SOFTWARE—PRACTICE AND EXPERIENCE
*Softw. Pract. Exper.* 2006; **36**:1687–1710
Published online 13 July 2006 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe.741

# Decoding prefix codes‡

Mike Liddell and Alistair Moffat*,†

*Department of Computer Science and Software Engineering,*
*The University of Melbourne, Victoria 3010, Australia*

**SUMMARY**

**Minimum-redundancy prefix codes have been a mainstay of research and commercial compression systems since their discovery by David Huffman more than 50 years ago. In this experimental evaluation we compare techniques for decoding minimum-redundancy codes, and quantify the relative benefits of recently developed restricted codes that are designed to accelerate the decoding process. We find that table-based decoding techniques offer fast operation, provided that the size of the table is kept relatively small, and that approximate coding techniques can offer higher decoding rates than Huffman codes with varying degrees of loss of compression effectiveness. Copyright © 2006 John Wiley & Sons, Ltd.**

*Received 14 January 2005; Revised 21 December 2005; Accepted 21 December 2005*

KEY WORDS: Huffmann code; minimum-redundancy code; prefix code; canonical code; table-based decoding

## 1. INTRODUCTION

It is now more than 50 years since minimum-redundancy codes were developed by David Huffman [1]. Despite the development of other techniques such as arithmetic coding, minimum-redundancy codes remain a key component of many data compression systems, and for typical applications they provide both near-optimal compression effectiveness and fast decoding.

Many variants of the basic technique have been suggested, including high-throughput methods, low-memory methods and methods that support external operations such as compressed searching. In this paper we explore the alternatives that have been proposed for fast decoding, including through the use of non-minimal codes with restricted structures. The various methods have been tested in a uniform experimental framework using inputs that represent several major classes of message types.

*Correspondence to: Alistair Moffat, Department of Computer Science and Software Engineering, The University of Melbourne, Victoria 3010, Australia.
†E-mail: alistair@csse.unimelb.edu.au
‡Parts of this paper appeared in preliminary form in the Proceedings of the 2003 IEEE Data Compression Conference, Snowbird Utah, April 2002, pages 392–401.

Contract/grant sponsor: Australian Research Council

Previous experimental studies of prefix-code decoding are surprisingly few. Hirschberg and Lelewer [2] describe experiments on *ad hoc* tree-based methods and canonical codes following the original presentation by Schwartz and Kallick [3] and Connell [4], but primarily emphasize low-memory decoding rather than techniques which use additional memory to achieve high throughput.

Since then, Moffat and Turpin [5] have described a faster way of decoding canonical codes; Klein [6] has described a structure called skeleton trees for fast decoding and a range of authors have described table-driven decoding mechanisms, all based on similar approaches.

We believe that it is now timely to undertake an experimental evaluation of minimum-redundancy decoding techniques, and such a comparison is the main thrust of this paper. In particular, we explore the benefits and shortcomings of a number of competing alternatives, working with block-based semi-static implementations that can be used to achieve pseudo-adaptive coding. A secondary motivation for this experimentation with prefix decoding variants is the continued emergence of new suggestions for the decoding process. In particular, Nekritch [7] and Milidiú *et al.* [8] both advocate the use of large decoding tables for processing multiple symbols at a time—an idea that has reappeared regularly in the literature (see, e.g., Turpin and Moffat [9]). Similarly, Hashemian [10] has recently redeveloped the notion of canonical table-driven coding, notwithstanding previous descriptions of the same (or better) techniques.

Restricted forms of prefix code have the potential to yield further benefits in the decoding process. The idea is that some amount of compression effectiveness can be surrendered in order to construct a code with a structure particularly amenable to fast decoding. One such mechanism is the $K$-flat code structure described recently by Liddell [11], and Chen *et al.* [12] investigate a related problem. Byte-aligned codes have also been argued for in terms of fast decoding (de Moura *et al.* [13], Scholer *et al.* [14], Brisaboa *et al.* [15], Culpepper and Moffat [16]). In these methods a radix-256 (or related) code is constructed, so that all operations are on 8-bit quantities rather than individual bits. Another form of approximate code is the CARRY method of Anh and Moffat [17], discussed in more detail below.

In order to isolate the modeling part of the compression system from the coding effects we wish to study, our experiments presume that an independent and identically distributed stream of integers is to be compressed, transmitted to the decoder and then exactly reconstructed. The information embedded in the stream is of no relevance to this process and neither is the mechanism used to generate it. This is because our experiments are separate from any discussion of *modeling*, and we assume that the symbol numbers that are to be coded are uncorrelated. For further discussion of the relationship between modeling and coding, and for examples of their interaction, see Moffat and Turpin [18].

The particular test streams used in the experiments are the output of the modeling stages of several different compression systems. We could equally have served our purposes by generating integer data streams using a Bernoulli process, but the use of 'real' examples provides a greater level of authenticity. To reflect the abstraction involved, compression effectiveness is reported in terms of bits per symbol, where a 'symbol' is one integer in an input file consisting of a stream of 32-bit integers. Not relating the compression effectiveness results back to the original data stream and avoids conflating the quite distinct issues of choice of model and choice of coder.

The particular emphasis in this investigation is on the decoding end of the compression task, assuming a semi-static encoding mechanism. That is, given a bufferable stream of integers that has been generated by some model and then compressed using a minimum-redundancy code, we seek the best way of going about the task of reconstructing the stream. The outcome of our experiments is a set of recommendations as to practical tradeoffs in terms of implementation choices. In particular, we find that table-based decoding techniques offer fast operation, provided that the size of the table is kept relatively small, and that approximate coding techniques can offer higher decoding rates than Huffman codes with varying degrees of loss of compression effectiveness.

## 2. PREFIX CODES AND DECODING

In a semi-static code, the compressed message stream is an interleaved set of one or more *preludes* and an identical number of *code-streams*. Each prelude describes a mapping from codewords to source symbols, and allows the decoder to reconstitute a segment of the *source message* from the corresponding code-stream.

The classical approach to prefix-code decoding is to use the prelude information to build an explicit code tree. Each bit in the code-stream is then used to drive an edge traversal in the code tree, and a symbol (integer) is emitted each time a leaf is reached. This approach to Huffman decoding appears in a range of textbooks.

While simple, the tree-based approach has serious deficiencies (Moffat and Turpin [5]). Even with some level of packing and field reuse, the tree structure requires $2n-1$ nodes, and typically $4n$ words of memory. It is also slow to process—extracting single bits from the incoming code-stream is relatively expensive and at the same time pointer-following on a per-bit basis through a linked data structure is likely to lead to significant issues with cache misses.

### 2.1. Finite state decoding

Three distinct threads of development have been suggested to eliminate the need for bit-by-bit decoding.

The first is what might be loosely labeled the *finite state machine* approach (Choueka *et al*. [19], Tanaka [20], Siemiński [21]). Instead of using a single bit at a time to drive navigation through the code tree, fixed units of $k$ bits are used; and instead of a tree, a state machine is traversed. The machine still has $n-1$ states, exactly corresponding to the $n-1$ internal nodes of the explicit code tree. However, each of those states now has $2^k$ outgoing edges (rather than two), linking to one of the $n-1$ states and indexed from 0 to $2^k-1$ using a $k$-bit number (rather than indexed by 0 and 1 using a single bit). Finally, each edge is labeled with zero or more symbols from the source alphabet. For example, a $k=1$ machine is formed from the standard decoding tree by adding two edges from the parent of each pair of leaves back to the root and labeling those edge with the symbols that were previously assigned to the pair of leaves.

To operate the machine, the state corresponding to the root of the code tree is first selected. Then a $k$-bit unit from the compressed bit-stream is used to determine an outgoing edge. That edge is traversed and if it is labeled with symbol numbers those numbers are written to the output. Processing resumes from the destination state of that edge. Moffat and Turpin [18] give an example of state machine decoding.

For typical applications $k=4$ or $k=8$ are appropriate values. When the latter is chosen, the compressed stream is processed as a sequence of bytes, and no bit operations are required. However, the increase in decoding throughput is at the expense of a significant increase in the amount of memory required. At an absolute minimum, each node in the machine must store $2^k$ pointers.



DOI: 10.1002/spe

The labels on the edges, even if to simply say that zero symbols are to be output, are an additional cost and mean that the space requirement is at least $2^k(n-1)$ words, a considerable burden when $k=8$.

### 2.2. Table-driven decoding

The second thread of development is *table-driven decoding*. The idea here is to use an array of $2^L$ entries, where $L$ is the length of a longest codeword, together with a *buffer* variable that holds the next $L$ bits from the compressed message. The entry in position $p$ of the table indicates the number of bits in the first codeword within the bit-pattern $p$, and the corresponding symbol number. Decoding is a matter of using the table and the buffer to identify and output the next symbol, then shifting out the corresponding number of high-order bits and finally replacing them in the buffer by the same number of low-order bits extracted from the code-stream. Mechanisms of this kind have been described by Bassiouni and Mukherjee [22] and Hashemian [23], as well as implemented in practical compression systems such as `gzip`. A refinement is for the single large table to be replaced by a cascading sequence of smaller tables each processing the next $k < L$ bits of the input data, in a mechanism that incorporates some elements of the state machine approach.

