

A New Method to Obtain the *shift-table* in Boyer-Moore's String Matching Algorithm

Yang Wang

Computer Science Dept., Missouri State University

Springfield, MO 65897

USA

yangwang@missouristate.edu

Abstract

The Boyer-Moore algorithm uses two pre-computed tables for searching a string: skip, which utilizes the occurrence heuristic of symbols in a pattern, and shift, which utilizes the match heuristic of the pattern. Researchers have pointed out that the difficulty of understanding the computation of the shift table has hindered utilization of the algorithm in a wider range of applications. This paper describes an alternative way to compute the shift table. We believe that the new method is more intuitive and straightforward both conceptually and logically than the original method, and thus, easier to understand and to implement. Also, the new method has $O(m)$ complexity in both required space and time for a pattern of length m . Therefore, it preserves the high performance of the Boyer-Moore algorithm.

1. Introduction

One common string-matching problem is to search for an occurrence of a given pattern string as a substring of a longer string of symbols. This problem used to be common only in the keyword search facility of text-editing programs. Since then, it has arisen frequently in various kinds of applications, such as DNA sequence matching, data compression, and information security.

We give some relevant definitions below. Let Σ be a finite set of symbols and Σ^* (the Kleene closure of Σ) be the set of all finite-length strings formed with the symbols in Σ , including the empty string ϵ . The length of a string x is denoted $|x|$. A string x of length m may

be written as $x_1x_2\cdots x_m$, where x_i , $1 \leq i \leq m$, is the i^{th} symbol of the string. A substring of x is a string composed of a set of contiguous symbols within x . We use $x(i, j)$, $1 \leq i \leq j \leq m$, to denote the substring of x , $x_i x_{i+1} \cdots x_j$. A suffix of x is an empty string ϵ or a substring $x(i, m)$, $1 \leq i \leq m$. The above string problem can be formalized as: given a pattern string $x \in \Sigma^*$ with $|x|=m$, and another string $y \in \Sigma^*$ with $|y|=n$, where $0 < m \leq n$, if $\exists i$, $1 \leq i \leq n-m+1$, such that $y(i, i+m-1) = x(1, m)$, return i , otherwise return 0. We refer to $y(i, i+m-1)$ as a matching substring of string y to string x .

Many researchers have tackled this problem and a number of algorithms have been introduced [1~6]. Extensive analysis and comparison on the performance of the algorithms have been conducted. A common conclusion is that "the Boyer-Moore approach provides, on the whole, a significantly faster search method" [1]. Unfortunately, "Although providing a high performance, the degree to which the Boyer-Moore approach has been put into practice may have been curbed to a certain extent by conceptual difficulties in the preprocessing, particularly with the match heuristic" [1]. Sedgewick [7] also pointed out "both the Knuth-Morris-Pratt and the Boyer-Moore algorithms require some complicated preprocessing on the pattern that is difficult to understand and has limited the extent to which they are used". Horspool [5] noticed that "many programmers may not believe that the Boyer and Moore algorithm (if they have heard of it) is a truly practical approach." Hume and Sunday [8] also mentioned that "partially because the best algorithms presented in the literature are difficult to understand and to implement, knowledge of fast and practical algorithms is not commonplace".

This paper describes an alternative way to compute the shift table that utilizes the match heuristic of a pattern. We believe that this new method is conceptually more intuitive, and thus, easier to understand and to implement. Also, our method has the same $O(m)$ space and time complexities for a pattern of length m as the existing method. Therefore, it preserves the high performance of the Boyer-Moore algorithm. A brief description of Boyer-Moore algorithm is given in Section 2. The definition of a new array *suffixLength* with its theoretical properties and the algorithm to obtain the array from a pattern string are discussed in Section 3. An algorithm computing the shift table from *suffixLength* is described in Section 4. Finally, our conclusion is given in Section 5. (Due to space limits, the proofs for theorems and corollaries are omitted and available only upon request).

