Fast exact pattern matching in a bitstream and 256-ary strings

Igor O. Zavadskyi

Taras Shevchenko National University of Kyiv
Kyiv, Ukraine
2d Glushkova ave.
ihorza@gmail.com

Abstract. A few known techniques of exact pattern matching, such as 2-byte read, fast loop, and sliding search windows, are improved and applied to two related sub-problems. At first, we present a new family of pattern matching algorithms, performing efficiently over 256-ary alphabets. Taking them as an underlying solution, we build the algorithms for searching a string in a bitstream. It turns out that in both cases our algorithms outperform all the other tested methods for all tested pattern lengths.

1 Introduction

Finding all occurrences of a given substring in a larger body of text is one of the most fundamental problems in computer science. In this presentation we consider an 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.

The first non-trivial bitstream pattern matching algorithm was presented in 2007 by S.T. Klein and M.K. Ben-Nissan [16]. Since then, S. Faro and T. Lecroq developed a number of improved bitstream pattern matching techniques implemented in the binary-hash and binary-skip algorithms (both presented in [8]) and the most advanced Binary-Faro-Lecroq (BFL) algorithm [7]. Also, the adaptation of the FED algorithm (Fast matching with Encoded DNA sequences) [15] to binary search has to be mentioned, although it is somewhat inferior to BFL.

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 256-ary alphabet (assuming a byte is 8 bits). For example, the algorithm [16] is based on Boyer-Moore search method, while the BFL algorithm combines a multi-pattern version of the BNDM algorithm [17] with the simplified shift strategy of Commentz-Walter algorithm [3].

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. 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

Igor O. Zavadskyi: Fast exact pattern matching in a bitstream and 256-ary strings, pp. 33–47.
Proceedings of PSC 2020, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-06749-9 © Czech Technical University in Prague, Czech Republic

matching algorithms that efficiently perform on 256-ary alphabets is a key for solving the bitstream pattern-matching problem.

According to [9] and our own experiments, the following algorithms are of the most interest when the alphabet consists of 256 characters, in different testing environments and for different pattern lengths: comparison-based Fast Search (FS, [2]), Franek-Jennings-Smyth [10] and variations of algorithms exploiting the bit-parallelism idea: Simplified BNDM with q-grams (SBNDMq, [4]), hybrid of SBNDM and Boyer-Moore-Horspool algorithms (SBNDM-BMH, [11]), Forward SBNDM (FSBNDM, [6]), SD-NDM with a "greedy" fast loop (GSBNDM, [19]) and BNDM for long patterns (LB-NDM, [18]).

As shown in [19], the performance of BNDM-type algorithms can be improved by applying the technique of *2-byte read*, which is of special interest when $|\Sigma| = 256$. 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 256-ary 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 [13] and skipping the occurrence check with a help of so called *fast loop* [14].

For a 256-ary alphabet, featuring a simple comparison-based method with these techniques can outperform the algorithms based on bit-parallelism. Indeed, the experimental results in Tables 1−2 show that the comparison-based Fast Search algorithm with 6 or 8 sliding windows performs faster than all other known methods (except for those hereinafter developed) for almost all tested pattern lengths.

In this presentation 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 comparing to "1-byte read" algorithms, while spending rather less memory than the 2-byte read approach requires. Similar ideas were discussed at Stringmasters [21], although not published yet. Then we offer a few tricks, which improve the performance of the fast loop and sliding windows techniques. As a result, we construct a family of comparison-based algorithms, which outperform all other known solutions in discovering patterns of different lengths in texts with uniformly distributed characters from a byte-based 256-ary alphabet. These algorithms are called *Z-Byte* and discussed in Section 2. A family of bitstream search algorithms, based upon Z-Byte, is constructed in Section 3. We call these algorithms *Z-Bit*. Finally, the results of algorithm benchmarking are presented in Section 4.

Let us note that an attempt to combine multiple-character reads and multiple search windows has been done in our previous work [23]. However, the methods presented hereinafter are simpler, and when applying to 256-ary alphabet, faster.

