Speeding up Compressed Matching with SBNDM2

Kerttu Pollari-Malmi, Jussi Rautio, and Jorma Tarhio[⋆]

Department of Computer Science and Engineering
Aalto University, P.O. Box 15400, FI-00076 Aalto, Finland
{kerttu.pollari-malmi, jorma.tarhio}@aalto.fi, kipuna@gmail.com

Abstract. We consider a compression scheme for natural-language texts and genetic data. The method encodes characters with variable-length codewords of k-bit base symbols. We present a new search algorithm, based on the SBNDM2 algorithm, for this encoding. The results of practical experiments show that the method supersedes the previous comparable methods in search speed.

Keywords: compressed matching

1 Introduction

With the amount of information available constantly growing, fast information retrieval is a key concept in many on-line applications. The *string matching problem* is defined as follows: given a pattern $P = p_1 \cdots p_m$ and a text $T = t_1 \cdots t_n$ in an alphabet Σ, find all the occurrences of P in T. Various good solutions [7] have been presented for this problem. The most efficient solutions in practice are based on the Boyer-Moore algorithm [3] or on the BNDM algorithm [21].

The *compressed matching problem* [1] has gained much attention. In this problem, string matching is done in a compressed text without decompressing it. Researchers have proposed several methods [2,19,20,22] based on Huffman coding [15] or the Ziv-Lempel family [28,29]. Alternatively, *indexing methods* [11,26] can be used to speed up string matching. However, in this paper we concentrate on traditional compressed matching.

One presented idea is to encode whole words by using end-tagged dense code or (s, c)-dense code [4]. In both the approaches, codewords consist of one or more bytes. In end-tagged dense code, the first bit of each byte specifies whether the byte is the last byte of the codeword or not. Thus, there are 128 possible byte values which end the codeword, called *stoppers*, and 128 possible values for *continuers*, which are not the last byte of the codeword. In (s, c)-dense code, the proportion between stoppers and continuers can be chosen more freely to minimize the size of the compressed text. The bytes whose value is less than c are continuers whereas the bytes whose value is at least c but less than $s + c$ (the number of possible byte values) are stoppers. String matching in the compressed text is performed by compressing the pattern and searching for it by using the Boyer-Moore-Horspool algorithm [14]. Brisaboa et al. [5,6] also present dynamic versions of end-tagged dense code and (s, c)-dense code allowing fast searching. All these methods use word-based encoding.

Culpepper and Moffat [8,9] present another word-based compression approach where the value of the first byte of a codeword always specifies the length of the

[⋆] Work supported by the Academy of Finland (grant 134287).

codeword. The rest of bytes in the codeword can obtain any of the possible values. The compressed search can be performed in several different ways, for example using a Knuth-Morris-Pratt [17] or Boyer-Moore-Horspool type algorithm. However, the search algorithm requires that a shift never places the byte-level alignment between codeword boundaries.

The word-based compression methods are poor in the case of highly inflectional languages like Finnish. In this paper, we will concentrate on character-based compression. Shibata et al. [25] present a method called BM-BPE which finds text patterns faster in a compressed text than Agrep [27] finds the same patterns in an uncompressed text. Their search method is based on the Boyer-Moore algorithm and they employ a restricted version of byte pair encoding (BPE) [13], which replaces common pairs of characters with unused characters.

Maruyama et al. [18] present a compression method related to BPE using recursive pairing. However, in their method they select a digram A_iB_i (A_i and B_i are either symbols from alphabet Σ or variables presented earlier in the compression process) that occurs most frequently after $a_i \in \Sigma$ and replace the string $a_iA_iB_i$ with the string a_iX where X is a new variable. This is performed for every $a_i \in \Sigma$. Because the symbol preceding a digram is taken into account, the variable X can be used to replace different digrams after different symbols. They also present an efficient pattern matching algorithm for compressed text. The search algorithm is Knuth-Morris-Pratt [17]. The automaton is modified so that it memorizes the symbol read previously and its *Jump* and *Output* functions are also defined for variables occurring in the compressed text.

We present a different character-based method which is faster than BM-BPE. In our method, characters are encoded as variable-length *codewords*, which consist of *base symbols* containing a fixed number of bits. Our encoding approach is a generalization of that of de Moura et al. [19], where bytes are used as base symbols for coding words. We present several variants of our encoding scheme and give two methods for string matching in the compressed text.

Earlier we have presented a search algorithm [23,24] based on Tuned Boyer-Moore [16], which is a variation of the Boyer-Moore-Horspool algorithm [14]. The shift function is based on several base symbols in order to enable longer jumps than the ordinary occurrence heuristic. In this paper, we present a new search algorithm based on SBNDM2 [10] which is a variation of BNDM, the Backward Nondeterministic DAWG Matching algorithm [21]. SBNDM2 is a fast bit-parallel algorithm, which recognizes factors of the pattern by simulating a nondeterministic automaton of the reversed pattern. Fredriksson and Nikitin [12] have earlier used BNDM to search for patterns in the text that has been compressed by their own algorithm. However, in their tests BNDM on compressed text was slower than the Boyer-Moore-Horspool algorithm on uncompressed text, while in our experiments our SBNDM based search algorithm was faster.

