On Tuning the Bad-Character Rule: the Worst-Character Rule

Domenico Cantone and Simone Faro

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

Abstract. In this note we present the *worst-character rule*, an efficient variation of the *bad-character* heuristic for the exact string matching problem, firstly introduced in the well-known Boyer-Moore algorithm. Our proposed rule selects a position relative to the current shift which yields the largest average advancement, according to the characters distribution in the text. Experimental results show that the worst-character rule achieves very good results especially in the case of long patterns or small alphabets in random texts and in the case of texts in natural languages.

Keywords. string matching, experimental algorithms, text processing.

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. It is a very extensively studied problem in computer science, mainly due to its direct applications to such diverse areas as text, image and signal processing, information retrieval, computational biology, etc.

In this paper we present the *worst-character rule*, an efficient variation of the *bad-character* heuristic for the exact string matching problem, firstly introduced in the well-known Boyer-Moore algorithm [BM77]. Our proposed rule selects a position relative to the current shift which yields the largest average advancement, according to the characters distribution in the text. Experimental results show that the worst-character rule achieves very good results especially in the case of long patterns or small alphabets in random texts and in the case of texts in natural languages.

Before entering into details, we review some useful notations and terminology. A string P of length $m \geq 0$ 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

(A) GENERIC_STRING_MATCHER(T, P, n, m)

```
PRECOMPUTE_GLOBALS(P)
s := 0
while s ≤ n − m do
    j := CHECK_SHIFT(s, P, T)
    s := s+ SHIFT_INCREMENT(s, P, T, j)
```

(B) PRECOMPUTE_BC(P, Σ)

```
m = length(P)
for each c ∈ Σ do
    bc_P(c) = m
for i = 0 to m − 1 do
    bc_P(P[i]) = m − i − 1
```

Fig. 1. (A) The procedure GENERIC_STRING_MATCHER for searching the occurrences of a pattern P in a text T. **(B)** The procedure PRECOMPUTE_BC for computing the bad-character heuristic.

case the substring $T[s \mathinner{..} s+m-1]$ is called the *current window* of the text. If $T[s \mathinner{..} s+m-1] = P$, 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.

Most string matching algorithms have the general structure shown in Figure 1(A), where the procedure PRECOMPUTE_GLOBALS(P) computes useful mappings, in the form of tables, which may be accessed by the function named SHIFT_INCREMENT(s, P, T, j); the function CHECK_SHIFT(s, P, T) checks whether s is a valid shift and returns the position j of the last matched character in the pattern; the function SHIFT_INCREMENT(s, P, T, j) computes a *positive* shift increment according to the information tabulated by procedure named PRECOMPUTE_GLOBALS(P) and to the position j of the last matched character in the pattern. For instance, to look for valid shifts, the celebrated Boyer-Moore algorithm [BM77] scans the pattern 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 *bad-character* rules.

2 The bad-character rule

Information gathered during the execution of the SHIFT_INCREMENT(s, P, T, j) function, in combination with the knowledge of P, as suitably extracted by procedure PRECOMPUTE_GLOBALS(P), can yield shift increments larger than 1 and ultimately lead to more efficient algorithms. In this section we focus our attention on the use of the bad-character heuristic for preprocessing the pattern, introduced by Boyer and Moore in [BM77].

The Boyer-Moore algorithm is the progenitor of several algorithmic variants which aim at computing close to optimal shift increments very efficiently. Specifically, the Boyer-Moore algorithm checks whether s is a valid shift, by scanning the pattern P from right to left and, at the end of the matching phase, it computes the shift increment as the largest value suggested by the *good-suffix rule* and the *bad-character rule*, provided that both of them are applicable.

Specifically, the bad-character heuristic states that if $c = T[s+j-1] \neq P[j-1]$ is the first mismatching character, while scanning P and T from right to left with shift s, then P can be safely shifted in such a way that its rightmost occurrence of c, if present, is aligned with position $(s + j - 1)$ in T (provided that such an occurrence is in $P[0 \mathinner{..} j - 2]$, otherwise the bad-character rule has no effect). In the case in which c does not occur in P, then P can be safely shifted just past position $(s + j - 1)$ in T. More formally, the shift increment suggested by the bad-character heuristic is given by the expression $(j - bc_P(T[s + j - 1]) - 1)$, where $bc_P(c) =_{\text{Def}} \max(\{0 \leq k < m \mid P[k] = c\} \cup \{-1\})$, for $c \in \Sigma$. Procedure PRECOMPUTE_BC, shown in Figure 1(B), computes the function bc_P during the preprocessing phase 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 bad-character heuristic, some variants of the Boyer-Moore algorithm were based just on it and dropped the good-suffix heuristic.

