# Constructing Word-Based Text Compression Algorithms

**R. Nigel Horspool**
Department of Computer Science
University of Victoria, P.O. Box 3055
Victoria, B.C., Canada V8W 3P6
nigelh@csr.uvic.ca

**Gordon V. Cormack**
Department of Computer Science
University of Waterloo
Waterloo, Ont., Canada N2L 3G1
gvcormack@waterloo.edu

**Abstract**

Text compression algorithms are normally defined in terms of a source alphabet $\Sigma$ of 8-bit ASCII codes. We consider choosing $\Sigma$ to be an alphabet whose symbols are the words of English or, in general, alternate maximal strings of alphanumeric characters and non-alphanumeric characters. The compression algorithm would be able to take advantage of longer-range correlations between words and thus achieve better compression. The large size of $\Sigma$ leads to some implementation problems, but these are overcome to construct word-based LZW, word-based Adaptive Huffman, and word-based Context Modelling compression algorithms.

## 1 Introduction

Most text compression algorithms perform compression at the character level. If the algorithm is adaptive (as, for example, with any of the Ziv-Lempel methods), the algorithm slowly learns correlations between adjacent pairs of characters, then triples, quadruples and so on. The algorithm rarely has a chance to take advantage of longer range correlations before either the end of input is reached or the tables maintained by the algorithm are filled to capacity. If text compression algorithms were to use larger units than single characters as the basic storage element, they would be able to take advantage of the longer range correlations and, perhaps, achieve better compression performance. Faster compression may also be possible by working with larger units.

In this paper, we explore the use of *words* as the basic unit. When the source file is an English-language document, say, we have no difficulty in recognizing a word as consisting of a sequence of consecutive letters. Each word is separated from the next by space and/or punctuation characters. Following the same approach as Bentley et al. [2], we generalize slightly by considering a text file to consist of alternating alphanumeric-strings and punctuation-strings, where a word-string is a maximal sequence of alphanumeric characters and a punctuation-string is a maximal sequence of non-alphanumeric characters. We use the generic name *word* to refer to either an alphanumeric string or a punctuation string. The generalization permits us to decompose all kinds of text files — program source code, input to a word-processor, etc. — into sequences of words.

Existing compression algorithms that consider the input as a sequence of words are ad hoc in nature. The scheme described by Bentley et al. [2] maintains a list of words sorted into least-recently used order. A word is encoded by its position in this dynamically changing list. Words near the front of the list tend to have shorter codes than those near the end and, assuming words in frequent use stay near the front of the list, compression is achieved. The general approach is called Move-To-Front or MTF in [1]. Generalizations of the scheme that use other heuristics than MTF to manage the list appear in [5] and [6].

A less ad hoc approach would be to consider words as forming the symbols of an alphabet. Such an alphabet can, in principle, be used as the basis of any existing compression algorithm. For example, LZW (aka the UNIX *compress* command) [7] could work by encoding sequences of words instead of sequences of characters. If particular sequences tend to recur in the source text, compression would be achieved.

However, we need to overcome a major problem with word-based compression algorithms. The number of distinct words that the compression algorithm has to cope with is, for all practical purposes, unbounded. Thus it makes no sense to implement an algorithm that requires a pre-determined finite alphabet. To use LZW as an example again, we cannot initialize the LZW string table with all sequences of length one, as required in the usual implementations of LZW. Instead, we have to modify the algorithms so that they either pre-determine the set of words used in the source input (an inherently two-pass strategy) or they dynamically expand the source alphabet as each new word occurs. We, of course, advocate single-pass strategies as being more useful for practical applications.

The following sections of this paper will consider the problem of generalizing a compression algorithm to be word-based, then particular word-based algorithms will be described, and finally some experimental results will be reported.