We present test results of four variations of our algorithms together with two reference algorithms for compressed texts as well as two other reference algorithms for uncompressed texts. We are interested only in search speed. The tests were run on DNA and on English and Finnish texts. Our SBNDM2 based search method was the fastest among all the tested algorithms for English patterns of at least 9 characters.

The paper has been organized as follows. Section 2 provides an enhanced presentation of our coding scheme [24]. Our two search algorithms are described in Section 3.

Then Section 4 reports the results of our practical experiments before the conclusions in Section 5.

2 Stopper encoding

2.1 Stoppers and continuers

We apply a semi-static encoding scheme called *stopper encoding* for characters, where the codes are based on frequencies of characters in the text to be compressed. The frequencies of characters are gathered in the first pass of the text before the encoding in the second pass. Alternatively, fixed frequencies based on the language and the type of the text may be used.

A codeword is a variable-length sequence of *base symbols* which are represented as k bits, where k is a parameter of our scheme. The different variants of our scheme are denoted as SE_k, where SE stands for stopper encoding.

Because the length of a code varies, we need a mechanism to recognize where a new one starts. A simple solution is to reserve some of the base symbols as *stoppers* which can only be used as the last base symbol of a code. All other base symbols are *continuers* which can be used anywhere but at the end of a code. If $u_1 \cdots u_j$ is a code, then u_1, …, u_{j-1} are continuers and u_j is a stopper.

De Moura et al. [19] use a scheme related to our approach. They apply 8-bit base symbols to encode words where one bit is used to describe whether the base symbol is a stopper or a continuer. Thus they have 128 stoppers and 128 continuers. Brisaboa et al. [4] use the optimal number of stoppers.

2.2 Number of stoppers

It is an optimization problem to choose the number of stoppers to achieve the best compression ratio (the size of the compressed file divided by that of the original file). The optimal number of stoppers depends on the number of different characters and the frequencies of the characters. If there are s stoppers of k bits, there are $c = 2^k - s$ continuers of k bits. A codeword of l base symbols consists of $l - 1$ continuers and one stopper. Thus, there are sc^{l-1} valid codewords of length l.

Let $\Sigma = \{a_1, a_2, \ldots, a_q\}$ be the alphabet, and let $f(a_i)$ be the frequency of a_i. For simplicity, we assume that the alphabet is ordered according to the frequency, i.e. $f(a_i) \geq f(a_j)$ for $i < j$.

Now $a_1, \ldots, a_s$ are encoded with one base symbol, $a_{s+1}, \ldots, a_{s+cs}$ are encoded with two base symbols, $a_{s+cs+1}, \ldots, a_{s+c^2s}$ are encoded with three base symbols, and so on. So the number of characters that can be encoded with l or less base symbols is

$$\sum_{j=1}^{l} c^{j-1}s = \frac{s(c^l - 1)}{c - 1}.$$

From this we can calculate the number of k-bit base symbols $l(a_i, k, s)$ needed to encode the character a_i in the case of s stoppers. It must be possible to encode at least i first characters by using $l(a_i, k, s)$ or less base symbols and thus

$$i \leq \frac{s(c^{l(a_i,k,s)} - 1)}{c - 1}.$$

By solving the inequality, we obtain

$$l(a_i, k, s) \geq \log_c(i(c-1)/s+1).$$

Because $l(a_i, k, s)$ must be an integer and we want to choose the smallest possible integer,

$$l(a_i, k, s) = \lceil \log_c(i(c-1)/s+1) \rceil.$$

Then the compression ratio C is

$$C = \sum_{i=1}^{q} \frac{f(a_i)l(a_i, k, s)k}{8}. \tag{1}$$

Let us consider 4-bit base symbols as an example. Let us assume that all the characters are equally frequent. Table 1 shows the optimal numbers of stoppers for different sizes of the alphabet.

# of characters	*Stoppers*	*Base symb./char at end*
1–31	15	1.55
31–43	14	1.70
43–53	13	1.77
53–61	12	1.82
61–67	11	1.85
67–71	10	1.87
71–73	9	1.89
73–587	8	2.87

Table 1. Optimal stopper selection.

To find the optimal number of stoppers, the frequencies of characters need to be calculated first. After that, formula (1) can be used for all reasonable numbers of stoppers ($8-16$ for 4-bit base symbols), and the number producing the lowest compression ratio can be picked. Another possibility is presented by Brisaboa et al. in [4], where binary search is used to find a minimum compression ratio calculated by using formula (1). They consider the curve which presents the compression ratio as a function of the number of stoppers. At each step, it is checked whether the current point is in the decreasing or increasing part of the curve and the search moves towards the decreasing direction. Using this strategy demands that there exists a unique minimum of the curve, but in practice natural language texts usually have that property.

2.3 Building the encoding table

After the number of stoppers (and with it, the compression ratio) has been decided, an encoding table can be created. The average search time is smaller if the distribution of base symbols is as uniform as possible. We use a heuristic algorithm, which produces comparable results with an optimal solution. With this algorithm, the encoding can be decided directly from the order of the frequencies of characters.

