Variation of Boyer-Moore String Searching Algorithm

REPORTED BY:

Yuqian Zhang (CSCI-700)

Hui Shao (CSCI-323)

Semester: Spring 2016

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. For example, the pattern “an occurrence” is a substring of the first sentence of this paragraph. There is always a way that we can solve the problem by hiring brute force of all alignments, which is also called naïve method.

Instead, the Boyer-Moore string searching algorithm preprocesses the string being searched for (the pattern), but not the string being searched in (the text). More specifically, this algorithm searches for occurrences of Pattern in Text by performing explicit character comparisons at different alignments. Boyer-Moore uses information gained by preprocessing Pattern to skip as many alignments as possible.

The Boyer-Moore string searching algorithm is commonly used in the keyword search facility of text-editing programs or word processors. Since then, it has arisen frequently in various kinds of applications, such as DNA sequence matching, data compression, information security, image processing, computer forensics, and anti-plagiarism software.

Thus, this project will focus on some variations of Boyer-Moore string searching algorithm. There are BMH algorithm, BMHS algorithm, BMHS2 algorithm and BMI algorithm.

ALGORITHM OVERVIEW

- **Boyer-Moore**

In 1977, Robert S. Boyer and J Strother Moore developed the Boyer-Moore string search algorithm. The Boyer–Moore string search algorithm is an efficient string searching algorithm that is the standard benchmark for practical string search literature. The BM algorithm scans the characters of the pattern from right to left beginning with the rightmost one and performs the comparisons from right to left. In case of a mismatch (or a complete match of the whole pattern) it uses two pre-computed functions to shift the window to the right. These two shift functions are called the good-suffix shift (also called matching shift and the bad-character shift (also called the occurrence shift).

Assume that a mismatch occurs between the character P[i] =b of the pattern and the character T [i+j] =a of the text during an attempt at position j. Then, P[i+1 .. m-1]=T[i+j+1 .. j+m-1]=u and P[i]≠T[i+j].The good-suffix shift consists in aligning the segment T[i+j+1 .. j+m-1]=P[i+1 .. m-1] with its rightmost occurrence in P that is preceded by a character different from P[i].

BM algorithm will carry through shift computing as follow.

(1) good-suffix function The algorithm looks up string u leader character is not b in P from right to left. If there exist such segment, shift right P to get a new attempt window. If there exists no such segment, the shift consists in aligning the longest suffix v of T[i+j+1 .. j+m- 1] with a matching prefix of P.

(2) bad-char function The bad-character shift consists in aligning the text character T [i+j] with its rightmost occurrence in P [0 ... m- 2]. If T[i+j] does not occur in the pattern P, no occurrence of P in T can include T[i+j], and the left end of the window is aligned with the character immediately after T[i+j], namely T[i+j+1]

Algoritm BMMatch(T, P):
Input: String T (text with n characters and P (pattern) with m characters
Output: Starting index of first substring of T matching p, or an indication that P is not a substring of T

```
i <-- m - 1
j <-- m - 1
repeat
        if P[j] = T[i] then
                if j = 0 then
                        return i {a match!}
        else {check next character}
```

```
            i <-- i - 1
            j <-- j - 1
        else { P[j] <> T[i] move the pattern}
            i <-- i + m - j - 1 {reset i to where it began in most-recent test}
            i <-- i + max(j - last(T[i]), match(j)) {shift P relative to T}
            {note that even if j-last(T[i]) is negative, we will still perform a positive shift,
            because match (j) is always at least 1.}
            j <-- m-1
until i > n - 1

return "There is no substring of T matching P."
```

Refer to java code in Exhibit 1

In preprocessing phase, time and space complexity is O (m+ σ), where σ is the size of the finite character set relevant with pattern and text. In searching phase time complexity is in O (m*n). There are 3n text character comparisons in the worst case when searching for a non periodic pattern. Under best performance time complexity is O (n/m). Under the worst time complexity is O (m*n).

- **Boyer-Moore-Horspool**

In 1980, Horspool simplified BM algorithm and proposed **BMH** algorithm Although it only used the information of the table Right, BMH algorithm acquired no bad efficiency.

The preprocessing of good suffix is hard to be understood and implemented; BMH algorithm only uses the bad characters shift. In BMH algorithm, no matter the location of mismatching, the distance of shift to right is determined by the character in the text string which is aligned to the last one of pattern string.

