On the Expected Sublinearity of the Boyer–Moore Algorithm

R. Schaback

The author gratefully dedicates this paper to the memory of his academic teacher, Prof. Dr. H. Werner.

Abstract. This paper analyzes the expected performance of a simplified version BM' of the Boyer–Moore string matching algorithm. A probabilistic automaton A is set up which models the expected behavior of BM' under the assumption that both text and pattern are generated by a source which emits independent and uncorrelated symbols with an arbitrary distribution of probabilities. Formal developments lead then to the conclusion that A takes expected sublinear time in a variety of situations. The sublinear behavior can be quantitatively predicted by simple formulae involving the pattern length m and the alphabet's probabilistic properties. Finally, empirical evidence is provided which is in satisfactory accordance with the theory.

Keywords : *String searching, Pattern matching, Average case analysis of algorithms.* [1]

1 The problem

Let $\mathcal{A}$ be a finite alphabet, $|\mathcal{A}| =: n$, and suppose strings

$$T = t_1 ... t_N,\ t_i \in \mathcal{A},\ 1 \le i \le N \text{ (the ``text'')}$$

$$S = s_1 ... s_m,\ s_i \in \mathcal{A},\ 1 \le i \le m \le N \text{ (the ``pattern'')}$$

are given. To avoid formal difficulties in later discussions, we assume S to be expanded to the left by "jokers", i.e. characters that match any other character from $\mathcal{A}$.

To determine the leftmost occurrence of S as a substring of T means then:

$$\textit{Find the smallest } j,\ \ m \le j \le N,\ \ \textit{such that } s_{m-i} = t_{j-i} \textit{ for } 0 \le i \le m-1,$$

$$\textit{or output that no such } j \textit{ exists.}$$

[1]The formulation of this paper was significantly improved by the constructive criticism of one of the referees.

To solve this problem, Boyer and Moore [2] defined an algorithm of the following form:

```
$j := m;\;\; k := 0;$
$REPEAT$
${\{}At\ this\ stage,\ k\ characters\ match : t_{j-i} = s_{m-i},\ 0 \le i \le k-1\}$
  $IF\ t_{j-k} = s_{m-k}$
  $THEN\ k := k+1$
  $ELSE\ BEGIN$
    $j := j + MOVE(m, k, t_{j-k});$
    $k := 0$
    $END$
$UNTIL\ (k = m)\ OR\ (j > N);$
$IF\ k = m\ THEN\ output\ (Pattern\ is\ found\ at\ position\ j)$
$ELSE\ output\ (No\ occurrence\ of\ pattern\ in\ text);$
```

Variations of this algorithm depend on the function

$$MOVE\ (m, k : INTEGER; t : CHARACTER) : INTEGER$$

which determines how many positions S may be moved forward along T to the next position where S can occur as a substring of T. In case $k = 0$, i.e. if the text character $t = t_j$ does not match the last pattern character s_m, the pattern can be moved along the text as long as t does not occur within the pattern:

$$MOVE(m, 0, t) := \min\{p \mid 1 \le p \le m,\ s_{m-p} = t\}\,. \tag{1}$$

For $k > 0$ there are k matching characters $s_{m-k+1} \dots s_m = t_{j-k+1} \dots t_j$ and we know that $s_{m-k} \ne t_{j-k}$. The simplest way to shift the pattern is to ignore most of this information and to match $t_{j-k+1} = s_{m-k+1}$ with the next possible occurrence in the pattern:

$$MOVE(m, k, \cdot) := MOVE(0, m-k+1, s_{m-k+1}) \quad (1 \le k < m). \tag{2}$$

We use the notation BM' for the Boyer–Moore algorithm using (1) and (2) and consider the j–increment given by the $MOVE$ function as the *progress* of the algorithm.

