SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 12, 57–66 (1982)

A Comparison of Three String Matching Algorithms

G. DE V. SMIT

*Institute for Applied Computer Science, University of Stellenbosch, Stellenbosch, South Africa**

SUMMARY

Three string matching algorithms—straightforward, Knuth–Morris–Pratt and Boyer–Moore—are examined and their time complexities discussed. A comparison of their actual average behaviour is made, based on empirical data presented. It is shown that the Boyer–Moore algorithm is extremely efficient in most cases and that, contrary to the impression one might get from the analytical results, the Knuth–Morris–Pratt algorithm is not significantly better on the average than the straightforward algorithm.

KEY WORDS Pattern matching String matching Searching Time complexity

INTRODUCTION

The problem of detecting the (non-)occurrence of a particular string of characters (the pattern) in another, usually longer string of characters (the text string), forms an integral part of many text-editing, data retrieval and symbol manipulation problems. In this paper three different string matching algorithms are compared in the light of empirical evidence presented. The algorithms are: an obvious, straightforward algorithm, the Knuth–Morris–Pratt algorithm and the Boyer–Moore algorithm.

We use the following notation throughout: p_i is the ith character in the pattern $p = p_1 \ldots p_m$ of length m and t_j is the jth character in the text string $t = t_1 \ldots t_n$ of length n. Both p and t are strings over the alphabet I of size q.

A STRAIGHTFORWARD ALGORITHM (SF)

This algorithm is probably the one which first comes to mind whenever the string matching problem is addressed. The pattern is placed over the text string at its extreme left so that t_1 and p_1 are aligned. The pattern and text characters are then scanned to the right for a mismatch. If a mismatch is found, the pattern is shifted one position to the right and the scan is restarted at the new position of p_1.

Algorithm 1. A straightforward algorithm

Input: PAT and TXT, arrays of pattern and text characters; $m>0$ and $n>0$, the number of characters in PAT and TXT, respectively.

Output: Success or failure indicator and if success, the location of the pattern in the text.

---

* Current address: H. le Roux and Associates, Box 261071, Excom 2023 RSH

0038–0644/82/010057–10$01.00 *Received 5 September 1980*

```
1.  i : = 0;
2.  while (i<n−m+1) do
3.    i : = i+1; /* i = current guess where pattern begins in text */
4.    j : = i; k : = 1;
5.    while (PAT(k) = TXT(j)) do
6.      if (k = m) then
7.        return ("pattern found at", i)
        else do
8.        j : = j+1;
9.        k : = k+1;
        end;
      end;
    end;
10. return ("pattern not found");
```

It is not difficult to see that the worst case execution time occurs if, for every possible starting position of the pattern in the text, all but the last character of the pattern matches the corresponding character in the text (e.g. when $a^{m-1}b$ is searched for in a^n, with $n \gg m$). In this case $O(mn)$ comparisons are needed to determine that the pattern does not occur in the text. The expected running time is much less, however, since a mismatch between pattern and text usually occurs early in the scan so that the pattern is shifted forward quite rapidly.

Apart from its $O(mn)$ worst case execution time, the SF algorithm has another property that makes it undesirable in certain applications: it involves 'backing up' in the text string (line 4: $j := i$). This leads to inefficiencies if the whole text string is not available in memory and buffering operations are necessary. The next algorithm does not have this property and has a worst case running time of only $O(m+n)$.

THE KNUTH–MORRIS–PRATT ALGORITHM (KMP)

By making use of the theory of finite automata, Knuth, Morris and Pratt[1] devised an algorithm which constructs a deterministic pattern matching machine M_p to recognize the shortest instance of I^*p in the text string in $O(n)$ time. (I^* is the Kleene closure [Reference 2, Section 9.1] of the input alphabet I.) The time to construct M_p is $O(m)$.