Like Boyer–Moore, Boyer–Moore–Horspool preprocesses the pattern to produce a table containing, for each symbol in the alphabet, the number of characters that can safely be skipped. The preprocessing phase, in pseudo code, is as follows (for an alphabet of 256 symbols, i.e., bytes):

```
function preprocess(pattern)
    T ← new table of 256 integers
    for i from 0 to 256 exclusive
        T[i] ← length(pattern)
    for i from 0 to length(pattern) - 1 exclusive
        T[pattern[i]] ← length(pattern) - 1 - i
```

```
    return T
```

Pattern search proceeds as follows. [disputed – discuss] The procedure search reports the index of the first occurrence of needle in haystack.

```
function search(needle, haystack)
    T ← preprocess(needle)
    skip ← 0
    while length(haystack) - skip ≥ length(needle)
        i ← length(needle) - 1
        while haystack[skip + i] = needle[i]
            if i = 0
                return skip
            i ← i - 1
        skip ← skip + T[haystack[skip + length(needle) - 1]]
    return not-found
```

In preprocessing phase, time complexity is O(m+ s). In searching phase, time complexity is O(m*n). In the best performance, time complexity is O(n /m). Practical applications show that BMH algorithm is much more efficient than BM algorithm.

- **Boyer-Moore-Horspool-Sunday**

In 1990, Sundays proposed **BMHS** algorithm that improved the BMH algorithm.

The core idea is in the calculation of Bad char function; consider the situation of the next character, namely the use of the next character T[m] to determine the right offset. If the character does not appear in the matching string is skip that step by pattern length + 1; otherwise, the mobile step= match strings in the far right of the character to the end of the range+1.In the matching process, the mode string must not be asked to compare, it does not match is found, the algorithm can skip as many characters to match the next step to improve the matching efficiency.

BMHS algorithm worst case time complexity is O (m*n), the best case time complexity is O (n/m+1). For a short pattern string matching problem, the algorithm is faster.

- **Boyer-Moore-Horspool-Sundays2**

In 2010, Lin quan Xie, Xiao ming liu proposed **BMHS2**, which is strictly based on the analysis of BMHS algorithm to improve is in the match fails, the text string matches last bit characters to participate in the next match, a character string in the case appear to increase the last bit character and appear in the character string matching the first characters of a position if there is consideration.

The idea of algorithm is when mismatch occur at any position then the Right Shift value is determined by Nextto-Last character and Last character of Text corresponding to Pattern that is T[i+m] and T[i+m-1] where m is length of Pattern.

(1) If not in pattern than right shift by m+1.

(2) If occur at first position than right shift by m.

(3) If occur other than first position than shift calculated is X than ß Consider Last character of Text corresponding to pattern and calculate shift, if shift calculated by this is X than shift by X. ß Otherwise shift by m+1.

BMHS2 algorithm worst case time complexity is O(m*n), the best case time complexity is O(n), where n is length of text and the maximum moving distance of m+1.

- **Boyer-Moore-Improvement**

In 2010, **BMI** algorithm is proposed by Jingbo Yuan, Jisen Zheng, Shunli Ding which is improvement of BM algorithm. The BMI algorithm combines with the good-suffix function and the advantages of BMH and BMHS. At the same time the BMI algorithm also takes into account the singleness and combination features of the Next-Character and the Last- Character.

The basic idea behind the algorithm is to achieve the maximum shift distance in the event of a mismatch. Assume that now P [0]…P[m-1] correspond to T[i]…T[i+m-1] during the attempt. If a mismatch occurs, the shift right position will be calculated with function Onechar(x) and TwoChar(x) as following formula (1) and (2).

$$\text{OneChar(X)} = \begin{cases} -1 & \text{Character } x \text{ not in pattern} \\ j & \text{the rightmost position of character } x \text{ in pattern} \end{cases} \quad (1)$$

$$\text{TwoChar(X)} = \begin{cases} -1 & \text{two characters } T[j+m-1]x \text{ not in pattern} \\ j & \text{the rightmost position of characters } T[j+m-1]x \text{ in pattern} \end{cases} \quad (2)$$

Under best performance the time complexity of BM and BMH algorithm all are O(n/m), the time complexity of BMHS and BMI algorithm all are O(n/m+1), but the average time complexity of BMI algorithm is better.

