# A Very Fast String Matching Algorithm Based on Condensed Alphabets[⋆]

Simone Faro

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

**Abstract.** String matching is the problem of finding all the substrings of a text which correspond to a given pattern. It's one of the most investigated problem in computer science, mainly due to its various applications in many fields. In recent years most solutions to the problem focused on efficiency and flexibility of the searching procedure and effective techniques appeared to speed-up previous solutions. In this paper we present a simple and very efficient algorithm for string matching. It can be seen as an extension of the Skip-Search algorithm to condensed alphabets with the aim of reducing the number of verifications during the searching phase. From our experimental results it turns out that the new variant obtains in most cases the best running time when compared against the most effective algorithms in literature. This makes the new algorithm one of the most flexible solutions in practical cases.

**Keywords:** Exact text analysis, string matching, experimental algorithms, text processing

## 1 Introduction

The *exact string matching problem* is one of the most studied problem in computer science. It consists in finding all the (possibly overlapping) occurrences of an input pattern $x$ in a text $y$, over the same alphabet $\Sigma$ of size $\sigma$. A huge number of solutions have been devised since the 1980s [6, 17] and, in spite of such wide literature, much work has been produced in the last few years, indicating that the need for efficient solutions to this problem is high.

Solutions to such problem can be divided in two classes, *counting* solutions simply counts the number of occurrences of the pattern in the text, while *reporting* solutions are also able to report the exact positions in which the pattern occurs. Solutions in the first class are faster in general. In this paper we are interested in the second class of algorithms.

From a theoretical point of view the exact string matching problem has been extensively studied. If we indicate with $m$ and $n$ the lengths of the pattern and of the text, respectively, the problem can be solved in $\mathcal{O}(n)$ worst case time complexity [20]. However, in many practical cases it is possible to avoid reading all the characters of the text achieving sub-linear performances on the average.

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

The optimal average $\mathcal{O}(\frac{n \log_\sigma m}{m})$ time complexity [23] was reached for the first time by the Backward-DAWG-Matching algorithm [8] (BDM). Interested readers can refer to [6, 14, 17] for a survey of the most efficient solutions to the problem.

In recent years most solutions to the problem focused on efficiency and flexibility and effective techniques appeared to speed-up previous formerly efficient solutions. Among such techniques *bit-parallelism*, *string-packing*, *q-grams*, *filtering* and *hashing* deserve a special mention since they inspired a lot of work. *Filtering* and *hashing* are two techniques particularly relevant in this paper.

Specifically, instead of checking at each position of the text if the pattern occurs, it seems to be more efficient to *filter* positions of the text by checking only if the corresponding content *looks like* the input pattern. When a resemblance is detected a more detailed check is performed.

The first algorithm to take advantage of such technique was the well known Karp-Rabin algorithm [19] in 1987. It uses *hashing* function for computing a fingerprint value of the pattern. Subsequently a fingerprint value for each text substring of length $m$ is computed. Then a naive check at a given position of the text is performed only if the fingerprint value of the corresponding substring is equal to the fingerprint value of the pattern. The overall worst case time complexity of the algorithm is $\mathcal{O}(nm)$ but a linear behavior can be observed on average. The first solution based on filtering and hashing, showing a sub-linear average behavior, was presented by Lecroq in 2007 [21]. The algorithm, named Hash$q$, is simply a generalization of the Boyer-Moore-Horspool algorithm to condensed alphabets. In this case groups of $q$ characters (or $q$-grams) are hashed in a single fingerprint value, generating an extended condensed alphabet. Such condensed alphabet correspond to the set of all fingerprint values generated by all possible combinations of $q$ characters drawn from the original alphabet.

The idea of extending efficient solutions by condensed alphabets has been later extensively adopted in string matching [17]. However, although several algorithms have been proposed in the last decade, the Hash$q$ algorithm is still one of the most effective solutions in practical cases [14].

In this paper we present a simple, yet very efficient, algorithm for the exact string matching problem based on a well known filtering solution, the Skip-Search algorithm [7], extended with condensed alphabets. We will observe how the use of a condensed alphabet allows to drastically reduce the number of verifications of such filtering approach. The worst case time complexity of the algorithm is $\mathcal{O}(nm)$. However, despite its quadratic worst case behavior, we will show in our experimental evaluation that such extension leads to one of the most efficient and flexible algorithms for string matching. Specifically it turns out that the new solution obtains the best results, in terms of running times, in most cases and especially for small alphabets and long patterns.

The paper is organized as follows. In Section 2 we describe in detail the Skip-Search algorithm and its variants. In Section 3 we introduce and analyze the new algorithm based on condensed alphabets. In Section 4 we compare the new presented algorithm against the most effective solutions known in literature. We drawn our conclusions in Section 5.

## 2 The Skip Search and the Alpha Skip Search Algorithms

The Skip Search algorithm is an elegant and efficient solution to the exact pattern matching problem, firstly presented in [7] and subsequently adapted to many other problems and variants of exact pattern matching.

Let $x$ and $y$ be a pattern and a text of length $m$ and $n$, respectively, over a common alphabet $\Sigma$ of size $\sigma$. For each character $c$ of the alphabet, the Skip Search algorithm collects in a bucket $B[c]$ all the positions of that character in the pattern $x$, so that for each $c \in \Sigma$ we have:

$$B[c] = \{i \ : \ 0 \leq i \leq m-1 \text{ and } x[i] = c\}.$$

Plainly, the space and time complexity needed for the construction of the array $B$ of buckets is $\mathcal{O}(m+\sigma)$.

Thus if a character occurs $k$ times in the pattern, there are $k$ corresponding positions in the bucket of the character. Notice that when the pattern is shorter than the alphabet size, some buckets are empty. This observation turns out to be particularly suitable for our purpose.

The search phase of the Skip Search algorithm examines all the characters $y[j]$ in the text at positions $j = km-1$, for $k = 1, 2, \ldots, \lfloor n/m \rfloor$. For each such character $y[j]$, the bucket $B[y[j]]$ allows one to compute the possible positions $h$ of the text in the neighborhood of $j$ at which the pattern could occur.

By performing a character-by-character comparison between $x$ and the substring $y[h \mathinner{..} h+m-1]$ until either a mismatch is found, or all the characters in the pattern $x$ have been considered, it can be tested whether $x$ actually occurs at position $h$ of the text.

The Skip Search algorithm has a quadratic worst-case time complexity, however, as shown in [7], the expected number of text character inspections is $\mathcal{O}(n)$. In addition it is interesting to observe that the Skip Search algorithms performs better in the case of large alphabets since most of the buckets in the array $B$ are empty.

