Phrase-Based Pattern Matching in Compressed Text

J. Shane Culpepper and Alistair Moffat

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

Abstract. Byte codes are a practical alternative to the traditional bit-oriented compression approaches when large alphabets are being used, and trade away a small amount of compression effectiveness for a relatively large gain in decoding efficiency. Byte codes also have the advantage of being searchable using standard string matching techniques. Here we describe methods for searching in byte-coded compressed text and investigate the impact of large alphabets on traditional string matching techniques. We also describe techniques for phrase-based searching in a restricted type of byte code, and present experimental results that compare our adapted methods with previous approaches.

1 Introduction

The compressed pattern matching problem is defined as: given a *pattern* P, a *text* T, and a corresponding *compressed text* Z generated by some compression algorithm, find all occurrences of P in T, that is, determine the set $\{|x| \mid T = xPy\}$, using P and Z.

The naive approach is to decompress the text before performing the pattern matching step, and fifteen years ago, this would probably have been the fastest mechanism. But ongoing growth in CPU power compared to I/O seek times in secondary storage devices has created a hardware speed gap, which allows increasingly complex algorithms to be utilised within the time that might otherwise be spent on I/O costs. It is, however, still necessary to balance efficiency (how quickly the compressed operation can be performed) and effectiveness (how good the compression is), and to take into account practical effects such as caching performance. In this framework, word-based modelling methods, combined with byte-aligned codes, offer several benefits [de Moura et al., 2000]. In particular, the use of byte codes allows use of available exact pattern matching algorithms, with only minimal modification required. The emphasis in previous research has been on variants of the Boyer-Moore approach, particularly the Horspool modification, see, for example, Fariña [2005]. While the BMH algorithm is clearly efficient on character-based alphabets in uncompressed text, it is unclear how it performs on the extended alphabets that arise from word-based compression models.

The traditional pattern matching problem has been studied for more than thirty years, and a broad range of efficient solutions have been proposed. All of the practical approaches use one of three searching techniques, and the notion of a *search window*, that positions the pattern relative to the text. The general techniques of interest include prefix-based searching, suffix-based searching, and factor-based searching. Several empirical studies of pattern searching strategies have also been conducted, and the reader is referred to, for example, the work of Navarro and Raffinot [2002], who consider the

impact of both varying pattern sizes and also varying alphabet size, and draw much of the previous work together.

However, relatively little is known about the performance impact of removing redundancy from the search text. This paper examines that question, and also evaluates the impact of large alphabets on uncompressed and compressed search times. We consider prefix-based and factor-based searching approaches as well as the favoured suffix-based approaches. We also examine the restricted-prefix byte codes introduced by Culpepper and Moffat [2005], and show that they too can be searched quickly using a modified Boyer-Moore-Horspool mechanism. Indeed, compression accelerates pattern matching so much that byte-coded sequences can be searched faster *after* compression than they can in their raw, uncompressed, form.

2 Byte-aligned compression

One of the first practical compressed pattern matching approaches was proposed by Manber [1997]. Manber's simple byte-pair encoding is efficient, but does not give competitive compression effectiveness. However, the idea of using bytes instead of bits was an important first step in creating algorithms that are both effective and efficient.

Simple byte coding techniques have also been used to compress sequences of integers in information retrieval systems, because they provide fast decoding compared to more principled bit-based codes. As an example application, consider the following text fragment taken from the popular children's book "Fox in Socks" [Seuss, 1965]:

Bim comes.\n Ben comes.\n Bim brings Ben broom.\n Ben brings Bim broom.\n

Instead of using a character-based approach to compression, de Moura et al. [2000] built on previous word-based approaches, and described what they called the *spaceless words model*. A spaceless words parser assumes that the text to be represented is a sequence of words followed by non-words, but with the added constraint that if any non-word is a single space, the space can be discarded by the encoder, and re-introduced later by the decoder. Words and non-words are assigned ordinal symbol identifiers as they appear, so that the sequence of words is transformed into a sequence of integer indices into a dictionary of strings. The resulting integer sequence can be represented by any coding method, including byte-aligned coding approaches. In the example, the text segment from "Fox in Socks" is transformed into the integer sequence:

$$1, 3, 4, 5, 3, 4, 1, 6, 5, 7, 4, 5, 6, 1, 7, 4\,,$$

where the "missing" symbol number 2 represents a single space character, and is not needed anywhere in this short message. Table 1 shows the sequential codewords assigned to this text fragment and the corresponding frequencies, and radix-4 codeword assignments for a range of byte-aligned codes.

