SN Computer Science (2022) 3:181
https://doi.org/10.1007/s42979-022-01052-w

ORIGINAL RESEARCH

Fast Exact Pattern Matching by the Means of a Character Bit Representation

Igor O. Zavadskyi[1]

Received: 29 July 2021 / Accepted: 5 February 2022 / Published online: 7 March 2022

Abstract
Different approaches to speeding up the classical exact pattern matching Boyer–Moore–Horspool algorithm are discussed. The first improves such known technique of exact pattern matching as a 2-byte read by introducing the so called ‘1.5-byte read’, which allows us to address the problem of a large shift table not fitting into the L1 cache. Then we discuss different types of a fast loop, experimentally test them and enrich the most efficient one with the so called ‘double loop’ technique. Also, a specific technique of increasing the maximal shift length in ‘1.5-byte read’-algorithms and a known technique of sliding windows are taken into consideration. Combining these approaches, we get different exact pattern matching solutions, well suited for different types of input data. The experiments have been provided on a byte-aligned text, encoded with (s, c)-dense codes, English text, protein, and DNA sequences. Based on a solution for a byte alphabet, we build the algorithm Z-bit, intended to be used for searching a string in a bitstream. The Z-Bit algorithm demonstrates the most impressive experimental results, outperforming other known solutions more than twice on all investigated pattern lengths. Our other algorithms appear to be the fastest on some ranges of pattern lengths.

Introduction

Finding all occurrences of a given substring in a larger body of a text is one of the most fundamental problems in computer science. In this paper, we consider either this general problem or its important sub-problem consisting in searching a bitstream pattern in a bitstream text. It is of interest since a vast variety of data is presented in a binary form, e.g. images, videos, archived data, etc. The recent invention of data compression codes, which perform close to entropy and at the same time support data search in a compressed file [1], also actualizes the demand for bitstream pattern matching.

This article is the extended version of the paper presented at Prague Stringology Conference PSC’20 (Igor O. Zavadskyi: Fast Exact Pattern Matching in a Bitstream and 256-ary Strings. In: Proceedings of the Prague Stringology Conference, 2020: p.33–47).

This article is part of the topical collection “String Processing and Combinatorial Algorithms” guest edited by Simone Faro.

Igor O. Zavadskyi
ihorza@gmail.com

1 Taras Shevchenko National University of Kyiv, 2d Glushkova ave., Kyiv, Ukraine

The first non-trivial bitstream pattern matching algorithm was presented in 2007 by Klein and Ben-Nissan [17]. Since then, Faro and Lecroq developed a number of improved bitstream pattern matching techniques implemented in the Binary-hash and Binary-skip algorithms (both presented in [11]) and the most advanced Binary–Faro–Lecroq (BFL) algorithm [10]. Also, the adaptation of the FED algorithm (Fast matching with Encoded DNA sequences) [16] to binary search has to be mentioned, although it is somewhat inferior to BFL on short and middle-sized patterns.

The general idea of any non-trivial bitstream pattern matching method consists in avoiding time consuming bit-level operations as far as possible. A search is performed on the byte level, and only when a candidate substring is found, the bit-level procedure checks if a true pattern occurrence takes place. Therefore, any bitstream pattern matching technique is based on the underlying algorithm of pattern matching on a byte alphabet. For example, the algorithm [17] is based on Boyer–Moore search method, while the BFL algorithm combines a multi-pattern version of the BNDM algorithm [20] with the simplified shift strategy of Commentz–Walter algorithm [4].

Of course, an underlying byte-level algorithm has to be implemented in a multi-pattern version, since it searches not a single pattern but 8 patterns corresponding to 8 possible alignments of a binary substring towards the byte boundaries (assuming a byte is 8 bits). However, the performance of a multi-pattern version of an algorithm strongly depends on the performance of its single-pattern version, and thereby the development of pattern matching algorithms that efficiently perform on a byte alphabet is a key for solving the bitstream pattern-matching problem.

According to [12] and our own experiments, the following algorithms are of the most interest for large alphabets (consisting of ≥ 64 characters), in different testing environments and for different pattern lengths: comparison-based Fast Search (FS, [3]); exploiting SIMD instructions Streaming SIMD Extensions Filter (SSEF, [18]) and Exact Packed String Matching (EPSM, [7]); variations of algorithms exploiting the bit-parallelism idea: Simplified BNDM with q-grams (SBNDMq, [5]), Forward SBNDM (FSBNDM, [9]), BNDM for long patterns (LBNDM, [21]), and Q-gram Filtering (QF, [6]).

As shown in [23], the performance of BNDM-type algorithms can be improved by applying the technique of *2-byte read*, which is of special interest when the alphabet consists of 256 characters, each occupying all 8 bits of some byte in memory; we call it a *byte aplphabet*. However, the performance of 2-byte reads suffers from the expansion of shift tables. As experiments show, the running time of an algorithm increases significantly when its shift tables together with some other preprocessed data cease to fit into processor L1 cache, which is typically 16–64 kB. For a byte alphabet, the size of a shift table with 2-byte indices is 64 kB, which exceeds the "L1 cache limit" in most cases.

Two other techniques, which can significantly improve the algorithm performance for different alphabets and pattern lengths, are based on using multiple sliding search windows [14] and skipping the occurrence check with a help of so called *fast loop* [15].

For a byte alphabet, featuring a simple comparison-based method with these techniques can outperform the algorithms based on bit-parallelism. Indeed, the experimental results in Table 2 show that the comparison-based Fast Search algorithm with 8 sliding windows performs faster than all other known methods (except for those hereinafter developed) for $8 \leq m \leq 128$.

In this paper, we improve all three aforementioned techniques. At first, we present a "compromise" solution between 2-byte and 1-byte reads, forming the index of a search table from the values of more than 1 but less than 2 sequential bytes of a text, typically 13–15 bits. We call this method a *1.5-byte read*. It allows us to increase the average length of a shift compared to "1-byte read" algorithms while spending rather less memory than the 2-byte read approach requires. Then we offer a few tricks, which improve the performance of a fast loop and sliding windows techniques. As a result, we construct a comparison-based algorithm, which outperforms all other known solutions in discovering patterns of lengths > 4 in a text on a byte alphabet. This algorithm is called *Z-Byte* and discussed in "Pattern matching over a byte alphabet". A bitstream search algorithm, based upon Z-Byte, is constructed in "Pattern matching in a bitstream", we call it *Z-Bit*.

Z-Byte algorithms either perform well on middle-size alphabets (e.g. for protein sequences). However, when the alphabet becomes smaller, its characters contain too many insignificant (or identical) bits and 1.5-read loses the efficiency. In this case, the bit-level shift operation together with or/xor/+ can be applied to 'attach' the significant bits of adjacent text characters to each other and thus to form the index of a shift table. The algorithm based on this principle we call *Bricks* since the process of constructing an index of a shift table from blocks of bits resembles building a wall from bricks. This algorithm is discussed in "Pattern matching on small alphabets". Finally, the results of algorithm benchmarking are presented in "Experimental results".

Let us note that an attempt to combine multiple-character reads and multiple search windows has been done in our previous work [26]. However, the methods presented hereinafter are simpler and faster.

Throughout the entire presentation we use the following notations:

- Σ—alphabet of an input text and a pattern
- $|\Sigma|$—size of the alphabet
- $\sigma = \lceil \log_2 |\Sigma| \rceil$—the number of bits required to store a character of Σ
- b—number of bits in a byte, by default $b = 8$
- k—number of significant bits in a 1.5-byte read, typically $b \leq k \leq 2b$

Pattern Matching Over a Byte Alphabet

In this section as well as in "Pattern matching on small alphabets", we assume that each character of a text occupies one byte of memory, and all bits of this byte are significant, i.e. $|\Sigma| = 256$. By $T[0 \ldots n-1]$ and $P[0 \ldots m-1]$ we denote a text and a pattern respectively.

1.5-Byte Read

At first, let us discuss the essence of the "1-byte read", "2-byte read" approaches and their "1.5-byte read" modification. Let Z be a one-dimensional shift table for some pattern matching algorithm and i is the index of some character of a text. The 1-byte read approach assumes the value $Z[T[i]]$ to be the shift length, as in the Boyer–Moore–Horspool algorithm (BMH, [13]), or other data the shift depends on. BMH method is shown schematically in Alg. 1.

Algorithm 1: Boyer-Moore-Horspool algorithm

```
1 foreach $c \in \Sigma$ do $Z[c] \leftarrow m$;                  // Preprocessing
2 for $i \leftarrow 0$ to $m-2$ do $Z[P[i]] \leftarrow m-1-i$;
3 $pos \leftarrow 0$;                                              // Search
4 while $pos \leq n-m$ do
5     check the occurrence at $pos$;
6     $pos \leftarrow pos + Z[T[pos+m-1]]$;
```

Line 6 corresponds to a "bad character" shift with the maximal possible value m. When $|\Sigma| = 256$ and the pattern is short, the probability of a maximal BMH shift is high enough. E.g. it equals $(255/256)^m$ for random text and random pattern. But when the pattern is longer, the "1-byte" bad character shift becomes not sufficient. For example, if $m = 512$, $(255/256)^m \approx 0.135$.

The situation can be amended by using 2 sequential bytes of a text as a basis of a bad-character shift. E.g. for $m = 512$, the probability of a maximal shift of a random pattern over a random text becomes greater than 0.99, although the length of a maximal shift length is decreased by 1 (the latter fact is important for short patterns only). The computationally efficient implementation of this approach is discussed in [23]: the expression $Z[T[i]]$ is transformed into $Z[word(T[i], T[i+1])]$, where the function *word* converts two sequential bytes of memory into a two-byte word in a processor register. In C programming language this function can be implemented by the type-casting mechanism, i.e. $word(T[i], T[i+1])$ equals to `*(unsigned short*)(T+i)`. In fact, the time complexity of calculating the $Z[T[i]]$ and $Z[word(T[i], T[i+1])]$ values is the same, while the memory complexity is significantly increased from 256 bytes needed for 1-byte reads to 64 kB occupying by a shift table with 2-byte indices.

