Simple Random Access Compression

Kimmo Fredriksson
Department of Computer Science
University of Kuopio
P.O. Box 1627, 70211 Kuopio, Finland
kimmo.fredriksson@uku.fi

Fedor Nikitin
Saint-Petersburg State University, Faculty of Applied Mathematics and Control Processes
Universitetskii prospekt 35, Petergof, Saint-Petersburg, Russia 198504
fedor.nikitin@gmail.com

Abstract. Given a sequence $S$ of $n$ symbols over some alphabet $\Sigma$ of size $\sigma$, we develop new compression methods that are (i) very simple to implement; (ii) provide $O(1)$ time random access to any symbol (or short substring) of the original 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. For example, we can achieve $n(H_k(S) + 1) + o(n(H_k(S) + 1))$ bits of space, for $k = o(\log_{\sigma}(n))$. Several applications are discussed, including text compression, (compressed) full-text indexing and string matching.

Keywords: succinct data structures; text compression; string matching; full-text indexing

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 [20], arithmetic coding [37], Ziv-Lempel [39] and block sorting (Burrows-Wheeler transformation) based compression [6]. Recently algorithms that can compress the input sequence close to the information theoretic minimum size and still allow retrieving any symbol (or a short substring) of the original sequence in constant time have been proposed [33, 16, 10]. These are either relatively complex or not well suited for the applications we are considering. Given a sequence $S$ of $n$ symbols over some alphabet $\Sigma$, all of them achieve $nH_k(S)+O\left(\frac{n}{\log \sigma(n)}(k \log(\sigma)+\log \log(n))\right)$ bits of space, where $H_k(S)$ is the $k$-th order empirical entropy of $S$, with the restriction that $k = o(\log_{\sigma}(n))$.

These methods have obvious applications in classical data compression in general, but also in compressing data structures in particular. Some examples include inverted word indexes [36], compressed suffix arrays [28], sparse dictionaries and bitmaps [27, 31], etc. In principle, these methods can be used to turn any (static) data structure into a compressed form, without sacrificing its functionality [33]. See also Sec. 5.

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 or short substring 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 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 $S$. Our main result is that we can compress $S$ to $nH_k(S) + O(n\sqrt{H_k(S)})$ or $nH_k(S)/\log_2(\phi) + O(n)$ bits of space, where $H_k(S)$ is the $k$-th order empirical entropy, and $\phi$ is the golden ratio, for any $k = o(\log_{\sigma}(n))$, such that we can retrieve any substring of length $O(\log_{\sigma}(n))$ of $S$ in $O(1)$ time. We show experimentally that our method gives very competitive compression ratios for natural language texts. We also give analysis for one of the full-text indexes given in [17]. Finally, we note that in the case we want to decompress or otherwise access the compressed sequence sequentially (rather than randomly), we can simply drop out some auxiliary data structures, and obtain an even simpler method.

Some historical remarks. Preliminary version of this work appeared in a Technical Report [13] (2006). Almost parallel and independently (SODA 2007), Ferragina and Venturini published a method [10] that is very similar to one of our methods, and again a few months later an extended work of our TR appeared in WEA 2007 [14]. Related ideas in different contexts have also appeared in [21, 3, 17]. This paper is an extended version of [13, 14].

2. Preliminaries

Let $S[0 \ldots n-1] = s_0, s_1, s_2, \ldots, s_{n-1}$ be a sequence of symbols over an alphabet $\Sigma$ of size $\sigma = |\Sigma|$. For a binary sequence $B[0 \ldots n-1]$ the function $\operatorname{rank}_b(B, i)$ returns the number of times the bit $b$ occurs in $B[0 \ldots i]$. Function $\operatorname{select}_b(B, i)$ is the inverse, i.e. it gives the index of the $i$-th bit that has value $b$. Note that for binary sequences $\operatorname{rank}_0(B, i) = i + 1 - \operatorname{rank}_1(B, i)$. Both $\operatorname{rank}$ and $\operatorname{select}$ can be computed in $O(1)$ time with only $o(n)$ bits of space in addition to the original sequence taking $n$ bits [21, 27]. It is also possible to achieve $nH_0(B) + o(n)$ total space, where $H_0(B)$ is the zero-order entropy of $B$ [31, 32], or even $nH_k(B) + o(n)$ [33], while retaining the $O(1)$ query times.
The zeroth-order empirical entropy of the sequence $S$ is defined to be

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

(1)

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

$$H_k(S) = \sum_{X \in \Sigma^k} \frac{f(X)}{n} H_0(S_X),$$  

(2)

where $X$ is a substring of $S$, and $f(X)$ denotes the number of occurrences of $X$ in $S$, and $S_X$ is the concatenation of the symbols occurring in $S$ just after the string $X$. The string $X$ is called the context of the following symbol. It holds that $H_k(S) \leq H_0(S) \leq \log_2(\sigma)$.

