String Matching Algorithms (SMAs): Survey & Empirical analysis

Kamal M. H. Alhendawi
PhD Candidate, School of Computer Sciences
Universiti Sains Malaysia (USM)
Penang, Malaysia

Ahmad Suhaime Baharudin
Senior Lecturer, School of Computer Sciences
Universiti Sains Malaysia (USM)
Penang, Malaysia

Corresponding Author E-mail: Hindawi.kamal@yahoo.com

Abstract—this paper introduces a short survey for well-known string matching algorithms, where these algorithms are commonly used, and provide significant contributions to computer sciences. In practice, our survey paper aims to present the idea, theoretical analysis, and an empirical testing for the execution time of each String Matching Algorithm (SMA) in order to assess the efficiency of these SMAs.

Keywords: String matching algorithms, pattern matching, Naïve, Knuth-Morris-Pratt, Boyer-Moore, Rabin-Karp, Horspool, Efficiency.

I. INTRODUCTION

The string or pattern matching algorithms (SMA), sometimes called string searching algorithms, are an important part of string algorithms class which aims to find the location of a specific text pattern within a larger body of text (e.g., a sentence, a paragraph, a book, etc.). Its application covers a wide range, including in editors, information retrieval, searching nucleotide or amino acid sequence patterns in genome [1].

With respect to this paper, first, it includes a survey for the well-known string matching algorithms in existence today in order to reveal the features, time complexity and Pseudo for a set of common SMAs. Accordingly, the current survey paper covers not only the three fundamental algorithms (Brute Force, Rabin-Karp, and Knuth-Morris-Pratt) but it also discusses another two string matching algorithms called Boyer-Moore and Hors-pool algorithms as many of theoretical comparison studies such as [2,3] includes these algorithms.

Second, as with most algorithms, the main considerations for string matching are speed and efficiency. Our paper performs an empirical testing for the efficiency of SMAs based on their execution time. The worth mentioning is that the efficiency is actually measured in terms of two factors which are the length of both text and pattern (n, m, respectively).

To sum up, the current survey paper includes a theoretical analysis for five of common SMAs, empirical testing of the execution time based on the change of two factors (text size and pattern size), then it measures the efficiency of each SMA in term of estimated execution time.

II. PRELIMINARIES

In practice, the following concepts including notations, variables and definitions are essential for understanding the string searching algorithms where these concepts are included within [4,5]:

A. Pattern and Text Variables

Let P be a string of size m

- A substring $P[i .. j]$ of P is the subsequence of P consisting of the characters with ranks between $i$ and $j$.
- A prefix of P is a substring of the type $P[0 .. i]$.
- A suffix of P is a substring of the type $P[i .. m - 1]$.

Given strings T (text) and P (pattern), the pattern matching problem consists of finding a substring of T equal to P.

B. Definitions (Complexity and Run Time Analysis)

An algorithm can be analyzed in terms of time efficiency or space utilization, while the running time of an algorithm is influenced by several factors.

- Time Complexity

Is the number of basic steps taken by algorithm, as a function of the size of its input?

$$T : N \rightarrow N$$

"Input size" can be defined in terms of the number of bits, nodes, elements, integers, and so on. A "step" is an operation that takes constant time, such as a variable
assignment, a comparison, an array access, an arithmetic function, and so on.

- **Definition 1:** Let \( f(n) \) and \( g(n) \) be two functions. We write: \( f(n) = \Theta(g(n)) \) or \( f = \Theta(g) \) (read "of \( n \) is big oh of \( g \) of \( n \)" or "of \( n \) is big oh of \( g \)") if there is a positive integer \( C \) such that \( f(n) \leq C \cdot g(n) \) for all positive integers \( n \).

- **Definition 2:** Let \( f(n) \) and \( g(n) \) be two functions. We write: \( f(n) = \Omega(g(n)) \) or \( f = \Omega(g) \) if there are positive integers \( C \) and \( N \) such that \( f(n) \geq C \cdot g(n) \) for all positive integers \( n \).

