Simple Compression Code Supporting Random Access and Fast String Matching

Kimmo Fredriksson* and Fedor Nikitin

Department of Computer Science and Statistics, University of Joensuu
PO Box 111, FIN–80101 Joensuu, Finland
{kfredrik,fnikitin}@cs.joensuu.fi

Abstract. Given a sequence S of n symbols over some alphabet Σ, we develop a new compression method that is (i) very simple to implement; (ii) provides $O(1)$ time random access to any symbol of the original sequence; (iii) allows efficient pattern matching over the compressed sequence. Our simplest solution uses at most $2h + o(h)$ bits of space, where $h = n(H_0(S) + 1)$, and $H_0(S)$ is the zeroth-order empirical entropy of S. We discuss a number of improvements and trade-offs over the basic method. The new method is applied to text compression. We also propose average case optimal string matching algorithms.

1 Introduction

The aim of *compression* is to represent the given data (a sequence) using as little space as possible. This is achieved by discovering and utilizing the redundancies of the input. Some well-known compression algorithms include Huffman coding [14], arithmetic coding [26], Ziv-Lempel [28] and Burrows-Wheeler compression [6]. Recently algorithms that can compress the input sequence close to the information theoretic minimum size and still allow retrieving any symbol (actually a short substring) of the original sequence in constant time have been proposed [10, 24]. These are relatively complex and not well suited for the applications we are considering. The task of *compressed pattern matching* [1, 17, 18] is to report all the occurrences of a given pattern in a compressed text. In this paper we give a new compression method for sequences. The main traits of the method are its extreme simplicity, good compression ratio on natural language, it provides constant time random access to any symbol of the original sequence and allows average optimal time pattern matching over the compressed sequence without decompression. We give several compression methods, having different space/time trade-offs. We analyze the compression ratios, and give several string matching algorithms to search a pattern over the compressed text. Our compression method is somewhat related to ETDC compression [3] for natural language texts.

* Supported by the Academy of Finland, grant 207022.

2 Preliminaries

Let $S[0 \dots n-1] = s_0, s_1, s_2, \dots, s_{n-1}$ be a sequence of symbols over an alphabet Σ of size $\sigma = |\Sigma|$. For a binary sequence $B[0 \dots n-1]$ the function $\texttt{rank}_b(B, i)$ returns the number of times the bit b occurs in $B[0 \dots i]$. Function $\texttt{select}_b(B, i)$ is the inverse, i.e. it gives the index of the ith bit that has value b. Note that for binary sequences $\texttt{rank}_0(B, i) = i + 1 - \texttt{rank}_1(B, i)$. Both `rank` and `select` can be computed in $O(1)$ time with only $o(n)$ bits of space in addition to the original sequence taking n bits [15, 19]. It is also possible to achieve $nH_0(B) + o(n)$ total space, where $H_0(B)$ is the zero-order entropy of B [22, 23], or even $H_k(B) + o(n)$ [24], while retaining the $O(1)$ query times.

The zeroth-order empirical entropy of the sequence S is defined to be

$$H_0(S) = -\sum_{s \in \Sigma} \frac{f(s)}{n} \log_2 \left(\frac{f(s)}{n} \right), \tag{1}$$

where $f(s)$ denotes the number of times s appears in S. The k-th order empirical entropy is

$$H_k(S) = -\sum_{i=0}^{n-1} p_i \log_2(p_i), \tag{2}$$

where $p_i = \text{Probability}(s_i \mid s_{i-k}, \dots, s_{i-1})$. In other words, the symbol probabilities depend on the context they appear on, i.e. on which are the previous k symbols in S. Obviously, $H_k(S) \leq H_0(S)$.

3 Simple dense coding

Our compression scheme first computes the frequencies of each alphabet symbol appearing in S. Assume that the symbol $s_i \in \Sigma$ occurs $f(s_i)$ times. The symbols are then sorted by their frequency, so that the most frequent symbol comes first. Let this list be $s_{i_0}, s_{i_1}, \dots, s_{i_{\sigma-1}}$, i.e. $i_0 \dots i_{\sigma-1}$ is a permutation of $\{0, \dots, \sigma-1\}$.

The coding scheme assigns binary codes with different lengths for the symbols as follows. We assign 0 for s_{i_0} and 1 for s_{i_1}. Then we use all binary codes of length 2. In that way the symbols $s_{i_2}, s_{i_3}, s_{i_4}, s_{i_5}$ get the codes $00, 01, 10, 11$, correspondingly. When all the codes with length 2 are exhausted we again increase length by 1 and assign codes of length 3 for the next symbols and so on until all symbols in the alphabet get their codes.