While they still employ non-aligned bit operations, table-driven approaches perform bit masking and extraction on a 'once per decoded symbol' basis rather than for every decoded bit—a definite advantage. However, space is again an issue and $O(2^L)$ words of memory are required during decoding, a plausible requirement for small alphabet sizes, where $L=16$ (for example), but unreasonable when $L=20$ or more.

### 2.3. Canonical codes

The third thread of development has been in the area of *canonical codes* (Schwartz and Kallick [3], Connell [4], Larmore and Hirschberg [24], Zobel and Moffat [25]). A canonical code is a prefix code in which the codewords of each different length $\ell$ form a contiguous set of $\ell$-bit binary integers, and in which all codewords of length $\ell$ lexicographically precede all codewords of length $\ell+1$, for all $1 \le \ell < L$, where $L$ is the maximum codeword length.

A canonical minimum-redundancy code is formed by taking the codeword lengths generated by Huffman's algorithm and assigning codewords to symbols in decreasing probability order, starting with a codeword of all zeroes. Figure 1 gives an example of a canonical code, and shows two tables that assist in manipulating it: the *base* array, which stores the first codeword of each bit-length; and the *count* array, which stores the number of codewords of that bit-length. Both of these arrays are indexed by codeword lengths, $1 \le \ell \le L$.

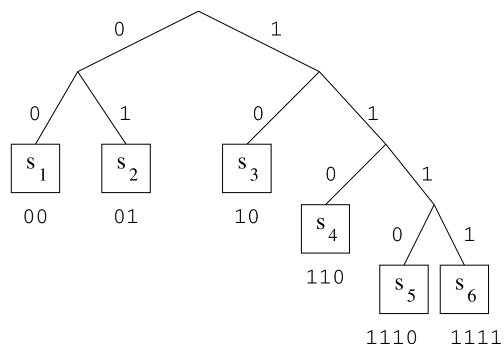
If $\ell_i$ is the length of the codeword to be assigned to symbol $i$, and $c_i$ is a decimal representation of that codeword, then

$$c_i = \begin{cases} 0 & \text{if } i = 1 \\ c_{i-1}+1 & \text{if } \ell_{i-1} = \ell_i \\ 2^{\ell_i - \ell_{i-1}} \cdot (c_{i-1}+1) & \text{otherwise} \end{cases}$$

In the example in Figure 1, the decimal codewords $c_i$, reading from left to right across the tree, are 0, 1, 2, 6, 14 and 15. The regular structure of canonical codes allows them to be decoded using a simple search process, described shortly.



DOI: 10.1002/spe




| $\ell$ | base codeword binary | base codeword decimal | count |
|---|---|---|---|
| 1 | – | – | 0 |
| 2 | `00` | 0 | 3 |
| 3 | `110` | 6 | 1 |
| 4 | `1110` | 14 | 2 |

Figure 1. The canonical Huffman code for the input $p_i = \{0.30, 0.26, 0.20, 0.15, 0.05, 0.04\}$, with both the code tree and an implicit description shown. The codeword lengths are determined by running Huffman's algorithm, but the actual codewords assigned to symbols are determined by the structure of the canonical tree dictated by those lengths. Codewords of a given length form a sequence, and leaf depths are non-decreasing across the tree. The table shows the base codeword, and the number of leaves, for each codeword length $\ell$ in the range $1 \leq \ell \leq 4$.

Before a canonical code can be used the source alphabet must be probability-sorted. In most applications, a mapping array from source symbols to probability-ordered surrogates is thus required. However, in semi-static compression a mapping array is typically required anyway, as it should not normally be assumed that all possible alphabet symbols appear in any particular message. That is, the fact that canonical codes are a restricted form of prefix code is not a practical impediment to their use in semi-static applications.

The simplest approach to decoding canonical codes is to add bits to a growing codeword, checking, after each bit is added, whether the corresponding *base* entry (see Figure 1) has been exceeded. For details of this approach, see Zobel and Moffat [25] or Moffat and Turpin [18]. The linear search in *base* requires bit-by-bit processing, but compared to the classic tree-based approach has the advantage of working sequentially within a short array section of length $L$ rather than across $2n$ tree nodes linked by pointers. Moreover, using a technique described by Moffat and Turpin [5], further acceleration is possible if the linear search is augmented by a small table, and an auxiliary buffer word containing $L$ or more bits. This fast approach is the baseline mechanism for our experiments, and is described in greater detail in Section 3.3.

### 2.4. Prelude considerations

In all semi-static coding methods, irrespective of the strategy used for decoding, a prelude describing the code must be transmitted to the decoder. Typical information required is:

1. the length of the block, $m$;
2. the highest symbol number appearing in the block, $s$;
3. the number of distinct symbol numbers in the block, $n$;



DOI: 10.1002/spe

4. a sub-alphabet description to identify the $n$ symbols in $1..s$ which appear in the block;
5. a description of the $n$ codewords, typically as the length $L$ of a longest codeword, followed by a list of $n$ integers $\ell_i$, each in $1 \ldots L$.

For the most part the prelude is a small part of the cost of each block but for concreteness we follow the layout suggested by Moffat and Turpin [18], and represent $s$, $n$, $m$ and $L$ as fixed-width binary numbers (32-bits in this work); the sub-alphabet description using the interpolative code (again, see Moffat and Turpin [18] for a description); and the code description as a sequence of codeword lengths in bits, in sub-alphabet order, storing in each case $L - \ell_i + 1$ in unary. Note that the codeword lengths are transmitted in symbol number order so that the decoder can reproduce the code employed by the encoder even in the face of ties on codeword lengths. Simply sending the number of codewords of each length is insufficient unless a complete permutation of the input symbols is also included in the prelude. Neglecting the construction cost of the prelude, and the transmission cost of getting it to the decoder, are recurring issues with work in this area.

## 3. EXPERIMENTAL FRAMEWORK

Experiments were conducted on a variety of prefix encoding and decoding routines incorporated into a single C++ application, in which the binary I/O routines are shared by all algorithms together with a range of other functions. The structure of the encoding and decoding routines is based largely on the implementation of Moffat and Turpin [5], although the program used for this investigation was created afresh to obtain the required versatility. The new system has almost identical compression effectiveness to that of Moffat and Turpin [5] (available at http://www.cs.mu.oz.au/~alistair/mr_coder/) when identical parameters choices are made, but the two programs are unable to decode each others' files.

### 3.1. Test data

Four input files were used in the experiments, described in Table I. Each of the files is based initially on text files describing *Wall Street Journal* articles, distributed as part of the TREC project (Harman [26]). The file *wsj20.bwt.mtf* represents the first 20 MB of text, processed using the Burrows–Wheeler transform, and then by a move-to-front transformation (Burrows and Wheeler [27]). The resulting data file is highly compressible (more than two thirds of the resultant symbols are '1's), and has a small alphabet. This file corresponds to what might be regarded as a classic coding situation over an ASCII-sized alphabet.

The next most compressible file is *wsj267.ind*. It represents the $d$-gaps in an inverted index for the 267 MB that makes up the first half of the *WSJ* TREC sub-collection. To generate this file, a document-level inverted index (covering all words and numbers) was constructed and then the set of inverted lists concatenated to make a single file. Each integer in the file represents the number of documents between occurrences in the collection of some word. This file has a very large alphabet but is still dominated by small values. It represents a lossy form of the original source text, in a form suitable for use in an text retrieval system (Witten *et al.* [28, ch. 3]).

The other two files represent two alternative ways of processing text via a lossless compression model. File *wsj267.seq.bwt.mtf* was generated by first transforming the same Disk 1 *WSJ* document

Table I. The test files and their properties. Self-information represents the average information content per symbol in the file, assuming that the entire file is a single message block. The final column shows the range of codeword lengths for a minimum-redundancy code, again when the whole of each message is processed as a single block.

| Name | Total symbols $m$ | Unique symbols $n$ | Self-information (bits per symbol) | Codes (bits) |
|---|---|---|---|---|
| *wsj20.bwt.mtf* | 20 971 525 | 122 | 2.13 | 1–23 |
| *wsj267.ind* | 41 389 467 | 111 605 | 6.76 | 2–25 |
| *wsj267.seq.bwt.mtf* | 63 317 166 | 73 489 | 8.16 | 2–26 |
| *wsj267.repair* | 19 254 349 | 319 757 | 17.63 | 10–24 |

collection to a stream of word numbers using a spaceless words model (see Moffat and Isal [29] for details), then applying the Burrows–Wheeler (BWT) transformation and finally a move-to-front (MTF) transformation. Due to the words transformation, the alphabet is much larger than when a purely character-based BWT is used and the average information content per symbol is also higher. Even so, approximately 25% of the symbols are '1's.

