On Improving the Worst Case Running Time of the Boyer-Moore String Matching Algorithm.

Conference Paper *in* Communications of the ACM · July 1978

DOI: 10.1007/3-540-08860-1_18 · Source: DBLP

Zvi Galil
Georgia Institute of Technology

Programming Techniques — R. Rivest, Editor

On Improving the Worst Case Running Time of the Boyer-Moore String Matching Algorithm

Zvi Galil
Tel-Aviv University

It is shown how to modify the Boyer-Moore string matching algorithm so that its worst case running time is linear even when multiple occurrences of the pattern are present in the text.

Key Words and Phrases: computational complexity, linear time, worst case, string matching, periodicity

CR Categories: 3.74, 4.40, 5.25

Introduction

The string matching problem is to find an [all] occurrence[s] of a pattern (a string) in a text (another string), or to decide that none exists. A recent algorithm that solves the problem is the Boyer-Moore algorithm [1]. Unlike its predecessors, the Knuth-Morris and Pratt algorithm (KMP) [3] and the straightforward algorithm, it compares the pattern with the text from the right end of the pattern. Whenever a mismatch occurs, it shifts the pattern according to a precomputed table. In the case that the text character positioned against the last character in the pattern does not appear in the pattern, we can immediately shift the pattern right a distance equal to the size of the pattern. Thus, when the alphabet size is large, we need to inspect only about n/m characters of the text on the average (where n and m are the sizes of the text and the pattern respectively). All previous algorithms inspect each character of the text at least once in every case. However, the worst case behavior is not linear. Boyer and Moore showed that their algorithm is $O(nm)$. The running time can essentially be proportional to $n + rm$, where r is the number of occurrences of the pattern in the text [3].

Knuth [3] describes two variations of the Boyer-Moore algorithm that preserve its main asset, namely excellent average time. (In fact one of them has even better average time.) These two variations are also linear time in the worst case. They, however, lose two of the other nice features of the original algorithm. The first one is simplicity and the second is linear preprocessing time. So if the preprocessing time is taken into account, they are not linear time in the worst case.

In this paper we present a third alternative. Our solution preserves the three good properties of the original algorithm and is linear time in the worst case. Surprisingly, it will be obtained from the original algorithm by making a trivial change. Less trivial will be some technical details implying its correctness.

In Section 1 we review the Boyer-Moore algorithm (we mainly use Guibas and Odlyzko's excellent exposition [2]), Knuth's variations, and some of the properties of these algorithms. In Section 2 we introduce our variation of the algorithm and prove some of its properties.

1. The Boyer-Moore Algorithm and Its Known Variations

The string matching problem is the following. Given an array *text*[1:n] representing the input text, and an array *pattern*[1:m] representing the pattern being sought, find all occurrences of the pattern in the text.

The Boyer-Moore algorithm solves this problem by repeatedly positioning the pattern over the text and attempting to match it. For each positioning that arises, the algorithm starts matching the pattern against the text from the *right* end of the pattern. If no mismatch occurs, then the pattern has been found. Otherwise the algorithm computes a shift; that is, an amount by which the pattern will be moved to the right before a new matching attempt is undertaken.

In the program below we keep two pointers: one (i) to the current character of the pattern being examined, and the other (j) to the position of the text which is aligned with the last character of the pattern. Thus at that instant characters at positions i through m of the pattern are aligned with characters at positions $j - m + i$ through j of the text.

```
j := m;
do while j ≤ n
  begin
      do i := m to 0 by −1 until pattern[i] ≠ text[j − m + i]
      if i = 0 then [output (match at j − m + 1); j := j + 1]
               else j := j + s(text[j − m + i], i)
  end
```

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

The research for this paper was supported in part by the Israel Commission for Basic Research.

Author's present address: Z. Galil, Dept. of Mathematical Sciences, Computer Science Division, Tel-Aviv University, Ramat-Aviv, Tel-Aviv, Israel.

