A Family of Exact Pattern Matching Algorithms with Multiple Adjacent Search Windows

Igor O. Zavadskyi

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

Abstract. A new family of comparison-based exact pattern matching algorithms is presented. They utilize the multi-dimensional arrays in order to process more than one adjacent search window in each iteration of the search loop. This approach leads to a lower average computing time by the cost of space. However, the excessive space consumption can be avoided due to a special technique of replacing a multi-dimensional array with a series of one-dimensional arrays of pointers. The algorithms of this family perform well for short or middle-size patterns, when the shift of a search window by several lengths at once is quite probable. Our algorithms outperform all other known algorithms for some values of pattern length on English text, genomic sequence and a random text over an alphabet of size 8 or 32.

1 Introduction

Pattern matching is one of the most fundamental techniques in computer science. The most common pattern matching problem is formulated as finding all the exact occurrences of a given substring in a larger body of text. Through the entire presentation we use the following notation:

- $T[0\,..\,n-1]$ - input text
- $P[0\,..\,m-1]$ - pattern to be searched
- n - length of the input text
- m - length of the pattern
- Σ - alphabet of the input text and the pattern
- $|\Sigma|$ - size of the alphabet
- $|\Sigma_P|$ - number of different symbols in the pattern

It is worthwhile to compare the pattern matching algorithm efficiency on the $(|\Sigma|, m)$-plane. Our research concerns mostly its middle left area, where $m \leq 64$ and $4 \leq |\Sigma| \leq 32$. In this area the modifications of the Boyer-Moore algorithm (BM) [2], e.g. the Boyer-Moore-Horspool algorithm (BMH) [12], Sunday's "Quick Search" (QS) [17] or Tuned Boye-Moore (TBM) [14] were considered the best for a long time. However, a number of more efficient exact pattern matching algorithms were invented after 2000. According to experiment results on a random text presented in [9] the most successful algorithms are TVSBS [18] for $m = 2$, $8 \leq |\Sigma| \leq 32$, EBOM [6] for $4 \leq m \leq 16$, $8 \leq |\Sigma| \leq 32$ or $m = 4$, $\Sigma = 4$ and also HASHq [15], SBNDMq [4] and FSBNDM [6] in the right subarea of $m \leq 64$, $4 \leq \Sigma \leq 32$ rectangle. The Shift-And algorithm [20] proposed in 1992 remains optimal for $m = 2$, $\Sigma = 4$. The mentioned algorithms cover all three known approaches to pattern-matching: TVSBS and HASHq are comparison-based; EBOM is automata based, while Shift-And, SBNDMq and FSBNDM algorithms use the bit-parallel operations. We develop

Igor O. Zavadskyi: A Family of Exact Pattern Matching Algorithms with Multiple Adjacent Search Windows, pp. 152–166.
Proceedings of PSC 2017, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-06193-0 © Czech Technical University in Prague, Czech Republic

a new comparison-based algorithm family. Almost all comparison-based algorithms, including our new ones, exploit the idea of bad-character shift, which originates from the BM search. It is to shift the search window to align the character or characters around its end to their most right occurrence in the pattern. The BMH algorithm is based on this idea only. We develop a generalization of the BMH algorithm, which allows to perform several bad-character shifts in each iteration of the search loop.

Let us discuss the search loop of the BMH (Algorithm 1). The bad character shift is performed in the line 8 and its length is equal to $D[T[pos + m - 1]]$, where pos is the starting position of the search window and D is the shift array defined by

$D(c) = \min(\{1 \le k < m | P[m - 1 - k] = c\} \cup \{m\})$.

If the ratio $|\Sigma_P|/|\Sigma|$ is small enough, the character $T[pos + m - 1]$ most likely does not occur in the pattern and the length of the shift is maximum, i.e. it stands m. These maximum length shifts are the main factor responsible for the efficiency of BMH in the left up area of the $(|\Sigma|, m)$-plane. And if the ratio $|\Sigma_P|/|\Sigma|$ is particularly small, one can assume that probably not only the character $T[pos + m - 1]$ does not belong to the pattern, but the characters $T[pos + 2m - 1]$, $T[pos + 3m - 1]$ etc. as well. This means that the search window can be shifted by several lengths at once, or, in other words, several adjacent search windows can be processed in the same iteration of the search loop. This is the main idea of the multiple adjacent window search algorithms (MAW), as well as of the Tuned Boyer-Moore algorithm.

Of course, at least k characters of the input text must be read and processed in each substring of the length km in order not to miss the possible pattern match. Thus, at least k readings of text characters should be done for each substring of the length km – just the same number as in k iterations of the single-window algorithm like BMH or QS. However, we can reduce the number of other operations. For this we use the k-dimensional array, unlike the TBM algorithm, where the search loop is unrolled. Such array occupies rather more memory than the shift table in a single-window search algorithm and its filling takes more preprocessing time. Nevertheless, these space overheads are not that big comparing to memory size of modern computers, while time overheads are more than covered in the main search loop.

Algorithm 1: The search phase of the Boyer-Moore-Horspool algorithm

```
1 pos ← 0 ;
2 while pos ≤ n − m do
3     j ← 0;
4     while T[pos + j] = P[j]  AND j < m do
5         j ← j + 1;
6     if j = m then
7         output pos;
8     pos ← pos + D[T[pos + m − 1]];
```

The idea of using two or more search windows is not new. Apart from the mentioned Tuned Boyer-Moore algorithm, it was implemented in Two- and Four-Sliding-Windows algorithms [13], variants of Backward-SNR-DAWG-Matching [7] for multiple windows [5]. Also, this idea was applied to different algorithms in [8]. However, search windows in these algorithms are not adjacent. Thus, they are well suited for parallel processing or multi-pattern search, but do not take the advantage of multi-dimensional search array. Though the idea of two-dimensional search array was also implemented in a number of algorithms, for instance, in the Berry-Ravindran algorithm [1], TVSBS and EBOM, in most of them it was proposed to use the consequent

characters of a text as indices. This significantly increases the probability of the maximum length shift if it is low for a single-character check but otherwise leads to superfluous density of the checks. In other words, if even single-character check causes the maximum shift with high probability, likely, there is no need to check two adjacent characters to shift the search window by m or $m+1$ positions. In this case it may be better to perform some special fast check of the characters $T[pos]$ and $T[pos+m]$ in the same iteration, which could shift the search window by $2m$ positions. These considerations lead to MAW2 algorithm. In general, we denote by MAWq the MAW algorithm based on the processing of q adjacent windows.

