# Evaluation and Improvement of Fast Algorithms for Exact Matching on Genome Sequences

Simone Faro

Dipartimento di Matematica e Informatica, Università di Catania, Viale A.Doria n.6, 95125 Catania, Italy, faro@dmi.unict.it

**Abstract.** With the availability of large amounts of DNA data, exact matching of nucleotide sequences has become an important application in modern computational biology and in meta-genomics. In the last decade several efficient solutions for the exact string matching problem have been developed and most of them are very fast in practical cases. However when the length of the pattern is short or the alphabet size is small (as in the case of DNA sequences) the problem becomes more difficult to be solved efficiently. In this paper we review and compare the most efficient solutions for the online exact matching problem appeared in the latest years when applied for searching on genome sequences. In addition we also propose some new variants of an efficient string matching algorithm. From our experimental results it turns out that the newly presented variants are very fast in most practical cases.

**Keywords:** Exact sequence analysis, string matching, experimental algorithms, automata based solution

## 1 Introduction

In molecular biology, nucleotide sequences are the fundamental information for each species and a comparison between such sequences is an interesting and basic problem. Generally biological information is stored in strings of nucleic acids (DNA, RNA) or amino acids (proteins). With the availability of large amounts of DNA data, matching of nucleotide sequences has become an important application and there is an increasing demand for fast computer methods for analysis and data retrieval [15]. There are various kinds of comparison tools which provide aligning and approximate matching (see for instance [19, 15]), however most of them are based on exact matching in order to speed up the process. Moreover exact string matching is widely used in computational biology for a variety of other tasks. Thus the need for fast matching algorithms on DNA sequences.

In this article we consider the problem of finding all the (possibly overlapping) occurrences of a pattern $P$ of length $m$ in a text $T$ of length $n$, both drawn over an alphabet $\Sigma$ of size $\sigma$. We focus on the case where the text $T$ and the pattern $P$ are DNA sequences over a finite alphabet $\Sigma = \{\mathtt{a}, \mathtt{c}, \mathtt{g}, \mathtt{t}\}$ of constant size $\sigma = 4$. We are interested here in the problem where the pattern is given first and can then be searched in various texts, thus a preprocessing phase is allowed (and in

most cases suggested) on the pattern. This problem is referred in literature as the *exact online string matching problem*.

The problem of searching DNA sequences has been extensively studied in the last years and its importance in modern biology has led to produce much works. In the field of single string matching, Kalsi *et al.* [13] performed an experimental comparison of the most efficient algorithms for searching biological sequences. In addition in [8, 11] Faro and Lecroq presented an extensive evaluation of (almost) all existing exact string matching algorithms (up to 2010) under various conditions, including alphabet of four characters and DNA sequences. In 2002 Navarro and Raffinot presented a comparison [18] of all matching algorithms on biological sequences, including multiple pattern matching algorithms. More recently, in 2011, Kouzinopoulos and Margaritis conducted another experimental comparison [14] taking into account the most recent solutions.

In recent years a lot of work has been made in this field and several algorithms can be considered as potential candidates to be among the fastest solutions to search genome sequences.

In this paper we present a brief survey of the most efficient solutions to the string matching problem presented in the last few years and compare them in the task of searching genome sequences. In addition we also present some efficient variants of one of the previous presented algorithms and compare them, in terms of running times, in order to evaluate their performances under various conditions. From our experimental results it turns out that some algorithms appeared in the latest years are among the fastest solutions for searching genome sequences. In addition the newly presented variants obtain the best results in almost all the practical cases.

The paper is organized as follows. In Section 2 we review the previous results known in literature based and describe the latest and most efficient solutions for searching genome sequences, including the BSDM algorithm. Then in Section 3 we present some new variants of the BSDM algorithm. In Section 4 we compare the newly presented solutions with the most efficient algorithms known in literature. We draw our conclusions in Section 5.

## 2 Fast Algorithms for Searching Genome Sequences

Basically a string matching algorithm uses a window to scan the text. The size of this window is equal to the length of the pattern. It first aligns the left ends of the window and the text. Then it checks if the pattern occurs in the window (this specific work is called an *attempt*) and then shifts the window to the right. It repeats the same procedure again until the right end of the window goes beyond the right end of the text.

When a similarity has been detected a naive check of the occurrence is performed. In order to detect the similarity between the pattern and the text window efficient algorithms use *bit-parallelism* or *character comparisons*. Both techniques can be improved by using condensed alphabets and hashing.

In particular the pattern $P$ is arranged using a condensed alphabet. In such a representation groups of $q$ adjacent characters of the pattern are condensed in a single character by using a suitable hash function $h : \Sigma^q \to \{0, \ldots, \text{MAX}\}$, for a constant value MAX. In practice, the value of $q$ varies with $m$ and the size of the alphabet and the value of the constant MAX varies with the memory space available.