Among the variants of the Skip Search algorithm, the most relevant one for our purposes is the Alpha Skip Search algorithm [7], which collects buckets for substrings of the pattern rather than for its single characters.

During the preprocessing phase of the Alpha Skip Search algorithm, all the factors of length $\ell = \lfloor \log_\sigma m \rfloor$ occurring in the pattern $x$ are arranged in a trie $T_x$, for fast retrieval. In addition, for each leaf $\nu$ of $T_x$ a bucket is maintained which stores the positions in $x$ of the factor corresponding to $\nu$. Provided that the alphabet size is considered as a constant, the worst-case running time of the preprocessing phase is linear.

The searching phase consists in looking into the buckets of the text factors $y[j \mathinner{..} j+\ell-1]$, for all $j = k(m-\ell+1)-1$ such that $1 \leq k \leq \lfloor (n-\ell)/m \rfloor$, and then test, as in the previous case, whether there is an occurrence of the pattern at the indicated positions of the text.

The worst-case time complexity of the searching phase is quadratic, though the expected number of text character comparisons is $\mathcal{O}(n \log_\sigma m/(m - \log_\sigma m))$.

## 3 A new Fast Variant of the Skip-Search Algorithm

In this section we present an efficient extension of the Skip-Search algorithm using condensed alphabets. The resulting algorithm has a quadratic worst case time complexity while on average it shows a sublinear behavior.

Let $x$ be a pattern of length $m$ and let $y$ be a text of length $n$. Moreover suppose both strings are drawn from a common alphabet $\Sigma$ of size $\sigma$ and suppose $q$ is a constant value, with $1 \leq q \leq m$. The algorithm can be divided in a preprocessing and a searching phase.

The preprocessing phase of the algorithm indexes all subsequences of the pattern (of length $q$) in order to be able to locate them during the searching phase. For efficiency reasons, each substring of length $q$ is converted into a numeric value, called *fingerprint*, which is used to index the substring. A fingerprint value ranges in the interval $\{0 \mathinner{.\,.} 2^{\alpha} - 1\}$, for a given bound $\alpha$. In our setting the value $\alpha$ is set to 16, so that a fingerprint can fit into a single 16-bit register.

The procedure FNG for computing the fingerprints is shown in Fig. 1 (on the left). Given a sequence $x$ of length $m$, an index $i$ such that $0 \leq i < m-q$, and two integers $k$ and $q$ such that $kq \leq \alpha$, the procedure FNG computes the fingerprint $v$ of the substring $x[i \mathinner{.\,.} i+q-1]$. Specifically the fingerprint $v$ is computed as

$$v = \sum_{j=0}^{q-1} (x[i+j] \ll kj) \ .$$

Plainly, the time complexity of the procedure FNG is $\mathcal{O}(q)$. Observe that the the fingerprint value is not unique for each substring of length $q$, i.e. two different strings can be associated with the same fingerprint value. The preprocessing phase of the algorithm, which is reported in Fig. 1 (on the left), consists in compiling the fingerprints of all possible substrings of length $q$ contained in the pattern $x$. Thus a fingerprint value $v$, with $0 \leq v < 2^{\alpha}$, is computed for each subsequence $x[i \mathinner{.\,.} i+q-1]$, for $0 \leq i < m-q$. To this purpose a table $F$ of size $2^{\alpha}$ is maintained for storing, for any possible fingerprint value $v$, the set of positions $i$ such that $\textsc{fng}(x, i, q, k) = v$. More precisely, for $0 \leq v < 2^{\alpha}$, we have

$$F[v] = \Big\{ i \mid 0 \leq i < m-q \text{ and } \textsc{fng}(x, i, q, k) = v \Big\}.$$

The preprocessing phase of the algorithm requires some additional space to store the $(m-q)$ possible alignments in the $2^{\alpha}$ locations of the table $F$. Thus, the space requirement of the algorithm is $\mathcal{O}(m-q+2^{\alpha})$ that approximates to $\mathcal{O}(m)$, since $\alpha$ is constant. The first loop of the preprocessing phase just initializes the table $F$, while the second loop is run $(m-q)$ times, which makes the overall time complexity of such phase $\mathcal{O}(m+2^{\alpha})$ that, again, approximates to $\mathcal{O}(m)$.

Along the same line of the Skip Search algorithm, the basic idea of the searching phase is to compute a fingerprint value every $(m-q+1)$ positions of the text $y$ and to check whether the pattern appears in $y$, involving the block $y[j \mathinner{.\,.} j+q-1]$. If the fingerprint value indicates that some of the alignments are possible, then the candidate positions are checked naively for matching.

```
FNG$(x, i, q, k)$
2. $v \leftarrow 0$
3. for $j \leftarrow q-1$ downto $0$ do
4.     $v \leftarrow (v \ll k) + x[i+j]$
5. return $v$

PREPROCESSING$(x, q, m, k)$
1. for $v \leftarrow 0$ to $2^{\alpha}-1$ do
2.     $F[v] \leftarrow \emptyset$
3. for $i \leftarrow 0$ to $m-q$ do
4.     $v \leftarrow$ FNG$(x, i, q, k)$
5.     $F[v] \leftarrow F[v] \cup \{(i+q-1)\}$
6. return $F$

SKIPq$(x, r, y, n, q, k)$
1. $F \leftarrow$Preprocessing$(x, q, m, k)$
2. for $j \leftarrow m-1$ to $n$ step $m-q+1$ do
3.     $v \leftarrow$ FNG$(y, j, q, k)$
4.     for each $i \in F[v]$ do
5.         if $x = y[j-i \,..\, j-i+m-1]$
6.             then output $(j-i)$
```

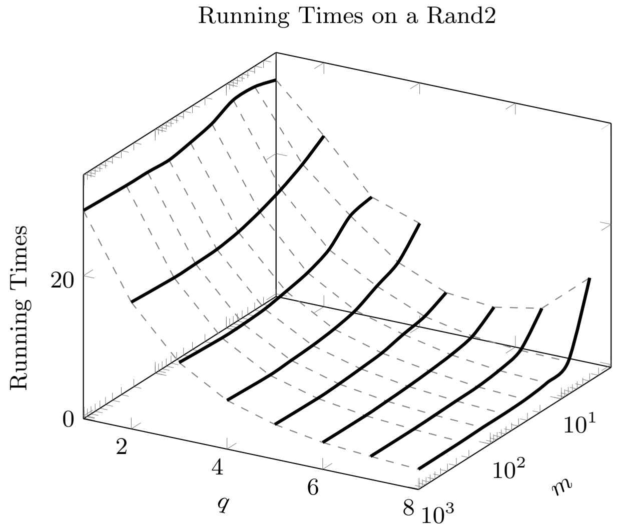
**Fig. 1.** The pseudo-code of the Skip$q$ algorithm for the exact string matching problem.