The attempt to speed up a search using two adjacent search windows was made in [19] (QLQS algorithm). Authors use two one-dimensional search tables ("forward" and "backward"). This allows to increase the average shift length against the single-character algorithms like QS, but, unlike the MAW2, does not guarantee that we make the maximum possible safe shift based on "couple of characters" heuristic. Our experiments show that the QLQS performs slower than the MAW2 (tables 4-6).

The two-dimensional search array was combined with the adjacent search windows in [3]. The so-called jumping-occurrence heuristic allows to perform the shift by checking two characters in an adjustable distance. When this distance is maximum, i.e. $m+1$, this solution called JOM becomes quite similar to MAW2 and even provide longer shifts in average due to forward character checks. However, this "forward" logic requires the pattern occurrence check in each iteration of JOM, while in the MAW2 this check is performed only under certain condition, which is satisfied infrequently. Besides that, accessing the array element in the MAW2 requires fewer additions. All this makes the MAW2 algorithm faster than JOM, as experiments show.

Nevertheless, the assumption that not only the character $T[pos]$ does not belong to the pattern, but the character $T[pos+m]$ as well, is quite strong. As experiments show, the MAW2 algorithm based on this assumption outperforms all the other only when the pattern is very short (3–6) and alphabet size is around 32 (table 6). If the pattern is longer or alphabet smaller, the adjacent characters check is efficient. In this case we could check the character bigrams in the right of the adjacent windows. For example, using the four-dimensional search table to check the characters $T[pos]$, $T[pos+1]$, $T[pos+m]$ and $T[pos+m+1]$, we obtain the MAW22 algorithm (2 adjacent search windows and 2 adjacent characters to check in each).

The similar combination of characters in shift heuristic is checked in the SBNDM algorithm with the "greedy" skip-loop (GSB) [16]. However, since it is based on one-dimensional search tables only, either the maximum or the average shift is shorter than in MAW22. Also, the operational complexity of the "skip" iteration of Greedy FSBNDM algorithm is higher (table 2). As a result, the GSB algorithm is essentially slower than the MAW22 on short alphabets ($|\Sigma| = 4$, table 3, or $|\Sigma| = 8$, table 5), where the MAW22 algorithm appears to be the most efficient.

However, for alphabets containing 25–30 or more characters, the 4-dimensional search table becomes too large and its processing slows down due to caching or other memory access issues. For this reason we investigate how to utilize the pointers in order to reduce the size of the search tables significantly, while increase the access time only a little. The resultant algorithms are called "MAW with pointers" (MAWP).

This paper is organized as follows. In section 2 we optimize the BMH search loop to construct an algorithm that checks single characters in two adjacent windows, it is MAW2. We discuss the types of search window shifts in MAW2 and compare the BMH and MAW2 complexity at operational level. In section 3 we generalize the

MAW2 algorithm with checking the bigrams of characters in the adjacent search windows. This is the MAW22 algorithm. In section 4 we discuss how to reduce the size of multi-dimensional search tables. In section 5 we describe the generalization of the MAW2 and MAW22 to the case of more than 2 adjacent search windows. Thus we obtain the MAWq and MAWq2 algorithm families. In section 6 the preprocessing stage of the MAWq and MAWq2 algorithms is discussed. In section 7 we present the experimental results and make the final conclusions in section 8.

2 The MAW2 algorithm with two adjacent search windows

In this section we discuss how to process two adjacent search windows of the length m, which could be considered as one window of double length $2m$. We try to reduce the total number of computing operations required to process the substring of the length $2m$. Let us examine the search loop of BMH (Algorithm 1). Two reads from the shift table D in the line 8 in two iterations can be replaced by one read from the two-dimensional shift table $M2_{|\Sigma|\times|\Sigma|}$ defined as follows: $M2[i][j]$ is the leftmost possible position of the first character of a pattern under the assumption that $T[m-1] = i$ and $T[2m-1] = j$. All shifts defined by the table $M2$ can be divided into 4 types shown in Figure 1.

- (a) Neither i nor j belongs to the pattern P. Then P can be safely shifted by $2m$ positions forward.
- (b) Character i doesn't belong to the pattern P, but j does. In this case P can be safely shifted by more than $m-1$ symbols but less than $2m$. Namely, the rightmost occurrence of j in P should be aligned with $T[2m-1]$.
- (c) Character i belongs to P and $P[m-1] \neq i$. Then P can be safely shifted forward by less than m symbols. Namely, the rightmost occurrence of i in P should be aligned with $T[m-1]$.
- (d) $P[m-1] = i$. Then the pattern can be matched at the current position. One should check if $T[0] \dots T[m-2]$ coincides with the pattern before proceed forward.

The search loop of the MAW2 is shown in Algorithm 2. The text is assumed to be appended by the "stop" pattern. Note that the condition in line 4 is met only in the case (d), otherwise only lines 2, 3, 4 and 12 are executed.

Algorithm 2: The search phase of the Two Adjacent Windows algorithm MAW2

```
1  pos ← m − 1 ;
2  while pos < n do
3      r ← M2[T[pos]][T[pos + m]] ;
4      if r = 0 then
5          j ← 0 ;
6          while j < m − 1 AND T[pos − (m − 1) + j] = P[j] do
7              j ← j + 1 ;
8          if j = m − 1 then
9              output pos − (m − 1) ;
10         pos ← pos + D[T[pos]] ;
11     else
12         pos ← pos + r ;
```