Theorem 1. *For the proposed coding scheme the following holds:*

1. *The binary code for the symbol $s_{i_j} \in \Sigma$ is of length $\lfloor \log_2(j+2) \rfloor$.*
2. *The code for the symbol $s_{i_j} \in \Sigma$ is binary representation of the number $j + 2 - 2^{\lfloor \log_2(j+2) \rfloor}$ of $\lfloor \log_2(j+2) \rfloor$ bits.*

Proof. Let a_ℓ and b_ℓ be indices of the first and the last symbol in alphabet Σ, which have the binary codes of length ℓ. We have $a_1 = 0$ and $b_1 = 1$. The values a_ℓ and b_ℓ for $\ell > 1$ can be defined by recurrent formulas

$$a_\ell = b_{\ell-1} + 1, \quad b_\ell = a_\ell + 2^\ell - 1. \tag{3}$$

In order to get the values a_ℓ and b_ℓ as functions of ℓ, we first substitute the first formula in (3) to the second one and have

$$b_\ell = b_{\ell-1} + 2^\ell. \tag{4}$$

By applying the above formula many times we have a series

$$\begin{aligned} b_\ell &= b_{\ell-2} + 2^{\ell-1} + 2^\ell, \\ b_\ell &= b_{\ell-3} + 2^{\ell-2} + 2^{\ell-1} + 2^\ell, \\ &\dots \\ b_\ell &= b_1 + 2^2 + 2^3 + \ldots + 2^\ell. \end{aligned}$$

Finally, b_ℓ as a function of ℓ becomes

$$b_\ell = 1 + \sum_{k=2}^{\ell} 2^k = \sum_{k=0}^{\ell} 2^k - 2 = 2^{\ell+1} - 3. \tag{5}$$

Using (3) we get

$$a_\ell = 2^\ell - 3 + 1 = 2^\ell - 2. \tag{6}$$

If j is given the length of the code for s_{i_j} is defined equal to ℓ, satisfying

$$a_\ell \le j \le b_\ell. \tag{7}$$

According to above explicit formulas for a_ℓ and b_ℓ we have

$$2^\ell - 2 \le j \le 2^{\ell+1} - 3 \quad \Longleftrightarrow \quad 2^\ell \le j + 2 \le 2^{\ell+1} - 1, \tag{8}$$

and finally

$$\ell \le \log_2(j+2) \le \log_2(2^{\ell+1} - 1), \tag{9}$$

whose solution is easily seen to be $\ell = \lfloor \log_2(j+2) \rfloor$. For the setting the second statement it is sufficient to observe that the code for the symbol $s_j \in \Sigma$ is $j - a_\ell$. By applying simple transformations we have

$$j - a_\ell = j - (2^\ell - 2) = j + 2 - 2^\ell = j + 2 - 2^{\lfloor \log_2(j+2) \rfloor}.$$

So, the second statement is also proved. □

The whole sequence is then compressed just by concatenating the codewords for each of the symbols of the original sequence. We denote the compressed binary sequence as $S' = S'[0 \dots h-1]$, where h is the number of bits in the sequence. Table 1 illustrates.

Table 1. Example of compressing the string `banana`.

$S = \texttt{banana}$	$f(\texttt{a}) = 3$	$C[\texttt{a}] = 0 = 0_2$	$T[0][0] = \texttt{a}$
$S' = 0001010$	$f(\texttt{n}) = 2$	$C[\texttt{n}] = 1 = 1_2$	$T[0][1] = \texttt{n}$
$D = 1011111$	$f(\texttt{b}) = 1$	$C[\texttt{b}] = 0 = 00_2$	$T[1][0] = \texttt{b}$

3.1 Constant time random access to the compressed sequence

The seemingly fatal problem of the above approach is that the codes are not *prefix codes*, and we have not used any delimiting method to mark the codeword boundaries, and hence the original sequence would be impossible to obtain. However, we also create an auxiliary binary sequence $D[0 \dots h-1]$, where h is the length of S' in bits. $D[i] = 1$ iff $S'[i]$ starts a new codeword, and 0 otherwise, see Table 1. We also need a symbol table T, such that for each different codeword length we have table of the possible codewords of the corresponding length. In other words, we have a table $T[0 \dots \lfloor \log_2(\sigma+1) \rfloor - 1]$, such that table $T[i][0 \dots 2^{i+1} - 1]$ lists the codewords of length i. Then, given a bit-string r, $T[|r|-1][r]$ gives the decoded symbol for codeword r. This information is enough for decoding. However, D also gives us *random access to any codeword of* S'. That is, the ith codeword of S' starts at the bit position $\texttt{select}_1(D, i)$, and ends at the position $\texttt{select}_1(D, i+1) - 1$. This in turn allows to access any symbol of the original sequence S in constant time. The bit-string

$$r = S'[\texttt{select}_1(D, i) \dots \texttt{select}_1(D, i+1) - 1] \tag{10}$$

gives us the codeword for the ith symbol, and hence $S[i] = T[|r|-1][r]$, where $|r|$ is the length of the bitstring r. Note that $|r| = O(\log(n))$ and hence in the RAM model of computation r can be extracted in $O(1)$ time. We call the method Simple Dense Coding (SDC). We note that similar idea (in somewhat different context) as our D vector was used in [11] with Huffman coding. However, the possibility was already mentioned in [15].