The shift $s(\mathrm{CH}, i)$ is computed using two heuristics. The *match* heuristic is based on the idea that when the pattern is moved right, it has to (1) match over all the characters previously matched, and (2) bring a *different* character over the character of the text that caused the mismatch. Thus the *match shift* is defined by

$$s.\mathrm{match}(i) = \min\{t \mid t \geq 1 \text{ and } (t \geq i \text{ or } \mathbf{pattern}[i-t] \neq \mathbf{pattern}[i]) \text{ and } ((t \geq k \text{ or } \mathbf{pattern}[k-t] = \mathbf{pattern}[k]) \text{ for } i < k \leq m)\}.$$

Secondly, the *occurrence heuristic* uses the fact that we must bring over CH = text$[j - m + i]$ (the character that caused the mismatch), the first character of the pattern that will match it. Thus the *occurrence shift* is defined by

$$s.\mathrm{occ}(\mathrm{CH}, i) = \min\{t - m + i \mid t = m \text{ or } (0 \leq t < m \text{ and } \mathbf{pattern}[m-t] = \mathrm{CH})\} \equiv i + s.\mathrm{occ}'(\mathrm{CH}).$$

Both shifts can be obtained from *precomputed* tables based solely on the pattern and the alphabet used. The match heuristic requires a table of length equal to the pattern length, while the occurrence heuristic requires a table of size equal to the alphabet size. Given these two shifts, the Boyer-Moore algorithm chooses the largest one. Thus s is defined by

$$s(\mathrm{CH}, i) = \max\{s.\mathrm{match}(i), s.\mathrm{occ}(\mathrm{CH}, i)\}.$$

Note that when an occurrence which ends at position j is found, the next search starts at position $j + 1$. A simple observation is that in this case one can apply the shift strategy too. This idea is not new. It was applied to the KMP. By defining $s(\mathrm{CH}, 0) = s.\mathrm{match}(0)$ for every CH, one can use the following version of the Boyer-Moore algorithm.

```
BM:  $j := m$;
     do while $j \leq n$
       begin
         do $i := m$ to 0 by $-1$ until pattern$[i] \neq$
            text$[j - m + i]$
         if $i = 0$ then output(match at $j - m + 1$)
         $j := j + s(\text{text}[j - m + i], i)$
       end
```

In the definition of $s(\mathrm{CH}, 0)$ we ignored the occurrence heuristic because in the case $i = 0$ there was no mismatch. In this case (since $t \geq i = 0$), we have

$$s(\mathrm{CH}, 0) = \min\{t \mid t \geq 1 \text{ and } \mathbf{pattern}[k] = \mathbf{pattern}[k+t] \text{ for } 1 \leq k \leq m - t\} \tag{1}$$

The preparation of s.occ obviously takes $O(m + q)$ steps, where q is the alphabet size. The preparation of s.match takes $O(m)$ (including s.match(0)) because it is essentially identical to the table of shifts calculated in the KMP. One just has to consider the reverse of the pattern.

In [3] Knuth proved:

THEOREM 1. *If the text does not contain any occurrence of the pattern, the total number of comparisons in the Boyer-Moore algorithm is at most 7n.*

The proof of this theorem is quite involved. Recently, Guibas and Odlyzko [2] improved the bound to $4n$ and conjectured that it is $2n$. Their proof is nontrivial too. In contrast the proof of a $2n$ bound for the KMP is straightforward.

As a result of Theorem 1, Knuth showed:

COROLLARY 2. *The worst case running time of the Boyer-Moore algorithm is $O(n + rm)$ character comparisons, if the pattern occurs r times in the text.*

PROOF. Let $T(n, r)$ be the worst case running time as a function of n and r, when m is fixed. Theorem 1 implies that $T(n, 0) \leqq 7n$. Furthermore, if $r > 0$ and the first appearance of the pattern ends at position n_0, we have

$$T(n, r) \leqq 7(n_0 - 1) + m + T(n - n_0 + m - 1, r - 1). \tag{2}$$

It follows that $T(n, r) \leqq 7n + 8rm - 14r$. □

Note that these proofs hold for the BM.