The basic byte coding method (`bc`) uses codes that are fully static and easy to construct. It represents input integers using a radix-256 code in which values greater than 127 are *continuers* and are always followed by another byte, while values less than 128 are *stoppers*. The codewords generated are prefix-free, and it is easy to identify codeword boundaries directly in the compressed output, since the last byte of each codeword is less than 128. Note, however, that the code is static, and that actual frequency of each

Word	Sym.	Freq.	bc	phc	thc	dbc	scbc	rpbc
.\n	4	4	10 01	00	00 10	00	00	00
Bim	1	3	00	01	00 11	01	01	01
Ben	5	3	11 00	10	01 10 10	10 00	10	10
comes	3	2	10 00	11 00	01 10 11	10 01	11 00	11 00
brings	6	2	11 01	11 01	01 11 10	11 00	11 01	11 01
broom	7	2	10 10 00	11 10	01 11 11	11 01	11 10	11 10
(*space*)	2	0	01	—	—	—	—	—

Table 1. Symbol assignments and corresponding radix-4 codewords generated using a spaceless words model on a text fragment from "Fox in Socks". In the column `bc` the codewords are assigned based on ordinal symbol ordering; all other columns take the symbol frequency into account and bypass symbol 2, which does not appear in the message.

symbol is ignored. The "`bc`" column of Table 1 shows the codewords assigned when the set of symbol identifiers are taken at face value, and a radix-4 code computed (rather than the more usual radix-256 one). In a radix-4 version of `bc`, the "byte" values 00 and 01 are stoppers and the values 10 and 11 denote continuers. Note that symbol 2, which represents a single space, is assigned a code even though it does not appear in transformed source message, and that the most frequent symbol is not necessarily assigned the shortest codeword.

Another option is to calculate a radix-256 Huffman code, denoted `phc` (for plain Huffman code) in Table 1. Now an optimal code is computed for the set of symbol frequencies, and the source message represented accordingly. However, while `phc` provides maximal flexibility in assignment of codewords, it is impossible to search directly in the compressed text because one codeword can be a suffix of another codeword. Consider the codewords assigned by `phc` for the words *Bim* and *brings* in Table 1. The codeword 01 assigned to *Bim* is a suffix of the codeword 11 01 assigned to *brings*, and a search for *Bim* will result in a match against the second part of *brings*. In the example code the ambiguity could be resolved by looking at the preceding "byte" to see if it contains 11, but in a larger code, direct searching is impossible, since codeword boundaries are not identifiable.

To reintroduce searchability, de Moura et al. [2000] described *tagged Huffman codes* (`thc`), as a variation of the arrangement used in `bc`. Tagged Huffman codes are radix-128 Huffman codes which use 7 bits in each byte to store the Huffman code and 1 bit to signal the beginning of a codeword. With the extra tag bit inserted, `thc` codes are *suffix free* and allow any string matching algorithm such as `shift-or` or `horspool` to be used directly on the compressed text. The suffix-free property ensures that no false matches occur. Note that the cost of `thc` is exaggerated in Table 1 since in two-bit nibblets, only one actual data bit can be stored. Experimentally, searching in `thc` sequences is fast [de Moura et al., 2000], and searches are two to eight times faster than if the cost of decompression is added to the cost of uncompressed searching.

Brisaboa et al. [2003b] then noted that a static byte code could also be used, and in a system they call *end-tagged dense codes* (`dbc`), applied the same `bc` coding mechanism, but with the alphabet permuted into a new ordering dictated by decreasing occurrence frequency. A *prelude* describing the permutation is then necessary, to ensure that the decoder knows which source symbol should be assigned which codeword. In general,

the cost of the permutation is recovered through the use of shorter codewords for more frequent symbols, and overall compression is improved. The prelude proposed by Brisaboa et al. [2003b] is a rank-based mapping. For example, in Table 1, the symbol 5 is the 3rd most frequent and is assigned the codeword 10 00. Note that symbol 2, which does not appear at all in the example message, is no longer allocated a codeword – this is the "dense" part of the name. A drawback of the use of a prelude is that decoding is slower than the direct use of `bc`, because each decoded symbol must now be de-permuted via a large array, and cache-miss issues intrude [Culpepper and Moffat, 2005].