3.2 Space complexity

The number of bits required by S' is

$$h = \sum_{j=0}^{\sigma-1} f(s_{i_j}) \ \lfloor \log_2(j+2) \rfloor, \tag{11}$$

and hence the average number of bits per symbol is h/n.

Theorem 2. *The number of bits required by S' is at most $n(H_0(S)+1)$.*

Proof. The zero-order empirical entropy of S is

$$-\sum_{j=0}^{\sigma-1} \frac{f(s_{i_j})}{n} \log_2 \left(\frac{f(s_{i_j})}{n} \right), \tag{12}$$

and thus

$$n(H_0(S)+1) = n\sum_{j=0}^{\sigma-1}\frac{f(s_{i_j})}{n}\log_2\left(\frac{n}{f(s_{i_j})}\right)+n = \sum_{j=0}^{\sigma-1} f(s_{i_j})\left(\log_2\frac{n}{f(s_{i_j})}+1\right). \tag{13}$$

We will show that the inequality

$$\lfloor \log_2(j+2)\rfloor \le \log_2(j+2) \le \left(\log_2\left(\frac{n}{f(s_{i_j})}\right)+1\right) = \log_2\left(\frac{2n}{f(s_{i_j})}\right) \tag{14}$$

holds for every j, which is the same as

$$j+2 \le \frac{2n}{f(s_{i_j})} \iff (j+2)f(s_{i_j}) \le 2n. \tag{15}$$

Note that for $j=0$ the maximum value for $f(s_{i_j})$ is $n-\sigma+1$, and hence the inequality holds for $j=0$, $\sigma \ge 2$. In general, we have that $f(s_{i_{j+1}}) \le f(s_{i_j})$, so the maximum value for $f(s_{i_1})$ is $n/2$, since otherwise it would be larger than $f(s_{i_0})$, a contradiction. In general $f(s_{i_j}) \le n/(j+1)$, and the inequality becomes

$$(j+2)f(s_{i_j}) \le 2n \iff (j+2)n/(j+1) \le 2n \iff (j+2)/(j+1) \le 2, \tag{16}$$

which holds always. □

In general, our coding cannot achieve $H_0(S)$ bits per symbol, since we cannot represent fractional bits (as in arithmetic coding). However, if the distribution of the source symbols is not very skewed, it is possible that $h/n < H_0(S)$. This does not violate the information theoretic lower bound, since in addition to S' we need also the bit sequence D, taking another h bits. Therefore the total space we need is $2h$ bits, which is at most $2n(H_0(S)+1)$ bits. However, this can be improved.

Note that we do not actually need D, but only a data structure that can answer $\texttt{select}_1(D,i)$ queries in $O(1)$ time. This is possible using just $h' = hH_0(D) + o(n) + O(\log\log(h))$ bits of space [23]. Therefore the total space we need is only $h+h'$ bits. $H_0(D)$ is easy to compute as we know that D has exactly n bits set to 1, and $h-n$ bits to 0. Hence

$$H_0(D) \;=\; -\frac{n}{h}\log_2\left(\frac{n}{h}\right) \;-\; \frac{h-n}{h}\log_2\left(\frac{h-n}{h}\right). \tag{17}$$

This means that $H_0(D)$ is maximized when $\frac{n}{h} = \frac{1}{2}$, but on the other hand h' depends also on h. Thus, h'/h shrinks as h grows, and hence for increasing $H_0(S)$ (or for non-compressible sequences, in the worst case $H_0(S) = \log_2(\sigma)$) the contribution of $hH_0(D)$ to the total size becomes more and more negligible.

Finally, the space for the symbol table T is $\sigma\lceil \log_2(\sigma)\rceil$ bits, totally negligible in most applications. However, see Sec. 3.5 and Sec. 3.6 for examples of large alphabets.

3.3 Trade-offs between h and h'

So far we have used the minimum possible number of bits for the codewords. Consider now that we round each of the codeword lengths up to the next integers divisible by some constant u, i.e. the lengths are of the form $i \times u$, for $i = \{1, 2, \ldots, \lceil \log_2(\sigma) \rceil / u\}$. So far we have used $u = 1$. Using $u > 1$ obviously only increases the length of S', the compressed sequence. But the benefit is that each of the codewords in S' can start only at positions of the form $j \times u$, for $j = \{0, 1, 2, \ldots\}$. This has two consequences:

1. the bit sequence D need to store only every uth bit;
2. every removed bit is a 0 bit.