**Rank and select.** Our basic method for the random access decompression of the compressed sequence relies on the \texttt{select} data structure. In the case of sequential decompression this is not needed (contrary to [10], which still needs the equivalent of \texttt{select} even in this case), but some applications (such as string matching, which is performed sequentially) may need \texttt{rank} data structure. There are many different solutions for both, having different space and trade-offs. Improving these data structures is still an active research topic. While many of the solutions need only $o(n)$ bits of additional space and take only $O(1)$ time per query, the hidden constants may lead to significant differences in practice. Our methods takes them as a “black-box”, and thus have similar trade-offs. We do not go into all the details of the various solutions, but nevertheless present the basics to give a flavor of the main ideas.

Consider $\texttt{rank}_1(B, i)$, i.e. the number of 1 bits up to position $i$ in $B$. The basic idea is to (conceptually) partition $B$ to large blocks, of $\log^2(n)$ bits each, and again each large block to small blocks, each of $s = \log_2(n)/2$ bits. The answer for each boundary of the large block is explicitly stored in an array, which thus takes $O(n \log(n)/\log^2(n)) = O(n/\log(n)) = o(n)$ bits in total. Similarly, the explicit relative answer (i.e. from the beginning from the previous large block) is stored for each of the small block boundary, which thus takes a total of $O(n \log \log(n)/\log(n)) = o(n)$ bits. Hence the final answer can be obtained by two table lookups, and finally resorting to counting the 1 bits in one small block (up to position $i \mod s$), which can be done either with a machine instruction, or with another helper table of size $O(\sqrt{n} \log^2(n))$ bits. The \texttt{select} function can be computed similarly, but is somewhat more involved. The main difference is that the sequence $B$ itself is not partitioned directly to equal sized blocks, but rather over the possible parameter values for the query; in other words, the blocks will have equal number of 1-bits. It is also possible to compute \texttt{select} by simply doing binary search over \texttt{rank} queries. These basic ideas come in many different variations. For detailed descriptions refer e.g. to [21, 7, 27, 22, 15, 28, 30]. See also Sec. 4.1.

### 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 $s_{i_0}, s_{i_1}, \ldots, s_{i_{\sigma-1}}$, i.e. $i_0 \ldots i_{\sigma-1}$ is a permutation of $\{0, \ldots, \sigma - 1\}$.
The coding scheme assigns binary codes with different lengths for the symbols as follows. We assign $0$ for $s_{i0}$ and $1$ for $s_{i1}$. Then we use all binary codes of length $2$. In that way the symbols $s_{i2}, s_{i3}, s_{i4}, s_{i5}$ 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 3.1.** For the proposed coding scheme the following holds:

1. The binary code for the symbol $s_{ij} \in \Sigma$ is of length $\lfloor \log_2(j + 2) \rfloor$.

2. The code for the symbol $s_{ij} \in \Sigma$ 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 $a_\ell$ and $b_\ell$ be indices of the first and the last symbol in alphabet $\Sigma$, which have the binary codes of length $\ell$. We have $a_1 = 0$ and $b_1 = 1$. The values $a_\ell$ and $b_\ell$ for $\ell > 1$ can be defined by recurrent formulas

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

In order to get the values $a_\ell$ and $b_\ell$ as functions of $\ell$, we first substitute the first formula in (3) to the second one and have

$$b_\ell = b_{\ell-1} + 2^\ell. \quad (4)$$

By applying the above formula many times we have a series

$$b_\ell = b_{\ell-2} + 2^{\ell-1} + 2^\ell,$$

$$b_\ell = b_{\ell-3} + 2^{\ell-2} + 2^{\ell-1} + 2^\ell,$$

$$\ldots$$

$$b_\ell = b_1 + 2^2 + 2^3 + \ldots + 2^\ell.$$

Finally, $b_\ell$ as a function of $\ell$ becomes

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

Using (3) we get

$$a_\ell = 2^\ell - 3 + 1 = 2^\ell - 2. \quad (6)$$

If $j$ is given the length of the code for $s_{ij}$ is defined equal to $\ell$, satisfying

$$a_\ell \leq j \leq b_\ell. \quad (7)$$

According to above explicit formulas for $a_\ell$ and $b_\ell$ we have

$$2^\ell - 2 \leq j \leq 2^{\ell+1} - 3 \iff 2^\ell \leq j + 2 \leq 2^{\ell+1} - 1, \quad (8)$$
Table 1. Example of compressing the string banana.