The *bit-parallelism technique* [1] takes advantage of the intrinsic parallelism of the bit operations inside a computer word, allowing to cut down the number of operations that an algorithm performs by a factor up to $\omega$, where $\omega$ is the number of bits in a computer word. This technique is particularly suitable for simulating non-deterministic automata for a single pattern [1] and for multiple patterns [3].

In the following sections we briefly review some of the most recent and efficient solutions for the exact string matching problem.

**The Backward DAWG Matching Algorithm.**

One of the first application of the suffix automaton to get optimal pattern matching algorithms on the average was presented in [4]. The algorithm which makes use of the suffix automaton of the reverse pattern is called Backward-DAWG-Matching algorithm (BDM). Such algorithm moves a window of size $m$ on the text. For each new position of the window, the automaton of the reverse of $P$ is used to search for a factor of $P$ from the right to the left of the window. The basic idea of the BDM algorithm is that if the backward search failed on a letter $c$ after the reading of a word $u$ then $cu$ is not a factor of $p$ and moving the beginning of the window just after $c$ is secure. If a suffix of length $m$ is recognized then an occurrence of the pattern was found.

**The Backward Nondeterministic DAWG Matching Algorithm.**

The BNDM algorithm [17] simulates the suffix automaton for $P^r$ (the reverse of $P$) with the bit-parallelism technique, for a given string $P$ of length $m$. The bit-parallel representation uses an array $B$ of $|\Sigma|$ bit-vectors, each of size $m$, where the $i$-th bit of $B[c]$ is set if and only if $P[i] = c$, for $c \in \Sigma$, $0 \leq i < m$. Automaton configurations are then encoded as a bit-vector $D$ of $m$ bits, where each bit corresponds to a state of the suffix automaton (the initial state does not need to be represented, as it is always active). In this context the $i$-th bit of $D$ is set iff the corresponding state is active. $D$ is initialized to $1^m$ and the first transition on character $c$ is implemented as $D \leftarrow (D \;\&\; B[c])$. Any subsequent transition on character $c$ can be implemented as $D \leftarrow ((D \ll 1) \;\&\; B[c])$.

The BNDM Algorithm works by shifting a window of length $m$ over the text. Specifically, for each window alignment, it searches the pattern by scanning the current window backwards and updating the automaton configuration accordingly. Each time a suffix of $P^r$ (i.e., a prefix of $P$) is found, namely when prior to the left shift the $m$-th bit of $D\&B[c]$ is set, the window position is recorded. A search ends when either $D$ becomes zero (i.e., when no further prefixes of $P$ can be

found) or the algorithm has performed $m$ iterations (i.e., when a match has been found). The window is then shifted to the start position of the longest recognized proper prefix.

In [7] an effective variant of the BNDM algorithm was presented. Such variant, called Forward-BNDM (FBNDM), takes into account the forward character (i.e. the character which is just after the current window of the text) for computing the shift advancement. This leads to a more efficient solution especially in the case of short pattern. The FBNDM algorithm has been later improved in many ways.

**The BNDM Algorithm with Extended Shift.**

Durian *et al.* presented in [6] another efficient algorithm for simulating the suffix automaton in the case of long patterns. The algorithm is called BNDM with eXtended Shift (BXS). The idea is to cut the pattern into $\lceil m/\omega \rceil$ consecutive substrings of length $w$ except for the rightmost piece which may be shorter. Then the substrings are superimposed getting a superimposed pattern of length $\omega$. In each position of the superimposed pattern a character from any piece (in corresponding position) is accepted. Then a modified version of BNDM is used for searching consecutive occurrences of the superimposed pattern using bit vectors of length $\omega$ but still shifting the pattern by up to $m$ positions. The main modification in the automaton simulation consists in moving the rightmost bit, when set, to the first position of the bit array, thus simulating a circular automaton. Like in several other cases, the BXS algorithm works as a filter algorithm, thus an additional verification phase is needed when a candidate occurrence has been located.

**The Factorized BNDM Algorithm.**

Cantone *et al.* presented in [2] an alternative technique, still suitable for bit-parallelism, to encode the nondeterministic suffix automaton of a given string in a more compact way. Their encoding is based on factorizations of strings in which no character occurs more than once in any factor. It turns out that the nondeterministic automaton can be encoded with $k$ bits, where $k$ is the size of the factorization. Though in the worst case $k = m$, on the average $k$ is much smaller than $m$, making it possible to encode large automata in a single or few computer words. As a consequence, their bit-parallel variant of the BNDM, called Factorized BNDM algorithm (KBNDM) based on such approach tends to be faster in the case of sufficiently long patterns.

