Improved and Self-Tuned Occurrence Heuristics*

Domenico Cantone and Simone Faro

Università di Catania, Dipartimento di Matematica e Informatica
Viale Andrea Doria 6, I-95125 Catania, Italy
{cantone,faro}@dmi.unict.it

Abstract. In this note we present three efficient variations of the *occurrence heuristic*, adopted by many exact string matching algorithms and firstly introduced in the well-known Boyer-Moore algorithm. Our first heuristic, called *improved-occurrence heuristic*, is a simple improvement of the rule introduced by Sunday in his Quick-Search algorithm. Our second heuristic, called *worst-occurrence heuristic*, achieves its speed-up by selecting the relative position which yields the largest average advancement. Finally, our third heuristic, called *jumping-occurrence heuristic*, uses two characters for computing the next shift, whose distance allows one to maximize the average advancement. The worst-occurrence and jumping-occurrence heuristics tune their parameters according to the text characters' distribution. Experimental results show that the new proposed heuristics achieve very good results on average, especially in the case of small alphabets.

Keywords: string matching, experimental algorithms, text-processing, occurrence heuristics, frequency of characters, tuned-search approach

1 Introduction

Given a text t and a pattern p over some alphabet Σ, the *string matching problem* consists in finding *all* occurrences of the pattern p in the text t. In a computational model in which the matching algorithm is restricted to read all the characters of the text one by one, the optimal complexity is $\mathcal{O}(n)$. However, in several practical cases it is not necessary to read all text characters, achieving sublinear performances on average. The optimal average complexity is $\mathcal{O}(n/m \log \sigma)$ [18] and it is interesting to note that many of such algorithms have an even worse $\mathcal{O}(nm)$-time complexity in the worst-case [9,4,5,6,10,11,12].

This is the case for the celebrated Boyer-Moore (BM) algorithm [2], the progenitor of several algorithmic variants which aim at efficiently computing shift increments close to optimal. The Boyer-Moore algorithm computes shift increments as the maximum value suggested by the *good-suffix heuristic* and the *occurrence heuristic*, provided that both of them are applicable. However, many subsequent efficient variants of the Boyer-Moore algorithm just dropped the good-suffix heuristic and based the calculation of the shift increments only on variants of the occurrence heuristic. Some of such variants are still considered among the most efficient algorithms in practical cases (see [9]).

The occurrence heuristic uses a single character for shifting. Specifically, it states that when a mismatch is found at a given position j of the text, then the pattern can be safely shifted in such a way that its rightmost occurrence of the mismatching character in the text, if present, is aligned with the relative position j in the text.

* This work has been partially supported by project PRISMA PON04a2 A/F funded by the Italian Ministry of University within PON 2007-2013 framework.

Domenico Cantone and Simone Faro: Improved and Self-Tuned Occurrence Heuristics, pp. 92–106.
Proceedings of PSC 2013, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05330-0 © Czech Technical University in Prague, Czech Republic

In this paper we present three improvements of the occurrence heuristic which turn out to be more efficient in practical cases, especially in the case of small alphabets. In particular, we will introduce the following heuristics:

1. the *improved-occurrence heuristic*, which is based on the match of the rightmost character of the pattern with the corresponding character in the text;
2. the *worst-occurrence heuristic*, which selects a relative position yielding the largest average advancement according to the text characters' distribution;
3. the *jumping-occurrence heuristic*, which uses two characters for computing the shift advancements in the searching phase. The relative distance between the two characters is computed so as to maximize the average shift advancements, based on the text characters' distribution.

The paper is organized as follows. Some useful notations and terminology are preliminarily recalled in Section 2. Then, in Section 3 we briefly revise the occurrence heuristic and some of its variants. In Section 4 we present the first of our proposed occurrence heuristics, namely the improved-occurrence heuristic, and in Sections 5 and 6 we introduce the worst-occurrence and the jumping-occurrence heuristics, respectively. Finally, in Section 7 we present and comment on experimental results on the performance of our proposed heuristics in comparison with the best known algorithms present in literature based on the occurrence heuristic. Finally, we draw our conclusions in Section 8.

2 Notations and Terminology

A string p of length $|p| = m \geq 0$ over a finite alphabet Σ is represented as a finite array $p[0\,..\,m-1]$. By $p[i]$ we denote the $(i+1)$-st character of p, for $0 \leq i < m$. Likewise, by $p[i\,..\,j]$ we denote the substring of p contained between the $(i+1)$-st and the $(j+1)$-st characters of p, where $0 \leq i \leq j < m$.

Let t be a text of length n and let p be a pattern of length m. If the character $p[0]$ is aligned with the character $t[s]$ of the text, so that $p[i]$ is aligned with $t[s+i]$, for $0 \leq i \leq m-1$, we say that the pattern p has *shift* s in t. In this case, the substring $t[s\,..\,s+m-1]$ is called the *current window* of the text. If $t[s\,..\,s+m-1] = p$, we say that the shift s is *valid*. Then the *string matching problem* consists in finding all valid shifts of p in t, for given pattern p and text t.

In general, most string matching algorithms work as follows. They scan the text by sliding a text window whose size is generally equal to m. For each text window, its characters are compared with the corresponding characters of the pattern or suitable transitions are performed on some kind of automaton (this specific phase is called a *matching attempt*). After a complete match of the pattern is found or a mismatch is detected, the current window is shifted to the right by a certain number of positions. This phase is usually referred to as the *sliding window mechanism*. When the search starts, the left end of the text and of the current window are aligned. Subsequently, the sliding window mechanism is repeated until the right end of the window goes past the right end of the text. Each matching attempt can be naturally associated with the position s in the text where the current window $t[s\,..\,s+m-1]$ is positioned.

3 The Occurrence Heuristic and Some of its Variants

The well-known *occurrence heuristic* was introduced for the first time in [2] as one of the shift rules used by the Boyer-Moore algorithm. The work in [17,8] provides a uniform framework for describing all safe shifts provided by the Boyer-Moore-type pattern matching algorithms. Specifically, during a matching attempt the Boyer-Moore algorithm scans the current window (of the text) from right to left and, at the end of the matching phase, it computes the shift increment as the largest value given by the *good-suffix* and the *occurrence* heuristics.

The occurrence heuristic states that if $c = t[s+i] \neq p[i]$ is the first mismatching character (with $0 \leq i \leq m-1$), while scanning p and t (with shift s) from right to left, then p can be safely shifted in such a way that its rightmost occurrence of c, if present, is aligned with position $(s+i)$ in t (provided that such an occurrence lies in $p[0\,..\,i-1]$, otherwise the occurrence heuristic has no effect). In the case in which c does not occur in p, then p can be safely shifted just past position $(s+i)$ in t. More formally, the shift increment suggested by the occurrence heuristic is given by $(bc_p(t[s+i]) + i - m + 1)$, where, for $c \in \Sigma$, $bc_p(c) =_{\text{Def}} \min(\{k \mid 0 \leq k \leq m-1 \text{ and } p[m-k-1] = c\} \cup \{m\})$.