<table>
<thead>
<tr>
<th>$S =$ banana</th>
<th>$f(a) = 3$</th>
<th>$C[a] = 00_2$</th>
<th>$T[0][0] = a$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S' = 0001010$</td>
<td>$f(n) = 2$</td>
<td>$C[n] = 11_2$</td>
<td>$T[0][1] = n$</td>
</tr>
<tr>
<td>$D = 1011111$</td>
<td>$f(b) = 1$</td>
<td>$C[b] = 00_2$</td>
<td>$T[1][0] = b$</td>
</tr>
</tbody>
</table>

and finally

$$\ell \leq \log_2(j + 2) \leq \log_2(2^{\ell+1} - 1),$$

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 $S' = S'[0 \ldots h - 1]$, where $h$ is the number of bits in the sequence. Table 1 illustrates.

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 $D[0 \ldots h - 1]$, 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 $T[0 \ldots \lfloor \log_2(\sigma + 1) \rfloor - 1]$, such that table $T[i][0 \ldots 2^{i+1} - 1]$ 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 $i$th codeword of $S'$ starts at the bit position $\text{select}_1(D, i)$, and ends at the position $\text{select}_1(D, i + 1) - 1$. This in turn allows to access any symbol of the original sequence $S$ in constant time. The bit-string

$$r = S'[\text{select}_1(D, i) \ldots \text{select}_1(D, i + 1) - 1]$$

(10)

gives us the codeword for the $i$th 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 [17] with Huffman coding. However, the possibility was already mentioned in [21].
3.2. Space complexity

The number of bits required by $S'$ is

$$h = \sum_{j=0}^{\sigma-1} f(s_{ij}) \left\lfloor \log_2(j+2) \right\rfloor,$$

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

**Theorem 3.2.** The number of bits required by $S'$ is at most $n(H_0(S) + 1)$.

**Proof:**

The zero-order empirical entropy of $S$ is

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

and thus $n(H_0(S) + 1)$ is equal to

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

We will show that the inequality

$$\left\lfloor \log_2(j+2) \right\rfloor \leq \log_2(j+2) \leq \left( \log_2 \left( \frac{n}{f(s_{ij})} \right) + 1 \right) = \log_2 \left( \frac{2n}{f(s_{ij})} \right)$$

holds for every $j$, which is the same as

$$j + 2 \leq \frac{2n}{f(s_{ij})} \iff (j+2)f(s_{ij}) \leq 2n.$$ (15)

Note that for $j = 0$ the maximum value for $f(s_{ij})$ is $n - \sigma + 1$, and hence the inequality holds for $j = 0$, $\sigma \geq 2$. In general, we have that $f(s_{ij+1}) \leq f(s_{ij})$, so the maximum value for $f(s_{ij})$ is $n/2$, since otherwise it would be larger than $f(s_{i0})$, a contradiction. In general $f(s_{ij}) \leq n/(j+1)$, and the inequality becomes

$$(j + 2)f(s_{ij}) \leq 2n \iff (j+2)n/(j+1) \leq 2n \iff (j+2)/(j+1) \leq 2,$$ (16)

which holds always. \qed

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. We note that the analysis is very pessimistic, and in
practice the constant is much less than 2 (less than 1.5 in all our experiments, see Sec. 6). However, this can be improved.

Note that we do not actually need $D$, but only a data structure that can answer 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 [32]. 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

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

This means that $H_0(D)$ is maximized when $\frac{n}{h} = \frac{1}{2}$, i.e. $H_0(D) = 1$, and the space complexity becomes

$$2n(H_0(S) + 1) + o(n) + O(\log \log(n)),$$

where $H_0(D) \leq 1$, and

$$n(H_0(S) + 1)H_0(D) \approx n \log_2 \left( \frac{(H_0(S) + 1)^{H_0(S) + 1}}{H_0(S)^{H_0(S)}} \right),$$

where the approximation is precise if $h = n(H_0(S) + 1)$ (which is pessimistic).