The final test file was generated using the RE-PAIR process of Larson and Moffat [30]. The same 267 MB of text was this time processed to find repeated strings using a grammar-based modeling stage that repeatedly replaces pairs of characters by a new symbol until no more pairs occur. This method generates a smaller file of integers, but over a larger alphabet, and with more information per symbol.

All of the input files are of interest, in that they are the output of good compression models that have, in different implementations, been coupled with a Huffman coders. The final column in Table I shows the range of Huffman codeword lengths allocated when each message is processed as a single monolithic block. The interesting feature here is that codes well in excess of 20 bits occur naturally, even in the file that represents character-level BWT data.

### 3.2. Hardware

The machine used for the experimental work was a 2.4 GHz Intel Xeon with 512 kB on-die L2 cache and 1 GB RAM running Debian GNU/Linux 3.0, and with local SCSI disks. The dual-processor nature was of no consequence as the test application is a single process. The operating system was Debian GNU/Linux with Kernel version 2.4.18 and the compiler was GNU g++ version 2.95.4, used with all major optimizations enabled.

All timings presented in the tables below are the average of ten consecutive runs, performed at a time when the machine was free of other user processes, and are the sum of system plus user CPU times.

### 3.3. A decoding baseline

The baseline decoding procedure, denoted CANONICAL and illustrated in Algorithm 1, closely follows Moffat and Turpin [18, p. 60], which in turn draws on Moffat and Turpin [5] and Zobel and Moffat [25].



DOI: 10.1002/spe

| Symbol | $s_1$ | $s_2$ | $s_3$ | $s_4$ | $s_5$ | $s_6$ |
|---|---|---|---|---|---|---|
| Codeword length ($l_i$) | 2 | 2 | 2 | 3 | 4 | 4 |
| Codeword | `00` | `01` | `10` | `110` | `1110` | `1111` |

(a) The code for the six symbol input described in Figure 1.

| $\ell$ | *base_sym*[$\ell$] | *base_cwd*[$\ell$] | *lj_limit*[$\ell$] |
|---|---|---|---|
| 1 | 0 | `0` | `0000` |
| 2 | 1 | `00` | `1100` |
| 3 | 4 | `110` | `1110` |
| 4 | 5 | `1110` | `10000` |

(b) The tables used by the canonical decoding process (Algorithm 1).

Figure 2. An example showing: (a) a canonical code for $n = 6$ symbols; (b) the table used for canonical decoding, assuming a look-ahead of $L = 4$ bits. The final entry in the *lj_limit* array is a sentinel, and is the value $2^L$ stored in $L + 1$ bits.

An $L$-bit variable *buffer* is used as a window into the pending part of the compressed message, together with a table that lists, for each codeword length $1 \le \ell \le L$, the values:

*base_sym*[$\ell$]—the first symbol number with codeword length $\ell$;
*base_cwd*[$\ell$]—the smallest codeword (as an integer) of length $\ell$;
*lj_limit*[$\ell$]—the first codeword of length greater than $\ell$, padded on the right with zeros to a total of $L$ bits.

The array *lj_limit* is used in conjunction with the $L$-bit window *buffer*—the smallest value of $\ell$ such that *buffer* < *lj_limit*[$\ell$] is exactly the length of the next codeword in the stream. The actual codeword is extracted as the leading $\ell$ bits of the buffer. The buffer is then shifted left by that amount, and replenished from the right by an injection of new bits from the input stream. For details of this process, the reader is referred to Moffat and Turpin [18]. An example showing the values of the three tables is shown in Figure 2.

Algorithm 1 assumes a bit-I/O function *input_bits(b)*, which obtains $b$ bits from the message stream. It also makes use of the bit manipulation functions *right_shift*() and *left_shift_and_mask*(). A further function—*leftmost_bits*()—is used in the methods presented below.

The decoder mapping is the only table in Algorithm 1 that is proportional in size to $n$, the number of symbols in the alphabet, rather than to $L$, the longest codeword length. Hence, just a single cache miss



DOI: 10.1002/spe

**Algorithm 1** (CANONICAL): baseline method for decoding a compressed block

create *base_sym*[], *base_cwd*[], *lj_limit*[], and *decode_map*[] from the prelude information.<br>
set $buffer = input\_bits(L)$.<br>
**for** $i = 1$ **to** $m$ **do**<br>
    /* perform a linear search to locate $\min\{\ell\}$ such that $buffer < lj\_limit[\ell]$ */<br>
    set $\ell = L_{min}$.<br>
    **while** $buffer \geq lj\_limit[\ell]$ **do**<br>
        set $\ell = \ell + 1$.<br>
    set $codeword = right\_shift(buffer, L - \ell)$.<br>
    set $sym = base\_sym[\ell] + (codeword - base\_cwd[\ell])$.<br>
    set $true\_sym = decode\_map[sym]$.<br>
    output *true_sym*, or add it to an output buffer.<br>
    set $buffer = left\_shift\_and\_mask(buffer, \ell) + input\_bits(\ell)$.

occurs per decoded symbol, even when $n$ is large. The rest of the algorithm is dominated by processor and I/O time. Given that I/O costs cannot be altered, the most significant component of Algorithm 1 is the search for $\ell$. During that search an average of $\ell_{\text{av}} \approx \mathcal{H}$ loop iterations are required, where $\mathcal{H}$ is the per-symbol self-information of the source message.

The single tuning variable for the baseline method is the size of the blocks used to partition the input, which is set by the encoding process. During encoding, a large block size requires more memory for data structures but reduces the number of times a code must be calculated. During decoding, memory requirements are largely independent of the block size and space consumption is primarily proportional to the alphabet size as there is no need for full-block buffering. The decoder mapping table is the dominant memory cost during decoding, and requires one word per distinct alphabet symbol. Total decoding time is greater for small block sizes than large, but the dependence is not as great as during encoding.

On the other hand, compression effectiveness is not directly connected to block size. As a general rule, a homogeneous message benefits from a large block size because the prelude overheads are amortized over more symbols. On the other hand, heterogeneous source messages tend to be compressed better with smaller blocks, in which the codes assigned are sensitive to localized frequency distributions and the cost of the additional preludes is compensated for by more precise codeword assignments.

The effect of different block sizes is shown in Table II. To generate this table, the baseline mechanism was used to compress the test sequences and note made of: the combined cost of all preludes, the combined cost of all codewords, the total time taken in the encoder to calculate codes and transmit the preludes of all blocks, and the total time taken to encode the sequences within the blocks using those codes. The fastest overall encoding time for each file is highlighted, as is the best overall compression rate (prelude plus codewords) for each file.

The principal outcome from these initial experiments is that too-small block sizes dramatically increase the percentage of time spent on code creation and prelude transmission, and adversely affect encoding speed. In terms of effectiveness, all of the four test files exhibit some level of localized variability, meaning that very large block sizes tend to erode compression effectiveness.



DOI: 10.1002/spe

Table II. Baseline encoding with different block sizes, showing the number of bits per symbol spent on preludes and actual codewords, the time required in the encoder to calculate the prefix codes and transmit the block preludes, the time taken to emit the actual codewords, and the total encoding time.

| | | Bits per symbol | | | Encoding time (seconds) | | |
|---|---|---|---|---|---|---|---|
| File | Block | Prelude | Encoding | Total | Setup | Encoding | Total |
| *wsj20.bwt.mtf* | $10^5$ | 0.00 | 2.10 | **2.10** | 0.01 | 0.51 | **0.53** |
| *wsj20.bwt.mtf* | $10^6$ | 0.00 | 2.13 | 2.13 | 0.00 | 0.53 | 0.54 |
| *wsj20.bwt.mtf* | $10^7$ | 0.00 | 2.18 | 2.18 | 0.00 | 0.53 | 0.54 |
| *wsj267.ind* | $10^5$ | 0.23 | 6.39 | **6.63** | 0.86 | 2.03 | 2.97 |
| *wsj267.ind* | $10^6$ | 0.09 | 6.66 | 6.75 | 0.34 | 1.36 | 1.73 |
| *wsj267.ind* | $10^7$ | 0.03 | 6.77 | 6.80 | 0.13 | 1.31 | **1.47** |
| *wsj267.seq.bwt.mtf* | $10^5$ | 0.52 | 7.81 | 8.33 | 2.86 | 3.49 | 6.46 |
| *wsj267.seq.bwt.mtf* | $10^6$ | 0.14 | 8.06 | **8.20** | 0.99 | 2.75 | 3.81 |
| *wsj267.seq.bwt.mtf* | $10^7$ | 0.03 | 8.17 | **8.20** | 0.20 | 2.69 | **2.93** |
| *wsj267.repair* | $10^5$ | 3.19 | 15.21 | 18.40 | 5.01 | 5.54 | 10.63 |
| *wsj267.repair* | $10^6$ | 0.81 | 16.56 | **17.38** | 2.20 | 5.43 | 7.66 |
| *wsj267.repair* | $10^7$ | 0.17 | 17.46 | 17.63 | 0.37 | 5.65 | **6.03** |

As a compromise between speed, compression effectiveness and memory use during encoding, and in order to limit the number of variables, a fixed block size of $m = 10^6$ was chosen for all of the subsequent experiments. At this block size, all four test files are represented at very close to the whole-of-message self-information rates given in the fourth column of Table I.