IMPLEMENTATION CONSIDERATIONS

The implementation recursive:

Two recursive models describe such a multiplier in the Boyer-Moore logic :

- the first one characterizes each output of the basic row(s), in terms of the elementary module,
- the second one represents the output(s) of the whole multiplier, in terms of the basic row(s).

In the function definitions Row and Mult below, the parameters X_1, X_2, x_2, P_1, P_2 and p_2 have the following meaning :

- the two bit-vectors given as input of the multiplier are referred to as X_1 and X_2, where X_1 is input horizontally and X_2 vertically (they do not play similar roles); X_1 and X_2 have the same length.
  The variable x_2 denotes the bit n° i of the vector X_2 if the function Row describes the row n° i of the multiplier.
- Scalar or vectorial data are propagated inside the multiplier, either between the cells of the row (they are scalar) or between two successive rows (they are vectorial). In order to illustrate these two aspects, we give two possible configurations :

  Configuration 1 : there are only vectorial data propagated between the rows; we denote these parameters P_1 and P_2.

  Configuration 2 : P_1 is a vector propagated between the rows, and there is a carry propagated between the cells of the row, denoted p_2.

In the following, bit-vectors are considered as lists of booleans : the functions "bit" and "vec" extract respectively the first bit and the rest of the vector; the expression bitv(X,Y) builds a new bit-vector by catenating X as the least significant bit of the vector Y; the constant (btm) is the empty vector.

-1- In the case of Configuration 1, the basic row has two outputs P_1 and P_2, and the associated functions Row_{P1} and Row_{P2} are built on the same pattern Row_{Pi} :

```
Row_Pi (X1, x2, P1, P2) =def
   if bitvp (X1) then          --- if X1 is a bit-vector
      if X1 = (btm)            --- if X1 is empty, the output is empty
      then (btm)
      --- if X1 is not empty, we recursively call Row_Pi :
      else bitv ( F_pi (bit (X1), x2, bit (P1), bit (P2)),
                  Row_Pi ( vec (X1),
                           x2,
                           vec (P1),
                           vec (P2)))
   else (btm)
```

The function Rowp1 (resp. Rowp2) builds the bit-vector the first bit of which is the result of Fp1(resp. Fp2) which is the function associated with the corresponding output of the basic cell, and the rest is computed by the recursive call to Rowp1 (resp. Rowp2) with the vector inputs of one element less.

-2- And if P_1 is a vector propagated between the rows and p_2 is a carry propagated between the cells of the row (Configuration 2), there is only one function for the output P_1 of the row :

$Row_{P1}(X_1, x_2, P_1, p_2) =_{def}$
  if $bitvp(X_1)$ **then** --- if X_1 is a bit-vector
    if $X_1 = (btm)$ --- if X_1 is empty, the result contains only the carry p_2
    then $bitv(p_2, (btm))$
    --- if X_1 is not empty, we recursively call Row_{P1} :
    else $bitv($ $F_{p1}(bit(X_1), x_2, bit(P_1), p_2),$
      $Row_{P1}($ $vec(X_1),$
        $x_2,$
        $vec(P_1),$
        $F_{p2}(bit(X_1), x_2, bit(P_1), p_2)))$
  else $bitv(p_2, (btm))$

The function Rowp1 builds the bit-vector the first bit of which is the result of Fp1 (the function associated with the corresponding output of the basic cell), and the rest is computed by the recursive call to Rowp1 with the vector inputs of one element less and carry P2, which is propagated in the row, updated by the function Fp2.

Choices for data structures:

Basically, it is mostly implement into arrays.

Elements are accessed using an integer index to specify which element is required (although the elements may be of almost any type). Typical implementations allocate contiguous memory words for the elements of arrays (but this is not always a necessity). Arrays may be fixed-length or resizable.

Choices of programming language, hardware/ operating system:

Various implementations exist in different programming languages. In C++, Boost provides the generic Boyer–Moore search implementation under the Algorithm library. In Go (programming language) there is an implementation in search.go. D (programming language) uses a BoyerMooreFinder for predicate based matching within ranges as a part of the Phobos Runtime Library. The Boyer-Moore algorithm is also used in GNU's grep. Thus, generalized used operating system like UNIX, WINs supports it well.

Tracking of operations:

The implementation of the Boyer-Moore algorithm uses a skip loop as proposed in Hume and Sunday 1991 gnores the second shift array as proposed in Horspool 1980.