Informally the algorithm can be described as follows: Initial pattern and text placement are as in the SF algorithm and scanning is done to the right. When a mismatch occurs, however, the pattern is shifted to the right in such a way that the scan can be restarted at the point of mismatch in the text string. No backtracking is therefore required.

Suppose the mismatch occurs between t_k and p_j (cf. Figure 1). By using a precomputed failure function $FAIL$, the pattern is shifted forward so that p_i, where $i = FAIL(j)$, is aligned with t_k and $p_1 \ldots p_{i-1}$ matches $t_{k1-i+1} \ldots t_{k-1}$, provided $p_i \neq p_j$ (cf. Figure 2). The scan is then restarted at t_k (comparing it with p_i). (Note that $0 \leqslant i = FAIL(j) < j$. If $i = 0$ then the pattern is shifted right past t_k so that t_{k+1} and p_1 are aligned, and the scan is restarted at t_{k+1}.) $FAIL(j)$ is thus the largest integer $i<j$ such that $p_1 \ldots p_{i-1}$ is a suffix of $p_1 \ldots p_{j-1}$ and $p_i \neq p_j$. $FAIL(j)$ for $0 \leqslant j \leqslant m$ can be computed in $O(m)$ time by the following algorithm:

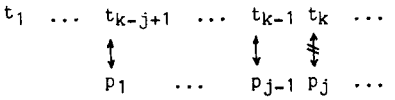

Figure 1. A mismatch at t_k

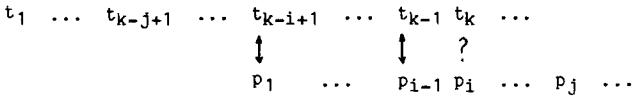

Figure 2. The pattern is shifted forward

Algorithm 2. Computing the failure table for KMP pattern matching

Input: PAT, a string of characters; $m>0$, the length of PAT.
Output: $FAIL$, an array representing the failure function for PAT.

```
1. j := 1; i := FAIL(1) := 0;
2. while ( j<m ) do
     /* FAIL(1) ... FAIL(j) is known - compute FAIL(j+1) */
3.   while ( i>0 and PAT(j) ≠ PAT(i) ) do
4.   end; i := FAIL(i);
5.   i := i+1; j := j+1;
6.   if ( PAT(j) = PAT(i) ) then
7.     FAIL(j) := FAIL(i)
     else
8.     FAIL(j) := i;
   end;
```

That algorithm 2 is $O(m)$ in the worst case can be proved if one observes that i is initially 0 and is incremented $m-1$ times by 1. Since i remains non-negative, line 4 (which always decreases i) can be executed at most $m-1$ times.

Once the failure function for a specific pattern has been computed, the actual pattern matching is straightforward.

Algorithm 3. Knuth–Morris–Pratt pattern matching

Input: PAT and TXT, arrays of pattern and text characters; $m>0$ and $n>0$ the number of characters in PAT and TXT respectively; $FAIL$ the failure function computed by algorithm 2.
Output: Success or failure indicator and if success, the location of the pattern in the text.

```
1. j := k := 1;
2. while ( k≤n ) do
3.   while ( j>0 and TXT(k) ≠ PAT(j) ) do
4.     j := FAIL(j);
     end;
5.   if ( j = m ) then
6.     return ( "pattern found at", k-m )
     else do
7.     k := k+1;
```

```
8.        j := j+1;
        end;
      end;
9.  return ( "pattern not found" );
```

Algorithm 3 can be shown to be $O(n)$ in the worst case by using the same arguments as for algorithm 2, with i replaced by j. Combining algorithms 2 and 3 it is possible to determine that a pattern of length m does not occur in the first i positions of a text string, in $O(m+i)$ steps.

THE BOYER–MOORE ALGORITHM (BM)

In both the SF and KMP algorithms the pattern is scanned from left to right and each text character is examined at least once. Boyer and Moore[3] noticed that more information about the text string can be obtained if scanning is done from right to left in the pattern. In fact, enough information can be gathered to determine that, without having to examine them, some text characters cannot be part of an occurrence of the pattern in the text string and therefore need not be considered in the search.