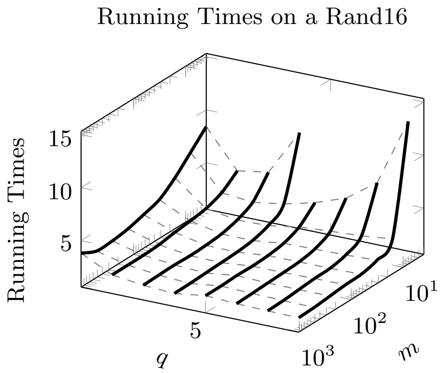
The pseudo-code provided in Fig. 1 (on the right) reports the skeleton of the algorithm. The main loop investigates the blocks of the text $y$ in steps of $(m-q+1)$ blocks. If the fingerprint $v$ computed on $y[j \,..\, j+q-1]$ points to a nonempty bucket of the table $F$, then the positions listed in $F[v]$ are verified accordingly.

In particular $F[v]$ contains a linked list of the values $i$ marking the pattern $x$ and the beginning position of the pattern in the text. While looking for occurrences on $y[j \,..\, j+q-1]$, if $F[v]$ contains the value $i$, this indicates the pattern $x$ may potentially begin at position $(j-i)$ of the text. In that case, a matching test is to be performed between $x$ and $y[j-i \,..\, j-i+m-1]$ via a character-by-character inspection.

The total number of filtering operations is exactly $n/(m-q)$. At each attempt, the maximum number of verification requests is $(m-q)$, since the filter provides information about that number of appropriate alignments of the pattern. On the other hand, if the computed fingerprint points to an empty location in $F$, then there is obviously no need for verification. The verification cost for a pattern $x$ of length $m$ is assumed to be $\mathcal{O}(m)$, with the brute-force checking approach. Hence, in the worst case the time complexity of the verification is $\mathcal{O}(m(m-q))$, which happens when all alignments in $x$ must be verified at any possible beginning position. Hence, the best case complexity is $\mathcal{O}(n/(m-q))$, while the worst case complexity is $\mathcal{O}(nm)$.

Fig.2 shows experimental evaluations to campare the performances of the algorithm under various conditions and for different values of the parameter $q$. Experimental evaluations have been conducted on random text of 4Mb over alphabets of size 2, 16 and 128, respectively, with a uniform distribution of characters (a detailed description of the experimental settings can be found in Section 4).





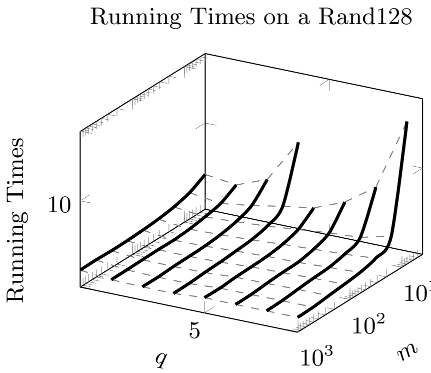


**Fig. 2.** Running times of the Skip-Search extended with condensed alphabets using groups of $q$ characters. We report running times of the algorithms for different values of $q$. Experimental test have been conducted on random text over alphabets of size 2, 16 and 128, respectively, with a uniform distribution of characters.

It turns out from experimental evaluations shown in Fig.2 that the performances of the algorithm strongly depend on the values of $m$, $q$ and $\sigma$. When the size of the alphabet is small then larger values of the parameter $q$ are more effective. Such difference is less sensible when the size of the alphabet gets larger. However it turns out that the smaller is the length of the pattern the lower is the performance of the algorithm. This behavior is more evident for larger values of the parameter $q$. Thus, the choice of the parameter $q$ should be directed to larger values when the size of alphabet decreases or when the length of the pattern increases. Conversely the values of $q$ should get smaller.

## 4 Experimental Results

In this section we evaluate the performance of the new presented algorithms against the most efficient solution known in literature for the online exact string matching problem. Specifically we compare the following 15 algorithms implemented in 79 variants, depending on the values of their parameters:

- AOSO$q$: the Average-Optimal variant [18] of the Shift-Or algorithm [2] using $q$.grams, with $1 \leq q \leq 6$;
- BNDM$q$: the Backward-Nondeterministic-DAWG-Matching algorithm [22] implemented using $q$-grams with $1 \leq q \leq 8$;
- BSDM$q$: the Backward-SNR-DAWG-Matching algorithm [15] using condensed alphabets with groups of $q$ characters, with $1 \leq q \leq 8$;
- BXS$q$: the Backward-Nondeterministic-DAWG-Matching algorithm [22] with Extended Shift [10] implemented using $q$-grams and $1 \leq q \leq 8$;
- EBOM: the extended version [13] of the BOM algorithm [1];
- FSBNDM$qs$: the Forward Simplified version [13] of the BNDM algorithm [22] implemented using $q$-grams and $1 \leq q \leq 8$;
- KBNDM: the Factorized variant [5] BNDM algorithm [22];
- SBNDM$q$: the Simplified version of the Backward-Nondeterministic-DAWG-Matching algorithm [1] implemented using $q$-grams and $1 \leq q \leq 8$;
- FS-$w$: the Multiple Windows version [16] of the Fast Search algorithm [3] implemented using $w$ sliding windows, with $2 \leq w \leq 6$;
- HASH$q$: the Hashing algorithm [21] using $q$-grams, with $3 \leq q \leq 5$;
- IOM: the Improved Occurrence Matcher [4]
- WOM: the Worst Occurrence Matcher [4];
- JOM: the Jumping Occurrence Matcher [4];
- ASKIP the Alpha variant of the SKip-Search algorithm [7];
- SKIP$q$: the new Skip Search variants using $q$-grams, with $1 \leq q \leq 8$ (observe that when $q = 1$ we have the original Skip-Search algorithm [7]);

For the sake of completeness we evaluate also the following two string matching algorithms for *counting* occurrences.

- EPSM: the Exact Packed String Matching algorithm [12];
- TSO$q$: the Two-Way variant of [9] the Shift-Or algorithm [2] 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[1]. 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.

We report experimental evaluations on a three random sequences (see Tables 1 and 2 ) and on three real data (see Tables 3, 4 and 5 ). Specifically random sequences are over alphabets of 2 and 16 characters, with a uniform distribution.

[1] The SMART tool is available online at `http://www.dmi.unict.it/~faro/smart/`.

