz.au).

Publisher Item Identifier S 0018-9219(00)09986-2.

being encoded) in such a way that replacing the occurrences of these phrases in the message with references to the table of phrases decreases the length of the message. Furthermore, since we wish to transmit the phrase table as part of the compressed message, the derivation scheme used should allow a compact encoding of the phrase set. This latter requirement does not apply to incremental dictionary-based methods, and they may create their dictionary without concern for how it might be represented.

Finally, in this section, note that we use the concept *symbol* in a more general sense than is usual in compression mechanisms. We allow our algorithm to introduce new symbols; thus, symbols are not restricted to only be input items. To distinguish input symbols from created ones, we denote them as *characters*. Thus, there are k possible distinct characters but a larger variable number, k', of distinct symbols manipulated within the algorithm. It is further supposed that the compression process commences with a string of $n+1$ symbols, where n is the length of the input string and the final symbol represents end-of-message.

II. Previous Approaches

Extensive treatment of off-line substitution methods in the so-called *macro model* is given by Storer [1, chapter 5]. In addition to presenting several practical schemes, this survey also proves the intractability of *optimal* off-line substitution.

Several authors have considered the particular approach we propose in this paper, but without examination of how the mechanism might be implemented so as to be both effective and efficient. The earliest work we are aware of is that of Solomonoff [2]. Other original work is due to Wolff [3], [4]. An early exploration of phrase derivation is by Rubin [5]. He suggests several strategies, and gives experimental results. The basic idea for our scheme, as well as for some other similar approaches to dictionary derivation [6], [7], is clearly related to the *incremental encoding* schemes suggested by Rubin. However, Rubin paid relatively little attention to the issues of computational complexity and dictionary encoding techniques.

To facilitate a compact encoding of the phrase table, we employ a *hierarchical* scheme where longer phrases are encoded through references to shorter ones. This is, in some ways, similar to the LZ'78 mechanism [8], and the extension to that developed by Miller and Wegman [9]. The drawback of the aggressive phrase construction policies of LZ'78 mechanisms is that the dictionary is diluted by phrases that do not, in fact, get productively used, and compression suffers. In our proposal, every phrase is used either to directly code at least two distinct parts of the source message or as a building block of a longer phrase that is itself used twice or more.

Our derivation scheme is also loosely related to the grammar-based compression method *Sequitur* of Nevill-Manning and Witten [10]. In *Sequitur*, the input message is processed incrementally, and rules in a context-free grammar are created and then revised in a symbol-by-symbol manner, with the decoder inferring the rules from the compressed message stream. But because *Sequitur* processes the message in a left-to-right manner and maintains its two invariants (uniqueness and utility) at all times, it does not necessarily choose as grammar rules the phrases that might eventually lead to the most compact representation. Hence, *Sequitur* is best categorized as an on-line algorithm with strong links to the LZ'78 family, and the obvious question is whether a holistic approach to constructing a grammar to represent the message can yield better compression.

Our scheme also has some points in common with the compression regime described by Manber [11]. To obtain fast searching of compressed text, Manber considers a simple compression mechanism based upon character digrams and then compresses a search string using the same rules, so that the two compressed representations can be directly compared using standard pattern matching algorithms. We also replace frequent pairs but continue the process recursively until no more pairs of symbols can be reduced. Hence, the name of our program, *Re-Pair*, for *recursive pairing*.

Apostolico and Lonardi [12] present an off-line compression scheme with a phrase derivation scheme that uses a suffix tree. The suffix tree is augmented to maintain statistics for contexts without overlap, which requires super-linear $O(n \log n)$ construction time. However, although this scheme involves off-line phrase derivation, the transmission of the dictionary is performed incrementally. Thus, it is not fully off-line in the sense of our algorithm and does not offer the same potential for random access decoding of the compressed data.

Nakamura and Murashima [6] independently proposed a compression scheme that comprises the same phrase generation scheme as ours but differs in the representation of the dictionary and message encoding. As it is in the Apostolico and Lonardi scheme, the dictionary is transmitted adaptively.

Another independent work based on a character-pair phrase generation scheme is that of Cannane and Williams [7]. Their approach is specialized for processing very large files using limited primary storage. It involves scanning through the input in multiple passes during dictionary construction. Hence, their algorithm potentially requires multiplicatively more encoding time than does our single-pass encoding algorithm but has the advantage of not requiring that large inputs be split into separate blocks when memory constraints must be met.

Finally, Bentley and McIlroy [13] have also considered compression by phrase substitution but proceed by identifying *long* matches rather than *frequent* matches as we do. Theirs is also an adaptive mechanism.

III. Recursive Pairing

The phrase derivation algorithm used in *Re-Pair* consists of replacing the most frequent pair of symbols in the source message by a new symbol, reevaluating the frequencies of all of the symbol pairs with respect to the now-extended alphabet and then repeating the process until there is no pair

```
1. Identify symbols $a$ and $b$ such that $ab$ is the most
frequent pair of adjacent symbols in the message. If
no pair appears more than once, stop.
2. Introduce a new symbol $A$ and replace all
occurrences of $ab$ with $A$.
3. Repeat from step 1.
```

Fig. 1. Algorithm R, the basic pair replacement mechanism.

of adjacent symbols that occurs twice. Algorithm R in Fig. 1 captures this mechanism. Although this simple scheme is not especially well known, similar techniques have, as noted in Section II, also been described by other authors [5]–[7].

Algorithm R reduces the message to a new sequence of symbols, each of which represents either a unit character or a pair of recursively defined symbols. That is, each of these final symbols is a phrase, and the phrase set is organized in the form of hierarchical graph structure with unit symbols at the terminal nodes. A zero-order entropy code for the reduced message is the final step in the compression process, and the penultimate step is, of course, transmission of the dictionary of phrases.

We have not specified in which order pairs should be scheduled for replacement when there are several pairs of equal maximum frequency. While this does influence the outcome of the algorithm, in general it appears to be of minor importance. The current implementation resolves ties by choosing the least recently accessed pair for replacement, which avoids skewness in the hierarchy by discriminating against recently created pairs.

Fig. 2 illustrates the action of *Re-Pair* on the string `singing do wah diddy diddy dum diddy do` (with thanks to Manfred Mann for the useful lyrics). For example, the fifth new symbol created, `E`, corresponds to the phrase `.diddy`, a natural word-length repetition in the initial string. After all repeated pairs have been replaced, the reduced message is `sHHG.wahEEAumEG`, shown in the last line of the figure. Note that three of the introduced symbols—`B`, `C`, and `F`—do not explicitly appear in the final message, but must still be transmitted as part of the dictionary of phrases.