Observe that the table bc_p of the occurrence heuristic, for a given a pattern p of length m, can be computed in $\mathcal{O}(m+\sigma)$ time and $\mathcal{O}(\sigma)$ space, where σ is the size of the alphabet Σ.

Due to the simplicity and ease of implementation of the occurrence heuristic, some variants of the Boyer-Moore algorithm were based just on it, dropping the good-suffix heuristic. For instance, Horspool [13] suggested the following simplification of the original Boyer-Moore algorithm, which performs better in practical cases. He just dropped the good-suffix heuristic and proposed to compute shift advancements in such a way that the rightmost character $t[s+m-1]$ of the current window is aligned with its rightmost occurrence on $p[0\,..\,m-2]$, if present; otherwise the pattern is advanced just past the window. This amounts to advance the shift by $hbc_p(t[s+m-1])$ positions, where $hbc_p(c) =_{\text{Def}} \min(\{k \mid 1 \leq k \leq m-1 \text{ and } p[m-k-1] = c\} \cup \{m\})$.

The Quick-Search algorithm, presented in [16], also uses a modification of the original occurrence heuristic, much along the same lines of the Horspool algorithm. Specifically, it is based on the following observation: when a mismatching character is encountered, the pattern is always shifted to the right by at least one character, but never by more than m characters. Thus, the character $t[s+m]$ is always involved in testing for the next alignment. So, one can apply the bad character rule to $t[s+m]$, rather than to the mismatching character, possibly obtaining larger shift advancements. This corresponds to advance the shift by $qbc_p(t[s+m])$ positions, where $qbc_p(c) =_{\text{Def}} \min(\{k \mid 1 \leq k \leq m-1 \text{ and } p[m-k] = c\} \cup \{m+1\})$.

Other efficient variants of the Boyer-Moore algorithm extend the previous algorithms in that their occurrence heuristics use two characters rather than just one. For instance the Zhu-Takaoka algorithm [19] extends the Horspool algorithm by using the last two characters $t[s+m-2]$ and $t[s+m-1]$ in place of only $t[s+m-1]$. A more effective algorithm, due to Berry and Ravindran [1], extends the Quick-Search algorithm in a similar manner, by using the characters $t[s+m]$ and $t[s+m+1]$ in place of only $t[s+m]$. It is to be noticed, though, that the precomputation of the table used by an occurrence heuristic based on two text characters requires $\mathcal{O}(\sigma^2)$-space and $\mathcal{O}(m+\sigma^2)$-time complexity.

4 A Simple Improved Occurrence Heuristic

For a given shift s, the Horspool and the Quick-Search algorithms compute their shift advancements by applying the occurrence heuristic on a fixed position $s+q$ of the text, with $q = m-1$ and $q = m$, respectively. We refer to the value q as the *occurrence relative position*.

In favorable conditions, it may be possible to use an occurrence relative position $q > m$, which may lead to even larger advancements, provided that no matching can ever possibly be skipped. In such a situation, we say that the occurrence relative position q is *safe (for shifting)*.

To this purpose, we begin by introducing the *generalized occurrence function* $gbc_p(i,c)$. Suppose the pattern p has shift s in the text t. For a given occurrence relative position $0 \le i \le 2m-1$, $gbc_p(i, t[s+i])$ is the shift advancement such that the character $t[s+i]$ is aligned with its rightmost occurrence in $p[0\,..\,\min(i,m)-1]$, if present; otherwise $gbc_p(i, t[s+i])$ evaluates to $i+1$ (this corresponds to advance the pattern just past position $s+i$ of the text). This amounts to putting

$$gbc_p(i,c) =_{\text{Def}} \min(\{i-k \mid 0 \le k < \min(i,m) \text{ and } p[k] = c\} \cup \{i+1\}),$$

for $c \in \Sigma$ and $i \ge 0$.[1] Plainly, $gbc_p(i,c) \ge 1$ always holds. Additionally, the shift rules of the Horspool and Quick-Search algorithms can be expressed in terms of the generalized occurrence function just defined by $hbc_p(c) = gbc_p(m-1,c)$ and $qbc_p(c) = gbc_p(m,c)$, respectively, for $c \in \Sigma$.

We will define our improved occurrence heuristic (IOH) in terms of the generalized occurrence function $gbc_p(i,c)$. Let again s be the shift of the current text window. We distinguish the following two cases:

Case $p[m-1] = t[s+m-1]$**:**
Let i_0 be the rightmost position in the substring $p[0\,..\,m-2]$ such that $p[i_0] = p[m-1]$, provided that $p[m-1]$ occur in $p[0\,..\,m-2]$; otherwise let i_0 be -1. Then the occurrence relative position $q_1 = 2m-i_0-2$ is safe for shifting, since no occurrence of the character $p[m-1]$ exists from position i_0+1 to position $m-2$. More formally, q_1 can be defined as

$$q_1 =_{\text{Def}} \min(\{2m-i-2 \mid p[i] = p[m-1] \text{ and } 0 \le i \le m-2\} \cup \{2m-1\})\,.$$

Case $p[m-1] \neq t[s+m-1]$**:**
In this case, let i_0 be the rightmost position in $p[0\,..\,m-2]$ such that $p[i_0] \neq p[m-1]$, provided that $p[0\,..\,m-2]$ contain some character distinct from $p[m-1]$, otherwise let i_0 be -1. Then the occurrence relative position $q_2 = 2m-i_0-2$ is safe for shifting, since no character different from $p[m-1]$ exists from position i_0+1 to position $m-2$. More formally, q_2 is defined as

$$q_2 =_{\text{Def}} \min(\{2m-i-2 \mid p[i] \neq p[m-1] \text{ and } 0 \le i \le m-2\} \cup \{2m-1\})\,.$$

The two occurrence relative positions q_1 and q_2 are then used by our heuristic IOH to calculate the shift advancements during the searching phase of the algorithm ImprovedOccurrenceMatcher in Figure 1, based on the following two occurrence functions

$$ibc1_p(c) =_{\text{Def}} gbc_p(q_1,c)\,, \qquad ibc2_p(c) =_{\text{Def}} gbc_p(q_2,c)\,.$$

These are computed by procedure PrecomputeIOH, shown in Figure 1, in $O(m+\sigma)$ time and $O(\sigma)$ space.