When the Boyer-Moore algorithm implicitly shifts the pattern to the right, it forgets all it "knows" about characters already matched; this is why Theorem 1 is not trivial. Knuth [3] describes a way to make the algorithm remember. He adds a set finite number of states that record all characters in positions $j - m + 1$ through j in the text that are known to match the corresponding positions in the pattern. This version of the BM inspects each character of the text at most once and thus is linear time in the worst case. In addition to the loss in simplicity, the main drawback of this approach is that the number of states can be very large. (An immediate upper bound on this number is 2^m, but it is not known how large it can be.) An example is given of a pattern of size m that requires at least $m^2/2$ states!

Another variation suggested by Knuth [3] is the following: Choose $r = [2 \log_q m]$. If text$[m - r] \ldots$ text$[m]$ does not occur in the pattern, shift the pattern to the right $m - r$ places; otherwise use the KMP to shift the pattern $m - r$ places. This algorithm has a linear-time worst case running time and average running time of $O(n(\log_q m)/m)$. (Recently Yao [7] showed that if $n \geq 2m$, then almost all patterns of size m require $\Omega(n(\log_q m)/n)$ in the average no matter what algorithm is used.) This variation is not as simple as the BM or the KMP. The pattern is sometimes checked from right to left and some other times (though less frequently) from left to right. Furthermore in order to be able to find out whether text$[m - r] \ldots$ text$[m]$ appears in the pattern, an index tree like the one used by Wiener [6] or McCreight [5] is needed. The time to prepare such a tree is Cqm, where C is quite large. Moreover, for large q (e.g. $q = m$) this bound is not linear.

2. The Improved Algorithm

Before we present our variation of the BM, we need to introduce certain notation and prove some properties of strings.

Given two strings, w and a prefix u of w, we say that u is a *period* of w if w is a prefix of u^k for some $k \geq 1$. Equivalently u is a period of w if $w = uv$ and v is a prefix of w. We will say that w is *periodic* if it has a period of size not larger than $|w|/2$ (i.e. $k \geq 2$ in the definition above).

Periodicity Lemma ([3], [4]): If w has periods of sizes p and q and $|w| \geq p + q$, then w has a period of size $gcd(p, q)$.

Let r, n, and m be as in Section 1, and let the pattern be x.

LEMMA 3. *If $r > 2n/m$, then the pattern is periodic.*

PROOF. If $r > 2n/m$, then there are two occurrences of x at position p and $p + k$ for some $1 \leq p \leq n - m + 1$ and $k \leq m/2$. Let $x = uv$ where u is the prefix of x of size k. It follows that v is a prefix of x (see Figure 1) and hence u is a period of x, the size of which is not larger than $|x|/2$; i.e. x is periodic. □

COROLLARY 4. *If x is not periodic, then the BM is linear-time in the worst case.*

PROOF. The time is bounded by $C(n + rm)$ and $r \leq 2n/m$; hence the bound is $3Cn$. □

Now, assume that the pattern is periodic, u is its smallest period, and $|u| = k (k \leq m/2)$. It follows from (1) in Section 1 and the definitions above that $k = s(\text{CH}, 0)$, and hence it is actually computed in the preprocessing stage. Two occurrences at positions p and q will be said to be *close* if $|p - q| \leq m/2$. We define the relation of *neighborhood* to be the reflexive transitive closure of closeness, and let a *chunk* be a maximal class of neighbors. Let p be the largest position in a chunk. The two facts below follow from the definition of a chunk.

Fact 1. A new chunk cannot start before position $p + m/2$.

Fact 2. The number of chunks cannot be larger than $2n/m$.