The BM algorithm initially aligns the pattern and text strings at their left, but then compares p_m and t_m, p_{m-1} and t_{m-1}, etc. until the pattern or a mismatch is found. If the latter occurs, a right shift of the pattern is computed using the larger of two precomputed functional values, $DELTA1$ and $DELTA2$. $DELTA1$ is based on the fact that, after the shift, the text character causing the mismatch must be aligned with a pattern character matching it. $DELTA2$ uses the principle that when the pattern is moved right, it must match over the characters already matched and a different pattern character must be brought over the text character at which the mismatch occurred.

$DELTA1$ is a function of the text character c at which the mismatch occurred. In particular:

$$DELTA1(c) = \textbf{if } c \textit{ does not occur in the pattern } \textbf{then } m,$$
$$\textbf{else } \textit{the smallest } j \textit{ such that } 0 \leqslant j < m \textit{ and } p_{m-j} = c.$$
$$= min\{\, j \mid j = m \textit{ or } (0 \leqslant j < m \text{ and } p_{m-j} = c) \,\}$$

Suppose the BM algorithm has already examined the first $r \geqslant 0$ characters in the text string without finding an occurrence of the pattern. Suppose further $t_{r+m-l+1} \ldots t_{r+m}$ matches $p_{m-l} \ldots p_m$ and $t_{r+m-l} \neq p_{m-l}$. Let $i = r+m$. $DELTA1(c)$ with $c = t_{i-l}$ is illustrated in Figure 3. The shift indicated by $DELTA1(t_{i-l})$ moves the pattern to the right to align p_{m-j} with t_{i-l} (i.e. $r = i-l+DELTA1(t_{i-l})$ is computed and the scan is restarted at t_r). Note that the pattern is shifted past the $m-j$ text characters following t_{i-m} without their being examined.

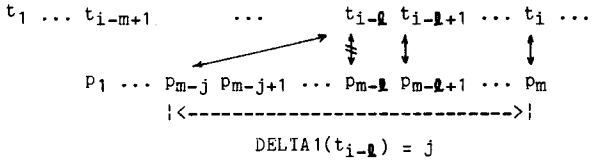

Figure 3. Illustration of $DELTA1$

DELTA2 is a function of the position in the pattern at which the mismatch occurred. Suppose, as in the case above, the mismatch occurred at p_{m-l}. Let $S = p_{m-l+1} \ldots p_m$ (cf. Figure 4). It is known that S matches $t_{i-l+1} \ldots t_i$ and that $t_{i-l}S$ does not match $p_{m-l}S$, so that the pattern may be shifted to the right to align the S in the text string with the rightmost occurrence of S in the pattern which is not preceded by p_{m-l} (cf. Figure 5). If a prefix $p_1 \ldots p_j$ of the pattern is a suffix of S, *then* $p_1 \ldots p_j$ is also regarded as an occurrence of S in the pattern. *DELTA2* is constructed in such a way that the shift described above can be accomplished by computing $r = i - l + DELTA2(m-l)$ and restarting the scan at t_r. For a mismatch at p_j, $0 < j \leqslant m$, $DELTA2(j)$ can thus be defined as follows:

$$DELTA2(j) = min\{\, k+m-j \mid k \geqslant 1 \text{ and } [(k \geqslant i \text{ or } p_{i-k} = p_i) \text{ for } j < i \leqslant m] \text{ and } [k \geqslant j \text{ or } p_{j-k} \neq p_j] \,\}$$

DELTA1 and *DELTA2* can be computed in $O(q+m)$ time by the following algorithm due to Knuth (reference 1, Section 8).