[1] A restricted variant of the generalized occurrence function gbc_p was presented in [7].

```
PrecomputeIOH(p, m, step)
 1.  for each c ∈ Σ do
 2.      ibc[c] ← step + 1
 3.  for i ← 0 to m − 1 do
 4.      ibc[p[i]] ← step − i
 5.  return ibc
```

```
ImprovedOccurrenceMatcher(p, m, t, n)
 1.  step_1 ← step_2 ← 2m − 1
 2.  for i ← 0 to m − 2 do
 3.      if p[i] = p[m − 1] then
 4.      then step_1 ← 2m − i − 2
 5.      else step_2 ← 2m − i − 2
 6.  ibc_1 ← PrecomputeIOH(p, m, step_1)
 7.  ibc_2 ← PrecomputeIOH(p, m, step_2)
 8.  s ← 0
 9.  while (s ≤ n − m) do
10.      if (p[m − 1] = t[s + m − 1]) then
11.          i ← 0
12.          while (i < m and p[i] = t[s + i]) do
13.              i ← i + 1
14.          if (i = m) then Output(s)
15.          s ← s + ibc_1[t[s + step_1]]
16.      else s ← s + ibc_2[t[s + step_2]]
```

Figure 1. A string matching algorithm based on the heuristic IOH.

5 A Self-Tuned Occurrence Heuristic

For a pattern p of length m, a text t, and a shift s, the heuristic IOH presented in the previous section computes shift advancements using the rule $ibc1_p$ or $ibc2_p$, based on two different relative positions, according to whether the last character of the pattern p matches its corresponding text character $t[s+m-1]$ or not. Differently, the Horspool and the Quick-Search algorithms compute their shift advancements by applying the occurrence heuristic on a fixed position $s+q$ of the text, with q equal, respectively, to $m-1$ and to m. In this section we will show that, given a pattern p and a text t with known character distribution, we can compute efficiently an occurrence relative position, to be called *worst-occurrence relative position*, which ensures the largest shift advancement on the average. The *worst-occurrence heuristic* (WOH) is then the corresponding occurrence heuristic based on the worst-occurrence relative position.

5.1 Finding the worst-occurrence relative position

Again, let t and p be respectively a text and a pattern over a common alphabet Σ and let $f : \Sigma \to [0,1]$ be the relative frequency function of the characters of t, so that $\sum_{c\in\Sigma} f(c) = 1$ holds.

For a given occurrence relative position $0 \le i \le m$, the average shift advancement of the generalized occurrence function gbc_p is given by the function

$$adv_{p,f}(i) =_{\text{Def}} \sum_{c\in\Sigma} f(c) \cdot gbc_p(i,c)\,. \tag{1}$$

We then define the *worst-occurrence relative position* q^* as the smallest position $0 \le q \le m$ which maximizes $adv_{p,f}(q)$, i.e.,

$$q^* =_{\text{Def}} \min\{q \mid 0 \le q \le m \text{ and } adv_{p,f}(q) = \max_{0\le i\le m} adv_{p,f}(i)\}\,.$$

Procedure FindWorstOccurrence in Figure 2 computes efficiently the position q^*, by exploiting the recurrence

$$adv_{p,f}(i) = \begin{cases} 1 & \text{if } i = 0 \\ adv_{p,f}(i-1) - f(p[i-1]) \cdot gbc_p(i-1, p[i-1]) + 1 & \text{if } 1 \le i \le m \end{cases}$$

```
FindWorstOccurrence(p, m, Σ, f)
 1.  for each c ∈ Σ do
 2.        lp[c] ← −1
 3.  adv ← 1
 4.  max ← 1
 5.  q ← 0
 6.  for i ← 1 to m do
 7.        gbc ← i − lp[p[i − 1]] − 1
 8.        adv ← adv − f(p[i − 1]) · gbc + 1
 9.        lp[p[i − 1]] ← i − 1
10.        if (adv > max) then
11.              max ← adv
12.              q ← i
13.  return q
```

```
PrecomputeWOH(p, m, q)
1.  for each c ∈ Σ do
2.        wo[c] ← q + 1
3.  for i ← 0 to q − 1 do
4.        wo[p[i]] ← q − i
5.  return wo

WorstOccurrenceMatcher(p, m, t, n)
1.  q ← FindWorstOccurrence(p, m, Σ, f)
2.  wo ← PrecomputeWOH(p, m, q)
3.  s ← 0
4.  while (s ≤ n − m) do
5.        i ← 0
6.        while (i < m and p[i] = t[s + i]) do
7.              i ← i + 1
8.        if (i = m) then Output(s)
9.        s ← s + wo[t[s + q]]
```

Figure 2. The procedure FindWorstOccurrence, the procedure PrecomputeWOH and the algorithm WorstOccurrenceMatcher.

for the calculation of the function $adv_{p,f}$ (lines 3 and 8), which, in turn, is based on the recurrence

$$gbc_p(i,c) = \begin{cases} 1 & \text{if } i = 0 \text{ or } c = p[i-1] \\ gbc_p(i-1,c) + 1 & \text{otherwise,} \end{cases}$$

for $0 \leq i \leq m$ and $c \in \Sigma$.

Notice that the entries of the generalized occurrence function gbc_p present in the above recurrence relation for $adv_{p,f}$ are only of the form $gbc_p(j, p[j])$. These can be expressed readily in terms of the *last-position* functions $lp_p^i : \Sigma \to \{-1, 0, \ldots, m-1\}$, defined (for $i = 0, 1, \ldots, m$) by

$$lp_p^i(c) =_{\text{Def}} \max(\{j \mid 0 \leq j < i \text{ and } p[j] = c\} \cup \{-1\}),$$

i.e., $lp_p^i(c)$ is the rightmost position of c in $p[0\,..\,i-1]$, if c is present in $p[0\,..\,i-1]$, otherwise $lp_p^i(c)$ is -1. In fact, we have

$$gbc_p(i, p[i]) = i - lp_p^i(p[i]),$$

for $0 \leq i \leq m-1$ (cf. line 7 of the **for-loop**).

The last-position functions can efficiently be computed during a left to right scanning of the pattern. These are maintained as a single array lp of size σ by the procedure FindWorstOccurrence. The array lp is initialized at lines 1-2 and subsequently updated at line 9 of the **for-loop**, by resorting to the recursive relation

$$lp_p^i(c) = \begin{cases} -1 & \text{if } i = 0 \\ i - 1 & \text{if } i > 0 \wedge c = p[i-1] \\ lp_p^{i-1}(c) & \text{if } i > 0 \wedge c \neq p[i-1]. \end{cases}$$

It is easy to oberve that the procedure FindWorstOccurrence has an overall $\mathcal{O}(m + \sigma)$-time and $\mathcal{O}(\sigma)$-space complexity.