IV. Implementation

In this section, we describe a phrase derivation implementation that compresses a string of n characters in $O(n)$ time and space. Many options are available, but for brevity only a single set of choices is described here, and a number of alternatives are omitted.

A. Data Structures

Our implementation involves three data structures to access pairs in the input sequence.

- An array storing the sequence of symbol numbers—initially, the characters of the input message. Each record in the array contains three words: one that holds the symbol number and two that are used as threading pointers.
- A hash table with an entry for each *active pair*, which denotes a combination of two adjacent symbols in a pair that is still under consideration for replacement by a single symbol, and a pointer to the first appearance of each active pair in the symbol array.
- A specialized priority queue, implemented as an array of roughly $\sqrt{n}$ linked lists recording the active pairs that occur less than that number of times and one final list recording the more frequent ones.

Pair	String
	`singing.do.wah.diddy.diddy.dum.diddy.do`
`A` $\rightarrow$ `.d`	`singingAo.wahAiddyAiddyAumAiddyAo`
`B` $\rightarrow$ `dd`	`singingAo.wahAiByAiByAumAiByAo`
`C` $\rightarrow$ `Ai`	`singingAo.wahCByCByAumCByAo`
`D` $\rightarrow$ `By`	`singingAo.wahCDCDAumCDAo`
`E` $\rightarrow$ `CD`	`singingAo.wahEEAumEAo`
`F` $\rightarrow$ `in`	`sFgFgAo.wahEEAumEAo`
`G` $\rightarrow$ `Ao`	`sFgFgG.wahEEAumEG`
`H` $\rightarrow$ `Fg`	`sHHG.wahEEAumEG`

Fig. 2. Example of recursive pairing on the string *singing do wah diddy diddy dum diddy do*, with periods representing blank characters and uppercase letters representing new symbols.

Fig. 3 shows the full structure of the suggested implementation. The two pointers of each record in the sequence array are used to thread records together in a set of doubly linked lists, one for each active pair. In combination with the hash table, this gives us direct access to all positions of the sequence array that hold a given active pair.

As pairs are aggregated, some positions of the array become empty, as one of the two records combined is left vacant. To allow skipping over sequences of adjacent records in constant time, the empty space is also threaded: In the first record in a sequence of empty records, the forward thread points to the first nonempty record beyond this sequence, while in the last record, the backward thread points to the last nonempty record before the sequence.

The hash table and priority queue make use of the same set of underlying records, each of which holds a counter for the number of occurrences of that active pair and the pointer to the first location at which that pair occurs.

Note that the count of any existing active pair never increases. When the count of a pair decreases as a result of its left or right part being absorbed in a pair replacement, that pair either remains on the final priority list or is moved to a list residing at a lower index in the array. Moreover, the count of any new active pairs introduced during the replacement process cannot exceed the count of the pair being replaced. Hence, the maximum count is a monotonically decreasing entity, and locating the next most frequent active pair can be done in constant time per pairing operation. When the last list of frequent items is nonempty, it is scanned in $O(\sqrt{n})$ time to find the greatest frequency pair, and when this is identified, at least $\sqrt{n}$ pairs are replaced as a result. Once all the pairs on the final list have been dealt with, the rest of the priority

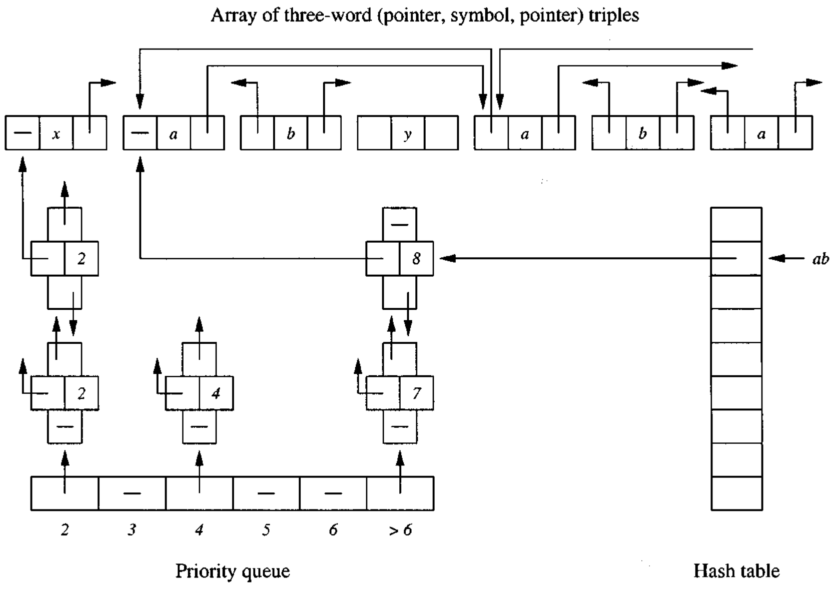


Fig. 3. Data structures during phrase construction. In the example, character pair ab is assumed to be one of two symbol pairs that appear more than six times, with the first appearance of ab being the one illustrated, in context $xaby$. Pair xa is assumed to appear twice, with the one shown being the first.

```
1. Locate the first or next sequence entry associated
   with $ab$. Identify the adjacent symbols $x$ and $y$ to
   establish the context $xaby$.
2. Decrement the counts of the adjacent pairs $xa$
   and $by$. If any of the pairs reaches a count of one,
   delete its priority queue record.
3. Replace $ab$ in the sequence, leaving $xAy$.
4. Increase the counts of the pairs $xA$ and $Ay$. This
   involves creating records for them and adding them
   to the hash table and priority queue if necessary (see
   Section IV-D).
```

Fig. 4. Algorithm P, replacing a single pair of symbols.

array is walked from its last position (roughly $\sqrt{n}$) down to position one, using $O(n)$ total time.

The priority queue is initialized in linear time by scanning the original sequence and updating counts and entry lists through hash table lookups. The total time consumed by all executions of Step 1 of Algorithm R is thus $O(n)$.