For instance, Horspool [Hor80] 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 $T[s+m-1]$ is aligned with its rightmost occurrence on $P[0 \mathinner{..} m - 2]$, if present; otherwise the pattern is advanced just past the window. This corresponds to advance the shift by $hbc_P(T[s + m - 1])$ positions, where

$$hbc_P(c) =_{\text{Def}} \min(\{1 \leq k < m \mid P[m - 1 - k] = c\} \cup \{m\}) \ .$$

The resulting algorithm performs well in practice and can be immediately translated into programming code (see Baeza-Yates and Régnier [BYR92] for a simple implementation in the **C** programming language).

Likewise, the Quick-Search algorithm, presented in [Sun90], uses a modification of the original 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

$$qbc_P(c) =_{\text{Def}} \min(\{1 \leq k \leq m \mid P[m - k] = c\} \cup \{m + 1\}) \ .$$

Finally, the Smith algorithm [Smi91] computes its shift advancements by taking the largest value suggested by the Horspool and the Quick-Search bad-character rules. Its preprocessing phase is performed in $O(m+\sigma)$-time and $O(\sigma)$-space complexity, while its searching phase has a quadratic worst case time.

Although the role of the good-suffix heuristic in practical string matching algorithms has recently been reappraised [CF03b,CF03c,CF05], also in consideration of the fact that often it is as effective as the bad-character heuristic, especially in the case of non-periodic patterns, the bad character heuristic is still considered one of the powerfull method for speed up the performance of string matching algorithms (see for instance [FL08,FL09]).

3 The worst-character rule

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

It may be possible that other bad-character relative positions generate larger shift advancements. We will show below how, given a pattern P and a text T with known character distribution, we can compute efficiently the bad-character relative position, to be called *worst-character relative position*, which ensures the largest shift advancements on the average. The *worst-character rule* is then the bad-character rule based on such a worst-character relative position.

3.1 Finding the worst-character relative position

To begin with, we introduce the *generalized bad-character function* $gbc_P(i,c)$. Suppose the pattern P has shift s in the text T. For a given bad-character relative position i, with $0 \leq i \leq m$, $gbc_P(i, T[s+i])$ is the shift advancement such that the character $T[s+i]$ is aligned with its rightmost occurrence in $P[0 \,..\, i-1]$, 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). Thus,

$$gbc_P(i,c) =_{\text{Def}} \min(\{1 \leq k \leq i \mid P[i-k] = c\} \cup \{i+1\}), \quad \text{for } c \in \Sigma,\ 0 \leq i \leq m\,.$$

Plainly, $gbc_P(i,c) \geq 1$ always holds. Additionally, the shift rules of the Horspool and Quick-Search algorithms can be expressed in terms of the generalized bad-character function by $hbc_P(c) = gbc_P(m-1, c)$ and $qbc_P(c) = gbc_P(m,c)$, respectively, for $c \in \Sigma$.

Next, let $f : \Sigma \to [0,1]$ be the relative frequency of the characters in the text T. Given a fixed pattern P and a bad-character relative position $0 \leq i \leq m$, the average shift advancement of the generalized bad-character function on i is given by the function

$$adv_{P,f}(i) =_{Def} \sum_{c \in \Sigma} f(c) \cdot gbc_P(i,c)\,.$$

Thus, the *worst-character relative position* of a given pattern P and a given relative frequency function f can be defined as the smallest position $0 \leq q \leq m$ such that

$$adv_{P,f}(q) = \max_{0 \leq j \leq m} adv_{P,f}(j)\,.$$

The procedure FIND_WORST_CHARACTER, shown in Figure 2(A), computes the worst-character relative position for a given input pattern P and a given relative frequency function f over Σ in $O(m+\sigma)$-time and $O(\sigma)$-space. It exploits the recurrence

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

for the computation of $adv_{P,f}(i)$, for $i = 0, \dots, m$, which is based, in turn, on the fact that

$$gbc_P(i,c) = \begin{cases} gbc_P(i-1,c) & \text{if } P[i-1] \neq c \\ 1 & \text{otherwise}, \end{cases}$$

for $c \in \Sigma$ and $i = 0, 1, \dots, m$.