The item (2) means that the probability of 1-bit occurring increases to $\frac{n}{h/u}$. The extreme case of $u = \log_2(\sigma)$ turns D into a vector of n 1-bits, effectively making it (and S') useless. However, if we do not compress D, then the parameter u allows easy optimization of the total space required. Notice that when using $u > 1$, the codeword length becomes

$$\lfloor \log_2((2^u - 1)j + 2^u) \rfloor_u \leq \log_2((2^u - 1)j + 2^u) \tag{18}$$

bits, where $\lfloor x \rfloor_u = \lfloor x/u \rfloor u$. Then we have the following:

Theorem 3. *The number of bits required by S' is at most $n(H_0(S) + u)$.*

Proof. The theorem is easily proved by following the steps of the proof of Theorem 2. □

The space required by D is then at most $n(H_0(S) + u)/u$ bits. Summing up, the total space is optimized for $u = \sqrt{H_0(S)}$, which leads to total space of

$$n\left(H_0(S) + 2\sqrt{H_0(S)} + 1\right) + o\left(n\left(\sqrt{H_0(S)} + 1\right)\right) \tag{19}$$

bits, where the last term is for the $\texttt{select}_1$ data structure [19].

Note that $u = 7$ would correspond to byte based End Tagged Dense Code (ETDC) [3] if we do not compress D. By compressing D our space is smaller and we also achieve random access to any codeword, see Sec. 3.5.

3.4 Context based modelling

We note that we could easily use context based modelling to obtain h of the form $nH_k(S)$. The only problem is that for large alphabets k must be quite small, since the symbol table size is multiplied by σ^k. This can be controlled by using a k that depends on S. For example, using $k = \frac{1}{2}\log_\sigma(n)$ the space complexity is multiplied by $\sqrt{n}$, negligible for constant size alphabets.

3.5 Word alphabets

Our method can be used to obtain very good compression ratios for natural language texts by using the σ distinct words of the text as the alphabet. By Heaps' Law [12], $\sigma = n^{\alpha}$, where n is the total number of words in the text, and $\alpha < 1$ is language dependent constant, for English $\alpha = 0.4 \ldots 0.6$. These words form a dictionary $W[0 \ldots \sigma - 1]$ of σ strings, sorted by their frequency. The compression algorithm then codes the jth most frequent word as an integer j using $\lfloor \log_2(j + 2) \rfloor$ bits. Again, the bit-vector D provides random access to any word of the original text.

As already mentioned, using $u = 7$ corresponds to the ETDC method [3]. ETDC uses 7 bits in each 8 bit byte to encode the codewords similarly as in our method. The last bit is saved for a flag that indicates whether the current byte is the last byte of the codeword. Our benefit is that as we store these flag bits into a separate vector D, we can compress D as well, and simultaneously obtain random access to the original text words.

3.6 Self-delimiting integers

Assume that S is a sequence of integers in range $\{0, \ldots, \sigma - 1\}$. Note that our compression scheme can be directly applied to represent S succinctly, even without assigning the codewords based on the frequencies of the integers. In fact, we can just directly encode the number $S[i]$ with $\lfloor \log_2(S[i]+2) \rfloor$ bits, and again using the auxiliary sequence D to mark the starting positions of the codewords. This approach does not need any symbol tables, so the space requirement does not depend on σ. Still, if σ and the entropy of the sequence is small, we can resort to codewords based on the symbol frequencies.

This method can be used to replace e.g. Elias δ-coding [8], which achieves

$$\lfloor \log_2(x) \rfloor + 2 \lfloor \log_2(1 + \lfloor \log_2(x) \rfloor) \rfloor + 1 \tag{20}$$

bits to code an integer x. Elias codes are self-delimiting prefix codes, so the sequence can be uniquely decompressed. However, Elias codes do not provide constant time random access to the ith integer of the sequence.

Again, we can use u to tune the space complexity of our method.

4 Random access Fibonacci coding

In this section we present another coding method that does not need the auxiliary sequence D. The method is a slight modification of the technique used in [11]. We also give analysis of the compression ratio achieved with this coding. Fibonacci coding uses the well-known Fibonacci numbers. Fibonacci numbers $\{f_n\}_{n=1}^{\infty}$ are the positive integers defined recursively as $f_n = f_{n-1} + f_{n-2}$, where $f_1 = f_2 = 1$. The Fibonacci numbers also have a closed-form solution called Binet's formula:

$$F(n) = (\phi^n - (1 - \phi)^n)/\sqrt{5} \tag{21}$$

where $\phi = \frac{1+\sqrt{5}}{2} \approx 1.61803$ is the golden ratio.

Zeckendorf's theorem [4] states that for any positive integer x there exists unique representation as

$$x = \sum_{i=0}^{k} f_{c_i} \tag{22}$$

where $c_i \geq c_{i-1} + 2$ for any $i \geq 1$. The last condition means that the sequence $\{f_{c_i}\}$ does not contain two consecutive Fibonacci numbers. Moreover, the Zeckendorf's representation of integer can be found by a greedy heuristic. The Fibonacci coding of positive integers uses the Zeckendorf's representation of integer. The code for x is a bit stream of length $\ell(x) + 1$, where

$$\ell(x) = \max\{i \mid f_i \leq x\} \tag{23}$$

The last bit in the position $\ell(x) + 1$ is set to 1. The value of ith bit is set to 1 if the Fibonacci number f_i occurred in Zeckendorf's representation and is set to 0, otherwise. By definition, the bit in position $\ell(x)$ is always set to 1. Hence, at the end of the codeword we have two consecutive ones. On the other hand two consecutive ones can not appear anywhere else within codeword. This allows us to distinguish the codewors for the separate symbols in the encoded sequence, moreover the code satisfies the prefix property.