The shift function is similar to the one proposed in Sunday 1990. The alphabet is fixed to 8bit (256 characters).

Depending on the size of the text and the probable location of the pattern in the text, it may be better to use the naive approach as it is implemented in String.indexOf() to pattern matching. The size of the alphabet and the length of the pattern are further factors that need to be taken into consideration.

DATA CONSIDERATIONS

- Boyer-Moore algorithm is extremely fast on large alphabet (relative to the length of the pattern).
- The payoff is not as for binary strings or for very short patterns.
- For binary strings Knuth-Morris-Pratt algorithm is recommended.
- For the very shortest patterns, the naïve algorithm may be better.

Thus, this algorithm is most suitable for large alphabet data.

TRACKING OF OPERATIONS

In our example:

Text string: “STRINGMATCHINGISTOFINDTHEPATTERN” size = n

Pattern string: “PATTERN” size = m

We have calculated

	BM	BMH	BMHS	BMHS2	BMI
Shift	5	5	4	4	4
Comparison	13	13	13	11	11
Best-case time	O(n/m)	O(n/m)	O(n/m+1)	O(n/m+1)	O(n/m+1)
Worst-case time	O(m*n)	O(m*n)	O(m*n)	O(m*n)	O(m*n)

In our example BM and BMH performance was equal as SHIFT=5 and Comparison=13.

In case of BMHS SHIFT decreases to 4 but Comparison remains to 13, so we can’t say that BMHS always perform better than BMH, it totally depends on Input. In example performance of BMI and BMHS2 is equal but we also can’t say that their performance remains always same, it is also depending on Input.

BMH algorithm is more efficient when last character does not occur in pattern.

BMHS is more effective than BMH when last character occurs in pattern but next to last character does not occur in pattern.

BMHS2 perform better when next to last character does not occur in pattern or occur at first position in pattern.

BMI algorithm perform better when Next to Last character does not occur in pattern; Or when Last character does not occur in pattern; Or when combination of Last character with Next to Last character does not occur in pattern.

GRAPH

TABLE 1.BM Example (5 Shift and 13 Comparisons)

	S	T	R	I	N	G	M	A	T	C	H	I	N	G	I	S	T	O	F	I	N	D	T	H	E	P	A	T	T	E	R	N
1	P	A	T	T	E	R	N																									
2								P	A	T	T	E	R	N																		
3															P	A	T	T	E	R	N										
4																						P	A	T	T	E	R	N			
5																									P	A	T	T	E	R	N	
6																										P	A	T	T	E	R	N

TABLE 2.BMH Example (5 Shift and 13 Comparisons)

	S	T	R	I	N	G	M	A	T	C	H	I	N	G	I	S	T	O	F	I	N	D	T	H	E	P	A	T	T	E	R	N
1	P	A	T	T	E	R	N																									
2								P	A	T	T	E	R	N																		
3															P	A	T	T	E	R	N										
4																						P	A	T	T	E	R	N			
5																									P	A	T	T	E	R	N	
6																										P	A	T	T	E	R	N

TABLE 3.BMHS Example (4 Shift and 13 Comparisons)

	S	T	R	I	N	G	M	A	T	C	H	I	N	G	I	S	T	O	F	I	N	D	T	H	E	P	A	T	T	E	R	N
1	P	A	T	T	E	R	N																									
2							P	A	T	T	E	R	N																			
3															P	A	T	T	E	R	N										
4																							P	A	T	T	E	R	N			
5																										P	A	T	T	E	R	N

TABLE 4.BMHS2 Example (4 Shift and 11 Comparisons)

	S	T	R	I	N	G	M	A	T	C	H	I	N	G	I	S	T	O	F	I	N	D	T	H	E	P	A	T	T	E	R	N
1	P	A	T	T	E	R	N																									
2									P	A	T	T	E	R	N																	
3																	P	A	T	T	E	R	N									
4																									P	A	T	T	E	R	N	
5																										P	A	T	T	E	R	N

TABLE 6.BMI Example (4 Shift and 11 Comparisons)

	S	T	R	I	N	G	M	A	T	C	H	I	N	G	I	S	T	O	F	I	N	D	T	H	E	P	A	T	T	E	R	N
1	P	A	T	T	E	R	N																									
2								P	A	T	T	E	R	N																		
3																P	A	T	T	E	R	N										
4																							P	A	T	T	E	R	N			
5																										P	A	T	T	E	R	N

CONCLUSION

The comparison of BM and its relative algorithm is performed on the basis two factors; one is number of comparison performed and second is search time.