Observe that in the above recurrence only entries of the generalized bad-character function of the form $gbc_P(i, P[i])$ are needed. To compute such values, the characters of the pattern are processed from left to right and, for each position i, the *last position* function $lp^i_P : \Sigma \to \{-1, 0, \dots m-1\}$, which gives the rightmost occurrence of each character $c \in \Sigma$ in $P[0\,..\,i-1]$, is also computed. The value of $lp^i_P(c)$ is set to -1 if either $i = 0$ or c is not present in $P[0\,..\,i-1]$. Formally, for $c \in \Sigma$,

$$lp^i_P(c) = \max(\{0 \leq j < i \mid P[j] = c\} \cup \{-1\}).$$

Observe that at the i-th iteration of the **for**-loop of procedure FIND_WORST_CHARACTER, only the value $lp^i_P(P[i])$ is needed. The function lp^i_P is maintained as an array of dimension σ and computed by the following recursive relation

$$lp^i_P(c) = \begin{cases} -1 & \text{if } i = 0 \\ i-1 & \text{if } i > 0 \text{ and } c = P[i-1] \\ lp^{i-1}_P(c) & \text{if } i > 0 \text{ and } c \neq P[i-1]. \end{cases}$$

The initialization of lp^0_P is plainly done in $O(\sigma)$-time, while the computation of lp^i_P, for $i > 0$, can be done in constant time from array lp^{i-1}_P. Finally, the values $gbc_P(i, P[i])$ are computed using the following relation

$$gbc_P(i, P[i]) = \begin{cases} 1 & \text{if } i = 0 \\ i - lp^i_P(P[i]) & \text{if } 0 < i < m. \end{cases}$$

3.2 The worst-character heuristic

The position q computed by procedure FIND_WORST_CHARACTER is then used by the worst-character heuristic to calculate shift advancements during the searching phase. In particular the worst-character heuristic computes shift advancements 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 $wc_P(T[s+q])$ positions, where

$$wc_P(c) =_{\text{Def}} \min(\{1 \leq k \leq q \mid P[q-k] = c\} \cup \{q+1\}) \ .$$

Observe that if $q = 0$ then the advancement is always equal to 1. The resulting algorithm can be immediately translated into programming code (see Figure 2(C) for a simple implementation). The procedure PRECOMPUTE_WC, shown in Figure 2(B), computes the table which implements the worst-character heuristic in $O(m + \sigma)$-time and space.

```
(A) FIND_WORST_CHARACTER(P, Σ, f)
    m = length(P)
    for each c ∈ Σ do
        lp_P(c) = −1
    q = 0
    adv_{P,f}(0) = 1
    max = 1
    lp_P(P[0]) = 0
    δ = f(P[0])
    for i = 1 to m do
        adv_{P,f}(i) = adv_{P,f}(i − 1) + 1 − δ
        δ = f(P[i]) · (i − lp_P(P[i]))
        lp_P(P[i]) = i
        if adv_{P,f}(i) > max then
            max = adv_{P,f}(i)
            q = i
    return q
```

```
(B) PRECOMPUTE_WC(P, Σ, q)
    m = length(P)
    for each c ∈ Σ do
        wc(c) = q + 1
    for i = 0 to q − 1 do
        wc(P[i]) = q − i
```

```
(C) WORST_CHARACTER_MATCHER(P, T, m, n)
    q = FIND_WORST_CHARACTER(P, Σ, f)
    wc = PRECOMPUTE_WC(P, Σ, q)
    s = 0
    while s ≤ n − m do
        j = m − 1
        while j ≥ 0 and P[j] = T[s + j] do
            j = j − 1
        if j < 0 then OUTPUT(s)
        s = s + wc(T[s + q])
```

Fig. 2. (A) The procedure FIND_WORST_CHARACTER for computing the worst-character relative position of the pattern P. **(B)** The procedure PRECOMPUTE_WC for computing the worst-character heuristic **(C)** The WORST_CHARACTER_MATCHER algorithm which makes use of the worst-character heuristic.

4 Experimental Results

To evaluate experimentally the impact of the worst-character heuristic, we have chosen to test the WORST_CHARACTER_MATCHER algorithm (in short WC), given in Figure 2(C), with three algorithms based on variations of the bad-character rule, namely the Horspool algorithm (in short, HOR), the Quick-Search algorithm (in short QS), and the Smith algorithm (in short SM). Experimental results have been evaluated in terms of running times and average advancement given by the shift heuristics. All algorithms have been implemented in the C programming language and were used to search for the same strings in large fixed text buffers on a PC with AMD Athlon processor of 1.19GHz. In particular, all algorithms have been tested on four Randσ problems and on four Exp$^\lambda\sigma$ problems, for alphabet sizes $\sigma = 2, 4, 8, 16$. For each problem, the patterns have been constructed by selecting 200 random substrings of length m from the files, for $m = 2, 4, 8, 16, 32, 64, 128, 256, 512$.

