# Dictionary Selection using Partial Matching

*Dzung T. Hoang*[1]
Digital Video Systems, Inc.
2710 Walsh Ave., Suite 200
Santa Clara, CA 95051
dth@dvsystems.com

*Philip M. Long*[2]
ISCS Department
National University of Singapore
Singapore, 119260
plong@iscs.nus.edu.sg

*Jeffrey Scott Vitter*[3]
Duke University
Box 90129
Durham, NC 27708–0129
jsv@cs.duke.edu

[1]Supported in part by an NSF Graduate Fellowship and by Air Force Office of Strategic Research grants F49620–92–J–0515 and F49620–94–1–0217.

[2]Supported in part by Air Force Office of Strategic Research grants F49620–92–J–0515 and F49620–94–1–0217. Telephone +65–772–6772, FAX +65–779–4580.

[3]Supported in part by Air Force Office of Scientific Research grants F49620–92–J–0515 and F49620–94–1–0217 and by a Universities Space Research Association/CESDIS associate membership.

**Abstract**

Our motivation is a desire to find text compressors that compress better than existing dictionary coders, but run faster than statistical coders. We describe a new method for text compression using multiple dictionaries, one for each context of preceeding characters, where the contexts have varying lengths. The context to be used is determined using an escape mechanism similar to that of PPM methods. We describe modifications of three popular dictionary coders along these lines and experiments evaluating their effectiveness using the text files in the Calgary corpus. Our results suggest that modifying LZ77, LZFG, and LZW along these lines yields improvements in compression of about 3%, 6%, and 15%, respectively.

Keywords: Text compression, dictionary compression, statistical compression, PPM, multiple dictionary compression.

# 1 Introduction

Text compression methods are usually divided into two categories: statistical algorithms use previously coded characters to estimate probabilities for the character to be coded, and these probabilities are used in conjunction with arithmetic coding. Dictionary-based algorithms replace sections of text by pointers into a dictionary. The PPM (prediction by partial matching) methods [4, 1] are examples of statistical coders, and the Ziv-Lempel methods [6, 15, 17, 18, 19] are examples of dictionary-based coders. Statistical coders typically give better compression while dictionary coders run faster.

A recent line of research has looked into constructing lossless coders that combine the compression efficiency of statistical coders with the speed of dictionary-based coders. With statistical coders, there has been some work on techniques for arithmetic coding [9, 10, 12] that trade arithmetic precision (and therefore compression) for speed. Furthermore, efficient data structures and coding heuristics have been developed to improve the running time and resource requirements of statistical modeling. While these techniques have succeeded in speeding up statistical coding without sacrificing much compression, they are ultimately limited by coding one character at a time.

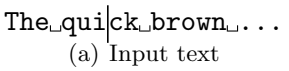
From the dictionary coding side, recent work [7, 13] has attempted to improve the compression of dictionary-based methods without slowing them down too much by maintaining separate dictionaries, one for each source character. A dictionary is chosen for coding based upon the last character coded. Since the last character coded is known by both the encoder and decoder, the same dictionary will be used for both encoding and decoding.

One motivation for using multiple dictionaries is that each dictionary can be made smaller, and therefore pointers into each dictionary can be represented more compactly. A drawback is that matches in a smaller dictionary may tend to be shorter. Will the compression efficiency lost by having shorter matches be more than offset by the more concise representation of dictionary pointers? Insights gained from results concerning the equivalence between dictionary and statistical coding suggest that this should be the case. It has been shown [11, 2], that some of the Ziv-Lempel algorithms are, in a sense, equivalent to corresponding statistical coders. Loosely speaking, a statistical method of this sort uses a varying number of previous characters as a context in order to calculate its estimated probability distribution for the next character. It cycles, first using a context of zero previous characters, then one and continuing for a varying amount of time, typically to around seven characters but sometimes longer, then returns to a context of zero characters. The proofs of these results lead to the intuition that a similar phenomenon exists with regard to the other Ziv-Lempel algorithms.

The use multiple dictionaries corresponds to an additional level of context modeling on top of the variable-context mechanism inherent in a greedy dictionary coder. In effect, a statistical equivalent to an order-1 multiple-dictionary scheme would start its cycle by using a context of one previous character, then two, and so on, instead of starting with an order-0 context. This directly addresses the observation in [1] that "the compression achieved by dictionary schemes suffers because of the loss of context at phrase boundaries" since some context is retained at phrase boundaries with context dictionaries.

Another drawback for using multiple dictionaries is that this practice results in some overhead. However, the increase in overhead can potentially be met with a decrease in search time, due to the smaller dictionaries.