### III. Exact Pattern Matching Algorithms

A common problem in computer science, and specially in bioinformatics is to search a database of sequences for a known sequence\(^{[6,7]}\). Given a pattern string \( p = p_1 \cdots p_m \) and a longer text string \( t = t_1 \cdots t_n \), the Pattern Matching problem is to find any and all occurrences of pattern \( p \) in text \( t \). Where the pattern Matching Problem solved by using exact pattern matching algorithms. To make focus on this topic, we discuss some of well-known exact pattern matching algorithms showing main features, time complexity, as well as Pseudo code for each algorithm.

It is very important to apply all SMAs on the same example or problem in order to implement input validity and result consistency toward the measurement of execution time and the assessment of the efficiency for each SMA. With respect to the specifications of the unified matching examples or problems, a detailed description and analysis are provided in the empirical testing section.

#### A. Naive(Brute-Force) Pattern Matching Algorithm

As mentioned in \(^{[8,9]}\), brute-force pattern matching algorithm compares the pattern \( P \) with the text \( T \) for each possible shift of \( P \) relative to \( T \), until either

- a match is found, or
- all placements of the pattern have been tried

1) **Brute-Force Main Features:**

- Brute-force pattern matching runs in time \( O(nm) \)
- no preprocessing phase;
- constant extra space needed;
- always shifts the window by exactly 1 position to the right;
- comparisons can be done in any order;
- Example of worst case:
  - \( T = aab \cdots ah \)
  - \( P = aaah \)
  - may occur in images and DNA sequences

2) **Pseudo code of Brute-Force Algorithm**

#### B. Boyer-Moore Algorithm

According to \(^{[8,9]}\), The Boyer-Moore’s pattern matching algorithm is based on two heuristics:

- Looking-glass heuristic: Compare \( P \) with a subsequence of \( T \) moving backwards
- Character-jump heuristic: When a mismatch occurs at \( T[i] = c \)
  - If \( P \) contains \( c \), shift \( P \) to align the last occurrence of \( c \) in \( P \) with \( T[i] \)
  - Else, shift \( P \) to align \( P[0] \) with \( T[i+1] \)

1) **Boyer-Moore Main Features:**

- performs the comparisons from right to left;
- preprocessing phase in \( O(m+\sigma) \) time and space complexity;
- searching phase in \( O(mn) \) time complexity;
- \( 3n \) text character comparisons in the worst case when searching for a non periodic pattern;
- \( O(n/m) \) best performance.

2) **Pseudo code of Boyer-Moore:**

```plaintext
Algorithm BoyerMooreMatch(T, P, S)
L ← lastOccurrenceFunction(P, S)
i ← m - 1 ; j ← m - 1
repeat
    if T[i] = P[j]
        if j = 0
            return i [ match at i ]
        else
            i ← i - 1 ; j ← j - 1
        else [ character-jump ]
            l ← 1; i ← i - l
            i ← i + m - min(j, 1 + l)
            j ← m - 1
    until i > n - 1
return -1 [ no match ]
```
C. The KMP Algorithm

Due to the studies [8,9], Knuth-Morris-Pratt’s algorithm compares the pattern to the text in left-to-right, but shifts the pattern more intelligently than the brute-force algorithm.

1) KMP Main Features:
- performs the comparisons from left to right;
- preprocessing phase in O(m) space and time complexity;
- When a mismatch occurs, what is the most we can shift the pattern so as to avoid redundant comparisons?
- searching phase in O(n+m) time complexity (independent from the alphabet size);
- delay bounded by \( \log_\Phi(m) \) where \( \Phi \) is the golden ratio (\( \Phi = \frac{1 + \sqrt{5}}{2} \)).