Brisaboa et al. [2003a] further realised that partitioning values other than 128 are possible, and that the sets of stoppers and continuers can be of different sizes – that better compression can be achieved by calculating an optimal partition based on the probability distribution of the input symbols. Brisaboa et al. [2003a] call this method (S, C)-dense coding (`scbc`). The only constraint is that the number of stoppers plus the number of continuers must satisfy $S + C = R$, where, as before, R is the radix of the coding system. For example, if $R = 4$ (as is used in the examples shown in Table 1) there are three possible (S, C)-dense arrangements: $(1, 3)$, $(2, 2)$, and $(3, 1)$. Note that the $(2, 2)$-arrangement corresponds to `dbc`. Table 1 shows the $(3, 1)$-arrangement, the best choice for the example text.

The most recent byte code variant provides a more flexible compromise between `phc` and the `scbc` coding approach [Culpepper and Moffat, 2005]. This method, called *restricted prefix byte coding* (`rpbc`), uses the first byte of each codeword to completely describe its length. Additional bytes can then use all of the remaining codespace. This allows compression gains, because different probability distributions can be more closely approximated by codes. Culpepper and Moffat showed that optimal codes can be calculated using a simple brute force method; and that additional compression gains are possible if care is taken when constructing the prelude. In Table 1, the optimal `rpbc` code turns out to be $(1, 1, 1, 2)$-arrangement, where the set of four values describe the codeword lengths associated with each of the four possible first "bytes".

The compression gain of `rpbc` does not come without cost. It is harder to track codeword boundaries in the compressed text, and backwards decoding – starting at a given codeword, and moving backwards in the byte stream to identify preceding codewords – is not possible. These new constraints make searching directly in the compressed text more challenging, particularly when using suffix-based searching algorithms.

3 Searching in byte-aligned compressed text

The ability to apply any searching algorithm with minimal modification is one of the key strengths of the byte-aligned compression systems. For example, the only alteration necessary to search directly in a stopper-continuer byte code is to add a false match filter that tests the byte immediately prior to a proposed match location. If that prior byte is a continuer then this proposed location is a false match, since it is not aligned on a codeword boundary in the byte stream. If it is a stopper, then the proposed match can be accepted as a valid appearance of the compressed codeword sequence. On the other hand, the `rpbc` method requires a different approach to false matches, because the codeword set is assigned exhaustively rather than partially, and it no longer suffices to look at the prior byte.

Algorithm 1 : Brute force searching in `rpbc`. Function $create_tables$ appears in Culpepper and Moffat [2005].

input: an `rpbc`-compressed array txt of compressed length $txtlen$ bytes, a compressed pattern pat of length $patlen$ bytes when compressed, and `rpbc` control parameters v_1, v_2, v_3, and v_4, with $v_1 + v_2 + v_3 + v_4 \leq R$, where R is the radix (typically 256).

1: set $t \leftarrow 0$ and $p \leftarrow 0$ and $occurrences \leftarrow \{\}$
2: $create_tables(v_1, v_2, v_3, v_4, R)$
3: **while** $t \leq txtlen - patlen$ **do**
4:     **while** $p < patlen$ **and** $pat[p] = txt[t + p]$ **do**
5:         set $p \leftarrow p + 1$
6:     **if** $p = patlen$ **then**
7:         set $occurrences \leftarrow occurrences \cup \{t\}$
8:     set $t \leftarrow t + suffix[txt[t]] + 1$ and $p \leftarrow 0$

output: the set of occurrences at which pat appears in txt, presented as a set of byte offsets in the compressed text txt.

Algorithm 2 : Jump-based searching in `rpbc`.

input: an array txt of $txtlen$ bytes representing `rpbc`-compressed symbols being searched, with a current pattern alignment that currently associates the first byte of the `rpbc`-compressed pattern with $txt[t]$; and an integer b that represents the number of bytes by which the pattern needs to be shifted.

1: **while** $b > 0$ **do**
2:     set $s \leftarrow suffix[txt[t]]$
3:     set $t \leftarrow t + s + 1$
4:     set $b \leftarrow b - s - 1$

output: pointer t indicates a new offset in txt that is again `rpbc`-symbol aligned.

The simplest way of making `rpbc`-coded sequences searchable is to ensure that false matches can never occur, by only testing valid pattern-to-text alignments. To do this, the pattern shifting step of the search process must ensure that codeword boundaries are identified and respected. Of itself this is not an onerous requirement, since in the `rpbc` code the first byte of each codeword can be used to index a table that unambiguously records how many more bytes there are in that codeword. On the other hand, the extra table lookup is required for each source symbol that is skipped, and potentially disrupts the tight searching loops that are the hallmark of efficient pattern matching algorithms.