The work of [7] placed a heavy emphasis on speed, and therefore compression gain was not



(a) Input text

| index | string |
|---|---|
| 0 | t |
| 1 | z |
| 2 | et |
| 3 | lt |
| 4 | ⟨Esc⟩ |

(b) qui dictionary

| index | string |
|---|---|
| 0 | ck |
| 1 | de |
| 2 | ld |
| 3 | sh |
| 4 | lty |
| 5 | dity |
| 6 | lding |
| 7 | ⟨Esc⟩ |

(c) ui dictionary

| index | string |
|---|---|
| 0 | d |
| 1 | n |
| 2 | t |
| 3 | ck |
| 4 | nd |
| ⋮ | ⋮ |
| 500 | zation |
| 501 | ⟨Esc⟩ |

(d) i dictionary

Figure 1: Example of a DPM coder with maximum order $o = 3$. At the current position, indicated by a vertical bar in (a), the available contexts are qui, ui, i, and $\Lambda$ (the zero-order context). The dictionaries for each context, except for $\Lambda$, are shown in (b), (c), and (d).

demonstrated. The compression results of the CSD algorithm of [13], which is based upon the LZW [17] dictionary coder, show more promise. However, the CSD algorithm is limited to using a single-character context.

In this paper, we carry the multiple-dictionary-based approaches of [7, 13] further with a general scheme that combines the variable context length modeling techniques of PPM with various LZ dictionary coders. Since the publication of a preliminary version of this paper [8], further work has continued this line of research (see [3]).

## 2 Dictionaries with Partial Matching

At a high level, the new hybrid method, which we call Dictionaries with Partial Matching, or DPM for short, works as follows. The coder maintains a list of contexts of length 0 up to some upper limit $o$. For each context in the list, the coder maintains a dictionary containing strings that have previously occurred following the context. To encode the input starting from the current position, the coder first looks for the longest context in its collection that matches a context of the current position: call the context $\sigma$. It then looks for the longest string in $\sigma$'s dictionary that matches a prefix of the input starting at the current position. If there is a match, the identity of the matching string of $\sigma$'s dictionary is transmitted, otherwise a special *escape* symbol is transmitted, and the process is repeated with the next longest context. If even the length-0 context fails to find a match, a final escape is sent, and the character in the position to be coded is sent literally. If the identity of a string in some dictionary is sent, the position to be coded is advanced to the first position in the input past those "matching" the given string. If a literal is coded, the position is moved forward one. In either case, the dictionaries corresponding to the matching contexts are updated. Optionally, the context dictionaries that were escaped from may be updated as well.

We now illustrate DPM with an example in Figure 1. At the current position, indicated by

a vertical bar in (a), the available contexts are `qui`, `ui`, `i`, and $\Lambda$ (the zero-order context). The dictionaries for each context, except for $\Lambda$, are shown in (b), (c), and (d). A search of the `qui` dictionary does not yield a match, and so the escape symbol is coded using $\lceil \log_2 5 \rceil = 3$ bits. The `ui` dictionary is then searched, yielding the longest match at index 0. The index is coded with $\lceil \log_2 8 \rceil = 3$ bits.

The above example illustrates the basic operations of DPM. It also demonstrates the savings in coding a dictionary pointer that results from coding with a small high-order dictionary. If the `i` dictionary were used, nine bits would be needed to code a pointer. With the same memory usage, a conventional dictionary coder would require even more bits to code a pointer.

The example leaves out many important details that would go into the design and implementation of an actual DPM coder. In order to get a better idea of what these issues are, we consider three instantiations of the DPM idea, one for each of three dictionary-based coders: LZ77 [18, 6], LZ78 [19, 17], and LZFG [6, 14]. The selection of a dictionary method involves choosing a data structure for maintaining the dictionaries, a policy for adding and removing strings from the dictionaries, and a scheme for coding dictionary pointers and escapes. LZ77-PM, the coder described in Section 3, makes these decisions in a manner analogous to the A1 instantiation in [6] of the LZ77 method. LZW-PM, the coder of Section 4, is based upon LZW [17], an LZ78 variant. LZFG-PM, the coder of Section 5, is based upon an implementation of LZFG by Slyz [14].

The LZ77-PM and LZW-PM schemes have been simulated in software with programs that keep count of the number of encoded bits, but do not actually generate a compressed output. Since the main interest is in the compression gain by adopting DPM, no attempt has been made to use efficient data structures or searching techniques, except for a bare minimum to make the simulations run in a reasonable amount of time with the available memory.

The LZFG-PM scheme, on the other hand, has been implemented as a fully functional encoder and decoder. Memory use is limited by fixing the size of the dictionary data structures. This allows for evaluation of DPM in a real-world setting where memory resources are limited.

# 3 LZ77-PM

In an LZ77-style coder, the dictionary is implicitly defined as all substrings, within length restrictions, that are contained in a buffer of the recent past input. In defining a multiple-dictionary extension of LZ77, we seek to preserve the nature of the dictionary. A natural extension of LZ77 to multiple dictionaries, then, is to have the effective dictionary for context $\sigma$ contain all substrings, within length restrictions, that have occurred in the past following the context $\sigma$. This is the approach described in [7] for single-character contexts, where each context dictionary is implemented as a linked list with hashing.