## 4. IMPROVEMENTS

As noted, we took the CANONICAL implementation of Algorithm 1 as the baseline for our experimentation. Moffat and Turpin [5], the developers of this approach, provide experimental results that compare Algorithm 1 to the other methods described in Section 2. In this section we consider enhancements to Algorithm 1, starting with one suggested by Moffat and Turpin [5].

### 4.1. Adding a 'start' array

Steps 5–7 in Algorithm 1 perform a linear search over possible values of $\ell$, starting with the smallest such value for the code in question, $L_{\min}$. To accelerate the search, Moffat and Turpin [5] suggest that a precomputed table be used to more precisely initialize the control variable $\ell$. The table, called *start*[] is indexed by an integer representing the first $b$ bits of *buffer*. The value of each entry of *start*[] is the first value of $\ell$ that is consistent with the $b$-bit prefix. For the example code shown in Figure 2, and assuming that $b = 2$, the four entries in the *start* array would be 2, 2, 2 and 3, respectively, since the



DOI: 10.1002/spe

**Algorithm 2** (START): use of a *start* array of size $2^b$ to accelerate the linear search

1: replace step 5 of Algorithm 1 by:
2: set $p = leftmost\_bits(buffer, b)$.
3: set $\ell = start[p]$.

shortest codewords that start with any of `00`, `01` and `10` are of length 2 and the shortest codeword that starts with `11` is of length 3. This method is denoted as START, and is summarized in Algorithm 2.

For typical values of $b$, the additional memory requirement of the *start* array is insignificant—just 256 bytes, for example, when $b = 8$. Moreover, there is no need for the *start* array to be made larger. When $b = 8$ all codewords of $\ell_i = 8$ bits or shorter are identified without further looping, and while longer codewords still require a limited amount of searching, there is a considerable reduction in the number of iterations required.

Using a range of input files, Moffat and Turpin [5] demonstrated that START, with $b = 8$ bits, almost entirely circumvents the cost of the linear search and showed that START is faster than the table-based finite machine approach of Choueka *et al.* [19] and the tree-table method of Hashemian [23].

### 4.2. The 'extended' method for decoding

The START method quickly determines the leading codeword, and hence the leading symbol, in the input buffer. However, it identifies only one codeword at a time, even if the codewords are short and the buffer contains multiple codewords. An additional cost is that after each codeword is identified a two-step translation is required to determine the symbol associated with the codeword.

A well-known technique for improving decoding speed is to extend the coding tables so that (potentially) multiple codewords can be identified and output in each operation. This concept has been suggested, in various forms, by Choueka *et al.* [19], Sieminski [21], Hashemian [23], Nekritch [7], and Milidiú *et al.* [8]. These methods make use of additional lookup tables, here called *extended tables*, which contain information about the fully decoded symbols associated with various bit strings. Some of the suggestions utilize a hierarchy of tables as a means to trade off speed versus space; here we consider a single extended decode table so as to obtain maximal decoding rates. The multi-symbol method described here is simply called the EXTENDED method.

In the simplest implementation of this approach, the *buffer* contains $x \geq L$ bits and all $x$ bits are used to index the decoding table. The table thus contains $2^x$ entries, each of which consists of a list of symbols to be written when that bit pattern appears in the buffer plus an integer indicating how many of the $x$ bits in the buffer are consumed by those symbols.

As an example, consider again the code shown in Figure 2 (for which $L = 4$), and suppose that the *buffer* of pending bits always contains $x = 5$ bits. The corresponding extended table contains 32 entries, with the entry for the bit-string `01001` (integer 17), for example, being the tuple $([s_2, s_1], 4)$, indicating that those two symbols should be output, and only $x - 4 = 1$ bits retained. Similarly, the entry for `11011` (integer 27) would be $([s_3], 3)$. The number of symbols output at each step is approximately proportional to $x/\ell_{av}$, where $\ell_{av}$ is the average codeword length.

To make this technique effective, the lists of fully decoded symbol numbers are concatenated into a single string, and accessed via offsets from the start of it. Figure 3 shows part of the extended



DOI: 10.1002/spe

| *Bit-stream* | *nbits* | *offset* | *nsyms* |
|---|---|---|---|
| `00000` | 4 | 0 | 2 |
| `00001` | 4 | 0 | 2 |
| `00010` | 4 | 2 | 2 |
| `00011` | 4 | 2 | 2 |
| `00100` | 4 | 4 | 2 |
| `00101` | 4 | 4 | 2 |
| `00110` | 5 | 6 | 2 |
| `00111` | 2 | 8 | 1 |
| ... | | | |
| `11111` | 4 | 35 | 1 |

(a) Part of the extended decode table.

| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | $\cdots$ | 35 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| symbol | $s_1$ | $s_1$ | $s_1$ | $s_2$ | $s_1$ | $s_3$ | $s_1$ | $s_4$ | $s_1$ | $\ldots$ | $s_6$ |

(b) Part of the string of symbols accessed using the offsets supplied by the table.

Figure 3. The extended decode table for the code of Figure 2 when $x = 5$ bits are used to index the table: (a) some of the $2^x$ entries in the extended decode table; (b) part of the concatenated strings listing the decoded symbols.

decoding table for the code given in Figure 2. As there is commonality of output symbols when not all $x$ bits are being consumed, it is beneficial to allow multiple entries in the extended table to indicate the same substring. Other overlaps might also be identified and exploited to further compact the string of Figure 3(b). For example, whenever a suffix of one string of symbols is a prefix of the next string the repetition can be eliminated. We have not included that optimization in our implementation.

The total number of fully decoded symbols required in the concatenated string is approximately $x2^x/\ell_{\text{av}}$, as the set of all $x$-bit strings is similar to a random string of $x2^x$ bits. The decoder can build the concatenated string using a resizeable array or can calculate $x2^x/\ell_{\text{av}}$, add on a margin for error and then allocate a single array. Alternatively, the coder can explicitly calculate the length of the required string and include it as an additional integer in the prelude.

Previous work has suggested that the extended decode table be used as a stand-alone technique and structured so that every step produces at least one symbol. For small alphabets this may sometimes be an achievable requirement, with perhaps $x = 16$ when $L \le 16$. However, in general, assuming (a minimum of) $2^L$ table entries is not tenable, even for ASCII-sized alphabets and $n = 256$. For example, all of our four test files have maximum codeword lengths of $L = 23$ or larger when taken as a single block (Table I), including the BWT-MTF file. Building a table of $2^{23}$ entries and a



DOI: 10.1002/spe

**Algorithm 3** (EXTENDED): Decoding using an $x$-bit extended table

/* Create the decode table */
allocate arrays *nsyms*, *nbits*, *offset*, and *start*[] to hold $2^x$ elements each.
allocate array *symbol*[] of a size sufficient to store the concatenated strings.
decode the prelude, and construct the canonical decoding tables.
**for** $v = 0$ **to** $2^x - 1$ **do**
    use the canonical decoding tables to determine the values *nsyms*[$v$], *nbits*[$v$], *offset*[$v$], and *start*[$v$], where $v$ is an $x$-bit string.
    **if** the output symbols associated with $v$ differ from the symbols associated with $v - 1$ **then**
        store the output symbols associated with $v$ in the array *symbols*[].

/* Do the actual decoding to reconstruct a block of $m$ symbols */
set *buffer* = *input_bits*(max{$x$, $L$}).
set *ndecoded* = 0.
**while** *ndecoded* $< m$ **do**
    set $p$ = *leftmost_bits*(*buffer*, $x$).
    **if** *nsyms*[$p$] $\geq 1$ **and** *ndecoded* + *nsyms*[$p$] $\leq m$ **then**
        output the *nsyms*[$p$] symbol values which appear at *offset*[$p$].
        set *buffer* = *left_shift_and_mask*(*buffer*, *nbits*[$p$]) + *input_bits*(*nbits*[$p$]).
        set *ndecoded* = *ndecoded* + *nsyms*[$p$].
    **else**
        use Algorithm 2 to decode the leading codeword in *buffer*, including updating *buffer*.
        set *ndecoded* = *ndecoded* + 1.

symbol string of nearly $23 \times 2^{23}/2.13 \approx 90$ million symbols would be both costly of memory space and time-consuming to initialize at decode time.