However, for reasonable pattern lengths, e.g. less than 1000 bytes, the memory complexity can be significantly reduced with a little impact on the shift length. This can be achieved by using not all bits of a 2-byte word in the index of a shift table. Some of bits can be suppressed by applying the mask: $Z[word(T[i], T[i+1])\&mask]$. For example, if $m = 512$ and the mask contains 14 'one' bits, the probability of a maximal shift for a random text and a random pattern will be $((2^{14}-1)/2^{14})^{511} \approx 0.97$, which is only 0.022 less than that one for 2-byte read. At the same time, the shift table will contain 2^{14} vs. 2^{16} elements, i.e. 4 times less. Of course, one extra operation *&mask* has to be performed in the search loop, but in most cases, this expense will be more than covered by the fact that the shift table fits into L1 cache.

Table 1 Running times of BMH algorithm variations on a random text, $|\Sigma| = 256$ (hundredths of seconds)

Row	m	2	4	8	16	32	64	128	256	512	1024	2048	4096
1	BMH	155.67	79.72	41.58	23.42	14.08	10.81	9.97	15.29	17.65	19.61	23.78	25.69
2	BMH-12	333.55	111.29	48.56	24.52	13.8	10.24	13.85	18.84	12.27	8.17	5.98	3.94
3	BMH-13	331.45	110.99	48.08	24.16	13.51	10.21	14	18.41	11.97	7.76	5.51	3.23
4	BMH-14	330.29	111.62	47.86	24.4	14	10.49	14.11	18.24	11.54	7.59	5.32	2.99
5	BMH-15	342.13	120.81	53.33	27.06	15.53	10.93	14.19	18.13	11.2	7.6	5.29	2.95
6	BMH-16	401.19	137.57	59.87	29.21	16.59	11.57	14.59	18.25	11.25	7.79	5.38	2.96
7	Value skip	151.72	77.56	39.19	20.45	11.94	9.9	10.17	18.76	20.22	21.55	22.77	23.12
8	Value skip + 1.5-read	357.29^{13}	119.21^{13}	52.0^{12}	24.69^{13}	12.58^{13}	9.59^{13}	12.85^{12}	16.06^{13}	9.73^{14}	7.36^{14}	5.34^{15}	3.06^{15}
9	Flag skip	22.94	14.92	12.27	9.92	8.69	8.28	8.03	9.98	13.96	17.27	23.02	23.09
10	Flag skip + 1.5-read	42.32^{16}	15.07^{16}	8.36^{16}	6.63^{16}	5.35^{16}	5.03^{15}	4.22^{16}	2.86^{15}	1.69^{16}	1.1^{16}	0.76^{16}	0.61^{16}
11	Double flag skip + 1.5-read	42.27^{16}	14.83^{16}	8.33^{16}	6.45^{16}	5.37^{16}	4.92^{13}	4.24^{16}	2.74^{13}	1.61^{15}	0.96^{16}	0.58^{15}	0.43^{16}
12	Double flag skip reverse + 1.5-read	27.91^{16}	14.46^{16}	10.43^{16}	8.02^{16}	6.14^{12}	5.09^{12}	4.31^{12}	2.81^{12}	1.61^{15}	0.97^{15}	0.6^{15}	0.43^{16}

Experimental results of featuring the BMH algorithm with 2-byte and 1.5-reads are presented in the upper part of Table 1. A random pattern was searched 1000 times in a random 10MB text on a byte alphabet, the computer parameters are given in "Experimental results". The average running times of the original BMH algorithm are given in row 1, while the BMH algorithms featured with k-bit reads are denoted as BMH-k and represented in rows 2–6. BMH-16 is the algorithm with a 2-byte read, where the operation and *mask* is not applied. For all algorithms we use the `unsigned char` shift array (with 1-byte elements), when $m \leq 256$, and `unsigned short` (2-byte elements) for longer patterns.

As seen, the original version of BMH is faster on short patterns, $m \leq 16$, when 2-byte and 1.5-byte reads are superfluous, and decreasing of a maximal shift length by 1 is important. As the pattern is longer, the 2-byte and 1.5-byte reads become more efficient. In particular, for $m = 4096$, the original BMH version is more than 8 times slower than versions based on 14-bit, 15-bit and 16-bit reads. If to compare 1.5-byte read and 2-byte read versions, the former is faster for all pattern lengths, and this superiority is more significant for short patterns when either 1.5 read or 2-byte read almost always provides the maximum length shift. E.g., when $m = 2$, the BMH-14-bit performs 20% faster than the 2-byte read version despite extra&*mask* operation, which shows the benefit of fitting the shift table into the L1 cache. However, when the pattern is longer, the '2-byte shifts' become longer than '1.5-byte shifts', and this neutralizes the gain. The same reason explains a balance between performances of different 1.5-byte read versions. Thus, on short patterns, 12-, 13-, and 14-bit versions perform at the same rate, since their shift tables perfectly fit into 32 kB L1 cache, while the table of a 15-bit version mostly fits (the cache may also contain some other data), and this is why the 15-bit method performs a bit slower. At the same time, when a pattern is longer, each extra significant bit in the *mask* increases the average shift length essentially and makes the search faster, which we can observe for $m \geq 2048$.

Fast Loop

The other disadvantage of Algorithm 1 consists in the necessity to check the possible occurrence of a pattern at each iteration of the search loop. However, the occurrence check can be avoided by applying the so called "fast loop", which was first introduced in [15]. This technique is implemented in a number of algorithms and is particularly effective when more than one character of a text is processed at each iteration, e.g. in the FS [3], SBNDM [22] and other algorithms.

Let us distinguish two types of a fast loop:

- A shift table element contains the length of a shift. Then the search position is shifted by this length until it is zero. This type of a fast loop we call *skip by value*. It is implemented mostly in algorithms based on character comparison, e.g. TBM [15], FS [3], MAW [26].
- A shift table element contains a flag indicating whether a shift of some constant length, e.g. m or $m - 1$, is *safe*, i.e. cannot cause missing the pattern occurrence. If it is, the constant length shift is performed and the fast skip loop continues, otherwise the loop breaks, the pattern occurrence has to be checked, and the value of the next shift is calculated by some other algorithm. This type of a fast loop we call *skip by flag*. It is implemented mostly in algorithms based on bit-parallelism or oracle automatons, e.g. EBOM [9] or SBNDM [22].

The advantage of a skip by value is quite obvious: we avoid the occurrence check even if the shift is non-maximal. On the other hand, a flag skip allows us to avoid loading the shift length value from memory to a computer register, which is quite a consuming operation. Actually, the main factor to choose a *value* or a *flag* fast loop is the probability of a maximal (or almost maximal) shift. When it is higher, the flag skip becomes preferable.

Featuring the BMH method with the flag fast loop and 1.5-byte reads, we get the Algorithm 2. The shift table Z contains the "flag" information. That is, if $Z[T[i]] = 1$, the shift of the search window $m - 1$ characters right is safe. The fast loop is implemented in lines 7 and 8. Although two bytes of a text are read in line 7, only $k < 2b$ bits of each two-byte word are used to form the index of the shift table Z. After exiting the fast loop, we check the occurrence and get the shift value from the Quick Search shift table (QS, [24]). To exit the fast and main loops at the end of a text correctly, it should be appended by a stop pattern.

Algorithm 2: Search algorithm with the fast loop and k-bit reads

```
$mask \leftarrow 2^k - 1$;                                   // Preprocessing
foreach $i \in [0; 2^k)$ do $Z[i] \leftarrow 1$;
for $i \leftarrow 0$ to $m - 2$ do
    $Z[word(P[i], P[i+1])\&mask] \leftarrow 0$
$pos \leftarrow m - 2$;                                      // Search
while $pos < n$ do
    while $Z[word(T[pos], T[pos+1])\&mask] \neq 0$ do
        $pos \leftarrow pos + m - 1$;
    check the occurrence at $pos - m + 2$;
    $pos \leftarrow pos + QS[pos + 2]$;
```

A value and a flag skip loop performances are presented in rows 7–10 of Table 1. The performance of the original BMH featured with a fast loop is shown in rows 7 and 9. As seen, when $m \leq 1024$, the flag skip loop outperforms the value skip more or less strongly, while for longer patterns they operate roughly at the same level. Rows 8 and 10 of Table 1 represent the performance of BMH with 1.5- or 2-byte read and a fast loop as well (the number of significant bits in the mask is put in brackets). In this case, the flag skip approach outperforms the value skip strongly for all tested pattern lengths since the probability of a maximal shift provided by a 1.5- or 2-byte read can be considered high enough even for long patterns. Therefore, in combination with 1.5- or 2-byte read, the 'flag' fast loop is definitely more productive than the 'value' fast loop when it comes to pattern search on a byte alphabet.

Double Fast Loop

Taking into account that the 1.5-byte or 2-byte read implies a high probability of a maximal shift, we develop a simple and efficient method to process the exit from a fast loop. The main idea is based upon the assumption that the inequality $Z[word(T[i-1], T[i])\&mask] \neq 0$ holds with roughly the same probability as the inequality $Z[word(T[i], T[i+1])\&mask] \neq 0$. It implies that even if the fast loop condition fails, we can take one step back and, very likely, continue the fast loop from that position. This *double fast loop* is shown in Alg. 3, which preprocessing phase is the same as in Alg. 2. If the condition in line 5 fails, we continue the main fast loop in line 9 with 1 character shorter shift and then in lines 3–4. Otherwise, the occurrence check is performed.

In general, the double fast loop resembles the factor-based approach limited to 1 step in the case when the main fast loop fails. And this limitation to 1 step makes sense since the logic of a factor-based approach 'process longer suffixes of a search window until the fast loop can continue or the pattern length is reached' requires rather more operations to implement than simple stepping 1 character back, and therefore it seems to be superfluous in many cases; in particular, if the alphabet is large.

The double fast loop is especially efficient for long patterns, when the probability of the fast loop termination is higher, while the relative impact of the left shift (lines 5 and 9) is less. This can be confirmed by comparison of rows 10 and 11 in Table 1. While for short patterns the performance of a fast loop and a double fast loop is roughly the same, for $m = 4096$ the performance gain from a double fast loop reaches 30%.