Throughout the entire presentation we use the following notations:

- Σ - alphabet of an input text and a pattern
- $|\Sigma|$ - size of the alphabet
- b - number of bits in a byte, by default $b = 8$
- k - number of significant bits in a 1.5-byte read, typically $b < k < 2b$

2 Pattern matching over an alphabet of 256 characters

In this section 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 $T[0\,..\,n-1]$ and $P[0\,..\,m-1]$ we denote a text and a pattern respectively.

2.1 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, [12]), or other data the shift depends on. BMH method is shown schematically in Alg. 1.

Algorithm 1: Boyer-Moore-Horspool algorithm

```
1 foreach c ∈ Σ do Z[c] ← m;                          // Preprocessing
2 for i ← 0 to m − 2 do Z[P[i]] ← m − 1 − i;
3 pos ← 0;                                             // Search
4 while pos ≤ n − m do
5 |   check the occurrence at pos;
6 |   pos ← pos + Z[T[pos + m − 1]];
```

Line 6 corresponds to a "bad character" shift, which maximal possible value is 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. The computationally efficient implementation of this approach is discussed in [19]: 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 random text and 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.

2.2 Double 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 [14]. 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 EBOM [6] and SBNDM [11] algorithms. In this case the probability of a maximal shift is high enough, and this implies that the exact length of a shift may not be stored in a shift table. Instead, we only determine 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, we make this constant length shift, otherwise the occurrence has to be checked, and the value of a next shift is calculated by some other algorithm. This approach allows us to avoid loading the shift length value from memory to a computer register, which is quite consuming operation.

Featuring the BMH method with the fast loop of aforementioned type and 1.5-byte reads, we get the Algorithm 2. The shift table Z contains not the shift lengths, but some "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, [20]). 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 1.5-byte reads

```
1  mask ← 2^k − 1;                                          // Preprocessing
2  foreach i ∈ [0; 2^k) do Z[i] ← 1;
3  for i ← 0 to m − 2 do
4  |  Z[word(P[i], P[i + 1])] ← 0
5  pos ← m − 2;                                             // Search
6  while pos < n do
7  |  while Z[word(T[pos], T[pos + 1])&mask] ≠ 0 do
8  |  |  pos ← pos + m − 1;
9  |  check the occurrence at pos − m + 2;
10 |  pos ← pos + QS[pos + 2];
```

We tested the Alg. 2 performance for $|\Sigma| = 256$ and different pattern lengths. The results show that this algorithm often outperform other known Boyer-Moore-based algorithms, such as FS or FJS, although the latter ones implement more sophisticated techniques to process the situation when the fast loop is terminated. However, taking into account that the 1.5-byte read implies high probability of a maximal shift, we developed other simple and efficient method to process the exit from the 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 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 make 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. 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 in line 7 is less. Note that this loop always starts from adding $m-1$ to the current position *pos* in line 4, and to compensate this addition, we subtract $m-1$ from the QS shift length in line 10 and initialize *pos* with the value -1 in line 1.

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

```
1  pos ← −1;
2  while pos < n do
3  |  repeat
4  |  |  pos ← pos + m − 1;
5  |  |  while Z[word(T[pos], T[pos + 1])&mask] ≠ 0 do
6  |  |  |  pos ← pos + m − 1;
7  |  |  pos ← pos − 1;
8  |  until Z[word(T[pos], T[pos + 1])&mask] ≠ 0 ;
9  |  check the occurrence at pos − m + 3;
10 |  pos ← pos + QS[pos + 3] − m + 2;
```

We denote the Alg. 3 by Zk-Byte, where k is the number of 'one' bits in the mask ($8 < k < 16$). If $k = 16$, lines 5 and 8 contain full 2-byte reads and variable *mask* is not needed, while in algorithm Z8 the reference to the shift table looks as $Z[T[pos]]$ and the variable *pos* can be incremented by m in lines 4 and 6. In the next subsection we discuss how to shift the search window m characters right for any value of k.