The key change we propose in our implementation of the EXTENDED method is to build a decoding table that is indexed by a prefix of $x$ bits of the current *buffer* variable, rather than using all of the $L$ bits that it contains. As not every string of $x < L$ bits can represent a codeword, it is necessary to also introduce a fall-back system for the cases where no complete codeword appears in the $x$ bit string. The most satisfactory method is to revert to a seeded linear search, following the approach shown in Algorithm 2. In this case the linear search can be started at $\ell = x$. Alternatively, an explicit *start* table can be maintained, perhaps as another field in the extended decoding table, assuming $b = x$. That is, the extended decoding table is used as an accelerator for the START table, rather than a replacement.

Algorithm 3 gives details of the processes involved. One important point to note is that a complete computation of the extended table is required before any symbols can be decoded, as it is not practical for the extended table to be included verbatim in the prelude. When $x$ is large, the cost of this pre-computation is a significant part of total decoding time.

Care is needed at the end of blocks of data to avoid inferring the existence of symbols that were not in fact coded. The simplest way of avoiding this issue is for all symbols within $L$ bits of the last bit of the compressed message to be handled one at a time, by falling back to either Algorithm 1 or Algorithm 3.



DOI: 10.1002/spe

Table III. Decoding methods based on canonical codes. Method 'Canon' is the CANONICAL baseline method; 'Start' improves upon it by accelerating the search for the next codeword (method START with $b = 8$); and the variants Ext-$x$ use an extended table (method EXTENDED) with keys of $x$ bits, so that, for example, the entries for Ext-15 use a table containing 32 768 rows. The running time of the decoder is given as a preprocessing time (including the cost of processing the preludes and creating any necessary decoding tables), a symbol decoding time, and a total time including all costs. A block size of $m = 1\,000\,000$ symbols was used in all experiments. For each input file, the fastest decoding method (of those shown in this table) is indicated in bold.

| | | Decoding time (seconds) | | | |
|---|---|---|---|---|---|
| File | Approach | Setup | Decoding | Total | Memory (MB) |
| *wsj20.bwt.mtf* | Canon | 0.00 | 0.74 | 0.74 | 0.02 |
| *wsj20.bwt.mtf* | Start | 0.00 | 0.61 | 0.61 | 0.02 |
| *wsj20.bwt.mtf* | Ext-10 | 0.00 | 0.26 | **0.26** | 0.05 |
| *wsj20.bwt.mtf* | Ext-15 | 0.21 | 0.35 | 0.56 | 1.38 |
| *wsj20.bwt.mtf* | Ext-20 | 8.57 | 0.66 | 9.39 | 55.65 |
| *wsj267.ind* | Canon | 0.08 | 1.76 | 1.85 | 0.33 |
| *wsj267.ind* | Start | 0.10 | 1.47 | **1.56** | 0.33 |
| *wsj267.ind* | Ext-10 | 0.27 | 1.32 | 1.59 | 0.35 |
| *wsj267.ind* | Ext-15 | 0.47 | 1.42 | 1.89 | 0.99 |
| *wsj267.ind* | Ext-20 | 8.38 | 4.14 | 12.69 | 33.94 |
| *wsj267.seq.bwt.mtf* | Canon | 0.27 | 2.90 | 3.18 | 0.56 |
| *wsj267.seq.bwt.mtf* | Start | 0.24 | 2.35 | **2.59** | 0.56 |
| *wsj267.seq.bwt.mtf* | Ext-10 | 0.86 | 2.50 | 3.37 | 0.57 |
| *wsj267.seq.bwt.mtf* | Ext-15 | 1.14 | 3.29 | 4.46 | 1.11 |
| *wsj267.seq.bwt.mtf* | Ext-20 | 13.37 | 8.62 | 22.20 | 34.71 |
| *wsj267.repair* | Canon | 0.57 | 1.58 | 2.16 | 2.80 |
| *wsj267.repair* | Start | 0.58 | 1.39 | **1.98** | 2.80 |
| *wsj267.repair* | Ext-10 | 1.76 | 1.78 | 3.55 | 2.81 |
| *wsj267.repair* | Ext-15 | 1.78 | 2.68 | 4.48 | 3.19 |
| *wsj267.repair* | Ext-20 | 4.14 | 4.74 | 8.93 | 16.60 |

Similar considerations apply when the output of the decoder is being formed into fixed-length blocks for output.

Table III shows the time and memory usage of the decoding variants described thus far, with the decoding cost broken into two components: the time spent preparing to decode, including the time reading the prelude and creating the necessary tables; and then the time spent actually decoding. Three configurations for the EXTENDED method are shown in Table III, to illustrate the dependence of table creation and decoding speed on the size of the extended table. The final column shows the memory space required during decoding, measured in megabytes.

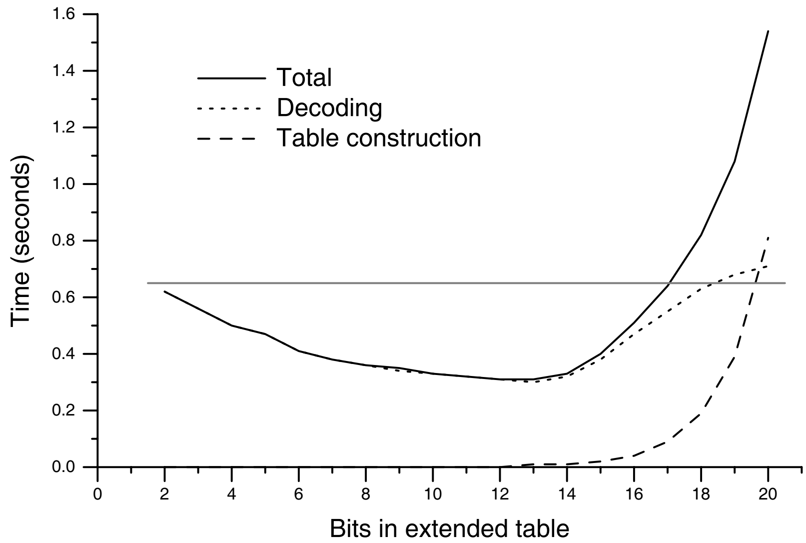



Figure 4. Component costs for the EXTENDED decoding method, applied to file *wsj20.bwt.mtf* with a block size of 10 485 760 symbols. The three curves represent the time spent building the extended table, the cost of using it to decode symbols, and the total time spent decoding. The horizontal line shows the time taken to decode using the START method, for the same block size.

For all of the four test files the START method is superior to the basic canonical decoding system, confirming the claims of Moffat and Turpin [5]. The usefulness of the EXTENDED approach is less clear, and it is faster on only one of the inputs (file *wsj20.bwt.mtf*). The distinguishing feature of that file is that the average code word length is short, meaning that even with a 10-bit table multiple symbols are decoded with most table accesses. On the other files, the EXTENDED method offers no advantage.

There are three reasons for this pattern of behavior. First, the extended tables are naturally suited to inputs that exhibit a low $\ell_{av}$, since more symbols are on average emitted at each cycle of table lookup. Second, creation of big tables is expensive. Building an $x$-bit table requires that $2^x$ table entries be created, and involves the implicit decompression of $O(x2^x)$ bits. When $x = 20$, for example, each of *nsyms*, *nbits* and *offset* contain more than a million entries. Third, there are caching effects that mitigate against the use of large tables, not even taking into account the cost of initializing them. When $x = 20$, the tables exceed the cache size of the hardware used for these experiments meaning that accesses to the tables involve accesses to main memory. Moreover, the pattern of accesses is essentially random. That is, the great majority of table lookups cause cache misses once the table data structures exceed the cache size. This third effect is why decoding with a large table is slower than decoding with a smaller table, even after the tables have been constructed.

With larger blocks, the table creation costs can be amortized over more symbols. Even so, small tables work best, and throughput is high only when the corresponding $\ell_{av}$ is a small fraction of



DOI: 10.1002/spe

the width of the table. The relationship of the various portions of the decoder running time to the parameter $x$, for the *wsj20.bwt.mtf* data file, and processed as two blocks of $10 \times 2^{20} \approx 10^7$ symbols, is shown in Figure 4. The overall decoding cost is again decomposed into the time spent on prelude handling plus table creation and the time spent on codeword processing. As expected, the cost of creating the extended table increases exponentially with $x$, the bit-width of the table index. The relationship between decoding time and $x$ is more complex: at first the decoding time decreases as $x$ gets larger, since more symbols are produced per lookup. However, the cost of cache misses eventually overcomes that gain, and there is a pronounced knee in the graph once the cache size of the hardware is approached. For large values of $x$, the benefit of multi-symbol decoding is lost and the START method (shown by the horizontal line) is faster.

When $\ell_{\mathrm{av}}$ is larger—as is the case with the other three test files, plotted in graphs that are not shown here—similar relationships are observed but with the START line always less than the EXTENDED curve.