Each Randσ and Exp$^\lambda\sigma$ problem consists in searching a set of 200 random patterns of a given length in a 20Mb random text over a common alphabet of size σ. Randσ and Exp$^\lambda\sigma$ problems differ in the distribution of characters in the text buffer.

In a Randσ the characters of the text buffer have a uniform distribution, i.e. the relative characters frequency is defined by the law $f(c) = 1/\sigma$, for all $c \in \Sigma$.

In an Exp$^\lambda\sigma$ problem the distribution of characters follows the inverse-rank power-law of degree λ, a model that gives a very good approximation of the relative frequency function of characters in terms of their ranks both in natural language dictionaries and texts (cf. [CF03a]). Formally, in a text in natural language the relative frequency of the character c_i of rank i can be approximated by

$$f(c_i) = \frac{(\sigma - i + 1)^\lambda}{\sum_{j=1}^{\sigma} j^\lambda}, \quad \text{for } i = 1, \ldots, \sigma\,,$$

where the value of the degree $\lambda \in \mathbb{R}$ can be determined experimentally and usually ranges in the interval [3..10] (cf. [CF03a]). In our tests we have set $\lambda = 5$.

In the following tables running times are expressed in hundredths of seconds, while the average advancements are expressed in number of characters.

$\sigma = 2$	2	4	8	16	32	64	128	256
HOR	47.78	47.55	46.70	47.90	44.49	44.42	44.79	44.35
QS	**40.07**	45.15	44.56	45.13	42.00	41.36	41.48	41.29
SM	60.74	59.58	58.65	61.98	60.48	60.33	60.80	60.51
WC	41.36	**43.78**	**37.76**	**33.55**	**28.08**	**25.59**	**23.94**	**22.53**
$\sigma = 4$	2	4	8	16	32	64	128	256
HOR	37.85	28.78	23.17	22.20	22.29	21.84	21.64	21.82
QS	**29.97**	**25.63**	22.25	20.91	21.16	21.00	20.87	20.97
SM	48.24	38.77	30.54	29.09	29.55	28.70	28.33	28.72
WC	30.88	26.61	**22.22**	**20.01**	**18.95**	**18.36**	**17.96**	**17.45**
$\sigma = 8$	2	4	8	16	32	64	128	256
HOR	30.01	22.15	18.55	17.33	17.07	17.00	16.95	17.05
QS	**23.54**	**20.15**	**17.80**	**16.96**	16.77	16.79	16.73	16.73
SM	39.49	30.16	22.93	20.01	19.37	19.29	19.21	19.31
WC	23.98	20.58	18.20	17.03	**16.59**	**16.42**	**16.23**	**16.19**
$\sigma = 16$	2	4	8	16	32	64	128	256
HOR	25.75	19.87	17.30	16.50	16.11	15.96	15.94	16.09
QS	**20.71**	**18.77**	**16.75**	**16.29**	**16.00**	**15.90**	15.92	15.98
SM	34.79	26.22	20.39	18.10	16.89	16.66	16.52	16.83
WC	21.08	19.01	17.06	16.39	16.07	16.02	**15.79**	**15.83**