Finally, the space for the symbol table $T$ is $\sigma \lceil \log_2(\sigma) \rceil$ bits, totally negligible in most applications. However, see Sec. 5.2 and Sec. 5.3 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 $i = \{1, 2, \ldots, \lceil \log_2(\sigma + 1) \rceil / u\}$. 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 $u$th 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 $u = \lceil \log_2(\sigma) \rceil$ 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

$$\lceil \log_2((2^u - 1)j + 2^u) \rceil_u \leq \log_2((2^u - 1)j + 2^u)$$

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

**Theorem 3.3.** 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 3.2.

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

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

(21) bits, where the last term is for the select data structure [27].

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. 5.2.

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 [17]. 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:

$$F(n) = \frac{(\phi^n - (1 - \phi)^n)}{\sqrt{5}}$$

(22)

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

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

(23)

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

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

(24)

The last bit in the position $\ell(x) + 1$ is set to 1. The value of $i$th 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 codewords 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 4.1. The length $|c(x)|$ of the Fibonacci codeword $c(x)$ for a positive integer $x$ satisfies $|c(x)| \leq \log_\phi(\sqrt{5}x) + 1/3$.

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

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

(25)

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

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

(26)

The term $-\log_\phi(1 - ((1 - \phi)/\phi)^{\ell(x)})$ is maximized for $\ell(x) = 2$, where it becomes $1 - \log_\phi(2 - 1/\phi) < 1/3$.

Lemma 4.2. The average length $\overline{c}$ of the Fibonacci codewords for a sequence $S$ is $(H_0(S) + \log_2(\sqrt{5}) + 1/3)/\log_2(\phi)$.

Proof: The average code length is given by

$$\overline{c} = \frac{1}{n} \sum_{j=0}^{\sigma-1} \frac{f(s_j)}{n} \ell(j) \leq \frac{1}{\log_2(\phi)} \sum_{j=0}^{\sigma-1} \frac{f(s_j)}{n} \left( \log_2(j) + \log_2(\sqrt{5}) + 1/3 \right)$$

(27)

where the inequality is obtained by applying Lemma 4.1. As $f(s_j) \geq f(s_{j+1})$ and $\sum f(s_j) = n$, we have that $j f(s_j) \leq n$, and hence $j \leq n/f(s_i)$. Substituting this in the above sum, we obtain the claim.

Theorem 4.1. The sequence $S$ encoded with Fibonacci coding can be represented in

$$h = n \frac{H_0(S) + \log_2(\sqrt{5}) + 1/3}{\log_2(\phi)}$$

bits.

Proof: Follows from Lemma 4.2.

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 decoding from the $i$th 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 \text{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 \textit{select} query presented in [23]. The data structure for the \text{select}_{11}(S', i) query can be constructed using the
same idea solution as for classical \text{select}_{1} query presented by Clark [7]. The method (adapted to
\text{select}_{11} below) uses three levels of auxiliary directories:

- Record the absolute positions of every \( (\log_2(h) \log_2 \log_2(h)) \)-th \textit{non-overlapping} 11 sequence,
  \( i.e. R[i] = \text{select}_{11}(i \log_2(h) \log_2 \log_2(h)) \). This needs \( \lceil \log_2(h) \rceil \) bits per entry, and
  \( O(h / \log \log(h)) \) bits in total (\( i.e. o(h) \) bits).
- Let \( r(i) = R[i] - R[i - 1] \) (subrange size).
- For \( r(i) \geq (\log_2(h) \log_2 \log_2(h))^2 \) explicitly store the positions of every 11 in the corresponding
  subrange. Each entry takes \( \lceil \log_2(h) \rceil \) bits, and \( r(i) \log_2(h) \) bits in total.
- If \( r(i) < (\log_2(h) \log_2 \log_2(h))^2 \) store in the table \( R' \) the positions, relative to the start of the
  range, of every \( (\log_2(r(i)) \log_2 \log_2(h)) \)-th occurrence of sequence 11 in the range. Each entry
  takes \( \log_2(r(i)) \) bits, and \( r(i) / \log_2 \log_2(h) \) bits in total.
- Let \( r'(i) = R'[i] - R'[i - 1] \).
- If \( r'(i) \geq \log_2(r'(i)) \log_2(r'(i)) \log_2 \log_2(h) \), store all answers explicitly.
- If \( r'(i) < \log_2(r'(i)) \log_2 \log_2(h) \), Clark showed that \( r'(i) < 16(\log_2 \log_2(h))^4 = o(\log_2(h)) \),
  so the range is small enough to answer the query by doing a constant number of table lookups for
  short bitstrings.