Base symbols of two and four bits are the easiest cases. In this encoding with 2^k different base symbols and s stoppers, the base symbols $0, 1, \ldots, s-1$ will act as stoppers and the base symbols $s, \ldots, 2^k - 1$ as continuers. At first, the s most common characters are assigned their own stopper base symbol, in the inverse order of frequency (the most common character receiving the base symbol $s-1$ and so on). After that, starting from the $s+1$:th common character, each character is assigned a two-symbol codeword in numerical order (first the ones with the smallest possible continuer, in the order starting from the smallest stopper). Exactly the same ordering is used with codewords of three or more base symbols. We call this technique *folding*, since the order of the most frequent characters is reversed until a certain folding point. The aim of folding is to equalize the distribution of base symbols in order to speed up searching. The resulting encoding of the characters in the KJV Bible is presented in Table 2.

ch	code	ch	code	ch	code	ch	code
␣	`d`	w	`e1`	D	`e9`	N	`fe1`
e	`c`	y	`f1`	T	`f9`	P	`ff1`
t	`b`	c	`e2`	R	`ea`	C	`ee2`
h	`a`	g	`f2`	G	`fa`	x	`ef2`
a	`9`	b	`e3`	J	`eb`	q	`fe2`
o	`8`	p	`f3`	S	`fb`	Z	`ff2`
n	`7`	↩	`e4`	B	`ec`	Y	`ee3`
s	`6`	v	`f4`	?	`fc`	K	`ef3`
i	`5`	.	`e5`	H	`ed`	!	`fe3`
r	`4`	k	`f5`	M	`fd`	U	`ff3`
d	`3`	A	`e6`	E	`ee0`	(	`ee4`
l	`2`	I	`f6`	j	`ef0`	)	`ef4`
u	`1`	:	`e7`	W	`fe0`	V	`fe4`
f	`0`	;	`f7`	F	`ff0`	-	`ff4`
m	`e0`	L	`e8`	’	`ee1`	Q	`ee5`
,	`f0`	O	`f8`	z	`ef1`		

Table 2. Encoding table for the King James Bible, 4-bit version, 14 stoppers. The characters have been sorted by the order of their relative frequencies.

bsym	freq	bsym	freq	bsym	freq	bsym	freq
`0`	4.85%	`4`	4.58%	`8`	5.11%	`c`	8.48%
`1`	4.31%	`5`	4.62%	`9`	5.56%	`d`	16.21%
`2`	4.62%	`6`	4.38%	`a`	5.96%	`e`	8.03%
`3`	4.80%	`7`	4.99%	`b`	6.51%	`f`	7.01

Table 3. Relative frequencies of base symbols in the encoded text (average: 6.25%).

Table 3 shows the relative frequencies of the base symbols in our example. The frequencies of the first stoppers depend on the frequencies of the most common characters. E.g., the frequency of `d` depends on the frequency of the space symbol.

2.4 Variant: code splitting

Consider a normal, uncompressed text file, composed of natural-language text. In the file, most of the information (*entropy*) in every character is located in the 5–6 lowest bits. A search algorithm could exploit this by having the fast loop ignore the highest bits, and take the lower bits of several characters at once. The idea is not usable as such, because the time required for `shift` and `and` operations is too much for a fast loop to perform efficiently. However, it is the basis for a technique called *code splitting.*

Applying code splitting to SE_6 means that each 6-bit base symbol is split into two parts: the 4-bit *low part* and the 2-bit *high part*, which are stored separately. The search algorithm first works with low parts only, trying to find a match in them. Only after such a match is found, the high parts are checked. Depending on the alphabet, it may also be applicable to split the codes into 4-bit high part and 2-bit low part and to start the search with the high part. We denote a code-split variant as $SE_{6,4}$ or $SE_{6,2}$, where the smaller number is the number of low bits.

Code splitting can also be applied to SE_8. $SE_{8,4}$ is a special case. The characters of the original text are split as such into two 4-bit arrays. This speeds up search but does not involve compression. Code splitting is useful in evenly-distributed alphabets where no compression could be gained, or as an end-coding method for some compression method, including de Moura [19].

3 String matching

We start with a description of our old searching algorithm [24]. This will help to understand the details of the new algorithm, which is more complicated.

3.1 Boyer-Moore based algorithm

Let us consider a text with less than 16 different characters. With the SE_4 schema, any character of this text can be represented with a codeword consisting of a single four-bit base symbol. For effective use, two consecutive base symbols are stored into each byte of the encoded text, producing an exact compression ratio of 50%.

Instead of searching directly the 4-bit array, which would require expensive `shift` and `and` operations, we use the 8-bit bytes. There are two 8-bit *alignments* of each 4-bit pattern: one starting from the beginning of a 8-bit character and the other one from the middle of it, as presented in Table 4. For example, consider the encoded pattern (in hexadecimal) `618e05`. Occurrences of both `61-8e-05` and `*6-18-e0-5*`[1] clearly need to be reported as matches.

The final algorithm works exactly in the same way: by finding an occurrence of the encoded pattern consisting of longer codewords in the text. One additional constraint exists: the base symbol directly preceding the presumed match must be a stopper symbol. Otherwise, the meaning of the first base symbol is altered. It is not the beginning of a codeword, but it belongs to another codeword which begins before the presumed match. Thus, the match is not complete.

