A New String Matching Algorithm Based on Logical Indexing

Daniar Heri Kurniawan  
Department of Informatics  
School of Electrical Engineering and Informatics  
Institute Technology, Bandung  
Bandung, Indonesia  
daniar.h.k@gmail.com

Ir. Rinaldi Munir, M.T.  
Department of Informatics  
School of Electrical Engineering and Informatics  
Institute Technology, Bandung  
Bandung, Indonesia  
rinaldi@informatika.org

Abstract—Searching process performance is very important in this modern world that consists of various advanced technology. String matching algorithm is one of the most commonly applied algorithms for searching. String that contains sequences of character is the simplest representation of any complex data in our real life, such as search engine database, finger print minutiae, or DNA sequence. The current most famous string algorithms are Knuth-Morris-Pratt (KMP) algorithm and Boyer-Moore (BM) algorithm. Boyer-Moore algorithm and KMP algorithm are not efficient in some cases. We introduced a new variation of string matching algorithm based on Logical-Indexing (LI), which is more efficient than those algorithms. Logical-Indexing algorithm implements a new function or method and defines different jumping rules compared to others. The indexes of texts and patterns are used to reduce the number of comparisons. Index enables us to analyze the condition of comparison without directly comparing each character in the pattern. Theoretically, we have proven that Logical-Indexing algorithm can skip more characters than KMP and BM algorithm by analyzing KMP’s and BM’s weaknesses then giving the better solution that are implemented in LI. Furthermore, experiments conducted on various combinations of pattern matching cases also demonstrate that the average number of LI’s direct comparisons is smaller than KMP’s and BM’s algorithm.

Keywords—LI algorithm; full-jumping; margin-jumping; matched-jumping; paired-character; occurrenceTable;

I. INTRODUCTION

Searching algorithm is routinely applied in various computer applications, such as finding part of DNA in bioinformatics engineering. Consequently, searching performance is an important factor in computer science. There are two main types of string matching algorithm. If we refer to the starting index of comparison, the first is Knuth-Morris-Pratt or KMP algorithm, and the second is Boyer-Moore or BM algorithm. Both of them have their own strengths and weaknesses. Besides, we also study and analyze another algorithm such as Sheik-Sumit-Anindya-Balakrishnan-Sekar algorithm (SSABS) and Fast Algorithm for Approximate String Matching (FAAST).

In this paper, we present a new method to process any searching data regardless of its size. We call this algorithm Logical-Indexing (LI) algorithm. Logical-Indexing algorithm is not only applicable for small alphabet size, but also larger scale. This algorithm has four core functions or methods: createCharTable(), createOccurrenceTable(), createMargin(), and findPairedChar(). Measuring LI’s performance, we only consider BM algorithm and KMP algorithm to be analyzed and compared with LI because the other variant basically have been represented by those algorithms.

The organization of this paper is as follows: in the next section, we give review of existing algorithms. Then in section 3, we focus on defining and describing LI algorithm with terms definition, informal explanation, java language implementation, processing stages, and complexities analysis. In section 4, we have empirical evidence of our algorithm based on the testing results compared to BM and KMP algorithm. The result and analysis are written in section 5. Finally, the conclusions and future work are presented in section 6.

II. REVIEW OF EXISTING ALGORITHMS

A. Knuth-Morris-Pratt (KMP) Algorithm

It is a common sense to try searching at every starting position of the text, abandoning the search as soon as an incorrect character is found [1]. Knuth-Morris-Pratt algorithm is not efficient because the length of pattern’s shift is depending on how long the matched-string. In the other words, if we have already compared many characters and then we find a mismatch, at that point we may have a long shift but we need to restart comparing again after long comparison that we did. Therefore, the purpose to minimize comparison by have a long shift does not work efficiently in this algorithm.