5.2 The worst-occurrence heuristic

The *worst-occurrence heuristic* uses the position q^* computed by the procedure FindWorstOccurrence to calculate shift advancements during the searching phase in such a way that the character $t[s+q^*]$ is aligned with its rightmost occurrence on $p[0\,..\,q^*-1]$, if present; otherwise the pattern is advanced just past position $s+q^*$ of the text. This corresponds to advance the shift by $wo_p(t[s+q^*])$ positions, where

$$wo_p(c) =_{\text{Def}} \min(\{i \mid 1 \leq i \leq q^* \text{ and } p[q^*-i] = c\} \cup \{q^*+1\})\,.$$

Observe that, for $q^* = 0$, the advancement is equal to 1. The resulting algorithm can be immediately translated into programming code (see Figure 2 for a simple implementation). The procedure PrecomputeWOH, shown in Figure 2, computes the table which implements the worst-occurrence heuristic in $\mathcal{O}(m+\sigma)$ time and $\mathcal{O}(\sigma)$ space.

5.3 Finding the relative frequency of characters

The frequency of characters in texts has often been used in string matching algorithms for speeding up the searching process [16,1,15]. Such an approach is particularly useful when one is searching texts in natural languages, whose character distributions are well studied, and therefore known in advance. However, also in the case of texts in natural languages, the exact character distribution can not be predicted, since character frequencies may depend both on the writer and on the subject. The situation may become even worse in the case of other types of sequences. In such contexts, different approaches can be adopted for retrieving good approximations of the frequency of characters in order to apply accurately the worst-occurrence heuristic presented above. Here we propose some of them.

(*i*) In a preprocessing phase, compute the character frequencies of an initial segment of the text (say of no more than γ characters).

(*ii*) Run the first γ iterations of the algorithm WorstOccurrenceMatcher, assuming *a priori* a default distribution of characters (e.g., the uniform distribution). At the same time, compute the relative frequency of the first γ characters and then recompute the occurrence heuristic according to the estimated frequency.

(*iii*) While running the algorithm WorstOccurrenceMatcher, keep updating the relative frequencies of the characters. At regular intervals (say of γ characters), or when the difference between the current relative frequencies and the one used in the worst-occurrence heuristic exceeds a threshold, recompute the heuristic.

From our tests, it turns out that when the distribution of characters does not vary very much along the text, a good approximation of the frequencies can be computed even for quite small values of γ in the case of strategies (*i*) and (*ii*). For instance, in our experiments reported in Section 7 we used the value $\gamma = 100$, in combination with strategy (*i*). When the character frequencies tend to vary very much along the text (for instance, in the case of multi-language texts or in musical sequences), strategy (*iii*) might be preferable. However, one must keep in mind that the overhead can sensibly affect the algorithm performance.

6 A Jumping-Occurrence Heuristic

We recall that, for a pattern p of length m, the occurrence heuristics of the Zhu-Takaoka [19] and the Berry-Ravindran [1] algorithms are based on two consecutive characters, starting at positions $m-2$ and m, respectively. In both cases, the distance between the two characters involved in the occurrence heuristics is 1. We refer to such a distance as the *occurrence jump distance*.

It may be possible that other occurrence jump distances generate larger shift advancements. We will show in this section how, given a pattern p and a text t with known character distribution, we can compute efficiently an optimal occurrence jump distance which ensures the largest shift advancements on the average. The *jumping-occurrence heuristic* will be then the occurrence heuristic based on two characters with optimal occurrence jump distance.

6.1 Finding the optimal occurrence jump distance

Again, let p be a pattern of length m. To begin with, we introduce the *generalized double occurrence function* $gbc_p^2(i,j,c_1,c_2)$ *relative to* p, with $0 \le i \le m$, $1 \le j \le m$ and $c_1, c_2 \in \Sigma$, intended to calculate the largest *safe* shift advancement for p compatible with the constraints $t[s+i] = c_1$ and $t[s+i+j] = c_2$, when p has shift s with respect to a text t. Thus, we put:

$$\begin{aligned} gbc_p^2(i,j,c_1,c_2) =_{\text{Def}} \min(&\{i-k \mid m-j \le k < i \wedge p[k] = c_1\} \\ &\cup \{i-k \mid 0 \le k < \min(m-j,i) \wedge p[k] = c_1 \wedge p[k+j] = c_2\} \\ &\cup \{i+j-k \mid 0 \le k < j \wedge p[k] = c_2\} \\ &\cup \{i+j+1\}\,). \end{aligned} \tag{2}$$

Plainly, $gbc_p^2(i,j,c_1,c_2) \ge 1$ always holds and it can easily be checked that

$$gbc_p(i,c_1) < i+j-m+1 \quad \Longrightarrow \quad gbc_p^2(i,j,c_1,c_2) = gbc_p(i,c_1)\,. \tag{3}$$

Additionally, the shift rules of the Zhu-Takaoka and Berry-Ravindran algorithms can be expressed in terms of the generalized double occurrence function as, respectively, $gbc_p^2(m-2,1,c_1,c_2)$ and $gbc_p^2(m,1,c_1,c_2)$.

In the following we will refer to the parameters i and j of gbc_p^2 as the *relative occurrence position* and the *occurrence jump distance*, respectively. For fixed values of the relative occurrence position and the occurrence jump distance, the generalized double occurrence function can be computed in $\mathcal{O}(\sigma^2 + m\sigma)$ time and $\mathcal{O}(\sigma^2)$ space.

Let us fix, momentarily, the relative occurrence position i to $m-1$ and let $f : \Sigma \to [0,1]$ be the relative frequency of the characters in the text t. For a given $1 \le \ell \le m$, the probability that the generalized occurrence function gbc_p yields a shift advancement of length at least ℓ when inspecting the character at relative position $m-1$ is

$$Pr\{gbc_p(m-1,c) \ge \ell \mid c \in \Sigma\} = \sum_{\substack{c \in \Sigma \\ gbc_p(m-1,c) \ge \ell}} f(c)\,.$$

Example 1. Let $p = \mathsf{ACGAACT}$ be a pattern of $m = 7$ characters over the alphabet $\Sigma = \{\mathsf{A},\mathsf{C},\mathsf{G},\mathsf{T}\}$ of four *elements* with a relative frequency f such that $f(\mathsf{A}) = 0.3$, $f(\mathsf{C}) = 0.1$, $f(\mathsf{G}) = 0.4$ and $f(\mathsf{T}) = 0.2$. The shift advancements given by

each character at the relative occurrence position $m-1=6$ are $gbc_p(6,\mathsf{A})=2$, $gbc_p(6,\mathsf{C})=1$, $gbc_p(6,\mathsf{G})=4$, and $gbc_p(6,\mathsf{T})=7$, respectively. Thus, $adv_{p,f}(6)=3.7$.