# 2 Using Word-Based Alphabets

Following the scheme of [2], we can decompose textual input into a sequence of words, where alternate words are composed from alphanumeric characters and from non-alphanumeric characters. For example, a line of a Pascal source code file that reads

```
xCoord2 := xCoord2 + delta;
```

would be decomposed into the following elements, where spaces are made visible and the line-feed character at the end of line is shown as a C-style character constant `'\n'`.

| "____" | "xCoord2" | "_:=_" | "xCoord2" | "_+_" | "delta" | ";\n" |
|---|---|---|---|---|---|---|
| *punct* | *alpha* | *punct* | *alpha* | *punct* | *alpha* | *punct* |

It is easy to transform an existing compression algorithm to operate on the alphabet of words if an extra pass over the source data is permitted. An initial pass enters the words (both alphanumeric and non-alphanumeric) into a dictionary. Once the pass is complete, we know all the symbols of the word alphabet and it should now be possible to construct a version of the compression algorithm that uses this new alphabet. Of course, the dictionary (or, more likely, a compressed form of the dictionary) must be transmitted with the output from the compression algorithm.

For the majority of applications, two passes over the source text are undesirable. An adaptive scheme that dynamically expands the dictionary as new words are encountered is preferable. A general mechanism for handling a new word involves the use of an escape code. When the compression algorithm hits a new word, it can output an escape code followed by some representation of the text of the new word. Then it can add the new word to the next available slot in the dictionary and continue as though the word had been present in the dictionary all the time.

This general escape mechanism, however, does not extract the maximum amount of redundancy from the compression algorithm. As the following examples show, it is possible to integrate the escape mechanism into the compression algorithm more tightly and do better. For one thing, we should be able to take advantage of the fact the the alphanumeric and non-alphanumeric words strictly alternate. Thus, we should use two word-based source alphabets $\Sigma_A$, the alphabet of alphanumeric words, and $\Sigma_P$, the alphabet of punctuation strings. If the symbols from the two alphabets are identified by symbol numbers, the numberings need not be disjoint, as the decoding algorithm should always know by context which source alphabet to expect. A second way in which the escape mechanism can be better integrated is by making either an occurrence of the escape code or the symbol that corresponds to a new word implicit. Such integration may increase the complexity of the implementation somewhat.

# 3 Some Word-Based Algorithms

## 3.1 Word-Based Adaptive Huffman Coding

Adaptive Huffman coding is the basis of the UNIX *compact* program. The compression program maintains a count of how many times each symbol has occurred so far in the source text. To encode the next symbol, the symbol counts are used as estimates of the relative probabilities of the symbols and a table of Huffman codes based on these frequencies is constructed. The Huffman code in the table is used to encode the next symbol. (A minor detail that needs to be taken into account is that symbols in the alphabet that have not yet occurred in the source text must be assigned a non-zero probability estimate.) The decoding algorithm can re-create the same set of symbol frequencies from its de-compressed text and use the table to re-construct the same table of Huffman codes. Thus it can uniquely decode one symbol, update the frequency count of that symbol, update its table of Huffman codes and then decode the next symbol, and so on.

Algorithms exist for efficiently updating the Huffman codes when small incremental changes to the probability estimates are made (as is the case here) [3], [4]. In spite of the widespread use of these algorithms in implementations, Adaptive Huffman coding is not renowned for its speed (nor for its compression performance).

The overall structure of a word-based Adaptive Huffman algorithm may take the form shown in Figure 1. The algorithm uses two tables of frequencies, `AFreq` and `PFreq`, and two tables of Huffman codes, `AHuffman` and `PHuffman`, for the two different alphabets $\Sigma_A$ and $\Sigma_P$. Details concerning the initialization and the proper termination of the algorithm at the end of input are omitted for brevity.

**Figure 1 Word-Based Adaptive Huffman Algorithm**

```
repeat
    read one alphanumeric word, AW;
    if AW ∉ Σ_A then
        output AHuffman[Escape];
        output text of AW;
        Σ_A := Σ_A ∪ {AW};
        AFreq[AW] := 1;
        AFreq[Escape] := AFreq[Escape] + 1;
    else
        output AHuffman[AW];
        AFreq[AW] := AFreq[AW] + 1;
    endif
    AHuffman := recomputed table of Huffman codes constructed
        from the frequency table, AFreq;
    read one non-alphanumeric word, PW;
    if PW ∉ Σ_P then
        ...
        ...    (* continuing similarly to the above *)
        ...
until end of input is reached;
```

In what form should the text of new words be transmitted? New words occur in short input files with a relatively high frequency and efficient encoding of them is highly desirable. To be consistent with the top-level word-based compression strategy, we propose that adaptive Huffman coding be used for the individual characters of the words. Since an algorithm for updating the Huffman codes must already be available for the two source alphabets $\Sigma_A$ and $\Sigma_P$, it would not impose a burden on the implementer to use it for four different source alphabets. (Four because the decoder knows whether a new word is going to be composed from alphanumeric characters or non-alphanumeric characters, and thus the two alphabets may be encoded separately.)