B. Pair Replacement Operation

To account for the replacement operation in Step 2 of Algorithm R, observe that since the length of the sequence decreases for each replacement, the total number of replacements is $O(n)$. Replacement of a single appearance of pair ab with a new symbol A involves the operations shown in Fig. 4, each of which must be accomplished in constant time.

Care must be taken for sequences of identical symbols, since these introduce overlapping pairs. For example, replacing aa with A in the subsequence $aaaa$ should yield two occurrences of A, not three. If the initial scanning for pairs as well as replacements is done in strict left-to-right order (which is natural), this is a simple matter of remembering the last few positions encountered in scanning or replacing pairs and excluding any pair that overlaps one that was just counted.

Operations 1 and 3 in Algorithm P can be accomplished in $O(1)$ time using the threading pointers of the sequence array.

Then, in Steps 2 and 4, entries are moved from one linked list in the priority queue to another. These movements take $O(1)$ time because each pair record includes the index of the list that contains it and the lists are doubly linked. Hence, total processing time is $O(1)$ per symbol.

C. Memory Space

Initially, each symbol in the input message is stored as a three-word triple in an array of $3n$ words. One word is used to store a symbol number and the other two are pointers threading together equal pairs of symbols in the sequence.

The priority queue structure requires an array of $\lceil\sqrt{n+1}\rceil - 1$ words, plus a record for each distinct pair that appears twice or more in the source message. Each record stores the frequency of that pair, plus a pointer to the first appearance in the sequence of that pair. Moreover, the lists are doubly linked, so two further words per record are required for the list pointers.

Prior to any pair replacements there can be at most k^2 distinct pairs in the priority lists. Thereafter, each pair replacement causes at most one new item to be added to the priority lists—both left and right combinations (xA and Ay in Fig. 3) might be new but each must occur twice before they need to be added to the priority lists, and so, in an amortized sense, at most one new active pair is created for each pair replaced. Each of these new records requires a further four words of space.

With careful attention to detail, it is possible to limit the amount of extra memory required by priority list nodes to just n words.

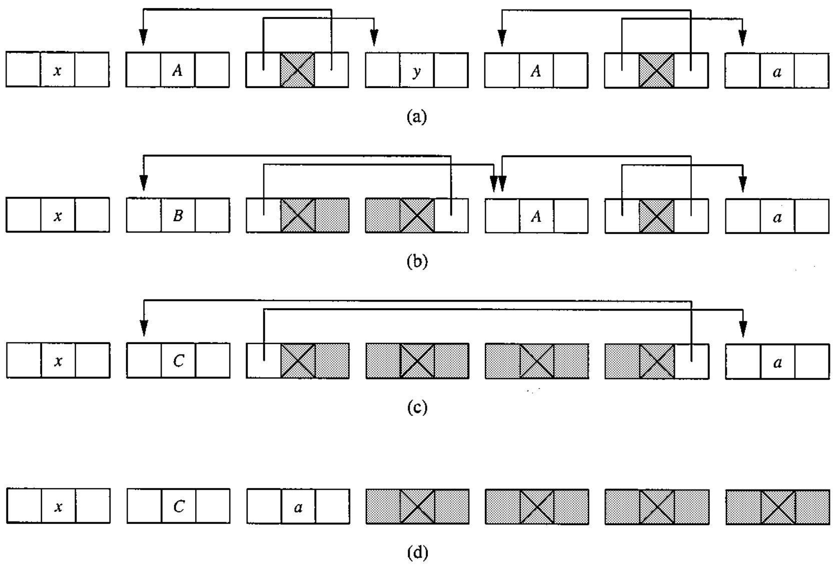


Fig. 5. Spanning the gaps between sequence records (a) after pair ab is reduced to A (in two places, see Fig. 3); (b) after Ay is reduced to B; (c) after BA is reduced to C; (d) after a compaction phase. The normal thread pointers are omitted.

Suppose that the pair reduction process is commenced with $3n$ words in the sequence array and $4k^2$ words in use for the k^2 initial priority list items. When $n/4$ pair replacements have taken place, and at most $n/4$ new priority list items (taking a total of n words of memory) have been created, the processing is temporarily suspended and a *compaction* phase (Fig. 5) commenced. The purpose of compaction is to pack all sequence records still being used into a single section of the sequence array, and free the memory occupied by empty sequence records for other use. Since $n/4$ replacements have taken place, the first compaction phase frees a block of at least $3n/4$ words.

Pair replacement then resumes. The memory freed is sufficient for the construction of another $3n/16$ priority list nodes, which in turn can only happen after a minimum of $3n/16$ further pair replacements have taken place. A second compaction, this time over only three-fourths of the length of the original $3n$-word array, then takes place and frees up $9n/16$ words, which is enough to allow the replacement process to resume and reduce another $9n/64$ pairs.

This alternating sequence of compactions and reductions continues until all pairs have been resolved, and, by construction, the ith compaction will take place (at the earliest) after $3^{i-1}n/4^i$ pair replacements and will be required to pack $3^{i-1}n/4^{i-1}$ three-word records into $3^i n/4^i$ three-word spaces, and, in doing so, frees space for

$$n \cdot \left(\frac{3^{i-1}}{4^i} - \frac{3^i}{4^{i+1}}\right) \cdot \frac{3}{4} = n \cdot \frac{3^i}{4^{i+1}}$$

list records, since each sequence record is three-fourths the size of a list record. That is, the memory freed by one compaction is exactly sufficient to accept all newly created priority lists records generated prior to the next compaction, and apart from the n words added during the first phase, no more memory needs to be allocated.

Moreover, since the time taken by each compaction operation is linear in the number of records scanned, the total cost of all compactions is

$$O\left(n \cdot \sum_{i=0}^{\infty} (3/4)^i\right) = O(n).$$

D. Pair Record Considerations

Our arrangement supposes that records are not created for new pairs unless it is clear that they will appear in the reduced sequence more than once. For this reason, when replacing ab with A, the list of occurrences for ab is scanned twice.

In the first pass, no counts are incremented. Instead, we check, for each occurrence of ab in the context $xaby$ if there is already a hash table entry for xA and Ay, respectively. If not, we need to find out if the current position is the first or second appearance of that new pair along the ab list. We allocate one special bit per hash table entry to record this. At the first appearance of xA, we set this bit in the hash table entry for xa and for Ay analogously in the entry for by. If either of these hash table entries does not exist, we know immediately that the corresponding new pair cannot occur twice and skip it. (If xa is not in the hash table, this means that it occurs only once in the sequence; therefore, xA will occur only once as well.) If we find the bit already set, we know that this is the second appearance of that pair and allocate a new priority queue entry for xA or Ay, which we link into the hash table and priority queue. The first-occurrence bit can then be reset.