B. Boyer-Moore (BM) Algorithm

Boyer-Moore algorithm is one of the most efficient algorithms compared to the other algorithms available in the literature. The basic idea behind the algorithm is to gain more information by matching the pattern from the right than from the left [2]. Boyer-Moore algorithm uses two main processes, which are looking glass technique and character jump technique. Looking glass technique is the way to find pattern in the text by matching the pattern from the right side. Character jump technique is an index shifting technique based on some conditions. Character jump technique is based on shift caused by delta1 or delta2.

This algorithm had been extended since the first version. The most recent version uses two precomputed functions to shift the window to the right. These two shift functions are called the good-suffix shift and the bad-character shift. However, BM algorithm is less effective in using last
occurrence index than LI because we use it twice in every mismatch, instead of once. Thus, LI algorithm has bigger possibility to do a longer shift because the probability of finding two character sequentially (paired-character) is less than finding one character. The detail explanation about paired-character will be given in the next section.

C. Sheik-Sumit-Anindya-Balakrishnan-Sekar (SSABS) Algorithm

The SSABS algorithm carries the order of comparisons out by comparing the last character of the window and the pattern, and after a match, the algorithm further compares the first character of the window and the pattern. By doing so, an initial resemblance can be established between the pattern and the window, and the remaining characters are compared from right to left until a complete match or a mismatch occurs [3]. This algorithm is not implementing any additional rule or logic because it only changes the order of BM’s comparison. Therefore, it is also less effective in using last occurrence index than LI because we use it twice in every mismatch, instead of once. The weakness is similar to BM algorithm.

D. Fast Algorithm for Approximate String Matching (FAAST)

The other one, FAAST algorithm theoretically improved BM algorithm in some cases, especially on small alphabet size. Brief explanation of their algorithm is that the algorithm requires at least x matches in the last k + x characters when calculating shift distances, where x is a small integer value (typically 2 or 3 in their experiments). However, as the alphabet size and the x value get large, they notice that the time and memory required for the shift distance calculation increase quickly, which in turn deteriorates the performance of FAAST [4].

III. THE PROPOSED ALGORITHM

A. Terms Definition

There are some terms used to explain this algorithm. Assume S is a string of size M and P is a string of size N.

- **Text** is the place where we want to find a pattern inside of it. In this case, text is the value of S.
- **Pattern** is sequence of character, which has length shorter than text. Pattern is value of P. Pattern’s head is y0 and pattern’s back is yn.
- **Prefix of S** is a substring S[1 ... M-1].
- **Suffix of S** is a substring S[M-1 ... M].
- **Index** is a position representation, which starts from zero until string’s length minus one. There are three kinds of index: 1) Pattern’s index is an index that relative to the pattern; 2) Text’s index is an index that relative to the text; and 3) Comparison’s index is an index of current comparison between text and pattern. This index is derived from text’s index since the index of pattern will move along relative to text’s index.

Assume S is a text, P is a pattern, and n is comparison number. After fourth comparison, the mismatch occurs and pattern’s index will be divided as a following part’s color that will be described below:

\[
S = \text{“bacxybaababaxbacacxaba“} \\
P = \text{“bacxaba“} \\
n = 321
\]

- Matched-string (\text{“ba“}) is the substring that is already matched before mismatch occurs;
- Unmatched-string (\text{“bacxa“}) is the substring that has not been compared when the mismatch occurs;
- Paired-character (\text{“xy“}) is the text’s substring that consists of two characters or one character if the mismatch occurs in the text’s first index. Assume that we have mismatch’s index relative to the text’s index, so:
  - the first character = \text{S[mismatch’s index -1]} and
  - the second character = \text{S[mismatch’s index]};
- **Margin-string** is the longest string between suffix of matched-string and prefix of pattern. In this case, margin-string = “ba”. The front-margin is “ba” (purple) and the back-margin is “ba” (blue);
- One phase is defined when a pattern is not moved or shifted while comparison is occurring. When the pattern is shifted, we continue to the next phase of comparing.
- In-pair is a condition when paired-character can be found in unmatched-string.