To see the necessary modification, Algorithm 1 shows how a brute-force pattern searching mechanism is modified to maintain codeword alignments. Step 8 is the critical one; normally it would shift the text pointer t by one, in order to accommodate the next byte alignment. But, because source symbols typically span multiple bytes, the increment to t is augmented by $suffix[txt[t]]$, the number of trailing bytes in the codeword that commences at $txt[t]$.

The symbol-stepping approach is not possible with stopper-continuer byte codes since there is no way to compute the codeword length without examining each byte of the codeword. That is, fewer comparisons are necessary on average in the `rpbc`-brute force approach than in (say) a `scbc`-brute force approach; and false matches are impossible since a shift never places the byte-level alignment between codeword boundaries.

A similar technique can be employed in any searching approach that employs longer shifts, such as the horspool algorithm. For example, suppose that a pattern alignment shift of b bytes is indicated by a state-based searching process that is operating at the byte level, a shift that would normally be effected by an assignment of the form $t \leftarrow t + b$. Algorithm 2 shows how the assignment is replaced by a loop that steps at least that many bytes forward, while retaining codeword alignment. This modification can be applied to any jump-based pattern matching algorithm, including the kmp and horspool techniques, and when shift values are returned from the processing tables which fall in the middle of a codeword, the next codeword boundary is found via a longer shift. However, additional lookups of text prefix bytes are needed to find codeword boundaries, possibly affecting overall performance.

The other key issue with the rpbc codes is that, in the form described by Culpepper and Moffat [2005], they cannot be decoded backwards. Reverse decoding is useful when, for example, a small snippet is required, to show a context surrounding the location of an identified match in the compressed text. One possibility – viable because the first-byte of every codeword is touched during the loop shown in Algorithm 2 – is to maintain a stack or sliding window of codeword starting points. The window would need to be as long as the maximum extent of any backwards decoding.

A more elegant solution is also possible, by separating the prefix bytes and the suffix bytes into separate compressed sequences. This approach, which we denote rpbc_pa, offers additional pattern matching alternatives. For example, using the code shown in Table 1, the integer sequence

$$1, 3, 4, 5, 3, 4, 1, 6, 5, 7, 4, 5, 6, 1, 7, 4,$$

can be represented as a set of first "bytes"

$$01, 11, 00, 10, 11, 00, 01, 11, 10, 11, 00, 10, 11, 01, 11, 00,$$

and a corresponding set of suffix bytes,

$$,00, \quad , \quad ,00, \quad , \quad ,01, \quad ,10, \quad , \quad ,01, \quad ,10, \quad ,$$

where the commas show which of the first bytes each suffix byte is associated with.

Because all the first bytes are extracted out into a single sequence, they can be accessed either backwards or forwards. And the first bytes indicate the length of each codeword. That is, if the current location is known in both the sequence of first bytes and also in the sequence of suffix bytes, backwards decoding is now possible.

The searching process must change, and is carried out in two parts. First, the sequence of first bytes is searched, looking for matches against the first bytes of the codewords that make up the pattern. As the sequence of first bytes is processed, the cumulative sum of $\mathit{suffix}[\mathit{txt}[t]]$ is noted for each location t at which there is a first-byte match against the pattern. Once a set of candidate locations has been identified, the suffix bytes at those locations are checked against the suffix bytes of the pattern's codewords. Sentinels, or partial cumulative sums, can be used at predetermined locations in the prefix array to remove the requirement of inspecting each prefix byte, but at the cost of compression effectiveness. This rpbc_pa (prefix array) version of rpbc is one of the methods evaluated in the next section.