In the second pass, priority queue entries for pairs that have entries in the hash table have their counts incremented. The first pass guarantees that these are the active pairs.

The hash table structure contains a pointer to the priority list record for each pair, from whence the pair itself can be identified by following the pointer in that record into the symbol sequence array. The number of entries in the hash table never exceeds $n/2$, since each entry corresponds to a pair of adjacent symbols in the message that appears at least twice, and there are at most n symbols in the message. If it is supposed that a peak loading of one-half is appropriate, the hash table must have space for n pointers.

To allow deletion to be handled, linear probing is used to resolve collisions [14, page 526]. When a record is deleted, rather than simply tag it as such in the hash table, all of the records between its location and the next empty cell are reinserted. The cost of this miniature rehashing is asymptotically less than the square of the cost of an unsuccessful search, which is $O(1)$ expected time for a given table loading. Hence, all of lookup, insertion, and deletion require $O(1)$ expected time.

A fourth data structure not yet described is the hierarchical phrase graph. Each record in this directed acyclic graph requires two words of memory, indicating the left and right components of this particular phrase, and is required at exactly the same time as the priority list item for that particular pair is being processed. Hence, the same record can be reused, and no additional space is required.

E. *Total Dictionary Space*

Summed over all data structures, the memory required is never more than $5n+4k^2+4k'+\lceil\sqrt{n+1}\rceil-1$ words, where n is the number of symbols in the source message, k is the cardinality of the source alphabet, and k' is the cardinality of the final dictionary. This is dominated by the $5n$ component except in pathological situations in which k' might be close to $n/2$.

Hence, summarizing our findings regarding the complexity of the dictionary construction algorithm, we conclude that dictionary creation through recursive pair replacement with an input of n symbols from an alphabet of size k, generating a total of k' phrases is accomplished in $O(n)$ expected time, using $5n+4k^2+4k'+\lceil\sqrt{n+1}\rceil-1$ words of primary storage.

In the decoder, two words of memory are required for each phrase in the hierarchy. As is demonstrated by the experiments in Section VIII, this is a very modest requirement.

V. Compression Effectiveness

We now consider the manner in which the phrase derivation scheme described in Section III achieves compression, when the reduced message is coded using a zero-order entropy code. Transmission of the dictionary is disregarded in this section.

A. *Symbolwise Equivalent*

To understand the structure of a dictionary-based model, it is helpful to consider the structure of its *symbolwise equivalent* model—a model that processes one character at a time with an entropy coder [15], [16].

Consider the final sequence of phrases. Suppose that there are k' distinct phrases in the sequence and n' phrases in total. Then, each occurrence of a phrase that appears ℓ times in the final sequence generates approximately $-\log(\ell/n')$ bits in the compressed message, since the final phrases are entropy coded. Let one such phrase, of length r, be described by $x_1\cdots x_r\$$, where \$ represents an *end of phrase* symbol, and let ℓ be its frequency. Let $N(c|s)$ be the number of phrases in the final sequence (of the n') that have sc as a prefix. For example, $N(\mathrm{a}|\epsilon)$ is the count of the number of phrases that commence with character "a" (ϵ is the empty string) and $N(\mathrm{b}|\mathrm{a})$ is the number of phrases that commence with "ab."

Now, consider the expression

$$-\log\frac{N(x_1|\epsilon)}{n'}-\log\frac{N(x_2|x_1)}{N(x_1|\epsilon)}-\cdots-\log\frac{N(\$|x_1\cdots x_r)}{N(x_r|x_1\cdots x_{r-1})}$$

which telescopes to

$$-\log\frac{N(\$|x_1\cdots x_r)}{n'}=-\log\frac{\ell}{n'}$$

since $N(\$|x_1\cdots x_r)=\ell$, the number of times the phrase $x_1\cdots x_r$ appears. That is, the overall code for each phrase can be interpreted as a zero-order code for the first symbol with probabilities evaluated relative to the commencing letters of the set of phrases, followed by a first-order probability for the second symbol with probabilities evaluated in the context of first letters of the set of phrases, and so on.

B. *Sources of Redundancy*

Given the symbolwise equivalent, it is unlikely that the *Re-Pair* mechanism can outperform a well-tuned context-based model, since the latter uses a high-order prediction for every character in the message, whereas, like other dictionary-based methods, *Re-Pair* essentially resets its prior context to the empty string at the start of each phrase. However, the same improvements that have been suggested for other dictionary-based models can be used if compression effectiveness is to be maximized. For example, Gutmann and Bell [17] suggest that the probability for each phrase be conditioned upon the last character of the previous phrase. (A full first-order model on phrases is, of course, pointless, since by construction the reduced message contains no repeated pairs of symbols.)

Another way in which compression effectiveness can be improved is to note that, by virtue of the way in which phrases are constructed, the final sequence contains no repeated symbol pairs, nor any pairs that constitute phrases in the dictionary. That is, if phrase pair AB has previously appeared in the final message, or if $C=AB$ is in the phrase table (for some C), then, when phrase A appears, the next phrase cannot be B. In this case, phrase B and any others that match the criteria can be *excluded* from consideration at the next coding step, and the remaining probabilities adjusted upwards, in the same way that in PPM-style methods characters can be excluded because they are known to not be possible (see Witten *et al.* [18]).

Both conditioning and exclusions are complex to implement, and if compression effectiveness (rather than compression efficiency) is the goal, then a full context-based mechanism is a better basic choice of algorithm. Our current implementation includes neither of these two improvements.

VI. Encoding the Dictionary

The hierarchical organization of the phrase table offers a natural way to encode it compactly as backward-referring pairs. This is achieved by encoding the phrases in *generations*. The first generation is the set of phrases that consist of two primitive symbols, the second generation the phrases constructed by combining first-generation objects, and so on.

