

Tuning the Boyer–Moore–Horspool String Searching Algorithm

TIMO RAITA

*Department of Computer Science, University of Turku, Lemminkaisenkatu 14A,
SF-20520 Turku, Finland*

SUMMARY

Substring search is a common activity in computing. The fastest known search method is that of Boyer and Moore with the improvements introduced by Horspool. This paper presents a new implementation which takes advantage of the dependencies between the characters. The resulting code runs 25 per cent faster than the best currently-known routine.

KEY WORDS Design of algorithms String searching Pattern matching

INTRODUCTION

In the string-searching problem we have a pattern *pat*, of length *m*, all occurrences of which are to be found in a text string *text*, of length *n* (usually $n \gg m$). This problem has been studied extensively; see e.g. [Reference 1](#). One of the fastest known algorithms is that of Boyer and Moore.² The theoretical time complexity (measured in the number of symbol comparisons) of the method is $O(n + rm)$ in the worst case, where *r* is the total number of matches. The method is fast also in practice: experiments have shown that on the average the algorithm has a sublinear behaviour on the length of a typical text. The workspace needed is $m + c + O(1)$, where *c* is the size of the alphabet over which *text* and *pat* are written.

In the preprocessing phase of the Boyer–Moore algorithm, *pat* is scanned to form two tables which express how much the pattern is to be shifted forward in relation to the text when a match/mismatch is found. The first table defines a *match heuristic* and the second one an *occurrence heuristic*. The pattern is matched from right to left, i.e. starting from *pat*[*m*]. When a mismatch is found between *pat*[*i*] and the text symbol *x*, the match heuristic tells how much the pattern can be shifted in order to align the tested portion of the text with an identical portion in the pattern, i.e. it defines the rightmost repetition of *pat*[*i* + 1].. *pat*[*m*] in *pat*. The occurrence heuristic expresses the rightmost occurrence of *x* in the pattern. The pattern is shifted according to the larger shift given by the two heuristics.

The original algorithm has been analysed extensively, and several variants of it have been introduced.^{3–9} The fastest variant has been shown¹ to be that of Horspool.⁷ This method uses only the occurrence heuristic. Moreover, the text symbol that

aligns with `pat[m]` is always chosen (regardless of the position where the mismatch occurred) as the basis for the shift according to the occurrence heuristic:

```

procedure bmhsearch(var txt: txttype; n: integer; pat: pattype; m: integer);
type
  occtype = array [chr(0) .. chr(alphabetsize- 1)] of integer;
var
  i,j, k: integer;
  ch: char;
  occheur : occtype;
begin
  (* prepare the occurrence heuristic table d *)
  for ch := chr(0) to chr(alphabetsize- 1) do occheur[ch] := m;
  for j := 1 to m-1 do occheur[pat[j]] := m-j;

  (* Add sentinels to the front of the text and the pattern *)
  pat[0] := Symbol_not_in_text;
  txt[0] := Symbol_not_in_pat;

  (* The actual search loop *)
  i := m.

  while i <= n do
  begin
    k := i;
    j := m;
    while txt[k] = pat[j] do
    begin
      k := k-1;
      j := j-1;
    end;
    if j = 0 then writeln(output, 'Match at position ', k+1);
    i := i + occheur[txt[i]];
  end;
end;

```

NEW VARIANTS

Horspool's implementation performs extremely well when we search for a random pattern in a random text. In practice, however, neither the pattern nor the text is random; there exist strong dependencies between successive symbols.¹⁰ The dependencies may extend even over 30 symbols. They are strongest with respect to the nearest neighboring ones and weaken noticeably at word boundaries. This suggests that it is not profitable to compare the pattern symbols strictly from right to left: if the last symbol of the pattern matches with the corresponding text symbol, we should next try to match the first pattern symbol, because the dependencies are weakest between these two. If both symbols match, the next candidate is the middle symbol of `pat`. Thus we have the following general principle: *after each successful symbol match, choose a symbol to which the dependencies from all the symbols already*

probed are the weakest. Hence, if $pat[m]$ and $pat[1]$ match, we choose at step i ($i = 1, 2, \dots, p$) the symbols that have indices $km/2^i$ ($k = 1, 3, 5, \dots, 2^i - 1$), assuming $m = 2^p$. Note that the probing can be done in any order, because the match heuristic is not used and the symbol x which aligns with $pat[m]$ is used for shifting. If the shift is done according to the original scheme of Boyer and Moore, i.e. using the text symbol that caused the mismatch, the symbols should be processed at each step in decreasing index order to obtain larger shifts on the average.