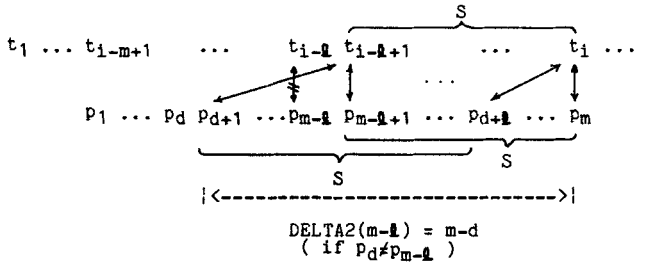

Figure 4. Illustration of DELTA2

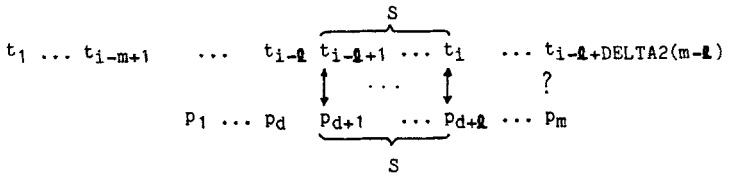

Figure 5. Pattern-shift according to DELTA2 of Figure 4

Algorithm 4. Computing DELTA1 and DELTA2

Input: *PAT*, a string of characters; $m > 0$, the length of *PAT*.
Output: Arrays *DELTA1* and *DELTA2*.

```
1. for every character c in the input alphabet do DELTA1(c) : = m;
2. for j : = m to 1 by −1 do /* compute DELTA1, initialize DELTA2 */
3.    if ( DELTA1(PAT(j)) = m ) then DELTA1(PAT(j)) : = m−j;
4.    DELTA2(j) : = 2*m−j;
   end;
   /* compute DELTA2 */
5. j : = m; t : = m+1;
6. while ( j>0 ) do
7.    f(j) : = t;
8.    while ( t≤m and PAT(j) ≠ PAT(t) ) do
```

```
 9.      DELTA2(t) : = min(DELTA2(t), m−j);
10.      t : = f(t);
       end;
11.    j : = j−1;
12.    t : = t−1;
     end;
13. for k : = 1 to t do DELTA2(k) : = min(DELTA2(k), m+t−k);
    /* The following steps were added by Mehlhorn[4] to ensure */
    /* correct values for DELTA2 in all cases*/
14. tp : = f(t);
15. while ( t≤m ) do
16.    while ( t≤tp ) do
17.      DELTA2(t) : = min(DELTA2(t), tp−t+m);
18.      t : = t+1;
       end;
19.    tp : = f(tp);
     end;
```

The details of the scan alogrithm are given by the following algorithm.

Algorithm 5: Boyer–Moore string matching

Input: PAT and TXT, arrays of pattern and text characters; $m>0$ and $n>0$ the number of characters in PAT and TXT respectively; arrays $DELTA1$ and $DELTA2$ as computed by alogorithm 4.

Output: Success or failure indicator and if success, the location of the pattern in the text.

```
 1. k : = m;
 2. while ( k≤n ) do
 3.    j : = m; /* j indexes the pattern and k the text */
 4.    while ( j>0 and TXT(k) ≠ PAT(j) ) do
 5.      j : = j−1;
 6.      k : = k−1;
       end;
 7.    if ( j = 0 ) then
 8.      return ( "pattern found at", k+1 )
       else /* shift the pattern */
 9.      k : = k+max( DELTA1(TXT(k)), DELTA2(j) );
     end;
10. return ( "pattern not found" );
```

Boyer and Moore showed that their algorithm can be expected to make fewer than $i+m$ references to the text string before finding the pattern at location i, whilst Knuth (Reference 1, Section 8) and also Guibas and Odlyzko[5] proved that the BM alogrithm is $O(n)$ in the worst case.

EMPIRICAL COMPARISON

The algorithms presented above were implemented in their most efficient forms (as suggested by their authors) and the following performance test executed:

20 search patterns of length m, for each m from 1 to 14, were randomly selected from an Afrikaans text string of length 5000. (This selection was done by using a pseudo-random number generator to indicate the starting positions of the patterns in the text string.) For each pattern length at least one pattern not occurring in the text string was generated.