### 4.3. Extended decoding with length-restricted codes

Milidiú *et al*. [8] suggest the use of length-restricted codes in conjunction with extended decoding, arguing that a length-restricted code reduces the size the extended decode table created using $x = L$ bits, where $L$ is an enforced maximum codeword length. However, as discussed in the previous section, there is no fundamental requirement that $x \geq L$, as fall-through to a simpler mechanism is possible whenever the extended table produces no symbols and is necessary in any case to gracefully handle block boundaries. In other words, the argument of Milidiú *et al.* [8] is based on an unnecessary premise.

In fact, the use of a length-restricted code compared to an unrestricted code gives rise to two competing effects in the decoder. The first is that the number of table entries producing no symbols is reduced. The second is that the value of $\ell_{\mathrm{av}}$ increases through application of a length-restriction, and thus the average number of symbols produced per table access is reduced. The second of these is the more important, and tends to cause length-restricted codes to decode slightly more slowly than unrestricted codes. Another way of looking at this effect to simply note that more bits need to be decoded, and so more table accesses are used to decode them.

A further complication with length-restricted codes is that the length limit is controlled by the alphabet size and for three of the four applications considered here, tight limits are simply not possible. For the four test files (Table I), the minimum length restrictions possible are $L = 7$, $L = 17$, $L = 17$ and $L = 19$. From this point of view, length-restriction is only useful if no backup START mechanism is being used to handle long codewords, and if the average codeword length is short anyway.

## 5. RESTRICTED DECODING STRUCTURES

In the case of length-restricted codes, the encoder is restricted but the decoder does not change. Another possible way of boosting decoding throughput is to restrict the form of the code used, and make use of an altered decoder specialized to that restricted structure. In particular, when general canonical codes are being decoded, determining the length of each codeword in the compressed message stream is an important step in decoding each symbol. In this section we consider ways of accelerating that component of the running time.



DOI: 10.1002/spe

### 5.1. Approximate codes

One mechanism that has been suggested for fast decoding is to use *byte-aligned* codes, particularly for monotonic-decreasing probability distributions (Scholer *et al*. [14]). For example, the $(S, C)$-dense codes of Brisaboa *et al*. [15] sort the symbol set for each block into decreasing frequency order and then apply a byte code controlled by a single parameter. Byte-based minimum-redundancy codes have also proved attractive in applications where searching may be required (de Moura *et al*. [13]).

Also designed primarily for monotonic-decreasing probability distributions is the *word-aligned* coding scheme of Anh and Moffat [17]. This approach uses clusters of binary codewords, all of the same bit length, to represent the input stream. Each cluster is sized so that it fits a single machine word, meaning that each 32-bit word contains a set of binary codes all the same bit-length. Fast decoding is then possible because most of the decoding steps involve nothing more than extracting fixed-width binary codewords, without any crossing of word boundaries.

One of the key advantages of the word-aligned code is that it is adjusts over quite short distances to local variations in symbol probabilities. For example, runs of small values are coded compactly in just a few bits each. Anh and Moffat [17] give results that show this method to be as fast as byte-aligned codes in terms of decoding throughput and, for the most part, to provide better compression.

As it is not dependent on actual symbol probabilities, the word-aligned code can be ineffective when the symbol probability distribution is not monotonic decreasing. In this case a mapping can be used, in the same way as is required when canonical minimum-redundancy codes are implemented. In the experiments described below no mapping is used, and the figures given for the CARRY mechanism reflect the raw performance of the word-aligned code.

### 5.2. The $K$-flat code structure

Another mechanism for simplified prefix codes, called $K$-flat codes, has been developed recently (Liddell [11]). The basic $K$-flat code structure is illustrated by Figure 5. A $K$-flat code tree has two important properties: (1) it has a binary upper section of $K = 2^k$ nodes, for some integer $k$; and (2) each of the nodes at depth $k$ is the root of a strictly binary sub-tree, in which all leaves are at the same depth. That is, the first $k$ bits of every codeword completely specify the total number of bits in that codeword and can be thought of as a *sub-tree selector* that picks one of the $K$ binary sub-trees. Figure 5 shows an example $K$-flat tree in which $n = 6$, $k = 2$ and $K = 4$. The tree can be characterized by the sizes of the four sub-trees and described as $\{1, 1, 1, 3\}$.

The first two restrictions encapsulate the essential nature of $K$-flat trees. To reduce the number of isomorphic trees for a given input sequence two additional constraints are helpful, so as to force the construction of a canonical $K$-flat arrangement: (3) across the $K$ sub-trees, the depths are non-decreasing; and (4) the first $K - 1$ sub-trees are complete, and only the last sub-tree may be partially full. The tree shown in Figure 5 is one of two (for $n = 6$ and $K = 4$) that fit the full set of four constraints—the other is described by the arrangement $\{1, 1, 2, 2\}$.

Methods for calculating both minimum-redundancy and approximate $K$-flat codes are available (Liddell [11]). The techniques for creating minimum-redundancy $K$-flat codes are based on dynamic programming, with a basic implementation requiring $O(Kn^2)$ time and $O(Kn)$ space. Liddell [11] shows that the resource requirements can be reduced to $O(Kn \log n)$ time, and $O(n \log n)$ space.



DOI: 10.1002/spe

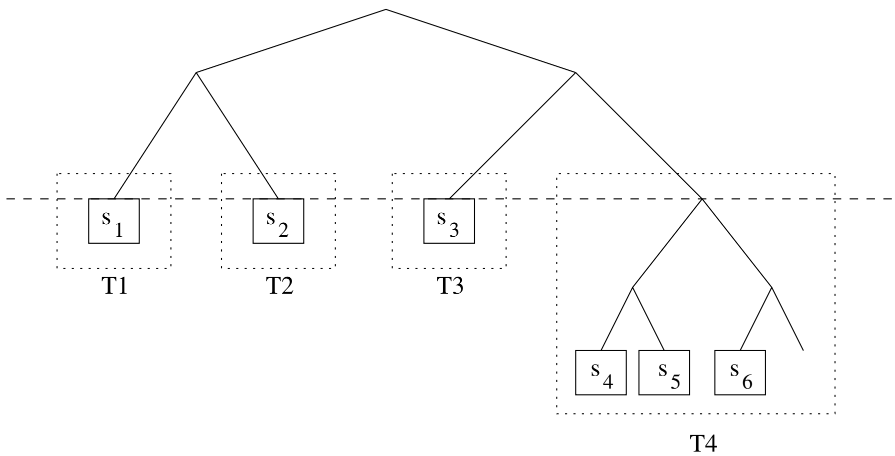



Figure 5. The basic structure of a $K$-flat code, showing one possible code arrangement for six symbols over $K = 4$ sub-trees. The codeword for each symbol consists of a 2-bit binary prefix, followed by a binary suffix. For example, each symbol in $T_4$ has a codeword of length $2 + 2 = 4$ bits. In this arrangement one potential codeword is unused.

Approximate $K$-flat codes offer a faster construction process, with only slightly lower compression effectiveness, and are the main focus in this experimental evaluation.

### 5.3. Calculating approximate $K$-flat codes

Given that the selector is a binary code, each of the sub-trees should have approximately the same sum of leaf probabilities. This observation means that one simple way to determine an approximate $K$-flat code is to adopt a Fano-like approach and split the sorted source probability distribution into $K$ sub-parts each containing a number of symbols that is a power of two (except for the last tree), and with each sub-part of approximately equal weight. That is, starting with an $n$-symbol decreasing probability-sorted alphabet, we seek a $K$-way partition such that the first $K - 1$ groups each contain a power of two elements: the depth of the binary tree for each partition is at least as large as for the previous one, and the sum of the weights in each partition is approximately the same.

Algorithm 4 describes this process in more detail. The target sum for the first tree is $\sum_{1 \le i \le n} f_i / K$. A first tree size is chosen, selecting the alternative that produces the smallest deviation in size from that target value. Once a first tree has been identified and removed from the front of the symbol list the process repeats, with the remaining elements partitioned into (now) $K - 1$ equal-weight segments. To ensure the sequence of tree sizes is non-decreasing, the search for each tree commences with the size used for the previous search (variable *min_d*), and against an upper limit *max_d* on the size of the tree that can safely be removed at each stage. Each cycle of search-then-remove takes at most $O(\log n)$ time, and there are $K$ such steps. Preprocessing at step 1 to generate the array $F$ of cumulative weights takes a further $O(n)$ time. Ignatchenko [31] describes a similar procedure for a restricted case.