The \text{select}_{11}(j) query can then be performed as follows: first compute the position in \( R \) (\( i.e.
\( i = \lfloor j / (\log_2(h) \log_2 \log_2(h)) \rfloor \)) and the corresponding \( r(i) \) value. If \( r(i) \geq (\log_2(h) \log_2 \log_2(h))^2 \),
the result can be directly retrieved from the second level directory. Otherwise, we basically repeat the
above for the second level directory, and either retrieve the answer from the third level directory, or by
doing a constant number of table lookups for short bitstrings.

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.
The tables (and their sizes) are basically the same as in Clark’s construction, hence we can build a data
structure taking \( o(h) \) bits of space (in additional to the original sequence) and supporting \text{select}_{11}
queries in \( O(1) \) time.

We remark that this may not be the most practical solution, see \( e.g. \) [22, 30].

### 5. Comparison, extensions and applications

In this section we provide several extensions over the basic techniques, and present some applications.
However, we begin with a quick comparison of our techniques against each other and against Huffman
coding.
First consider SDC and Huffman coding. SDC needs at most $2(H_0(S) + 1)$ bits per symbol, or just $H_0(S) + 2\sqrt{H_0(S)} + 1$ bits using the technique of Sec. 3.3. Huffman coding needs only at most $H_0(S) + 1$ bits per symbol. However, the analysis of SDC is pessimistic. Without the auxiliary vector $D$, the number of bits per symbol is actually less than for Huffman coding. This is clear as we can use all the possible codes, while Huffman coding is limited to prefix codes. For example, if the symbol distribution is flat, i.e. every symbol has the same probability of occurrence, then $H_0(S) = \log_2(\sigma)$, but the bits per symbol for SDC is lower bounded by $\log_2(\sigma) - 2$ and upper bounded by $\lceil \log_2(\sigma) \rceil - 1 + O(\log_2(\sigma)/\sigma)$. For example, for flat distribution of 7-bit ASCII alphabet SDC obtains about 5.11 bits per symbol. Likewise, for DNA alphabet ($\sigma = 4$) we get 1.5 bits per symbol, and for protein alphabet ($\sigma = 20$) 2.9 bits per symbol. These, of course, need also the auxiliary vector, and the total space becomes worse than for Huffman.

Consider now SDC and Fibonacci coding. The latter is better than the former when (approximately)

$$
\frac{H_0(S) + \log_2(\sqrt{5}) + 1/3}{\log_2(\phi)} < H_0(S) + 2\sqrt{H_0(S)} + 1
$$

This gives

$$
\left(\frac{\log_2(\phi) \pm \sqrt{(\log_2(\sqrt{5}) + 4/3) \log_2(\phi) - \log_2(\sqrt{5}) - 1/3}}{1 - \log_2(\phi)}\right)^2
$$

for the limits. In practice this means that Fibonacci coding is better when $H_0(S)$ is less than about 15 bits.

### 5.1. Context based modelling and retrieving short substrings in $O(1)$ time

So far we have considered only retrieving a single symbol in $O(1)$ time. This is enough for the applications we consider shortly. However, for some applications one might want to retrieve substrings of $S$. We now show how our scheme can be used to retrieve substrings of length $s = O(\log_\sigma(n))$ in constant time. The bit-string in $S'$ corresponding to the substring in $S$ is again obtained by select queries, i.e. if we want the substring $S[i \ldots i + s - 1]$, we first get

$$
r = S'[\text{select}_1(D, i) \ldots \text{select}_1(D, i + s - 1)].
$$

The length $|r| = O(s \log_2(\sigma)) = O(\log_\sigma(n) \log_2(\sigma)) = O(\log_2(n))$ in the worst case, so $r$ can be retrieved in $O(1)$ time in the RAM model. To decode $r$ we need also

$$
t = D[\text{select}_1(D, i) \ldots \text{select}_1(D, i + s - 1)].
$$

Hence $r$ can be decoded by a table look-up $S[i \ldots i + s - 1] = T'[r][t]$. To keep the table small, we can use e.g. $\leq \frac{1}{4} \log_2(n)$ long bit-strings to decode the substring in 4 pieces. This keeps the additional space complexity term $O(\sqrt{n} \log(n))$. Fibonacci coded sequences can be handled similarly, except that we do not need $t$.