2.3 Longer shifts

Let $T[pos]$ and $T[pos+1]$ be the two last characters of a search window. If the condition $Z[word(T[pos], T[pos+1])\&mask] \neq 0$ in lines 5 and 8 of Alg. 3 is satisfied, these two bytes of a text cannot belong to the pattern together. Still, the character $T[pos+1]$ 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 5 and 8 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. 1 (a)), which makes the size of the shift table the same as for the 2-byte read. However, the situation in Fig. 1 (b) - 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.

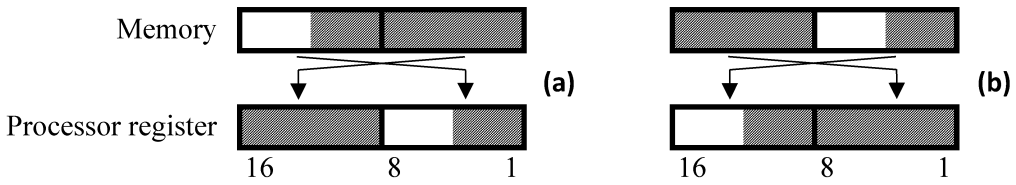

Figure 1. Loading a 2-byte value from memory on a little endian machine. 'White' bits are reset to zero by *mask*, 'grey' bits remain significant. *mask* resets the bits of the highest byte (a) or lowest byte (b).

Nonetheless, on a little endian machine the permutation shown in Fig. 1 (b) 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.

Algorithm 4: The reverse search algorithm RZk-Byte

```
 1 mask ← 2^k − 1;                                          // Preprocessing
 2 foreach i ∈ [0; 2^k) do Z[i] ← 1;
 3 for i ← 0 to m − 2 do
 4 |  Z[word(P[i], P[i + 1])] ← 0
 5 for i ← 0 to 2^(k−b) do
 6 |  Z[(i << b)|P[m − 1]] ← 0
 7 foreach c ∈ Σ do RQS[c] ← m + 1;
 8 for i ← m − 1 downto 0 do RQS[P[i]] ← i + 1;
 9 pos ← n;                                                 // Search
10 repeat
11 |  repeat
12 |  |  pos ← pos − m;
13 |  |  while Z[word(T[pos], T[pos + 1]) & mask] ≠ 0 do
14 |  |  |  pos ← pos − m;
15 |  |  pos ← pos + 1;
16 |  until Z[word(T[pos], T[pos + 1]) & mask] ≠ 0;
17 |  pos ← pos − 1;
18 |  check the occurrence at pos;
19 |  pos ← pos − RQS[T[pos − 1]] + m;
20 until pos ≥ m;
```

The text T is assumed to be prepended with a stop pattern ($T[-m \ldots -1] = P[0 \ldots m-1]$) to exit the double fast loop, given in lines 11−16, 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 first two bytes are used to determine the possibility of a maximal shift by m characters. The algorithm steps 1 character forward between the internal and external fast loops (line 15) and reverts this step in line 17 if the external fast loop is terminated. After

checking the occurrence we make the "Reverse Quick Search" shift using the shift table RQS, which is filled in lines 7−8. Lines 5−6 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 ones in Alg. 3.

2.4 Sliding windows

The performance of Z-Byte and RZ-Byte algorithms can be improved significantly by implementing a two sliding windows technique [13]. However, for reverse methods it should be slightly changed, since we can search a text from right to left only, while in 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.

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

```
1  pos1 ← ⌊n/2⌋;
2  pos2 ← n − m;
3  while pos1 ≥ 0 do
4  |   while Z[word(T[pos1], T[pos1 + 1]) & mask] ≠ 0 &
   |    Z[word(T[pos2], T[pos2 + 1]) & mask] ≠ 0 do
5  |   |   pos1 ← pos1 − m;
6  |   |   pos2 ← pos2 − m;
7  |   if Z[word(T[pos1 + 1], T[pos1 + 2]) & mask] = 0 then
8  |   |   check the occurrence at pos1;
9  |   |   pos1 ← pos1 − RQS[T[pos1 − 1]];
10 |   else
11 |   |   pos1 ← pos1 − m + 1;
12 |   if Z[word(T[pos2 + 1], T[pos2 + 2]) & mask] = 0 then
13 |   |   check the occurrence at pos2;
14 |   |   pos2 ← pos2 − RQS[T[pos2 − 1]];
15 |   else
16 |   |   pos2 ← pos2 − m + 1;
17 while pos2 > ⌊n/2⌋ do
18 |   while Z[word(T[pos2], T[pos2 + 1]) & mask] ≠ 0 do
19 |   |   pos2 ← pos2 − m;
20 |   if pos2 > ⌊n/2⌋ then
21 |   |   check the occurrence at pos2;
22 |   pos2 ← pos2 − RQS[T[pos2 − 1]];
```

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 [13]. 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.