We now describe an PM modification of a simple LZ77 coder that is almost identical to the A1 coder from [6].

## 3.1 Baseline Coder

The baseline (non-PM) coder works as follows. First, it divides the input into buffers of size 256K (where K denotes 1,024 characters), and codes each buffer separately. While coding a particular buffer, the dictionary at any given time consists of all strings in the buffer preceding the current position that are of length at least 2 and at most 16. At each coding step, the encoder finds

the longest string in the dictionary that is identical to the string of the same length obtained by reading forward in the buffer starting at the current position. If $p$ is the buffer position being coded (counting from 1), the encoder tells the decoder which string "matched" by giving the position earlier in the buffer where the match occurred along with the length of the match. The position is encoded using $\lceil \log_2 p \rceil$ bits. The length is encoded in binary using four bits. If there is no matching string in the dictionary, the remaining 4-bit codeword (the first 15 were used to encode lengths 2 through 16) is transmitted, and the character at position $p$ is coded literally; i.e., as it appears in the input.

### 3.2 PM Modification

Our PM modification involves the addition of context dictionaries together with a PPM-style escape mechanism. As in the baseline coder, the input is coded in 256K buffers. The context dictionaries are not explicitly maintained. Instead, when searching for a match in context $\sigma$, $\sigma$ is included as the prefix of the search string. For example, let $\alpha$ be the input string of length 16 (the maximal match length) that starts at the current coding position. A search in context $\sigma$ would involve searching for the longest prefix of $\sigma\alpha$ that has occurred in the buffer before the current position. In conducting this search, the number of times $\sigma$ has occurred previously in the buffer is recorded by counter $n$. The position $k$ of the longest match is noted as the value of $n$ the first time the string is matched. In essence, the search in context $\sigma$ returns the length and index of the longest match within the implicit context dictionary. Let $N$ denote the final value of counter $n$, $\beta$ the longest prefix of $\alpha$ that is matched, $l_\beta$ the length of $\beta$, and $k$ the value of $n$ the first time $\beta$ was matched. If $N$ is above a given threshold $T_\sigma$, the encoder uses the current context for coding. Otherwise, the encoder escapes to a lower-order context. Since the decoder can perform the same test, no escape symbol needs to be sent. If $N > T_\sigma$ and no matching string is found, the encoder sends an escape symbol and escapes to the next lower-order context. If $N > T_\sigma$ and a match is found, the index $k$ and the length $l_\beta$ of the match are transmitted using $\lceil \log_2 N \rceil$ and 4 bits, respectively.

By introducing the threshold $T_\sigma$, we are delaying the use of a context dictionary until it reaches a certain size. A reason for this is that a small dictionary is not likely to yield a match. Therefore, the expense of having to transmit an escape symbol every time there is no match is great relative to the potential savings in coding by using the dictionary when there is a match. Although a similar problem exists in statistical coding, the implications for dictionary coding are more severe since the escape symbol must be coded in a more restricted setting where an integral number of bits is used. For example, the escape symbol may be coded as a flag bit, as a special length value, or as a special index value.

In the baseline coder, only strings with at least two characters are coded by dictionary reference. This is done since it is more economical to code a one-character string as a literal using 12 bits than as a dictionary pointer using $4 + \lceil \log_2 p \rceil$ bits when $p > 256$, where $p$ is the current coding position in the buffer. We face a similar situation with the PM modification. Experiments show that coding with a minimum length of 2 is better than with a minimum of length of 1.

We have explored several ways to encode the escape symbol. The first is to code the escape as a length-0 match without a match index, using 4 bits, as is done in the baseline coder when coding a literal. The second is to code the escape with a match index of 0 without a length field, using $\lceil \log_2 N + 1 \rceil$ bits. In experiments with text files in the Calgary corpus [1], coding the escape as a length-0 match and using a minimum match-length of 2 yielded consistently better compression.

### 3.3 Experimental Results

We performed coding experiments to observe the effects of the threshold $T_\sigma$ and the maximum context order in coding a test file (`paper2`) from the Calgary corpus.[1] In one experiment, we limited the maximum context order to 1 and varied the threshold for order-1 contexts. In a second experiment, we fixed the threshold for order-1 contexts to 300 and varied the threshold for order-2 contexts. The compression results are summarized in Table 1 and are shown in more detail in Figure 2 as plots of the distribution of coding bits versus the threshold value. As expected, the proportion of bits spent coding in the highest context order decreases as the threshold is increased.