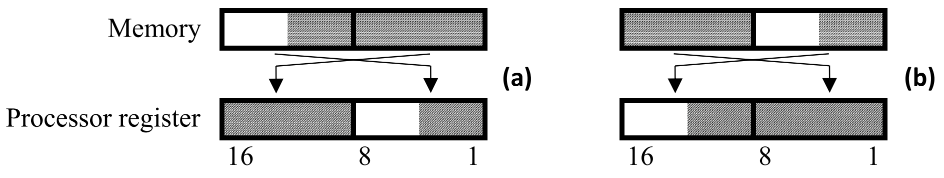

Fig. 1 Loading a 2-byte value from memory on a little endian machine. 'White' bits are reset to zero by *mask*, 'grey' bits remain significant. *mask* resets the bits of the highest byte (**a**) or lowest byte (**b**)

Algorithm 3: Search phase of the algorithm with the double fast loop

```
1 $pos \leftarrow m - 2$;
2 while $pos < n$ do
3     while $Z[word(T[pos], T[pos+1])\&mask] \neq 0$ do
4         $pos \leftarrow pos + m - 1$;
5     if $Z[word(T[pos-1], T[pos])\&mask] = 0$ then
6         check the occurrence at $pos - (m - 2)$;
7         $pos \leftarrow pos + QS[pos + 2]$;
8     else
9         $pos \leftarrow pos + m - 2$;
```

We denote the Alg. 3 by Zk-Byte, where k is the number of ‘one’ bits in the mask ($8 \leq k \leq 16$). If $k = 16$, lines 3 and 5 contain full 2-byte reads and variable *mask* is not needed, while in algorithm Z8 the reference to the shift table looks as $Z[T[\text{pos}]]$ and the variable pos can be incremented by m in line 4 and by $m - 1$ in line 9. In the next subsection, we discuss how to shift the search window m characters right for any value of k.

Longer Shifts

Let $T[\text{pos}]$ and $T[\text{pos} + 1]$ be the two last characters of a search window. If the condition $Z[word(T[\text{pos}], T[\text{pos} + 1])$ $\&mask] \neq 0$ in lines 3 and 5 of Alg. 3 is satisfied, these two bytes of a text cannot belong to the pattern together. Still, the character $T[\text{pos} + 1]$ can coincide with $P[0]$ and that’s why the search window can be shifted safely by $m - 1$ characters at most, not by m. If the pattern is short, this decrement of a maximal safe shift length can have a meaningful effect on algorithm performance.

This situation can be amended by adjusting the array Z: we can assign 0 to all elements of the form $Z[word(c, P[0])\&mask]$, $c \in \Sigma$. As a result, if the last byte of a search window coincides with $P[0]$, the shift will be considered as non-maximal, and m can be assumed to be the value of the maximal shift. It seems that this will increase the probability of a non-maximal shift in a random text by 1/256 at most, which is not too much. However, the *endianness* of a machine, i.e., an order in which the bytes of a value are loaded from memory into a processor register, has to be taken into account. Let us examine the function *word*, used in lines 3 and 5 of Alg. 3. It converts two sequential bytes of memory into a two-byte number in a processor register. In most programming languages this function can be implemented by the type-casting mechanism, e.g. `(unsigned short*)` operator in C language. However, its result depends on the endianness. On a little endian machine (e.g. x86 processor) bytes of a value are loaded into a register in the reverse order (Fig. 1). This is an unwanted situation if we compose a shift table index of the full last byte of a search window and a part of the second to last because a resultant value will be shifted in a register to the left (Fig. 1a), which makes the size of the shift table the same as for the 2-byte read. However, the situation in Fig. 1b—not full last byte and full second to last—is also unwanted, because only the part of the last byte of a search window should coincide with the part of $P[0]$ byte to make the shift non-maximal. Then, the increase of the non-maximal shift probability for a random text will be up to $1/2^{k-8}$, which significantly reduces the probability of a fast loop continuation. For example, if $k = 12$, the latter probability will be reduced by up to 1/16.

Nonetheless, on a little endian machine the permutation, shown in Fig. 1b, becomes admissible if the text is searched from right to left. Then, extending the maximal shift from $m - 1$ to m decreases the probability of a fast loop termination by up to 1/256, and the index of a shift table is composed of the low bits of a two-byte word. We call right-to-left search algorithms *reverse* and denote them by the letter “R”, e.g. RZ12-Byte. The reverse search method RZk-Byte is shown in Alg. 4.

The text T is assumed to be prepended with a stop pattern ($T[-m \ldots -1] = P[0 \ldots m-1]$) to exit the fast loop, given in lines 11−12, when the search finishes. The search window starts at the position $n - m$ and moves to the left. The variable pos always addresses the beginning of a search window in which the first two bytes are used to determine the possibility of a maximal shift by m characters. The algorithm steps 1 character forward after the fast loop and repeats the fast loop check (line 13). If the fast loop can continue from this new position, it continues in lines 17 and then 11. Otherwise, after checking the occurrence (line 14), we make the “Reverse Quick Search” shift using the shift table RQS (line 15), which is filled in lines 7–8. Lines 5—6 are intended to block the maximal shift when the first character of a search window coincides with the last character of the pattern; this code is equivalent to $Z[word(c, P[m-1])$ $\&mask] \leftarrow 0, c \in \Sigma$. Finally, lines 1–4 of the preprocessing stage are similar to those in Alg. 3.

Algorithm 4: The reverse search algorithm RZk-Byte

```
1  $mask \leftarrow 2^k - 1$;                                        // Preprocessing
2  foreach $i \in [0; 2^k)$ do $Z[i] \leftarrow 1$;
3  for $i \leftarrow 0$ to $m - 2$ do
4      $Z[word(P[i], P[i+1])\&mask] \leftarrow 0$
5  for $i \leftarrow 0$ to $2^{k-b}$ do
6      $Z[(i << b)|P[m-1]] \leftarrow 0$
7  foreach $c \in \Sigma$ do $RQS[c] \leftarrow m + 1$;
8  for $i \leftarrow m - 1$ downto $0$ do $RQS[P[i]] \leftarrow i + 1$;
9  $pos \leftarrow n - m$;                                             // Search
10 while $pos \geq m$ do
11     while $Z[word(T[pos], T[pos+1])\ \&\ mask] \neq 0$ do
12         $pos \leftarrow pos - m$
13     if $Z[word(T[pos+1], T[pos+2])\ \&\ mask] \neq 0$ then
14         check the occurrence at $pos$;
15         $pos \leftarrow pos - RQS[T[pos-1]]$;
16     else
17         $pos \leftarrow pos - (m-1)$;
```

The evaluation of Alg. 4 performance is given in row 12 of Table 1. As expected, it appears faster than Alg. 3 on short patterns ($m \leq 4$) due to the effect of 1 byte longer shifts. On long patterns, the algorithms perform almost at the same level, while on middle patterns the Alg. 4 is somewhat inferior to Alg. 3.

Sliding Windows

The performance of Z-Byte and RZ-Byte algorithms can be improved significantly by implementing a two sliding windows technique [14]. However, for reverse methods, it should be slightly changed, since we can search a text from right to left only, while in the standard method windows are moving towards each other until they meet. The search phase of Alg. 4 can be rewritten as shown in Alg. 5. Both sliding windows are moving towards the beginning of a text. The first window specified by pos1 starts in the middle of a text, while the second one specified by pos2 starts in the end. The main loop is finished when the first window reaches the beginning of a text. After that, the second window may be moved further if it has not reached the middle position yet (lines 17–22). The preprocessing phase of Alg. 5 is just the same as of Alg. 4.

Let us note that filling the shift table Z with zeros and ones allows us to join the 1.5-character checks with the bitwise '&' operation (line 4) instead of the slower logical *AND* as in "classical" sliding windows approach [14]. Also, the external fast loop is replaced with separate 'if-else' blocks for each of two sliding windows (lines 7–11 and 12–16). This is done to take the advantage of the situation when the maximal shift can be made only in one of two sliding windows.

Algorithm 5: The search phase of the reverse search algorithm with 2 sliding windows RZk-Byte-w2

```
$pos1 \leftarrow \lfloor n/2 \rfloor$;
$pos2 \leftarrow n - m$;
while $pos1 \geq 0$ do
    while $Z[word(T[pos1], T[pos1+1]) \,\&\, mask] \neq 0 \,\&$
      $Z[word(T[pos2], T[pos2+1]) \,\&\, mask] \neq 0$ do
        $pos1 \leftarrow pos1 - m$;
        $pos2 \leftarrow pos2 - m$;
    if $Z[word(T[pos1+1], T[pos1+2]) \,\&\, mask] = 0$ then
        check the occurrence at $pos1$;
        $pos1 \leftarrow pos1 - RQS[T[pos1-1]]$;
    else
        $pos1 \leftarrow pos1 - m + 1$;
    if $Z[word(T[pos2+1], T[pos2+2]) \,\&\, mask] = 0$ then
        check the occurrence at $pos2$;
        $pos2 \leftarrow pos2 - RQS[T[pos2-1]]$;
    else
        $pos2 \leftarrow pos2 - m + 1$;
while $pos2 > \lfloor n/2 \rfloor$ do
    while $Z[word(T[pos2], T[pos2+1]) \,\&\, mask] \neq 0$ do
        $pos2 \leftarrow pos2 - m$;
    if $pos2 > \lfloor n/2 \rfloor$ then
        check the occurrence at $pos2$;
    $pos2 \leftarrow pos2 - RQS[T[pos2-1]]$;
```

We denote the Alg. 5 as RZk-Byte-w2—the reverse search algorithm with a k-bit read and 2 sliding windows. As opposed to the "classical" approach, we can use an odd number of sliding windows. For example, the parallel searches in the algorithm RZk-Byte-w3 will start at positions $\lfloor n/3 \rfloor$, $\lfloor 2n/3 \rfloor$, and n. Of course, each pair of sliding windows in non-reverse Z-algorithms can be moved towards each other. However, for large enough texts the experiments show no significant difference in performance between "bi-directional" and "uni-directional" sliding windows methods.

Pattern Matching in a Bitstream

In this section we denote the array of full bytes of a pattern by $P[0 \ldots m-1]$, and $T[0 \ldots n-1]$ denotes the input text. The last bytes $P[m]$ and $T[n-1]$ are not full and padded with zeros if the pattern and/or text bit length is not a factor of 8. Otherwise, $P[m]$ is assumed to be 0, while $T[n-1]$ is a full byte. The byte next to the text, $T[n]$, is always 0 as well as $P[m+1]$. We denote the bit length of a pattern by l, and $p[0 \ldots l-1]$ is the array of pattern bits. By "search

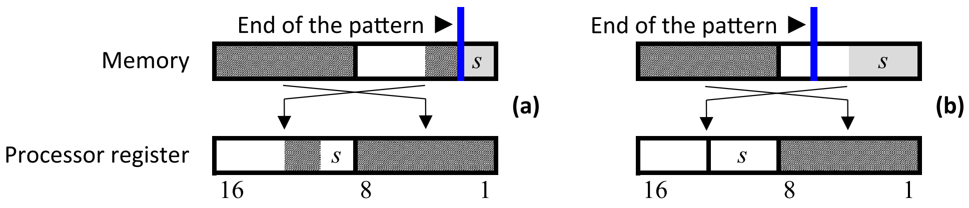

Fig. 2 The *bitWord* permutation (Alg. 9) of the right tail of a pattern. The bits, remaining significant after the permutation, are highlighted with dark grey. The bits that would have remained significant if not for the end of a pattern, highlighted with light grey. **a** The pattern ends within significant bits of the lowest byte; **b** the pattern ends within insignificant bits

window” we mean a $(m-1)$-byte substring of a text that is supposed to belong to the pattern.