Let the primitive symbols be generation 0, and the number of items up to and including generation i be k_i. Define k_i for $i < 0$ to be zero, $k = k_0$ to be the size of the input alphabet, and s_i to be the size of generation i. That is, $k_i = \sum_{j \le i} s_j$. Finally, note that each phrase in generation i can be assumed to have at least one of its components in generation $i-1$, as it could otherwise have been placed in a earlier generation. Therefore, the universe from which the phrases in generation i are drawn has size $k_{i-1}^2 - k_{i-2}^2$. Items are numbered from 0, so that the primitives have numbers 0 through $k-1$ and generation i has numbers k_{i-1} through $k_i - 1$. Each item is a pair (l, r) of integers, where l and r are the ordinal symbol numbers of the left and right components.

Given this enumeration, the task of transmitting the dictionary becomes the problem of identifying and transmitting the generations. And to transmit the ith of these generations, a subset of size $s_i = k_i - k_{i-1}$, drawn from the range $k_{i-1}^2 - k_{i-2}^2$, must be represented. This section considers three strategies for the low-level encoding of the generations: arithmetic coding with a Bernoulli model, spelling out the pairs literally, and binary interpolative encoding.

A. Bernoulli Model

If the s_i combinations that comprise the ith generation are randomly scattered over the $k_{i-1}^2 - k_{i-2}^2$ possible locations, then an arithmetic coder and a Bernoulli model will code the ith generation in

$$\log \binom{k_{i-1}^2 - k_{i-2}^2}{k_i - k_{i-1}} \text{ bits.}$$

For efficiency reasons, we do not advocate the use of arithmetic coding for this application; nevertheless, calculating the cost of doing so over all generations gives a good estimate of the underlying entropy of the dictionary and is reported as a reference point in the experimental results in Section VIII. Note that the cost of transmitting the input alphabet is disregarded in this estimate.

B. Literal Pair Enumeration

On the other hand, the most straightforward way of encoding the generations is as pairs of numbers denoting the ordinal numbers of the corresponding left and right elements, encoding $(l_1, r_1), (l_2, r_2), \cdots$ as the number sequence $l_1, r_1, l_2, r_2, \cdots$. A few optimizations to limit the range of these integers, and, thereby, reduce the required number of bits to encode them, are immediately obvious.

- *Numbers are contained in previous generations.* The maximum element when encoding generation i is k_{i-1}.
- *Pairs must have one of their elements in the immediately prior generation.* If, in generation i, the left element of a pair is less than or equal to k_{i-2}, then the right element is greater than k_{i-2}.
- *The pairs in each generation may be coded in lexicographically sorted order.* The left elements will then appear in nondecreasing order, and when the left element is the same as the previous one, the right element is strictly larger than in the previous pair.

Given these observations, the left elements of the pairs in the sequence grow slowly, with long sequences of equal elements, while the right elements are more varied. Experimentally, the most efficient encoding—of those tested—is to use a zero-origin gamma code (see Witten *et al.* [18, Section 3.3] for examples of this representation) for transmitting the input alphabet as well as the differences between successive left elements in the pair sequence, and a binary code for the corresponding right elements, tracking the remaining range (for the current left element) so that a minimal number of bits can be used.

C. Interpolative Encoding

There are many other ways of representing a subset of values over a constrained range [18, chapter 3]. When the subset is expected to be nonrandom over the range—as is the case here because, intuitively at least, some symbols are more likely to form pairs than others—the interpolative coding method of Moffat and Stuiver [19] can be used. In this method, a sorted list of integer values in a known range is represented by first coding the middle item as a binary number and then recursively transmitting the left and right sublists, both within the narrowed range established by the now-available knowledge of the value of the middle item. When the middle item lies toward one of the ends of the range, all subsequent codes in that section of the list will thus be shorter than if a normal gap-based mechanism had been used. In extreme cases of clustering, values can be transmitted in less than 1 bit each.

To actually encode the phrases with interpolative coding the two-dimensional (2-D) pairs data must be converted to single numbers. A direct approach is to enumerate the possible pairs using the same lexicographically sorted ordering as the literal pairs. Pair (l, r) in generation i is then assigned the number

$$\phi(l, r) = \begin{cases} l(k_{i-1} - k_{i-2}) + r - k_{i-2} & \text{for } l < k_{i-2}, \\ lk_{i-1} + r - k_{i-2}^2 & \text{for } l \ge k_{i-2}. \end{cases}$$

The resulting enumeration, which we call *horizontal slide*, is shown in the left half of Fig. 6.

The function $\phi(l, r)$ is not symmetric in its arguments, and any 2-D clusters in the matrix are broken up into several parts. Furthermore, the lower left part of the matrix can be expected to have the higher density, and the interpolative coding should be able to exploit this clustering. This leads to the enumeration shown in the right part of Fig. 6, which

l								
6	33	34	35	36	37	38	39	
5	26	27	28	29	30	31	32	
4	19	20	21	22	23	24	25	
3	12	13	14	15	16	17	18	
2				8	9	10	11	
1				4	5	6	7	
0				0	1	2	3	
	0	1	2	3	4	5	6	r

l								
6	7	15	23	30	35	38	39	
5	6	14	22	29	34	37	36	
4	5	13	21	28	33	32	31	
3	4	12	20	27	26	25	24	
2				19	18	17	16	
1				11	10	9	8	
0				3	2	1	0	
	0	1	2	3	4	5	6	r

Fig. 6. Pair enumeration of the horizontal and chiastic slides respectively, when $k_{i-1} = 7$ and $k_{i-2} = 3$.

we refer to as the *chiastic slide*. With this scheme, (l, r) in generation i gets number

$$\chi(l, r) = \begin{cases} 2l(k_{i-1} - k_{i-2}) \\ \quad + k_{i-1} - r - 1 & \text{for } l < k_{i-2}, \\ (2r+1)(k_{i-1} - k_{i-2}) \\ \quad + l - k_{i-2} & \text{for } r < k_{i-2}, \\ l(2k_{i-1} - l) + k_{i-1} \\ \quad - r - k_{i-1}^2 - 1 & \text{for } k_{i-2} \le l \le r, \\ r(2k_{i-1} - r - 2) + k_{i-1} \\ \quad - l - k_{i-1}^2 - 1 & \text{for } k_{i-2} \le r < l. \end{cases}$$

Calculating $\chi(l, r)$ is a costly operation if performed for each pair, and the closed form for $\chi^{-1}(x)$, for decoding, includes division as well as a square root. Fortunately, since the encoding is performed generation-wise and numbers are strictly increasing, values can be precomputed or accumulated, and incremental processing is fast.

VII. Tradeoffs

The *Re-Pair* mechanism offers a number of tradeoffs between time and space. This section briefly canvasses some of these.