The original Boyer–Moore algorithm BM tries to match the whole subpattern $s_{m-k+1} \dots s_m$ with its rightmost reoccurrence in the pattern; an additional condition makes sure that the unsuccessful subpattern $s_{m-k} \dots s_m$ does not occur again:

$$MOVE^*(m, k, \cdot) := \min\left\{p \,\middle|\, \begin{array}{l} 1 \le p \le m,\ s_{m-i} = s_{m-p-i},\ 0 \le i \le k-1 \\ s_{m-k} \ne s_{m-p-k}\ \ if\ p+k < m \end{array}\right\} (1 \le k < m).$$

These definitions of $MOVE$ formally require the pattern s to be expanded to the left by $\max(1, m-1)$ jokers.

Since more progress may be possible when matching the rightmost occurrence of $t = t_{j-k}$ in the subpattern $s_1 \dots s_{m-k}$, BM finally takes the maximum of $MOVE^*(m, k, \cdot)$ and $MOVE(m-k, 0, t)$ to define $MOVE(m, k, t)$. The effective construction of lookup tables for the implementation of $MOVE$ is treated in the cited literature (e.g. [6]).

The worst–case performance of BM increases linearly with N, measured by the number of pairwise character comparisons. Knuth, Morris, and Pratt [6] first proved the bound $7N$ in case that the pattern does not occur in the text. This bound has later been improved to $4N$ by Guibas and Odlyzko [5], while the lower bound $N - m + 1$ for **any** algorithm was established by Rivest [7]. The case of r occurrences of the pattern in the text led to the bound $7N + 8rm - 14r$ in [6] (see also [4]), and Galil [4] introduced a variation of BM to get the bound $14N$ for **any** r. Finally, Apostolico and Giancarlo [1] proved the bound $2N$ for their variation of BM.

In this paper we neglect the problem of multiple occurrences of the pattern in the text and do not incorporate the corresponding variations of [4] and [1]. These are extensions for handling multiple occurrences and could be added on, if necessary. Since the occurrence of a large random pattern in a large random text is a highly improbable event, a study of the expected behavior of BM for large patterns can neglect multiple occurrences altogether.

The expected behavior of BM was studied only for the very special case of equally probable characters. The sublinearity (i.e. the expected number of character comparisons is less than N) was proved in the original paper [2] by Boyer and Moore, while Knuth, Morris, and Pratt [6] gave a variation with expected $\mathcal{O}(\frac{N}{m \log m})$ character comparisons. This behavior is optimal due to results of Yao [8].

This paper tries to eliminate the assumption of equally probable characters and to bridge the gap between theory and applications. For patterns and texts from natural languages the observed average behavior of BM and BM' is clearly sublinear, and we give useful formulae that predict this behavior.

In addition, the $\mathcal{O}(\frac{N}{m \log m})$ result of [6] is carried over to the general case of arbitrary character probabilities, while a blocked variation of the simplified algorithm BM' will need $\mathcal{O}(\frac{N}{m \log^2 m})$ comparisons.

2 Probabilistic assumptions

In the sequel, we consider an alphabet $\mathcal{A}$ with characters c having probabilities

$$0 < p(c) < 1, \ \sum_{c \in \mathcal{A}} p(c) = 1. \tag{3}$$

We use q to denote the probability $\sum_{c \in \mathcal{A}} p^2(c)$ that two randomly chosen characters match.

We switch now from BM' to a **completely randomized** model algorithm by assuming that

1. any reference to some character of the text or the pattern will produce a (possibly new) random character;
2. the $MOVE(m, k, t)$ function is replaced by its expected value $M(m, k)$ for all characters t and patterns $S \in \mathcal{A}^m$.

The first assumption means that each reference to some t_{j-k} or s_{m-k} acts like a procedure call that generates a random character independent of S, T, k, and j.

This seems to be quite restrictive and unrealistic at first glance, but we shall see later that the progress of the algorithm is so large that multiple references to text characters are rare events under a variety of circumstances. Therefore, the randomized algorithm can be expected to perform on its random data sources in the same way as the deterministic version of BM' performs on an input that is randomly chosen before execution. Furthermore, later results will show that for alphabets with the probabilistic properties of natural languages, the algorithm BM' spends most of the time comparing the last pattern character s_m with text characters t_i for values of i that are far away from each other. Then the randomized algorithm will model the behavior of the deterministic algorithm quite well even for natural languages, since natural language characters sampled over large intervals can be considered as random characters with fixed probabilities.