### 2.1 The Backward SNR DWAG Matching

Faro and Lecroq presented in [9] a fast and simple variant of the BDM algorithm which does not make use of bit parallelism still using a compact representation of the underlying automaton. It consist in computing the longest substring of

the pattern with no repetitions, i.e. in which each character is repeated at most once, and in constructing the suffix automaton of such a substring. This leads to a simple encoding and, by convenient alphabet transformations, to quite long automata. The algorithm is named Backward-SNR-DAWG-Matching (BSDM), where SNR is the acronym of *substring with no repetitions*.

The main interesting aspect of such technique is that only an integer value between 0 and $m$ is needed to represent the whole automaton. Since each character is repeated at most once we need only to maintain the information about the current active state, if one.

However it turns out that in many practical cases the length of the maximal SNR is not large enough if compared with the size of the pattern. This happens especially for patterns over small alphabets, as in the case of genome sequences, or for patterns with characters occurring many times, as in the case of a natural language text. In order to allow longer SNR it is convenient to use a condensed alphabet whose characters are obtained by combining groups of $q$ characters, for a fixed value $q$. It turns out that the length of the maximal SNR, though quite less than $m$ in most cases, is quite larger than the size of a computer word (which typically is 32 or 64). This leads to larger shift in a suffix automata based algorithm.

Since BSDM is a filter based algorithm, as in many other cases, a naive test is needed when a candidate occurrences of the pattern is found.

### The Two-Way Shift-And Algorithm

In [5] Durian *et al.* presented the Two-Way Shift-Or algorithm (TSO) which extends the original Shift-And algorithm [1] and obtains more effective results in practical cases. Specifically it uses the same vector $B$ as the Shift-Or algorithm but traverses the text with a fixed step of $m$ positions. At each step $i$, an alignment window $T[i-m+1...i+m-1]$ is inspected. The positions $T[i...i+m-1]$ correspond to the end positions of possible matches and at the same time, to the positions of the state vector $D$. Inspection starts at the character $T[i]$, and it proceeds with a pair of characters $T[i-j]$ and $T[i+j]$ until corresponding bits in $D$ become $1^m$ or $j=m$ holds. In TSO, the testing of the state vector $D$ is slightly faster, when the bit-vectors are seated to the highest order bits. The Two-Way Shift-And algorithm (TSA) is the dual of TSO which turns out to be faster in practical cases, especially when implemented with $q$ enrolling characters (TSA$q$).

Such two-way algorithms check text in alignment windows of $m$ consecutive text positions, thus a mismatch can be detected immediately based on the first examined text character. In the best case the performance can be $\mathcal{O}(n/m)$. On the other hand in the worst case all text characters except the last characters in each alignment window are examined twice.

The main problem associated with these solutions is that they are not able to retrive the positions of the occurrences of the pattern but only its number. A modification of such solutions which retrieve all positions of the occurrences can be obtained but with slower performance.

## 3 New Improvements of the BSDM Algorithm

In this section we propose some new variants of the BSDM algorithm described above, which turns out to be one of the best solutions for searching DNA sequences. Specifically we focus on reducing the number of false positives detected during the searching phase in order to reduce the number of naive tests. This can be done by using different and more effective hash function in the implementation of the condensed alphabet. In addition we use an effective technique recently introduced in [9] consisting in the use of several sliding windows while searching the pattern along the text, and which is able to speed up the whole process up to a factor of 1.3 under suitable conditions.

### 3.1 Improved Hash Functions for Condensed Alphabets

As we observed above, most filtering algorithms obtain better performances when used for searching sequences over large alphabets. When the size of the underlying alphabet is small it is possible to extend it by arranging the pattern $P$ using a condensed alphabet. In such a representation groups of $q$ adjacent characters of the pattern are condensed in a single character by using a suitable hash function $h : \Sigma^q \to \{0, \ldots, \text{MAX}\}$, for a constant value MAX. In practice, the value of $q$ varies with $m$ and the size of the alphabet and the value of the constant MAX varies with the memory space available.[1] Thus a pattern $P$ of length $m$ translates in a condensed pattern $P^{(q)}$ of length $m - q + 1$ where, for $0 \leq i \leq m - q$

$$P^{(q)}[i] = h(P[i \,..\, i + q - 1]).$$

The hashing method adopted in standard implementations of condensed alphabets is based on a shift-and-addition procedure. Specifically, if $x \in \Sigma^q$, with $x = x[0 \,..\, q - 1]$, then $h(x)$ can be efficiently computed by

$$h(x) = \sum_{i=0}^{q-1} ((P[i] \;\&\; M) \ll k(q - i - 1)) \tag{1}$$

where $k$ is a constant and $M$ is a bit-mask both dependent on $q$. In practice $k$ is set to $\lfloor \omega/q \rfloor$ and $M$ is set to $0^{\omega-k}1^k$, where $\omega$ is the size of the register used for hashing $q$-grams. The hash function shown in (1) has been used, for instance, in the Hash$q$ algorithm [16] and in the Wu-Manber algorithm [20] for the exact multiple pattern matching problem.