The probabilities to have a shift advancement of length at least ℓ, for $1\leq\ell\leq 8$, are given by the following values

$Pr\{gbc6c\geq 1\mid c\in\Sigma\}=f(\mathsf{A})+f(\mathsf{C})+f(\mathsf{G})+f(\mathsf{T})=1$;
$Pr\{gbc6c\geq 2\mid c\in\Sigma\}=f(\mathsf{A})+f(\mathsf{G})+f(\mathsf{T})=0.9$;
$Pr\{gbc6c\geq 3\mid c\in\Sigma\}=f(\mathsf{G})+f(\mathsf{T})=0.6$;
$Pr\{gbc6c\geq 4\mid c\in\Sigma\}=f(\mathsf{G})+f(\mathsf{T})=0.6$;
$Pr\{gbc6c\geq 5\mid c\in\Sigma\}=f(\mathsf{T})=0.2$
$Pr\{gbc6c\geq 6\mid c\in\Sigma\}=f(\mathsf{T})=0.2$
$Pr\{gbc6c\geq 7\mid c\in\Sigma\}=f(\mathsf{T})=0.2$
$Pr\{gbc6c\geq 8\mid c\in\Sigma\}=0$.

Let j be a fixed relative jump distance to be used by the generalized double occurrence function gbc_p^2 with relative occurrence position $m-1$. In order for the character $t[s+m-1+j]$, at the relative position $m-1+j$, to be involved in the computation of the shift advancement by the function gbc_p^2, we must have

$$gbc_p(m-1,t[s+m-1])\geq j$$

(cf. (3)). Thus, for a fixed bound $0\leq\beta\leq 1$, the computation of the shift advancement will involve the second character with a probability of at least β if and only if its jump distance j satisfies

$$Pr\{gbc_p(m-1,c)\geq j\mid c\in\Sigma\}\geq\beta\,.$$

This suggests to use the following relative jump distance

$$j^*_\beta=_{\text{Def}}\max\Big\{\ell\mid 1\leq\ell\leq m\text{ and }Pr\big\{gbcm-1c\geq\ell\mid c\in\Sigma\big\}\geq\beta\Big\}$$

in the jumping-occurrence heuristic to be presented in the next section, at least in the case in which the relative occurrence position i is $m-1$. Plainly, the same argument can be generalized to any relative occurrence position.

In Example 1, if we set the bound $\beta=0.5$, we obtain a relative jump distance $j^*_{0.5}=4$. In other words, for the relative jump distance $j^*_{0.5}=4$, the character $t[s+10]$ will be involved in the computation of the shift advancement in at least 50% of the times, whereas in the remaining cases only the first character $t[s+6]$ will be involved. In practical cases we set $\beta=0.9$. This will yield, in Example 1, a relative jump distance $j^*_{0.9}=2$.

6.2 The Jumping-Occurrence Heuristic

For a pattern p of length m, the *jumping-occurrence heuristic* makes use of the occurrence relative position q^* returned by the procedure FindWorstOccurrence described in Section 5.1. Such a position q^* and the corresponding jump distance j^*_β computed by procedure FindJumpDistance are then used by the jumping-occurrence heuristic to calculate shift advancements during the searching phase in such a way that the characters $t[s+q^*]$ and $t[s+q^*+j^*_\beta]$ are aligned with their rightmost occurrence in p. In particular, this corresponds to advance the shift by $jbc_{p,\beta}(t[s+q^*],t[s+q^*-j^*_\beta])$ positions, where

$$jbc_{p,\beta}(c_1,c_2)=_{\text{Def}}\ gbc_p^2(q^*,j^*_\beta,c_1,c_2)\,.$$

The resulting algorithm is shown in Figure 3. The procedure PrecomputeJOH computes the table which implements the jumping-occurrence heuristic in $\mathcal{O}(\sigma^2+m\sigma)$ time and $\mathcal{O}(\sigma^2)$ space.

```
PrecomputeJOH(p, m, i, j)
 1. for each a ∈ Σ do
 2.       for each b ∈ Σ do
 3.             jbc(a, b) ← i + 1 + j
 4. for each a ∈ Σ do
 5.       for k ← 0 to j − 1 do
 6.             jbc(a, p[k]) ← i + 1 + j − 1 − k
 7. for k ← 0 to i + 1 − j − 1 do
 8.             jbc(p[k], p[k + len]) ← i + 1 − 1 − k
 9. for k ← i + 1 − j to m − 1 do
10.       for each a ∈ Σ do
11.             jbc(p[k], a) ← i + 1 − 1 − j
```

```
FindJumpDistance(p, m, i, Σ, f, β)
 1. for each c ∈ Σ do v[c] ← 1
 2. frq ← j ← 1
 3. while (frq ≥ β and j ≤ i + 1) do
 4.       if (v[p[i + 1 − j]] = 1) then
 5.             v[p[i + 1 − j]] = 0
 6.             frq ← frq − f(p[i + 1 − j])
 7.       j ← j + 1
 8. return j − 1

JumpingOccurrenceMatcher(p, m, t, n)
 1. i ← FindWorstOccurrence(p, m, Σ, f)
 2. j ← FindJumpDistance(p, m, i, Σ, f, 0.9)
 3. jbc ← PrecomputeJOH(p, m, i, j)
 4. s ← 0
 5. while (s ≤ n − m) do
 6.       k ← 0
 7.       while (k < m and p[k] = t[s + k]) do  k ← k + 1
 8.       if (k = m) then Output(s)
 9.       s ← s + jbc(t[s + i], t[s + i + j])
```

Figure 3. The procedure PrecomputeJOH (for computing the table implementing the jumping-occurrence heuristic), the procedure FindJumpDistance (for computing the jump relative distance for a pattern p and a relative frequency function f), and the algorithm JumpingOccurrence-Matcher.