Running times in hundredths of seconds for Randσ problems

$\sigma = 2$	2	4	8	16	32	64	128	256
HOR	1.50	1.88	2.05	1.97	2.01	1.96	1.95	1.97
QS	1.72	1.89	2.09	2.01	1.95	1.97	1.96	1.98
SM	**1.96**	**2.44**	2.71	2.59	2.61	2.56	2.56	2.59
WC	1.72	2.18	**2.75**	**3.16**	**3.66**	**4.09**	**4.60**	**5.20**
$\sigma = 4$	2	4	8	16	32	64	128	256
HOR	1.75	2.75	3.62	3.84	3.85	4.00	4.11	3.96
QS	2.30	3.05	3.79	3.97	3.89	3.95	4.07	3.99
SM	**2.49**	**3.74**	**5.05**	**5.42**	5.39	5.57	5.77	5.57
WC	2.30	3.05	4.09	4.94	**5.92**	**6.75**	**7.37**	**8.36**
$\sigma = 8$	2	4	8	16	32	64	128	256
HOR	1.87	3.31	5.28	7.04	7.95	8.08	8.11	8.06
QS	2.63	3.89	5.62	7.15	7.95	8.03	8.05	8.07
SM	**2.74**	**4.40**	**7.08**	**9.89**	**11.49**	**11.66**	11.67	11.68
WC	2.63	3.89	5.62	7.60	9.61	11.13	**12.29**	**13.44**
$\sigma = 8$	2	4	8	16	32	64	128	256
HOR	1.93	3.63	6.46	10.24	14.10	15.91	16.25	15.79
QS	2.81	4.41	7.05	10.62	14.21	15.95	16.23	15.65
SM	**2.87**	**4.72**	**8.18**	**13.67**	**20.00**	**23.35**	**23.87**	**22.96**
WC	2.81	4.41	7.05	10.62	14.80	18.18	20.41	22.36

Average advancement for Randσ problems

$\sigma = 2$	2	4	8	16	32	64	128	256
HOR	48.99	65.21	99.57	126.85	139.51	137.22	137.22	133.16
QS	45.74	61.74	94.95	123.09	135.82	134.06	124.21	120.76
SM	86.60	123.34	149.58	186.02	205.72	201.96	205.68	201.45
WC	**43.63**	**60.30**	**90.91**	**110.29**	**114.81**	**113.07**	**85.10**	**70.05**
$\sigma = 4$	2	4	8	16	32	64	128	256
HOR	45.49	46.64	46.67	44.27	36.25	33.87	32.65	31.84
QS	36.96	42.33	44.41	41.07	33.32	30.69	30.40	29.77
SM	64.76	67.11	62.51	58.60	49.80	45.93	43.66	42.82
WC	**35.73**	**39.98**	**40.33**	**34.58**	**27.15**	**24.16**	**22.40**	**21.24**
$\sigma = 8$	2	4	8	16	32	64	128	256
HOR	39.09	33.88	26.99	24.10	22.18	21.39	21.25	20.54
QS	**31.50**	**29.99**	25.70	22.80	21.68	20.63	20.62	19.83
SM	50.46	43.41	34.09	29.50	26.62	24.94	24.64	23.13
WC	32.36	30.12	**24.97**	**21.30**	**19.67**	**18.81**	**18.21**	**17.62**
$\sigma = 16$	2	4	8	16	32	64	128	256
HOR	33.11	24.80	20.14	18.25	17.58	17.26	17.02	16.47
QS	**25.98**	**22.18**	**19.23**	17.72	17.13	16.93	16.82	16.30
SM	42.62	33.12	25.09	21.12	19.55	18.96	18.60	17.90
WC	26.66	22.61	19.52	**17.70**	**17.02**	**16.66**	**16.40**	**16.02**

Running times in hundredths of seconds for four $\mathsf{Exp}^5\sigma$ problems

$\sigma = 2$	2	4	8	16	32	64	128	256
HOR	1.04	1.11	1.23	1.41	1.63	1.87	1.86	1.97
QS	1.08	1.14	1.24	1.40	1.64	1.86	1.85	1.97
SM	**1.10**	1.17	1.29	1.49	1.72	1.97	1.98	2.06
WC	**1.10**	**1.21**	**1.41**	**1.67**	**2.02**	**2.34**	**2.90**	**3.55**
$\sigma = 4$	2	4	8	16	32	64	128	256
HOR	1.32	1.65	2.04	2.24	2.53	2.81	3.08	3.17
QS	1.54	1.74	2.10	2.35	2.53	2.87	3.06	3.12
SM	**1.68**	**2.03**	**2.60**	2.88	3.29	3.70	4.13	4.28
WC	1.62	1.98	2.45	**3.09**	**3.80**	**4.60**	**5.59**	**6.34**
$\sigma = 8$	2	4	8	16	32	64	128	256
HOR	1.63	2.29	3.10	3.71	4.49	4.99	5.24	5.74
QS	2.02	2.59	3.20	3.77	4.38	4.98	5.15	5.80
SM	**2.25**	**3.17**	**4.34**	**5.36**	**6.65**	**7.59**	8.08	9.05
WC	2.04	2.70	3.58	4.68	5.91	6.95	**8.24**	**9.66**
$\sigma = 16$	2	4	8	16	32	64	128	256
HOR	1.79	2.99	4.46	5.97	7.29	8.26	9.65	10.25
QS	1.79	2.99	4.46	5.97	7.29	8.26	9.65	10.25
SM	**2.61**	**4.07**	**6.17**	**8.69**	**11.08**	**13.03**	**15.44**	**16.76**
WC	2.46	3.49	4.87	6.72	8.57	10.55	12.83	14.95

Average advancements for four $\mathsf{Exp}^5\sigma$ problems

The above experimental results show that the algorithm based on the worst-character heuristic obtains the best runtime performances in most cases, especially for long patterns and small alphabets, and it is second only to the Quick-Search algorithm, in the case of small patterns, as the alphabet size increases.