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 "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.

3 Pattern matching in a bitstream

In this section we denote the array of full bytes of a pattern by $P[0\,..\,m-1]$, and $T[0\,..\,n-1]$ 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 $P[m+1]$. The bit length of a pattern we denote by l, and $p[0\,..\,l-1]$ is the array of pattern bits. By "search 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 RZk-Bit algorithm. 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 $T[pos], \ldots, T[pos + m-2]$, while bytes $T[pos-1]$, $T[pos+m-1]$, and possibly $T[pos+m]$, 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 $(T[pos], T[pos+1])$ does not belong to the pattern, and (b) some prefix of the pair $(T[pos], T[pos+1])$ of length greater than 8 does not coincide with the pattern suffix. The shift table Z is filled in accordance with this conditions at the preprocessing phase (Alg. 8) and checked during the search phase (lines 5 and 8 of Alg. 6). The whole block of code in lines $3-8$ of Alg. 6 implements the double fast loop discussed in Subsection 2.2.

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, $q = 0, \ldots, b-1$. This check is performed by the procedure $CheckMatch(q, pos)$ invoked in line 11 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 $p[q] \ldots p[q+b-1]$ of a pattern coincides with the byte c. This implies that all possible occurrences such that the byte $T[pos]$ is the leftmost full byte of a text that belongs to the pattern, are checked in lines 10 and 11 of Alg. 6.

After the occurrence check, *pos* is safely moved 1 byte to the left after the addition in line 12 and subtraction in line 4 on the next iteration of the main loop.

Algorithm 6: The search phase of the RZ*k*-Bit algorithm

```
1  pos ← n − 1;
2  repeat
3  |  repeat
4  |  |  pos ← pos − m + 1;
5  |  |  while Z[word(T[pos], T[pos + 1]) & mask] ≠ 0 do
6  |  |  |  pos ← pos − m + 1;
7  |  |  pos ← pos + 1;
8  |  until Z[word(T[pos], T[pos + 1]) & mask] ≠ 0;
9  |  pos ← pos − 1;
10 |  foreach q ∈ λ[T[pos]] do
11 |  |  CheckMatch(q, pos);
12 |  pos ← pos + m − 2;
13 until pos ≥ m − 1;
```

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 $T[pos-1]$ 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 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.

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

```
1  start ← j ← pos − 1;
2  while j − start < m AND
     byte((T[j] << (b − q))|(T[j + 1] >> q)) = P[j − start] do
3  |  j ← 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 · 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 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. 1 (b). However, the function $bitWord$ invoked in line 5 of Alg. 8 performs the bits permutation shown if Fig. 1 (b) 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, in a bitstream the mentioned substring may intersect with 3 adjacent bytes of a pattern, and the permutation becomes trickier. It is explained in the comments to Alg. 9.

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