Hereinafter we assume a little endianness and discuss the “right-to-left” bitstream search by the example of the RZk-Bit algorithm (on a bitstream the ‘reverse’ method runs essentially faster than the ‘direct’ method for short patterns and at the same level for longer ones). Its general structure is similar to the structure of the underlying RZk-Byte algorithm. At each iteration of the fast loop, we try to move the search window as far as possible to the left. pos addresses the leftmost byte of a search window. Thus, the window occupies the bytes $T[\mathrm{pos}], \dots, T[\mathrm{pos}+m-2]$, while bytes $T[\mathrm{pos}-1]$, $T[\mathrm{pos}+m-1]$, and possibly $T[\mathrm{pos}+m]$, may contain the left and right “tails” of the pattern.

The search window can be safely moved $m-1$ bytes to the left if two conditions are met (taking into account the endianness and applying the *mask*): (a) the pair of bytes $(T[\mathrm{pos}], T[\mathrm{pos}+1])$ does not belong to the pattern, and (b) some prefix of the pair $(T[\mathrm{pos}], T[\mathrm{pos}+1])$ of length greater than 8 does not coincide with the pattern suffix. The shift table Z is filled following these conditions at the preprocessing phase (Alg. 8) and checked during the search phase (lines 3 and 5 of Alg. 6). The whole block of code in lines 3–10 of Alg. 6 implements the double fast loop discussed in “Fast loop”.

After the exit from the double fast loop, we check if the pattern can be aligned with the search window shifted q bits to the left, $q = 0, \dots, b-1$. This check is performed by the procedure *CheckMatch*(q, pos) invoked in line 7 of Alg. 6. It is not efficient to test all b possible values of q. Instead, we store in the set $\lambda[c]$ all values $q < b$ such that the factor $p[q] \dots p[q+b-1]$ of a pattern coincides with the byte c. This implies that all possible occurrences such that the byte $T[\mathrm{pos}]$ is the leftmost full byte of a text that belongs to the pattern are checked in lines 6 and 7 of Alg. 6. After the occurrence check, pos is safely moved 1 byte to the left in line 8, and the fast loop starts again at the next iteration of the main loop.

Algorithm 6: The search phase of the RZk-Bit algorithm

```
1  $pos \leftarrow n-1$;
2  while $pos \geq m-1$ do
3      while $Z[word(T[pos], T[pos+1])\ \&\ mask] \neq 0$ do
4          $pos \leftarrow pos-(m-1)$
5      if $Z[word(T[pos+1], T[pos+2])\ \&\ mask] = 0$ then
6          foreach $q \in \lambda[T[pos]]$ do
7              $CheckMatch(q, pos)$;
8          $pos \leftarrow pos-1$;
9      else
10         $pos \leftarrow pos-(m-2)$;
```

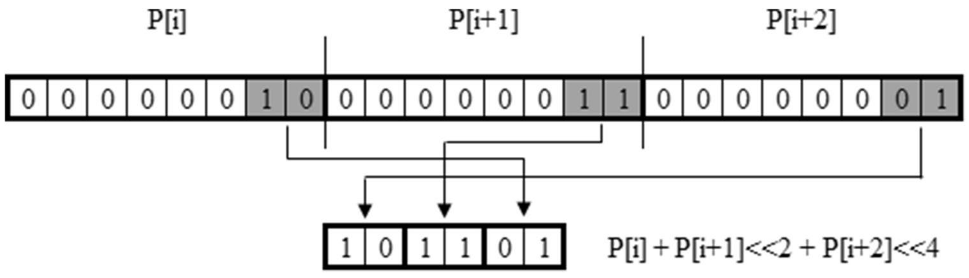

Fig. 3 Conversion a q-gram of pattern characters into a $q\sigma$-bit value

The body of the procedure *CheckMatch*(q, pos) is given in Alg. 7. We have to check if a window, occupying q lowest bits of the byte $T[\mathrm{pos}-1]$ and the next $l-q$ bits of a text, coincides with the pattern. The function *byte* in line 2 truncates a 2-byte value to its lowest byte. It can be implemented as a type-casting `(unsigned char)` in C language. Generally, in line 2 we compose a series of bytes from pairs of the adjacent bytes of a text taking q lowest bits of the byte $T[j]$ and $b-q$ highest bits of the byte $T[j+1]$. The composed bytes are compared with the bytes of the pattern. If they all match, the last byte is composed and compared with $P[m]$ in line 5. Since the byte $P[m]$ is not full, the composed byte is truncated by the mask *lastMask* to significant bits of $P[m]$ only.

checked if this substring coincides with some substring of a text. Each of these substrings does not consist of a continuous sequence of significant bits but has a "hole" of insignificant ones shown in the upper part of Fig. 1b. However, the function *bitWord* invoked in line 5 of Alg. 8 performs the bits permutation shown if Fig. 1b and returns the mentioned substring in the compact form shown in the lower part of that figure. If such substring is aligned to the boundaries of a two-byte word, this permutation becomes trivial and can be completed by the type-casting. This is how the function *word* is calculated in the search phases of RZ-Byte and RZ-Bit methods. However, the mentioned substring may intersect with 3 adjacent bytes of a pattern in a bitstream,

Algorithm 7: Procedure $CheckMatch(q, pos)$

```
1 $start \leftarrow j \leftarrow pos - 1$;
2 while $j - start < m$ AND
    $byte((T[j] << (b-q)) | (T[j+1] >> q)) = P[j - start]$ do
3     $j \leftarrow j + 1$
4 if $j - start = m$ then
5     if $((T[j] << (b-q)) | (T[j+1] >> q)) \& lastMask = P[m]$ then
6         output$(start \cdot k + q)$
```

The value *lastMask* as well as other values and tables, which remain constant through the search phase, is calculated at the preprocessing phase, shown in Alg. 8. Let us explain how the loop in lines 4–11 of Alg. 8 works, in which the shift table Z is constructed. For each 16-bit substring of a pattern, starting at the bit position i, the corresponding element of the array Z is specified with the flag 0, which denotes a non-maximal shift (the short suffixes of the pattern, $l-i \leq b$, are not processed, since their possible alignments with search window prefixes have no impact on a safe shift of the length $m-1$). During the search phase, it will be and the permutation becomes trickier. It is explained in the comments to Alg. 9.

The formatted substring is assigned to the variable t in line 5 of Alg. 8. If $l-i \geq 2b$, the whole substring belongs to the pattern, and $Z[t]=0$ (line 7). Otherwise, the substring is truncated at the end of the pattern, and after the *bitWord* permutation a "hole" of insignificant bits may appear inside the formatted substring, Fig. 2a. The length s of this "hole" is calculated in lines 9 (Fig. 2a) or 10 (Fig. 2b). In line 11 the "hole" in the substring t is filled with all possible 2^s values. This way, the set of indices of zero elements of the array Z is constructed.

Table 2 Running times on SCDC-encoded text (short patterns on top, long patterns on bottom)

m	2	4	6	8	12	16	24
WFR*q*	42.28	30.1	23.81^{2}	18.18^{2}	13.51^{2}	11.44^{2}	9.28^{4}
QF(*Q*, *S*)	$103^{2,3}$	$49.47^{2,3}$	$25.71^{3,4}$	$17.86^{3,4}$	$12.35^{3,4}$	$9.1^{3,4}$	$7.5^{3,4}$
FS-w*q*	38.31^{8}	20.36^{8}	13.73^{8}	10.8^{8}	7.69^{8}	6.4^{8}	5.37^{8}
BSDM*q*	59.48^{2}	21.22^{2}	13.73^{2}	10.93^{2}	8.92^{2}	8.22^{2}	7.25^{2}
skip-*q*	86.57^{2}	30.65^{2}	19.58^{2}	14.92^{2}	10.56^{2}	9.11^{2}	7.74^{2}
FSBNDM*qf*	89.57^{31}	33.17^{31}	21.08^{31}	16.49^{31}	10.93^{31}	9.02^{31}	7.33^{31}
SBNDM*q*	76.9^{2}	28.38^{2}	18.4^{2}	14.51^{2}	10.42^{2}	9.12^{2}	7.61^{2}
LBNDM	61.03	38.35	31.86	27.52	23.69	21.63	18.5
EPSM	**10.63**	**12.02**	11.77	11.79	10.89	9.11	6.68
Z-Byte(*k*)-w*q*	30.2^{8-2}	13.04^{14-3}	$\mathbf{8.37}^{14-3}$	$\mathbf{6.56}^{14-3}$	$\mathbf{5.03}^{14-3}$	$\mathbf{4.71}^{14-3}$	$\mathbf{4.3}^{13-3}$

m	32	64	128	256	512	1024	2048	4096
WFR*q*	7.88^{4}	6.09^{4}	5.43^{4}	3.78^{4}	2.18^{4}	1.32^{4}	1.01^{4}	0.99^{6}
QF(*Q*, *S*)	$6.73^{3,4}$	$5.31^{4,3}$	$4.6^{4,3}$	$3.77^{4,3}$	$2.05^{4,3}$	$1.4^{4,3}$	$1.51^{4,3}$	$2.2^{4,3}$
FS-w*q*	4.98^{8}	4.57^{8}	4.17^{8}	4.63^{8}	3.54^{8}	2.83^{8}	2.65^{8}	2.85^{8}
BSDM*q*	6.83^{2}	5.92^{4}	5.32^{4}	4.65^{4}	3.91^{4}	3.43^{4}	3.22^{4}	3.08^{4}
skip-*q*	7.06^{2}	5.91^{4}	5.19^{4}	4.15^{4}	2.6^{4}	1.98^{4}	1.98^{4}	4.13^{4}
FSBNDM*qf*	6.6^{31}	5.98^{51}	5.72^{51}	4.32^{51}	2.48^{51}	1.39^{51}	0.9^{51}	0.82^{51}
SBNDM*q*	6.81^{2}	5.9^{4}	5.36^{4}	3.94^{4}	2.12^{4}	1.24^{4}	0.84^{4}	0.66^{6}
LBNDM	16.75	12.06	9.38	6.28	4.08	3.03	2.43	1.98
SSEF	12.97	6.36	4.98	3.39	2.11	1.67	2.05	5.05
EPSM	6.04	4.98	4.34	3.28	2.53	2.34	2.79	5.52
Z-Byte(*k*)-w*q*	$\mathbf{4.08}^{14-3}$	$\mathbf{3.75}^{14-3}$	$\mathbf{3.55}^{14-5}$	$\mathbf{2.23}^{16-3}$	$\mathbf{1.17}^{16-3}$	$\mathbf{0.64}^{16-3}$	$\mathbf{0.56}^{16-3}$	$\mathbf{0.47}^{15-5}$