For the case of real data evaluations we used a genome sequence, a protein sequence and an english text. All sequences have a length of 5MB, are provided by theSMART research tool and are available online for download.

During the experimental evaluations patterns of length $m$ were randomly extracted from the sequences, with $m$ ranging over the set of values $\{2^i \mid 2 \leq i \leq 10\}$. For each case, the mean over the running times, expressed in hundredths of seconds, of 500 runs has been reported.

In the following tables we report the running times of our evaluations. Each table is divided in four blocks. The first and the second block present the most effective algorithms known in literature based on automata and comparison of characters, respectively. Best results among this two sets of algorithms have been bold-faced in order to easily locate the best solutions among previous known algorithms. The third block contains the running times of the new algorithm, including the speed up (in percentage) obtained against the best running time in the first two blocks. Positive values indicate a braking of the running time while a negative percentage represent and improvements of the performance. Running times with an improvement of the performance have been bold-faced.

The last block reports the running times obtained by the best two algorithms for *counting* occurrences (we do not compare them against the other algorithms).

Among the previous solutions it turns out that the BSDM$q$ algorithm is fastest in the case of small alphabets (2 and 16 characters), however it is second to the Hash$q$ algorithm for $\sigma = 2$ and very long patterns, and to the EBOM algorithm for $\sigma = 16$ and short patterns. Regarding the performance of the new algorithm, it obtains always the best results in the case of small alphabets. In such cases the gain in performance is up to 7%. When the size of the alphabet increases the new algorithm maintain the best results only in the case of medium and long patterns (i.e. for $m \geq 32$).

The same behavior can be observed in the case of real data experimental results, where the new solution obtains the best running time in most cases. In the case of a genome sequence it is always the best choice, with a gain in performance up to 12%. Such gain is less evident when the size of the alphabet increases. It is up to 2.5% when searching protein sequences and natural language texts.

## 5 Conclusions

In this paper we presented a simple, yet efficient, variant of the Skip-Search algorithm, based on condensed alphabets. Although such extension has been applied to many algorithms in recent years, it turns out that when applied to the Skip-Search algorithm, it produces a very fast searching procedure. It will be interesting to investigate whether the use of multiple hash function can reduce the number of false positives detected during the filtering phase.

**Table 1.** Experimental results on a random sequence over an alphabet of 2 characters.

| $m$ | 4 | 8 | 16 | 32 | 64 | 128 | 256 | 512 | 1024 |
|---|---|---|---|---|---|---|---|---|---|
| AOSO$q$ | 41.70$^{(2)}$ | 35.62$^{(4)}$ | 14.37$^{(4)}$ | 4.54$^{(4)}$ | 4.64$^{(4)}$ | 4.59$^{(4)}$ | 4.58$^{(4)}$ | 4.62$^{(4)}$ | 4.57$^{(4)}$ |
| BNDM$q$ | 16.55$^{(4)}$ | 8.67$^{(6)}$ | 5.10$^{(6)}$ | 4.06$^{(6)}$ | 4.92$^{(4)}$ | 4.92$^{(4)}$ | 4.92$^{(4)}$ | 4.89$^{(4)}$ | 4.90$^{(4)}$ |
| BSDM$q$ | **15.35**$^{(4)}$ | **7.52**$^{(6)}$ | **4.30**$^{(8)}$ | **3.30**$^{(8)}$ | **2.99**$^{(8)}$ | **2.79**$^{(8)}$ | **2.78**$^{(8)}$ | 2.74$^{(8)}$ | 2.73$^{(8)}$ |
| BXS$q$ | 22.33$^{(4)}$ | 10.97$^{(6)}$ | 5.47$^{(8)}$ | 3.74$^{(8)}$ | 3.75$^{(8)}$ | 3.75$^{(8)}$ | 3.76$^{(8)}$ | 3.74$^{(8)}$ | 3.75$^{(8)}$ |
| EBOM | 25.12 | 20.32 | 13.65 | 8.72 | 6.27 | 4.62 | 3.77 | 3.28 | 2.98 |
| FSBNDM$qs$ | 16.66$^{(4,1)}$ | 7.80$^{(6,1)}$ | 5.08$^{(6,1)}$ | 4.05$^{(6,1)}$ | 4.00$^{(6,1)}$ | 4.07$^{(6,1)}$ | 4.05$^{(6,1)}$ | 4.05$^{(6,1)}$ | 4.05$^{(6,1)}$ |
| KBNDM | 27.67 | 19.08 | 11.31 | 7.07 | 5.82 | 5.71 | 5.75 | 5.78 | 5.79 |
| SBNDM$q$ | 15.36$^{(4)}$ | 8.33$^{(6)}$ | 5.05$^{(6)}$ | 4.01$^{(6)}$ | 4.05$^{(6)}$ | 4.08$^{(6)}$ | 4.07$^{(6)}$ | 4.08$^{(6)}$ | 4.09$^{(6)}$ |
| FS-$w$ | 26.30$^{(2)}$ | 20.20$^{(2)}$ | 15.16$^{(2)}$ | 11.63$^{(2)}$ | 9.42$^{(2)}$ | 8.00$^{(2)}$ | 6.90$^{(2)}$ | 6.11$^{(2)}$ | 5.52$^{(2)}$ |
| FJS | 28.78 | 33.80 | 36.59 | 34.94 | 36.25 | 36.02 | 36.51 | 36.36 | 36.55 |
| HASH$q$ | 25.44$^{(3)}$ | 11.94$^{(5)}$ | 6.07$^{(5)}$ | 3.99$^{(8)}$ | 3.20$^{(8)}$ | 2.97$^{(8)}$ | 2.95$^{(8)}$ | **2.71**$^{(8)}$ | **2.62**$^{(8)}$ |
| ASKIP | 34.24 | 22.43 | 13.14 | 6.50 | 4.62 | 3.62 | 3.27 | 3.16 | 3.52 |
| IOM | 23.90 | 24.64 | 26.64 | 26.58 | 26.76 | 26.60 | 26.69 | 26.51 | 26.67 |
| WOM | 30.32 | 26.31 | 23.00 | 20.37 | 17.94 | 16.47 | 14.94 | 13.78 | 12.76 |
| SKIP$q$ | **14.37**$^{(4)}$ | **7.24**$^{(6)}$ | **4.30**$^{(8)}$ | **3.20**$^{(8)}$ | **2.77**$^{(8)}$ | **2.65**$^{(8)}$ | **2.65**$^{(8)}$ | **2.55**$^{(8)}$ | **2.51**$^{(8)}$ |
| *speed-up* | -6.4% | -3.7% | 0.0% | -3.0% | -7.3% | -5.0% | -4.7% | -5.9% | -4.2% |
| EPSM | 6.51 | 8.46 | 3.20 | 2.45 | 2.25 | 2.22 | 2.30 | 2.31 | 2.30 |
| TSO$q$ | 12.67$^{(5)}$ | 10.01$^{(5)}$ | 6.67$^{(5)}$ | 4.39$^{(5)}$ | 3.17$^{(5)}$ | - | - | - | - |