2) Pseudo Code of KMP Algorithm

```
Algorithm KMPMatch(T, P)
    F ← failureFunction(P)
    i ← 0
    j ← 0
    while i < n
        if T[i] = P[j]
            if j = m - 1
                return i - j { match }
            else
                i ← i + 1
                j ← j + 1
        else
            i ← i + 1
        return -1 { no match }
```

D. The Karp-Rabin

According to [8,9] The Rabin–Karp algorithm is a string searching algorithm created by Michael O. Rabin and Richard M. Karp in 1987 that uses hashing to find any one of a set of pattern strings in a text. For text of length n and p patterns of combined length m, its average and best case running time is O(n+m) in space O(p), but its worst-case time is O(nm).

- preprocessing phase in O(m) time complexity and constant space;
- searching phase in O(mn) time complexity;
- O(n+m) expected running time.

E. Horspool Algorithm

[8,9] say that Horspool Likes the Boyer-Moore algorithm, the Horspool algorithm assumes its best case if every time in the first comparison a text symbol is found that does not occur at all in the pattern. Then the algorithm performs just O(nm) comparisons[8,9].

As in the Boyer-Moore algorithm, the pattern is compared from right to left with the text. After a complete match or in case of a mismatch, the pattern is shifted according to the pre-computed function occ.

1) Horspool Main Features:
- simplification of the Boyer-Moore algorithm;
- easy to implement;
- preprocessing phase in O(m+\( \sigma \)) time and O(\( \sigma \)) space complexity;
- searching phase in O(mn) time complexity;
- the average number of comparisons for one text character is between \( 1/\sigma \) and \( 2/(\sigma+1) \).
2) Pseudo code of Horspool Algorithm

**Shift table (p,s)**

// Purpose To fill the shift table, based on the pattern string
// Inputs p- pattern string to be searched
// Output t- shift table is returned through parameter

```
m ← length(p)
for i = 0 to 127 do
    s[i] ← m
end for
for i ← 0 to m-2 do
    s[p[i]] ← m - 1 - i
end for
return
```

**Horspool Pattern Matching (p,t)**

// Purpose To check whether the pattern string is present in the text string
// Inputs p- pattern string to be searched
t- text string where searching takes place
// Output string t if search successful -1 otherwise

```
shift_table(p,s)
n ← length(t)
m ← length(p)
i ← m - 1
while i <= n - 1 do
    k ← 0
    while k <= m - 1 and t[i-k]=p[m-1-k] do
        k ← k + 1
    end while
    end while
return -1
```

---

**Table 1: Time Complexity of String Matching Algorithms**

<table>
<thead>
<tr>
<th>String Matching algorithms</th>
<th>Preprocessing</th>
<th>Matching</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naïve (Brute Force)</td>
<td>0</td>
<td>average (m+n), Worst O(nm)</td>
</tr>
<tr>
<td>Rabin-Karp Algorithm</td>
<td>O (m)</td>
<td>average (m+n), Worst O(nm)</td>
</tr>
<tr>
<td>Knuth-Morris-Prat Algorithm</td>
<td>O (m)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Boyer-Moore Algorithm</td>
<td>O (m +</td>
<td>Σ</td>
</tr>
<tr>
<td>Horspool Algorithm</td>
<td>O(m+σ)</td>
<td>O (mn)</td>
</tr>
</tbody>
</table>

---

**F. Time Complexity of string matching Algorithms**

As we make a survey study that concerns with the time complexity as one of the measures used for revealing and assessing the efficiency of the string matching algorithms.

Actually, many theoretical studies and material in the field of time complexity are reviewed, and specially [9,10] made a theoretical comparison mentioned that all of the above 5-string matching algorithms have a preprocessing time except Naïve or Brute Force algorithm.

Let m be the length of the pattern and let n be the length of the searchable text, then the following Table 1 shows the time complexity of each string matching algorithm.

---

**IV. EMPIRICAL TESTING OF EFFICIENCY**

With respect to the second part of this study, it focuses on assessing the efficiency of each of five string matching algorithms based on the empirical testing of execution time. According to [10,11], we consider the programming environment, the hardware specification, and the size of either input text or pattern.

Regarding the first two factors, the hardware specifications and programming environment, We use ASUS laptop (with hard disk 500GB, processor i3, and 3GB Ram), and Java object-oriented programming environment respectively.