Depending on the underlying alphabet, better hash function could be adopted in order to reduce the collisions in the hash value associated with different groups of characters. For instance the DNA alphabet is composed by the four characters {a, c, g, t}, whose ASCII codes are {01000001, 01000011, 01000111, 01010100}. Using $k = 2$ and a suitable masking leads to a perfect hashing. However for larger alphabets or when $q$ is greater than 5 only a resemblance can be used.

[1] In our implementation we use a value of MAX equal to $2^{16}$ and use a 16-bit register for each hash value.

In our analysis we took into account six different hash functions (including the perfect hash) and evaluated them in terms of number of collisions and performances. Specifically we considered the following set of hash functions, where we set for simplicity the value of $q$ to 4. However it is easy to extend them to greater values of the parameter $q$

| | | | |
|---|---|---|---|
| 1. | Shift-Addition | $(a \ll 6) + (b \ll 4) + (c \ll 2) + d$ | 5% |
| 2. | Short-Shift-Addition | $(a \ll 3) + (b \ll 2) + (c \ll 1) + d$ | 38% |
| 3. | Addition | $a + b + c + d$ | 88% |
| 4. | Shift-Substract | $(a \ll 6) - (b \ll 4) - (c \ll 2) - d$ | 5% |
| 5. | Shift-And | $(a \ll 6)$ and $(b \ll 4)$ and $(c \ll 2)$ and $d$ | 99% |
| 6. | Shift-Or | $(a \ll 6)$ or $(b \ll 4)$ or $(c \ll 2)$ or $d$ | 95% |
| 7. | Shift-Xor | $(a \ll 6)$ xor $(b \ll 4)$ xor $(c \ll 2)$ xor $d$ | 3% |
| 8. | Perfect-Hash | $(\alpha(a) \ll 6) + (\alpha(b) \ll 4) + (\alpha(c) \ll 2) + \alpha(d)$ | 0% |

Where $\alpha(c) = (c \text{ and } 6) \gg 1$, for each $c$ in the set $\{\mathsf{a}, \mathsf{c}, \mathsf{g}, \mathsf{t}\}$.

From our analysis it turns out that the hash functions 1. and 3. obtain up to 5.5% of collisions. When the length of the shift decreases (function 2.) the number of collisions increases to 38% and reach the percentage of 88% when the the shift is reduced to 0. This percentage rises up to 99.2% for hash functions 5. and 6. where the bitwise and and or are used in place of arithmetic operations. However it decreases to 3.2% in the case of function 7 where the bitwise xor is used. Of course the number of collision is 0 in the case of function 8.

**Table 1.** Experimental evaluation of the BSDM4 algorithm implemented with 8 different hash functions. Running times are expressed in milliseconds and has been computed as the mean of 500 searches on a genome sequence of 5Mb.

| $m$ | 4 | 8 | 16 | 32 | 64 | 128 | 256 | 512 | 1024 | 2048 |
|---|---|---|---|---|---|---|---|---|---|---|
| FUNC.1 | 8.41 | 3.70 | 2.78 | 2.35 | 2.26 | 2.21 | 2.15 | 2.09 | 2.13 | 2.12 |
| FUNC.2 | 8.82 | 3.95 | 3.03 | 2.63 | 2.49 | 2.46 | 2.38 | 2.38 | 2.35 | 2.36 |
| FUNC.3 | 11.08 | 6.96 | 6.06 | 5.77 | 5.55 | 5.35 | 5.14 | 5.12 | 4.93 | 4.81 |
| FUNC.4 | 8.32 | 3.69 | 2.77 | 2.36 | 2.25 | 2.20 | 2.14 | 2.06 | 2.11 | 2.12 |
| FUNC.5 | 42.51 | 36.21 | 27.11 | 18.62 | 16.66 | 17.32 | 16.84 | 16.99 | 17.09 | 16.84 |
| FUNC.6 | 19.58 | 14.27 | 11.24 | 9.09 | 8.01 | 7.45 | 6.73 | 6.44 | 6.10 | 5.98 |
| FUNC.7 | 7.96 | 3.56 | 2.67 | 2.35 | 2.22 | 2.14 | 2.06 | 2.08 | 2.07 | 2.06 |
| FUNC.8 | 10.93 | 4.47 | 3.43 | 2.92 | 2.77 | 2.71 | 2.62 | 2.61 | 2.58 | 2.58 |

Table 1 shows the evaluation, in terms of running times, of the BSDM4 algorithm when the 8 different hash functions presented above are used. In the table running times are expressed in milliseconds and has been computed as the mean of 500 searches on a genome sequence of 5Mb.[2] It turns out that the number of collisions generated by the hash function partially reflects the performance of the respective algorithm. However it is also affected by the number of operations

[2] The details of experimental settings can be found in Section 4.

needed for computing the hash value. Thus the variant using function 8. does not obtain the best results since the number of operation is doubled. Best results are obtained in all cases by the variant using the xor bitwise operation (i.e. function n.7).