2. Outline of Boyer-Moore algorithm

With the Boyer-Moore search algorithm, the pattern x is slid across the string y from left to right, but the actual symbol comparisons between x and y are carried out from the pattern's right to left. If a mismatch occurs at $x_j \neq y_i$, $1 \leq i \leq n$, $1 \leq j \leq m$, then, $y(i-j+1, i+m-j)$ can no longer be a suffix of a matching substring, the pattern x may be shifted right to start another round of comparisons. The shift is based on two considerations. The first consideration utilizes the match heuristic of the pattern x . Let

$$s = \begin{cases} \min\{t \mid t \geq 1 \text{ and } x_{m-t} \neq x_m\}, & \text{if } j=m \\ \min\{t \mid t \geq 1 \text{ and } ((t \geq j \text{ or } x_{j-t} \neq x_j) \text{ and } ((t \geq k \text{ or } x_{k-t} = x_k) \text{ for } j < k \leq m))\}, & \text{otherwise} \end{cases}$$

Then, pattern x can be shifted right s positions. After the shift, symbol x_m will align with $y_{i+(m-j)+s}$, and comparisons will be resumed from right to left starting with x_m and $y_{i+(m-j)+s}$. The change in the value of i is $(m-j)+s$ and is stored in *shift[j]* before a search. This computation uses an auxiliary array f that is defined as

$$f = \begin{cases} m+1, & \text{if } j=m \\ \min\{i \mid j < i \leq m \text{ and } x(i+1, m) = x(j+1, j+m-i)\}, & \text{otherwise} \end{cases}$$

It is challenging to clearly understand the relationship between *shift* and f , and thus, the algorithm to compute *shift* using f . The second consideration utilizes the occurrence heuristic of the pattern x , which can be shifted right $j-(m-q)$ positions if the right most occurrence of symbol y_i in x is at the location $m-q$; or shifted right j positions, if y_i does not

occur in x . Again, the change in the value of i is computed and stored in *skip[y_i]* before a search. The actual shift size for x is the greater of *shift[j]* and *skip[y_i]*.

3. Array *suffixLength*

First, we will introduce the definition of the array *suffixLength* and state some of its theoretical properties. Then, we will describe an algorithm to compute array *suffixLength* from a given pattern in a linear time.

3.1. Definitions and properties

Definition 1. For a given pattern string $x(1, m)$, *suffixLength* is an array of size m , where *suffixLength_i*, $1 \leq i \leq m$, is the length of the longest string such that $x(j, i) = x(m-(i-j), m)$, $1 \leq j \leq i$.

In other words, $x(j, i)$ is the longest substring ending at x_i that is the same as a suffix of x . Thus, for $1 \leq i \leq m$,

$$\text{suffixLength}_i = \begin{cases} 0 & \text{if } x_i \neq x_m \\ \max\{i-j+1 \mid x(j, i) = x(m-(i-j), m) \text{ and } 1 \leq j \leq i\} & \text{otherwise} \end{cases}$$

Clearly, for a pattern of length m , we always have *suffixLength_m* = m .

Definition 2. Given an array $A = \{a_1, a_2, \dots, a_m\}$, if $a_i = i$, $1 \leq i \leq m$, we refer to i as a value-equal-index (VEI) element of A .

Again, obviously, for a pattern of length m , *suffixLength_m* is a VEI element of *suffixLength*. Based on the definition, we have the following corollaries.

Corollary 1. For $1 \leq i \leq m$, $0 \leq \text{suffixLength}_i \leq i$ and *suffixLength_i* $\neq 0$ if and only if $x_i = x_m$.

Corollary 2. For $1 \leq i \leq m$, i is a VEI element of *suffixLength*, i.e., *suffixLength_i* = i , if and only if $x(1, i) = x(m-i+1, m)$.

Corollary 3. If i , $1 \leq i \leq m$, is a VEI element of *suffixLength*, then, it is the left most occurrence of i in array *suffixLength*.

Corollary 4. For $1 \leq i \leq m$, *suffixLength_i* = k and $0 < k < i$ if and only if $x(i-k+1, i) = x(m-k+1, m)$ and $x_{i-k} \neq x_{m-k}$.

3.2. Compute *suffixLength*

From Definition 1, we know that $suffixLength_m = m$. From Corollary 1, we also know that for $1 \leq i \leq m$, if $x_i = x_m$, $suffixLength_i \neq 0$, otherwise, $suffixLength_i = 0$. Now, we discuss how to obtain the entire *suffixLength* when symbol x_m appears also at s other places, $0 \leq s < m$, in pattern x . Without loss of generality, we assume that $x_{i_s} = \dots = x_{i_1} = x_{i_0} = x_m$, where $1 \leq i_s < \dots < i_1 < i_0 = m$. The idea is to determine the other non-zero elements of *suffixLength* from right to left, i.e., $suffixLength_{i_1}$, $suffixLength_{i_2}$, \dots , and $suffixLength_{i_s}$, by comparing each symbol of the pattern, starting with x_{m-1} , from right to left only once against a suffix of the pattern. If a symbol in the pattern has been compared with a symbol in a suffix, we call the symbol a compared symbol, otherwise, an uncomparing symbol.