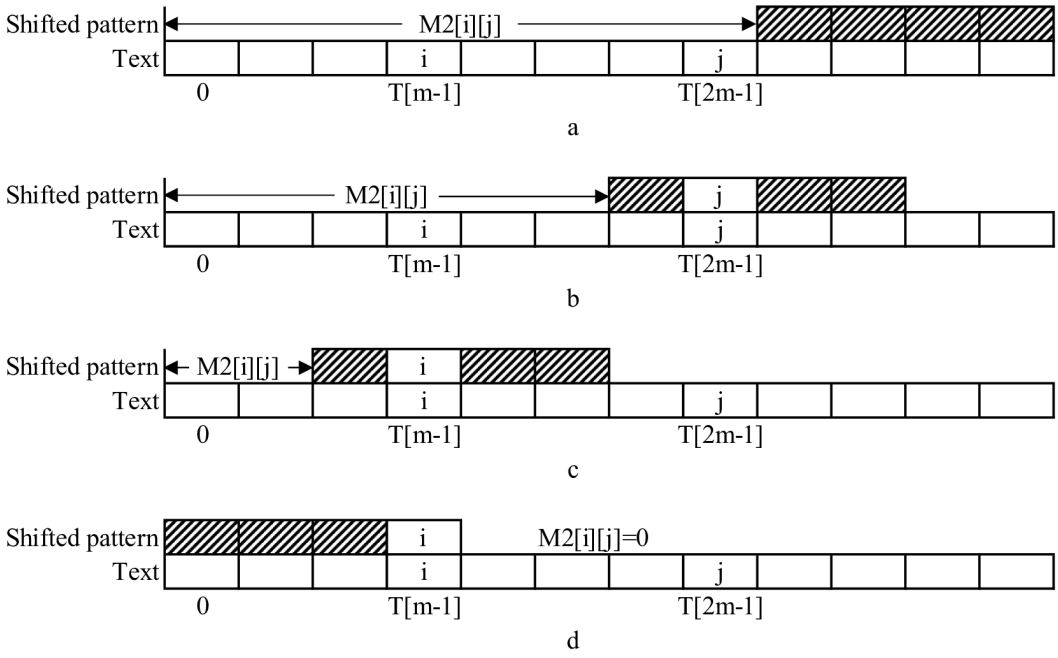

Figure 1. Pattern shifts in the MAW2 algorithm

Let us calculate the number of operations in the BMH and MAW2 algorithms required to shift the search window by $2m$ characters forward (the case of the maximum possible shift). This is the most probable case when the pattern length is small compared to alphabet size. In this case only lines 2, 3, 4, 6 and 8 in two iterations of the BMH algorithm and only lines 2, 3, 4 and 12 in some iteration of the MAW2 algorithm are executed. Note that getting an element of a one-dimensional array like $D[x]$ is equivalent to $*(D + x)$ in C notation, which requires one addition and two readings from memory, while getting an element of a two-dimensional $|\Sigma| \times |\Sigma|$ array like $M2[x][y]$ is equivalent to $*(M2 + |\Sigma| * x + y)$, which requires two additions, one multiplication and three readings from memory ($|\Sigma|$ is a constant).

The calculations are shown in Table 1. The number of operations in the MAW2 algorithm is more than twice less compared to BMH. The subtractions in expressions $m - 1$, $n - m$, $z - 1$ are not counted, since these values can be calculated in the preprocessing stage. Also, the comparison $j < m$ is not counted in the AND conjunction in BMH, since it is not actual in the case of the maximum shift. Let us note that multiplication is no longer time consuming on modern computers and exceeds the time of other operation by 20-30% at most.

One can observe that one iteration of the MAW2 search loop requires fewer operations even than one iteration of the BMH search loop in the case when the condition $r = 0$ is not met in the line 4 of Algorithm 2 (Figure 1a–c). Therefore, in the case shown in Figure 1c the MAW2 algorithm search loop still executes faster than the BMH search loop, while the equality $M2[i][j] = D[i]$ holds, i.e. the shift length in the MAW2 algorithm is just the same as in the BMH.

Of course, the advantage of the MAW2 search loop over the BMH search loop in the case (b) is lower than in the case (a) and in the case (c) is lower than in the case (b). While the ratio $|\Sigma_P|/|\Sigma|$ increases, the balance between the cases (a), (b) and (c) moves to (b) and (c) and then to (c) only. If $|\Sigma_P|/|\Sigma|$ is close to 1, the case (c)

Table 1. The operational complexity of the BMH and MAW2 algorithms

Operation	BMH	MAW2
Comparisons	6 = (lines 2, 4 and 6)×2	2 = lines 2 and 4
Assignments	4 = (lines 3 and 8)×2	2 = lines 3 and 12
Memory reads	28 = (2 in line 2; 1 in line 3; 5 in line 4; 1 in line 6; 5 in line 8)×2	10 = 2 in line 2; 6 in line 3; 2 in line 12
Additions	14 = (3 in line 4; 4 in line 8)×2	6 = 5 in line 3; 1 in line 12
Multiplications	–	1 in line 3
Total	52	21

occurs almost always and outperformance of the MAW2 search loop over the BMH search loop is small. The case shown in Figure 1d occurs with the probability $1/|\Sigma|$ regardless of $|\Sigma_P|$ value, for random text. This is when the internal loop of the MAW2 in the lines 6 and 7 of Algorithm 2 is executed and has the same number of iterations as the internal loop of BMH. However, each iteration of the MAW2 internal loop requires one operation more than that one of the BMH.

Thus, the search loop of the MAW2 algorithm is essentially faster than the search loop of the BMH algorithm when the following conditions are met: (1) the alphabet is large enough to make the case (d) not frequent; (2) the ratio $|\Sigma_P|/|\Sigma|$ is small enough to make the case (c) not frequent. In fact, any alphabet of size 8 and bigger could be considered as "large enough" to make the case (d) not frequent. The violation of condition (2) forces the MAW2 algorithm search loop to run only a bit faster than the BMH search loop. However, for the wide range of pattern length / alphabet size combinations the MAW2 outperforms the BMH essentially.

3 The bigram extension

As mentioned above, the MAW2 algorithm exploits the very classical approach consisting in checking single characters, i.e. characters that are far apart in the text. Although this idea could give some advantage when the ratio $|\Sigma_P|/|\Sigma|$ is small, a number of algorithms invented since 1990s show that checking two or more adjacent characters (q-grams) is more efficient in general case. Namely, such checks are performed in the EBOM, FSBNDM, Hashq and other algorithms, which are considered as the fastest ones in some areas of $(|\Sigma|, m)$-plane. Let us note that the MAW technique can be applied also to q-gram checks. In this case the fundamental assumption is that probably not only the pair of characters $(T[pos], T[pos+1])$ does not belong to the pattern, but the pair $(T[pos+m], T[pos+m+1])$ as well. Of course, this assumption is realistic for much wider range of values $(|\Sigma|, m)$ than that one for the pair of single characters. And it leads to the MAW22 algorithm (2 adjacent search

windows with 2 bigram checks in each). Its search loop is shown in Algorithm 3. It is assumed that the text is appended by the "stop" pattern.

Algorithm 3: The search phase of the MAW22 algorithm

```
1  pos ← m − 2;
2  while true do
3      r ← M22[T[pos]][T[pos + 1]][T[pos + m]][T[pos + m + 1]] ;
4      if r = 0 then
5          j ← 0;
6          while T[pos − (m − 2) + j] = P[j] AND j < m − 2 do
7              j ← j + 1;
8          if j = m − 2 then
9              output pos − (m − 2);
10             if pos ≥ n then
11                 break;
12             pos ← pos + D[T[pos]] ;
13     else
14         pos ← pos + r;
```

In Algorithm 3 the single bad character shift table is denoted by D, just as in the MAW2 or BHM algorithm, while the bigram shift table $M22$ is organized as follows. $M22[i][j][k][t]$ is the leftmost possible position of the first character of the pattern under the assumption that $T[m-2] = i$, $T[m-1] = j$, $T[2m-2] = k$, $T[2m-1] = t$.