This can be improved by applying the method directly to substrings of length $s$, i.e. by considering the sequence $S'[0 \ldots n - 1]$ over the alphabet $\Sigma$ as a sequence $Q[0 \ldots n/s - 1]$ (w.l.o.g. we assume that $s$ divides $n$) over an alphabet $\Sigma^s$. That is, we define that $Q[i] = S[i \ldots i + s - 1]$. If we use
\[ s = \left\lfloor \frac{1}{2} \log_\sigma(n) \right\rfloor, \text{ the symbol table requires only } O(\sqrt{n \log(n)}) \text{ bits. Applying then the basic method} \]
\[ \text{over } Q \text{ to obtain its compressed representation } Q', \text{ the space becomes } 2n(H_0(Q) + 1)/s + o(n) = 2nH_0(Q)/s + o(n) \text{ bits, including the auxiliary vector. However, as shown in [16, 11] this can be also} \]
\[ \text{bounded as} \]
\[ 2\left( nH_k(S) + \sum_{i=0}^{n/s} s \right) + o(n) = 2n(H_k(S) + 1) + o(n), \tag{32} \]
\[ \text{for any } k = o(\log_\sigma(n)). \text{ Using this representation, any substring of length } s \text{ of } S \text{ can be retrieved in } O(1) \]
\[ \text{time, by retrieving at most two symbols of } Q. \text{ This is essentially the same method as in [11], except that they did not need the auxiliary vector, but the code word beginnings were encoded using another method, and consequently their space complexity was } nH_k(S) + O(n \log \log(n)/ \log_\sigma(n)). \]
\[ \text{The method described in Sec. 3.3 is orthogonal, and applies to this case as well, giving a space} \]
\[ \text{complexity of} \]
\[ nH_k(S) + nu + nH_k(S)/u + n + o(n), \tag{33} \]
\[ \text{which is again optimized for } u = \sqrt{H_k(S)}, \text{ giving a total space of} \]
\[ n\left( H_k(S) + 2\sqrt{H_k(S)} + 1 \right) + o(n). \tag{34} \]

5.2. Word alphabets

Our method can be used to obtain very good compression ratios for natural language texts by using the \( \sigma \) distinct words of the text as the alphabet. The 0-th order statistical model is known to work very well in this case. By Heaps’ Law [18], \( \sigma = n^\alpha \), 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 \( W[0 \ldots \sigma - 1] \) \( \sigma \) strings, sorted by their frequency. The compression algorithm then codes the \( j \)th most frequent word as an integer \( j \) using \( \lfloor \log_2(j + 2) \rfloor \) bits (with SDC). Again, the bit-vector \( D \) provides random access to any word of the original text. Obviously, Fibonacci coding works as well.

As already mentioned, using \( u = 7 \) (with SDC) 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.

5.3. 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 with SDC, and again using the auxiliary sequence \( D \) to mark the starting positions of the codewords. Fibonacci coding works similarly, but we do not need the auxiliary vector. This approach does not need any symbol tables, so the space requirement does not depend on \( \sigma \). Still, if \( \sigma \) 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
\[ \lfloor \log_2(x) \rfloor + 2\lfloor \log_2(1 + \lfloor \log_2(x) \rfloor) \rfloor + 1 \tag{35} \]
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 \( i \)th integer of the sequence.

5.4. Succinct full-text indexing

*Full-text index* is a data structure that allows searching all the occurrences of a (relatively short) pattern string from a text string, without scanning the whole text. Traditional full-text indexes (such as suffix tree [35] or suffix array [25]) need to keep the original text alongside the additional indexing structures. *Self-indexes* [9] allow the text to be discarded, as it can be reconstructed from the index itself. Finally, *succinct self-indexes* try to squeeze the space of the index close to the entropy bounds (while keeping as good query time as possible).