It is also noteworthy that because positions in the original text are not mapped one-to-one into positions in the encoded text, an occurrence in the encoded text cannot be converted to a position in the original text. However, we get the *context* of the encoded pattern, and with fast on-line decoding, it can be expanded.

The search algorithm, based on Tuned Boyer-Moore (TBM) [16], is called *Boyer-Moore for Stopper Encoding*, or BM-SE. It is a direct extension of TBM allowing multiple patterns and classes of characters. The algorithm contains a fast loop designed to quickly skip over most of the candidate positions, and a slow loop to verify possible matches produced by the fast loop.

[1] The asterisk `*` denotes a wild card, i.e. any base symbol.

...	6 1	8 e	0 5	...
...	... 6	1 8	e 0	5

Table 4. An example of BM-SE, pattern `618e05`.

The last byte of both alignments not containing wild cards, the *pointer byte*, has a skip length of 0. The previous bytes have their skip lengths relative to their distances from the pointer byte. The skip lengths for the example pattern `618e05` are presented in Table 5. The fast loop operates by reading a character from text, and obtaining its *skip length*. If the skip length is 0, the position is examined with a slow loop, otherwise the algorithm moves forward a number of positions equal to the skip length.

A half-byte (one containing a wild-card) at the end of the pattern is ignored for the fast loop, whereas one in the beginning sets all 16 variations to have the desired jump length. This works according to the bad-character, or occurrence, heuristic of Boyer and Moore [3].

encoded ch	skip
`05, e0`	0
`8e, 18`	1
`61, *6`	2
`**`	3

Table 5. Skip lengths. The symbol `*` is a wild card.

String matching with 2-bit base symbols works basically in the same way as with 4-bit ones. There are 4 parallel alignments instead of 2, and the number of characters enabled by wild-cards can vary from 4 to 64. Compression and string matching with 2-bit base symbols and its various extension possibilities are presented in [23].

3.2 SBNDM2 matching

Our new search solution, called SBNDM2-SE, uses the SBNDM2 algorithm [10] for string matching in SE-coded texts. SBNDM2 is a simplified version of BNDM [21]. The idea of SBNDM2 is to maintain a state vector D, which has a one in each position where a factor of the pattern starts such that the factor is a suffix of the processed text string. For example, if the processed text string is “abc” and the pattern is “pabcabdabc”, the value of D is 0100000100. To make maintaining D possible, table B which associates each character a with a bit mask expressing its locations in the pattern is precomputed before searching.

When a new alignment is checked, at first two characters of the text are read before testing the state vector D. We use a precomputed table F such that for each pair of characters, $F[c_i c_j] = B[c_i] \& (B[c_j] \ll 1)$.

At each alignment i we check whether $D = F[t_i, t_{i+1}]$ is zero. If it is, it means that the characters t_i and t_{i+1} do not appear successively in the pattern and we can fast proceed to the next alignment $m-1$ positions forward. If D is not zero, we continue by calculating $D \leftarrow B[t_{i-1}] \& (D \ll 1)$ and decrementing the value of i by 1 until D becomes zero. This happens when either the pattern has been found or the processed substring does not belong to the pattern. The former case can been distinguished from the latter by the number of characters processed before D became zero.

When we apply the SBNDM2 algorithm to an SE-encoded text, we start by encoding the pattern with the same algorithm as the text was encoded. If SE_4 is used, the length of the base symbol is 4 bits. This means that the pattern may start either from the beginning of a 8-bit character or from the middle of it. To find the occurrence in both cases, we search for two patterns simultaneously: one which starts from the beginning of the encoded pattern and the other which starts from the second half-byte of the encoded pattern. In SBNDM2, it is possible to search for two equal length patterns by searching for the pattern which has been constructed by concatenating the two patterns. For example, if the encoded pattern (in hexadecimal) is `618e0`, we search for both pattern `61-8e` and pattern `18-e0` simultaneously by searching the pattern `61-8e-18-e0`. An extra bit vector (denoted by f in the pseudocode of the algorithm) is used to prevent from finding matches which consist of the end of the first half and the beginning of the second half like `8e-18` in the example case. If SBNDM2 finds the first or the second half of the longer pattern, it is checked whether this is the real occurrence of the pattern by matching the bytes of the encoded pattern and the bytes of the text one by one. It is also checked that the first or last half-byte is found in the text.

If the length of the encoded pattern is even, for example in the case of the pattern `618e05`, the long pattern is constructed by concatenating strings where the first is formed by dropping the last byte of the encoded string and the latter by dropping the first and last half-byte of the encoded string. In the case of the example string, the pattern `61-8e-18-e0` would be searched for by SBNDM2. Again, if the first or the second half of the longer pattern is found, the match is checked by comparing the bytes of the text and the pattern one by one. The pseudocode is shown as Algorithm 1.

If $SE_{8,4}$ encoding is used, SBNDM2 algorithm is applied only to the file containing the low part (the 4 lowest bits) of each encoded byte. The search is performed similarly to the search in the case of SE_4. If an occurrence is found, the high bits are checked one by one.