Algorithm 8: The preprocessing phase of the RZk-Bit algorithm

```
1  $mask \leftarrow 2^k - 1$; $m \leftarrow \lfloor l/b \rfloor$; $lastMask \leftarrow byte((2^b - 1) << (b - l \bmod b))$;
2  foreach $i \in [0; b)$ do $c \leftarrow (P[0] << i) | (P[1] >> (b - i))$; $\lambda[c] \leftarrow \lambda[c] \cup \{i\}$;
3  foreach $i \in [0; 2^k)$ do $Z[i] \leftarrow 1$;
4  foreach $i \in [0; l - b)$ do
5      $t \leftarrow bitWord(i, mask)$;
6      if $l - i \geq 2b$ then
7          $Z[t] \leftarrow 0$;
8      else
9          if $l - i > 3b - k$ then $s \leftarrow 2b - (l - i)$;        // 2 (a)
10         else $s \leftarrow k - b$;                                   // 2 (b)
11         foreach $j \in [0; 2^s)$ do $Z[t | (j << b)] \leftarrow 0$;
```

Table 3 Running times on a bitstream (hundredth of seconds)

Pattern length (bits)	25	50	100	200	400	800	1600	3200
RZ(k)-Bit	39.92^{16}	14.42^{14}	10.02^{15}	7.69^{15}	5.21^{16}	4.27^{14}	3.19^{15}	2.32^{14}
RZ-Bit16-w2	**27.72**	**11.74**	**7.91**	**6.08**	**4.9**	**4.24**	**3.11**	**1.97**
BFL	134.8	44.96	21.61	13.57	14.72	38.34	114.81	169.37
FED	205.08	68.11	30.03	15.95	16.82	29.09	41.87	68.96

Algorithm 9: Function $bitWord(i, mask)$, little endian machine

```
1 $r \leftarrow \lfloor i/b \rfloor$;                              // index of the highest byte
2 $s \leftarrow (P[r] << 2b)|(P[r+1] << b)|P[r+2]$;        // load 3 bytes from
    memory
3 $s \leftarrow s << (i \mod b)$; // shift to the left edge of a 3-byte word
4 $mask1 \leftarrow (2^b - 1) << 2b$; // mask for the highest byte, 0xff0000
5 $mask2 \leftarrow mask\&((2^b - 1) << b)$;       // mask for the middle byte
6 return $((s\&mask1) >> 2b)|(s\&mask2)$;     // move highest byte to
    lowest
```

Pattern Matching on Small Alphabets

Assume the text alphabet consists of ≤ 128 characters, and the leftmost bits of a character are insignificant. Then the value $word(T[\text{pos}1], T[\text{pos}1+1])\&mask$ consists of the 'hole' of insignificant bits. Nonetheless, even under this assumption algorithms 2–5 work correctly without change. However, their efficiency is less as the hole size is bigger. For example, when $|\Sigma| = 16$ and $k = 12$, only 8 bits of each 12-bit element of a shift table are significant. In this case, a method based on combining only significant bits of several adjacent bytes into a k-bit value may perform faster.

Assume $\sigma = \lceil \log_2 |\Sigma| \rceil$ is the number of bits required to store a character of alphabet Σ, q is the number of characters processed at each iteration of a search loop. Then the i-th q-gram of adjacent pattern characters can be converted into a $q\sigma$-bit index of a shift table by the expression $P[i] + P[i+1] << \sigma + \cdots + P[i+q-1] << (q-1)\sigma$. This formula is illustrated in Fig. 3 for $|\Sigma| = 4, \sigma = 2, q = 3$.

Applying this conversion to pattern characters in the preprocessing phase and texts characters in the search phase, we get the pattern matching Alg. 10. We call it a 'Bricks' algorithm since it presumes building an index of a shift table from σ-bit values like a wall from bricks. If the 'wall' is constructed from q σ-bit 'bricks, we denote the algorithm as Bricks$\sigma - q$.

The search phase consists of the main loop (lines 5–9) including the fast loop (lines 6–7). If the fast loop condition is satisfied, this means that the q-character suffix $T[\text{pos} \dots \text{pos} + q - 1]$ of a search window does not belong to

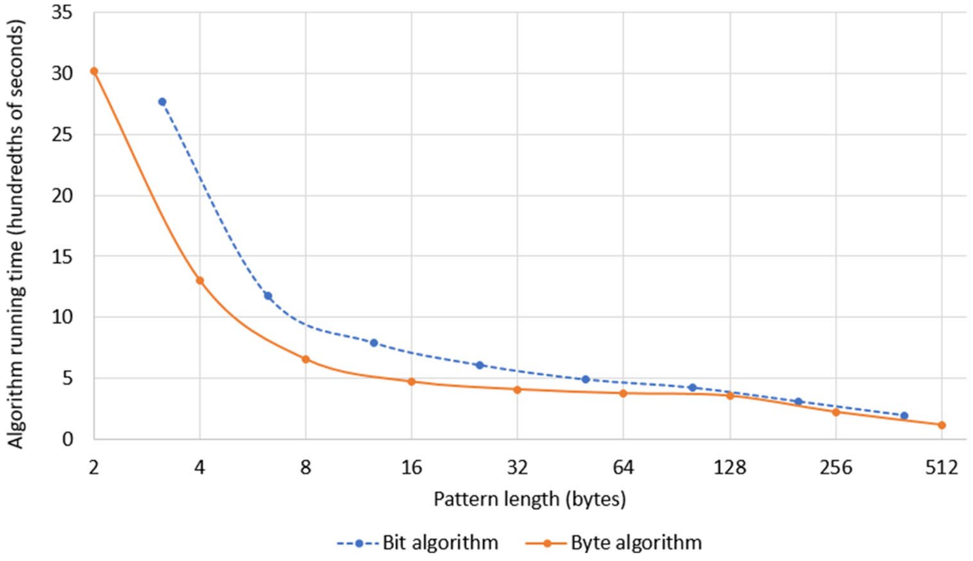

Fig. 4 The running times of the bitstream pattern search algorithm and the underlying byte version

Table 4 Running times on English text (short patterns on top, long patterns on bottom)

m	2	4	8	10	12	14	16	18	20
WFRq	109.53	46.99^{2}	26.03^{2}	21.83^{4}	17.64^{4}	15.22^{4}	13.56^{4}	12.2^{4}	11.3^{4}
QF(Q, S)	–	54.39^{4,3}	18.82^{4,3}	14.79^{4,3}	12.36^{4,3}	10.94^{4,3}	9.95^{4,3}	9.3^{4,3}	8.85^{4,3}
FS-wq	90.03^{6}	48.66^{6}	27.92^{6}	23.51^{6}	20.84^{6}	19.34^{6}	17.25^{6}	16.73^{6}	15.55^{6}
BSDMq	73.13^{2}	33.24^{2}	22.09^{2}	18.51^{4}	14.74^{4}	12.8^{4}	11.5^{4}	10.53^{4}	9.95^{4}
skip-q	99.65^{2}	43.4^{2}	26.75^{2}	20.22^{4}	16.28^{4}	13.91^{4}	12.34^{4}	11.26^{4}	10.48^{4}
FSBNDMqf	99.43^{31}	36.48^{31}	18.11^{31}	15.42^{31}	12.94^{31}	11.79^{31}	11.19^{31}	10.73^{31}	10.36^{31}
SBNDMq	66.18^{2}	29.7^{2}	20.1^{2}	17.83^{2}	14.25^{4}	12.31^{4}	11.31^{4}	10.28^{4}	9.69^{4}
EBOM	88.71	35.4	20.95	18.47	16.86	16.18	15.75	15.5	14.8
EPSM	**9.21**	**11.78**	**12.86**	**12.41**	12.34	12.05	**9.43**	9.58	9.82
Z16-w3	38.95	21.01	15.39	14.7	13.92	13.65	13.41	13.14	13.02
Bricks$\sigma q(k)$	67.24$^{22(16)}$	31.69$^{22(16)}$	16.81$^{43(15)}$	13.68$^{43(15)}$	**11.65**$^{43(14)}$	**10.53**$^{43(13)}$	9.85$^{43(14)}$	**9.19**$^{43(15)}$	**8.84**$^{43(14)}$
m	24	32	64	128	256	512	1024	2048	4096
WFRq	9.98^{4}	8.54^{4}	6.64^{4}	5.87^{4}	4.39^{4}	2.84^{4}	1.64^{6}	1.17^{6}	1.03^{6}
QF(Q, S)	8.14^{4,3}	7.22^{4,3}	6.01^{4,3}	5.31^{4,3}	4.04^{4,3}	2.71^{4,4}	1.94^{4,3}	1.55^{4,3}	1.55^{4,3}
FS-wq	14.29^{6}	12.22^{6}	9.66^{6}	8.23^{8}	7.76^{8}	7.89^{8}	7.24^{8}	6.43^{8}	5.77^{8}
BSDMq	9.03^{4}	8.14^{4}	6.64^{6}	6.16^{6}	6.04^{6}	5.88^{8}	5.42^{8}	4.92^{8}	4.69^{8}
skip-q	9.32^{4}	8.03^{4}	6.48^{4}	5.73^{4}	4.57^{4}	3.43^{4}	2.57^{8}	3.0^{8}	3.6^{8}
FSBNDMqf	9.61^{51}	8.05^{51}	6.74^{51}	6.64^{51}	6.44^{51}	5.44^{51}	4.83^{51}	3.55^{51}	2.25^{51}
SBNDMq	8.92^{4}	7.73^{4}	6.59^{4}	6.25^{4}	5.61^{6}	4.54^{6}	4.26^{6}	3.25^{6}	1.8^{6}
EBOM	14.22	13.5	12.32	10.39	8.81	8.22	10.42	7.8	10.4
SSEF	–	11.56	6.51	5.19	3.4	**2.09**	1.47	2.5	3.83
EPSM	**7.12**	**6.21**	**5.2**	**4.42**	**3.32**	2.6	2.47	3.51	5.28
Z16-w3	12.55	11.77	9.65	7.82	7.092	6.74	4.85	3.84	3.1
Bricks$\sigma q(k)$	8.24$^{34(15)}$	7.18$^{34(16)}$	5.96$^{25(13)}$	5.06$^{34(15)}$	3.81$^{34(15)}$	2.39$^{34(15)}$	**1.46**$^{26(15)}$	**0.93**$^{26(15)}$	**0.72**$^{26(15)}$