4 Experimental results

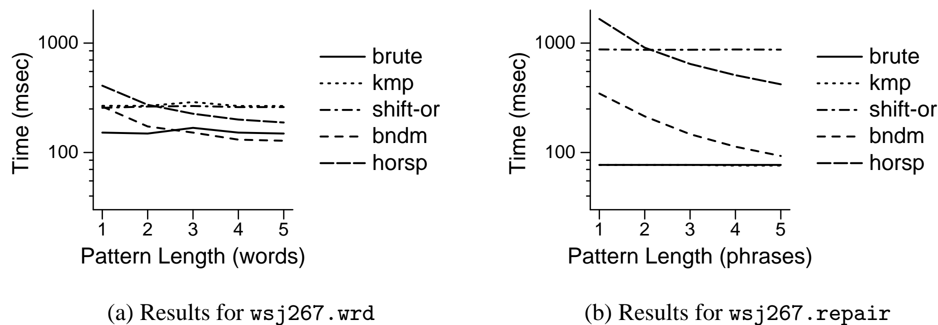
To evaluate the speed at which the various byte codes can be searched, we built two files of symbols from a 267 MB segment of SGML-tagged newspaper text, drawn from the *WSJ* component of the TREC data (see `trec.nist.gov`). The first one, `wsj267.wrd`, is the sequence of integers generated by the spaceless word model that was described earlier. The second file, `wsj267.repair`, is a sequence of integers representing phrases generated via an off-line, word-pair based encoding method called RE-PAIR [Larsson and Moffat, 2000]. Each symbol number represents a repeated phrase identified in the original word sequence, and because of the way the file is constructed, no pair of symbol numbers repeats. As well as integer-on-integer searching and character-on-character searching, five byte coding algorithms were investigated: `bc`, `dbc`, `scbc`, `rpbc`, and `rpbc_pa`. The `bc` and `dbc` methods were uniformly a little slower then `scbc`, and are not shown in the graphs below.

The average length of queries in web search systems is around 2.4 words per query [Spink et al., 2001]. To mirror this type of searching, took patterns of length 1 to 5 symbols, representing (in the case of `wsj267.wrd`) sequences of 1 to 5 words, or (in the case of `wsj267.repair`), 1 to 5 phrases. One hundred queries of each length were generated from the uncompressed integer sequences in the source files, by generating a random offset into it, and recording the sequence of symbols at that point. This process ensured that each pattern appeared at least once.

The integer pattern so generated can then be processed in different ways. For example, it can be used to measure the speed of an integer-on-integer search process; or converted back to the underlying character string and used in a character-on-character manner; or converted into codewords using any of the byte coding schemes, and then applied in a compressed codeword-on-codeword approach. For example, the three-symbol sequence $910, 2685, 153$ represents the original sequence "*offer may be*". The five different byte coding methods result in different corresponding patterns, with lengths varying from 4 bytes to 6 bytes.

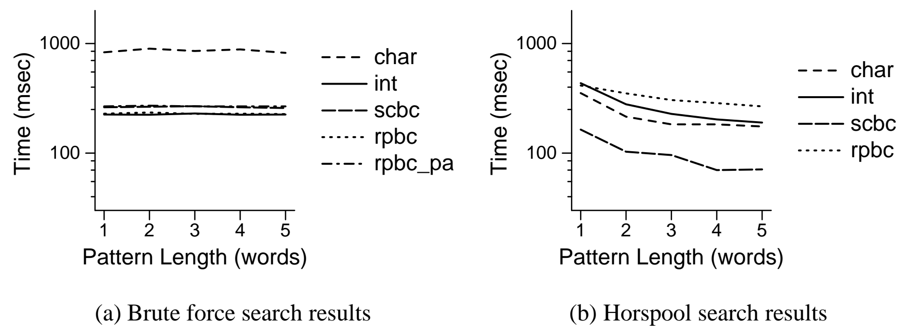
The first experiment was designed to evaluate the cost of integer-on-integer searching techniques. Figure 1 shows the measured performance of several different pattern matching techniques, without any compression having been applied. Algorithms which use preprocessed lookup tables proportional to the size of the alphabet tend to perform poorly when pattern lengths are short. Accesses to the large lookup table result in cache misses, which offset any gains achieved by the improved shifts. This effect continues until the search patterns become moderately long. In fact, brute force outperforms all of the more principled algorithms for patterns of three words or less, irrespective of the input file's probability distribution.

Figure 2 shows the speed at which the same patterns can be searched in the compressed domain, using two different pattern search algorithms, and a range of different byte-aligned coding methods. Both graphs in this figure relate to the spaceless words file `wsj267.wrd`; with the additional `char` method representing character-on-character searching in the uncompressed original form of the source file; and with the `int` method representing integer-by-integer searching in the uncompressed sequence of integers. When coupled with the brute-force searching approach (Figure 2a), `rpbc` performs faster than any of the other byte code methods, and at the same speed as searching in the uncompressed integer file. With decompression costs (assuming that the data is



(a) Results for `wsj267.wrd` (b) Results for `wsj267.repair`

Fig. 1. Baseline searching times for uncompressed, integer-on-integer pattern matching, using a 2.8 Ghz Intel Xeon with 2 GB of RAM. The methods are brute force; the Knuth-Morris-Pratt method; Shift-Or searching; Backward Nondeterministic DAWG Matching; and the Horspool variant of the Boyer-Moore method. All of these approaches are described by Navarro and Raffinot [2002].