Concerning the average advancements, it turns out that the proposed heuristic is quite close to the Smith heuristic, which generally shows the best behavior. We notice, though, that in the case of long patterns and small alphabets the presented heuristic proposes the longest average advancements.

Finally we observe that the performances of the worst-character heuristic increase when tested on an $\mathsf{Exp}^5\sigma$ problem.

5 Conclusions

Several efficient variations of the bad-character heuristic have been proposed in the last years with the aim of obtaining better performances in practical cases. For instance, the Berry-Ravindran algorithm [BR99] generalizes the Quick-Search algorithm by using in its bad-character rule the last two characters, rather than just the last one. Another example is the Tuned-Boyer-Moore algorithm [HS91] which introduces, using the Horspool bad-character rule, an efficient implementation of the searching phase. Finally, algorithms in the Fast-Search family [CF05] combine the bad-character rule with the good-suffix heuristic by computing an $\mathcal{O}(\sigma \times m)$-space function.

In this paper we have presented the *worst-character rule*, a variation of the *bad-character* heuristic, which is based on the position relative to the current shift which yields the largest average advancement, according to the characters distribution in the text. We have also shown experimental evidence that the worst-character rule achieves very good results in practice, especially in the case of long patterns or small alphabets in random texts and in the case of texts in natural languages.

References

[BM77] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, 1977.

[BR99] T. Berry and S. Ravindran. A fast string matching algorithm and experimental results. In J. Holub and M. Šimánek, editors, *Proceedings of the Prague Stringology Conference '99*, pages 16–28, Czech Technical University, Prague, Czech Republic, 1999. Collaborative Report DC–99–05.

[BYR92] R. A. Baeza-Yates and M. Régnier. Average running time of the Boyer-Moore-Horspool algorithm. *Theor. Comput. Sci.*, 92(1):19–31, 1992.

[CF03a] D. Cantone and S. Faro. On the frequency of characters in natural language texts. In F. Spoto, G. Scollo, and A. Nijholt, editors, *Proceedings of Algebraic Methods in Language Processing (AMiLP 2003), 3rd AMAST Workshop on Language Processing*, pages 68–82, 2003.

[CF03b] D. Cantone and S. Faro. Fast-Search: a new variant of the Boyer-Moore string matching algorithm. In K. Jansen, M. Margraf, M. Mastrolilli, and J.D.P. Rolim, editors, *Proceedings of Second International Workshop on Experimental and Efficient Algorithms (WEA 2003)*, volume 2647 of *Lecture Notes in Computer Science*, pages 47–58. Springer-Verlag, 2003.

[CF03c] D. Cantone and S. Faro. Forward-Fast-Search: another fast variant of the Boyer-Moore string matching algorithm. In M. Šimánek, editor, *Proceedings of the Prague Stringology Conference '03*, pages 10–24, Czech Technical University, Prague, Czech Republic, 2003.

[CF05] 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):589–608, 2005.

[FL08] Simone Faro and Thierry Lecroq. Efficient variants of the backward-oracle-matching algorithm. In *Stringology*, pages 146–160, 2008.

[FL09] Simone Faro and Thierry Lecroq. Efficient variants of the backward-oracle-matching algorithm. *Int. J. Found. Comput. Sci.*, 20(6):967–984, 2009.

[Hor80] R. N. Horspool. Practical fast searching in strings. *Softw. Pract. Exp.*, 10(6):501–506, 1980.

[HS91] A. Hume and D. M. Sunday. Fast string searching. *Softw. Pract. Exp.*, 21(11):1221–1248, 1991.

[Smi91] P. D. Smith. Experiments with a very fast substring search algorithm. *Softw. Pract. Exp.*, 21(10):1065–1074, 1991.

[Sun90] D. M. Sunday. A very fast substring search algorithm. *Commun. ACM*, 33(8):132–142, 1990.