the pattern and this window can be safely moved $m - q + 1$ characters forward (line 7). Otherwise, the occurrence check is performed in line 8. After the occurrence check a search window can be safely moved by a 'Quick search' shift [24] in line 9 (filling the table QS is not shown in the preprocessing phase).

```
Algorithm 10: Bricksσ-q algorithm for q-grams on Σ, σ = log |Σ|
1 foreach $i \in [0; 2^{q\sigma})$ do $Z[i] \leftarrow 1$;            // Preprocessing
2 for $i \leftarrow 0$ to $m - q$ do
3     $Z[P[i] + P[i+1] << \sigma + \ldots + P[i+q-1] << (q-1)\sigma] \leftarrow 0$
4 $pos \leftarrow m - q$;                                              // Search
5 while $pos \leq n - m + q$ do
6     while $Z[T[pos] + T[pos+1] << \sigma + \ldots + T[pos+q-1] << (q-1)\sigma] \neq 0$ do
7         $pos \leftarrow pos + m - q + 1$
8     check the occurrence at $pos - m + q$
9     $pos \leftarrow pos + QS[T[pos+q]]$
```

Table 5 Running times on a protein sequence (short patterns on top, long patterns on bottom)

m	2	4	6	8	12	16	18	20	22
WFRq	33.38	14.33^{2}	10.03^{2}	8.18^{2}	5.17^{4}	3.87^{4}	3.47^{4}	3.02^{4}	2.79^{4}
QF(Q, S)	–	$16.05^{3,4}$	$8.18^{3,4}$	$5.51^{3,4}$	$3.41^{4,3}$	$2.64^{4,3}$	$2.4^{4,3}$	$2.1^{4,3}$	$2.06^{4,3}$
FS-wq	25.06^{6}	13.36^{6}	9.3^{6}	7.36^{6}	5.52^{6}	4.56^{6}	4.26^{6}	4.06^{6}	3.87^{6}
BSDMq	23.77^{2}	10.57^{2}	8.03^{2}	7.08^{2}	4.61^{4}	3.48^{4}	3.19^{4}	2.91^{4}	2.7^{4}
skip-q	31.91^{2}	13.74^{2}	10.03^{2}	8.39^{4}	4.85^{4}	3.48^{4}	3.13^{4}	2.79^{4}	2.6^{4}
FSBNDMqf	26.16^{31}	9.08^{31}	5.84^{31}	4.55^{31}	3.16^{31}	2.63^{31}	2.59^{31}	3.86^{31}	2.4^{31}
SBNDMq	20.57^{2}	8.09^{2}	5.86^{2}	4.59^{2}	3.72^{2}	3.06^{4}	2.74^{4}	2.45^{4}	2.25^{4}
EBOM	27.6	9.17	6.45	5.01	3.67	3.23	3.02	2.84	2.83
EPSM	**3.16**	**3.56**	3.5	3.56	3.43	2.73	2.77	2.77	2.88
Z16-w3	10.64	4.34	**3.12**	**2.62**	**2.19**	**2.07**	**2.07**	**1.99**	**2.02**
Bricks$\sigma q(k)$	$27.52^{22(16)}$	$13.72^{22(16)}$	$7.39^{43(16)}$	$5.17^{43(15)}$	$3.48^{43(14)}$	$2.73^{43(15)}$	$2.56^{43(15)}$	$2.32^{43(14)}$	$2.3^{43(14)}$
m	24	32	64	128	256	512	1024	2048	4096
WFRq	2.7^{4}	2.09^{4}	1.41^{4}	0.84^{6}	0.56^{6}	0.32^{6}	0.24^{6}	0.26^{6}	0.44^{6}
QF(Q, S)	$1.89^{4,3}$	$1.6^{4,3}$	$0.98^{4,3}$	**0.68**^{4,3}	**0.45**^{4,3}	$0.27^{4,3}$	$0.21^{4,3}$	$0.25^{4,3}$	$0.4^{4,3}$
FS-wq	3.61^{6}	3.31^{6}	2.69^{6}	2.38^{6}	2.18^{6}	2.2^{6}	2.13^{6}	2.23^{6}	2.42^{6}
BSDMq	2.44^{4}	2.22^{4}	1.67^{4}	1.38^{8}	1.17^{8}	1.23^{8}	1.09^{8}	1.15^{8}	1.2^{8}
skip-q	2.29^{4}	1.96^{4}	1.29^{4}	0.98^{4}	0.63^{4}	0.65^{4}	0.73^{4}	1.02^{4}	3.05^{4}
FSBNDMqf	2.13^{31}	1.83^{51}	1.21^{51}	0.95^{51}	1.27^{51}	1.06^{51}	0.83^{51}	0.55^{51}	0.27^{51}
SBNDMq	2.02^{4}	1.68^{4}	1.12^{4}	0.86^{4}	0.67^{6}	0.97^{6}	0.8^{6}	0.54^{6}	0.28^{6}
EBOM	2.71	2.51	2.67	3.08	2.94	3.61	5.91	9.8	14.67
SSEF	–	2.42	1.37	0.96	0.51	0.49	0.64	1.16	1.94
EPSM	**1.65**	**1.33**	**0.94**	0.74	0.48	0.6	0.83	1.35	2.33
Z16-w3	1.89	1.86	1.73	1.44	1.2	0.93	0.88	1.33	2.23
Bricks$\sigma q(k)$	$2.12^{43(15)}$	$1.82^{25(14)}$	$1.14^{25(14)}$	$0.78^{25(14)}$	$0.48^{26(15)}$	**0.24**$^{26(14)}$	**0.18**$^{26(14)}$	**0.15**$^{26(14)}$	**0.15**$^{26(15)}$

The main shortcoming of Alg. 10 appears on short patterns when the value $m - q + 1$ and thus the shift in the fast loop becomes too small. One way to increase the shift length is to ensure that suffixes of a search window do not coincide with the prefixes of a pattern. Assume suffixes of a search window/prefixes of a pattern of length $q - t, \dots, q - 1$ are checked. Then the search window can be safely moved $m - q + t + 1$ characters forward and we get Algorithm 11.

Table 6 Running times on a genome sequence (short patterns on top, long patterns on bottom)

m	2	4	8	10	12	14	16	18	20	22
WFRq	71.51^{2}	53.77^{2}	14.4^{4}	10.63^{4}	8.87^{4}	7.74^{4}	7.05^{4}	6.46^{4}	5.76^{6}	5.26^{6}
QF(Q, S)	–	30.91^{4,3}	10.24^{4,3}	**8.38**^{4,3}	**7.33**^{4,3}	**6.3**^{4,3}	5.12^{4,3}	4.99^{4,3}	4.69^{4,3}	4.44^{4,3}
BSDMq	67.41^{2}	50.98^{2}	13.467^{4}	10.62^{4}	9.29^{4}	8.32^{4}	7.37^{6}	6.59^{6}	6.03^{6}	5.49^{6}
skip-q	71.35^{2}	58.85^{4}	15.22^{4}	11.94^{4}	10.11^{4}	8.93^{4}	8.13^{4}	7.33^{6}	6.56^{6}	5.99^{6}
FSBNDMqf	70.28^{31}	33.99^{31}	15.19^{41}	12.27^{41}	10.46^{41}	9.08^{41}	8.41^{41}	7.66^{41}	7.13^{41}	6.74^{41}
SBNDMq	57.28^{2}	46.7^{2}	12.88^{4}	10.06^{4}	8.5^{4}	7.51^{4}	6.95^{4}	6.41^{4}	5.65^{6}	5.09^{6}
Hash$q$	–	98.24^{3}	34.21^{3}	26.24^{3}	21.71^{3}	18.46^{3}	16.46^{3}	14.76^{3}	13.46^{3}	12.28^{5}
EPSM	**4.42**	**5.04**	**9.07**	8.84	8.96	9.0	**4.11**	**4.17**	**4.29**	4.32
Bricks2q-t(k)	58.0$^{2(16)}$	29.04$^{3(16)}$	12.17$^{4(15)}$	9.15$^{5-1(16)}$	7.39$^{5-1(16)}$	6.33$^{5-1(16)}$	5.75$^{5(13)}$	5.18$^{5(13)}$	4.74$^{6(14)}$	**4.26**$^{6(14)}$

m	24	32	64	128	256	512	1024	2048	4096
WFRq	4.83^{6}	3.81^{6}	2.52^{6}	1.82^{6}	1.37^{6}	0.89^{8}	0.69^{8}	0.63^{8}	0.78^{8}
QF(Q, S)	4.21^{4,3}	3.6^{6,2}	2.61^{6,2}	1.95^{6,2}	1.48^{6,2}	1.23^{6,2}	1.14^{6,2}	1.03^{6,2}	1.05^{6,2}
BSDMq	5.26^{6}	4.59^{6}	3.46^{8}	3.09^{8}	2.88^{8}	2.88^{8}	2.91^{8}	2.99^{8}	2.95^{8}
skip-q	5.55^{6}	4.56^{8}	3.1^{8}	2.56^{8}	2.05^{8}	1.64^{8}	1.63^{8}	2.03^{8}	3.41^{8}
FSBNDMqf	6.37^{41}	5.39^{41}	5.49^{41}	5.38^{41}	5.52^{41}	3.57^{51}	2.07^{51}	1.05^{51}	0.47^{51}
SBNDMq	4.61^{6}	3.49^{6}	4.61^{6}	5.22^{8}	5.07^{8}	3.12^{6}	1.31^{6}	0.74^{6}	0.37^{6}
Hash$q$	11.07^{5}	8.06^{5}	4.89^{5}	3.81^{8}	3.46^{8}	2.13^{8}	1.41^{8}	0.93^{8}	0.61^{8}
SSEF	–	2.47	**1.59**	**1.2**	**0.71**	0.64	0.95	1.69	3.18
EPSM	**2.94**	**2.27**	1.62	1.25	0.94	0.88	1.2	1.91	3.42
Bricks2q-t(k)	3.92$^{6(14)}$	3.13$^{6(14)}$	2.21$^{6(15)}$	1.61$^{6(14)}$	1.11$^{7(15)}$	**0.63**$^{7(14)}$	**0.38**$^{8(15)}$	**0.28**$^{8(15)}$	**0.27**$^{8(15)}$