The Fibonacci dual theorem [5] states that in Zeckendorf's representation the first Fibonacci number never occurs in the representation. In other words, we can skip the first bit reserved for the first Fibonacci number and therefore we can make the codewords shorter. Thus we can obtain the following Lemma.

Lemma 1. *The length $|c(x)|$ of the Fibonacci codeword $c(x)$ for a positive integer x satisfies $|c(x)| \leq \log_\phi(\sqrt{5}x)$.*

Proof. Obviously, the worst case for the code length is when the value x is a Fibonacci number itself. Then the code is sequence of $\ell(x) - 2$ zeroes ending with two ones. Thus, we have to estimate the value $\ell(x)$, supposing that x is Fibonacci number. Using the Binet's formula we have

$$x = (\phi^{\ell(x)} - (1 - \phi)^{\ell(x)})/\sqrt{5} \tag{24}$$

Taking logarithms from the both sides we get, after some algebra:

$$\log_2(\sqrt{5}x) = \ell(x)\log_2(\phi) + \log_2(1 + ((\phi - 1)/\phi)^{\ell(x)}). \tag{25}$$

Due to that $0 < (\phi - 1)/\phi < 1$ we have that $\log_2(\sqrt{5}x) \geq \ell(x)\log_2(\phi)$, and hence as $\phi > 1$ we have $\ell(x) \leq \log_2(\sqrt{5}x)/\log_2(\phi)$. □

Theorem 4. *The sequence S encoded with Fibonacci coding can be represented in*

$$h \;=\; n\,\frac{H_0(S) + \log_2(\sqrt{5})}{\log_2(\phi)}$$

bits.

Proof. Follows from Lemma 1. □

4.1 Random access to Fibonacci coded sequences

As it was mentioned we have one attractive property of the Fibonacci code. The two consecutive ones can only appear at the end of the codeword and nowhere else (however, it is possible that the encoded *sequence* has more than two consecutive one bits, namely when codeword `11` is repeated).

If we want to start encoding from the ith symbol we should find the $(i-1)$th pair of two ones, assuming that the pairs are not overlapped. When the position j of this pair is defined we can start decoding from the position $j+2$. Thus, for our task it is enough to be able to determine the position of $(i-1)$th pair of non-overlapping ones in constant time. The query which does it we denote as $\texttt{select}_{11}$. Notice that as we do not allow the pairs to be overlapped, this query does not answer the question where the certain occurrence of substring `11` starts in the bitstream and it differs from the extended `select` query presented in [16]. The data structure for the $\texttt{select}_{11}$ query can be constructed using the same idea as for classical $\texttt{select}_1$ query solution presented by Clark [7].

The important remark which makes the Clark's approach applicable is that during construction of the block directories every sequence `11` of interest is included entirely in range. It allows us to use look-up tables, as the situation of the sequence `11` belonging into two ranges at the same time is impossible. We omit the details, but by following the Clark's construction, we can build a data structure taking $o(h)$ bits of space (in additional to the original sequence) and supporting $\texttt{select}_{11}$ queries in $O(1)$ time.

5 Fast string matching

We now present several efficient string matching algorithms working on the compressed texts. We assume SDC here, altough the algorithms work with minor modification in Fibonacci coded texts as well. The basic idea of all the algorithms is that we compress the pattern using the same method and dictionary as for compressing the text, so as to be able to directly comprare a text substring against the pattern. We denote the compressed text and the auxiliary bitvector as S' and $D_{S'}$, both consisting of h bits. For clarity of presentation, we assume that $u = 1$. Likewise, the compressed sequences for the pattern are denoted as P' and $D_{P'}$, of m bits.

5.1 BMH approach

The well-known Boyer-Moore-Horspool algorithm (BMH) [13] works as follows. The pattern P is aligned against a text window $S[i-m+1 \ldots i]$ The invariant is that every occurrence of P ending before the position i is already reported. Then the symbol $P[m-1]$ (i.e. the last symbol of P) is compared against $S[i]$. If they match, the whole pattern is compared against the window, and a possible match is reported. Then the next window to be compared is $S[i-m+1+\mathit{shift} \ldots i+\mathit{shift}]$