```
1  mask ← 2^k − 1; m ← ⌊l/b⌋; lastMask ← byte((2^b − 1) << (b − l mod b));
2  foreach i ∈ [0; b) do c ← (P[0] << i)|(P[1] >> (b − i)) λ[c] ← λ[c] ∪ {i};
3  foreach i ∈ [0; 2^k) do Z[i] ← 1;
4  foreach i ∈ [0; l − b) do
5  |  t ← bitWord(i, mask);
6  |  if l − i ≥ 2b then
7  |  |  Z[t] ← 0;
8  |  else
9  |  |  if l − i > 3b − k then s ← 2b − (l − i);          // 2 (a)
10 |  |  else s ← k − b;                                   // 2 (b)
11 |  |  foreach j ∈ [0; 2^s) do Z[t|(j << b)] ← 0;
```

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. 2 (a). The length s of this "hole" is calculated in lines 9 (Fig. 2 (a)) or 10 (Fig. 2 (b)). 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.

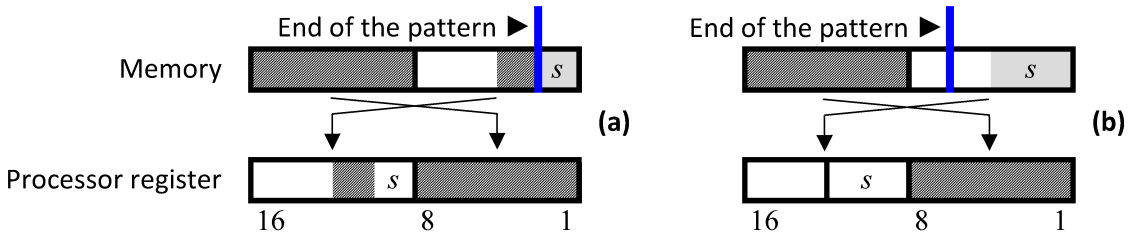

Figure 2. The *bitWord* permutation (Alg. 9) of the right tail of a pattern. The bits, remaining significant after the permutation, highlighted with dark grey. The bits that would have remained significant if not for the end of a pattern, highlighted with light grey. (a) The pattern ends within significant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

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

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

4 Benchmarking experiments

The experimental algorithms execution times are shown in Table 1 (256-ary search) and Table 2 (bitstream search). The algorithms were implemented in C programming language and compiled with GNU GCC compiler. The C codes of some Z-algorithms are published in [22], while most of other algorithms have been taken from [5].

4.1 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 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 R/RZ algorithms may be increased in above-mentioned proportions. However, even after this adjustment Z/ZR algorithms still remain the fastest ones for all pattern lengths.

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 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 ($8 < k < 16$) and RZ-Bit algorithms, the short patterns are considered as a special case and processed with a single fast loop.

4.2 Testing environment

The executables have been run on 40 computers with different processors and L1 cache varying from 24KB up to 64KB; all computers were little endian machines. The experimental 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 Fibonacci codes, (s, c)-dense codes and other codes with delimiters. This assigns a special meaning to a pattern matching problem in application to delimiter-encoded data. Also, the algorithms have been tested on a random 10MB text, however, the results are not shown in this presentation, since they differ from the timing given below insignificantly and do not change the general picture of algorithms superiority.

The results, given in milliseconds, represent a weighted average time of 500×40 runs of each algorithm searching patterns of length $2 - 512$ randomly taken from the text, a new pattern for each run. In a bitstream search the pattern starts from a random bit. Weight numbers has been taken on pro rata basis of average computer benchmark. In other words, the below presented time values have been calculated as follows.

1. Get the average time $T_{a,m,c}$ of 500 runs of each algorithm a for each pattern length m on each computer c.
2. Multiply $T_{a,m,c}$ by $total_time/40c_time$, where $total_time$ is the sum of all values $T_{a,m,c}$ for all computers / algorithms / pattern lengths, c_time is the sum of all values $T_{a,m,c}$ for a specific computer c, and 40 is the number of computers.