A. Encoder

The description of Algorithm R in Section III stipulated that pair replacement should continue until no pair occurs twice, but this *all the way* threshold can be modified if faster encoding or a tighter bound on the dictionary space requirement is required. At the potential expense of compression effectiveness, pair replacement can be brought to a premature halt, stopping when either the dictionary reaches a predetermined maximum size or when the count of the most frequent pair reaches a certain value, and transmitting the sequence as it stands at that time. The decoding algorithm is unaffected by this change and acts in exactly the same manner as previously.

A second tradeoff is between encoding space and time. If more memory space can be allocated then encoding will be faster, since longer intervals between compaction phases will be possible. In the limit, if $4n$ words can be allocated to the priority list records ($8n$ words in total for all structures), then no compactions at all are required, even on pathological input sequences. Another possibility is to commence the reduction from symbols formed by combining adjacent character pairs into 16-bit initial symbols. Doing so would approximately halve the amount of memory required, but would reduce compression effectiveness, since all phrases would be restricted to commence and terminate on even-byte boundaries.

B. Decoder

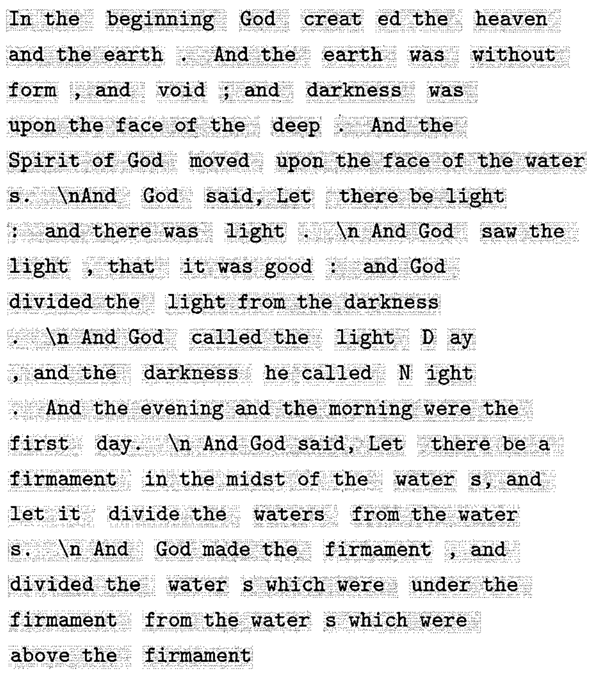
The decoder offers a particularly convenient tradeoff regarding throughput and memory usage. The simplest—and most compact—decoding data structure is to simply reproduce the phrase hierarchy, and then, for each symbol number decoded, undertake an in-order traversal of the hierarchy and output a character as each terminal node is encountered. While compact, this structure leads to relatively slow decoding. The alternative is to expand all of the phrases to form strings, and then output strings directly as symbol numbers are decoded. This form of decoding is more akin to the mechanism used by LZ'77 decoders such as *GZip*, and extremely fast decompression results. Moreover, the transition between these two extremes is adjustable. For a given amount of memory—a parameter that is set at decode time and independent of the encoding process—the most frequent (or recent) strings can be held in full, and others at least partially expanded via recursive processing of the phrase graph. One possible implementation of this tradeoff is to retain a sliding window of recently decoded text (which can be combined with output buffering) in the style of LZ'77 compression mechanisms. Phrases that reappear within the scope of the window can then be decoded by simply copying characters out of the window, rather than recursively expanding the phrase.

Table 1
Representing the Phrase Hierarchy for the *Large Canterbury Corpus* and Three Other Files, Measured in Bits Per Character and Using a 1-MB Block Size

file	*size (kB)*	*p.-ent.*	*lit.p.*	*hori.*	*chi.*
E.coli	4 529	0.19	0.21	0.14	0.12
bible.txt	3 952	0.38	0.38	0.36	0.36
world192.txt	2 415	0.42	0.42	0.39	0.38
WSJ20	20 480	0.43	0.43	0.41	0.39
Random-1	128	0.40	0.82	0.44	0.44
Random-2	128	5.33	5.93	5.23	5.02

VIII. Experimental Results

Table 1 shows the cost of representing the phrase hierarchy for the four representations discussed in Section VI. The six test files are the three files of the *Large Canterbury Corpus* or LCC (files and compression results available at http://corpus.canterbury.ac.nz/); the file *WSJ20*, which is 20 MB extracted from the *Wall Street Journal* (English text, including SGML markup); the file *Random-1*, which consists of a random sequence of 8-bit bytes; and the file *Random-2*



```
In the  beginning  God  creat ed the  heaven
and the earth  .  And the  earth  was  without
form , and  void ; and  darkness  was
upon the face of the  deep  .  And the
Spirit of God  moved  upon the face of the water
s.  \nAnd  God  said, Let  there be light
:  and there was  light  .  \n And God  saw the
light , that  it was good  :  and God
divided the  light from the darkness
.  \n And God  called the  light  D ay
, and the  darkness  he called  N ight
.  And the evening and the morning were the
first  day.  \n And God said, Let  there be a
firmament  in the midst of the  water s, and
let it  divide the  waters  from the water
s.  \n And  God made the  firmament , and
divided the  water s which were  under the
firmament  from the water s which were
above the  firmament
```

Fig. 7. Phrase representation for the first few verses of the King James Bible using a 1-MB block size.

Table 2
Phrase Statistics When Using 1-MB Block Size

file	*max. pairs*	*av. phr.*	*longest*	*av. len.*
E.coli	39 778	16 587	1 800	6.2
bible.txt	28 681	26 994	548	9.3
world192.txt	29 112	24 072	393	10.2
WSJ20	32 069	31 318	1 265	7.8
Random-1	38 878	14 471	4	1.5
Random-2	48 262	53 931	65 534	26 214

that contains a sequence of 65 536 random 8-bit bytes, followed by an exact repetition of that sequence.

In Table 1, *p.-ent.* is the phrase table entropy estimate calculated as described in Section VI-A, and the *lit.p.*, *hori.*, and *chi.* columns show, respectively, the space required for phrase tables encoded as *literal pairs* (Section VI-B), with interpolative coding (Section VI-C) using the *horizontal slide*, and with interpolative coding and the *chiastic slide*. The chiastic slide is best on all of the files except *Random-1*.