We do not go into the details of the succinct self-indexes. For a survey, refer to [28]. However, for most of the existing methods, the low-order term of the space complexity depends on the alphabet size \( \sigma \). In theory this is usually negligible, but in practice it can be reasonably large. One solution to cope with this was presented in [17]. They present two variants, the first one (based on Huffman coding) achieves space complexity of \( n(2H_0(S) + 3 + \varepsilon)(1 + o(1)) \) bits, for any constant \( 0 < \varepsilon < 1 \), for a text string \( S \) of length \( n \) symbols. The space complexity of the second variant (FM-KZ) was not analyzed. Their second variant, FM-KZ, uses a variant of Fibonacci coding. Their coding method is slightly different from ours. The difference boils down to using an additional 0-bit to terminate each codeword (that is, the terminator becomes string 110). However, the nature of the problem allows them not to store the trailing zero bit explicitly, so the codeword lengths are actually precisely the same as in our variant. The result is that the space complexity is as for FM-Huffman, if we just replace \( 2(2H_0(S) + 1) \) with \( (H_0(S) + \log_2(\sqrt{5} + 1/3))/\log_2(\phi) \).

Note that none of the space complexities have dependence on \( \sigma \) (other than what may implicitly follow from \( H_0(S) \)). These are not the best results (but still reasonably good) for this problem, but the proposed methods are relatively easy to implement, as compared to the theoretically best approaches.

5.5. Fast string matching

The task of *compressed pattern matching* [1, 24, 26] is to report all the occurrences of a given pattern in a compressed text, without decompression. We now present several efficient string matching algorithms working on the compressed texts. We assume SDC here, although 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 compare 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. Note that in this application we do not need the select data structures for the matching, as the input is scanned sequentially. On the other hand, if the matching locations in the original sequence must be reported, we can easily handle that by using \( \text{rank}_1(D, i) \), where \( i \) is the location in the compressed sequence.
The average time of Alg. 1 clearly depends on the parameter of size $P$ in Proof:

$b = O(\log(m))$

for $i \leftarrow 0$ to $(1 << b) - 1$ do $\text{shift}[i] \leftarrow m$

for $i \leftarrow 1$ to $b - 1$ do

for $j \leftarrow 0$ to $(1 << (b - i)) - 1$ do $\text{shift}[(c << (b - i)) | j] \leftarrow m - i$

for $i \leftarrow 0$ to $m - b - 1$ do $\text{shift}[P[i + b - 1]] = m - i - b$

$a \leftarrow P'[m - b \ldots m - 1]$

$\text{occ} \leftarrow 0$

for $i \leftarrow m - 1$ to $n - 1$ do

$c \leftarrow T'[i - b + 1 \ldots i]$

if $a = c$ and $D_T[i - m + 1 \ldots i] = D_P$ and $T'[i - m + 1 \ldots i] = P'$ then $\text{occ} \leftarrow \text{occ} + 1$

$i \leftarrow i + \text{shift}[c]$

return $\text{occ}$

5.5.1. BMH approach

The well-known Boyer-Moore-Horspool algorithm (BMH) [19] 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 $S[i - m + 1 + \text{shift}\ldots i + \text{shift}]$ (regardless of whether $S[i]$ was equal to $P'[m - 1]$ or not), where $\text{shift}$ is computed as

$$\text{shift} = m - \max\{j \mid P'[j] = S[i], 0 \leq j < m - 1\}$$

(36)

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 5.1.** 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 $b = O(\log(m))$. 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), or only $O(h)$ on average. We therefore want to choose $b$ so that the probability $p$ of
case (2) is low enough. To keep the total time at most $O(h/(m - b))$ on average, we select $b$ so that $phm = O(h/(m - b))$, where $p = 1/2^b$, assuming that the bit values have uniform distribution. Hence it is enough that $hm/2^b < h/(m - b)$, or more strictly that $hm/2^b \leq h/m$, i.e. $b \geq 2 \log_2(m)$. The preprocessing time is $O(2^b + m)$, which is $O(m^2)$ for $b = 2 \log_2(m)$.

We note that this breaks the lower bound of $O(h \log_2(m)/m)$, which is based on comparison model [38], 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 [34], could be generalized just as easily.

5.5.2. Shift-Or and BNDM

The two well-known bit-parallel string matching algorithms Shift-Or [2] and BNDM [29] can be directly applied to our case, and even simplified: as already noted in [29], 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 $D_P^r$ 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 $[m/w]$. However, we note that the “superalphabet” trick of the previous section works for these two algorithms as well (see also [12]). 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 $b$ first steps in $O(1)$ time, and then continue the algorithm normally. Using $b = 3 \log_2(m)$ gives $O(h/m)$ average search time.