Our randomized algorithm replaces the comparison of t_{j-k} with s_{m-k} with a random decision between two alternatives with probabilites q and $1-q$, respectively. Moreover, the terminating conditions of BM' (including detection of the pattern and exhaustion of the text) are completely ignored in order to simplify the following discussion. Thus we get the following algorithm A:

```
$j := m;\ \ k := 0;$
$REPEAT$
   $With\ probability\ q\ :\ k := k+1$
   $ELSE\ (with\ probability\ 1-q)$
      $BEGIN$
      $j := j + M(m,k)$
      $k := 0$
      $END$
$UNTIL\ FALSE;$
```

The variable k in the algorithm A denotes a "state", corresponding to the situation of k matches in BM and BM'. In this sense A is a probabilistic automaton with an unbounded number of states. Transitions from state k to state $k+1$ occur with the probability q of a match, yielding no progress in j. With the probability $1-q$ of a mismatch, transitions from state k to state zero with progress $M(m,k)$ occur. Reaching state m corresponds to an occurrence of the pattern in the text; higher states of A are purely formal. Each $REPEAT$–cycle will be called a *step*, and since BM has one character comparison for each step, we have one unit of "cost" per step in A. Expected sublinearity means then that the expected progress per step is larger than one.

3 Theoretical Considerations

The expected behavior of A is described by the following

Lemma 3.1 *The probability μ_{rk} that A is in state k after r steps is*

$$\mu_{rk} = q^k(1-q) \ \textit{for all} \ \ r > k \geq 0.$$

The expected progress counted in states up to $m-1$ is

$$E_m = (1-q)^2 \sum_{k=0}^{m-1} q^k M(m,k) \tag{4}$$

after at least m steps.

Proof: Algorithm A starts in state 0 with probability one. Then the probability μ_{rk} of A being in state k after r steps is

$$\begin{array}{rcll} \mu_{rk} & = & q^k(1-q), & 0 \leq k < r \\ \mu_{rr} & = & q^r, & \\ \mu_{rk} & = & 0, & k > r, \end{array}$$

as is easily seen by induction. After k steps, transitions from state k to state 0 occur with probabilities $(1-q)^2 q^k$, and these lead to the progress $M(m,k)$. Summing up gives (4). ■

The lemma implies that

- the transient start phase of algorithm A to reach a step–independent probability for states $0 \ldots m-1$ is short (m steps),
- the finiteness of the number $m+1$ of actual states of BM and BM' as opposite to the infinite number of states of A, does not matter much because choosing small values of q ensures that higher states of A are very improbable.

We now evaluate $M(m,k)$.

Lemma 3.2 *If S is a random string of length m (with a joker s_0 added) and c is a random character, the random function*

$$f_m(c,S) = \min \{k \mid 1 \leq k \leq m, \ s_{m-k} = c\}$$

has the expected value

$$F_m = |\mathcal{A}| - \sum_{c \in \mathcal{A}} (1-p(c))^m. \tag{5}$$

Furthermore,

$$\begin{array}{rcll} M(m,0) & = & F_m & \\ M(m,k) & = & F_{m-k+1} & (1 \leq k < m) \\ M(m,k) & = & 0 & \text{otherwise.} \end{array} \tag{6}$$

Proof: Clearly, f_m has the expected value

$$F_m = \sum_{i=1}^{m-1} i \cdot \sum_{c \in \mathcal{A}} (1-p(c))^{i-1} \cdot p^2(c) + m \cdot \sum_{c \in \mathcal{A}} (1-p(c))^{m-1} p(c),$$

because the progress i, $1 \leq i < m$, implies $(i-1)$ mismatches and one match, while $i = m$ implies $m-1$ mismatches (note that we ignore s_m). A little calculation gives (5), and (1), (2) imply (6). ■

We now can write (4) as

$$E_m = (1-q)^2 \left(F_m + \sum_{k=1}^{m-1} q^k F_{m-k+1} \right) \tag{7}$$

where the values F_k are available from (5).