Evidently, the MAW22 algorithm could be successful thanks to shifts that are essentially longer than m. Otherwise a search window shift is too short to compensate such expensive operation as accessing the 4-dimensional array element. As experiments show (Tables 3 and 5), the probability of "good" shifts is high enough to make the MAW22 algorithm the fastest one for some pattern lengths when $4 \leq |\Sigma| \leq 8$. Accordingly to [9], in this area of $(|\Sigma|, m)$-plane the best results belonged to the EBOM (foremost), Hashq and variations of the SBNDM algorithm. Among the SBNDM family the GSB algorithm is of the most interest, since it exploits the same idea of two bigram checks in the skip loop, as the MAW22 (Algorithm 5). The EBOM algorithm also contains the special fast skip loop shown in Algorithm 4 (we present the fast practical implementation of the EBOM taken from [10]). In two iterations of the EBOM skip loop or one iteration of the GSB skip loop the search window can be shifted by $2m - 2$ characters at most.

The number of basic operations in two iterations of the EBOM and one iteration of the MAW22 and GSB in the case of the maximum shift is shown in Table 2. Only the lines 2, 3, 4 and 14 of Algorithm 3 are executed. As is seen, the MAW22 algorithm performs only one operation less than EBOM and 4 operations less than GSB. However, the maximum shift in the MAW22 is 2 characters longer than the maximum shift in the GSB or double maximum shift in the EBOM. This is quite noticeable difference for short patterns. This maximum shift by $2m$ positions is achieved in the MAW22 when (1) neither the pair (i, j) nor the pair (k, t) belongs to the pattern, (2) t is not the first character of the pattern and (3) j and k are not the last and the

Table 2. The operational complexity of MAW22, GSB and EBOM algorithms

Operation	EBOM	GSB	MAW22
Comparisons	2=(1 in line 1)×2	2 (line 1)	1 in line 4
Assignments	2=(1 in line 2)×2	2 (lines 1 and 2)	1 in line 3
Memory reads	14 = (5 in line 1; 2 in line 2)×2	16 = 14 in line 1; 2 in line 2	14 = 11 in line 3; 1 in line 4; 2 in line 14
Additions	12 = (5 in line 1; 1 in line 2)×2	12 = 11 in line 1; 1 in line 2	12 = 11 in line 3; 1 in line 14
Multiplications	2=(1 in line 1)×2	–	3 in line 3
Bitwise shift	–	2 in line 1	–
Logical AND	–	1 in line 1	–
Total	32	35	31

first characters of the pattern respectively. If only conditions (1) and (3) are met, the length of the shift is equal to $2m - 1$.

Algorithm 4: The skip search loop of the EBOM algorithm

```
1 while FT[T[pos]][T[pos − 1]] = θ do
2     pos ← pos + m − 1;
```

Algorithm 5: The "greedy" skip loop of the GSB algorithm

```
1 while (D ← ((B[T[i + 1]] << 1)&B[T[i]])) = 0 AND
    ((B[T[i + m]] << 1)&B[T[i + m − 1]]) = 0 do
2     i ← i + 2m − 2;
```

In the case of a non-maximum shift the MAW22 algorithm most often gives the shift length at once, i.e. in the line 3 of Algorithm 3, which requires just the same number of operations as in the case of a maximum shift. At the same time any non-maximum shift in EBOM requires at least 2 extra readings from two-dimensional arrays as well as a non-maximum shift in GSB.

The computational experiments (Section 7) show that the MAW22 algorithm strongly outperforms the MAW2 for small alphabets ($|\Sigma| \leq 8$). Moreover, the MAW22 outperforms all the other known algorithms on short patterns in genomic sequences ($3 \leq m \leq 11$, $|\Sigma| = 4$) and on short and medium-size patterns ($3 \leq m \leq 72$) when $|\Sigma| = 8$. The MAW22 remains more efficient than the MAW2 on a random alphabet of size 16, although both are slightly inferior to other algorithms, such as EBOM, SDNDMq2 or Hash3. The MAW22 algorithm becomes too slow for larger alphabets, $|\Sigma| \geq 32$. This is not only due to superfluous character checks, but mostly due to enlarging the search table that may not fit into the cache memory.

4 Reducing the size of the search tables

The aforementioned cashing problems make the MAW22 algorithm impractical for search in natural language texts and other useful applications, where $|\Sigma|$ is greater than $25 - 30$. However, since the array $M22$ contains not more than $2m$ different values, likely it can be represented in a more compact form. Of course, we should invent such representation that does not reduce the access speed greatly. For this goal, we use 4 one-dimensional arrays $V_0, \ldots, V_3$. The array V_3 contains the shift lengths, while for $i < 3$ the array V_i contains the pointers to some elements of V_{i+1}. In C language these arrays can be declared as follows:

```
int ***V0[], **V1[], *V2[], V3[];
```

Also we need the pointers

```
int ***p1, **p2, *p3;
```

And the shift length r can be retrieved from the arrays as follows:

```
p1 = V0[T[pos]];
p2 = p1[T[pos + 1]];
p3 = p2[T[pos + m]];
r = p3[T[pos + m + 1]];
```

To explain how the arrays V_i are organized let us assume that the search window is aligned with the beginning of the text and $x_0 = T[m-2], x_1 = T[m-1], x_2 = T[2m-2], x_3 = T[2m-1]$. The maximum possible safe shift based on the knowledge of $x_0, \ldots, x_i$ we call the shift over the vector $(x_0, \ldots, x_i)$. The arrays V_i can be divided into chunks of $|\Sigma|$ elements each and each element of V_i contains the pointer to the beginning of some chunk of V_{i+1}. The j-th chunk of V_i is processed under the assumption that the shift over $(x_0, \ldots, x_{i-1})$ is equal to j. And the k-th element of each chunk of V_i corresponds to the shift over the vector $(x_0, \ldots, x_i)$ under the assumption that $x_i = k$. In other words, if the k-th element of the j-th chunk of the array V_i contains the pointer to the beginning of the t-th chunk of the array V_{i+1}, this means that the shift over the vector $(x_0, \ldots, x_i)$ is t under the assumption that $x_i = k$ and the shift over the vector $(x_0, \ldots, x_{i-1})$ is j.