6. Experimental results

We have run experiments to evaluate the performance of our algorithms. We have concentrated on the compression ratios using word-based modelling. Some experiments were also run to evaluate string matching performance over compressed sequences. The experimental results are not exhaustive, but are mainly intended to show the potential of the proposed methods. 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 [26]: 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 dictionaries.

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

\(^1\)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.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Size</th>
<th>σ (words+separators)</th>
<th>words</th>
<th>(H_0) (words)</th>
</tr>
</thead>
<tbody>
<tr>
<td>dickens</td>
<td>English text</td>
<td>10,192,446 B</td>
<td>34,381 + 1,071</td>
<td>1,819,394</td>
<td>9.92 bits</td>
</tr>
<tr>
<td>world192</td>
<td>English text</td>
<td>2,473,400 B</td>
<td>22,917 + 498</td>
<td>343,139</td>
<td>10.91 bits</td>
</tr>
<tr>
<td>samba</td>
<td>source code</td>
<td>6,760,204 B</td>
<td>29,822 + 15,544</td>
<td>924,640</td>
<td>10.40 bits</td>
</tr>
<tr>
<td>XML</td>
<td>XML source</td>
<td>5,303,867 B</td>
<td>19,582 + 1,495</td>
<td>847,806</td>
<td>9.10 bits</td>
</tr>
</tbody>
</table>

(b) Compression ratios

<table>
<thead>
<tr>
<th>File</th>
<th>gzip -9</th>
<th>bzip2 -9</th>
<th>SDC</th>
<th>SDC W</th>
<th>FibC</th>
<th>FibC W</th>
<th>Huffman</th>
<th>ETDC</th>
<th>(H_0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>dickens</td>
<td>37.7%</td>
<td>27.4%</td>
<td>35.3%</td>
<td>29.3%</td>
<td>31.5%</td>
<td>25.4%</td>
<td>28.3%</td>
<td>32.9%</td>
<td>26.2%</td>
</tr>
<tr>
<td>world192</td>
<td>29.1%</td>
<td>19.7%</td>
<td>35.5%</td>
<td>29.3%</td>
<td>31.6%</td>
<td>25.3%</td>
<td>29.0%</td>
<td>34.4%</td>
<td>24.4%</td>
</tr>
<tr>
<td>samba</td>
<td>20.1%</td>
<td>16.3%</td>
<td>36.1%</td>
<td>24.9%</td>
<td>32.2%</td>
<td>21.4%</td>
<td>30.3%</td>
<td>38.4%</td>
<td>27.4%</td>
</tr>
<tr>
<td>XML</td>
<td>12.3%</td>
<td>8.0%</td>
<td>33.0%</td>
<td>24.5%</td>
<td>30.6%</td>
<td>21.6%</td>
<td>28.7%</td>
<td>38.6%</td>
<td>26.4%</td>
</tr>
</tbody>
</table>

(c) Compression ratios with and w/o select data structure

<table>
<thead>
<tr>
<th>File</th>
<th>with (u = 1)</th>
<th>with (u = 2)</th>
<th>with (u = 3)</th>
<th>with (u = 4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>dickens</td>
<td>39.1%</td>
<td>35.4%</td>
<td>37.1%</td>
<td>33.5%</td>
</tr>
<tr>
<td>world192</td>
<td>38.6%</td>
<td>35.2%</td>
<td>37.0%</td>
<td>33.7%</td>
</tr>
<tr>
<td>samba</td>
<td>39.1%</td>
<td>35.4%</td>
<td>37.6%</td>
<td>33.8%</td>
</tr>
<tr>
<td>XML</td>
<td>36.6%</td>
<td>34.4%</td>
<td>34.7%</td>
<td>32.6%</td>
</tr>
</tbody>
</table>

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 separators), and for word stream only. We used the darray method [30]. For SDC coding this can be directly applied on \(D\) vector. For FibC this needs some modifications, but these are quite easy and straightforward. 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 \textit{dickens} file. We compared our algorithms against the BMH algorithm on the original uncompressed text. We used patterns consisting of 1...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 FBHM (our BMH variant running on compressed texts) achieve reasonably good performance, although they lose to plain BMH, which has very simple implementation. For FBHM 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.
the original sequence. The method gives good compression ratio for natural language texts, and allows
bits. The $x$-axis ($m$) is the pattern length in words.

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. The technique has many other applications in succinct data structures in general.

References