**Table 2.** Experimental results on a random sequence over an alphabet of 16 characters.

| $m$ | 4 | 8 | 16 | 32 | 64 | 128 | 256 | 512 | 1024 |
|---|---|---|---|---|---|---|---|---|---|
| AOSO$q$ | 10.70$^{(2)}$ | 4.29$^{(4)}$ | 3.75$^{(4)}$ | 3.75$^{(4)}$ | 3.07$^{(6)}$ | 3.11$^{(6)}$ | 3.13$^{(6)}$ | 3.09$^{(6)}$ | 3.14$^{(6)}$ |
| BNDM$q$ | 11.99$^{(4)}$ | 4.20$^{(4)}$ | 2.96$^{(4)}$ | 2.38$^{(4)}$ | 2.39$^{(4)}$ | 2.41$^{(4)}$ | 2.38$^{(4)}$ | 2.35$^{(4)}$ | 2.39$^{(4)}$ |
| BSDM$q$ | 4.82$^{(2)}$ | 3.79$^{(4)}$ | 2.67$^{(4)}$ | 2.27$^{(4)}$ | **2.10**$^{(4)}$ | **2.02**$^{(4)}$ | **1.98**$^{(4)}$ | **1.96**$^{(4)}$ | **1.99**$^{(4)}$ |
| BXS$q$ | 6.86$^{(2)}$ | 4.29$^{(2)}$ | 3.08$^{(2)}$ | 2.46$^{(2)}$ | 2.47$^{(4)}$ | 2.50$^{(4)}$ | 2.51$^{(4)}$ | 2.52$^{(4)}$ | 2.48$^{(4)}$ |
| EBOM | **3.75** | **2.92** | **2.51** | **2.25** | 2.16 | 2.19 | 2.16 | 2.16 | 2.30 |
| FSBNDM$qs$ | 4.28$^{(2,0)}$ | 3.21$^{(2,0)}$ | 2.55$^{(3,1)}$ | 2.18$^{(3,1)}$ | 2.20$^{(3,1)}$ | 2.18$^{(3,1)}$ | 2.19$^{(3,1)}$ | 2.19$^{(3,1)}$ | 2.21$^{(3,1)}$ |
| KBNDM | 7.38 | 4.94 | 3.76 | 3.17 | 2.95 | 3.04 | 2.90 | 3.00 | 2.98 |
| SBNDM$q$ | 5.26$^{(2)}$ | 3.62$^{(2)}$ | 2.73$^{(2)}$ | 2.31$^{(2)}$ | 2.36$^{(4)}$ | 2.37$^{(4)}$ | 2.37$^{(4)}$ | 2.39$^{(4)}$ | 2.42$^{(4)}$ |
| FS-$w$ | 5.15$^{(6)}$ | 3.69$^{(6)}$ | 3.05$^{(6)}$ | 2.72$^{(6)}$ | 2.72$^{(6)}$ | 2.68$^{(6)}$ | 2.64$^{(6)}$ | 2.64$^{(6)}$ | 2.66$^{(6)}$ |
| FJS | 9.38 | 7.08 | 5.27 | 4.94 | 4.65 | 4.54 | 4.18 | 4.65 | 4.49 |
| HASH$q$ | 19.81$^{(3)}$ | 8.35$^{(3)}$ | 5.07$^{(3)}$ | 3.61$^{(5)}$ | 3.10$^{(5)}$ | 2.97$^{(5)}$ | 2.89$^{(5)}$ | 2.73$^{(5)}$ | 2.62$^{(5)}$ |
| ASKIP | 9.04 | 7.48 | 4.98 | 3.35 | 2.84 | 2.82 | 3.07 | 3.76 | 6.23 |
| IOM | 8.96 | 6.40 | 5.11 | 4.49 | 4.35 | 4.28 | 4.28 | 4.31 | 4.35 |
| WOM | 9.39 | 6.66 | 5.16 | 4.43 | 4.17 | 3.97 | 3.85 | 3.80 | 3.72 |
| SKIP$q$ | 4.85$^{(2)}$ | 3.70$^{(3)}$ | 3.13$^{(3)}$ | **2.24**$^{(4)}$ | **2.06**$^{(4)}$ | **1.99**$^{(4)}$ | **1.93**$^{(4)}$ | **1.86**$^{(4)}$ | **1.84**$^{(4)}$ |
| *speed-up* | +29% | +26% | +24% | -0.4% | -1.9% | -1.5% | -2.5% | -5.1% | -7.5% |
| EPSM | 6.62 | 25.55 | 2.72 | 2.10 | 1.94 | 1.89 | 1.81 | 1.74 | 1.78 |
| TSO$q$ | 5.45$^{(5)}$ | 3.88$^{(5)}$ | 3.11$^{(5)}$ | 2.51$^{(5)}$ | 2.20$^{(5)}$ | - | - | - | - |