Alg. 1 SearchBMH(T', D_T, n, P', D_P, m)

```
1   b ← O(log(m))
2   for i ← 0 to (1 << b) − 1 do shift[i] ← m
3   for i ← 1 to b − 1 do
4       c ← P[0 . . . i − 1]
5       for j ← 0 to (1 << (b − i)) − 1 do shift[(c << (b − i)) | j] ← m − i
6   for i ← 0 to m − b − 1 do shift[P'[i . . . i + b − 1]] = m − i − b
7   a ← P'[m − b . . . m − 1]
8   occ ← 0
9   for i ← m − 1 to n − 1 do
10      c ← T'[i − b + 1 . . . i]
11      if a = c AND D_T[i − m + 1 . . . i] = D_P AND T'[i − m + 1 . . . i] = P' then occ ← occ + 1
12      i ← i + shift[c]
13  return occ
```

(regardless of whether $S[i]$ was equal to $P[m-1]$ or not), where *shift* is computed as

$$shift = m - \max(j \mid P[j] = S[i], 0 \leq j < m-1) \tag{26}$$

If $S[i]$ does not occur in P, then the shift value is m. The shift function is easy to compute at the preprocessing time, needing $O(\sigma+m)$ time and $O(\sigma)$ space. The algorithm is very simple to implement and one of the most efficient algorithms in practice for reasonably large alphabets (say, $\sigma > m$), when the average case time approaches the best case time, i.e. $O(n/m)$. In our case, however, we have binary alphabet and the shift values yielded are close to 1. However, we can form a "super-alphabet" from the consecutive bits. That is, we can read b bits at a time and treat the bitstring as a symbol from an alphabet of size 2^b. I.e. we read the bitstring $S'[i-b+1 \ldots i]$ and compute the shift function so as to align this bitstring against its right-most occurrence in P'. If such occurrence is not found, we compare the suffixes of $S'[i-b+1 \ldots i]$ against the prefixes of $P'[0 \ldots b]$. If no occurrence is still found, the shift is again m (bits). We must still verify any occurrence by comparing $D_{S'}[i-m+1 \ldots i]$ against $D_{P'}$ to check that the codewords are synchronized. Alg. 1 shows complete pseudo code.

Theorem 5. *Alg. 1 runs in $O(m^2 + h/m)$ average time for the optimal b.*

Proof. The average time of Alg. 1 clearly depends on the parameter b. If $S'[i-b+1 \ldots i]$ does not occur in P', then the shift is at least $m-b+1$ bits. Note that in RAM model of computation obtaining the bitstring and thus computing the shift takes $O(1)$ time as long as $b = O(\log(n))$. The total time needed for these cases (1) is thus $O(h/(m-b))$. Now, assume that if $S'[i-b+1 \ldots i]$ occurs in P' we verify the whole pattern and shift only by one bit. The total time needed for these cases (2) is at most $O(hm)$ (actually only $O(hm/w)$ time, as we can compare w bits at the time, where w is the number of bits in a machine word), but only $O(h)$ on average. We therefore want to choose b so that the probability p of case (2) is low enough. The total time is therefore $O(h/(m-b) + phm)$. Assuming that the bit values have uniform distribution, $p = 1/2^b$, and the total time is optimized for

$$h/(m-b) > hm/2^b \quad \Rightarrow \quad b = \Omega(\log(m^2)), \tag{27}$$

and the average case time then becomes $O(h/m)$. The preprocessing time is $O(2^b + m)$ which is $O(m^2)$ for $b = \log_2(m^2)$. □

We note that this breaks the lower bound of $O(h \log(m)/m)$, which is based on comparison model [27], and is thus optimal. However, our method is not based on comparing single symbols and we effectively avoid the $\log(m)$ term by "comparing" b symbols at a time. On the other hand, it is easy to see that increasing b beyond $O(\log(m))$ does not improve our algorithm. Finally, note that other BMH variants, such as the one by Sunday [25], could be generalized just as easily.

5.2 Shift-Or and BNDM

The two well-known bit-parallel string matching algorithms Shift-Or [2] and BNDM [20] can be directly applied to our case, and even simplified: as already noted in [20], the preprocessing phase can be completely removed, as for binary alphabets the pattern itself and its bit-wise complement can serve as the preprocessed auxiliary table the algorithms need. However, we still need to verify the occurrences using the $D_{P'}$ sequence. The average case running times of Shift-Or and BNDM become $O(h)$ and $O(h \log(m)/m)$ for $m \leq w$. For longer patterns these must be multiplied by $\lceil m/w \rceil$. We omit the details for lack of space. However, we note that the "superalphabet" trick of the previous section works for these two algorithms as well (see also [9]). For example, we can improve BNDM by precomputing the steps taken by the algorithm by the first b bits read in a text window, and at the search phase we use a look-up table to perform the steps in $O(1)$ time, and then continue the algorithm normally. The average case time of BNDM is improved to $O(h\lceil \log(m)/b \rceil/m)$, and we see that $b = \log(m)$ gives again $O(h/m)$ average time. Preprocessing time and space become $O(m)$.