Theorem 1. Given a pattern $x(1, m)$, when $1 \leq s$, if the location of i_k , $1 \leq k \leq s$, is known and $suffixLength_{i_0}$, $suffixLength_{i_1}$, \dots , and $suffixLength_{i_{k-1}}$ have been found, then, $suffixLength_{i_k}$ can be determined in a constant time plus the time to compare zero or some uncomparing symbols each once with a symbol in a suffix of the pattern.

Theorem 2. Given a pattern $x(1, m)$, its *suffixLength* can be determined in $O(m)$ time.

Basing on the discussion above, we give the pseudocode for computing *suffixLength* below.

```

suffixLengthm = m;
for (i ← 1 to m-1)
    suffixLengthi = 0;
i ← m-1; q ← m-1;
while (i > 0)
{ while (i > 0 and xi ≠ xm)
    i ← i-1;
  if (i > 0)
  { if (q = m-1)
      j ← i-1;
    while (j > 0 and xj = xq)
      { j ← j-1; q ← q-1; }
    suffixLengthi ← i-j; q ← m-1; p ← i-1;
    while (p > j)
    { while (p > j and xp ≠ xm)
        p ← p-1;
      if (p > j)
      { t ← suffixLengthm-(i-p);
        if (p-j > t)
          suffixLengthp ← t;

```

```

    else if (p-j < t)
      suffixLengthp ← p-j;
    else if (j > 0 and xj = xm-(p-j))
      { q ← m-(p-j)-1; i ← p; j ← j-1; p ← j; }
    else suffixLengthp ← t;
    p ← p-1;
  }
}
if (q = m-1) i ← j;
}
}

```

4. Compute *shift* from *suffixLength*

In this section, we will, first, take a closer look at consideration with the match heuristic of a pattern x of the Boyer-Moore algorithm described in section 2. Then, we will discuss the relationship between *shift* and *suffixLength*, and thus derive an algorithm to compute *shift* from *suffixLength* accordingly.

4.1. Analysis of match heuristic

As described in section 2, if a mismatch occurs at x_j , we can shift pattern x right s places.

If $j = m$, there exist two possible cases: a) $\exists x_{m-t} \neq x_m$, $1 \leq t < m$. The right most such x_{m-t} provides the smallest t and also corresponds to the right most zero in *suffixLength* whose location is at i . So, we have $i = m-t$, thus, $t = m-i$. b) $x_{m-t} = x_m$, for $1 \leq t < m$. In this case, where there is no zero element in *suffixLength*, we have $t = m$. Therefore,

$$s_m = \begin{cases} m-i, & \text{if } suffixLength_i \text{ is the right most} \\ & \text{zero element} \\ m, & \text{otherwise} \end{cases}$$

If $1 \leq j \leq m$, there exist three possible cases: a) $\exists 1 \leq t < j$, such that $x(j+1-t, m-t) = x(j+1, m)$ and $x_{j-t} \neq x_j$, i.e., \exists a suffix of length $m-j$ in x . The right most such suffix provides the smallest t and it corresponds to the right most $m-j$ in *suffixLength* whose location is at i . So, we have $t = m-i$. b) $\exists j \leq t < m$ such that $x(1, m-t) = x(t+1, m)$. i.e., \exists a suffix starting at x_1 , whose length is $m-t$, which is no bigger than $m-j$. Note that, in this case, the ending symbol of the suffix x_{m-t} corresponds to the VEI element $m-t$ in *suffixLength*. Since the right most such x_{m-t} provides the smallest t , if the right most VEI element, which is no bigger than $m-j$, is at the location i , then we have $t = m-i$. c) \exists no suffix whose length is less than m . In this case, we have $t = m$. Observe that, the value of t is increasing from case a) through case c). Thus, we have

$$s_j = \begin{cases} m-i, & \text{if } suffixLength_i \text{ is the right most } m-j; \\ & \text{otherwise} \\ m-i, & \text{if } i \text{ is the right most VEI element that} \\ & \text{is less than } m-j; \text{ otherwise} \\ m & \end{cases}$$