Fig. 7 shows the phrases isolated for the first few verses of the file *bible.txt*, based upon a dictionary built for a block consisting of the first 1 MB of the file. Quite remarkably, in the same 1-MB section, the longest phrase constructed (and used a total of five times) was

offered: $\backslash n$His offering was one silver charger, the weight whereof was a hundred and thirty shekels, one silver bowl of seventy shekels, after the shekel of the sanctuary; both of them full of fine flour mingled with oil for a meat offering: $\backslash n$One golden spoon of ten shekels, full of incense: $\backslash n$One young bullock, one ram, one lamb of the first year, for a burnt offering: $\backslash n$One kid of the goats for a sin offering: $\backslash n$And for a sacrifice of peace offerings, two oxen, five rams, five he goats, five lambs of the first year: this was the offering of

in which "$\backslash n$" indicates a newline character.

Table 2 gives some detailed statistics for the *Re-Pair* mechanism, again using 1-MB blocks. The *max. pairs* column shows the maximum number of pairs formed during the processing of any of the blocks, *av. phr.* is the average number of phrases constructed per block, *longest* is the length in characters of the longest phrase constructed in any of the blocks, and *av. len.* column is the average number of characters in each symbol of the reduced message. The number of pairs formed is considerably smaller than the length of the block, and the phrases isolated on the nonrandom files can be surprisingly long.

Table 3 shows the compression attained when the chiastic slide dictionary representation (column *chi.*) is combined with a semistatic minimum-redundancy coder (column *stat.*) for the reduced message [20], [21]. As expected, compression is good—better than is achieved by *GZip* [22], a LZ'77 mechanism—but not as good as the *PPMD* context-based mechanism [23], [24] reported in the last column.

Table 3
Compression Results for the *Large Canterbury Corpus* and WSJ20. The *Re-Pair* Results are for a Block Size of 4 MB, the *PPMD* Results are for a Fifth-Order Model Using 64 MB of Memory, and the *GZip* Results are with the -9 Option

file	*chi.*	*stat.*	*tot.*	*GZip*	*PPMD*
E.coli	0.11	1.98	2.09	2.24	1.99
bible.txt	0.29	1.47	1.76	2.33	1.58
world192.txt	0.31	1.31	1.62	2.33	1.52
average			1.83	2.30	1.70
WSJ20	0.29	1.68	1.98	2.91	1.72

Fig. 8 shows the compression rate attained by *Re-Pair* on file *WSJ20* as a function of block size. From the bottom of each bar, the three components for *Re-Pair* are the cost of specifying the pairs, the cost of indicating which of the symbols appears in the reduced message and transmitting the code word lengths for those that do, and the cost of using those code words to represent the reduced message. The rightmost bar in the figure shows the result of compressing the whole of *WSJ20* as a single block. To allow comparison, the solid line indicates the compression obtained by a *PPMD* implementation, in each case constrained to use approximately the same amount of memory space as the corresponding *Re-Pair* experiment, using the model order that gives the best compression effectiveness for that amount of memory. For example, when *Re-Pair* uses a block size of 64 kB, *PPMD* was permitted to use no more than 1 MB of data space, and third-order predictions gave the best compression. A seventh-order model gave the best compression when *PPMD* was allowed 256 MB of memory. The latter corresponds to the memory required by *Re-Pair* when the whole of WSJ20 (i.e., 20 MB) was handled as a single block. As was anticipated by the discussion in Section V-B, the compression achieved by *Re-Pair* approaches, but does not exceed, that of *PPMD*.

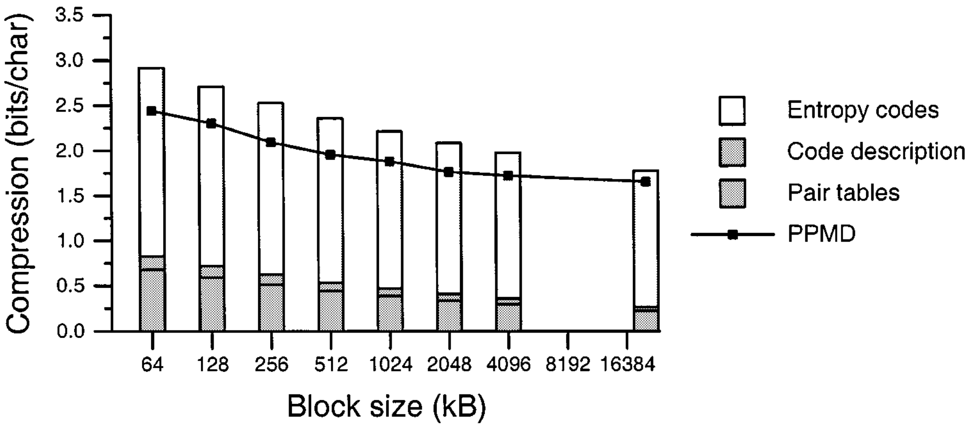


Fig. 8. Compression attained as a function of block size for file *WSJ20*. The solid line shows the compression obtained by *PPMD* using a comparable amount of memory for data structures.

Table 4
Experimental Time (CPU Seconds) for Encoding and Decoding 20 MB of Text (File WSJ20) Using Local Disks on an Otherwise Idle 450-MHz Pentium III

method	*encoding*	*decoding*
GZip −9	26.0	1.5
PPMD	41.2	41.7
Re-Pair	135.7	3.1

Execution time of the three compression processes on the file *WSJ20* are shown in Table 4. These times are CPU seconds for encoding and decoding the file on a 450-MHz Intel Pentium III with 512 MB of RAM and 512 kB cache. For *Re-Pair*, the times listed include the cost of the entropy coder, which is a fast Huffman-style mechanism based on the techniques of Moffat and Turpin [20], [21]. Entropy coding requires 2.5 s for encoding and 0.5 s for decoding. A *Re-Pair* block size of 4 MB was used, and the decoder made use of a sliding window of 512 kB of decoded text to minimize the number of phrases expanded more than once. The *Re-Pair* encoder required about 60 MB in total for all data structures, or about $3.75n$ words. The decoding memory space requirement was much less, at about 2 MB. The *PPMD* implementation uses a fifth-order context and 64 MB of memory in both encoder and decoder. *GZip* was used with the -9 option.

IX. Future Work

A number of areas for further development remain. One track that is worth following is exploiting the fact that our dictionary is static so as to yield efficient access operations when searching in the compressed data.