LEMMA 5. *Assume the positions of the occurrences in a chunk in increasing order are $p_1, p_2, \ldots, p_r$ for some $r > 1$. Then, for $1 < i \leq r$, $p_i - p_{i-1} = k$.*

PROOF. Consider the occurrences at positions p_{i-1} and p_i. By the definition of a chunk they must be close. Hence $k' \equiv p_i - p_{i-1} \leq m/2$, and as in Figure 1, the pattern has a period of size k'. Since $k' + k \leq m$, by the Periodicity Lemma it has a period of size $gcd(k', k)$ and by the minimality of k, k must divide k', i.e. $k' = lk$. But if $l > 1$, it is easy to see that there is an occurrence of the pattern at $p_{i-1} + k < p_i$ – contradiction. Hence $k' = k$. □

The main idea of the algorithm below is to discover each chunk at a time instead of each occurrence at a time as the BM does. Thus the algorithm does not forget large portions of overlapping occurrences. Assume $x = uuuu'(u = u'u'')$ and somewhere in the text we have $au^7\hat{u}b$ where $\hat{u}$ is a proper prefix of u' but $\hat{u}b$ is not a prefix of u and u does not end with a. (The periodicity starts after the a and ends before the b.)

The algorithm is identical to the BM as long as no

Fig. 1.

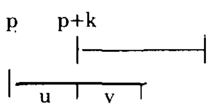
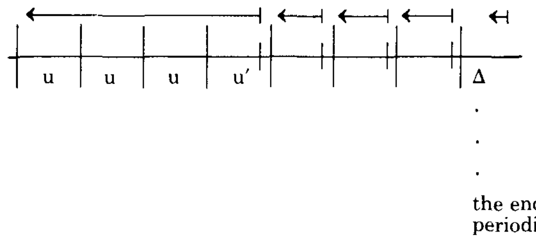


Fig. 2.



occurrence of the pattern is found. When an occurrence is found, the search for the next occurrence is started at position $j + k$ as in the BM. But in order to find the next occurrence in a chunk, it is enough to check only the last k characters of the pattern. If they match the corresponding symbols of the text, another occurrence in the chunk is found without checking the part overlapping the previous occurrences. In this way the second, third, and fourth occurrences are found in Figure 2. In case of a mismatch (as in the attempt to find a fifth occurrence in the chunk), the search for the first occurrence in the next chunk is started by continuing with the BM from this point. Fact 1 guarantees that we cannot miss any occurrence because $k \leq m/2$. In case $k \ll m/2$ we can shift much more than the BM shifts in the first mismatch after a chunk has been found (namely, $m/2 - k$). But since this change will complicate the algorithm, we do not include it here.

To implement the discussion above, it is enough to make a small change in the BM. The index i which always runs in the BM from m to 0 will now run down to 0 only for finding the first occurrence in the chunk and to $l_0 \equiv m - k + 1$ otherwise l_0 is computed during the preprocessing stage for essentially no extra cost.

The procedure BM′ – the modified BM is given below.

```
BM′: $j := m$; $l := 0$;
    do while $j \leq n$
      begin
        do $i := m$ to $l$ by $-1$ until pattern[$i$] $\neq$ text[$j - m + i$]
        if $i = l$ then [output(match at $j - m + 1$); $l := l_0$; $i := 0$]
              else $l := 0$
        $j := j + s$(text[$j - m + i$], $i$)
      end
```

THEOREM 6. *The BM′ is $O(n)$ in the worst case.*

PROOF. We show that the worst case running time of the BM′ is $O(n + r'm)$, where r' is the number of chunks in the text. The theorem then follows immediately from Fact 2. Let $T(n, r')$ be the worst case running time as a function of n and r', when m is fixed. Theorem 1 implies that $T(n, 0) \leq 7n$. In the case $r' > 0$, let n_0 [n'_0] be the last position of the first [last] occurrence in the first chunk. As in Theorem 1, the time taken until $j = n_0$ is bounded by $7(n_0 - 1)$. The time taken until $j > n_0'$ for the first time is therefore bounded by $7(n_0' - 1) + m$ because each position in the chunk is inspected exactly once. Hence eq. (2) in Section 1 holds when we replace n_0 and r by n_0' and r', respectively, and thus $T(n, r') \leq 7n + 8r'm - 14r'$ as in Theorem 1. □

Note that using the Guibas and Odlyzko bound, it follows that $T(n, r') \leq 4n + 5r'm \leq 14n$, and if their conjecture is true, $T(n, r') \leq 2n + 3r'm \leq 8n$.