### 3.2 A Multiple Sliding Windows Variant of the BSDM Algorithm

In this section we describe a multiple windows variant of the BSDM algorithm which improves the practical performances of the original solution. The general approach, introduced for the first time in [10], can be seen as a filtering method which consists in processing $k$ different windows of the text at the same time, with $k \geq 2$.

Suppose $P$ is a pattern of length $m$ and $T$ is a text of length $n$. Without loss in generality we can suppose that $n$ can be divided by $k$, otherwise the rightmost ($n \bmod k$) characters of the text could be associated with the last window (as described below). Moreover we assume for simplicity that $m < n/k$ and that the value $k$ is even.

Under the above assumptions the approach can be summarized as follows: if the algorithm searches for the pattern $P$ in $T$ using a text window of size $m$, then partition the text in $k/2$ partially overlapping substrings, $T_0, T_1, \ldots, T_{k/2-1}$, where $T_i$ is the substring $T[2i\lceil n/k \rceil \,..\, 2(i+1)n/k+m-2]$, for $i = 0, \ldots, (k-1)/2$, and $T_{k/2-1}$ (the last window) is set to $T[n-(2n/k)\,..\,n-1]$.

Then process simultaneously the $k$ different text windows, $w_0, w_1, \ldots, w_{k-1}$, where we set $w_{2i} = T[s_{2i} - m + 1\,..\,s_{2i}]$ (and call them *left windows*) and $w_{2i+1} = T[s_{2i+1}\,..\,s_{2i+1} + m - 1]$ (and call them *right windows*), for $i = 0, \ldots, (k-2)/2$.

The couple of windows ($w_{2i}$, $w_{2i+1}$), for $i = 0, \ldots, (k-2)/2$, is used to process the substring of the text $T_i$. Specifically the window $w_{2i}$ starts from position $s_{2i} = (2n/k)i + m - 1$ of $T$ and slides from left to right, while window $w_{2i+1}$ starts from position $s_{2i+1} = (2n/k)(i+1) - 1$ of $T$ and slides from right to left (the window $w_{k-1}$ starts from position $s_{k-1} = n - m$). For each couple of windows ($w_{2i}$, $w_{2i+1}$) the sliding process ends when the window $w_{2i}$ slides over the window $w_{2i+1}$, i.e. when $s_{2i} > s_{2i+1} + m - 1$. It is easy to prove that no candidate occurrence is left by the algorithm due to the $m-1$ overlapping characters between adjacent substrings $t_i$ and $t_{i+1}$, for $i = 0, \ldots, k-2$.

Fig. 1 presents a scheme of the search iteration of the multiple sliding windows matcher for $k = 1$, 2 and 4. It has been proved in [10] that this approach can be applied to all string matching algorithms, including the BSDM algorithm. Moreover it can be noticed that the worst case time complexity of the original algorithm does not degrade with the application of the multiple sliding windows approach.

On the other hand it turns out, when the alphabet is small as in the case of DNA sequences, that the performances of the original algorithm degrade by applying the new method, since the probability to find mixed candidate positions increases substantially.

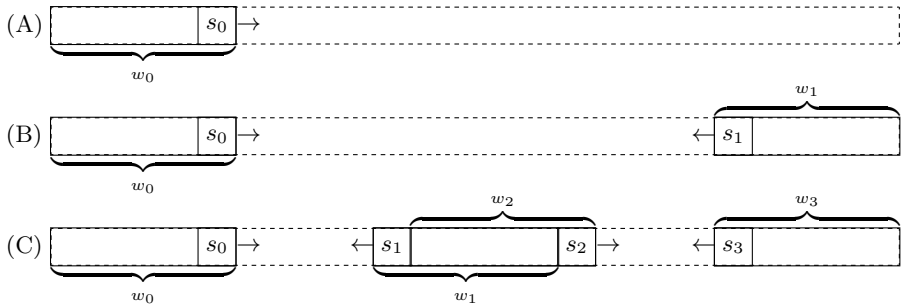


**Fig. 1.** A general scheme for the multiple sliding windows approach with (A) a single window, (B) two windows and (C) four windows ($w_1$ and $w_2$ are overlapping).