**Table 3.** Experimental results on a genome sequence.

| $m$ | 4 | 8 | 16 | 32 | 64 | 128 | 256 | 512 | 1024 |
|---|---|---|---|---|---|---|---|---|---|
| AOSO$q$ | 16.98$^{(2)}$ | 9.63$^{(2)}$ | 3.93$^{(4)}$ | 3.39$^{(4)}$ | 2.98$^{(6)}$ | 2.97$^{(6)}$ | 2.99$^{(6)}$ | 3.00$^{(6)}$ | 3.03$^{(6)}$ |
| BNDM$q$ | 11.13$^{(4)}$ | 4.10$^{(4)}$ | 2.99$^{(4)}$ | 2.47$^{(4)}$ | 2.38$^{(4)}$ | 2.39$^{(4)}$ | 2.41$^{(4)}$ | 2.47$^{(4)}$ | 2.45$^{(4)}$ |
| BSDM$q$ | 8.37$^{(4)}$ | $\underline{\mathbf{3.71}^{(4)}}$ | $\underline{\mathbf{2.78}^{(4)}}$ | $\underline{\mathbf{2.46}^{(4)}}$ | $\underline{\mathbf{2.25}^{(8)}}$ | $\underline{\mathbf{2.15}^{(8)}}$ | $\underline{\mathbf{2.11}^{(8)}}$ | $\underline{\mathbf{2.16}^{(6)}}$ | $\underline{\mathbf{2.11}^{(6)}}$ |
| BXS$q$ | 11.86$^{(2)}$ | 4.78$^{(4)}$ | 3.25$^{(4)}$ | 2.53$^{(6)}$ | 2.50$^{(6)}$ | 2.52$^{(4)}$ | 2.49$^{(4)}$ | 2.55$^{(4)}$ | 2.54$^{(4)}$ |
| EBOM | 7.72 | 7.15 | 5.66 | 4.10 | 3.17 | 2.67 | 2.40 | 2.32 | 2.41 |
| FSBNDM$qs$ | $\underline{\mathbf{6.46}^{(3,1)}}$ | 3.87$^{(4,1)}$ | 2.94$^{(4,1)}$ | 2.38$^{(4,1)}$ | 2.35$^{(6,2)}$ | 2.31$^{(6,1)}$ | 2.33$^{(6,1)}$ | 2.38$^{(3,1)}$ | 2.37$^{(6,1)}$ |
| KBNDM | 10.88 | 8.21 | 6.15 | 4.17 | 3.27 | 3.09 | 3.10 | 3.13 | 3.14 |
| SBNDM$q$ | 8.75$^{(2)}$ | 3.95$^{(4)}$ | 2.97$^{(4)}$ | 2.47$^{(4)}$ | 2.39$^{(4)}$ | 2.39$^{(4)}$ | 2.36$^{(4)}$ | 2.38$^{(4)}$ | 2.38$^{(4)}$ |
| FS-$w$ | 12.33$^{(2)}$ | 9.39$^{(2)}$ | 7.76$^{(2)}$ | 6.89$^{(2)}$ | 6.16$^{(2)}$ | 5.63$^{(2)}$ | 5.06$^{(2)}$ | 4.73$^{(2)}$ | 4.42$^{(2)}$ |
| FJS | 18.60 | 16.69 | 16.96 | 15.96 | 16.09 | 16.80 | 16.71 | 16.61 | 16.59 |
| HASH$q$ | 18.09$^{(3)}$ | 7.68$^{(3)}$ | 4.67$^{(5)}$ | 3.31$^{(5)}$ | 2.78$^{(5)}$ | 2.60$^{(5)}$ | 2.63$^{(5)}$ | 2.51$^{(5)}$ | 2.40$^{(5)}$ |
| ASKIP | 17.19 | 7.45 | 4.32 | 3.17 | 2.66 | 2.60 | 2.79 | 3.24 | 5.18 |
| IOM | 14.41 | 11.88 | 11.08 | 11.17 | 11.17 | 11.13 | 11.03 | 11.03 | 10.98 |
| WOM | 16.69 | 12.48 | 9.88 | 8.61 | 7.75 | 7.16 | 6.72 | 6.29 | 6.11 |
| SKIP$q$ | $\underline{\mathbf{6.02}^{(3)}}$ | $\underline{\mathbf{3.71}^{(4)}}$ | $\underline{\mathbf{2.76}^{(4)}}$ | **2.34**$^{(4)}$ | $\underline{\mathbf{2.15}^{(4)}}$ | $\underline{\mathbf{2.08}^{(6)}}$ | $\underline{\mathbf{2.01}^{(8)}}$ | $\underline{\mathbf{1.91}^{(8)}}$ | $\underline{\mathbf{1.88}^{(8)}}$ |
| *speed-up* | -6.8% | 0.0% | -0.7% | -4.9% | -4.4% | -3.3% | -4.7% | -12% | -11% |
| EPSM | 5.87 | 3.72 | 2.50 | 1.93 | 1.75 | 1.72 | 1.66 | 1.62 | 1.65 |
| TSO$q$ | 5.54$^{(5)}$ | 3.85$^{(5)}$ | 3.08$^{(5)}$ | 2.42$^{(5)}$ | 2.05$^{(5)}$ | - | - | - | - |

**Table 4.** Experimental results on a protein sequence.

| $m$ | 4 | 8 | 16 | 32 | 64 | 128 | 256 | 512 | 1024 |
|---|---|---|---|---|---|---|---|---|---|
| AOSO$q$ | 10.80$^{(2)}$ | 4.27$^{(4)}$ | 3.84$^{(4)}$ | 3.81$^{(4)}$ | 3.18$^{(4)}$ | 3.17$^{(4)}$ | 3.16$^{(4)}$ | 3.16$^{(4)}$ | 3.16$^{(4)}$ |
| BNDM$q$ | 12.20$^{(4)}$ | 4.29$^{(4)}$ | 3.06$^{(4)}$ | 2.46$^{(4)}$ | 2.45$^{(4)}$ | 2.43$^{(4)}$ | 2.42$^{(4)}$ | 2.40$^{(4)}$ | 2.40$^{(4)}$ |
| BSDM$q$ | 4.68$^{(2)}$ | 3.71$^{(2)}$ | 2.75$^{(4)}$ | 2.35$^{(4)}$ | **2.06**$^{(4)}$ | $\underline{\mathbf{1.98}^{(4)}}$ | $\underline{\mathbf{1.97}^{(4)}}$ | $\underline{\mathbf{1.97}^{(4)}}$ | $\underline{\mathbf{1.94}^{(4)}}$ |
| BXS$q$ | 6.91$^{(2)}$ | 4.29$^{(2)}$ | 3.12$^{(2)}$ | 2.52$^{(2)}$ | 2.48$^{(2)}$ | 2.52$^{(2)}$ | 2.50$^{(2)}$ | 2.51$^{(2)}$ | 2.52$^{(2)}$ |
| EBOM | $\underline{\mathbf{3.87}}$ | $\underline{\mathbf{2.94}}$ | $\underline{\mathbf{2.57}}$ | $\underline{\mathbf{2.29}}$ | 2.11 | 2.18 | 2.20 | 2.24 | 2.42 |
| FSBNDM$qs$ | 4.32$^{(2,0)}$ | 3.28$^{(2,0)}$ | 2.59$^{(3,1)}$ | 2.26$^{(3,1)}$ | 2.22$^{(3,1)}$ | 2.25$^{(3,1)}$ | 2.25$^{(3,1)}$ | 2.20$^{(3,1)}$ | 2.26$^{(3,1)}$ |
| KBNDM | 7.46 | 4.97 | 3.81 | 3.24 | 3.04 | 3.01 | 2.95 | 2.96 | 2.95 |
| SBNDM$q$ | 5.25$^{(2)}$ | 3.67$^{(2)}$ | 2.79$^{(2)}$ | 2.34$^{(2)}$ | 2.45$^{(4)}$ | 2.41$^{(4)}$ | 2.42$^{(4)}$ | 2.41$^{(4)}$ | 2.40$^{(4)}$ |
| FS-$w$ | 6.18$^{(2)}$ | 4.33$^{(2)}$ | 3.55$^{(2)}$ | 3.20$^{(2)}$ | 3.05$^{(2)}$ | 2.94$^{(2)}$ | 2.90$^{(2)}$ | 2.87$^{(2)}$ | 2.86$^{(2)}$ |
| FJS | 9.68 | 18.54 | 4.18 | 3.02 | 2.92 | 2.89 | 2.82 | 3.16 | 4.11 |
| HASH$q$ | 19.92$^{(3)}$ | 8.36$^{(3)}$ | 5.05$^{(3)}$ | 3.75$^{(5)}$ | 3.19$^{(5)}$ | 2.99$^{(5)}$ | 2.92$^{(5)}$ | 2.76$^{(5)}$ | 2.66$^{(5)}$ |
| ASKIP | 8.63 | 7.06 | 5.07 | 3.52 | 2.87 | 2.87 | 3.13 | 3.92 | 6.56 |
| IOM | 8.87 | 6.36 | 5.02 | 4.41 | 4.04 | 3.92 | 3.86 | 3.86 | 3.79 |
| WOM | 9.31 | 6.61 | 5.13 | 4.32 | 4.03 | 3.72 | 3.56 | 3.43 | 3.33 |
| SKIP$q$ | 4.56$^{(2)}$ | 3.41$^{(3)}$ | 2.64$^{(3)}$ | $\underline{\mathbf{2.26}^{(3)}}$ | $\underline{\mathbf{2.00}^{(3)}}$ | $\underline{\mathbf{1.94}^{(4)}}$ | $\underline{\mathbf{1.92}^{(4)}}$ | $\underline{\mathbf{1.90}^{(4)}}$ | $\underline{\mathbf{1.92}^{(4)}}$ |
| *speed-up* | +17% | +16% | +2.7% | -1.3% | -2.2% | -2.0% | -2.5% | -2.5% | -1.0% |
| EPSM | 6.67 | 25.55 | 2.77 | 2.16 | 1.91 | 1.91 | 1.90 | 1.83 | 1.86 |
| TSO$q$ | 5.41$^{(5)}$ | 3.90$^{(5)}$ | 3.29$^{(5)}$ | 2.59$^{(5)}$ | 2.17$^{(5)}$ | - | - | - | - |