Moreover, a detailed explanation of Java code of string matching algorithms are included within this paper (see appendix A).

The additional remaining two input data factors, input text size and input pattern size, we take the needed considerations in order to implement a valid testing environment and to obtain a consistent results, where the value of execution time (nano seconds ns) is measured in accordance with only one factor at once (i.e. either text size or pattern size). In the following two sub-sections, the experimental tests of algorithms’ efficiency are explained.
A. Execution Time VS Text size

With respect to execution time, we use Java object’s function called System.nanoTime() two times, one prior to the start of executing string matching algorithm. Also we use the same function at the end of executing this algorithm. Then we find the difference in order to calculate the algorithm execution time (see Figure 1).

![Fig. 1 : Calculating execution time of SM Algorithm](image)

As the first part of our empirical testing focuses on measuring the efficiency of the five string matching algorithms based on the change of input text size(n), we find the execution time of each algorithm at three different values of text size (n=34, n=1000, n=4000) and fixed pattern size(m=11), where these values are selected randomly.

Regarding the empirical testing of algorithm’s execution time, Table 2 reveals the the execution time’s change in accordance with the increase of text size.

![Table.2 : Execution Time VS change of Text Size](image)

B. Execution Time VS Pattern Size

Table 3 reveals that there is a relationship between execution time of string matching algorithm and size of input pattern, where initially, some of algorithms appeared inefficient at m=3 however these algorithms could be efficient at different text size such as m=24. The analysis of result and discussion are explained in the next section(Results and Discussions).

![Table.3 : Execution Time VS change of pattern Size(m)](image)

V. RESULTS & DISCUSSION

In this section, the analysis and the discussion of the results is classified into two subsections, where the first one concerns with the variation of algorithms efficiency depending on the change in text size(n) as an independent factor, and the second section concerns with variation of algorithms efficiency depending on the change in pattern size(m) as an independent factor.

A. Variations in Efficiency based on Text Size change

According the empirical testing of execution time of each string matching algorithms at different size of input text(n), we find:

1. As input text size small, the most efficient algorithm, with minimum execution time, is Naïve algorithm, and the second one is Boyer SM. However the least efficient algorithm is rabin-Karp algorithm (see the below figure 2, at n=34 and m =11).

2. According to below Figure 3 &4(at n=1000, m=11 and n=4000, m=11 respectively), as the input text size increased to be 4000, the most efficient algorithm is Horspool SM. However the efficiency of Naïve is considerably decreased, and also the efficiency of other remaining algorithms are also with less rate than Naïve.

Table.2 reveals that there is a relationship between execution time of string matching algorithm and size of input text, where initially, some of algorithms appeared inefficient at n=34 however these algorithms could be efficient at different text size such as n=1000. The analysis of result discussion are explained in the next section(Results and Discussions).
B. Variations in Efficiency According to pattern Size

Due to the experimental testing of execution time of string matching algorithms at various size of input pattern(m), we find:

1. As input pattern size small, the most efficient algorithm, with minimum execution time, is Boyer algorithm, and the second one is Horspool SM, however the least efficient algorithm is rabin-Karp algorithm (see the below figure 5, at n=34 and m=3).

2. As pattern size increased as m=24, the best efficient algorithms are Boyer followed by naive then horspool (see figures 6 and 7).
VI. CONCLUSION

Practically, as the paper presents a survey for one of the remarkable topics in the fields of applied sciences such as image processing and bioinformatics [13], it could be essential for specialists and researchers working in the related fields to obtain first, a theoretical survey regarding the time execution of common exact string matching algorithms. Second, they could benefit from the measurement mechanism of SMAs efficiency as this paper finds a relationship between the lengths of the string/substring and algorithm’s execution time, moreover, it also estimates this relationship empirically. Therefore, it is found that the efficiency(E) of SMA is a function of two independent variables(length of the text (n) and the size of pattern(m)); such that E = f(m,n).