Theorem 6 holds also for a nonperiodic pattern. In this case all chunks contain only one occurrence. However, it is probably better to use the BM in this case. (The BM′ changes l very frequently.) Note that if during preprocessing we set l_0 to 0 when $k > m/2$ (i.e. the pattern is not periodic), then the BM will be obtained from the BM′ by code optimization. One can view the BM′ as a simplified version of the first variation suggested by Knuth that uses states to remember overlapping sections. The difference is that the BM′ always has at most two states, depending on the value of l.

Received March 1978; revised January 1979

References
1. Boyer, R.S., and Moore, J.S. A fast string searching algorithm. *Comm. ACM 20,* 10 (Oct. 1977), 762–772.
2. Guibas, L.J., and Odlyzko, A.M. A new proof of the linearity of the Boyer-Moore string searching algorithms. Proc. 18th Ann. IEEE Symp. Foundations of Comptr. Sci., 1977, pp. 189–195.
3. Knuth, D.E., Morris, Jr., J.H., and Pratt, V.B. Fast pattern matching in strings. *SIAM J. Comptng. 6,* 2 (1977), 323–350.
4. Lyndon, R.C., and Schutzenberger, M.P. The equation $a^M = b^N c^P$ in a free group. *Michigan Math. J. 9* (1962), 289–298.
5. McCreight, E.M. A space economical suffix tree construction algorithm. *J. ACM 23,* 2 (April 1976), 262–272.
6. Weiner, P. Linear pattern matching algorithm. Proc. 14th Ann. IEEE Symp. Switching and Automata Theory, 1973, pp. 1–11.
7. Yao, A.C.C. The complexity of pattern matching for a random string. Mss., Comptr. Sci. Dept., Stanford U., Stanford, Calif., 1977.

Programming Techniques — R.L. Rivest, Editor

An Optimal Insertion Algorithm for One-Sided Height-Balanced Binary Search Trees

Kari-Jouko Räihä
University of Helsinki

Stuart H. Zweben
Ohio State University

An algorithm for inserting an element into a one-sided height-balanced (OSHB) binary search tree is presented. The algorithm operates in time $O(\log n)$, where n is the number of nodes in the tree. This represents an improvement over the best previously known insertion algorithms of Hirschberg and Kosaraju, which require time $O(\log^2 n)$. Moreover, the $O(\log n)$ complexity is optimal.

Earlier results have shown that deletion in such a structure can also be performed in $O(\log n)$ time. Thus the result of this paper gives a negative answer to the question of whether such trees should be the first examples of their kind, where deletion has a smaller time complexity than insertion. Furthermore, it can now be concluded that insertion, deletion, and retrieval in OSHB trees can be performed in the same time as the corresponding operations for the more general AVL trees, to within a constant factor. However, the insertion and deletion algorithms for OSHB trees appear much more complicated than the corresponding algorithms for AVL trees.

Key Words and Phrases: insertion, one-sided height-balanced trees, height-balanced trees, binary trees, search trees

CR Categories: 3.73, 3.74, 4.34, 5.25, 5.31

The work of K-J. Räihä was supported by the Academy of Finland. The result presented in this paper was discovered by the authors independently. They have since combined and edited their original manuscripts to produce this one.

Authors' addresses: K-J. Räihä, Department of Computer Science, University of Helsinki, Tukholmankatu 2, SF-00250 Helsinki 25, Finland; S.H. Zweben, Department of Computer and Information Science, The Ohio State University, Columbus, OH 43210.