For small values of m one can use (7) and (5) directly to estimate the efficiency of the algorithm A. Using the probability distributions of characters of natural languages one can tabulate (7) and (5) via (3). For example, Table 1 exhibits the corresponding values for the distribution of the 26 characters of the English language (data from [3]).

m	F_m	E_m	m	F_m	E_m
1	1.0000	0.8728	21	12.1401	11.3402
2	1.9342	1.7992	22	12.4420	11.6224
3	2.8081	2.6193	23	12.7304	11.8918
4	3.6263	3.3842	24	13.0059	12.1493
5	4.3935	4.1012	25	13.2695	12.3956
6	5.1136	4.7741	26	13.5218	12.6314
7	5.7902	5.4065	27	13.7636	12.8573
8	6.4268	6.0013	28	13.9954	13.0739
9	7.0263	6.5616	29	14.2178	13.2817
10	7.5915	7.0898	30	14.4313	13.4812
11	8.1250	7.5883	31	14.6366	13.6730
12	8.6291	8.0594	32	14.8339	13.8574
13	9.1059	8.5049	33	15.0238	14.0349
14	9.5573	8.9267	34	15.2067	14.2057
15	9.9851	9.3265	35	15.3829	14.3704
16	10.3911	9.7058	36	15.5528	14.5292
17	10.7765	10.0661	37	15.7168	14.6823
18	11.1429	10.4084	38	15.8750	14.8302
19	11.4916	10.7342	39	16.0279	14.9730
20	11.8236	11.0445	40	16.1756	15.1111

Table 1: F_m and E_m for the English alphabet, $q = 0.0658, n = |A| = 26$

Since later examples will show that A closely resembles BM and BM' even for natural languages, the user can easily estimate the expectable progress of BM and BM' by looking at such a table. Average patterns S of length 20 will for instance be moved forward about 11 characters per single-character comparison, when searched for in average English texts.

We now prove some lower bounds of F_m. First we concentrate on the case of small patterns:

Lemma 3.3 *For $m \leq 1/q$,*

$$F_m \geq m - m^2 q/2 \geq m/2. \tag{8}$$

Proof: For any real number $x \geq 0$ we have

$$1 - (1-x)^k \geq 1 - e^{-kx} \geq kx - \frac{k^2}{2} x^2$$

and (8) follows from (5). ■

The progress of A for small patterns from large alphabets with small values of q (this occurs for natural languages) can be predicted by a useful rule of thumb:

Theorem 3.1 *The algorithm A has expected progress*

$$E_m \geq (1-q)(1-q^2)(m - m^2 q/2) \geq (1-q)(1-q^2)m/2,$$

$$E_m \approx m/2 \text{ for small } q$$

per character comparison, provided that $2 \leq m \leq 1/q$.

Proof: Combine the two major terms of (7) with (8). ■

In Table 1, $1/q \approx 15.2$, so Theorem 3.1 is applicable for pattern lengths up to 15, when the probability distribution of characters in English texts is assumed. Within this range, expected progress is at least about $m/2$. This observation for a series of practical cases was the starting point for our investigation.