**Table 5.** Experimental results on a natural language sequence.

| $m$ | 4 | 8 | 16 | 32 | 64 | 128 | 256 | 512 | 1024 |
|---|---|---|---|---|---|---|---|---|---|
| AOSO$q$ | 11.14$^{(2)}$ | 4.58$^{(4)}$ | 3.89$^{(4)}$ | 3.76$^{(4)}$ | 3.16$^{(6)}$ | 3.16$^{(6)}$ | 3.18$^{(6)}$ | 3.21$^{(6)}$ | 3.16$^{(6)}$ |
| BNDM$q$ | 12.30$^{(4)}$ | 4.35$^{(4)}$ | 3.17$^{(4)}$ | 2.49$^{(4)}$ | 2.53$^{(4)}$ | 2.52$^{(4)}$ | 2.51$^{(4)}$ | 2.54$^{(4)}$ | 2.51$^{(4)}$ |
| BSDM$q$ | 4.73$^{(2)}$ | 3.85$^{(2)}$ | **2.86**$^{(4)}$ | **2.35**$^{(4)}$ | **2.20**$^{(4)}$ | **2.09**$^{(4)}$ | **2.07**$^{(4)}$ | **2.02**$^{(4)}$ | **2.00**$^{(4)}$ |
| BXS$q$ | 7.38$^{(2)}$ | 4.85$^{(2)}$ | 3.43$^{(4)}$ | 2.59$^{(4)}$ | 2.59$^{(4)}$ | 2.64$^{(4)}$ | 2.62$^{(4)}$ | 2.62$^{(4)}$ | 2.63$^{(4)}$ |
| EBOM | **4.33** | **3.47** | 3.05 | 2.74 | 2.54 | 2.51 | 2.40 | 2.40 | 2.57 |
| FSBNDM$qs$ | 4.66$^{(2,0)}$ | 3.55$^{(3,1)}$ | 2.77$^{(3,1)}$ | 2.39$^{(3,1)}$ | 2.39$^{(3,1)}$ | 2.38$^{(3,1)}$ | 2.41$^{(3,1)}$ | 2.42$^{(3,1)}$ | 2.43$^{(3,1)}$ |
| KBNDM | 7.84 | 5.49 | 4.22 | 3.59 | 3.28 | 3.08 | 3.04 | 3.03 | 3.03 |
| SBNDM$q$ | 5.75$^{(2)}$ | 4.18$^{(2)}$ | 3.13$^{(4)}$ | 2.43$^{(4)}$ | 2.52$^{(4)}$ | 2.50$^{(4)}$ | 2.52$^{(4)}$ | 2.51$^{(4)}$ | 2.52$^{(4)}$ |
| FS-$w$ | 6.05$^{(6)}$ | 4.25$^{(6)}$ | 3.39$^{(6)}$ | 2.89$^{(6)}$ | 2.73$^{(6)}$ | 2.54$^{(6)}$ | 2.43$^{(6)}$ | 2.40$^{(6)}$ | 2.39$^{(6)}$ |
| FJS | 7.06 | 25.33 | 3.68 | 2.95 | 2.96 | 2.81 | 3.18 | 3.42 | 3.83 |
| HASH$q$ | 19.96$^{(3)}$ | 8.34$^{(3)}$ | 5.02$^{(3)}$ | 3.68$^{(5)}$ | 3.17$^{(5)}$ | 2.95$^{(5)}$ | 2.96$^{(5)}$ | 2.76$^{(5)}$ | 2.65$^{(5)}$ |
| ASKIP | 10.17 | 7.78 | 5.25 | 3.65 | 2.97 | 2.89 | 3.14 | 3.77 | 5.90 |
| IOM | 9.37 | 6.67 | 5.26 | 4.38 | 3.96 | 3.73 | 3.47 | 3.30 | 3.20 |
| WOM | 9.98 | 7.01 | 5.28 | 4.32 | 3.91 | 3.53 | 3.25 | 3.11 | 3.02 |
| SKIP$q$ | 4.62$^{(2)}$ | 3.58$^{(3)}$ | **2.78**$^{(3)}$ | **2.30**$^{(3)}$ | **2.13**$^{(4)}$ | **2.07**$^{(4)}$ | **2.06**$^{(4)}$ | **1.97**$^{(4)}$ | **1.96**$^{(4)}$ |
| *speed-up* | +6.7% | -3.2% | -2.8% | -2.1% | -3.2% | -1.0% | -0.5% | -2.5% | -2.0% |
| EPSM | 6.72 | 26.36 | 2.86 | 2.13 | 1.94 | 1.94 | 1.92 | 1.86 | 1.87 |
| TSO$q$ | 5.54$^{(5)}$ | 4.05$^{(5)}$ | 3.26$^{(5)}$ | 2.61$^{(5)}$ | 2.23$^{(5)}$ | - | - | - | - |