We are also interested in exploring the possible correlations between blocks of text in a large file. It may be that the dictionary used in one block can most economically be transmitted as a variant of the dictionary used in the previous block, rather than encoded from scratch. Another possibility is for the phrase tables for a set of blocks to be merged, and shared between them. Each block would retain the same entropy codes (the top component of the bars in Fig. 8), and it might be that the increased cost of the entropy code description (the middle component of the bars) for each block of the file is sufficiently small as to result in a net saving.

Finally, following the lead shown by *Sequitur*, it will be interesting to assess the extent to which the dictionary construction technique used in *Re-Pair* generates a sensible structural decomposition for complex sequences of a non-textual nature.

Acknowledgment

The authors would like to thank A. Turpin and R. Wan (University of Melbourne) for contributions to the implementation of *Re-Pair*. K. Sadakane (Tohoku University), J. Storer (Brandeis University), and G. Wolff (University of Wales) drew the authors' attention to early work in this area.

References

[1] J. A. Storer, *Data Compression: Methods and Theory*. Rockville, MD: Computer Science, 1988.
[2] R. J. Solomonoff, "A formal theory of inductive inference—Parts I and II," *Inform. Control*, vol. 7, pp. 1–22 and 224–254, 1964.
[3] J. G. Wolff, "An algorithm for the segmentation of an artificial language analogue," *Brit. J. Psych.*, vol. 66, no. 1, pp. 79–90, 1975.
[4] ——, "Recoding of natural language for economy of transmission or storage," *Comput. J.*, vol. 21, pp. 42–44, 1978.
[5] F. Rubin, "Experiments in text compression," *Commun. ACM*, vol. 19, pp. 617–623, Nov. 1976.
[6] H. Nakamura and S. Murashima, "Data compression by concatenations of symbol pairs," in *Proc. IEEE Int. Symp. Information Theory and its Applications*, Victoria, BC, Canada, Sept. 1996, pp. 496–499.
[7] A. Cannane and H. E. Williams, "General-purpose compression for efficient retrieval," Dept. Comput. Sci., RMIT Univ., Melbourne, Australia, Tech. Rep. TR-99-6, June 1999.
[8] J. Ziv and A. Lempel, "Compression of individual sequences via variable-rate coding," *IEEE Trans. Inform. Theory*, vol. IT-24, pp. 530–536, Sept. 1978.
[9] V. S. Miller and M. N. Wegman, "Variations on a theme by Ziv and Lempel," in *Combinatorial Algorithms on Words*. ser. NATO ASI, A. Apostolico and Z. Galil, Eds. New York: Springer-Verlag, 1985, vol. F 12, pp. 131–140.
[10] C. G. Nevill-Manning and I. H. Witten, "Compression and explanation using hierarchical grammars," *Comput. J.*, vol. 40, no. 2/3, pp. 103–116, 1997.
[11] U. Manber, "'A text compression scheme that allows fast searching directly in the compressed file," *ACM Trans. Inform. Syst.*, vol. 15, pp. 124–136, Apr. 1997.
[12] A. Apostolico and S. Lonardi, "Greedy off-line textual substitution," in *Proc. IEEE Data Compression Conf.*, Mar.–Apr. 1998, pp. 119–128.

[13] J. L. Bentley and M. D. McIlroy, “Data compression using long common strings,” in *Proc. IEEE Data Compression Conf.*, 1999, pp. 287–295.
[14] D. E. Knuth, *Sorting and Searching*, 2nd ed. Reading, MA: Addison-Wesley, 1998, vol. 3, of The Art of Computer Programming.
[15] T. C. Bell and I. H. Witten, “The relationship between greedy parsing and symbolwise text compression,” *J. ACM*, vol. 41, pp. 708–724, July 1994.
[16] G. G. Langdon, “A note on the Ziv–Lempel model for compressing individual sequences,” *IEEE Trans. Inform. Theory*, vol. IT-29, pp. 284–287, Mar. 1983.
[17] P. C. Gutmann and T. C. Bell, “A hybrid approach to text compression,” in *Proc. IEEE Data Compression Conf.*, Mar. 1994, pp. 225–233.
[18] I. H. Witten, A. Moffat, and T. C. Bell, *Managing Gigabytes: Compressing and Indexing Documents and Images*, 2nd ed. San Mateo, CA: Morgan Kaufmann, 1999.
[19] A. Moffat and L. Stuiver, “Binary interpolative coding for effective index compression,” *Inform. Retrieval*, vol. 3, pp. 25–47, July 2000.
[20] A. Moffat and A. Turpin, “On the implementation of minimum-redundancy prefix codes,” *IEEE Trans. Commun.*, vol. 45, pp. 1200–1207, Oct. 1997.
[21] A. Turpin and A. Moffat, “Housekeeping for prefix coding,” *IEEE Trans. Commun.*, vol. 48, pp. 622–648, Apr. 2000.
[22] J. l. Gailly, “Gzip program and documentation,”, Source code available from ftp://prep.ai.mit.edu/pub/gnu/gzip-*.tar, 1993.
[23] A. Moffat, “Implementing the PPM data compression scheme,” *IEEE Trans. Commun.*, vol. 38, pp. 1917–1921, Nov. 1990.
[24] P. G. Howard, “The design and analysis of efficient lossless data compression systems,” Ph.D. dissertation, Dept. Comput. Sci., Brown Univ., Providence, RI, June 1993.

N. Jesper Larsson received the Ph.D. degree from Lund University, Lund, Sweden, in 1999. His doctoral dissertation, “Structures of String Matching and Data Compression,” describes efficient algorithms and data structures for string processing, including several schemes contributing to sequential data compression.

He is currently with Apptus Technologies, Lund, a company that specializes in the efficient management of massive amounts of data.

Alistair Moffat received the Ph.D. degree from the University of Canterbury, New Zealand, in 1986.

Since then, he has been a Member of Academic Staff at the University of Melbourne, Australia, and is currently an Associate Professor. His research interests include text and image compression, techniques for indexing and accessing large text databases, and algorithms for sorting and searching. He is a coauthor of the book *Managing Gigabytes: Compressing and Indexing Documents and Images*, (San Mateo, CA: Morgan Kaufmann, 1999, 2nd ed.) and has written more than 90 refereed papers.

Dr. Moffat is a member of the ACM and the IEEE Computer Society.