The three string matching algorithms were used in turn to find each pattern. The following counts were taken as representative of the cost of the algorithms:

Algorithms 1, 3 and 5: The number of references to the text string before the pattern is found (or the text string is exhausted).

Algorithm 2: The number of comparisons between pattern characters.

Algorithm 4: The number of times the assignments in lines 1 and 13 and the tests in lines 3, 8 and 16 are executed.

The cost of algorithms 2 and 4 was added to that of algorithms 3 and 5, respectively, giving cost factors for the KMP and BM algorithms that incorporate the precomputations done. By dividing each cost factor c by i, the penetration of the pattern in the

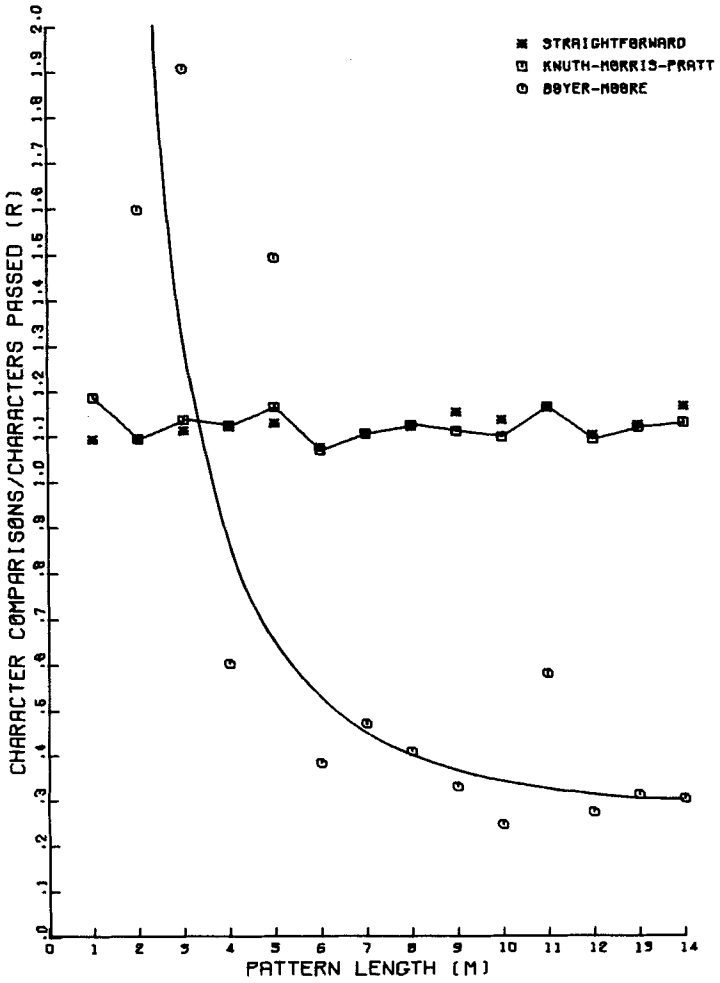

Figure 6. Number of comparisons per characters passed vs. the pattern length

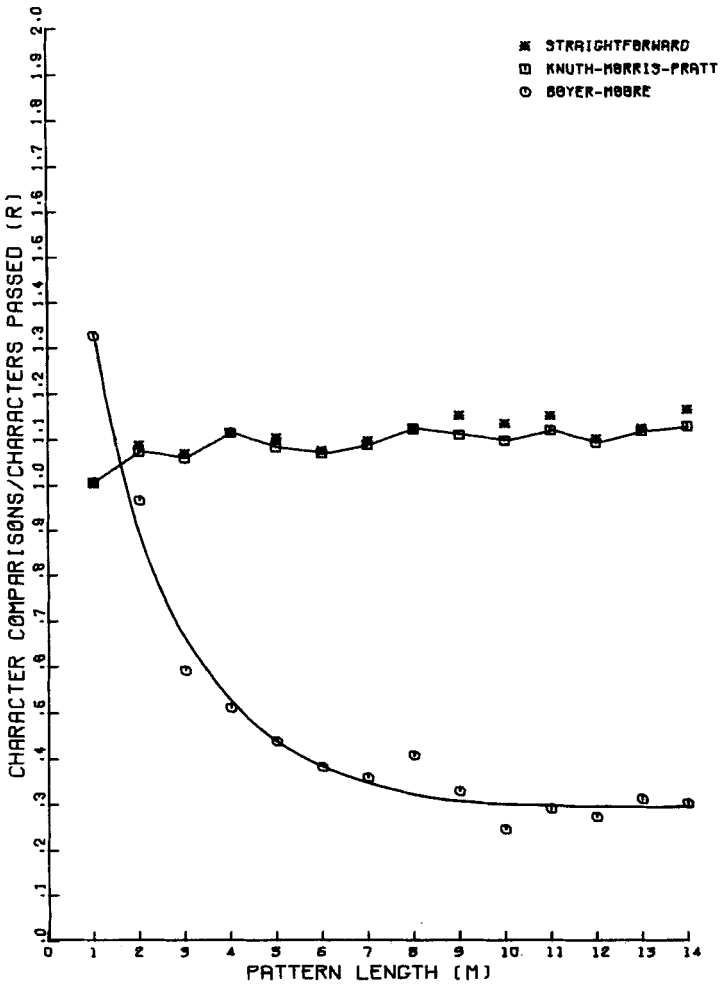

Figure 7. Number of comparisons per number of characters passed vs. the pattern length for penetrations exceeding 100 characters

text (i.e. the number of characters skipped in the text string before the pattern was found or the text was exhausted), a measure r independent of the penetration and implementation was obtained. These measures were averaged over the 20 patterns of each length and plotted against the pattern length. The resultant graphs are shown in Figures 6–8.

CONCLUSION

The performances of the SF and KMP algorithms do not differ much in the expected case, in that slightly more than one comparison is made for every character passed in the text string. Not reflected in the data however, are the worst case occurrences. In this case SF will perform much worse than is indicated by Figure 6 ($r = c/i$ would be close to m as $c \simeq i*m$ in the worst case), whereas the performance of the KMP algorithm would not show a significant change (r would be close to $1 + m/i$ as $c \simeq i + m$ in the worst case).

The performance of the BM algorithm is poor for short patterns (approx. less than 4 characters) and also when the pattern is found early (less than 100 positions) in the text