6.3 Approximating the Optimal Jump Distance

If one knows in advance the character distribution of a given text, procedure FindJumpDistance in Figure 3 provides an efficient way for computing the optimal jump distance. Otherwise, one can adopt any of the three different approaches outlined in Section 5.3 for computing an approximated character distribution, and then, based on this, calculate the corresponding optimal occurrence relative position and jump distance. A somewhat simplified approach, still based on the strategy (*ii*) presented in Section 5.3, which bypasses the call to procedure FindJumpDistance, can be summarized in the following steps:

- initialize to 0 an array *scnt* (shifts counter) of length m;
- compute the worst-occurrence heuristic and run the first γ iterations of the algorithm by using such a rule for shifting; in the meantime, count the shifts of length ℓ occurring in this phase, for each length $\ell = 1, \ldots, m$, by updating accordingly the entries of the array *scnt*;
- compute an approximation of the value j^*_β by putting

$$\widetilde{j^*_\beta} =_{\text{Def}} \min \left\{ j \;\middle|\; \frac{1}{\gamma} \sum_{i=1}^{j} scnt[i] \geq \beta \right\};$$

- compute the jumping-occurrence heuristic, based on the value $\widetilde{j^*_\beta}$, and resume the search from the last shift position which has been checked, using such a rule for shifting.

It turns out that a good approximation of the optimal jump distance can be obtained even with small values of the parameter γ.

7 Experimental Results

We evaluated experimentally the impact of our proposed variants of the occurrence heuristics (in combination with their corresponding matchers):

- Improved-Occurrence Matcher (in short, IOM), described in Section 4,
- Worst-Occurrence Matcher (in short, WOM), described in Section 5.2,
- Jumping-Occurrence Matcher (in short, JOM), described in Section 6.2,

by testing them against the following algorithms based on the best known implementations of the occurrence heuristic:[2]

- Horspool algorithm (in short, HOR), which uses a single character occurrence heuristic and whose advancements are computed by $gbc_p(m-1, t[s+m-1])$;
- Quick Search algorithm (in short, QS), which uses a single character occurrence heuristic and whose advancements are computed by $gbc_p(m, t[s+m])$;
- Smith algorithm (SMITH), which uses a single character heuristic, whose advancements are computed by $\max(gbc_p(m, t[s+m]), gbc_p(m-1, t[s+m-1]))$;
- Berry-Ravindran algorithm (in short, BR), which uses two characters for shifting and whose advancements are computed by $gbc_p^2(m, 1, t[s+m], t[s+m+1])$;
- Zhu-Takaoka algorithm (in short, ZT), which uses two characters for shifting and whose advancements are computed by $gbc_p^2(m, 1, t[s+m-2], t[s+m-1])$.

Our implementation of the WOM algorithm computes the frequency of characters in the searched text by using the strategy (i) described in Section 5.3, with the parameter $\gamma = 100$, whereas our implementation of the JOM algorithm is based on the approach described in Section 6.3, with the same parameter $\gamma = 100$.

All algorithms have been implemented in the C programming language and have been compiled with the GNU C Compiler, using the optimization options -O3. All experiments have been executed locally on a MacBook Pro with 4 Cores, a 2 GHz Intel Core i7 processor, 4 GB RAM 1333 MHz DDR3, 256 KB of L2 Cache, and 6 MB of Cache L3. They have been evaluated in terms of the average shift advancements and running times, including any preprocessing time, measured with a hardware cycle counter available on modern CPUs. The tests have been run on text buffers over small and large alphabets. However we report in this paper only experimental results relative to small alphabets, since the gain in running time obtained when searching texts over large alphabets is negligible. In particular, we report experimental evaluations on a random sequence over an alphabet of 2 characters, a genome sequence, and a protein sequence, all sequences of 4MB. All sequences, provided by the SMART research tool,[3] are available online for download. Patterns of length m were randomly extracted from the sequences, with m ranging over the set of values $\{2^i \mid 1 \leq i \leq 12\}$. For each case, the mean over the running times, expressed in hundredths of seconds, of 500 runs has been reported. Figure 4 shows the running times of the Jumping-Occurrence Matcher with different values of the parameter β, whereas Figure 5 reports the running times of the algorithms HOR, QS, SMITH, BR, ZT, and the matchers IOM, WOM, and JOM, implementing our new proposed occurrence heuristics. The running times in Figure 5 of the JOM algorithm refer to an implementation with the parameter $\beta = 0.9$.

[2] For each algorithm we indicate the corresponding function used for shifting when the pattern of length m is aligned with the text at a given shift s.

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

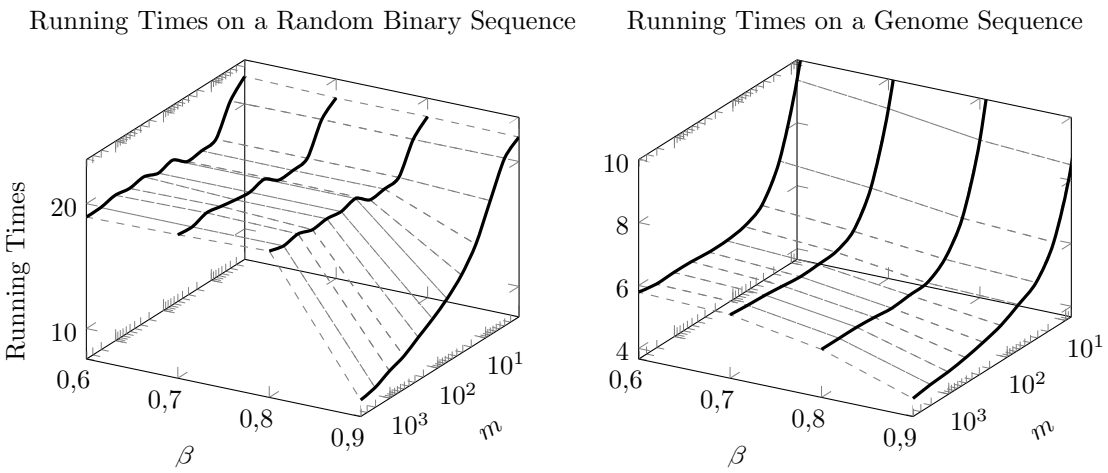

Figure 4. Running times of the Jumping-Occurrence Matcher for different values of the parameter β and pattern length m.

Running Times Evaluation

The experimental results in Figure 4 show that the choice of $\beta = 0.9$ is the best one for the jumping-occurrence heuristic in most cases. The gain in performance is more evident in the case of small alphabets or in the case of long patterns. In this latter case, the JOM algorithm with $\beta = 0.9$ is up to 50% faster. It is to be noticed, though, that in the case of large alphabets the improvement in running times is negligible.

From the experimental data in Figure 5, it follows that our proposed occurrence heuristics obtain always the best results. In particular the JOM algorithm is always the best choice for large alphabets. However, its speed-up is almost negligible in the case of large alphabets and long patterns, whereas it becomes more evident for very small alphabets, exhibiting a speed-up of more than 50% with respect to the best known algorithms. The IOM algorithm shows a very good behavior for short patterns. In fact, it turns out that it is the best solution in the case of short patterns and small alphabets, where it is more than 20% faster than other algorithms based on single character heuristics. However, its performance degrades as the length of the pattern increases. The WOM algorithm turns out to the best algorithm when the pattern is not short. Among the algorithms based on a single character occurrence heuristic, it shows an extremely fast behavior and for long patterns it is up to 50% faster than previous existing solutions. It is to be noticed that its running times are very close to those obtained by the JOM algorithm, which, however, is based on a two-characters heuristic.