Implementing only the initial phase of the principle, i.e. examining only the first and the last symbol of pat , the procedure body is changed as follows:

```

begin
  (* Prepare the occurrence heuristic table d *)
  for ch := chr(0) to chr(alphabetsize- 1) do occheur[ch] := m;
  for j := 1 to m-1 do occheur[pat[j]] := m-j;
  i := m;
  mminusone := m-1;
  last := pat[m];
  first := pat[1];

  (* Replace the first pattern symbol by a sentinel *)
  pat[1] := Symbol_not_in_text;

  (* The actual search loop *)
  while i <= n do
    begin
      if txt[i] = last then
        if txt[i-mminusone] = first then
          begin
            k := i-1;
            j := mminusone;
            while txt[k] = pat[j] do
              begin
                k := k-1;
                j := j-1;
              end;
            if j = 1 then writeln(output, ' Match at position ', k+ 1 );
          end;
          i := i + occheur[txt[i]];
        end;
      end;
    end;
  end;
end;

```

Note that j is tested for its final value only when at least $pat[1]$ and $pat[m]$ match. Also, there is no need for the auxiliary array slots $pat[0]$ and $text[0]$. The precondition of the procedure is that the length of pat is at least two (and can be omitted if we use the sentinel values $pat[0]$ and $text[0]$).

Including the test of the middle symbol, we have the code

```

midpoint := m div 2;

```

```

mminusmid := m-midpoint-1;
midchar := pattern[midpoint+ 1 ];

while i <= n do
begin
  if txt[i] = last then
    if txt[i-mminusone] = first then
      if txt[i-mminusmid] = midchar then
        begin

```

The disadvantage of this solution is that it compares the middle element twice, if at least the latter half of the pattern matches. It could be avoided, but the complicated logic would result in a slower program. In spite of the double checking, this variant performs better than the previous one if $m > 3$ (and only slightly worse with $m = 2$ or 3) due to the better selectivity of the if construction.

We have also implemented a version that tests all pattern symbols according to the given selection principle. No double checking is performed, but the preprocessing as well as the indirect references to the symbol positions in pat (actually, to the distances from the rear of the pattern) results in practice in a substantially longer processing time than those of the simplified routines.

The advantage of the version is that the knowledge of `Symbol_not_in_text` is no longer needed. If this is regarded as a severe restriction in the previous variants, we can substitute a for statement for the inner while loop:

```

begin
  k := i - 1 ;

  for j := mminusone downto 2 do
  begin
    if txt[k] <> pat[j] then goto 1;
    k := k - 1 ;
  end;
  writeln(output, ' Match at position ', k+ 1 );
end;

1:
  i := i + occheur[txt[i]];

```

The for loop makes at most $m - 3$ comparisons more ($m - 2$ in controlling the loop termination; one is saved because the j test can be omitted) than the while implementation. This does not reduce the speed of processing much, because the compiler can usually generate efficient code for the loop control,

EXPERIMENTS

The variant that tests the last, first and middle symbol before entering the inner while loop has turned out to be the fastest of the above routines. Its selectivity is very good compared to the one that checks only the first and last symbols: of all probing positions, the former chooses only 0.02–0.5 per cent for a closer examination

and 30-100 per cent of these were matches. The corresponding figures for the latter one were 0.3-1.5 per cent and 6-78 per cent. The tests were performed on a text of a technical report, written in English. The length of the text was 29,550 characters. The alphabet used was ASCII, implying a theoretical alphabet-size 128, the actual size being 85. Patterns of length 2-20 were selected randomly from the text. The search was repeated 30 times with each pattern length. Figure 1 depicts the results of the experiment.

The results show that the variant is 21-27 per cent faster than the original scheme with all pattern lengths. Because of the pattern-selection technique, at least one occurrence of pat is always found. However, the test results with non-occurring (English) patterns were very similar to those of Figure 1. It is evident that patterns that have a frequently-occurring suffix, e.g. '-ion' and '-ed' (possibly appended with a space character) are the ones where the profits of the new variant are most significant. Its performance improves also when m increases, because of the weakening dependencies. On the other hand, it deteriorates when the alphabet-size decreases, because the probability of $pat[1]$, $pat[m/2]$ and $pat[m]$ being equal to the corresponding text symbols increases. However, we believe that if there exist short-range dependencies in text, this phenomenon does not show up until the alphabet-size is as small as 2.