## 4 Experimental Results

In this section we briefly present experimental evaluations in order to understand the performances of the newly presented algorithm and to compare it against the best string matching algorithms for searching genome sequences. In particular we tested the following algorithms:

- the Backward-Nondeterministic-DAWG-Matching algorithm [17] (BNDM$q$) implemented using $q$-grams and a value of $q = 4$;
- the Extended Backward-Oracle-Matching algorithm [7] (EBOM);
- the Hashing algorithm [16] (HASH$q$) implemented using $q$-grams and a value of $q \in \{3, 4, 5\}$;
- the Simplified version of the BNDM algorithm [17] (SBNDM$q$) implemented using $q$-grams and a value of $q = 4$;
- the Forward Simplified version of the BNDM algorithm [7] (FSBNDM$q$) implemented using $q$-grams and a value of $q = 4$;
- the Multiple Windows version of the Forward Simplified BNDM algorithm [10] (FSBNDM-W4) implemented using 4 sliding windows;
- the Factorized BNDM algorithm [2] (KBNDM);
- the BNDM algorithm with Extended Shift [6] (BXS$q$) implemented using $q$-grams and a value of $q = 4$;
- The Backward-SNR-DAWG-Matching algorithm [9] using condensed alphabets with groups of $q$ characters, with $q \in \{1, 2, 4, 6, 7\}$ (BSDM$q$)
- The new BSDM algorithm using condensed alphabets and a *shift-xor* hash function, with $q \in \{1, 2, 4, 6, 7\}$ (BSDM$q$x)
- The Multiple Windows version of the new BSDM algorithm using condensed alphabets and a *shift-and-xor* hash function (BSDM$q$x-w2 and BSDM$q$x-w4) implemented using 2 and 4 sliding windows, respectively;

For the sake of completeness we evaluate also the following two string matching algorithms for *counting* occurrences. They do not report the positions but only the total number of all occurrences.

**Table 2.** Experimental results on a genome sequence. Best results have been bold faced. Running times are expressed in milliseconds. For the algorithms using variable $q$-grams we report in brackets the value of $q$ which obtains the best running times. The EPSM and TSO$q$ algorithms (indicated by a * symbol) are *counting* algorithm, i.e. it is able only to count occurrences.

| $m$ | 4 | 8 | 16 | 32 | 64 | 128 | 256 | 512 | 1024 | 2048 |
|---|---|---|---|---|---|---|---|---|---|---|
| BNDM$q$ | 11.14$^{(4)}$ | 4.12$^{(4)}$ | 3.02$^{(4)}$ | 2.41$^{(4)}$ | 2.41$^{(4)}$ | 2.39$^{(4)}$ | 2.23$^{(4)}$ | 2.40$^{(4)}$ | 2.32$^{(4)}$ | 2.33$^{(4)}$ |
| EBOM | **7.74** | 7.00 | 5.53 | 4.01 | 3.10 | 2.62 | 2.38 | 2.21 | 2.36 | 2.65 |
| HASH$q$ | 18.31$^{(3)}$ | 7.67$^{(3)}$ | 4.69$^{(5)}$ | 3.32$^{(5)}$ | 2.85$^{(5)}$ | 2.35$^{(5)}$ | 2.57$^{(5)}$ | 2.45$^{(5)}$ | 2.34$^{(5)}$ | 2.30$^{(5)}$ |
| SBNDM$q$ | 10.32$^{(4)}$ | 4.00$^{(4)}$ | 2.96$^{(4)}$ | 2.37$^{(4)}$ | 2.39$^{(4)}$ | 2.31$^{(4)}$ | 2.29$^{(4)}$ | 2.34$^{(4)}$ | 2.28$^{(4)}$ | 2.37$^{(4)}$ |
| BSDM$q$ | 8.43$^{(4)}$ | 3.02$^{(6)}$ | 2.41$^{(6)}$ | 2.42$^{(7)}$ | 2.13$^{(7)}$ | 1.98$^{(7)}$ | 2.01$^{(7)}$ | 2.00$^{(7)}$ | 2.00$^{(7)}$ | 2.00$^{(7)}$ |
| BXS$q$ | 15.34$^{(4)}$ | 4.50$^{(4)}$ | 3.11$^{(4)}$ | 2.39$^{(4)}$ | 2.42$^{(4)}$ | 2.37$^{(4)}$ | 2.40$^{(4)}$ | 2.40$^{(4)}$ | 2.40$^{(4)}$ | 2.41$^{(4)}$ |
| FS-w4 | 16.54 | 5.20 | 5.05 | 3.76 | 4.09 | 3.61 | 3.88 | 3.52 | 3.34 | 3.09 |
| FSBNDM-w4 | 17.10 | 8.09 | 4.30 | 2.99 | 3.00 | 2.89 | 2.94 | 2.94 | 2.97 | 2.95 |
| KBNDM | 10.80 | 8.00 | 5.88 | 3.98 | 3.03 | 2.91 | 2.94 | 2.92 | 2.97 | 2.97 |
| TSO$q$ (*) | **5.32**$^{(5)}$ | 3.68$^{(5)}$ | 2.88$^{(5)}$ | 2.28$^{(5)}$ | 1.96$^{(5)}$ | - | - | - | - | - |
| EPSM (*) | 5.87 | 3.72 | 2.50 | **1.93** | **1.75** | **1.72** | **1.66** | **1.62** | **1.65** | **165** |
| BSDM2x | 8.32 | 7.53 | 6.80 | 6.12 | 5.66 | 5.24 | 4.94 | 4.70 | 4.46 | 4.28 |
| BSDM4x | 8.09 | **3.01** | **2.40** | 2.21 | 2.07 | 2.03 | 1.96 | 1.96 | 1.94 | 1.94 |
| BSDM6x | - | 4.79 | 3.02 | 2.43 | 2.15 | 2.04 | 2.01 | 2.00 | 1.98 | 2.00 |
| BSDM7x | - | 6.85 | 3.10 | 2.35 | 2.07 | 1.96 | 1.92 | 1.92 | 1.92 | 1.97 |
| BSDM2x-w2 | 8.94 | 7.25 | 6.57 | 5.82 | 5.39 | 5.03 | 4.70 | 4.48 | 4.24 | 4.09 |
| BSDM2x-w4 | 9.51 | 7.27 | 6.66 | 5.91 | 5.52 | 5.08 | 4.66 | 4.42 | 4.19 | 4.05 |
| BSDM4x-w2 | 10.22 | 3.22 | 2.54 | **2.16** | 2.01 | 1.96 | 1.91 | 1.89 | 1.86 | 1.90 |
| BSDM4x-w4 | 12.72 | 3.59 | 2.66 | 2.25 | 2.11 | 2.02 | 1.96 | 1.94 | 1.92 | 1.95 |
| BSDM6x-w2 | - | 5.94 | 3.08 | 2.40 | 2.14 | 2.01 | 1.99 | 1.95 | 1.91 | 1.97 |
| BSDM6x-w4 | - | 9.32 | 3.30 | 2.36 | 2.07 | 1.88 | 1.85 | **1.83** | **1.81** | **1.84** |
| BSDM7x-w2 | - | 8.41 | 3.18 | 2.29 | **1.99** | **1.83** | **1.82** | **1.83** | **1.81** | **1.84** |
| BSDM7x-w4 | - | 7.47 | 3.01 | 2.30 | 2.02 | 1.86 | 1.85 | **1.83** | 1.82 | 1.88 |