Algorithm 11: Bricksσ-q-t algorithm for q-grams on Σ and checking t pattern prefixes, $\sigma = \log|\Sigma|$

```
foreach $i \in [0; 2^{q\sigma})$ do $Z[i] \leftarrow 1$;          // Preprocessing
for $i \leftarrow 0$ to $m - q$ do
    $Z[P[i] + P[i+1] << \sigma + \ldots + P[i+q-1] << (q-1)\sigma] \leftarrow 0$
for $i \leftarrow 0$ to $|\Sigma|$ do
    $Z[i + P[0] << \sigma + \ldots + P[q-2] << (q-1)\sigma] \leftarrow 0$
...
for $i \leftarrow 0$ to $|\Sigma|^t$ do
    $Z[i + P[0] << t\sigma + \ldots + P[q-t-1] << (q-1)\sigma] \leftarrow 0$
$pos \leftarrow m - q$;                                          // Search
while $pos \leq n - m + q$ do
    while $Z[T[pos] + T[pos+1] << \sigma + \ldots + T[pos+q-1] << (q-1)\sigma] \neq 0$ do
        $pos \leftarrow pos + m - q + t + 1$
    check the occurrence at $pos - m + q$
    $pos \leftarrow pos + QS[T[pos+q]]$
```

The preprocessing phase of Alg. 11 differs from the preprocessing phase of Alg. 10 with the loops in lines 4–8. In these loops, the value 0 is assigned to elements of shift table Z with indexes formed from prefixes of a pattern padded to the left with all possible combinations of Σ characters. This ensures that suffixes of a search window of lengths $q-t, \ldots, q-1$ do not coincide with prefixes of a pattern, and in the fast loop, this window can be moved safely t characters more to the right than in Alg. 10 (line 12 of Alg. 11 vs. line 7 of Alg. 10).

Let us note that occurrence check avoiding techniques discussed in "Pattern matching over a byte alphabet", such as the double fast loop or using the sliding windows, lead to measurable improvement in Bricks algorithm performance for relatively large alphabets ($|\Sigma| \geq 16$) and small patterns ($m < 8$) only, i.e. in the area where Bricks algorithms are outperformed by Z/RZ ones, as will be shown in "Experimental results". This may be explained by the fact that the computational complexity of a bad character check in Bricks algorithms (e.g. in line 6 of Alg. 10) is relatively large in comparison with the occurrence check complexity, particularly when the bad character check expression is long, which means $|\Sigma|$ is small. That is why we do not apply the double fast loop and sliding windows to algorithms of the Bricks family.

Also, it is worth noting that pattern search in genome sequences represented as a series of ACGT-characters requires the modification of Bricks algorithms. The binary representations of ASCII nucleotide characters are the following: $A = 01000001, C = 01000011, G = 01000111, T = 01010100$. And after applying expressions from lines 3 or 6 of Alg. 10 to these binary strings we may get the index values greater than $2^{q\sigma}$. Therefore, to limit the values of shift table indices, the $mask = 2^k - 1$ can be applied to 'Bricks' expressions ($k \geq q\sigma$). Also, note that the only pair of bits different in all binary representations of ACGT-characters consists of 2-nd and 3-rd bits from the right. Thus, it is reasonable to assume these pairs of bits of sequential characters in the 'Bricks' expression should not intersect. According to these considerations, the following expressions should be used in lines 3 and 6 of DNA-version of Alg. 10.

Algorithm 12: Bricksσ-$q(k)$ – modification of Alg. 10 for a DNA sequence; k is the number of bits in a mask

```
1 $mask \leftarrow 2^k - 1$
2 foreach $i \in [0; 2^k)$ do $Z[i] \leftarrow 1$;                    // Preprocessing
3 ...
4 $Z[(P[i] >> 1\ +\ P[i+1] << 1\ +\ \ldots\ +\ P[i+q-1] << (2q-3))\&mask] \leftarrow 0$
5 ...
6 while $Z[(T[pos] >> 1\ +\ T[pos+1] << 1\ +\ \ldots\ +\ T[pos+q-1] << (2q-3))\&mask] \neq 0$ do
```

Of course, the ‘false positives’ are possible, i.e. the situations when the expression in line 6 is 0, although the substring $T[\text{pos} \ldots \text{pos} + q - 1]$ does not belong to the pattern. This leads to extra occurrence checks, however, their number is negligibly small in practice. In fact, the shift table in Bricks algorithms implements some hash function, and the ‘Bricks’ approach, in general, resembles the Hashq algorithms [19], as well as the ‘filtering’ expression in SSEF [18] or QF [6]. The main differences of Bricks algorithms are: (a) more than 1 bit of each text character can participate in the shift table index calculation (unlike SSEF), (b) characters are shifted more than 1 bit relative to each other, (c) the exact value of a shift is not read from memory in the fast loop (unlike Hashq), and (d) filtering expression is not recurrent and the mask size does not depend on the number of processed characters (unlike QF). As experiments presented in “Experimental results” show, these differences make sense in practice.

Experimental Results

A wide range of Z/RZ-Byte, Z/RZ-Bit, and Bricks algorithms was tested with different parameters on different types of text and pattern lengths varying from 2 to 4096. Several known algorithms were tested for comparison, both in original and “2-byte read” versions, if applicable. The results are presented in Tables ***6–2. The test dataset contains the following texts:

- Text on a byte alphabet: 10 MB text contains a set of English Wikipedia articles encoded by (s, c)-dense code [2]. This code is byte-aligned, i.e. a phrase from the original text can be found in the encoded text as a sequence of bytes. The results are shown in Table 2.
- Text on a byte alphabet for a bitstream search: 10MB text contains a set of English Wikipedia articles encoded by the multi-delimiter code $D_{2,4-\infty}$ [1]. Although this code compresses texts not far from entropy (2–3% away), it makes possible the direct data search in a compressed file without its decompression, just like (s, c)-dense code. However, unlike SCDC, the multi-delimiter code is not byte-aligned and the pattern may not be aligned with the byte boundaries. Thus, the bitstream search should be applied. The results are shown in Table 3.
- Natural language text (Table 4): collected works of Charles Dickens, 10 MB text taken from Silesia corpus.
- Proteine (Table 5): the 3.2MB of the amino acid sequences of different proteins taken from SMART tool [8].
- DNA: e-coli genome sequence, 4.5 MB text. The search results are presented in Table 6.

The algorithms have been compiled with GCC compiler (full optimization for speed) and run on a little-endian computer with an Intel i5-8265U processor, 4×32 kB L1 cache, 8 GB RAM, OS Windows 10 64-bit. The results, given in hundredths of seconds, represent an average time of 1000 runs of each algorithm. The C code of some Bricks and Z-algorithms is published in [25], while most of the other algorithms have been taken from [8].

For each pattern length, one algorithm providing the best searching time has been selected in each algorithm family to be presented in a table. If the family is parametrized, the parameters are shown in superscript or in smaller font below the time value. The performance of the fastest algorithm, for each pattern length, is given in bold.

Z/RZ-Byte algorithms have been run in versions with 1, 3, and 5 sliding windows and the parameter k varying in the range 12–15, while the algorithms Z8 and Z16, based on a full byte read, have been tested in 1, 2, and 3 sliding windows versions. Also, all reasonable combinations of q and t parameters of Bricks algorithms have been tested for each text.

Let us discuss the performance of the algorithms on different real-world data. The EPSM algorithm wins the competition on very short patterns ($m \leq 4$) on all investigated datasets. This is because this algorithm process multiple adjacent short search windows simultaneously, using SIMD operations. However, while the pattern length, and hence the search window length increases, this advantage disappears, and results can be different.

Thus, on a byte alphabet, the Z-algorithms are the fastest for all patterns of length $m \geq 6$ (Table 2). For short and medium patterns the 14- or 13-bit read is preferable, while for long patterns ($m \geq 256$) a full 2-byte read or 15-bit read provides better results. This testifies that our "1.5-byte read" approach appears to be more efficient than a 2-byte read if the alphabet is large and the pattern is not extremely long. Indeed, in this case, the probability of a maximal shift based on analyzing the 2-byte suffix of a search window is not much higher than if we analyze 13–14 bits, however, the shift table fits into the L1 cache, which is a more important factor.

Among other algorithms, the bit-parallel (F)SBNDM and QF, as well as automaton-based WFR, demonstrate good performance on long patterns being however 30–120% inferior to best representatives of comparison-based Z-family. The main reason for that is that on large alphabets comparison-based algorithms can provide quite good average shift length having at the same time less operational complexity if the following improvements are applied: (a) involving bits of more than one character into bad-character comparisons; (b) making use of multiple sliding search windows; (c) tuning the fast loop technique.

This is confirmed by the performance of bitstream pattern matching algorithms, presented in Table 3. For all investigated pattern lengths, from 25 to 3200 bits, the RZBit16-w2 algorithm, based on principles (a)–(c), outperforms more than twice the BFL method, based upon the bit-parallel algorithm BNDM. Although RZBit16-w2 makes use of 2 sliding windows, the noted outperformance is essential even for no-sliding windows versions of RZ-Bit (row 1 of Table 3). Apart from the above-mentioned reasons, the superiority of the RZ-Bit family may be explained by the following factors: (d) although the maximal length of a long-shift in the BFL algorithm is $2m - 1$, in practice it is often $2m - 3$ or less, while any total maximal shift in two sliding windows of RZ-Bit algorithm is $2m - 2$, i.e. longer in average; (e) when the pattern is shortened below 60 bits, and importance of a long-shift decreases, the timing difference between RZ-Bit and BFL becomes especially large, which indicates a higher efficiency of a bit-alignment checking technique implemented in the RZ-Bit algorithm; (g) the long shift in BFL cannot be longer than 2 machine word lengths, which makes the algorithm inefficient in long pattern matching.