4 Experiments

We tested the stopper encoding schemes against other similar algorithms and each other. As a reference method for uncompressed texts, we use TBM (`uf_fwd_md2` from Hume and Sunday's widely known test bench [16]), and SBNDM2 [10]. As a reference method for compressed texts, we use BM-BPE [25] and Knuth-Morris-Pratt search algorithm on recursive-pairing compressed text developed by Maruyama et al. [18]. Their compression algorithm is denoted by BPX and their search algorithm for compressed text by KMP-BPX. The implementation is by courtesy of Maruyama. For BM-BPE, we used the recommended version with a maximum of 3 original characters per encoded character. The implementation is by courtesy of Takeda.

As to our own variants, we tested TBM and SBBDM2 based algorithms for SE_4 and $SE_{8,4}$ encoded tests (denoted by BM-SE_4 and BM-$SE_{8,4}$, SBNDM2-SE_4, and SBNDM2-$SE_{8,4}$, respectively). SE_4 is a representative of our compression scheme, and $SE_{8,4}$ represents code splitting. Note that $SE_{8,4}$ offers no compression at all.

The most important properties of these algorithms are search speed and compression ratio, in that order. In this experiment, only these properties are examined. Encoding and decoding speeds were not considered. In search, we focus on pattern lengths of 5, 10, 20, and 30. In the case of English and Finnish, the pattern length of 30 was not used, because some of the encoded patterns were too long for the 32-bit

Algorithm 1 SBNDM2 for SE$_4$-encoded text. ($P^2 = p_1p_2 \cdots p_{2m}$ is a string containing both the alignments of the encoded pattern, $T = t_1t_2 \cdots t_n$ is the encoded text.)

```
    /* Preprocessing */
 1: for all $c \in \Sigma'$ do $B[c] \leftarrow 0$  /* c is a byte in the compressed text. */
 2: for all $c_i, c_j \in \Sigma'$ do $F[c_ic_j] \leftarrow 0$
 3: for $j \leftarrow 1$ to $2m$ do
 4:     $B[p_j] \leftarrow B[p_j] \mid (1 << (2m - j))$
 5: for $i \leftarrow 1$ to $2m$ do
 6:     for $j \leftarrow 1$ to $2m$ do
 7:         $F[p_ip_j] \leftarrow B[p_i]$ & $(B[p_j] << 1)$
 8: $f \leftarrow$ a bit vector containing 0s in positions 0 (the least significant bit) and $m$ and otherwise 1s.
    /* Searching */
 9: $j \leftarrow m$
10: while (true) do
11:     while $(D \leftarrow F[t_{j-1}jt_j]) = 0$ do
12:         $j \leftarrow j + m - 1$
13:     pos $\leftarrow j$
14:     while $(D \leftarrow (D << 1)$ & $f$ & $B[t_{j-2}]) \neq 0$ do
15:         $j \leftarrow j - 1$
16:     $j \leftarrow j + m - 2$
17:     if $j \leq$ pos then
18:         if $j > n$ then
19:             End of text reached, exit.
20:         A possible occurrence ending at pos found, check
21:         by matching the bytes one by one.
22:         $j \leftarrow$ pos $+1$
```

bitvector used by SBNDM2-SE$_4$. (Because SBNDM2-SE$_4$ searches for two patterns at the same time, one which starts from the beginning of the encoded pattern and the other starting from the second half-byte of the encoded pattern, the maximum length of the encoded pattern is 34 half-bytes when the 32-bit bitvector is used.) After the preliminary experiments, we added tests for pattern lenghts of 6, 7, 8 and 9 for English texts and for pattern lengths of 11, 12, 13, 14 and 15 for Finnish texts to find out where SBNDM2-SE$_4$ becomes faster than SBNDM2.

As a DNA text, we used a 5 MB long part of a DNA of the fruitfly. As an English text, we used the KJV Bible such that the beginning of the text was concatenated to the end to lengthen the text to 5 MB. As a Finnish text, we used the Finnish Bible (1938). Because the original size of the Finnish text was about 4 MB, we concatenated the beginning of the text to the end to lengthen the text to 5 MB.

The compression ratios for various encoding algorithms are presented in Table 6. The ratios are given for the original (not extended) text files. For the compression of natural-language texts, the BPX algorithm is better and BPE slightly better than SE$_4$. This is true also for DNA text, because SE$_4$ always uses at least four bits for each symbol.

All experiments were run on a 2.40 GHz Lenovo ThinkPad t400s laptop with Debian Linux and the Intel® Core$^{\text{TM}}$2 Duo CPU P9400 2.40 GHz processor. The computer had 3851 megabytes of main memory. There were no other users using the test computer at the same time. The Linux function `sched_setaffinity` was used to bind the process to only one core. All the programs were compiled using `gcc` [version 4.6.3] with the optimization flag `-O3` and the flag `-m64` to produce 64 bit code. The

	KJV Bible	Finnish Bible	DNA
BPX	28.0%	32.6%	27.8%
BPE	51.0%	52.1%	34.0%
$SE_{8,4}$	100.0%	100.0%	100.0%
SE_4	58.8%	58.2%	50.0%

Table 6. Compression ratios.

programs were modified to measure their own execution time by using the `times()` function similarly to Hume and Sunday's test bench. The clocked time includes everything except program argument processing and reading the file containing the encoded text from disk.