B. Informal Description

This algorithm was called Logical-Indexing string matching algorithm because we optimize the use of index by adding some logical processing to minimize the number of direct comparison. Logical-Indexing algorithm is improving Boyer-Moore algorithm in reducing the number of comparison by applying new rules. We use two text’s characters to be analyzed instead of one when mismatch occurs. The purpose is to get a bigger pattern shift to skip any useless character. Moreover, we use different preprocessed or precomputed functions combination to improve BM’s precomputed functions.

The matching process started by placing the first index of pattern aligns with the first index of the text. This algorithm will start the comparison from the last character of the pattern. It will be compared to text’s character, which has the same index. If the comparison is true, the algorithm will decrease comparison’s index by one. At this point, algorithm starts to compare characters between pattern and text. If the result is false or a mismatch occurs, there are three possibilities, which are matched-jumping, margin-jumping, and full-jumping. This algorithm has some characteristics that will be described as follows:

1) **Smart Finding**: When the mismatch occurs, we will have paired-character to be examined whether it is in-pair or not. There are two possibilities as the results of this process which are matched-jumping and margin jumping. If the paired-character we are looking for is on the unmatched-string, so it will continue to do matched-jumping. In another
hand, it will be processed according to margin-jumping
condition. The idea of smart finding is to find paired-
character in the unmatched-string without direct comparison
since we have precomputed value that can be used.

2) Matched Jumping: When the paired-character is in-
pair (can be found in unmatched-string), so the pattern will
move until paired-character’s second index is aligned with
the index of mismatch. In the special case, if paired-
character is not in-pair but paired-character’s second
character is the same with pattern’s first character, we will
move the pattern until its first index reach mismatch’s index
(mismatch’s index is relative to text’s index). We call the
last condition as single-match

3) Margin Jumping: When the conditions of matched-
jumping cannot be fulfilled, the pattern will be moved or
dragged until its front-margin align with the previous
position of back-margin.

4) Full Jumping: When the condition of margin-jumping
cannot be applied, the pattern will be moved until its first
index placed next to its previous last index position. In
another words, the pattern will be moved to the right as long
as it’s length.

5) Avoid Double Comparison: The purpose is when the
algorithm is already compared some characters and all of
them are matched, it should not compare them twice in the
next phase. However, this feature only available in two
conditions which are matched-jumping and margin-jumping.
We will save mismatch’s index when one of the conditions
occurs. The first, when margin-jumping occurs, this can skip
M-1 character at the best case. The second, when matched-
jumping occurs, this can skip 2 character at normal case or
one character at single-match condition. This will be very
useful when we applied LI in any complex system because
the cost of direct comparison could be really big.

C. Preprocessing Stage

The first step before doing any comparison is building
some databases that will be useful in the matching stage. The
variables that are important to be noticed are marginTable,
occurrenceTable, and charTable. The first variable is
marginTable. This variable contains the length of margin-
string in every pattern’s index. The type of this variable is
vector or array of integer. Supposed that we have a pattern
P = “bacxaba”, table 1 depicts pattern’s index that will be used
and table 2 is pattern’s marginTable.

TABLE I. PATTERN

<table>
<thead>
<tr>
<th>j</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>P[j]</td>
<td>b</td>
<td>a</td>
<td>c</td>
<td>x</td>
<td>a</td>
<td>b</td>
<td>a</td>
</tr>
</tbody>
</table>

TABLE II. MARGIN TABLE

<table>
<thead>
<tr>
<th>j</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>marginTable[j]</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

We use createMargin() function to calculate
marginTable values. The technique is by calculating the
longest suffix of matched-string that is the same with prefix
of pattern. The index j is the position where the mismatch
occurs. Therefore, matched-string should be “ba” if we get
mismatch at fourth index after third comparison. At that
point, we have marginTable[4] = 2 which means that there
is a substring or suffix of matched-string with length of 2 that
is also a prefix of the pattern. It turns out it is called as
margin because the illustration at figure 3 shows that string
“ba” can be reached from the left as well as from the right