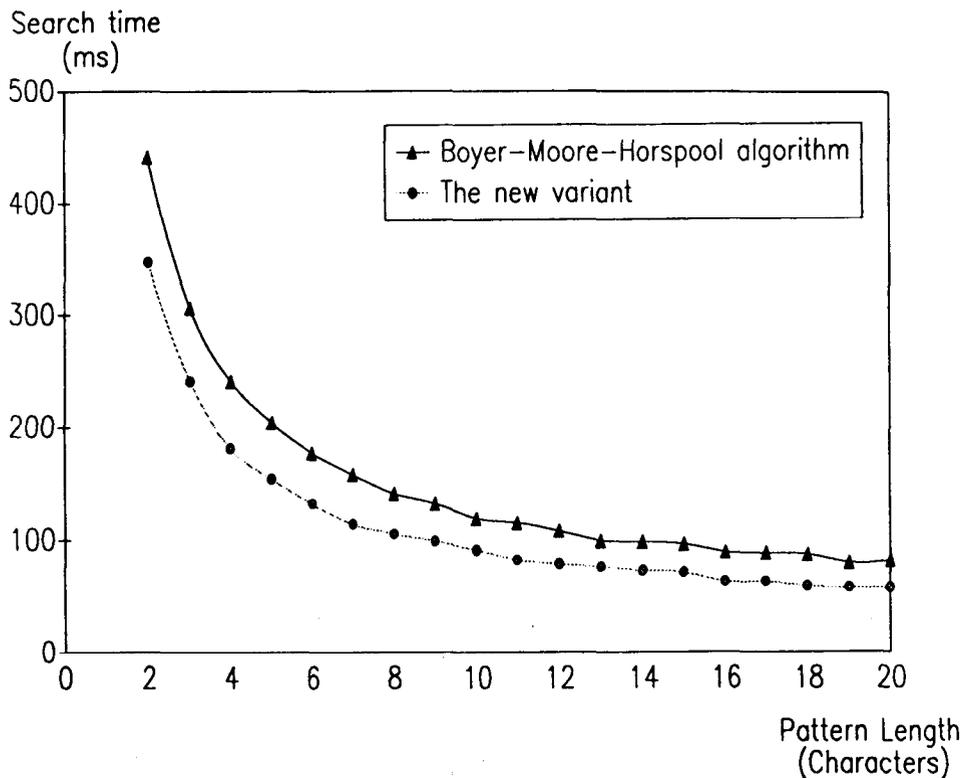


Figure 1. The performance of the Boyer-Moore-Horspool algorithm and the new variant. The tests were done on a Micro Vax-11 microcomputer using Pascal

ACKNOWLEDGEMENTS

This work was partly supported by the Academy of Finland. The author wishes to thank Jukka Teuhola for his valuable comments.

REFERENCES

1. R. A. Baeza-Yates, 'Algorithms for string searching: a survey', *SIGIR Forum*, **23**, (3,4), 34–58 (1989).
2. R. S. Boyer and J. S. Moore, 'A fast string searching algorithm', *Comm. ACM*, **20**, (10), 762–772 (1977).
3. A. Apostolic and R. Giancarlo, 'The Boyer–Moore–Galil string searching strategies revisited', *SIAM J. Comput.*, **15**, (1), 98–105, (1986).
4. R. A. Baeza-Yates, 'Improved string searching', *Softw. Pratt. Exp.*, **19**, 257–271 (1989).
5. Z. Galil, 'On improving the worst case running time of the Boyer–Moore string matching algorithm', *Comm. ACM*, **22**, (9), 505–508 (1979).
6. L. J. Guibas and A. M. Odlyzko, 'A new proof of the linearity of the Boyer–Moore string searching algorithm', *SIAM J. Comp.*, **9**, (4), 672–682 (1980).
7. R. N. Horspool, 'Practical fast searching in strings', *Softw. Pratt. Exp.*, **10**, 501–506 (1980).
8. W. Rytter, 'A correct preprocessing algorithm for Boyer–Moore string-searching', *SIAM J. Comput.*, **9**, (3), 509–512 (1980).
9. R. Schaback, 'On the expected sublinearity of the Boyer–Moore algorithm', *SIAM J. Comput.*, **17**, (4), 648–658 (1988).
10. S. Guiasu, *Information Theory with Applications*, McGraw-Hill, London, 1977.