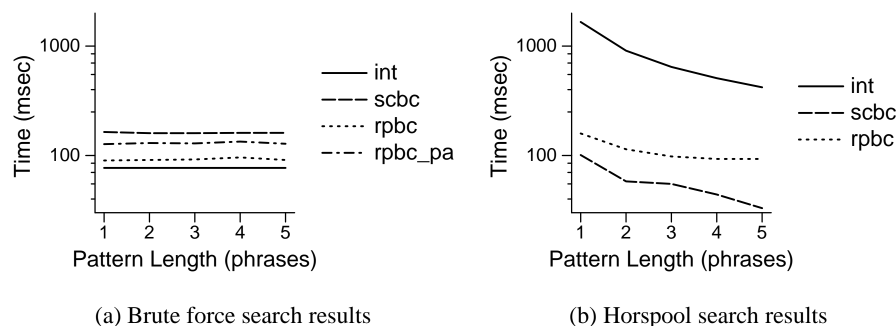
(a) Brute force search results (b) Horspool search results

Fig. 2. Searching `wsj267.wrd` using two different search techniques, and a range of uncompressed and compressed representations of text and patterns, using a 2.8 Ghz Intel Xeon with 2 GB of RAM.

stored in compressed form) included, the byte coding methods perform considerably better than the "decompress then search" baselines reflected in the `int` and `char` lines.

Figure 2b shows that the stopper-continuer byte code `scbc` performs better when coupled with the `horspool` searching method than when coupled with the brute force method. The `rpbc` variant has the same speed as in the brute force mode, and is clearly hampered by the additional operations involved in maintaining codeword boundaries.

Figure 3 shows the same experiment, but applied to file `wsj267.repair`. Now, when the symbol distribution is essentially flat and the alphabet size is large and dense, the integer-based `horspool` variant performs very poorly. Once again, the `rpbc` algorithm gives the same performance in the `horspool` environment as it does in the brute force one.



(a) Brute force search results

(b) Horspool search results

Fig. 3. Searching `wsj267.repair` using two different search techniques, and a range of uncompressed and compressed representations of text and patterns, using a 2.8 Ghz Intel Xeon with 2 GB of RAM.

Acknowledgement. The second author was funded by the Australian Research Council, and by the ARC Center for Perceptive and Intelligent Machines in Complex Environments. National ICT Australia (NICTA) is funded by the Australian Government's Backing Australia's Ability initiative, in part through the Australian Research Council.

References

N. R. Brisaboa, A. Fariña, G. Navarro, and M. F. Esteller. (S, C)-dense coding: An optimized compression code for natural language text databases. In M. A. Nascimento, editor, *Proceedings of the 10th International Symposium on String Processing and Information Retrieval*, volume 2857 of *LNCS*, pages 122–136, October 2003a.

N. R. Brisaboa, E. L. Iglesias, G. Navarro, and J. Paramá. An efficient compression code for text databases. In F. Sebastiani, editor, *Proceedings of the 25th European Conference on Information Retrieval Research*, volume 2633 of *LNCS*, pages 468–481, April 2003b.

J. S. Culpepper and A. Moffat. Enhanced byte codes with restricted prefix properties. In M. Consens and G. Navarro, editors, *Proceedings of the 12th International Symposium on String Processing and Information Retrieval*, volume 3772 of *LNCS*, pages 1–12, November 2005.

E. S. de Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Fast and flexible word searching on compressed text. *ACM Transactions on Information Systems*, 18(2):113–139, 2000.

A. Fariña. *New compression codes for text databases*. PhD thesis, Universidade de Coruña, April 2005.

N. J. Larsson and A. Moffat. Offline dictionary-based compression. *Proceedings of the IEEE*, 88 (11):1722–1732, November 2000.

U. Manber. A text compression scheme that allows fast searching directly in the compressed file. *ACM Transactions on Information Systems*, 5(2):124–136, April 1997.

G. Navarro and M. Raffinot. *Flexible pattern matching in strings*. Cambridge University Press, Cambridge, United Kingdom, first edition, 2002.

Dr. Seuss. *Fox in socks*. Random House, first edition, 1965. Written by T. Geisel.

A. Spink, D. Wolfram, B. J. Jansen, and T. Saracevic. Searching the web: The public and their queries. *Journal of the American Society for Information Science*, 52(3):226–234, 2001.