3. For each pair (a, m), get the average value of $T_{a,m,c}$ with respect to all computers.

This approach allows us to interpret all computers as units of equal importance independently of their actual benchmarks.

4.3 Experimental results

Generally, 44 different variations of Z-Byte and RZ-Byte algorithms were tested, with $1 - 6$ sliding windows and the parameter k varying in the range $12 - 15$. Also, the algorithms Z8 and Z16 based on full byte reads have been tested in 1, 2 and 3 sliding windows versions. It appears that when $m \geq 8$, the 13-bit or 14-bit reads are most efficient. That's why only the results for 13 and 14-bit versions of Z/RZ-Byte algorithms with 1.5 read are shown in Table 1 and only for those numbers of sliding windows, which give the optimal result for at least one pattern length. Let us note that when a pattern becomes longer, the efficiency of 14-bit reads in comparison to 13-bit reads increases. And when the pattern length is very short, $m \leq 4$, the 1.5-byte read appears to be superfluous and the algorithm Z8-w2, supporting 1-byte read and 2 sliding windows, demonstrates the best performance.

A number of known algorithms were tested for comparison, both in original and "2-byte read" versions, if applicable. As it is seen, a 2-byte read does not improve an algorithm's performance for $|\Sigma| = 256$. At the same time, different representatives of Z/RZ-families outperform all other known algorithms on all examined pattern lengths. This testifies that our "1.5-byte read" approach appears to be more efficient. Indeed, if the pattern is not extremely long, the probability of a maximal shift based on analysing 2 byte suffix of a search window is not much higher than if we analyse $13-14$ bits. However, in the latter case the shift table fits into L1 cache, which is rather more important factor. This is also confirmed by the performance of Z16-Byte-w2, the fastest algorithm among Z-algorithms with 2-byte read, which always under-performs some 1.5-read algorithms.

Among other algorithms the comparison-based Fast Search algorithms with 6 or 8 sliding windows demonstrate the best results for all pattern lengths. The ratio of running times of the best algorithm not belonging to Z/RZ-families to the winner is

shown in the last row of Table 1 for each pattern length. It is significant for short ($m \leq 8$) and long ($m \geq 128$) patterns, however, for middle-sized ones the Z/RZ timing is only slightly less than that of the FS with 6 or 8 sliding windows.

Thus, we can conclude that the bit-parallelism approach to pattern matching is not too efficient on a 256-ary alphabet. The main reason for that is that comparison-based algorithms provide quite good average shift length having at the same time less operational complexity. Instead of implementing the bit-parallelism, the better way for improvement consists in (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 2. As seen for all investigated pattern lengths, the RZ-Bit algorithms, based on the principles (a)-(c), outperform the BFL method, based upon the bit-parallel algorithm BNDM. And this outperformance is essential even for no-sliding windows versions of RZ-Bit. Apart from the above mentioned reasons, the superiority of the RZ-Bit family may be explained by the following factors: (d) although the

Pattern length m	2	4	8	16	32	64	128	256	512
Z8-Byte-w2	**40.63**	**26.22**	19.64	16.10	14.58	14.02	12.52	13.16	12.35
RZ13-Byte	67.50	35.70	21.21	14.91	11.80	10.67	8.82	6.42	4.51
RZ14-Byte	67.92	35.80	21.16	14.94	11.85	10.77	8.92	6.40	4.40
RZ13-Byte-w2	49.34	26.29	15.90	12.05	10.36	9.66	8.33	5.85	3.50