The second variable is occurrenceTable, which means
the table that saves previous occurrence of the pattern’s
characters relative from its index. For example, we use
pattern P = “bacxaba” that is already defined in table 1, the
following table is its occurrenceTable:

TABLE III. OCCURRENCE TABLE

<table>
<thead>
<tr>
<th>j</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>occurrenceTable[j]</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>1</td>
<td>0</td>
<td>4</td>
</tr>
</tbody>
</table>

The values of table 3 is computed using
createOccurrenceTable() function by iterating pattern’s
index from the first index. According to that table, the value of
occurrenceTable[6] is 4 and the value of occurrenceTable[4] is 1. We already knew that P[6] is the
same with P[4] which is equal with ‘a’. Therefore,
occurrenceTable[6] shows the previous index of character
’a’ before it appears at sixth index. Furthermore, some
characters do not have previous existence in the earlier index.
We marked that occurrence with value of -1, such as at j = 0.

There are many ways to implement this function. In my
implementation, we use container (list) to save the position
of each character and then will be updated with the newest
position. At first, the container is empty (initialized by -1) but
when the iteration is started, it will save the index of current
character in the current position. The current value will be
retrieved before updated by the same character. That looping
keeps going until the rest of the pattern iterated.

The occurrenceTable is very useful in almost every case
of matching using LI algorithm. We can try to find paired-
character until satisfy in-pair condition through accessing this
variable recursively. However, at the first time before
retrieving any data from occurrenceTable, we need to know
where the text’s character position in the pattern that caused
mismatch is. The last position of every character (we use 225
different characters) will be mapped by its last occurrence
and stored in variable charTable.

The third variable is charTable which is array of integer.
This variable will save the last occurrence of every alphabet
in the pattern. If there are some alphabets that are never exist
in the pattern, then the value of its charTable is -1. For
example if we want to find last occurrence of ‘a’ inside of the
pattern P, then charTable[‘a’] = 6 or if we try another
alphabet like ‘y’, so the value of charTable[‘y’] is -1.

The purpose of charTable is to give starting index for
finding paired-character in the unmatched-string. Suppose
that we have a mismatch at T_i, and then we will get its last
occurrence from charTable[T_i]. After that, we start to
compare it with the index of mismatch (assumed j), if

Fig. 1. Illustration of margin “ba”.
charTable[T[i]] > j, we will start to check the value of occurrenceTable[charTable[T[i]]] or even the value of occurrenceTable[occurrenceTable[...]] recursively until satisfy the proper conditions for starting. The proper condition will be satisfied when occurrenceTable[...] < j. However, if the result is -1, it will stop doing recursive and start analyzing for margin-jumping and full-jumping conditions.

D. Matching Stage

Suppose that we have the same pattern P = “bacxaba” that defined in table 1. We want to find pattern P inside of Text T = “bacxybaababaxbaacaabacxaba”. This matching process requires values from marginTable[] and occurrenceTable[] that have been described in the previous section. The value of marginTable[] is given at table 3 (Margin Table) and the value of occurrenceTable[] is given at Table 4 (Occurrence Table). Figure 2 - 7 consist of three line, including text, pattern, and the order of comparison. The third line also shows the number of comparison at the current position. However, we only give its last digit to make sure it is correctly-align with the character above of it.

**TABLE IV. TEXT**

<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>T[i]</td>
<td>b</td>
<td>a</td>
<td>c</td>
<td>x</td>
<td>y</td>
<td>b</td>
<td>a</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>a</td>
</tr>
</tbody>
</table>

At the first condition, the position of text and pattern will be left aligned as follows

1. bacxybaababaxbaacaabacxaba bacxaba
   321
   Fig. 2. Matching Process (First phase).

At the first phase, the character on both text and pattern will be compared starting from right most patterns’ index. Starting from sixth index, we decrease the index of comparison until we get a mismatch at the fourth index. Then we know that margin-string = “ba”, paired-character = “xy”, and unmatched-string = “bacx”. This algorithm will try to find that paired-character in the unmatched-string. In this case, it is failed. Therefore, the pattern is shifted until its front-margin aligns with back-margin or with a distance == Pattern.length-marginTable[4] = 5. After shifted 5 indexes to the right, it will appear as shown in figure 3.