In example we present a comparison on the basis of number of comparison performed that performance of BM, BMH and BMHS are almost equal as number of comparison is almost same

We also take a look at a comparison on the basis of search time in which BM and BMH perform almost same but BMHS search time increases. In BMI searching is faster than above four and BMHS2 search time is even less than BMI. So finally we can say that BMHS2 is best of all variation algorithms as search time and number of comparison both are less than in all other algorithm.

To sum up, composite Boyer-Moore algorithm is efficient in case of binary searching where small varieties of alphabet and long pattern.

The performance of algorithm depends on two factors, first on Input, number of inputs and type of inputs, second is methodology of algorithm, so there may be possible that some variation in performance occur as input changes.

FUTURE STUDY

The focus of future work is to improve existing algorithm and finding the efficient string searching algorithm so that searching speed can be increased and performance as well.

SOURCES

1. https://en.wikipedia.org/wiki/Boyer%E2%80%93Moore_string_search_algorithm
2. http://www8.cs.umu.se/kurser/TDBA59/VT01/mom3/slides/BM-alg.html
3. http://www-igm.univ-mlv.fr/~lecroq/string/node14.html
4. http://bioinformatics.iyte.edu.tr/supplements/peerj/javadoc/boyermoore/BoyerMooreHorspoolUFast.html
5. http://www.mathcs.emory.edu/~cheung/Courses/323/Syllabus/Text/Docs/Boyer-Moore-variants.pdf
6. Hume; Sunday (November 1991). "Fast String Searching". Software—Practice and Experience 21 (11): 1221–1248.

EXHIBITS

Exhibit 1

```
/**
     * Returns the index within this string of the first occurrence of the
     * specified substring. If it is not a substring, return -1.
     *
     * @param haystack The string to be scanned
     * @param needle The target string to search
     * @return The start index of the substring
     */
    public static int indexOf(char[] haystack, char[] needle) {
        if (needle.length == 0) {
            return 0;
        }
        int charTable[] = makeCharTable(needle);
        int offsetTable[] = makeOffsetTable(needle);
        for (int i = needle.length - 1, j; i < haystack.length;) {
            for (j = needle.length - 1; needle[j] == haystack[i]; --i, --j) {
                if (j == 0) {
                    return i;
                }
            }
            // i += needle.length - j; // For naive method
            i += Math.max(offsetTable[needle.length - 1 - j],
charTable[haystack[i]]);
        }
        return -1;
    }

    /**
     * Makes the jump table based on the mismatched character information.
     */
    private static int[] makeCharTable(char[] needle) {
        final int ALPHABET_SIZE = 256;
        int[] table = new int[ALPHABET_SIZE];
        for (int i = 0; i < table.length; ++i) {
            table[i] = needle.length;
        }
```

```
        for (int i = 0; i < needle.length - 1; ++i) {
            table[needle[i]] = needle.length - 1 - i;
        }
        return table;
    }

    /**
     * Makes the jump table based on the scan offset which mismatch occurs.
     */
    private static int[] makeOffsetTable(char[] needle) {
        int[] table = new int[needle.length];
        int lastPrefixPosition = needle.length;
        for (int i = needle.length - 1; i >= 0; --i) {
            if (isPrefix(needle, i + 1)) {
                lastPrefixPosition = i + 1;
            }
            table[needle.length - 1 - i] = lastPrefixPosition - i +
needle.length - 1;
        }
        for (int i = 0; i < needle.length - 1; ++i) {
            int slen = suffixLength(needle, i);
            table[slen] = needle.length - 1 - i + slen;
        }
        return table;
    }

    /**
     * Is needle[p:end] a prefix of needle?
     */
    private static boolean isPrefix(char[] needle, int p) {
        for (int i = p, j = 0; i < needle.length; ++i, ++j) {
            if (needle[i] != needle[j]) {
                return false;
            }
        }
        return true;
    }

    /**
     * Returns the maximum length of the substring ends at p and is a suffix.
```

```
     */
    private static int suffixLength(char[] needle, int p) {
        int len = 0;
        for (int i = p, j = needle.length - 1;
                 i >= 0 && needle[i] == needle[j]; --i, --j) {
            len += 1;
        }
        return len;
    }
```