We now treat the case of large m but still keep the alphabet fixed. Our main result in this direction is

Theorem 3.2 *A distribution of n character probabilities p_i leads to sublinearity of A for sufficiently large pattern lengths m, if $q = \sum_{i=1}^n p_i^2$ satisfies*

$$n(1-q) > 1.$$

This is the case, if

$$q < \hat{q}_n := 1 - \frac{1}{n}. \tag{9}$$

The proof will be a consequence of the lemma following below, if m is large enough. If we sort the characters of $\mathcal{A}$ in the form

$$1 > p_1 \geq p_2 \geq \ldots \geq p_n > 0, \;\; n = |\mathcal{A}|,$$

then there is some γ satisfying

$$q < \gamma < 1, \;\; 1 - p_n \leq \gamma, \tag{10}$$

and we get

Lemma 3.4

$$E_m \geq n(1-q)\left(1-q^m-\frac{\gamma^{m+1}}{\gamma-q}\right). \tag{11}$$

Proof: Equation (5) implies

$$F_m \geq n(1-(1-p_n)^m) \geq n(1-\gamma^m). \tag{12}$$

Using this in (7) gives

$$\begin{aligned} E_m &\geq (1-q)^2 \textstyle\sum_{k=0}^{m-1} q^k F_{m-k} \\ &\geq (1-q)^2 n \textstyle\sum_{k=0}^{m-1} q^k(1-\gamma^{m-k}) \\ &= (1-q)^2\left(n\frac{1-q^m}{1-q} - n\gamma^m\frac{1-q^m/\gamma^m}{1-q/\gamma}\right) \\ &\geq n(1-q)\left(1-q^m-\frac{\gamma^{m+1}}{\gamma-q}\right). \end{aligned} \tag{13}$$

■

Expected sublinearity of A means that the expected progress E_m per character comparison is greater than one. Equation (11) shows that for large patterns the product $n(1-q)$ occurs as the maximal expected progress; this proves Theorem 3.2. ■

Remarks

1. The model and the algorithm are in state 0 with probability circa $1-q$. Higher states k have probability $q^k(1-q)$ and are very improbable indeed for small values of q.

2. In state 0 the algorithms BM and BM' coincide. If A models BM' in state 0, then it models BM in that state, too.

3. In state 0 the last character s_m of S is responsible for the progress. In case of sublinearity this character is tested against different characters from T in the major part of the character comparisons. Then the probabilistic assumptions are not very restrictive; the model A will closely resemble BM' (and BM) in state 0 (and in general, because other states are improbable). Furthermore, the behavior of the model A and the algorithm BM' then is independent of the probability of pairs of characters; the single–character probabilities are sufficient to describe the situation, even for natural language strings.

4. A value of $q \approx 1$ spoils the performance of A and the quality of A as a model of BM', while a large alphabet size $n = |\mathcal{A}|$ and a large pattern size m act favorably.

5. Inequality (11) shows that the size of the alphabet times the probability of a mismatch is the limiting factor for the efficiency of A for large patterns. This indicates that further speed–up requires large alphabets or blocking strategies that let $\mathcal{A}$ increase with m.

6. The case $q \approx 1$ would imply that a single character must have a probability close to one. Since always $q \geq \frac{1}{n}$ in an n–character alphabet, the case of a binary alphabet can not lead to an efficiency larger than one and attains efficiency one only if both characters have equal probabilities. In this case, blocking will improve the performance of the algorithm (see below).

7. Uniformly distributed character probabilities lead to $q = 1/n$ and the conditions $m \leq n$ and $n > 2$ in Theorems 3.1 and 3.2, respectively.

8. For states $k \geq 1$ the efficiency of BM will exceed that of BM' (and A) locally, because it makes at least the progress of BM' after any single specific comparison. However, BM is not superior for every text and pattern, because it may run into unfavorable regions of the text which the simplified version may happen to avoid.

To get a further speed–up of the pattern–matching process, large alphabets with small values of q are needed. Therefore, we consider a b–fold blocking of the alphabet $\mathcal{A}$, $|\mathcal{A}| = n$, $1 \leq b < m$ and study first the blocked BM version proposed by Knuth, Morris and Pratt ([6], p. 341). Their result (and proof technique) can be generalized as follows.