Stability Evaluation

It is also useful to find out how accurately repeatable the results are. If only average running times are considered, some important details may be hidden. The Smart tool computes the stability of an algorithm as the standard deviation of the running times of the tests. The standard deviation measures the amplitude of the variation from the average, i.e., the mean of the running times. A low standard deviation indicates that the running times tend to be very close to the mean, underlying a high stability of the algorithm. On the other hand, a high standard deviation indicates that the running times are spread out over a large range of values, thus indicating a low stability.

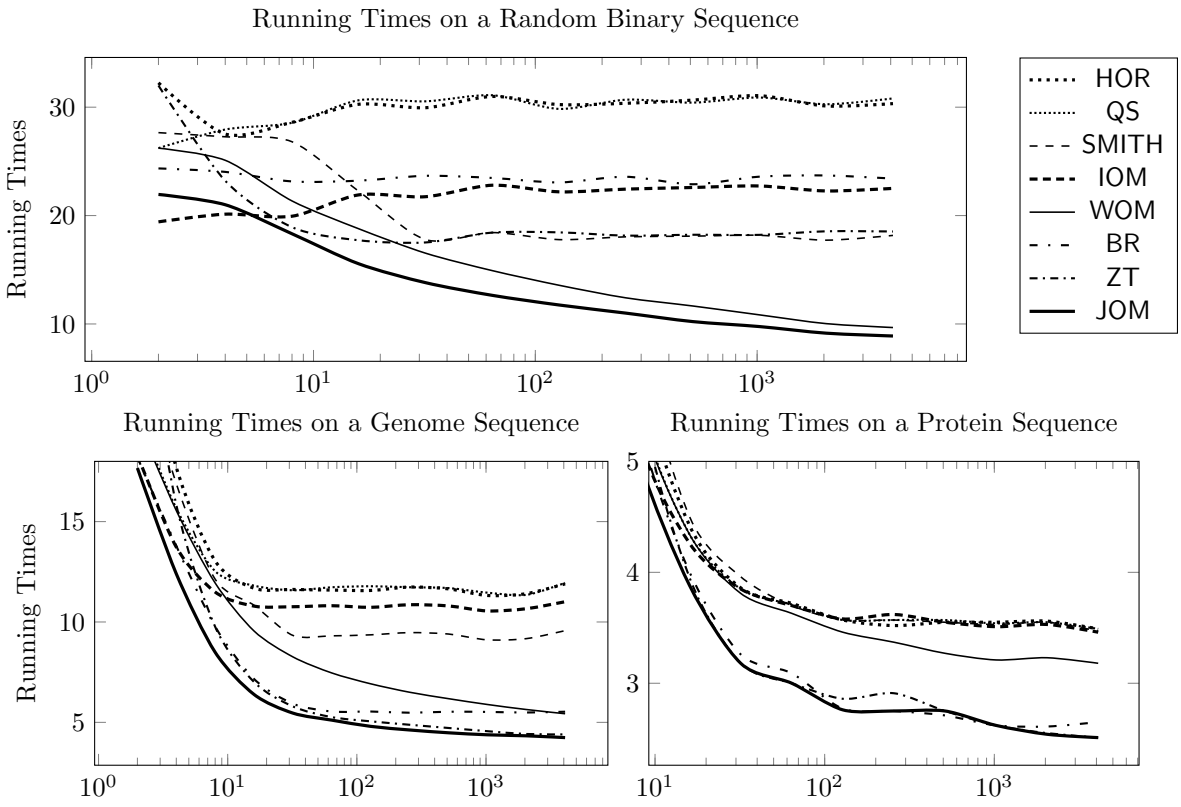

Figure 5. Running times obtained by comparing several efficient algorithms based on the occurrence heuristic shifting strategy. The x-axis represent length of patterns.

Figure 6 reports the standard deviation of the running times observed in our tests. It turns out that the WOM and the JOM heuristics are sensibly more stable than the remaining algorithms, especially in the case of long patterns and small alphabets.

While standard algorithms based on the one-character occurrence heuristic (as, for instance, HOR, QS and SMITH) become less stable as the length of the pattern increases, in some cases the algorithms based on our proposed occurrence heuristics show an opposite behavior, i.e., they become more stable as the length of the pattern increases. In particular, the IOM algorithm turns out to be the more stable algorithm in the case of short patterns, but it becomes less stable for long patterns. The converse behavior can be noticed in the case of the WOM algorithm, though we notice that the improvement in stability becomes negligible in the case of large alphabets.

Flexibility Evaluation

Flexibility is an important attribute of various types of systems. In the field of string matching, it refers to algorithms that can adapt when changes in the input data occur. Thus a string matching algorithm can be considered flexible when, for instance, it maintains good performance for both short and long patterns, or in the case of both small and large alphabets. By analyzing the running times reported in Figure 5, it turns out that the JOM algorithm is the more flexible one among the algorithms which have been tested, as it shows very good performance for all the lengths of the patterns and different sizes of the alphabet. The IOM algorithm turns out to be very efficient only for short patterns (and in some cases it is even more efficient than the JOM algorithm), but its performance degrades as the length of the pattern increases.