The formal definitions of the arrays $V_0, \ldots, V_3$ are as follows.

$$
\begin{aligned}
V_0[i] &= |\Sigma| \cdot \min(\{0 \le t < m-1 |\ P[m-2-t] = i\} \cup \{m-1\}) \\
V_1[|\Sigma|i + j] &= |\Sigma| \cdot \min(\{0 \le t < m-2 |\ P[m-2-t] = i \text{ AND } P[m-1-t] = j\} \cup \\
&\qquad \{P[0] = j \Rightarrow m-1\} \cup \{m\}) \\
V_2[|\Sigma|i + j] &= |\Sigma| \cdot \min(\{i < m-1 \Rightarrow i\} \cup \{m \le t < 2m-1 |\ P[2m-2-t] = j\} \cup \\
&\qquad \{i = m-1 \text{ AND } P[m-1] = j \Rightarrow m-1\} \cup \{2m-1\}) \\
V_3[|\Sigma|i + j] &= \qquad \min(\{i < m \Rightarrow i\} \cup \{P[0] = j \Rightarrow 2m-1\} \cup \{2m\} \cup \\
&\qquad \{m \le t < 2m-1 |\ P[2m-2-t] = i \text{ AND } P[2m-1-t] = j\})
\end{aligned}
$$

This principle is illustrated in Figure 2. The search tables for the pattern AGAT are shown, where Σ =AGCT and $A = 0, G = 1, C = 2, T = 3$. The pointer values are shown as the offsets from the beginning of the array V_i. The arrows show the pointer directions for the text characters x_0, x_1, x_2, x_3 =TAGG.

The search algorithms based on tables V_i we call the "Multiple adjacent windows with pointers" (MAWP). The search phases of MAWP2 and MAWP22 algorithms are just the same as in the MAW2 and MAW22 algorithms, except the computation of r, which is performed via 2 (MAWP2) or 4 (MAWP22) assignments, as shown above.

Let us calculate the size of the search tables for the MAWP2q algorithm. If the maximum shift over the vector $(x_0, \ldots, x_{i-1})$ is s, the table V_i contains $s+1$ chunks with $|\Sigma|$ elements in each. Therefore, the total size is $|\Sigma|(1+m+(m+1)+2m+\ldots+qm) = O(|\Sigma|mq^2)$, which is generally much less than $O(|\Sigma|^{2q})$ for the table $M2q$.

5 The multi-window extension

Let us consider the possibility of processing more than 2 adjacent search windows in one iteration of a search loop. The modification of the MAW2 algorithm is simple: the q-dimensional array Mq should be used instead of $M2$. It is defined as follows.

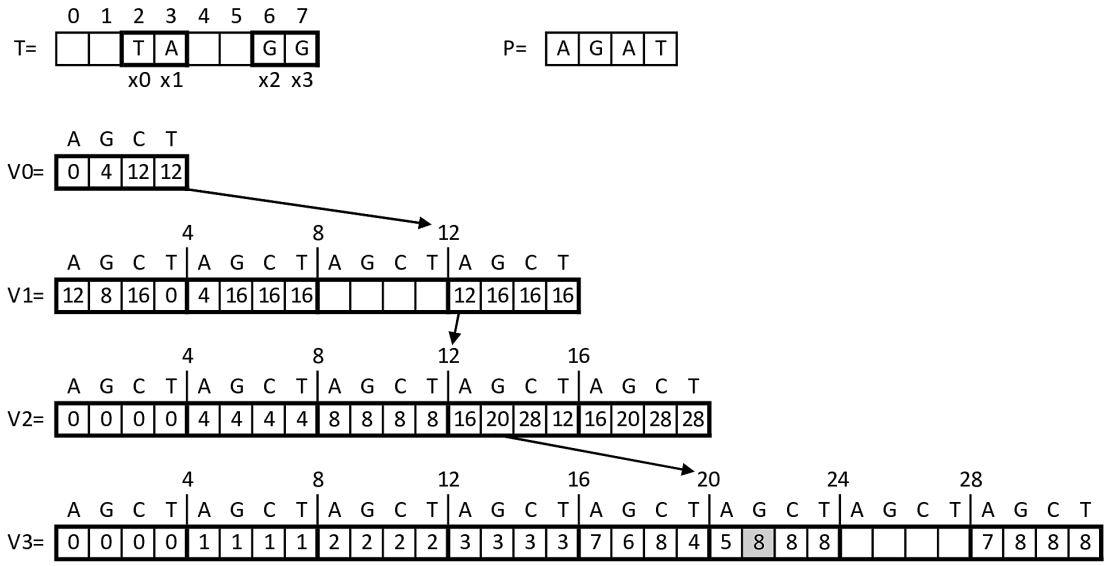

Figure 2. MAWP22 search tables structure

$Mq[i_1]\dots[i_q]$ is the leftmost possible position of a pattern under the assumption that $T[km-1]=i_k$, $k=1,\dots,q$. In Algorithm 2 only the line 3 should be changed in a following way:

$$r \leftarrow Mq[T[pos]][T[pos+m]]\dots[T[pos+qm]]. \tag{1}$$

Thus we obtain the Triple Adjacent Window (MAW3), Quadruple Adjacent Window (MAW4) and other Multiple Adjacent Windows algorithms. Using C notation assignment (1) can be rewritten as $r = *(Mq + b_{q-1}pos + \dots + b_1(pos+(q-1)m) + pos + qm)$, where $b_k = |\Sigma|^k$. The values $2m,\dots,qm$ can be pre-calculated to reduce the number of multiplications, while the values b_k are the constants.

Analogously, we can obtain the MAW32, MAW42 etc. algorithms using the $2q$-dimensional array instead of 4-dimensional in the line 2 of Algorithm 3.

However, every next dimension adds two additions, one multiplication and two memory reads in the case of MAWq and twice as large in the case of MAWq2. This overhead is covered by longer shifts until the value of $|\Sigma_P|/|\Sigma|$ is small enough. Nevertheless, any of MAWq or MAWq2 algorithms, $q \geq 3$, does not outperform the MAW2 or MAW22 respectively for any (Σ, m)-pair in the computational environment we used for the experiment. This is because (1) the probability of 3 or more adjacent maximum bad character or bad bigram shifts is not high enough and (2) filling and accessing the large search table may be time consuming.

6 Preprocessing

On the preprocessing stage of the MAWq algorithm filling the array Mq is the most time consuming operation. The following procedure completes this task.

Algorithm 6: Building the search table Mq

1 Assign the value qm to all elements of the array Mq;
2 **for** $t \leftarrow q$ downto 1 **do**
3 Replace the values $Mq[i_1]\dots[i_t]\dots[i_q]$, where $i_t \in P$, with $tm - r_t - 1$, where r_t is the position of the rightmost occurrence of i_t in P.