For the order-1 LZ77-PM coder, the compression efficiency peaks when the threshold is about 300. For the order-2 LZ77-PM coder, the compression rate approaches, but does not beat, that of the order-1 LZ77-PM coder, even when the threshold is 2000. This suggests that a single-character context is adequate to restore context at phrase boundaries in an LZ77-style coder.

Complete results for coding the Calgary corpus with an order-1 LZ77-PM coder are shown in Table 2. Since both the base LZ77 and LZ77-PM coders are limited to matches of length 16 or less, there is a upper bound of the compression ratio that is achievable. For example, this limit is reached for the file `pic`, a digital image that can be compressed fairly well with the other dictionary coders.

On average, DPM improves the compression rate by about 2%. It should be noted, however, that for the text files in the corpus, the improvement is about 3.4%. The non-text files in the corpus generally perform worse with DPM than without.

## 4 LZW-PM

In the LZ78 coder, a literal character is transmitted after every dictionary pointer. One reason for this is to insure that at least one character is coded with each encoding step. In [15], this is referred to as *pointer guaranteed progress*. The LZW coder [17] eliminates the need to transmit a literal after every match by initializing the dictionary to contain the source alphabet, that is, all strings of length 1. In this way, only dictionary pointers are coded. Since all characters are in the dictionary, a match will always be found. This is referred to as *dictionary guaranteed progress* in [15].

LZW uses the same dictionary update rule as LZ78. After each match, a single string consisting of the matched string concatenated with the first unmatched character is added to the dictionary. This is one case where the decoder's dictionary differs from the encoder's dictionary since the decoder does not yet know the identity of the unmatched character. This is not as serious a problem as it may first seem. If the encoder happens to send the newly added string as the next match, the decoder can recover the unmatched character as the first character in the next match, which is already known to the decoder.

### 4.1 Baseline Coder

In the UNIX `compress` implementation of LZW, the size of the dictionary is restricted. In defining our baseline coder, this restriction is removed. Also, instead of initializing the dictionary to contain the characters of the alphabet, the dictionary initially contains only the empty string. When the

[1] Tuning an algorithm by considering its behavior on data used later to evaluate the algorithm is in general to be avoided; however, in this case, the vast quantity of data compared with the small number of parameters tuned suggests that the effect on the results will be negligible (see [16]).

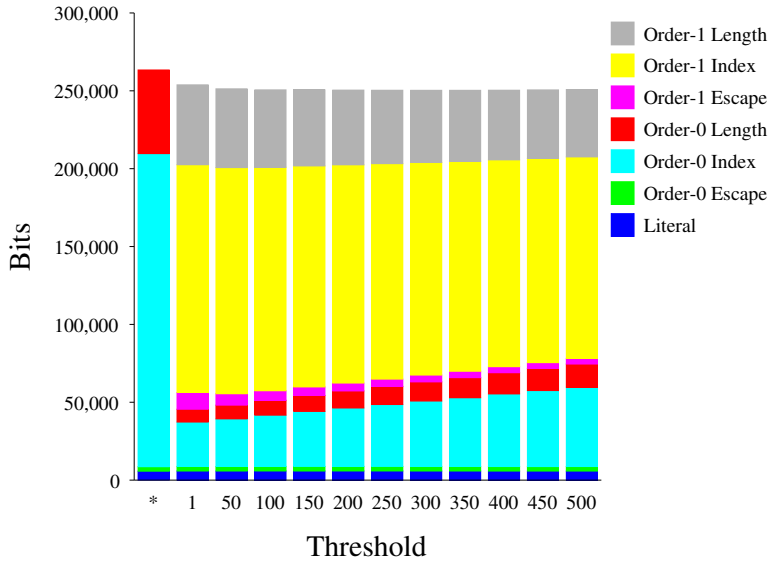
(a) Order-1 LZ77-PM

| Threshold | Bits for Literal | Bits for Order 0 | Bits for Order 1 | Total Bits | Total Bits/Symbol |
|---|---|---|---|---|---|
| (∗) | 5744 | 257099 | N/A | 262843 | 3.198 |
| 1 | 5864 | 39639 | 208335 | 253838 | 3.088 |
| 50 | 5848 | 42310 | 203126 | 251284 | 3.057 |
| 100 | 5872 | 45379 | 199363 | 250614 | 3.049 |
| 150 | 5864 | 48510 | 196226 | 250600 | 3.049 |
| 200 | 5880 | 51403 | 193176 | 250459 | 3.047 |
| 250 | 5872 | 54297 | 190202 | 250371 | 3.046 |
| 300 | 5864 | 57215 | 187244 | 250323 | 3.045 |
| 350 | 5840 | 59948 | 184541 | 250329 | 3.045 |
| 400 | 5816 | 63143 | 181435 | 250394 | 3.046 |
| 450 | 5808 | 65921 | 178865 | 250594 | 3.049 |
| 500 | 5816 | 68730 | 176297 | 250843 | 3.052 |