## References

1. Allauzen, C., Crochemore, M., Raffinot, M.: Factor oracle: A new structure for pattern matching. In: Pavelka, J., Tel, G., Bartosek, M. (eds.) SOFSEM '99, Theory and Practice of Informatics, 26th Conference on Current Trends in Theory and Practice of Informatics, Milovy, Czech Republic, November 27 - December 4, 1999, Proceedings. Lecture Notes in Computer Science, vol. 1725, pp. 295–310. Springer (1999), `http://dx.doi.org/10.1007/3-540-47849-3_18`
2. Baeza-Yates, R., Gonnet, G.H.: A new approach to text searching. Commun. ACM 35(10), 74–82 (Oct 1992), `http://doi.acm.org/10.1145/135239.135243`
3. Cantone, D., Faro, S.: Fast-search algorithms: New efficient variants of the boyer-moore pattern-matching algorithm. Journal of Automata, Languages and Combinatorics 10(5/6), 589–608 (2005)
4. Cantone, D., Faro, S.: Improved and self-tuned occurrence heuristics. In: Holub, J., Zdárek, J. (eds.) Proceedings of the Prague Stringology Conference 2013, Prague, Czech Republic, September 2-4, 2013. pp. 92–106. Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague (2013), `http://www.stringology.org/event/2013/p09.html`
5. Cantone, D., Faro, S., Giaquinta, E.: A compact representation of nondeterministic (suffix) automata for the bit-parallel approach. Inf. Comput. 213, 3–12 (2012), `http://dx.doi.org/10.1016/j.ic.2011.03.006`
6. Charras, C., Lecroq, T.: Handbook of Exact String Matching Algorithms. College Publications (2004)
7. Charras, C., Lecroq, T., Pehoushek, J.D.: A very fast string matching algorithm for small alphabeths and long patterns (extended abstract). In: Farach-Colton [11], pp. 55–64, `http://dx.doi.org/10.1007/BFb0030780`

8. Crochemore, M., Czumaj, A., Gasieniec, L., Jarominek, S., Lecroq, T., Plandowski, W., Rytter, W.: Speeding up two string-matching algorithms. Algorithmica 12(4/5), 247–267 (1994), http://dx.doi.org/10.1007/BF01185427
9. Durian, B., Chhabra, T., Ghuman, S.S., Hirvola, T., Peltola, H., Tarhio, J.: Improved two-way bit-parallel search. In: Holub, J., Zdárek, J. (eds.) Proceedings of the Prague Stringology Conference 2014, Prague, Czech Republic, September 1-3, 2014. pp. 71–83. Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague (2014)
10. Durian, B., Peltola, H., Salmela, L., Tarhio, J.: Bit-parallel search algorithms for long patterns. In: Festa, P. (ed.) Experimental Algorithms, 9th International Symposium, SEA 2010, Ischia Island, Naples, Italy, May 20-22, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6049, pp. 129–140. Springer (2010)
11. Farach-Colton, M. (ed.): Combinatorial Pattern Matching, 9th Annual Symposium, CPM 98, Piscataway, New Jersey, USA, July 20-22, 1998, Proceedings, Lecture Notes in Computer Science, vol. 1448. Springer (1998)
12. Faro, S., Külekci, M.O.: Fast and flexible packed string matching. J. Discrete Algorithms 28, 61–72 (2014), http://dx.doi.org/10.1016/j.jda.2014.07.003
13. Faro, S., Lecroq, T.: Efficient variants of the backward-oracle-matching algorithm. In: Holub, J., Žďárek, J. (eds.) Proceedings of the Prague Stringology Conference 2008. pp. 146–160. Czech Technical University in Prague, Czech Republic (2008)
14. Faro, S., Lecroq, T.: The exact string matching problem: a comprehensive experimental evaluation. CoRR abs/1012.2547 (2010)
15. Faro, S., Lecroq, T.: A fast suffix automata based algorithm for exact online string matching. In: Moreira, N., Reis, R. (eds.) Implementation and Application of Automata - 17th International Conference, CIAA 2012, Porto, Portugal, July 17-20, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7381, pp. 149–158. Springer (2012), http://dx.doi.org/10.1007/978-3-642-31606-7_13
16. Faro, S., Lecroq, T.: A multiple sliding windows approach to speed up string matching algorithms. In: Klasing, R. (ed.) Experimental Algorithms - 11th International Symposium, SEA 2012, Bordeaux, France, June 7-9, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7276, pp. 172–183. Springer (2012)
17. Faro, S., Lecroq, T.: The exact online string matching problem: A review of the most recent results. ACM Comput. Surv. 45(2), 13 (2013), http://doi.acm.org/10.1145/2431211.2431212
18. Fredriksson, K., Grabowski, S.: Practical and optimal string matching. In: Consens, M.P., Navarro, G. (eds.) String Processing and Information Retrieval, 12th International Conference, SPIRE 2005, Buenos Aires, Argentina, November 2-4, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3772, pp. 376–387. Springer (2005), http://dx.doi.org/10.1007/11575832_42
19. Karp, R.M., Rabin, M.O.: Efficient randomized pattern-matching algorithms. IBM Journal of Research and Development 31(2), 249–260 (1987)
20. Knuth, D.E., Morris, Jr, J.H., Pratt, V.R.: Fast pattern matching in strings. SIAM J. Comput. 6(1), 323–350 (1977)
21. Lecroq, T.: Fast exact string matching algorithms. Inf. Process. Lett. 102(6), 229–235 (2007), http://dx.doi.org/10.1016/j.ipl.2007.01.002
22. Navarro, G., Raffinot, M.: A bit-parallel approach to suffix automata: Fast extended string matching. In: Farach-Colton [11], pp. 14–33, http://dx.doi.org/10.1007/BFb0030778
23. Yao, A.C.: The complexity of pattern matching for a random string. SIAM J. Comput. 8(3), 368–387 (1979), http://dx.doi.org/10.1137/0208029