RZ14-Byte-w2	49.59	26.48	15.97	12.06	10.47	9.85	8.41	5.88	**3.41**
RZ13-Byte-w3	49.45	26.42	**15.67**	11.17	9.59	9.00	7.93	5.65	3.86
RZ14-Byte-w3	49.78	26.70	15.91	11.26	9.74	9.15	8.02	5.65	3.70
RZ13-Byte-w5	50.89	27.18	16.04	**11.08**	9.50	9.03	8.09	5.80	4.32
RZ14-Byte-w5	51.38	27.50	16.31	11.27	9.70	9.14	8.19	5.73	4.06
Z13-Byte-w3	94.31	30.75	15.95	11.20	**9.44**	**8.92**	**7.90**	**5.60**	3.83
Z14-Byte-w3	99.79	31.04	16.21	11.28	9.60	9.09	8.03	5.62	3.71
RZ16-Byte-w2	49.87	28.34	17.94	14.03	10.86	9.91	8.29	6.60	4.43
FSBNDM	69.76	35.97	20.29	13.63	11.61	11.67	12.04	12.21	12.26
FSBNDM-2byte	90.33	50.59	30.55	20.43	16.47	16.58	17.02	17.23	17.13
FSBNDM31	224.37	75.28	34.50	18.01	12.81	12.88	13.13	13.26	13.24
FSBNDM31-2byte	215.67	79.00	40.54	23.91	17.54	17.68	18.10	18.32	18.21
GSBNDMq2	–	43.05	20.84	13.53	11.45	11.53	11.70	11.82	11.82
GSBNDMq2-2byte	–	54.03	29.15	19.32	15.67	15.75	16.06	16.30	16.19
SBNDMq2	130.04	44.40	21.44	13.83	11.50	11.56	11.69	11.82	11.82
SBNDMq2-2byte	199.07	72.79	37.28	23.01	16.89	16.98	17.32	17.55	17.43
SBNDM/BMH	152.10	78.45	43.68	25.89	19.86	19.97	20.38	20.68	20.51
LBNDM	118.87	64.43	38.63	25.03	18.93	15.80	13.83	9.88	7.33
FJS	201.94	121.84	70.95	39.17	23.03	17.99	16.48	19.16	19.46
FS	259.70	130.52	68.52	36.15	21.24	17.33	16.53	22.84	21.14
FSw4	77.47	39.73	21.91	13.01	9.96	9.64	9.23	10.22	8.80
FSw6	62.46	32.35	18.23	11.44	9.65	9.47	8.97	8.97	7.56
FSw8	57.15	30.93	19.11	12.65	9.62	9.46	8.88	8.72	7.07
Best non-Z to best Z ratio	1.41	1.18	1.16	1.03	1.02	1.06	1.12	1.56	2.07

Table 1. Algorithms running times, $|\Sigma| = 256$ (milliseconds)

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 algorithms.

Pattern length	20	40	60	80	100	200	300	400	500
RZ13-Bit	157.65	48.28	34.83	26.03	22.22	15.27	13.17	12.52	12.04
RZ14-Bit	148.76	44.52	32.22	23.74	20.37	13.90	12.14	11.49	11.15
RZ15-Bit	145.22	42.73	31.23	23.10	19.86	13.74	12.12	11.51	11.13
RZ13-Bit-w2	129.49	35.91	26.81	20.88	18.25	13.89	12.29	11.84	11.54
RZ14-Bit-w2	120.65	33.24	24.81	19.07	**16.90**	13.08	11.78	11.35	11.10
RZ15-Bit-w2	116.87	32.57	24.50	18.96	17.03	13.15	11.94	11.53	11.31
RZ13-Bit-w3	135.61	38.50	28.40	21.96	19.09	13.90	12.25	11.82	11.51
RZ14-Bit-w3	126.74	35.19	26.08	19.79	17.25	12.74	11.43	11.03	10.77
RZ15-Bit-w3	122.25	34.03	25.47	19.44	17.10	**12.72**	**11.41**	**11.03**	**10.77**
RZ16-Bit-w2	**102.02**	**30.56**	**23.76**	**18.91**	17.05	13.35	12.26	11.95	11.66
BFL	277.77	91.46	63.80	43.23	36.63	23.38	21.65	25.29	31.32
FED	397.07	130.04	86.42	52.13	46.26	31.14	28.40	27.97	28.95
Best non-Z to best Z ratio	2.72	2.99	2.68	2.28	2.16	1.83	1.89	2.29	2.68

Table 2. Running times on a bitstream (milliseconds)

5 Conclusions

An efficient approach to pattern matching in a bitstream has been investigated and tested as well as underlying algorithms of string matching in 256-ary texts. It relies on improving the 2-byte read principle as well as tuning the fast loop and sliding windows techniques. The averaged results of testing provided on 40 computers show that different representatives of the developed families of algorithms outperform all other tested solutions for all studied pattern lengths.

References

1. A. V. Anisimov and I. O. Zavadskyi: *Variable-length prefix codes with multiple delimiters.* IEEE Transactions on Information Theory, 63(5) 2017, p. 2885–2895.
2. D. Cantone and S. Faro: *Fast-search algorithms: New efficient variants of the Boyer-Moore pattern-matching algorithm.* Journal of Automata, Languages and Combinatorics, 10(5/6) 2005, pp. 589–608.