2. bacxybaababaxbaacaabacxaba bacxaba
   7654
   Fig. 3. Matching Process (Second phase).

The second phase will start from the right most pattern’s index as always. Noted that the comparison’s index is relative to text’s index, so we said that the comparison in this phase starts at 11th index until we get mismatch at the 8th index. Just like before, LI algorithm will try to find paired-character, which is “ab”, and it is failed. However, it still has one chance to make matched-jumping. The alterative condition for matched-jumping is when T[mismatch’s index] == P[0]. Therefore, the pattern will be shifted until its first index reach mismatch’s index.

3. bacxybaabababaxbaacaabacxaba bacxaba
   8
   Fig. 4. Matching Process (Third phase).

In this third phase, mismatch occurs at the first comparison. In this condition, we know that margin-string is empty or zero, paired-character is “ax”, and unmatched-string is “bacxaba”. By using findPairedChar(), LI algorithm will know that the paired-character is not in-pair. Therefore, the pattern will be dragged according to full-jumping rule.

4. bacxybaabababaxbaacaabacxaba bacxaba
   9
   Fig. 5. Matching Process (Fourth phase).

In this fourth phase, mismatch occurs at the first comparison again. In this condition, we know that margin-string is empty or zero, paired-character is “ab”, and unmatched-string is “bacxaba”. By analyzing values from occurrenceTable and charTable, LI algorithm will know that the position of paired-character are at index 4 and index 5. Then we got one after calculate its jumping distance. It means that the pattern will be shifted one index to the right

5. bacxybaabababaxbaacaabacxaba bacxaba
   1 0
   Fig. 6. Matching Process (Fifth phase).

This fifth phase shows about skipping or avoiding double comparison after matched-jumping. After comparing from 22nd index, we find a mismatch at 19th index. However, we only need two comparisons because we already marked the 21st and 20th index in the previous phase. In this case, LI algorithm will try to find paired-character (“ca”) in the unmatched-string but it is failed because (“ca”) is not in-pair. Then we move the pattern until its front-margin aligned with back-margin position since marginTable[3] == 2.

6. bacxybaabababaxbaacaabacxaba bacxaba
   65432
   Fig. 7. Matching Process (Sixth phase).

In this last phase LI algorithm finish the comparison at index 23th with total 16 times comparison. In the previous phase, we save the index of mismatch to avoid double comparison. We are not comparing the 21st - 22nd index because it should be the same (match) according to the previous shifting.

E. Algorithm’s Pseudo Code

This following pseudo code is a general description of how this algorithm works, especially on matching stage. We assume that pattern’s index and text’s index always start from zero. According to common programming languages, such as Java and c++ language, string is an array of char that its length represent the number of char in the string. In this case, we save the pattern and the text inside of string
variable, so it can be treated as an array of char. For instance, if we want to access the $n^{th}$ character of a string $S$, we will write $S[n-1]$, because the index starts from zero.

The result after exiting while-looping does not show the detail comparison result, such as index of first-match and number of comparison, because in this paper we only give the main idea of LI algorithm. However, we already implemented this pseudo code’s idea successfully in Java language and finished the benchmarking that is given as empirical evidence.

```java
/* variable text and pattern are array of char */
/* i is text's iterator */
/* j is pattern's iterator */
i = PatternLength - 1;
j = PatternLength - 1;

while ( j >= 0 && i < textLength )

    /* start direct comparison */
    if (text[i] == pattern[j])
        i--; j--;
    else /* mismatch occurs */
        /* pairedChar = getPairedChar(text, i); */
        if ( isInPair ( pairedChar ) )
            /* condition 1 */
            markId = i;
            shift = matchedJumping;
            if ( singleMatch ( pairedChar ) )
                numOfMatch = 1;
            else
                numOfMatch = 2;
        elseif ( marginLength != 0 )
            /* condition 2 */
            markId = i;
            shift = marginJumping;
            numOfMatch = marginLength;
        else
            /* condition 3 */
            shift = fullJumping;
        endif
    endif

    /* avoid double comparison */
    if ( i == markId )
        i = i - numOfMatch;
        j = j - numOfMatch;
    endif
endwhile
```

### F. Analysis of the Proposed Algorithm

Assume that we have text of $n$ size and pattern of $m$ size. The time complexity of LI algorithm will be described at lemma 1 and lemma 2 below.