DOI: 10.1002/spe

**Algorithm 4**: Creating an approximate $K$-flat code

**for** $i = 0$ **to** $n$ **do**<br>
  set $F[i] = \sum_{1 \le x \le i} f_i$.<br>
set $min\_d = 1$, and set $curr = 0$.<br>
**for** $t = 1$ **to** $K - 1$ **do**<br>
  set $target = (F[n] - F[curr])/(K - t + 1)$.<br>
  set $max\_d = min\_d$.<br>
  **while** $2^{max\_d+1} \times (K - t) + 2^{max\_d} + 1 \le (n - curr)$ **do**<br>
    set $max\_d = max\_d + 1$.<br>
  set $best\_d = min\_d$, and set $best\_diff = |F[curr + 2^{min\_d}] - F[curr] - target|$.<br>
  **for** $d = min\_d + 1$ **to** $max\_d$ **do**<br>
    set $diff = |F[curr + 2^d] - F[curr] - target|$.<br>
    **if** $diff < best\_diff$ **then**<br>
      set $best\_d = d$, and $best\_diff = diff$.<br>
  set $depth[t] = best\_d$, and set $curr = curr + 2^{best\_d}$, and set $min\_d = best\_d$.<br>
set $depth[K] = \lceil \log_2(n - curr) \rceil$.<br>
output the $K$ tree depths $depth[1]$ to $depth[K]$.

The $K$-flat tree shown in Figure 5 is that which results when Algorithm 4 is applied to the probability distribution used earlier in Figure 1, namely $\{0.30, 0.26, 0.20, 0.15, 0.05, 0.04\}$.

The restrictions implied by the $K$-flat code structure allow savings in the prelude cost. Once the set of sub-tree sizes has been transmitted to the decoder, symbols can be associated with a sub-tree number rather than a codeword length. Coding tree numbers $t_i$ as $unary(K - t_i + 1)$—appropriate because the last trees are the largest—allows slightly more compact preludes than does coding $unary(L - \ell_i + 1)$.

### 5.4. Encoding costs

The first set of experiments performed with $K$-flat codes tested the compression effectiveness and resource requirements of the optimal and approximate $K$-flat encoders, to verify that the approximate codes generated by Algorithm 4 are reasonable. Table IV lists the results for the file *wsj267.seq.bwt.mtf*. For small values of $K$, the cost of optimal code calculation is acceptable. However, for large values, it becomes excessive and compared with the approximate calculation process, the gain in compression effectiveness is small.

Across the four files, three conclusions were drawn from these experiments. First, for inputs with low average codeword lengths, $K$-flat codes are ineffective, whereas for inputs with high entropy, such as *wsj267.seq.bwt.mtf* and *wsj267.repair*, the codes produced are of relatively high quality and are less than 10% redundant. Second, when $K$ is large, the cost of finding optimal $K$-flat codes becomes high, especially when the alphabet is large. Third, the cost of producing an approximate $K$-flat code is always small, and it is plausible to evaluate all viable values of $K$ and choose the best result.



DOI: 10.1002/spe

Table IV. Compression effectiveness and encoding speed for optimal and approximate $K$-flat coding methods when applied to the file *wsj267.seq.bwt.mtf*. The two determinants of the $K$-flat code construction process are $K$ and whether optimal or approximate codes are developed. The fastest encode time and the best compression ratio are shown in bold. Compression effectiveness can be compared with the 8.16 bits per symbol self-information for this file (Table I), and the 8.20 bits per symbol obtained by a minimum-redundancy code (Table II).

| | Parameters | | Bits per | Encoding time (seconds) | | |
|---|---|---|---|---|---|---|
| File | Mode | $K$ | Symbol | Setup | Encoding | Total |
| *wsj267.seq.bwt.mtf* | Approx | 2 | 11.04 | 0.42 | 2.76 | 3.23 |
| *wsj267.seq.bwt.mtf* | Optimal | 2 | 10.64 | 0.89 | 2.72 | 3.66 |
| *wsj267.seq.bwt.mtf* | Approx | 4 | 8.99 | 0.43 | 2.75 | **3.22** |
| *wsj267.seq.bwt.mtf* | Optimal | 4 | 8.87 | 1.67 | 2.76 | 4.47 |
| *wsj267.seq.bwt.mtf* | Approx | 8 | 8.51 | 0.42 | 2.76 | 3.23 |
| *wsj267.seq.bwt.mtf* | Optimal | 8 | **8.45** | 5.27 | 2.76 | 8.09 |
| *wsj267.seq.bwt.mtf* | Approx | 16 | 8.56 | 0.41 | 2.79 | 3.25 |
| *wsj267.seq.bwt.mtf* | Optimal | 16 | 8.56 | 17.68 | 2.84 | 20.55 |
| *wsj267.seq.bwt.mtf* | Approx | 32 | 8.86 | 0.43 | 2.77 | 3.27 |
| *wsj267.seq.bwt.mtf* | Optimal | 32 | 8.85 | 43.16 | 2.81 | 46.01 |

### 5.5. Decoding $K$-flat codes

The procedure for decoding a $K$-flat-encoded message is straightforward. The decoder maintains an $L$-bit buffer, as occurs with the Huffman decoding procedures. At each iteration, the first $k$ bits of the buffer are extracted and used to identify a sub-tree. A table of sub-tree sizes is then consulted to determine the number of bits required to identify a leaf within that sub-tree. Once those bits are read, the codeword is completely determined. The complete procedure for $K$-flat decoding is illustrated in Algorithm 5. The principal difference is the lack of a search procedure. However, many other steps remain in each iteration, and so the total gain which can be realized by employing $K$-flat codes is unclear. In particular, Algorithm 5 only decodes one symbol per iteration, so may not be competitive with the extended decode method on inputs with small $\ell_{\mathrm{av}}$.

## 6. COMPARISON

To compare the usefulness of the $K$-flat approach with conventional minimum-redundancy codes, we executed them in the same test harness as used previously. The results of those experiments are shown in Table V. For each of the four test files, the effectiveness of a 'best-$K$' code for that file using the approximate construction process is given, together with the speed at which it can be decoded using Algorithm 5. Also shown is the effectiveness of the minimum-redundancy codes described in Table II, so that the loss in effectiveness arising from the $K$-flat structure can be gauged together with the speed at which the those codes can be decoded using the START and EXTENDED mechanisms (Table III).



DOI: 10.1002/spe

**Algorithm 5** (K-FLAT): decoding a $K$-flat encoded block with parameters $K$ and $k$

calculate the size of each sub-tree, $size[t]$ for $1 \leq t \leq K$, by accumulating the sub-tree information from the prelude; and create table $base\_sym[t]$ to represent the first ordinal symbol number in each tree.
create $decode\_map[]$ to link ordinal symbol numbers to real symbol numbers.
**for** $t = 1$ **to** $K$ **do**
  set $suffix\_length[t] = \lceil \log_2 size[t] \rceil$.
set $buffer = input\_bits(L)$.
**for** $i = 1$ **to** $m$ **do**
  set $t = leftmost\_bits(buffer, k)$ and $sufflen = suffix\_length[t]$.
  set $codeword = leftmost\_bits(buffer, k + sufflen)$.
  set $offset = rightmost\_bits(codeword, sufflen)$.
  set $sym = base\_sym[t] + offset$.
  set $true\_sym = decode\_map[sym]$.
  write $true\_sym$, or add it to an output buffer.
  set $buffer = left\_shift\_and\_mask(buffer, k + sufflen) + input\_bits(k + sufflen)$.

The final coding mechanism shown is the CARRY method of Anh and Moffat [17], which packs binary codewords into 32-bit machine words as economically as possible subject to the constraint that all of the codes in each word have the same bit-length. As before, the best value in each category is presented in bold.

Table V confirms that on three of the four test files the $K$-flat codes can be decoded quickly compared to minimum-redundancy codes. The exception is the file *wsj20.bwt.mtf*, which has a low average cost per symbol and is (as discussed above) the file that is most amenable to the EXTENDED decoding approach. On this file the $K$-flat approach outperforms the START method, but not the EXTENDED method. However, none of the three coders that explicitly make use of symbol probabilities are as fast as the simpler CARRY method. It benefits both from the simple binary numbers employed, and also from the fact that it does not make use of a symbol mapping and hence does not suffer any cache misses at all during decoding. On the other hand, CARRY provides relatively poor compression effectiveness on the two files with the highest entropy.

Based on the various experiments we have undertaken, and as a concrete outcome of this project, we thus recommend the following.

- When minimum-redundancy codes are being used, the START mechanism should be the minimum approach implemented in order to obtain fast decoding speed on a wide range of input types.
- If compression effectiveness is the dominant concern, and decoding throughput a lesser concern, minimum-redundancy codes should be used in preference to $K$-flat codes, and in preference to the CARRY method.
- If minimum-redundancy codes are being used, and the input files exhibit an $\ell_{\mathrm{av}}$ value less than four or five, the EXTENDED decoding method is a useful accelerator. The table size should not be allowed to exceed 10 to 12 bits, meaning that it needs to be backed up by an underlying START implementation to handle long codewords.