The compression performance of word-based Adaptive Huffman coding is excellent, as the experimental data at the end of this paper shows. The execution speed is not so good however. As the $\Sigma_W$ and $\Sigma_P$ alphabets grow in size, the time required to update the tables of Huffman codes slowly and inexorably increases. The average time complexity of the update algorithms appears to be O(n), where n is the size of the alphabet; the worst-case time complexity is O(n log n). If word-based Adaptive Huffman coding were to become practical , some technique would be needed to prune infrequently used symbols from the $\Sigma_W$ and $\Sigma_P$ alphabets. (Certainly we should delete words whose Huffman codes become so long that fewer bits would be needed to re-transmit the word as a new word.) Several pruning strategies based on an analogy with page replacement algorithms in virtual memory systems are suggested in [6].

## 3.2 Word-Based LZW

The LZW (Lempel-Ziv-Welch) compression algorithm [7] is the basis of the UNIX *compress* program and of the compression strategies implemented in many commercial prod-

ucts, both hardware and software. Its main virtue is speed, while simultaneously achieving good, but not spectacular, compression performance.

LZW is easy to explain. The algorithm maintains a string table that associates a unique integer with each string. To encode the next segment of source text, the compression algorithm reads the longest possible sequence of characters that comprises a string in the table. It outputs the number associated with the string, using a simple binary numbering system. If the string that was read was $\omega$ and K is the following character in the source text, the new string $\omega K$ is added to the string table and the next unused number is associated with the string. The compression algorithm then continues, reading input characters starting with K looking for the longest string that is contained in the table.

The string table is initialized with all strings of length one. This guarantees that at least one character can be read from the input and matched against a string in the table. The strategy for adding new strings guarantees that if a string $\omega$ is in the table, then all prefixes of $\omega$ must also be present in the table. This property simplifies the task of matching a maximal length sequence of input characters against the strings in the table, and also permits the table to be implemented by an efficient data structure (such as a trie). As stated above, the encoding of string numbers is simple (and is another reason for the speed of LZW implementations). If, at some moment, the table holds N strings (and these will be numbered 0 through N-1), a binary number comprised of log(N+1) bits is used to encode the next string number to be output.[1]

A word-based LZW implementation cannot initialize the string table with all strings of length one. This is because the $\Sigma_W$ and $\Sigma_P$ alphabets may be very large and the symbols are not known in advance (unless a pre-pass over the text source is performed). Thus, we will again advocate the use of an escape mechanism.

The word-based LZW algorithm builds up two kinds of strings of symbols. All strings will consist of alternate symbols from $\Sigma_W$ and $\Sigma_P$, but we can segregate strings whose initial symbol is an element of $\Sigma_W$ in a separate table from those strings whose initial symbol is an element of $\Sigma_P$. Numbering of strings in the two tables need not be disjoint because the decoding algorithm can always deduce whether the next string it receives should begin with a $\Sigma_W$ or a $\Sigma_P$ symbol.

The structure of the main body of the algorithm has the form shown in Figure 2. We use $\lambda$ to denote the empty string and <<$\alpha$,b>> to denote a string constructed by appending the symbol *b* to the string $\alpha$. The two tables of strings are named `ATable` and `PTable`. A string numbering scheme that reserves a code (presumably 0) for Escape must be used. The empty string $\lambda$ may be implemented by the same number because an encoding of $\lambda$ is never output.