- EPSM: the Exact Packed String Matching algorithm [12];
- TSO$q$: the Two-Way variant of [5] the Shift-Or algorithm [1] implemented with a loop unrolling of $q$ characters, with $q = 5$;

All algorithms have been implemented in the C programming language and have been tested using the SMART tool[3]. The experiments were executed locally on an MacBook Pro with 4 Cores, a 2 GHz Intel Core i7 processor, 16 GB RAM 1600 MHz DDR3, 256 KB of L2 Cache and 6 MB of Cache L3. Algorithms have been compared in terms of running times, including any preprocessing time.

For the evaluation we use the genome sequence provided by the SMART research tool. Specifically it is a sequence of $4,638,690$ base pairs of *Escherichia coli*, maintained by the Large Canterbury Corpus.[4] In all cases the patterns were

[3] SMART, a String Matching Algorithms Research Tool, by Simone Faro and Thierry Lecroq, `http://www.dmi.unict.it/~faro/smart`

[4] `http://www.data-compression.info/Corpora/CanterburyCorpus/`

randomly extracted from the text and the value $m$ was made ranging from 4 to 2048. For each case we reported the mean over the running times of 500 runs.

From experimental results reported in Table 2 it turns out that the BSDM$q$ algorithm obtains the best results in almost all the cases. In particular the best running times are obtained with a value of $q$ equal to 7 (for long patterns) and 4 (for short patterns).

Comparing the new presented algorithms against the previous known solutions we can observe that the new BSDM$q$-W algorithms are the fastest in most cases, especially for long patterns. We can observe moreover that when the length of the pattern gets longer, better results are obtained for greater values of $q$. The same observation could be done for the number of windows used during the searching.

Specifically the BSDM algorithm implemented with 7-grams and two sliding windows obtains the best results when the pattern is longer than 32 characters. In this cases the BSDM$q$-W is up to 10% faster than BSDM$q$ and up to 13% faster than all previous known algorithms.

When the size of the pattern is between 8 and 32 the new BSDM$q$X algorithms obtain the best results, using 4-grams. In this cases the BSDM4X algorithm is up to $1,2$ times faster than the best among the previous solutions (the SBNDM$q$ algorithm).