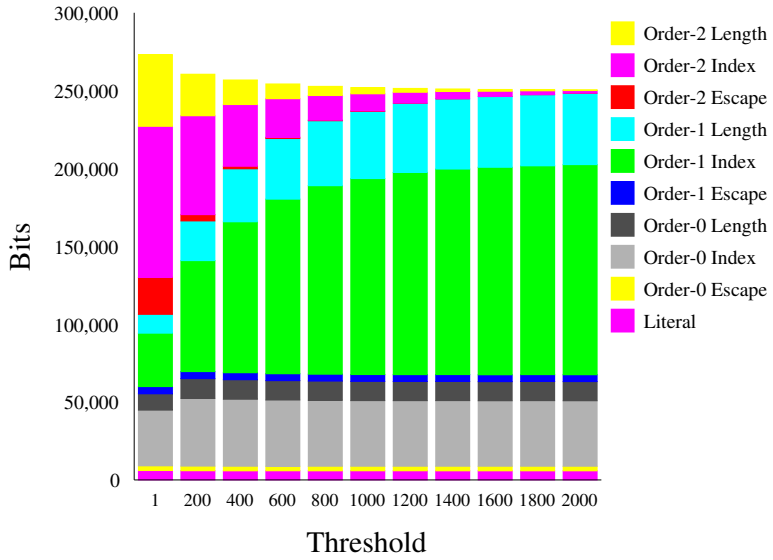
(b) Order-2 LZ77-PM

| Threshold | Bits for Literal | Bits for Order 0 | Bits for Order 1 | Bits for Order 2 | Total Bits | Total Bits/Symbol |
|---|---|---|---|---|---|---|
| 1 | 6040 | 49279 | 51009 | 166785 | 273113 | 3.323 |
| 200 | 5920 | 59124 | 101287 | 94204 | 260535 | 3.170 |
| 400 | 5864 | 58500 | 135435 | 57034 | 256833 | 3.125 |
| 600 | 5872 | 57882 | 155486 | 35040 | 254280 | 3.093 |
| 800 | 5864 | 57613 | 167182 | 22083 | 252742 | 3.075 |
| 1000 | 5872 | 57399 | 173406 | 15271 | 251948 | 3.065 |
| 1200 | 5872 | 57342 | 178608 | 9598 | 251420 | 3.059 |
| 1400 | 5864 | 57382 | 181428 | 6391 | 251065 | 3.054 |
| 1600 | 5856 | 57277 | 183037 | 4588 | 250758 | 3.051 |
| 1800 | 5864 | 57339 | 184155 | 3384 | 250742 | 3.050 |
| 2000 | 5864 | 57299 | 185195 | 2231 | 250589 | 3.049 |

Table 1: Distribution of bits for coding `paper2` with various threshold values. Results are shown in (a) for an order-1 LZ77-PM coder. The row marked with (∗) gives results for the baseline non-PM LZ77 coder. In (b), results are shown for an order-2 LZ77-PM coder with fixed order-1 threshold of 300.




(a) Order-1 LZ77-PM




(b) Order-2 LZ77-PM

Figure 2: Distribution of bits versus context threshold $T_\sigma$. The results are shown for coding file `paper2` of the Calgary corpus. In (a), except for the first column, the results are for varying the order-1 context threshold for LZ77-PM with maximum context order of 1. The first column of (a), indicated by $(*)$, shows the coding results for the non-PM baseline coder. In (b), the results are for varying order-2 context threshold with an order-1 threshold of 300 for LZ77-PM with maximum context order of 2.

longest match is the empty string, the next character in the file is coded literally. This can be viewed as an escape mechanism similar to coding the escape symbol as a match index in the LZ77-PM coder described above. In [15], this is referred to as *on-the-fly dictionary guaranteed progress*. This modification has a negligible effect on the compression and is done so that we can view the baseline coder as a special case of the PM coder where the maximum context is of order 0.

In LZW, the dictionary is updated in the same manner as in LZ78. The longest matching string concatenated with the first unmatched character is added to the dictionary. The result is that one entry is added for each dictionary match coded. In contrast, an LZ77 coder grows its effective dictionary at a faster rate.

### 4.2 PM Modification

In the LZW-PM coder evaluated in our experiments, we maintain an LZW dictionary for each previously seen context up to a maximum length, say $o$. As in the baseline coder, an important difference from LZW is that the dictionary initially contains the empty string in order to make use of the escape mechanism.

At each coding step, the longest matching context, say $\sigma$, is used for coding. The dictionary corresponding to $\sigma$ is searched for the longest match. Regardless of the length of the match, which may be zero, the LZ78 dictionary update rule is used. If the longest match is the empty string, an escape symbol is coded[2] and the next longest matching context is used for coding. This process is repeated until a non-empty match is found or until an escape is coded from the order-0 context. A character is coded literally after an escape from the order-0 context. The identity of a dictionary entry is coded using a phased-in binary code using either $\lfloor \log_2 D \rfloor$ or $\lceil \log_2 D \rceil$ bits, where $D$ is the size of the dictionary.