DOI: 10.1002/spe

Table V. Performance comparison: four different decoding methods on the four test files. The Start and Ext-10 methods are two different ways of decoding proper minimum-redundancy codes, the $K$-flat approach requires a specially constructed code that is not minimum-redundancy but allows fast decoding, and the Carry-12 mechanism is a word-aligned binary coding mechanism that uses only local properties of the input stream (Anh and Moffat [17]).

| File | Method | Bits per symbol | Decode time (seconds) |
|---|---|---|---|
| *wsj20.bwt.mtf* | Start | **2.13** | 0.61 |
| *wsj20.bwt.mtf* | Ext-10 | **2.13** | 0.26 |
| *wsj20.bwt.mtf* | $K$-flat-4 | 2.90 | 0.49 |
| *wsj20.bwt.mtf* | Carry-12 | 2.76 | **0.15** |
| *wsj267.ind* | Start | **6.66** | 1.56 |
| *wsj267.ind* | Ext-10 | **6.66** | 1.59 |
| *wsj267.ind* | $K$-flat-8 | 7.19 | 1.20 |
| *wsj267.ind* | Carry-12 | 7.01 | **0.30** |
| *wsj267.seq.bwt.mtf* | Start | **8.20** | 2.59 |
| *wsj267.seq.bwt.mtf* | Ext-10 | **8.20** | 3.37 |
| *wsj267.seq.bwt.mtf* | $K$-flat-8 | 8.51 | 2.15 |
| *wsj267.seq.bwt.mtf* | Carry-12 | 10.81 | **0.58** |
| *wsj267.repair* | Start | **17.38** | 1.98 |
| *wsj267.repair* | Ext-10 | **17.38** | 3.55 |
| *wsj267.repair* | $K$-flat-8 | 17.69 | 1.82 |
| *wsj267.repair* | Carry-12 | 28.65 | **0.24** |

- If decoding speed is important, and the input sequences are over a large alphabet and have a high $\ell_{\mathrm{av}}$, then $K$-flat codes may be appropriate. For typical values of $K$, approximate code construction is several times faster than exact code construction.
- If decoding speed is of paramount importance, even at the risk of significant loss of compression effectiveness, the CARRY method should be used.

Also worth noting is that when compression effectiveness completely outweighs speed, then arithmetic codes (see Moffat and Turpin [18] for a description) should be considered, especially if there is localized variation within the message sequences that can be exploited. On file *wsj20.bwt.mtf*, a structured arithmetic coder obtained 1.75 bits per symbol, but took 4.81 seconds to decode.

A final conclusion of this work is that practical decoding speed cannot be easily estimated from algorithmic analysis. In particular, the interplay between table creation cost, cache behavior and actual decoding processes is sufficiently complex that it is very hard to predict decoding relativities without actually implementing and testing the different methods. That is what we have done in this paper.

**ACKNOWLEDGEMENTS**

This work was supported by the Australian Research Council, by the ARC Center for Perceptive and Intelligent Machines in Complex Environments, and by the NICTA Victoria Laboratory. The CARRY software used in the experiments was written by Vo Ngoc Anh (University of Melbourne).

**REFERENCES**

1. Huffman DA. A method for the construction of minimum-redundancy codes. *Proceedings of the Institute Radio Engineers* 1952; **40**(9):1098–1101.
2. Hirschberg DS, Lelewer DA. Efficient decoding of prefix codes. *Communications of the ACM* 1990; **33**(4):449–459.
3. Schwartz ES, Kallick B. Generating a canonical prefix encoding. *Communications of the ACM* 1964; **7**(3):166–169.
4. Connell JB. A Huffman-Shannon-Fano code. *Proceedings of the IEEE* 1973; **61**(7):1046–1047.
5. Moffat A, Turpin A. On the implementation of minimum-redundancy prefix codes. *IEEE Transactions on Communications* 1997; **45**(10):1200–1207.
6. Klein ST. Skeleton trees for the efficient decoding of Huffman encoded texts. *Information Retrieval* 2000; **3**(1):7–23.
7. Nekritch Y. Decoding of canonical Huffman codes with look-up tables. *Technical Report TR 85209-CS*, Department of Computer Science, University of Bonn, 1999. Available at: http://theory.cs.uni-bonn.de/∼yasha/.
8. Milidiú RL, Laber ES, Moreno LO, Duarte JC. A fast decoding method for prefix codes. *Proceedings of the IEEE Data Compression Conference*, March 2003, Storer JA, Cohn M (eds.). IEEE Computer Society Press: Los Alamitos, CA, 2003; 438.
9. Turpin A, Moffat A. Comment on 'Efficient Huffman decoding' and 'An efficient finite-state machine implementation of Huffman decoders'. *Information Processing Letters* 1998; **68**(1):1–2.
10. Hashemian R. Condensed table of Huffman coding, a new approach to efficient decoding. *IEEE Transactions on Communications* 2004; **52**(1):6–8.
11. Liddell M. Restricted prefix codes, *PhD Thesis*, Department of Computer Science and Software Engineering, The University of Melbourne, Australia, November 2003.
12. Chen D, Chiang Y-J, Memon N, Wu X. Optimal alphabet partitioning for semi-adaptive coding of sources of unknown sparse distributions. *Proceedings of the 2003 IEEE Data Compression Conference*, Storer JA, Cohn M (eds.). IEEE Computer Society Press: Los Alamitos, CA, 2003; 372–381.
13. de Moura ES, Navarro G, Ziviani N, Baeza-Yates R. Fast and flexible word searching on compressed text. *ACM Transactions on Information Systems* 2000; **18**(2):113–139.
14. Scholer F, Williams HE, Yiannis J, Zobel J. Compression of inverted indexes for fast query evaluation. *Proceedings of the 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, Tampere, Finland, August 2002, Beaulieu M, Baeza-Yates R, Myaeng SH, Järvelin K (eds.). ACM Press: New York, 2002; 222–229.
15. Brisaboa NR, Fariña A, Navarro G, Esteller MF. $(S, C)$-dense coding: An optimized compression code for natural language text databases. *Proceedings of the String Processing and Information Retrieval Symposium* (*Lecture Notes in Computer Science*, vol. 2857), Nascimento MA (ed.). Springer, 2003; 122–136.
16. Culpepper JS, Moffat A. Enhanced byte codes with restricted prefix properties. *Proceedings of the String Processing and Information Retrieval Symposium*, Buenos Aires, November 2005 (*Lecture Notes in Computer Science*, vol. 3772), Consens MP, Navarro G (eds.). Springer, 2005; 1–12.
17. Anh VN, Moffat A. Inverted index compression using word-aligned binary codes. *Information Retrieval* 2005; **8**(1):151–166. Software available at: http://www.cs.mu.oz.au/∼alistair/carry/ [30 March 2006].
18. Moffat A, Turpin A. *Compression and Coding Algorithms*. Kluwer Academic: Boston, MA, 2002.
19. Choueka Y, Klein ST, Perl Y. Efficient variants of Huffman codes in high level languages. *Proceedings of the 8th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, Montreal, Canada, June 1985. ACM: New York, 1985; 122–130.
20. Tanaka H. Data structure of the Huffman codes and its application to efficient encoding and decoding. *IEEE Transactions on Information Theory* 1987; **IT-33**(1):154–156.
21. Siemiński A. Fast decoding of the Huffman codes. *Information Processing Letters* 1988; **26**(5):237–241.
22. Bassiouni MA, Mukherjee A. Efficient decoding of compressed data. *Journal of the American Society for Information Science* 1995; **46**(1):1–8.
23. Hashemian R. High speed search and memory efficient Huffman coding. *IEEE Transactions on Communications* 1995; **43**(10):2576–2581.
24. Larmore LL, Hirschberg DS. A fast algorithm for optimal length-limited Huffman codes. *Journal of the ACM* 1990; **37**(3):464–473.



DOI: 10.1002/spe

25. Zobel J, Moffat A. Adding compression to a full-text retrieval system. *Software—Practice and Experience* 1995; **25**(8):891–903.
26. Harman DK. Overview of the second text retrieval conference (TREC-2). *Information Processing and Management* 1995; **31**(3):271–289.
27. Burrows M, Wheeler DJ. A block-sorting lossless data compression algorithm. *Technical Report 124*, Digital Equipment Corporation, Palo Alto, CA, May 1994.
28. Witten IH, Moffat A, Bell TC. *Managing Gigabytes: Compressing and Indexing Documents and Images* (2nd edn). Morgan Kaufmann: San Francisco, CA, 1999.
29. Moffat A, Isal RYK. Word-based text compression using the Burrows–Wheeler transform. *Information Processing and Management* 2005; **41**(5):1175–1192.
30. Larsson NJ, Moffat A. Offline dictionary-based compression. *Proceedings of the IEEE* 2000; **88**(11):1722–1732.
31. Ignatchenko S. An algorithm for online data compression. *C/C++ Users Journal* 1998; **16**(10):63–71.




DOI: 10.1002/spe