Theorem 3.3 *There is an algorithm for pattern matching that inspects $\mathcal{O}(N\frac{\log_{1/q} m}{m})$ characters in a random text with arbitrarily distributed characters.*

Proof: We follow [6] to combine steps of BM with an arbitrary linear worst–case algorithm. Each iteration shifts the pattern at least $m - b$ positions to the right and consists of the following elementary steps:

1. The last b characters of the pattern are compared with a block B of b text characters.

2. In case of match, proceed to 4.

   In case of mismatch, a function similar to (1) can be used to decide whether B occurs in the pattern at all.

   If this is not the case, the pattern can be moved $m - b$ positions along the text and the next iteration can be started.

   Otherwise proceed to 3.

3. In this case two blocks of pattern and text match somewhere; we ignore the possible shift given by step 2 and proceed to 4.

4. Use a linear worst–case algorithm to move the pattern at least $m - b$ positions to the right (including the determination of possible occurrences of the pattern in the text) and perform another iteration. This step ignores the fact that there is some match of certain blocks of the pattern and the text.

Note that 1. and 2. are the BM part of the algorithm; they are equivalent to a state zero step of A on the blocked alphabet. Steps 3 and 4 use no more than $2m$ characters of the text and therefore require an amount $c \cdot m$ of work.

For small b and large m there will be no overlap of text blocks sampled in step 1 of the iteration. We therefore can assume that these parts of text and pattern are stochastically independent.

Lemma 3.5 *The probability to match a random block B of b characters with an arbitrary block of b characters in a random pattern of $m \geq b$ characters does not exceed $(m-b+1)q^b$.*

Proof: There are $m-b+1$ possible positions for B to occur as a block within the pattern. The probability to occur at a fixed position is q^b. The product of these numbers is a crude upper bound for the situation in the assertion. ■

The expected progress of the algorithm in each iteration cycle is at least $m-b$, and the expected number of character comparisons is at most

$$(m-b+1) \cdot q^b \cdot cm + 1 \cdot b$$

where we simply took the upper bound 1 for the probability of a mismatch of a random block. Now we use

$$b := \lfloor 2 \log_{1/q} m \rfloor$$

to get $q^b \approx m^{-2}$, and the expected efficiency will be

$$\frac{m-b}{const. + b} = \mathcal{O}(m/\log_{1/q} m). ■$$

We do not lose too much if we simply apply a blocked version of A:

Theorem 3.4 *If A works on b–fold blocks in b parallel versions with $b \approx \log_{1/q} m$, the expected number of single character comparisons is $\mathcal{O}(\frac{N}{m} \log^2_{1/q} m)$.*

Proof: We apply former results for the alphabet $\mathcal{A}^b$ and first use (8) to get

$$F_r \geq r - \frac{r^2}{2} q^b$$

for r blocks of b characters. We ignore the higher–order terms in E_r and find the lower bound

$$E_r \geq \left(1-q^b\right)^2 F_r \geq r \left(1-q^b\right)^2 \left(1 - \frac{r}{2} q^b\right).$$

Now consider a string of length $m < q^{-m}$ over A and a blocking factor b with

$$m < \frac{1}{q^b}, \text{ i.e. } b \geq \lceil log_{1/q} m \rceil, \ 1 \leq b \leq m.$$

For simplicity, we then can define $r \geq 1$ by

$$rb \leq m < (r+1)b$$

and consider r blocked steps in our efficiency measure, since the efficiency is monotonic with respect to pattern length. Then

$$E_r \geq \frac{1}{2}\left(\frac{m}{b}-1\right)\left(1-\frac{1}{m}\right)^2,$$

using

$$rq^b < \frac{r}{m} \leq \frac{1}{b} \leq 1,$$

is a lower bound of our efficiency measure progress/cost in both block–by–block or character–by–character units. Since we need b versions of the algorithm, one for each block alignment, the total efficiency $E_{m,b}$ will be at least

$$E_{m,b} \geq \frac{m}{2}\frac{1}{b}\left(\frac{1}{b}-\frac{1}{m}\right)\left(1-\frac{1}{m}\right)^2. \tag{14}$$