Coding the escape symbol as a dictionary entry approximately corresponds to the method used in PPMA [1], since it (approximately) involves the implicit assumption that the probability of seeing a previously unseen character in a given context is the inverse of the number of times the context has been seen. While other probability models for escapes performed better as part of PPM (see [1, 9]) and are therefore likely to yield better compression here, we chose the method above for simplicity and computational efficiency.

We found that, unlike the case of LZ77-PM, delaying the use of a context does not yield better compression for LZW-PM in general. This is most likely due to the different rates at which the dictionaries are updated. Since LZ77-PM bases its dictionaries on the contents of a buffer of past input, each time a match is coded, multiple entries are added to the effective dictionaries of several contexts corresponding to the matched string. In contrast, only one entry is added to each context matched in LZW-PM. Intuitively, in the early growing phase, the LZ77-PM dictionaries quickly become "diluted", increasing the number of bits required to code a match without a corresponding increase in the reliable of prediction. Only after the initial growing phase does the cost of coding a match balance the reliability of prediction.

### 4.3 Experimental Results

Experiments on the Calgary corpus show a significant improvement in the compression rate when DPM is applied to the baseline LZW coder. The best results for the corpus as a whole is obtained

[2]The empty string occupies one entry in the dictionary and an escape is coded by coding the index of the empty string.

for a maximum context order of 2. However, the best maximum context order for each file varies. For some of the binary files, notably `geo`, the compression degrades when using even an order-1 context dictionary. In contrast, the text files are better compressed by using context dictionaries. On average, introducing Partial Matching improves the compression rate by about 11%. For text files, the improvement is about 15%.

# 5 LZFG-PM

In the LZ77 coder, for a given match, several positions in the buffer may contain the same match. As a result, LZ77 effectively reserves codespace for pointers that will never be coded, which is wasteful. The LZFG coder of Fiala and Green [6] eliminates pointer redundancy by coding a string as a pointer into a trie, a digital search tree which stores parsed substrings that have occurred previously in the text. Each substring stored in the trie has a unique identifier. LZFG's efficient coding of pointers makes it an ideal candidate as a base dictionary coder for DPM. In addition, LZFG is a symmetric coder since the decoder must maintain the same trie. In a sense, DPM is also a symmetric coding scheme since the encoder and decoder must make the same context selection. Integrating DPM into LZFG, then, would have minimal impact on both the encoder and decoder. For these reasons, we chose to implement LZFG-PM as a fully functional encoder/decoder program.

## 5.1 Baseline Coder

We use an existing LZFG implementation [14] as the baseline coder. Some details of this implementation differ from the coder originally described by Fiala and Greene. In the base implementation, a sliding-window buffer of size 44K is used with the sliding occurring in 4K chunks. A limited number of internal nodes and leaves is globally allocated. Nodes and leaves are stored in self-organizing lists that use the least-recently-used (LRU) update heuristic. New nodes and leaves replace old ones when their allotted number has been reached. The interested reader is referred to [14] for further details of the implementation.

## 5.2 PM Modification

Following the outline of Section 2, LZFG-PM constructs separate tries for each context, including one for the order-0 context. Although the tries are separate, nodes and leaves are allocated from a common pool. When all nodes (respectively, leaves) have been allocated and there is a request for a new node (leaf), the oldest node (leaf) in a globally managed LRU list is deleted. In this way, fixed memory usage is assured. Furthermore, since the different contexts are competing for the same memory resources in a LRU list, context dictionaries that are used more frequently will suffer fewer deletions.

Coding of literal characters is handled differently in LZFG-PM than in LZFG. Whereas LZFG groups consecutive literals together and sends them in one chunk, LZFG-PM codes each literal individually. The reasoning is that since LZFG-PM allows pointers to code phrases of length 1 efficiently, literals are less likely to occur consecutively than in LZFG.

We have implemented two methods for coding escapes. In the first, we code the escape symbol as a special node, namely the root. In the second, we code the escape symbol as a special entry in the LRU list of leaves, using the same self-organizing-list heuristic. We find that coding an escape as a node gives slightly better compression than coding it as a leaf.

We use a heuristic to determine the appropriate context order to begin parsing the next phrase. If the previous context order is $C$ and the length of the previous phrase is $L$, we do not use contexts with order higher than $L + C - 1$ for coding. Starting at an order higher than $L + C - 1$ would not likely be fruitful since that would mean repeating a search that has failed in the past. This heuristic saves us in the cost of coding escapes.