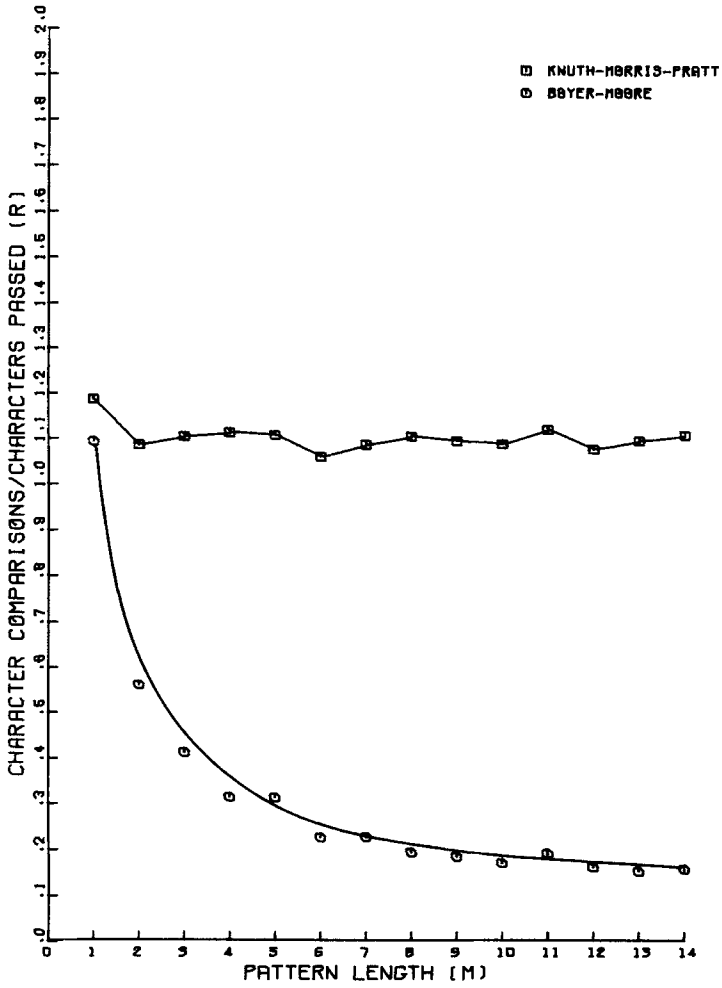

Figure 8. Number of comparisons per number of characters passed vs. the pattern length not considering the preprocessing

string. If both conditions occur simultaneously, performance as measured by the criteria above, is much worse than for the other algorithms. This is not because of the BM scan algorithm, but is a result of the computations needed to construct *DELTA*1. If the ASCII character set is used, at least 128 store instructions are needed in line 1 of algorithm 4. (This may not be the case in some implementations, as the initialization of *DELTA*1 can be done by using a block transfer instruction, which is much more efficient.)

In Figure 7 only cases where the penetration of the pattern was greater than 100 characters were considered (this constituted more than 85 per cent of the test cases). The 'sublinearity' of BM in the number of comparisons made for $m > 1$ can clearly be seen. For patterns of length 4 for example, only 0·51 comparisons are made per character passed or half of the characters passed in the text are not examined. If the precomputations are not taken into account (Figure 8), the comparisons drop to 0·31 per character passed or 2/3 of the characters passed over are never examined.

It is clear that none of the three algorithms presented is optimal under all circumstances. When patterns longer than 3 characters are searched, when the

expected penetration is more than 100 characters and when backing up in the text string will not pose problems, the BM algorithm is by far the best algorithm to use. Although the KMP algorithm gives a better analytical result than the SF algorithm, the worst case for the SF algorithm seldom appears in normal text. Therefore, because the precomputations necessary for the KMP algorithm may have adverse effects if the pattern penetration is not deep, the SF algorithm can be used in cases where the BM algorithm would be inefficient. Note however, that the probability of finding an early mismatch in the SF search (a prerequisite for SF efficiency), decreases with decreasing alphabet size. The KMP algorithm may therefore perform significantly better than the SF algorithm when comparing strings from a small alphabet, e.g. binary strings.

We conclude that, in a general text editor operating on lines of text (where buffering is not needed), the best solution for the string matching problem would be to use the BM algorithm for patterns longer than 3 characters and the SF algorithm in the other cases.

ACKNOWLEDGEMENT

I wish to thank Pieter Kritzinger for introducing me to the analysis of computer algorithms and for encouraging me to write this paper. Acknowledgement is also due to the anonymous referee for bringing to my attention Mehlhorn's correction of Knuth's *DELTA2* algorithm.

REFERENCES

1. D. E. Knuth, J. H. Morris, Jr and V. B. Pratt, 'Fast pattern matching in strings', *SIAM J. Computing*, **6**, 323–350 (1977).
2. A. V. Aho, J. E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass., 1976.
3. R. S. Boyer and J. S. Moore, 'A fast string search algorithm', *Comm. ACM.*, **20**, 762–772 (1977).
4. K. Mehlhorn, Private communication to D. E. Knuth (1977).
5. L. J. Guibas and A. M. Odlyzko, 'A new proof of the linearity of the Boyer–Moore string searching algorithms', *Proc. 18th Ann. IEEE Symp. Foundations of Computer Sci.*, 189–195 (1977).