The first step takes $O(|\Sigma|^q)$ time, while each iteration of the loop requires $O(m|\Sigma|^{q-1})$ time. The overall time complexity of the preprocessing stage is $O(|\Sigma|^q + qm|\Sigma|^{q-1})$.

The preprocessing stage of the MAWq2 algorithm is more complicated, however, the main principle remains the same:

Algorithm 7: Building the search table $Mq2$

```
1 Assign the value qm to all elements of the array Mq2;
2 Replace the values Mq2[i_1]...[i_{2q-1}][P[0]], where i_1, ..., i_{2q-1} are any
  characters, with qm − 1;
3 for t ← q downto 1 do
4     Replace the values Mq2[i_1]...[i_{2t-1}][i_{2t}]...[i_{2q}], where bigram (i_{2t-1}, i_{2t})
      belongs to the pattern, with tm − r_t − 1, where r_t is the rightmost position
      of this bigram in the pattern P (the position of i_{2t-1});
5     if t > 1 then
6         Replace the values Mq2[i_1]...[P[0]][P[m − 1]][i_{2t}]...[i_{2q}] with
          (t − 1)m − 1;
```

Using the special functions that copy memory blocks, like memcpy from memory.h C library, we have built the implementation [21] that is faster in times than the conventional method given above. The space complexity of Multi-window algorithms is, of course, strongly greater than that one of the BMH/TBM/QS. However, the array $M2$ occupies only 64Kb of memory even for a relatively large alphabet containing 256 symbols, which is absolutely admissible for present-day computers and programs. The size of the array $M22$ is equal 64Kb for $|\Sigma| = 16$, although it is 1Mb for $|\Sigma| = 32$, which may be too big to fit the search table into the cache memory and makes the preprocessing time significant (Table 7).

Also, the different methods of filling the MAWP search tables could be developed. We implement one of them in C language ([21]). It utilizes the modified BMH and BR search tables to obtain the bad character or bad bigram heuristic. Of course, the size of the BR shift table should be taken into account, which increases the space complexity to $O(|\Sigma|mq^2 + |\Sigma|^2)$. Nevertheless, it is significantly smaller than the size of the table $M2q$. And this makes the MAWP2q methods applicable to alphabets of size 128 and more, for example, to ASCII texts, without any transformation of their characters.

7 Experimental results

We implement the QLQS, GSB, JOM and different MAW/MAWP algorithms in C language and take the source code of a number of other known algorithms from the SMART tool [10]. We choose the algorithms that were considered the fastest ones at least for one $(|\Sigma|, m)$-combination, genome sequence or English text according to [9] or our own experiments. JOM, QS, TBM and BMH times are given for comparison with the MAW2. By $\text{JOM}_{\max}$ we denote the JOM algorithm, where the adjustable shift heuristic is replaced with its maximum possible value $m + 1$. On a random text such modification is more efficient than the original version. We use the Microsoft Visual Studio 2015 compiler with the Release configuration for Win32 platform to build the executables and run them on the Intel BYT-M Core2 2840 processor of 2.16 GHz, 1 MB of L2-cache, 4GB of RAM, Windows 10 OS. The texts over alphabets of size 8 and 32 contain 5 MB of randomly generated characters with the uniform distribution, while the 4.8MB English text (Bible, KJV version) and 4.4MB genome sequence

Table 3. Running times for a genome sequence $|\Sigma| = 4$

m	2	3	4	5	6	7	8	9	10	11	12	13	14	15
MAW22	15.43	**9.74**	**7.68**	**6.39**	**5.55**	**5.00**	**4.57**	**4.25**	**4.04**	**3.86**	**3.66**	3.55	3.48	3.34
MAW22P	17.33	11.09	8.64	7.22	6.28	5.67	5.19	4.83	4.59	4.38	4.17	4.04	3.96	3.81
MAW23	16.85	10.66	8.58	7.28	6.41	5.87	5.42	5.10	4.91	4.72	4.51	4.42	4.39	4.24
MAW2	22.67	16.39	13.91	12.46	11.55	11.15	10.82	10.52	10.54	10.43	10.09	10.15	10.04	10.18
QLQS	20.42	16.83	15.17	13.92	13.07	12.63	12.23	11.88	12.04	11.76	11.38	11.41	11.51	11.52
Hash3	–	31.12	14.62	9.97	7.69	6.35	5.46	4.84	4.38	3.99	3.70	**3.47**	3.28	3.11
EBOM	15.80	11.53	9.10	8.27	7.58	6.97	6.58	6.23	5.95	5.65	5.32	5.07	4.91	4.66
TVSBS	**14.28**	12.28	10.83	9.64	8.71	7.92	7.35	6.89	6.53	6.19	5.87	5.65	5.54	5.29
FSBNDM	23.36	15.04	11.76	9.63	8.14	7.21	6.44	5.81	5.36	4.95	4.58	4.30	4.05	3.81
SA	14.30	13.91	13.92	13.92	13.87	13.86	13.87	13.86	13.86	13.86	13.86	13.88	13.87	13.86
SBNDMq2	27.78	16.65	12.05	9.96	8.58	7.57	6.88	6.30	5.85	5.45	5.06	4.76	4.49	4.23
SBNDMq4	–	–	44.12	22.53	15.13	11.49	9.30	7.83	6.81	6.03	5.42	4.95	4.55	4.21
GSB	–	–	10.42	9.01	8.08	7.33	6.81	6.35	5.96	5.59	5.21	4.92	4.65	4.39
FSB31	34.54	18.85	12.80	9.83	8.02	6.80	6.01	5.38	4.91	4.55	4.23	3.97	3.78	3.57
FSB41	–	37.19	19.16	13.01	9.87	7.98	6.84	5.93	5.23	4.64	4.52	3.81	3.75	3.35
FSB51	–	–	19.31	12.97	9.87	8.13	6.69	5.78	5.20	4.65	4.44	3.83	3.73	3.24
BSDM4	–	–	27.34	13.43	9.73	7.43	6.45	5.28	4.95	3.98	**3.66**	3.60	**3.13**	**3.08**