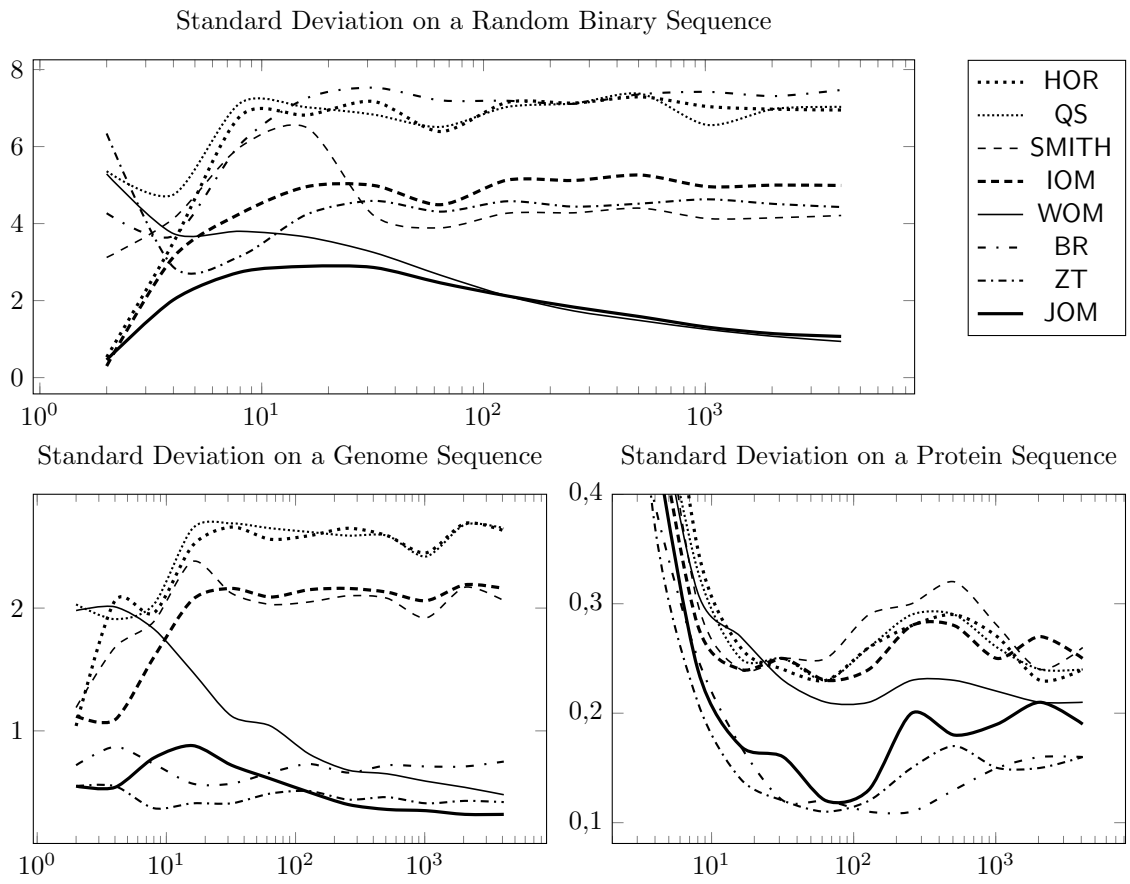

Figure 6. Standard Deviation of running times obtained by comparing several efficient algorithms based on the occurrence heuristic shifting strategy. The x-axis represent length of patterns.

An opposite observation can be done for the WOM algorithm, which maintains good performance only for medium and long patterns.

8 Conclusions

In this paper we have presented three new variations of the occurrence heuristic based on a smart computation of the relative position of the character used for computing the shift advancement. The proposed variations yield the largest average advancement, according to the characters distribution in the text. We have also shown experimental evidence that the new variants of the occurrence heuristics achieve very good results in practice, especially in the case of long patterns or small alphabets. We plan to conduct a probabilistic and a combinatorial analysis of the new proposed rules directed at giving theoretical support to the experimental evidence reported in the present work.

References

1. T. Berry and S. Ravindran: *A fast string matching algorithm and experimental results*, in Proceedings of the Prague Stringology Conference '99, J. Holub and M. Šimánek, eds., Czech Technical University, Prague, Czech Republic, 1999, pp. 16–28, Collaborative Report DC–99–05.
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: *Fast-search algorithms: New efficient variants of the Boyer-Moore pattern-matching algorithm.* Journal of Automata, Languages and Combinatorics, 10(5/6) 2005, pp. 589–608.
4. D. Cantone, S. Faro, and E. Giaquinta: *A compact representation of nondeterministic (suffix) automata for the bit-parallel approach*, in Combinatorial Pattern Matching, 21st Annual Symposium, CPM 2010, vol. 6129 of Lecture Notes in Computer Science, Springer, 2010, pp. 288–298.
5. D. Cantone, S. Faro, and E. Giaquinta: *A compact representation of nondeterministic (suffix) automata for the bit-parallel approach.* Inf. Comput., 213 2012, pp. 3–12.
6. D. Cantone, S. Faro, and E. Giaquinta: *On the bit-parallel simulation of the nondeterministic aho-corasick and suffix automata for a set of patterns.* J. Discrete Algorithms, 11 2012, pp. 25–36.
7. D. Cantone and S. Faro: *On tuning the bad-character rule: the worst-character rule*, Tech. Rep. arXiv:1012.1338v1, CoRR at arXiv.org - Cornell University Library, December 2010, Available at http://arxiv.org/abs/1012.1338.
8. L. Cleophas, B. Watson, and G. Zwaan: *A new taxonomy of sublinear right-to-left scanning keyword pattern matching algorithms.* Sci. Comput. Program., 75(11) Nov. 2010, pp. 1095–1112.
9. S. Faro and T. Lecroq: *The exact online string matching problem: A review of the most recent results.* ACM Comput. Surv., 45(2) 2013, p. 13.
10. S. Faro and M. O. Külekci: *Fast multiple string matching using streaming SIMD extensions technology*, in String Processing and Information Retrieval - 19th International Symposium, SPIRE 2012, vol. 7608 of Lecture Notes in Computer Science, Springer, 2012, pp. 217–228.
11. S. Faro and M. O. Külekci: *Fast packed string matching for short patterns*, in Proceedings of the 15th Meeting on Algorithm Engineering and Experiments, ALENEX 2013, SIAM, 2013, pp. 113–121.
12. S. Faro, and E. Pappalardo: *Ant-CSP: An Ant Colony Optimization Algorithm for the Closest String Problem*, in SOFSEM 2010: Theory and Practice of Computer Science, 36th Conference on Current Trends in Theory and Practice of Computer Science, vol. 5901 of Lecture Notes in Computer Science, Springer, 2010, pp. 360–281.
13. R. N. Horspool: *Practical fast searching in strings.* Softw. Pract. Exp., 10(6) 1980, pp. 501–506.
14. A. Hume and D. M. Sunday: *Fast string searching.* Softw. Pract. Exp., 21(11) 1991, pp. 1221–1248.
15. M. E. Nebel: *Fast string matching by using probabilities: on an optimal mismatch variant of Horspool's algorithm.* Theor. Comput. Sci., 359(1) 2006, pp. 329–343.
16. D. M. Sunday: *A very fast substring search algorithm.* Commun. ACM, 33(8) 1990, pp. 132–142.
17. B. Watson: *Taxonomies and toolkits of regular language algorithms*, Phd. Thesis., Faculty of Computing Science, Eindhoven University of Technology, 1995.
18. A. C. Yao: *The complexity of pattern matching for a random string.* SIAM J. Comput., 8(3) 1979, pp. 368–387.
19. R. F. Zhu and T. Takaoka: *On improving the average case of the Boyer-Moore string matching algorithm.* J. Inf. Process., 10(3) 1988, pp. 173–177.