Again, we must decide how the characters of a new word should be encoded. As before, the compression performance is compromised for small files unless a reasonably efficient coding scheme is used for the characters. Again, it is possible to apply the same

---

1. The original LZW algorithm, as described in [7], proposes a maximum table size of 4096 strings and that 12-bit numbers be used regardless of table occupancy.

**Figure 2 Word-Based LZW Compression Algorithm**

```
ω := λ;
repeat
    if currentIsAlph then read next alphanumeric word X;
    else read next non-alphanumeric word X; endif
    if X is a new word then
        if ω  λ then output string number of ω endif
        output Escape;  output text of X;
        ω := λ;
        startIsAlph := not currentIsAlph;
    else
        if startIsAlph then search ATable for string <<ω,X>>;
        else search PTable for string <<ω,X>>; endif
        if <<ω,X>> was found then
            ω := <<ω,X>>;
        else
            output string number of ω;
            add <<ω,X>> to the appropriate table, ATable or PTable;
            startIsAlph := currentIsAlph;
            ω := <<λ,X>>;
        endif
    endif
    currentIsAlph := not currentIsAlph;
until end of input is reached;
```

basis algorithm for the lower-level encoding as at the higher word-based coding level. There is a caveat however. LZW encodes sequences of characters. We must treat the end of each new word as being equivalent to the end-of-file, otherwise the encoding of parts of two consecutive new words would have to be combined. (We expect correlations between characters at the end of one new word and characters at the beginning of the next new word to be weak.)

Compression performance is again excellent. Speed degradation as the two word-string tables fill up is hardly noticeable when they are implemented as very large hash tables. However, a pruning strategy to prevent the tables from reaching too high an occupancy to permit fast look-ups (greater than 80% occupancy, say) is again desirable.

## 3.3 Word-Based First-Order Context Compression

The Adaptive Huffman coding algorithm, described above, uses a zero-order Markov model to predict properties of the source text. Much better compression may be achieved if a higher-order Markov model is used. Higher-order models form the basis of the PPM (prediction by partial match) family of compression algorithms [1], for example.

When the symbols are English words, we can expect correlations between successive symbols. For example, the word ‘the’ would have a high probability of being followed by a word that is a noun or adjective but a low probability of being followed by another article or a verb. An algorithm based on a first-order model could maintain statistics on the observed frequencies of word pairs in the text compressed so far. Then given

that the previous word to have been encoded is $W_i$, we can use the observed frequencies to estimate the conditional probabilities $P(W_j|W_i)$ for the next word. These probabilities may passed to an arithmetic coding subroutine for transmitting the next word.

If the compression algorithm is to be targeted to compression of natural language, we should treat the two alphabets differently. There should be strong correlations between successive alphanumeric words but, presumably, weak correlations between successive punctuation strings. Also, we can expect correlations between the punctuation and the alphanumeric words – for example, a punctuation string that contains a period would signify the end of a sentence and would therefore predict that the next alphanumeric word would begin with an upper-case letter (and would be quite likely to be an article). The algorithm structure shown in Figure 3 assumes only correlations between successive symbols from $\Sigma_A$. In the algorithm, the variable `PrevW` represents the previous alphanumeric word to have been transmitted. The arithmetic coding algorithm uses the frequency count `Freq[PrevW,X]` to estimate the relative probability for each word X in $\Sigma_A$.

The natural choice for encoding the text of punctuation strings and words is arithmetic coding applied to individual characters. The probability estimates needed as input to the algorithm can be based on the frequency of occurrence of each character. Compression performance is improved if probabilities obtained from the first-order model are blended with probabilities from the 0th-order model when the number of observations is too low to make reliable first-order predictions. Blending the two models is a scheme used in PPMC [1], for example. Blended probabilities are used to obtain the results reported in Table 1.

Another way to improve the significance of the statistics and also a way of reducing the volume of data that must be retained is to merge statistics for similar words. An obvious way to group words into a small number of classes is by their parts of speech in the English language. This gives rise to our final word-based algorithm. It is described in the following section.