A new dictionary entry is added to the context used for coding as well as to higher-order contexts that were escaped from. Contexts of order less than the coding context could also be updated. In our experiments we find that updating lower-order contexts results in marginal improvement (about 0.01 bits/symbol) in compression at the expense of more overhead. Therefore we adopt the strategy of updating only the coding context and the contexts that are escaped from.

### 5.3 Experimental Results

Experimental results for the LZFG and LZFG-PM coders are given in Table 2. For both coders, the buffer size is 44K. Both coders limit the total number of internal nodes to 16K and the number of leaves to 32K. We experimentally determined the best settings for the start-step-stop codes used to code the nodes and leaves based upon a subset of the test files. We also varied the maximum context order and obtained the best results for a maximum context order of 1. The results, given in Table 2, are for a maximum context order of 1. The improvement in the average compression rate with DPM is about 4.3%. For text files, the improvement is 6.4%.

Our implementations of LZFG and LZFG-PM are experimental prototypes that were designed for flexibility and not efficiency. As such, the running times are about an order of magnitude more than "commercial-grade" compressors such as `gzip`. For the file `book1`, LZFG-PM requires 22% more time to compress than LZFG. This gives us an indication of the overhead of DPM over conventional dictionary coders.

## 6 Discussion

In this paper, we have described a new approach to dictionary-based text compression, and we have demonstrated that it results in improved compression. The improvement is particularly significant when used with LZW. We expect that the use of the dictionary PM approach also yields practically fast coders. The fact that this approach succeeded in conjunction with the three coders studied in this paper suggests that the high-level specification of this approach can be combined with many dictionary methods, including those that might be developed in the future.

Why did the use of contexts improve compression significantly more in conjunction with some dictionary methods than with others? One plausible explanation is as follows. The LZ77 coder is the most liberal about adding entries to its dictionary, followed by LZFG then LZW. This parallels the level of improvement exhibited by introducing DPM, with LZ77 benefiting the least and LZW the most.

Past studies with PPM coders [1] have shown that, as the context order increases, the compression improves and then worsens, with the best compression when the maximum context order is about 4 or 5. The correspondence between (non-PM) dictionary methods and statistical methods suggests that, as a match continues, the probability estimates in the corresponding symbolwise statistical coder tend to get better as more contextual information is made available and then get worse as the estimates at high context orders become more unreliable. The effect of a context

| file | LZ77 | LZ77-PM | LZFG | LZFG-PM | LZW | LZW-PM | CSD |
|---|---|---|---|---|---|---|---|
| bib | 2.85 | 2.68 | 2.69 | 2.44 | 3.22 | 2.61 | 3.22 |
| book1 | 3.44 | 3.32 | 3.57 | 3.28 | 3.17 | 2.74 | 3.59 |
| book2 | 2.98 | 2.84 | 3.05 | 2.78 | 3.06 | 2.55 | 3.37 |
| geo | 6.71 | 6.96 | 5.65 | 5.63 | 5.79 | 6.51 | 6.74 |
| news | 3.51 | 3.42 | 3.42 | 3.25 | 3.62 | 3.18 | 4.13 |
| obj1 | 5.11 | 5.24 | 4.06 | 4.32 | 5.05 | 5.14 | 5.12 |
| obj2 | 3.51 | 3.40 | 3.10 | 3.04 | 4.08 | 3.42 | 3.80 |
| paper1 | 3.24 | 3.14 | 2.94 | 2.74 | 3.62 | 3.08 | 3.43 |
| paper2 | 3.20 | 3.05 | 3.02 | 2.80 | 3.38 | 2.94 | 3.28 |
| paper3 | 3.52 | 3.43 | 3.27 | 3.05 | 3.65 | 3.30 | N/A |
| paper4 | 3.94 | 3.89 | 3.55 | 3.41 | 3.98 | 3.69 | N/A |
| paper5 | 4.02 | 3.98 | 3.61 | 3.49 | 4.20 | 3.86 | N/A |
| paper6 | 3.34 | 3.25 | 3.00 | 2.80 | 3.77 | 3.19 | N/A |
| pic | 2.01 | 2.01 | 0.92 | 0.93 | 0.94 | 0.98 | 1.00 |
| progc | 3.28 | 3.22 | 2.89 | 2.74 | 3.72 | 3.18 | 3.42 |
| progl | 2.44 | 2.31 | 1.95 | 1.80 | 2.95 | 2.37 | 2.70 |
| progp | 2.42 | 2.34 | 1.91 | 1.81 | 3.00 | 2.40 | 2.66 |
| trans | 2.31 | 2.12 | 1.79 | 1.67 | 3.16 | 2.31 | 2.91 |
| Average | 3.44 | 3.37 | 3.02 | 2.89 | 3.58 | 3.19 | N/A |

Table 2: Compression results for files in the Calgary corpus. The results are expressed in bits/symbol. The LZ77 coder is almost identical to the A1 coder from [6]. LZ77-PM used a maximum context order of 1. The LZFG implementation used is from [14]. LZFG-PM used a maximum context order of 1. LZW-PM used a maximum context order of 2. CSD results are from [13]. CSD used a limited dictionary size (16K), while LZW and LZW-PM had unlimited dictionaries; the effects of this is most notable on the longer text files: `book1` and `book2`.

dictionary is, loosely speaking, to bypass the lower order contexts with relatively poor probability estimates, thereby increasing the average quality of the remaining probability estimates.