6 Experimental results

We have run experiments to evaluate the performance of our algorithms. The experiments were run on Celeron 1.5GHz with 512Mb of RAM, running GNU/Linux operating system. We have implemented all the algorithms in C, and compiled with `gcc 4.1.1`.

The test files are summarized in Table 2 (a), the files are from Silesia corpus[1] and Canterbury corpus[2]. We used a word based model [18]: we have two dictionaries, one for the text words and the other for "separators", where separator is defined to be any substring between two words. As there is strictly alternating order between the two, decompressing is easy as far as we know whether the text starts with a word or a separator. We used `zlib` library[3] to compress the

[1] `http://www-zo.iinf.polsl.gliwice.pl/~sdeor/corpus.htm`
[2] `http://corpus.canterbury.ac.nz/`
[3] `www.zlib.org`

Table 2. Test files and compression ratios.

(a)	Test files				
Name	Type	Size	σ (words+separators)	words	H_0(words)
dickens	English text	10,192,446 B	34,381 + 1,071	1,819,394	9.92 bits
world192.txt	English text	2,473,400 B	22,917 + 498	343,139	10.91 bits
samba	source code	6,760,204 B	29,822 + 15,544	924,640	10.40 bits
xml	xml source	5,303,867 B	19,582 + 1,495	847,806	9.10 bits

(b)	Compression ratios								
File	gzip -9	bzip2 -9	SDC	SDC W	FibC	FibC W	Huffman	ETDC	H_0
dickens	37.7%	27.4%	35.3%	29.3%	31.5%	25.4%	28.3%	32.9%	26.2%
world192.txt	29.1%	19.7%	35.5%	29.3%	31.6%	25.3%	29.0%	34.4%	24.4%
samba	20.1%	16.3%	36.1%	24.9%	32.2%	21.4%	30.3%	38.4%	27.4%
xml	12.3%	8.0%	33.0%	24.5%	30.6%	21.6%	28.7%	38.6%	26.4%

(c)	Compression ratios with and w/o `select` data structure							
	with `select`		with `select` only for words		w/o `select`, word stream only SDC			
File	SDC	FibC	SDC	FibC	$u = 1$	$u = 2$	$u = 3$	$u = 4$
dickens	39.1%	35.4%	37.1%	33.5%	29.3%	25.8%	25.2%	25.4%
world192.txt	38.6%	35.2%	37.0%	33.7%	29.3%	25.7%	24.9%	25.1%
samba	39.1%	35.4%	37.6%	33.8%	24.9%	21.7%	21.1%	21.3%
xml	36.6%	34.4%	34.7%	32.6%	24.5%	21.9%	21.6%	21.9%

dictionaries. We also experimented with a so called "space-less model", but omit the results for a lack of space.

Table 2 (b) gives the compression ratios for several different methods. The Huffman compression algorithm uses two dictionaries, while ETDC uses the space-less model. Note that SDC with $u = 7$ and spaceless model would achieve the same ratio. H_0 denotes the empirical entropy using the model of two separate dictionaries. SDC W and FibC W columns give the ratios using only the word stream (i.e. excluding the separators stream). This gives the text size for our search algorithms, since we ran them only for the words.

Table 2 (c) shows the compression ratios for SDC and FibC including the size of the `select` data structures. The values are for both streams (words and seperators), and for word stream only. We used the `darray` method [21]. For SDC coding this can be directly applied on D vector. For FibC this needs some modifications, but these are quite easy and straight-forward. The table shows also the effect of the parameter u for SDC. We show only the effect on word stream. In general we can, and should, optimize the parameter individually for words and separators, since the entropy for separators is usually much smaller. The optimum value is $u = 3$ in all cases, as can be deduced from Table 2 (a), i.e. the optimum is $\sqrt{H_0(\text{words})}$.

Fig. 1 shows the search performance using *dickens* file. We compared our algorithms against the BMH algorithm on the original uncompressed text. We used patterns consisting of $1 \ldots 4$ words and 300 patterns of each length randomly picked from the text. The compressed pattern lengths were about 6, 12, 18 and 25 bits, correspondingly. Shift-Or (SO) is quite slow, as expected (as the shift is always just 1 bit). However, BNDM, BNDMB (same as BNDM but using the parameter b) and FBMH (our BMH variant running on compressed texts)

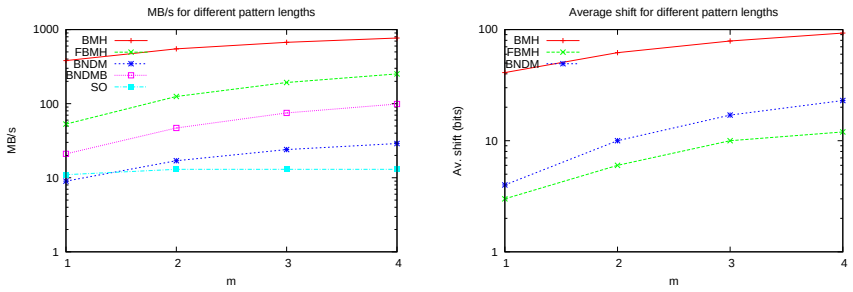

Fig. 1. Search performance. Left: MB/second processed by different algorithm. Right: the average shift in bits. The x-axis (m) is the pattern length in *words*.