For small patterns ($m = 4$) the best running time is obtained by the EBOM algorithm, even if we have to report the good performance of the TSO5 algorithm, although it's not able to report the positions of the found occurrences. However, it is interesting to observe that when $m = 4$ the BSDM4X algorithm obtains a result which is very close to the best running time.

## 5 Conclusions and Future Works

In this paper we reviewed the most recent and efficient solutions for searching exact matching on genome sequences. We also compared such solutions in terms of running times in order to identify the best solutions for such problem. In addition we also propose some efficient variants of the BSDM algorithm which turn out to be competitive with the previous solutions and obtain the best running times in most practical cases.

From experimental results it turns out that the new presented variants obtain the best results in most practical cases when tested join real genome sequences.

It will be interesting to investigate if some of the efficient solutions described above could be generalized also in the case of multiple pattern matching on genome sequences.

## Acknowledgments

This work has been supported by G.N.C.S., Istituto Nazionale di Alta Matematica "Francesco Severi".

## References


1. Baeza-Yates, R., Gonnet, G.H.: A new approach to text searching. Commun. ACM 35(10), 74–82 (1992)
2. Cantone, D., Faro, S., Giaquinta, E.: A compact representation of nondeterministic (suffix) automata for the bit-parallel approach. In: Proceedings of the Annual Symposium on Combinatorial Pattern Matching. pp. 288–298. LNCS 6129 (2010)
3. Cantone, D., Faro, S., Giaquinta, E.: A compact representation of nondeterministic (suffix) automata for the bit-parallel approach. Inform. Comput. 213, 3–12 (2012)
4. Crochemore, M., Rytter, W.: Text algorithms. Oxford University Press (1994)
5. Durian, B., Chhabra, T., Ghuman, S., Hirvola, T., Peltola, H., Tarhio, J.: Improved two-way bit-parallel search. In: Stringology Conference 2014. pp. 71–83 (2014)
6. Durian, B., Peltola, H., Salmela, L., Tarhio, J.: Bit-parallel search algorithms for long patterns. In: Symp. on Experim. Alg. pp. 129–140. LNCS 6049 (2010)
7. Faro, S., Lecroq, T.: Efficient variants of the Backward-Oracle-Matching algorithm. In: Prague Stringology Conference. pp. 146–160. Czech Technical University in Prague, Czech Republic (2008)
8. Faro, S., Lecroq, T.: The exact string matching problem: a comprehensive experimental evaluation. CoRR abs/1012.2547 (2010)
9. Faro, S., Lecroq, T.: A fast suffix automata based algorithm for exact online string matching. In: Proceedings of the 17th International Conference on Implementation and Application of Automata. LNCS, vol. 6049, pp. 149–158 (2012)
10. Faro, S., Lecroq, T.: A multiple sliding windows approach to speed up string matching slgorithms. In: Proceedings of the 11th International Symposium on Experimental Algorithms. LNCS, vol. 7276, pp. 172–183 (2012)
11. Faro, S., Lecroq, T.: The exact online string matching problem: a review of the most recent results. ACM Computing Surveys 45(2), 13 (2013)
12. Faro, S., Kulekci, M.O.: Fast and flexible packed string matching. Journal of Discrete Algorithms 28, 61 – 72 (2014)
13. Kalsi, P., Peltola, H., Tarhio, J.: Bioinformatics Research and Development, chap. Comparison of exact string matching algorithms for biological sequences, pp. 417–426. Springer Berlin Heidelberg (2008)
14. Kouzinopoulos, C.S., Michailidis, P.D., Margaritis, K.G.: Experimental results on multiple pattern matching algorithms for biological sequences. Bioinformatics pp. 274–277 (2011)
15. Langmead, B., Trapnell, C., Pop, M., Salzberg, S.: Ultrafast and memory-efficient alignment of short dna sequences to the human genome. Genome Biology 10(3), 5–10 (2009)
16. Lecroq, T.: Fast exact string matching algorithms. Inf. Process. Lett. 102(6), 229–235 (2007)
17. Navarro, G., Raffinot, M.: A bit-parallel approach to suffix automata: fast extended string matching. In: Proceedings of the Annual Symposium on Combinatorial Pattern Matching. pp. 14–33. LNCS 1448 (1998)
18. Navarro, G., Raffinot, M.: Flexible Pattern Matching in Strings: Practical On-line Search Algorithms for Texts and Biological Sequences. Cambridge University Press, New York, NY, USA (2002)
19. Rivals, E., Salmela, L., Kiiskinen, P., Kalsi, P., Tarhio, J.: 9th International Workshop on Algorithms in Bioinformatics, WABI, chap. mpscan: fast localisation of multiple reads in genomes, pp. 246–260. Springer, Berlin, Heidelberg (2009)
20. Wu, S., Manber, U.: A fast algorithm for multi-pattern searching. Report TR-94-17, Depart. of Computer Science, University of Arizona, Tucson, AZ (1994)