Table 4. Running times for English text

m	4	5	6	8	10	12	14	16	18	20	22	24
MAW22P	4.70	3.82	3.23	2.58	**2.13**	**1.87**	**1.70**	**1.53**	**1.47**	**1.37**	**1.32**	1.33
MAW2	4.40	3.74	3.32	2.76	2.38	2.16	2.01	1.85	1.81	1.70	1.64	1.67
MAW3P	5.15	4.47	4.02	3.43	3.00	2.76	2.59	2.40	2.38	2.25	2.21	2.24
JOM	5.75	5.69	5.24	4.28	4.02	3.45	2.97	2.73	2.47	2.31	2.24	2.23
BMH	5.49	4.60	4.03	3.31	2.78	2.46	2.25	2.03	1.97	1.83	1.74	1.75
QS	5.09	4.29	3.71	3.10	2.63	2.33	2.12	1.91	1.85	1.72	1.65	1.64
QLQS	5.31	4.67	4.13	3.57	3.13	2.83	2.64	2.39	2.36	2.21	2.15	2.12
Hash3	10.92	7.38	5.59	3.85	2.96	2.43	2.08	1.83	1.65	1.50	1.39	**1.32**
EBOM	4.38	**3.55**	**3.05**	**2.57**	2.18	1.96	1.82	1.59	1.63	1.47	1.41	1.54
TVSBS	**4.27**	3.71	3.30	2.70	2.29	2.02	1.83	1.65	1.55	1.42	1.34	1.33
FSBNDM	5.31	4.38	3.80	3.07	2.55	2.25	2.05	1.78	1.76	1.58	1.48	1.57
SBNDMq4	38.46	19.44	13.06	8.01	5.81	4.61	3.84	3.28	2.92	2.60	2.36	2.22
GSB	5.79	4.57	3.87	3.17	2.70	2.43	2.24	2.02	1.99	1.85	1.78	1.85
FSB31	9.54	7.25	5.86	4.35	3.46	2.91	2.53	2.21	2.06	1.85	1.71	1.69

(E.Coli bacterium) we take from the SMART tool. The patterns are randomly taken from the text (all algorithms run on the same texts and patterns).

To increase the confidence of the results we measure the time of 1000 runs of each algorithm on the same text and 1000 different patterns. We repeat the 1000-run series 10 times generating a new random text for each series and calculate the standard deviation of the series time. For any algorithm it is less than 5% when $m = 2$ and less than 1% for longer patterns. The running time includes the preprocessing time. We use the fast memory fill functions to build the shift tables on the preprocessing stage of the MAW algorithms.

The results for different texts and pattern lengths are shown in Tables 3–6. They represent the average running time over all 10 000 runs in milliseconds.

As seen, the MAW22 algorithm performs the best for the alphabet of size 8. In this case it outperforms all other algorithms not only for short patterns, but also for mid-size up to $m = 72$. For longer patterns the Hash5 algorithm becomes superior. For the genomic sequences ($|\Sigma| = 4$), the area of MAW22 superiority is narrower:

Table 5. Experimental results on rand8 problem

m	2	3	4	8	16	24	32	40	48	56	64	72	80	88
MAW22	10.76	**7.07**	**5.08**	**2.80**	**1.73**	**1.40**	**1.23**	**1.15**	**1.08**	**1.05**	**1.03**	**0.99**	0.99	0.99
MAW22P	12.55	8.11	5.83	3.22	1.96	1.59	1.40	1.30	1.22	1.19	1.18	1.13	1.12	1.13
MAW23	17.78	11.61	8.29	4.67	2.95	2.43	2.17	2.06	1.98	1.95	1.96	1.87	1.88	2.00
BMH	14.45	10.42	7.93	5.12	3.96	3.58	3.44	3.49	3.51	3.57	3.75	3.57	3.55	3.52
QS	12.11	9.30	7.41	4.82	3.63	3.23	3.10	3.15	3.14	3.17	3.36	3.20	3.20	3.16
QLQS	12.86	10.04	8.40	6.32	5.41	4.98	4.82	4.92	4.90	4.94	5.26	5.00	4.99	4.93
MAW2	12.32	8.57	6.65	4.61	3.92	3.64	3.55	3.61	3.64	3.70	3.88	3.70	3.68	3.65
Hash3	–	30.29	13.96	5.08	2.56	1.90	1.59	1.45	1.34	1.29	1.25	1.18	1.16	1.14
Hash5	–	–	–	8.33	3.07	2.03	1.59	1.37	1.23	1.14	1.06	1.01	**0.96**	**0.93**
EBOM	14.05	7.91	5.48	3.15	2.21	1.92	1.71	1.57	1.44	1.34	1.26	1.17	1.10	1.07
TVSBS	**9.88**	8.07	6.47	4.11	2.55	1.97	1.67	1.51	1.39	1.31	1.27	1.19	1.16	1.14
FSBNDM	14.83	9.70	7.00	4.02	2.53	2.01	1.68	1.60	1.46	1.35	1.26	1.27	1.24	1.20
SBNDMq2	26.43	13.96	9.16	4.52	2.64	2.07	1.75	1.66	1.54	1.42	1.32	1.29	1.25	1.21
GSB	–	–	8.71	4.69	3.09	2.58	2.28	2.14	1.94	1.80	1.69	1.67	1.66	1.63
FSB31	34.46	17.66	11.33	5.11	2.64	1.90	1.55	1.48	1.39	1.32	1.25	1.28	1.26	1.22
FSB41	–	41.78	20.64	7.16	3.29	2.26	1.76	1.54	1.38	1.27	1.18	1.17	1.15	1.13
FSB51	–	–	20.59	7.16	3.29	2.26	1.76	1.54	1.38	1.28	1.19	1.17	1.16	1.13

Table 6. Experimental results on rand32 problem

m	2	3	4	5	6	7	8	9	10
BMH	10.65	7.07	5.29	4.37	3.74	3.29	2.96	2.70	2.49
QS	7.18	5.30	4.23	3.61	3.18	2.85	2.60	2.40	2.25
QLQS	7.76	5.66	4.49	3.85	3.42	3.09	2.86	2.67	2.52
TBM	8.01	5.26	4.07	3.40	2.96	2.57	2.37	2.13	2.02
MAW2	7.17	**4.83**	**3.67**	**3.08**	**2.68**	2.39	2.19	2.04	1.91
MAW2P	8.26	5.55	4.21	3.51	3.06	2.72	2.48	2.31	2.15
MAW22P	10.51	6.86	5.09	4.15	3.53	3.09	2.75	2.49	2.30
JOM	8.05	6.46	5.25	4.84	4.06	3.61	3.29	3.09	2.91
JOM_{max}	7.97	6.16	4.70	4.29	3.31	3.08	2.88	2.72	2.65
Hash3	–	27.14	13.22	8.95	6.81	5.52	4.69	4.07	3.60
EBOM	12.97	6.44	4.32	3.34	2.74	**2.38**	**2.09**	**1.89**	**1.73**
TVSBS	**6.44**	5.10	4.14	3.60	3.18	2.87	2.62	2.40	2.25
FSBNDM	9.83	6.35	4.73	3.86	3.28	2.87	2.56	2.35	2.14
GSB	–	–	7.54	5.83	4.80	4.12	3.63	3.26	2.98
FSB31	33.79	16.57	10.84	8.23	6.66	5.61	4.87	4.32	3.88

$3 \leq m \leq 11$, then the BSDM4 and Hash3 algorithms perform better. For larger alphabets the MAW22 algorithm becomes inefficient due to the enlarging search table. However, the MAW22P algorithm appears on the scene. It outperforms all the other when searching the patterns of the length $10 \leq m \leq 22$ in English text. We test the English text in ASCII encoding and assume $|\Sigma| = 128$. The MAW2 algorithm appears to be the fastest one for short patterns ($3 \leq m \leq 6$) and a random text over the alphabet of size 32. It also performs well on short patterns and larger alphabets, but becomes slightly inferior to some other algorithms, such as FJS [11].

