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[ml 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 shown[1] to be that of Horspool.[7] This method uses only the occurrence heuristic. Moreover, the text symbol that

0038–0644/92/100879–06$08.00 *Received 9 May 1991*

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,... ,p) the symbols that have indices $km/2^i$ (k = 1,3,5,... ,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;
```

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;
```

```
mmminusmid  :  =  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.

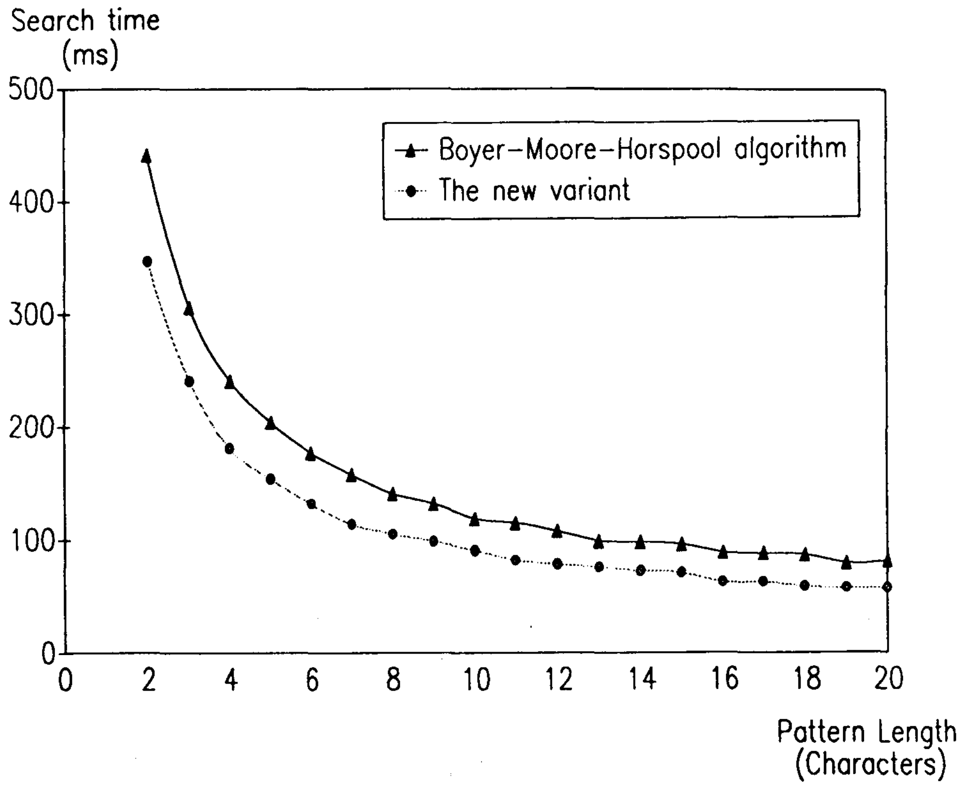

Figure I. The performance of the Boyer–Moore–Horspool algorithm and the new variant. The tests were done on a Micro Vax-11 microcomputer using Pascal

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS

Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS

Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS

Sync your system to PACER to automate legal marketing.

WHAT WILL YOU BUILD? | **sales@docketalarm.com** | **1-866-77-FASTCASE**

fastcase®
Smarter legal research.