In each test, the set of 200 patterns was searched for. The patterns were randomly picked from the corresponding text. The searches were performed sequentially one pattern at a time. The preprocessing was repeated 100 times and the searching 500 times for each pattern. The average times for the whole set are presented in Tables 7, 8, and 9, and graphically in Figures 1, 2, and 3.

For short DNA patterns, KMP-BPX and SBNDM2-SE_4 were the fastest. For longer DNA patterns, all our algorithms were clearly faster than KMP-BPX. SBNDM2-SE_4 was the fastest for longer patterns.

For short English patterns, SBNDM2 was clearly faster than our algorithms, which were comparable to KMP-BPX. For longer patterns, our algorithms outperformed KMP-BPX and BM-BPE, but mostly they were comparable to SBNDM2. SBNDM2-SE_4 was faster than SBNDM2 for longer patterns. To be more exact, SBNDM2-SE_4 was still slower than SBNDM2 for the pattern length 8, but the fastest among the tested methods for the pattern length 9.

The test results for Finnish patterns were rather similar to the results for English patterns, but the search times of SBNDM2-SE_4 and SBNDM2-$SE_{8,4}$ were nearer to each other. For some pattern lengths, SBNDM2-SE_4 was slightly faster than SBNDM2-$SE_{8,4}$, for some other pattern lengths it was the opposite. The smallest pattern length was 11 where SBNDM2-SE_4 and SBNDM2-$SE_{8,4}$ were faster than SBNDM2 and the fastest among the tested methods. Interestingly, most algorithmss were a bit faster with Finnish data than with English data.

pattern length →	5	10	20	30
algorithm ↓				
TBM	2336	1918	1779	1733
SBNDM2	1609	1014	582	407
BM-BPE	1387	728	424	336
KMP-BPX	969	951	952	957
BM-SE_4	1548	751	491	416
BM-$SE_{8,4}$	1562	766	501	425
SBNDM2-SE_4	974	397	210	169
SBNDM2-$SE_{8,4}$	1067	475	277	235

Table 7. Search times of DNA patterns in milliseconds, text size 5 MB. The pattern set contained 200 patterns.

pattern length → algorithm ↓	5	6	7	8	9	10	20
TBM	1072	939	831	756	706	668	465
SBNDM2	542	507	448	399	397	370	295
BM-BPE	1275	1250	923	871	860	701	431
KMP-BPX	960	959	960	959	960	962	966
BM-SE$_4$	925	749	626	543	489	440	239
BM-SE$_{8,4}$	1040	719	705	522	532	428	240
SBNDM2-SE$_4$	951	726	501	418	354	308	156
SBNDM2-SE$_{8,4}$	912	949	505	489	362	350	169

Table 8. Search times of English patterns in milliseconds, text size 5 MB. The pattern set contained 200 patterns.

pattern length → algorithm ↓	5	10	11	12	13	14	15	20
TBM	1007	630	593	567	540	527	504	439
SBNDM2	473	308	295	286	277	272	260	238
BM-BPE	1240	670	657	645	549	539	534	406
KMP-BPX	1118	1123	1126	1125	1127	1127	1126	1129
BM-SE$_4$	914	428	389	360	335	314	289	226
BM-SE$_{8,4}$	1000	403	414	345	348	298	301	222
SBNDM2-SE$_4$	969	311	266	240	221	205	189	143
SBNDM2-SE$_{8,4}$	907	327	254	254	210	212	182	148

Table 9. Search times of Finnish patterns in milliseconds, text size 5 MB. The pattern set contained 200 patterns.

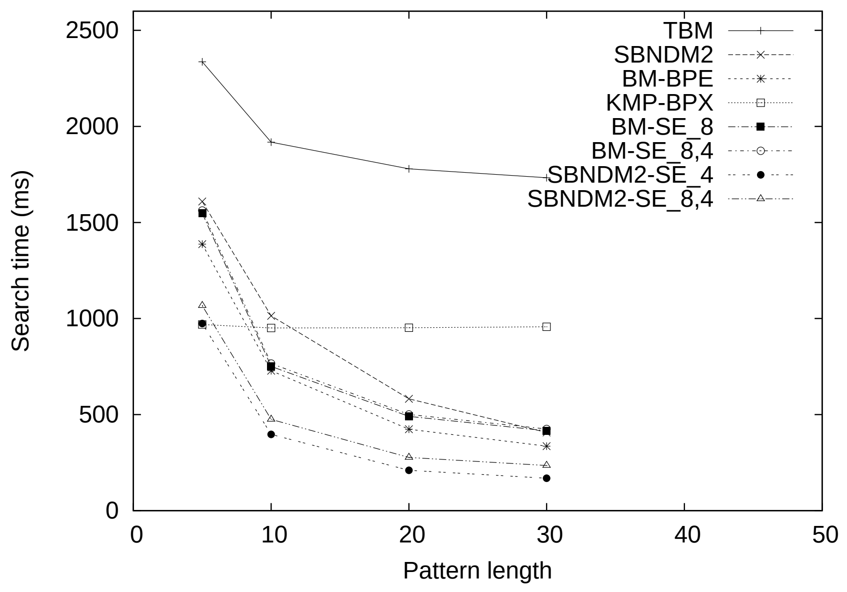

Figure 1. Search times of DNA patterns in milliseconds.