For $b^* = \lceil log_{1/q} m \rceil$ we have $b^* \leq 1 + log_{1/q} m$ and get

$$E_{m,b^*} \geq \frac{m}{2}\frac{1}{1+log_{1/q} m}\left(\frac{1}{1+log_{1/q} m}-\frac{1}{m}\right)\left(1-\frac{1}{m}\right)^2$$
$$= \mathcal{O}\left(\frac{m}{log^2_{1/q} m}\right) \text{ for } m \to \infty,$$

without any restrictions on n, m, and q except $m < q^{-m}$, which is always satisfied for large m. ■

The blocking strategy depends on m and q; for instance, equal probabilities for 0 and 1 in the binary alphabet give $q = \frac{1}{2}$ and $b^* = \lceil log_2 m \rceil$. One can use (14) for the usual blocking factors $b = 4, 8, 16, 32$, provided that $m < q^{-b}$ holds.

4 Empirical Observations

To check the validity of our model algorithm A we tested A, BM, and BM' on a variety of inputs. For a fixed alphabet $\mathcal{A}$ with a specified character distribution we generated large samples of random strings S and T for values of m between 2 and 40. For each m we plotted the expected efficiency (4) of A versus the means of the observed efficiencies of BM and BM' (see figures 1–3). For a binary alphabet (see fig. 1) BM exceeds BM' and A in efficiency. This is due to the fact that the efficiency of BM may well exceed the value of $(1-q)\cdot|\mathcal{A}| = 0.49374 \cdot 2 = 0.98748$, which essentially bounds the efficiency of its competitors. But our theoretical results indicate that blocking should be used to avoid $m \gg |\mathcal{A}|$, and therefore this example is of minor significance.

For larger alphabets ($|\mathcal{A}| = 8$ in figure 2, $|\mathcal{A}| = 26$ in figure 3) the efficiency as modeled by A does resemble the actual efficiency of both versions of the Boyer–Moore algorithm quite well (the vertical lines denote confidence intervals at the 1 % error level), but there is a small systematic overestimation of the efficiency of the simplified version that may be credited to multiple evaluations.

For the comparison on natural language strings we used a text of 4785 ASCII characters from a LaTeX source of part of a chapter of a course in computer science, written in German. We chose a random sample of patterns occurring in the text (fig. 4) and in a different chapter of the same course (fig. 5). Then we plotted the expected efficiency of A against the observed efficiencies of BM and BM' as before. The results indicate once again that A closely describes the behavior of BM and BM'. Of course the examples with occurring patterns (fig. 4) show an overestimation of the efficiency of A, because unexpectedly high states occur.

References

[1] A. Apostolico and R. Giancarlo, *The Boyer–Moore–Galil String Searching Strategies Revisited*, SIAM J. Comput., 15 (1986), 98–105

[2] R. S. Boyer and J. S. Moore, *A fast string searching algorithm*, Comm. ACM, 20 (1977), pp. 323–350.

[3] C. C. Foster, *Cryptanalysis for Microcomputers*, Hayden Book Company 1982

[4] Z. Galil, *On improving the worst case running time of the Boyer–Moore string searching algorithm*, Comm. ACM, 22 (1979), 505–508

[5] L. J. Guibas, and A. M. Odlyzko, *A new proof of the linearity of the Boyer–Moore string searching algorithm*, SIAM J. Comput., 9 (1980), 672–682.

[6] D. E. Knuth, J. H. Morris and V. R. Pratt, *Fast pattern matching in strings*, SIAM J. Comput., 6 (1977), pp. 323–350.

[7] R. L. Rivest, *On the worst-case behavior of string-searching algorithms*, SIAM J. Comput., 6 (1977), 669–674

[8] A. C.–C. Yao, *The Complexity of Pattern Matching for a Random String*, SIAM J. Comput., 8 (1979), 368–387

Prof. Dr. R. Schaback
Institut für Numerische und Angewandte Mathematik
der Universität Göttingen
Lotzestraße 16–18
D–3400–Göttingen
Federal Republic of Germany