For further research paper, I strongly recommend to apply this survey to other string matching algorithms and seeking for measuring the efficiency as a function of new variables such as programming language compiler.

VII. ACKNOWLEDGEMENT

The authors of this paper would like to convey their sincere thanks to Universiti Sains Malaysia (USM) for providing the fund for this research through the Short-Term research grant Number: 304/PKOMP/6311028. Also, special thanks should go to all academic staff at School of Computer Sciences for their contribution to our academic knowledge, specially, Assoc. Professor Nur’Aini Abdul Rashid and Dr. Tan Tien Ping.

APPENDIX A

Since Java language is used as the programming environment for testing time complexity of string matching algorithms, it is essential to explain Java code of each matching algorithm.

A. Naive(Brute Force) Algorithm:

```java
public class NaiveBrute {

    // return offset of first match or -1 if no match
    public static int naiveSearch(String pat, String txt) {
        int M = pat.length();
        int N = txt.length();

        for (int i = 0; i <= N - M; i++) {
            for (int j = 0; j < M; j++) {
                if (pat.charAt(i+j) != txt.charAt(j))
                    break;
            }
            if (j == M) return i; // found at offset i
        }
        return -1; // not found
    }
}
```

B. Rabin Karp Algorithm:

```java
public class RabinKarp {

    // create the Rabin-Karp KMP-NFA from pattern
    public NFAplus(String pattern) {
        this.pattern = pattern;
        int M = pattern.length();
        next = new int[M];
        int j = -1;
        for (int i = 0; i < M; i++) {
            if (i == 0) next[i] = -1;
            else if (next[i-1] != -1) {
                if (pattern.charAt(i) == pattern.charAt(next[i-1] + 1))
                    next[i] = next[i-1] + 1;
                else
                    next[i] = -1;
            }
        }
        j = -1;
    }
}
```

C. Horspool Algorithm:

```java
private void horspoolSearch() {
    int i = 0, j = 0;
    while (i <= N - M) {
        if (j == M) {
            j = M - 1;
            while (j > 0 && p[j] != txt[i+j])
                j--;
            if (j < 0) report(i);
            i = i + j;
            j = M - 1;
        }
        i = i + j;
    }
    return;
}
```

D. Knuth-Morris-Pratt Algorithm:

```java
public class KMPlus {

    private String pattern;
    private int[] next;

    // create the Rabin-Karp KMP-NFA from pattern
    public KMPlus(String pattern) {
        this.pattern = pattern;
        int M = pattern.length();
        next = new int[M];
        int j = -1;
        for (int i = 0; i < M; i++) {
            if (i == 0) next[i] = -1;
            else if (next[i-1] != -1) {
                if (pattern.charAt(i) == pattern.charAt(next[i-1] + 1))
                    next[i] = next[i-1] + 1;
                else
                    next[i] = -1;
            }
        }
    }
}
```
E. Boyer Moore Algorithm:

```java
public int search(String txt) {
    int N = pat.length();
    int M = txt.length();
    int skip;
    for (int i = 0; i < N - M; i += skip) {
        skip = 0;
        for (int j = N-1; j >= 0; j--) {
            if (pat.charAt(j) != txt.charAt(i+j)) {
                skip = Math.max(1, j - right[txt.charAt(i+j)]);
                break;
            }
        }
        if (skip == 0) return i; // found
    }
    return M; // not found
}
```

REFERENCES


AUTHOR BIOGRAPHIES

Kamal M. H. Alhendawi is currently a Ph.D. candidate in school of Computer Sciences, Universiti Sains Malaysia (USM). He received Master degree from Islamic University, Gaza in Data Envelopment Analysis (DEA). He had received the B.Sc. in computer sciences from Abu Dies Faculty of science and technology, Jerusalem University, Palestine. He has several publications in the field of management information system and information technology, and computer science. Prior to joining USM, he has an academic experience in the fields of system analysis, software engineering, object oriented programming, and network administration. His current research interests include information systems, knowledge management, business intelligence, and E-Learning.