achieve reasonably good performance, altough they lose to plain BMH, which has very simple implementation. For FBMH we used $b = m$ for $m \leq 10$ and $b = 10$ for larger m. For BNDMB we used $b = 2\log_2(m)$. We feel that the performance of searching in compressed texts could still be improved. In particular, using $u = 8$ allows us to use plain byte based BMH algorithm, with the exception that we have to verify the occurrences with the D vector.

7 Conclusions

We have presented a simple compression schemes that allow constant time access to any symbol of the original sequence. The method gives good compression ratio for natural language texts, and allows average-optimal time string matching without decompression.

References

1. A. Amir and G. Benson. Two-dimensional periodicity and its applications. In *Proceedings of SODA'92*, pages 440–452, 1992.
2. R. A. Baeza-Yates and G. H. Gonnet. A new approach to text searching. *Commun. ACM*, 35(10):74–82, 1992.
3. N. Brisaboa, E. Iglesias, G. Navarro, and J. Paramá. An efficient compression code for text databases. In *Proceedings of ECIR'03*, LNCS 2633, pages 468–481, 2003.
4. J. L. Brown. Zeckendorf's theorem and some applications. *Fib. Quart.*, 2:163–168.
5. J. L. Brown. A new characterization of the Fibonacci numbers. *Fib. Quart.*, 3:1–8, 1965.
6. M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.
7. D. R. Clark. *Compact Pat Trees.* PhD thesis, University of Waterloo, Ontario, Canada, 1998.
8. P. Elias. Universal codeword sets and representation of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203, 1975.
9. K. Fredriksson. Shift–or string matching with super-alphabets. *Information Processing Letters*, 87(1):201–204, 2003.

10. R. González and G. Navarro. Statistical encoding of succinct data structures. In *Proceedings of the 17th Annual Symposium on Combinatorial Pattern Matching (CPM 2006)*, LNCS 4009, pages 295–306, 2006.
11. Sz. Grabowski, G. Navarro, R. Przywarski, A. Salinger, and V. Mäkinen. A simple alphabet-independent FM-index. *International Journal of Foundations of Computer Science (IJFCS)*, 17(6):1365–1384, 2006.
12. H. S. Heaps. *Information retrieval: theoretical and computational aspects.* Academic Press, New York, NY, 1978.
13. R. N. Horspool. Practical fast searching in strings. *Softw. Pract. Exp.*, 10(6):501–506, 1980.
14. D. A. Huffman. A method for the construction of minimum redundancy codes. *Proceedings of I.R.E*, 40:1098–1101, 1951.
15. G. Jacobson. *Succinct static data structures.* PhD thesis, Carnegie Mellon University, 1989.
16. V. Mäkinen and G. Navarro. Rank and select revisited and extended. *Theoretical Computer Science*, 2006. Special issue on "The Burrows-Wheeler Transform and its Applications". To appear.
17. U. Manber. A text compression scheme that allows fast searching directly in the compressed file. *ACM Trans. Inform. Syst.*, 15(2):124–136, 1997.
18. E. Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Fast and flexible word searching on compressed text. *ACM Transactions on Information Systems (TOIS)*, 18(2):113–139, 2000.
19. J. Ian Munro. Tables. In *Proceedings of FSTTCS'96*, LNCS 1180, pages 37–42. Springer, 1996.
20. G. Navarro and M. Raffinot. Fast and flexible string matching by combining bit-parallelism and suffix automata. *ACM Journal of Experimental Algorithmics (JEA)*, 5(4), 2000.
21. D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. In *Proceedings of ALENEX'07*. ACM Press, 2007.
22. R. Pagh. Low redundancy in static dictionaries with o(1) worst case lookup time. In *Proceedings of ICALP'99*, pages 595–604. Springer-Verlag, 1999.
23. R. Raman, V. Raman, and S. S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proceedings of SODA'02*, pages 233–242. ACM Press, 2002.
24. K. Sadakane and R. Grossi. Squeezing succinct data structures into entropy bounds. In *Proceedings of SODA'06*, pages 1230–1239. ACM Press, 2006.
25. D. M. Sunday. A very fast substring search algorithm. *Commun. ACM*, 33(8):132–142, 1990.
26. I. H. Witten, R. M. Neal, and J. G. Cleary. Arithmetic coding for data compression. commun. acm, 30(6):520. *Communications of the ACM*, 30(6):520, 1987.
27. A. C. Yao. The complexity of pattern matching for a random string. *SIAM J. Comput.*, 8(3):368–387, 1979.
28. J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory*, 23:337–343, 1977.