A method that liberally updates its dictionary, all else being equal, makes longer matches at the expense of requiring more bits to identify the match. By tending to have longer matches, such a method has a relatively greater share of its coding inefficiency due to the later characters in its matches. For such a method, bypassing low-order contexts will not yield much of an improvement. Further, even if, for example, using an order-2 context to encode a single character tends to be worse than using an order-3 context, it is possible that the order-2 dictionaries tend to be better than the order-3 dictionaries. To see why this might be true, consider what happens in the corresponding statistical method when we use an order-2 context rather than an order-3 context. Loosely speaking, instead of cycling from contexts of length 3 up to some large number, we cycle from 2 through some large number. Adding the order 2 to the cycle will result in improved compression if, loosely speaking, using order-2 contexts is better than the average of the use of contexts of order 3 and higher.

A method that is "stingy" about adding entries to its dictionary, like LZW, has a greater share of its inefficiency due to the early characters in a match, and therefore we would expect it to be helped more by the addition of contexts, as is consistent with our experiments.

Cleary, Teahan and Witten [5] recently discovered an improvement of PPM, called PPM$*$, which does not impose any restriction on the length of the context used. Instead, PPM$*$ uses the longest of the present contexts which has appeared previously. DPM could conceivably be modified similarly: this is potentially a subject of future research.

## References


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

[2] T. C. Bell and I. H. Witten. Relationship between greedy parsing and symbolwise text compression. *Journal of the ACM*, 41(4):708–724, July 1994.

[3] C. Bloom. Using prediction to improve LZ77 coders. *Proceedings 1996 IEEE Data Compression Conference*, page 425, 1996. Full version available at `http://wwwvms.utexas.edu/ cbloom/dcc96.html`.

[4] J. G. Cleary and I. H. Witten. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communication*, COMM–32(4):396–402, April 1984.

[5] J.G. Cleary, W.J. Teahan, and I.H. Witten. Unbounded length contexts for PPM. *Proceedings of the 1995 Data Compression Conference*, pages 52–61, 1995.

[6] E. R. Fiala and D. H. Greene. Data compression with finite windows. *Communications of the ACM*, 32(4):490–505, April 1989.

[7] P. C. Gutmann and T. C. Bell. A hybrid approach to data compression. In *Proceedings 1994 Data Compression Conference*, pages 225–233, Snowbird, UT, March 1994. IEEE Computer Society Press.

[8] D. T. Hoang, P. M. Long, and J. S. Vitter. Multiple-dictionary compression using partial matching. In *Proceedings 1995 IEEE Data Compression Conference*, pages 272–281, Snowbird, Utah, March 1995.

[9] P. G. Howard and J. S. Vitter. Practical implementations of arithmetic coding. In J. A. Storer, editor, *Images and Text Compression*, pages 85–112. Kluwer Academic Publishers, Norwell, MA, 1992.

[10] P. G. Howard and J. S. Vitter. Design and analysis of fast text compression based on quasi-arithmetic coding. *Information Processing and Management*, 30(6):777–790, 1994.

[11] G. G. Langdon. A note on the Ziv-Lempel model for compressing individual sequences. *IEEE Transactions on Information Theory*, IT–29:284–287, March 1983.

[12] J. L. Mitchell and W. B. Pennebaker. Optimal hardware and software arithmetic coding procedures for the Q-Coder. *IBM Journal of Research and Development*, 32:727–736, November 1988.

[13] Y. Nakano, H. Yahagi, Y. Okada, and S. Yoshida. Highly efficient universal coding with classifying to subdictionaries for text compression. In *Proceedings 1994 Data Compression Conference*, pages 234–243, Snowbird, UT, March 1994. IEEE Computer Society Press.

[14] M. Slyz. Image compression using a Ziv-Lempel type coder. Master's thesis, University of Michigan School of Engineering, 1991.

[15] J. A. Storer. *Data Compression: Methods and Theory.* Computer Science Press, New York, NY, 1988.

[16] V. N. Vapnik. *Estimation of Dependencies based on Empirical Data.* Springer Verlag, 1982.

[17] T.A. Welch. A technique for high performance data compression. *IEEE Computer*, 17:8–19, 1984.

[18] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, IT–23(3):337–343, 1977.

[19] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, IT–24:530–536, September 1978.