3. B. Commentz-Walter: *A string matching algorithm fast on the average*, in Proceedings of International Colloquium on Automata, Languages, and Programming, 1979, pp. 118–132.
4. B. Durian, J. Holub, H. Peltola, and J. Tarhio: *Tuning BNDM with q-grams*, in Proceedings of the Workshop on Algorithm Engineering and Experiments, I. Finocchi and J. Hershberger, Eds. SIAM, New York, New York, USA, 2009, pp. 29–37.
5. S. Faro and T. Lecroq: *String matching research tool: Exact string matching algorithms.* https://www.dmi.unict.it/ faro/smart/algorithms.php.
6. S. Faro and T. Lecroq: *Efficient variants of the backward-oracle-matching algorithm*, in Proceedings of the Prague Stringology Conference, J. Holub and J. Zdarek, Eds. Czech Technical University in Prague, Czech Republic, 2008, pp. 146–160.

7. S. Faro and T. Lecroq: *An efficient matching algorithm for encoded DNA sequences and binary strings*, in Proceedings of Combinatorial Pattern Matching, G. Kucherov and E. Ukkonen, Eds., 2009, p. 106–115.
8. S. Faro and T. Lecroq: *Efficient pattern matching on binary strings*, in 35th International Conference on Current Trends in Theory and Practice of Computer Science, 2009, p. Poster.
9. S. Faro and T. Lecroq: *The exact online string matching problem: a review of the most recent results.* ACM Computing Surveys (CSUR), 45(2) 2013, p. article 13.
10. F. Franek, C. Jennings, and W. Smyth: *A simple fast hybrid pattern-matching algorithm.* J. Discret. Algorithms, 5(4) 2007, pp. 682–695.
11. J. Holub and B. Durian: *Fast variants of bit parallel approach to suffix automata.* Paper presented at the Second Haifa Annual International Stringology Research Workshop of the Israeli Science Foundation, 2005.
12. N. R. Horspool: *Practical fast searching in strings.* Soft.Pract.Exp., 10(6) 1980, pp. 501–506.
13. A. Hudaib, R. Al-khalid, D. Suleiman, M. A. A. Itriq, and A. Al-Anani: *A fast pattern matching algorithm with two sliding windows (tsw).* Journal of Computer Science, 4(5), p. 393–401.
14. A. Hume and D. Sunday: *Fast string searching.* Softw. Pract. Exp., 21(11) 1991, pp. 1221–1248.
15. J. Kim, E. Kim, and K. Park: *Fast matching method for DNA sequences*, in Proceedings of Combinatorics, Algorithms, Probabilistic and Experimental Methodologies, 2007, p. 271–281.
16. S. Klein and M. Ben-Nissan: *Accelerating Boyer Moore searches on binary texts*, in Proceedings of International Conference on Implementation and Application of Automata, CIAA-07, 2007, p. 130–143.
17. G. Navarro and M. Raffinot: *A bit-parallel approach to suffix automata: Fast extended string matching*, in Proceedings of Combinatorial Pattern Matching, 1998, pp. 14–33.
18. H. Peltola and J. Tarhio: *Alternative algorithms for bit-parallel string matching*, in Proceedings of the 10th International Symposium on String Processing and Information Retrieval SPIRE'03, 2003, pp. 80–94.
19. H. Peltola and J. Tarhio: *String matching with lookahead.* Discrete Applied Mathematics, 163 2014, pp. 352–360.
20. D. Sunday: *A very fast substring search algorithm.* Comm. ACM, 33(8) 1990, pp. 132–142.
21. B. W. Watson, F. Franek, J. Holub, C. Iliopoulos, and B. Smyth: *Fractional n-grams for shifting.* Idea presented at StringMasters 2014 at McMaster University, Hamilton, Ontario, Canada, 2014.
22. I. O. Zavadskyi: *The Z-family algorithms: Implementation in C programming language.* https://github.com/zavadsky/stringology.
23. I. O. Zavadskyi: *A family of exact pattern matching algorithms with multiple adjacent search windows*, in Proceedings of the Prague Stringology Conference, J. Holub and J. Zdarek, Eds. Czech Technical University in Prague, Czech Republic, 2017, p. 152–166.