**Figure 3 Arithmetic Coding with Word-Based First-Order Context Model**

```
PrevW := Escape;
repeat
    read next punctuation word AP and output text of AP;
    read next alphanumeric word, AW.
    if AW ∉ ΣA then
        output Escape by arithmetic coding; output text of AW;
        ΣA := ΣA ∪ {AW};
        Freq[PrevW,AW] := 1;
        Freq[PrevW,Escape] := Freq[PrevW,Escape] + 1;
    else
        output AW by arithmetic coding;
        Freq[PrevW,AW] := Freq[PrevW,AW] + 1;
    endif
    PrevW := AW;
until end of input;
```

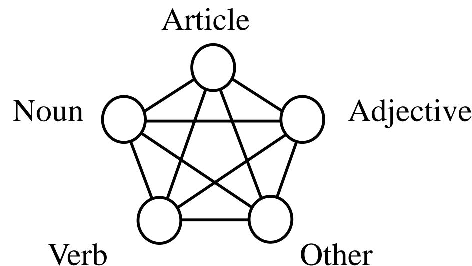
## 3.4 First-Order Context Modelling by Part-of-Speech

If we assume that the source text contains natural language, the sequence of words in the source should obey the rules of grammar for that language. For example, one possible form for an English language sentence is:

Subject Verb Object

where the *Subject* and the *Object* may be constructed as an *Article* followed by a *Noun*. The rules of grammar strongly influence the probabilities of certain words appearing in certain contexts. For example, after the word ‘the’ (an article), we would not expect to find another occurrence of an article or a verb, but we could find a noun or an adjective there. (However, the existence of a music group named ‘The The’ reminds us that we should not assume a zero probability for any combination of words, no matter how strange it seems.)

We therefore propose that a state-based model [1] be used for modelling the source text and be used for generating the prediction probabilities used by an arithmetic coding subroutine. For simplicity, we assume that there are just five parts of speech named *Article*, *Noun*, *Adjective*, *Verb* and *Other*. (We include pronouns such as ‘she’ and ‘him’ in the *Noun* category.) This yields a state diagram like the following:




Each state provides a set of probability estimates for the symbols in $\Sigma_A$. If the current state is *Article* and the next symbol is the word ‘funny’, we would use the probability estimates associated with the *Article* state to encode the word ‘funny’. Then we would make a transition to the *Adjective* state, since that is the part of speecf for the word ‘funny’.

Implementation of this state-based model requires that we know or can determine the parts of speech for all the symbols in $\Sigma_A$. This requires that the compression algorithm be supplied with an initial vocabulary that specifies the appropriate part of speech for words in the vocabulary. (Presumably some heuristics, such as deciding that a word ending with the letters ‘ly’ is probably an adverb could also be useful.) But it is unreasonable to require every word that might appear in any document to be included in the initial vocabulary. Therefore, we need a mechanism for inferring an appropriate part of speech for a new word after an occurrence in the text.

The first time a word occurs, and the word is not contained in the vocabulary, we should assign that word the part of speech that is most likely to follow the part of speech for the preceding word. Subsequently, we keep statistics on how well that word fits into its assigned part of speech. For example, if the new word is W, we keep track of how efficiently those words that immediately follow subsequent occurrences of W would be encoded if W were assigned to the Noun class, to the Adjective class, and so on. If better compression would have been achieved with W in a different class, W is dynamically re-

**Figure 4 Arithmetic Coding with State-Based Context Model**

```
PW := any common symbol in ΣA;  (* PW = Previous Word *)
State := PartOfSpeech[PW];
repeat
    read next word AW;
    if AW ∉ ΣA then
        output code for Escape and output text of AW;
        ΣA := ΣA ∪ {AW};
        Freq[State,Escape] := Freq[State,Escape] + 1;
        Freq[State,AW] := 1;
        PartOfSpeech[AW] := most probable for a successor of State;
    else
        output code for AW;
        Freq[State,AW] := Freq[State,AW] + 1;
        for P := each possible state do
            Cost[PW,P] := Cost[PW,P] - logarithm of the probability
                predicted for word AW by state P;
        PartOfSpeech[PW] := P such that Cost[PW,P] is a minimum;
    endif
    PW := AW;  State := PartOfSpeech[AW];
until end of input;
```

assigned to that other class. The decoding algorithm performs the same calculations and can therefore perform the same re-assignments. The overall structure of the compression algorithm is therefore as shown in Figure 4. For simplicity, punctuation strings are ignored in the presentation of the algorithm – they should be encoded by some other means.