4.2. Relation between *shift* and *suffixLength*

Recall that, $shift_j = m-j+s_j$. Thus, combining all cases, we have

$$shift_j = \begin{cases} m-j+(m-i)=m-i, & \text{if } suffixLength_i \text{ is the} \\ & \text{right most zero element} \\ m-j+m=m, & \text{otherwise} \end{cases}$$

for $j=m$, and

$$shift_j = \begin{cases} m-j+(m-i)=2m-j-i, & \text{if } suffixLength_i \text{ is} \\ & \text{the right most } m-j; \text{ otherwise} \\ m-j+(m-i)=2m-j-i, & \text{if } i \text{ is the right} \\ & \text{most VEI element that is less} \\ & \text{than } m-j; \text{ otherwise} \\ m-j+m=2m-j. & \end{cases}$$

for $1 \leq j \leq m$.

Thus, we have the algorithm to compute the shift table from array *suffixLength* as shown below:

```

i=m-1;
while (i>0 && suffixLen[i] != 0) i ← i-1;
shift[m]=m-i;
for (j ← 1 to m-1) shift[j] ← 2*m-j;

prevVEI ← m;
for (i ← m-1 downto 1)
  if (suffixLengthi = i)
    { for (j ← m-prevVEI+1 to m-i)
      shift[j] ← shift[j]-i;
      prevVEI ← i;
    }

for (i ← 1 to m-1)
  if (suffixLengthi ≠ 0 and suffixLengthi ≠ i)
    shift[m-suffixLengthi] ← m+suffixLengthi-i;

```

The algorithm is composed of three parts. The first part sets $shift_m$ and initializes other elements of *shift* with $2 * m - j$. The second part looks for the location of the right most VEI element, if there is any, in *suffixLength* for each j , $1 \leq j < m$. The third part looks for the location of the right most $m-j$, if there is any, in *suffixLength* for each j , $1 \leq j < m$. Clearly, the **while** and

for loops in the first and third parts are in $O(m)$ time. The nested **for** loops in the second part are also in $O(m)$ time, because when the outer loop scans through the $m-1$ elements of *suffixLength*, the inner loop will update at most $m-1$ elements of *shift* once each. Thus, *shift* can be obtained from array *suffixLength* in $O(m)$ time. Therefore, combining the time for computing *suffixLength*, for any given pattern $x(1, m)$, we can obtain the shift table in $O(m)$ time.

5. Conclusion

In this paper, we introduced a new auxiliary array *suffixLength* to compute the shift table used in the Boyer-Moore algorithm. We discussed some theoretical properties of array *suffixLength* and its relationships with the shift table. Consequently, we described an algorithm to compute array *suffixLength* and ultimately to obtain the shift table for a given pattern string $x(1, m)$. Our analysis of the algorithms showed that we can obtain the shift table from a given pattern in linear time. We believe that the new algorithm to obtain *shift* through array *suffixLength* is conceptually more straightforward than the existing one. Therefore, it would alleviate the difficulty in understanding the Boyer-Moore algorithm, so more programmers in a wider spectrum of applications could be more encouraged to utilize this most ingenious string matching algorithm.

References

- [1] G. A. Stephen, *String Searching Algorithms*, World Scientific Publishing Co., Singapore, 1994.
- [2] D. E. Knuth, J. H. Morris Jr., V. R. Pratt, Fast Patten Matching in Strings, *SIAM Journal on Computing*, 77(6):323-350, 1977.
- [3] R. S. Boyer, J. S. Moore, A Fast String Searching Algorithm, *Communications of the ACM*, 77(20):762-772, 1977.
- [4] A. Apostolico, R. Giancarlo, The Boyer-Moore-Galil String Searching Strategies Revisited, *SIAM Journal on Computing*, 86(15):98-105, 1986.
- [5] R. N. Horspool, Practical Fast Searching in Strings, *Software – Practice and Experience*, 80(10):501-506, 1980.
- [6] F. Cao. PAMA: A Fast String Matching Algorithm and Its Application in DNA Sequence Search, *Master Thesis*, Wayne State University, 2004.
- [7] R. Sedgewick, *Algorithms*, Addison Wesley, Reading, MA, 1983.
- [8] A. Hume, D. Sunday, Fast String Searching, *Software – Practice and Experience*, 91(21):1221-1248, 1991