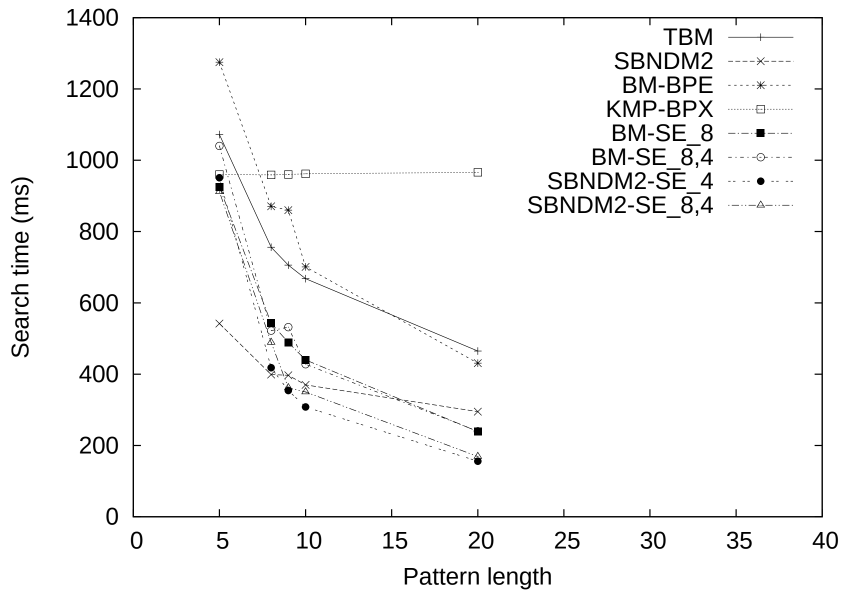

Figure 2. Search times of English patterns in milliseconds.

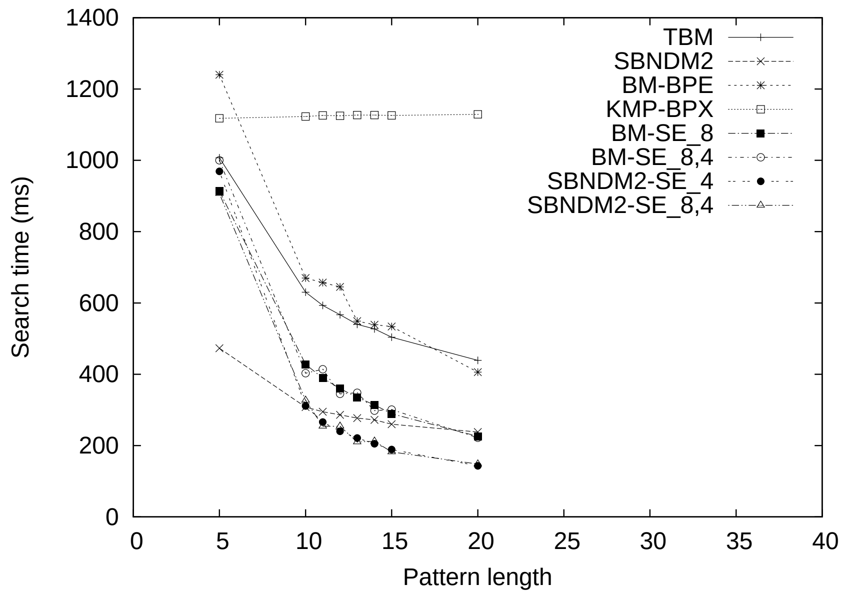

Figure 3. Search times of Finnish patterns in milliseconds.

5 Concluding remarks

Stopper encoding is a semi-static character-based compression method enabling fast searches. It is useful in applications where on-line updates and on-line decoding are needed. Stopper encoding resembles the semi-static Huffman encoding [15] and whole-word de Moura encoding [19]. Every character of the original text is encoded with variable-length codewords, which consist of fixed-length base symbols. The base symbols could have 2, 4, 6, or 8 bits. Base symbols of 6 or 8 bits allow code splitting to be applied, dividing the bits of each base symbol into two parts, which are stored, respectively, into two arrays in order to speed up searching.

We presented a new search algorithm for stopper encoding, based on the SBNDM2 algorithm [10]. Stopper encoding is not restricted to exact matching nor to this search algorithm, but it can be applied to string matching problems of other types as well.

We tested our algorithm experimentally. The running time of the search algorithm was compared with two reference algorithms for compressed texts as well as two other reference algorithms for uncompressed texts. Our SBNDM2 based search method SBNDM2-SE$_4$ was the fastest among all the tested algorithms for English patterns of *at least 9* characters and either SBNDM2-SE$_4$ or SBNDM2-SE$_{8,4}$ for Finnish patterns of *at least 11* characters. Especially, SBNDM2-SE$_4$ was faster than the standard search algorithms in uncompressed texts for these patterns.