**Lemma 1.** The time complexity is $O(n)$ in the best case.

**Proof.** Every character in the text will be compared once, so the minimum number of comparison is $n$ times (see figure 8 below). Furthermore, the number of phase is one in every best case.

**Lemma 2.** The time complexity is $O((n - m + 1)(m - 1))$ in the worst case.

**Proof.** That complexity consists of two parts, calculating the maximum number of phase and calculating the maximum number of comparison in the pattern. In the worst case, LI algorithm will let the pattern shifted to the right by one index if there is a mismatch at the second last of pattern’s character, so the maximum comparison for each phase is $(m - 1)$. The comparison number is also influenced by the phase. The maximum number of phase that possible to occur is $(n - m + 1)$. Therefore, if every phase has maximum number of pattern’s comparison, the total number of comparison will follow $[(n - m + 1)(m - 1)]$. Figure 9 will depict that condition.

**IV. Empirical Evidence**

We implement BM algorithm and KMP algorithm using java language in order to compare the number of direct comparison in various test cases. Boyer-Moore algorithm that we use is an updated version according to my literature [2]. Knuth-Morris-Pratt is also implemented as described in the first literature [1]. We run the program that will generate 256 types of characters randomly for the pattern and the text. We have validated the implementation of KMP and BM by comparing the results and the rules to their paper. The charts in the figure 10 and 11 will give the number of direct comparisons based on various text’s size and pattern’s size. Actually, we can use larger data’s size (more than 1,000,000 char length) to visualize the comparison but the performance of KMP will less than BM and LI. That condition will make us difficult to identify BM’s and LI’s chart. Therefore, we use string with 100,000 char length as the biggest data.
V. RESULT AND ANALYSIS

Theoretically, we proved that LI’s rules are better than BM and KMP because we have marginTable and occurrenceTable that make LI skips more character than BM and KMP algorithm. Technically, we found that Logical-Indexing much more efficient in reducing number of comparison based on the charts of experiments result in the previous section. We have analyzed two types of data, which have various combinations of text and pattern. In the figure 10, it shows the number of comparison increasing according to all algorithms, but LI has the smallest increment. Moreover, in the figure 11 also shows that LI has the smallest number of comparison.

VI. CONCLUSION

In this paper, we proposed a new string-matching algorithm by optimizing the use of its index. The most important variable in this algorithm is occurrenceTable, which is a variable that save the last occurrence of each character. It is used to find paired-character that will maximize the shift for next phase. This new algorithm has been tested, and it is significantly improving Boyer-Moore algorithm and Knuth-Morris-Pratt algorithm. This algorithm is not only editing BM’s lastOccurrence() function, but also we are adding a lot of new logical processing that never used by another algorithm, such as: finding paired-character, defining margin-area, calculating matched-jumping, and calculating margin-jumping.

The test case that has been used is sufficiently large, and at this point, it is enough to conclude that the proposed algorithm is efficient in reducing the number of comparison. Moreover, it can be applied in other contexts of searching, especially in a database that need a long time to compare a single data to another. Our further interest is to calculate the best length of paired-character dynamically regarding to the size of pattern. We also want to implement this algorithm in a real life problem to support high scale computation.

VII. ACKNOWLEDGMENT

The authors gratefully thank to Allah SWT that blessing us every time and to the Department of Informatics ITB that gives us a lot of inspiration to finish this paper, not only its beautiful environment but also its friendly academicians.

REFERENCES