# 4 Experimental Results and Discussion

The four word-based algorithms described in this paper have been implemented and tested on several text files, using the UNIX *compress* program as a benchmark. The compression results are summarized in Table 1.The pair of numbers shown for the state-based context compression reflects the fact that the algorithm normally requires initialization with a vocabulary giving the parts of speech for words. In our experiments, the initial vocabulary contained all the words used in the first test document (the *csh* manual description). The upper number in each pair shows compression performance when that vocabulary is used; the lower number shows performance when *no* vocabulary at all is used. I.e., the algorithm simply assigns words to one of five classes as it seems to find appropriate.

The overall result is that performance is consistently better than our benchmark, the UNIX *compress* program, no matter which word-based method is used. Words apparently recur sufficiently often in the test files that any scheme for compressing repeated references to words will perform well. The compression scheme using a state model based on parts of speech in the English language has probably the most potential for improvement. The current crude scheme could be improved by monitoring the punctuation to check for a period (or other characters that terminate a sentence) and be used to reset the model state to the start of a new sentence. At present, two strings with different capitalization (e.g. ‘the’ and ‘The’) are treated as two different words, but they could be combined into a sin-

gle entry and the end-of-sentence test could predict which form to use. The words ‘A’ and ‘An’ could be segregated into separate article classes and thus predict words that start with vowels or consonants. Word suffixes, such as ‘ly’, could be used to make better guesses as to a new word’s part of speech, etc. Much further experimentation is required. For now, we recommend use of a simple and fast word-based algorithm for text file compression. And of the possibilities considered, word-based LZW would seem to be the closest fit.

**Table 1 Comparative Compression Performance**

| | Original Size in bytes | Relative Size After Compression | | | | |
|---|---|---|---|---|---|---|
| | | UNIX compress | WB-Adpt Huffman | WB-LZW | State-Based | WB-First Order |
| On-line manual page for *csh* | 66772 | 40.4% | 29.8% | 34.3 | 21.6%<br>29.3% | 29.0% |
| On-line manual page for *make* | 63761 | 42.7% | 34.7% | 35.8% | 29.5%<br>33.6% | 31.7% |
| LaTeX file | 83106 | 44.7% | 36.1% | 32.0% | 36.0%<br>35.2% | 33.7% |
| C Source File | 24706 | 39.2% | 28.5% | 26.4% | 32.3%<br>27.0% | 25.3% |


## Acknowledgements

We are grateful to Kenny Wong for programming the word-based Adaptive Huffman coding algorithm whose results are reported here. We also thank the Natural Science and Engineering Research Council of Canada for their financial support.


## References


[1] Bell, T.C., Cleary, J.G., and Witten, I.H. *Text Compression*. Prentice-Hall , 1990.

[2] Bentley, J.L., Sleator, D.D., Tarjan, R.E., and Wei, V.K. “A Locally Adaptive Data Compression Scheme.” CACM 29, 4 (April 1986), pp. 320-330.

[3] Cormack, G.V., and Horspool, R.N. “Algorithms for Adaptive Huffman Codes.” Inf. Processing Letters 18, 3 (March 1983), pp. 159-166.

[4] Gallager, R.G. “Variations on a Theme by Huffman.” IEEE Trans. on Inf. Theory, IT-24, 6 (Nov. 1978), pp. 668-674.

[5] Horspool, R.N., and Cormack, G.V. “A General-Purpose Data Compression Technique with Practical Applications.” Proc. of CIPS Session ’84 (Calgary, Alberta), 1984, pp. 138-141.

[6] Horspool, R.N., and Cormack, G.V. “Technical Correspondence on ‘A Locally Adaptive Data Compression Scheme’.” CACM 30, 9 (Sept. 1987), pp. 792-794.

[7] Welch, T.A. “A Technique for High-Performance Data Compression.” IEEE Computer 17, 6 (June 1984), pp. 8-19.