Also, we measured the preprocessing and search time separately for 5MB random texts over the alphabets of different size and pattern lengths 10 and 20. Table 7 presents times of search in milliseconds and the ratio of preprocessing to search. As expected, the preprocessing time of the MAW22 grows rapidly depending on the alphabet size, making this algorithm impractical even for $|\Sigma| = 32$. At the same time, in all other algorithms the ratio of preprocessing to search remains reasonable even

Table 7. The balance between the preprocessing and the search

		MAW2		MAW2P		MAW22		MAW22P	
$\|\Sigma\|$	m	search	preprocessing	search	preprocessing	search	preprocessing	search	preprocessing
4	10	4.42	0.00013%	5.03	0.027%	1.90	0.044%	2.09	0.85%
4	20	3.69	0.00063%	4.25	0.065%	1.35	0.15%	1.49	2.35%
32	10	1.91	0.14%	2.14	0.31%	17.97	15.4%	2.27	4.1%
32	20	1.14	0.29%	1.27	0.45%	7.78	28.6%	1.12	6.4%
128	10	3.55	0.6%	3.94	0.3%	114.99	1151%	5.18	3.2%
128	20	2.94	2.2%	3.09	0.5%	61.7	2271.7%	3.38	5.4%

for $|\Sigma| = 128$. Note that in the MAW22 even the search phase itself is too expensive for $|\Sigma| \geq 32$ and this problem is also resolved in the MAW22P.

8 Conclusions

A new family of exact pattern matching algorithm is developed. They exploit the idea of processing more than one adjacent search window in each iteration of the search loop using the multi-dimensional search tables. This approach allows to decrease the search time by the cost of space. However, the space overhead is admissible even for large alphabets if we replace the multi-dimensional search tables with a series of one-dimensional tables linked by pointers. We carry out an experiment for English text, genome sequences and random texts over the alphabets of size 8 and 32. The performance of our algorithms was compared with other algorithms, which are known as the fastest ones for respective alphabet size / pattern length. In our computational environment the MAW22 algorithm outperforms all the other in searching genome sequences of length $3 \leq m \leq 11$ and random patterns of length $3 \leq m \leq 72$ for $|\Sigma| = 8$. The MAW22P algorithm demonstrates the best performance in searching the patterns of length $10 \leq m \leq 22$ in English text.

The multiple adjacent windows approach can be developed further. For example, it is worthwhile to investigate even more economic methods of packing the multi-dimensional shift information. This would make possible to construct practically efficient algorithms based on checking the q-grams ($q \geq 3$) in adjacent search windows.

References

1. T. Berry and S. Ravindran: *A fast string matching algorithm and experimental results*, in Proceedings of the Prague Stringology Club Workshop, Edited by Jan Holub and Milan Simanek, Eds. Czech Technical University in Prague, Czech Republic, 1999, pp. 16–28.
2. R. S. Boyer and J. S. Moore: *A fast string searching algorithm.* Commun. ACM, 20(10) 1977, pp. 762–772.
3. D. Cantone and S. Faro: *Improved and self-tuned occurrence heuristics.* Journal of Discrete Algorithms, 28 2014, pp. 73–84.
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: *Evaluation and improvement of fast algorithms for exact matching on genome sequences*, in Algorithms for computational biology, Proceedings of Third International Conference, 2016, pp. 145–157.
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: *A fast suffix automata based algorithm for exact online string matching*, in Moreira, N., Reis, R. (eds.) CIAA, vol. 7381, 2012, pp. 149–158.
8. S. Faro and T. Lecroq: *A multiple sliding windows approach to speed up string matching algorithms*, in 11-th International Symposium on Experimental Algorithms (SEA), 2012, pp. 172–183.
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. S. Faro, T. Lecroq, S. Borzi, S. D. Mauro, and A. Maggio: *The string matching algorithms research tool*, in Proceedings of the Prague Stringology Conference, J. Holub and J. Zdarek, Eds. Czech Technical University in Prague, Czech Republic, 2016, pp. 99–111.
11. F. Franek, C. Jennings, and W. Smyth: *A simple fast hybrid pattern-matching algorithm.* J. Discret. Algorithms, 5(4) 2007, pp. 682–695.
12. N. R. Horspool: *Practical fast searching in strings.* Soft.Pract.Exp., 10(6) 1980, pp. 501–506.
13. A. Hudaib, R. Al-Khalid, A. Al-Anani, M. Itriq, and D. Suleiman: *Four sliding windows pattern matching algorithm (fsw).* Journal of Software Engineering and Applications, 8(8) 2015, pp. 154–165.
14. A. Hume and D. Sunday: *Fast string searching.* Software - Practice & Experience, 21(11) 1991, pp. 1221–1248.
15. T. Lecroq: *Fast exact string matching algorithms.* Inf. Process. Lett., 102(6) 2007, pp. 229–235.
16. H. Peltola and J. Tarhio: *String matching with lookahead.* Discrete Applied Mathematics, 163 2014, pp. 352–360.
17. D. M. Sunday: *A very fast substring search algorithm.* Comm. ACM, 33(8) 1990, pp. 132–142.
18. R. Thathoo, A. Virmani, S. S. Lakshmi, N. Balakrishnan, and K. Sekar: *Tvsbs: A fast exact pattern matching algorithm for biological sequences.* Indian Acad. Sci., Current Sci., 91(1) 2006, pp. 47–53.
19. B. W. Watson, D. G. Kourie, and L. G. Cleophas: *Quantum leap pattern matching*, in Proceedings of the Prague Stringology Conference, J. Holub and J. Zdarek, Eds. Czech Technical University in Prague, Czech Republic, 2015, pp. 104–117.
20. S. Wu and U. Manber: *Fast text searching allowing errors.* Commun. ACM, 35(10) 1992, pp. 83–91.
21. I. O. Zavadskyi: *The maw/mawp algorithms.* (https://github.com/zavadsky/stringology).