The running times of best bitstream pattern matching algorithms and corresponding byte alphabet search algorithms are compared in Fig. 4. They differ significantly on short patterns only. This indicates that the bit-alignment checking technique implemented in the RZ-Bit algorithm is quite efficient and cannot be improved a lot, except for application to very short patterns.

When it comes to pattern matching in texts on smaller alphabets, the relative performance of Z/RZ-algorithms decreases and we also take into consideration the Bricks algorithm. Experiments show that Bricks algorithm demonstrates higher efficiency on smaller alphabets and longer patterns, while Z/RZ algorithms vice versa (Tables 4, 5, 6).

Among Bricksσ-q algorithms, the ones with smaller q are more efficient on shorter patterns. They provide longer fast loop shifts by the cost of fast loop continuation probability and this strategy is more efficient as the ratio $(m - q + 1)/m$ is smaller, i.e. m is smaller.

On English text (Table 4) Z/RZ-algorithms do not win the competition on any pattern length, while different variants of the Bricks algorithm appear to be the fastest for very long patterns ($m \geq 1024$) and some middle pattern lengths ($m = 12, 14, 18, 20$). In the latter area, the Bricks algorithm competes mostly with EPSM, and its outperformance on some disconnected intervals of pattern lengths can be partially explained by a stepped character of EPSM performance (i.e. it may not change on wide intervals as pattern length increases).

The special version of a Bricks algorithm (Alg. 12) is faster than the standard version (Alg. 10) not only in genomic but either in protein sequence search; its timing shown in Tables 5 and 6, with algorithm parameters in small font under the time values. In most cases the parameter t is omitted, except for $5 - 1(16)$, which denotes the combination of Alg. 11 and 12, $q = 5, t = 1$.

As seen, the Bricks algorithm outperforms all other algorithms in search of long patterns, $m \geq 512$, either for genomic or protein sequences. Also, it demonstrates good performance for medium patterns, $10 \leq m \leq 256$, being somewhat inferior to EPSM, QF, SSEF, and/or Z-algorithms (however, for genomic patterns of length 22 the Bricks26-14 algorithm appears to be the fastest one again).

Z-algorithms, being obviously inefficient in genomic search, demonstrate very good results for protein sequences. The algorithm $Z16 - w3$ with 2-byte read, double fast loop, and 3 sliding windows appears to be the fastest one among Z-algorithms on all pattern lengths and the fastest among all algorithms for $6 \leq m \leq 22$.

Let us note, that the outperformance of the Z16-w3 algorithm, implementing a 2-byte read, over 1.5-read versions of Z-algorithms on English text and protein sequences is expected since either in DNA, protein or English texts the highest bits of bytes are always zero. Thus, the shift table based on 2 characters occupies 32 kB of memory, and fitting to the L1 cache problem becomes almost irrelevant. The same reason explains why we use a full 2-byte read in the Bricks22 and Bricks23 algorithms: their filter expressions cannot exceed 2^{13}.

Conclusions

The different speeding mechanisms of the classical Boyer–Moore–Horspool algorithm were investigated and tested on real-world datasets of different nature. Combining these mechanisms, we get three parameterized algorithms. The Z/RZ-bit algorithm represents an efficient approach to pattern matching in a bitstream. The underlying solution, the Z/RZ-byte algorithm, can be used for string matching on large alphabets. It relies on improving the 2-byte read principle as well as tuning the fast loop and sliding windows techniques. And the third algorithm, Bricks, is efficient mostly on small alphabets. The test results show that different representatives of the developed set of algorithms outperform all other tested solutions on a wide range of pattern lengths and different data sets.

Appendix: Remarks on Implementation

There are three adjustments in C versions of our algorithms, which are not shown in Alg. 2–6 for simplicity.

The first adjustment deals with addressing the characters of a text. It is reasonable to assume that addressing a character via pointer like `*ptr` will be a bit faster than addressing the array element like `T[pos]`, because the latter expression in fact means `*(T+pos)` and requires one extra addition. Whilst other operations such as additions and comparisons have the same time complexity either for pointers or indices. Indeed, for our algorithms, the experiments show that the gain from using pointers instead of indices is up to 20% for patterns of length 2, up to 6% for patterns of length 4, 0.5–1.5% for patterns of length 8, and insignificant for longer patterns. For other algorithms, the results are rather contradictory. That is why all Z/RZ and Bricks algorithms were tested in "pointer" versions, while other algorithms have been run in their original versions. Thus, to make a comparison more relevant, the timing of Z/RZ/Bricks algorithms may be increased in the above-mentioned proportions.

The second adjustment has to be made in reverse algorithms in order to correctly process the algorithm stop condition. In algorithms, 4–6 the text was assumed to be prepended by a stop pattern. However, it can be problematic due to "left-to-right" organization of memory structures in programming languages: we can easily allocate extra memory after the text to append a string to it, but there is no technique to allocate extra memory just before the address stored in a given pointer T addressing the beginning of a text, except for shifting the whole text right or taking into account this specificity before invoking a search function. The other option is to: (1) before the main search loop, backup the beginning of a text and replace it with a stop pattern; (2) after the main loop, restore the beginning of a text and search the pattern in it using some other simple algorithm. This approach has been implemented in the tested reverse algorithms.

And the last adjustment relates to Z/RZ algorithms searching the short patterns, less than 3 bytes in length. In this case, the length of a maximal shift in Zk-Byte or RZ-Bit algorithms is equal to $m - 1 = 1$. And since in the double fast loop we step 1 character back, this loop may become endless. Therefore, in Zk-Byte ($8 < k \leq 16$) and RZ-Bit algorithms, the short patterns are considered as a special case and processed with a single fast loop.

Declarations

Conflict of interest Igor O. Zavadskyi states that there is no conflict of interest.

References

1. Anisimov AV, Zavadskyi IO. Variable-length prefix codes with multiple delimiters. IEEE Trans Inf Theory. 2017;63(5):2885–95. https://doi.org/10.1109/TIT.2017.2674670.
2. Brisaboa N, Farina A, Navarro G, Esteller M. (s,c)-dense coding: an optimized compression code for natural language text databases. In: Proceedings of symposium on string processing and information retrieval. 2003;2857, pp. 122–136. Manaus, Brazil.
3. Cantone D, Faro S. Fast-search algorithms: new efficient variants of the Boyer–Moore pattern-matching algorithm. J Autom Lang Comb. 2005;10(5/6):589–608.
4. Commentz-Walter B. A string matching algorithm fast on the average. In: Proceedings of international colloquium on automata, languages, and programming; 1979;pp. 118–132.
5. Durian B, Holub J, Peltola H, Tarhio J. Tuning BNDM with q-grams. In: Finocchi I, Hershberger J, editors. Poceedings of the workshop on algorithm engineering and experiments. New York: SIAM; 2009. p. 29–37. https://doi.org/10.1137/1.9781611972894.3.
6. Durian B, Peltola H, Salmela L, Tarhio J. Bit-parallel search algorithms for long patterns. In: 9th international symposium on experimental algorithms, Ischia Island, Naples, Italy. Springer; 2010. p. 129–40.
7. Faro S, Külekci M. Fast and flexible packed string matching. J Discret Algorithms. 2014;28:61–72.
8. Faro S, Lecroq T. String matching research tool: exact string matching algorithms. https://www.dmi.unict.it/faro/smart/algorithms.php.
9. Faro S, Lecroq T. Efficient variants of the backward-oracle-matching algorithm. In: Holub J, Zdarek J, editors. Proceedings of the Prague stringology conference. Czech Republic: Czech Technical

University in Prague; 2008. p. 146–60. https://doi.org/10.1142/S0129054109006991.
10. Faro S, Lecroq T. An efficient matching algorithm for encoded DNA sequences and binary strings. In: Kucherov G, Ukkonen E, editors. Proceedings of combinatorial pattern matching; 2009. p. 106–115.
11. Faro S, Lecroq T. Efficient pattern matching on binary strings. In: 35th international conference on current trends in theory and practice of computer science, p. Poster. 2009.
12. Faro S, Lecroq T. The exact online string matching problem: a review of the most recent results. ACM Comput Surv. 2013;45(2):article 13. https://doi.org/10.1145/2431211.2431212.
13. Horspool NR. Practical fast searching in strings. Soft Pract Exp. 1980;10(6):501–6. https://doi.org/10.1002/spe.4380100608.
14. Hudaib A, Al-khalid R, Suleiman D, Itriq MAA, Al-Anani A. A fast pattern matching algorithm with two sliding windows (TSW). J Comput Sci. 2008;4(5):393–401. https://doi.org/10.3844/jcssp.2008.393.401.
15. Hume A, Sunday D. Fast string searching. Softw Pract Exp. 1991;21(11):1221–48.
16. Kim J, Kim E, Park K. Fast matching method for DNA sequences. In: Proceedings of combinatorics, algorithms, probabilistic and experimental methodologies; 2007. p. 271–281.
17. Klein S, Ben-Nissan M. Accelerating Boyer Moore searches on binary texts. In: Proceedings of international conference on implementation and application of automata, CIAA-07. 2007; p. 130–143.
18. Külekci M. Filter based fast matching of long patterns by using simd instructions. In: Holub J, Zdarek J, editors. Proceedings of the Prague stringology conference. Czech Republic: Czech Technical University in Prague. 2009; p. 118–128.
19. Lecroq T. Fast exact string matching algorithms. Inf Process Lett. 2007;102(6):229–35.
20. Navarro G, Raffinot M. A bit-parallel approach to suffix automata: fast extended string matching. In: Proceedings of combinatorial pattern matching. 1998; p. 14–33.
21. Peltola H, Tarhio J. Alternative algorithms for bit-parallel string matching. In: Proceedings of the 10th international symposium on string processing and information retrieval SPIRE'03. 2003; p. 80–94.
22. Peltola H, Tarhio J. Alternative algorithms for bit-parallel string matching. 2003; p. 80–94.
23. Peltola H, Tarhio J. String matching with lookahead. Discret Appl Math. 2014;163:352–60.
24. Sunday D. A very fast substring search algorithm. Commun ACM. 1990;33(8):132–42.
25. Zavadskyi IO. Z/rz- and bricks algorithms: implementation in C programming language. https://github.com/zavadsky/stringology.
26. Zavadskyi IO. A family of exact pattern matching algorithms with multiple adjacent search windows. In: Holub J, Zdarek J, editors. Proceedings of the Prague stringology conference. Czech Republic: Czech Technical University in Prague. 2017; p. 152–166.