References

1. A. Amir and G. Benson: *Efficient two-dimensional compressed matching*, in Proc. of DCC, IEEE, 1992, pp. 279–288.
2. A. Amir, G. Benson, and M. Farach: *Let sleeping files lie: Pattern matching in z-compressed files*. J. Comput. Syst. Sci., 52(2) 1996, pp. 299–307.
3. R. S. Boyer and J. S. Moore: *A fast string searching algorithm*. Commun. ACM, 20(10) 1977, pp. 762–772.
4. N. R. Brisaboa, A. Fariña, G. Navarro, and J. R. Paramá: *Lightweight natural language text compression*. Inf. Retr., 10(1) 2007, pp. 1–33.
5. N. R. Brisaboa, A. Fariña, G. Navarro, and J. R. Paramá: *New adaptive compressors for natural language text*. Softw: Pract. Exper., 38(13) 2008, pp. 1429–1450.
6. N. R. Brisaboa, A. Fariña, G. Navarro, and J. R. Paramá: *Dynamic lightweight text compression*. ACM Trans. Inf. Syst., 28(3) 2010.
7. M. Crochemore and W. Rytter: *Jewels of stringology*, World Scientific, 2002.
8. J. S. Culpepper and A. Moffat: *Enhanced byte codes with restricted prefix properties*, in Proc. SPIRE, vol. 3772 of LNCS, Springer, 2005, pp. 1–12.
9. J. S. Culpepper and A. Moffat: *Phrase-based pattern matching in compressed text*, in Proc. SPIRE, vol. 4209 of LNCS, Springer, 2006, pp. 337–345.
10. B. Ďurian, J. Holub, H. Peltola, and J. Tarhio: *Improving practical exact string matching*. Inf. Process. Lett., 110(4) 2010, pp. 148–152.
11. P. Ferragina and G. Manzini: *An experimental study of an opportunistic index*, in Proc. SODA, ACM/SIAM, 2001, pp. 269–278.
12. K. Fredriksson and F. Nikitin: *Simple compression code supporting random access and fast string matching*, in Proc. WEA, vol. 4525 of LNCS, Springer, 2007, pp. 203–216.
13. P. Gage: *A new algorithm for data compression*. C Users J., 12(2) Feb. 1994, pp. 23–38.
14. R. N. Horspool: *Practical fast searching in strings*. Softw: Pract. Exper., 10(6) 1980, pp. 501–506.
15. D. Huffman: *A method for the construction of minimum-redundancy codes*. Proceedings of the IRE, 40(9) Sept 1952, pp. 1098–1101.
16. A. Hume and D. Sunday: *Fast string searching*. Softw: Pract. Exper., 21(11) 1991, pp. 1221–1248.
17. D. E. Knuth, J. H. Morris Jr., and V. R. Pratt: *Fast pattern matching in strings*. SIAM J. Comput., 6(2) 1977, pp. 323–350.
18. S. Maruyama, Y. Tanaka, H. Sakamoto, and M. Takeda: *Context-sensitive grammar transform: Compression and pattern matching*, in Proc. SPIRE, vol. 5280 of LNCS, Springer, 2008, pp. 27–38.
19. E. S. de Moura, G. Navarro, N. Ziviani, and R. A. Baeza-Yates: *Fast and flexible word searching on compressed text*. ACM Trans. Inf. Syst., 18(2) 2000, pp. 113–139.
20. G. Navarro, T. Kida, M. Takeda, A. Shinohara, and S. Arikawa: *Faster approximate string matching over compressed text*, in DCC, IEEE, 2001, pp. 459–468.
21. G. Navarro and M. Raffinot: *Fast and flexible string matching by combining bit-parallelism and suffix automata*. ACM Journal of Experimental Algorithmics, 5 2000, p. 4.

22. G. Navarro and J. Tarhio: *Boyer-Moore string matching over Ziv-Lempel compressed text*, in Proc. CPM, vol. 1848 of LNCS, Springer, 2000, pp. 166–180.
23. J. Rautio: *Context-dependent stopper encoding*, in Proc. Stringology, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University, 2005, pp. 143–152.
24. J. Rautio, J. Tanninen, and J. Tarhio: *String matching with stopper encoding and code splitting*, in Proc. CPM, vol. 2373 of LNCS, Springer, 2002, pp. 42–52.
25. Y. Shibata, T. Matsumoto, M. Takeda, A. Shinohara, and S. Arikawa: *A Boyer-Moore type algorithm for compressed pattern matching*, in Proc. CPM, vol. 1848 of LNCS, Springer, 2000, pp. 181–194.
26. I. H. Witten, A. Moffat, and T. C. Bell: *Managing Gigabytes: Compressing and Indexing Documents and Images, Second Edition*, Morgan Kaufmann, 1999.
27. S. Wu and U. Manber: *Agrep – a fast approximate pattern-matching tool*, in Proc. USENIX Technical Conference, Winter 1992, pp. 153–162.
28. J. Ziv and A. Lempel: *A universal algorithm for sequential data compression.* IEEE Transactions on Information Theory, 23(3) 1977, pp. 337–343.
29. J. Ziv and A. Lempel: *Compression of individual sequences via variable-rate coding.* IEEE Transactions on Information Theory, 24(5) 1978, pp. 530–536.