RESEARCH ARTICLE OPEN ACCESS

A Comparison of Single Keyword Pattern Matching Algorithms

Upendra Singh
Lecturer,Department of CSE&IT,Government Engineering College Bharatpur (Raj.) India.

Abstract:

With the growth of internet based application and services it mandatory to secure those applications. Firewall is not significantly able to secure outside the network. Thus network intrusion detection systems are used to protect them. Pattern matching is used in intrusion detection systems (NIDSs). In this paper an overview and comparison is presented of various single keyword pattern matching algorithms. We explain and compare the basic differences and run time complexity of each algorithm. Thus we can say that there is no perfect algorithm that can solves all the pattern matching problem. We present different solutions according to different scenarios.

Keywords— **Pattern Matching, Hash Value, Intrusion Detection Systems.**

I. INTRODUCTION

There are many pattern matching algorithms proposed by researchers but not all algorithms are suitable for all kind of application because some might be too slow, other might produce more false-positives, or other may need lots of system resources. Therefore we must take care about several different issues that are important when developing or implementing algorithms.

Pattern matching is the technique of searching a string containing text or binary data for some set of characters based on a specific search pattern. Pattern matching encompasses many aspects of Computer Science and Computer Engineering. Thus the pattern matching is "Given a set of keywords and an input string we need to find all instances of any keyword in the input string" [1]. The keyword set must has two properties – non-empty and finite. The set of keywords, to be non-empty, is necessary since pattern matching by definition cannot be performed against empty sets. Moreover, that set needs to be finite so that we can produce meaningful results. Pattern matching has many applications such as parsers, Intrusion Detection Systems, spam filters, digital libraries, word processors, search engines, and many more.

A. Keyword Set Size

The pattern matching algorithms are divides in two types on the basis of size of the set of keywords– single-keyword algorithms and multiple-keyword algorithms [1]. Single keyword pattern matching algorithms might have been perfectly suitable for certain applications (word processors) they were notorious for not being fast and efficient enough for other applications (search engines, IDSs). This is fact that single-keyword algorithms have rarely used in modern network security systems they are fundamentally important to explore the intricacies of pattern matching.

Multiple-keyword algorithms have many implementations, improvements. And many variations have been proposed by researches in the past decades.

B. Alphabet Size

The alphabet size also plays an important role when selecting a pattern matching algorithm to be implemented.

C. Keywords Length

The length of the keywords also plays important role in the algorithmic performance. It is evident that the greater the keyword length – the slower the algorithm will be and the more system resources will be needed to execute the comparison.

D. Computational Complexity

We can measure the performance of this algorithm in a theoretical and practical environment when we evaluate an algorithm based on the criteria of keyword size, alphabet size, and keyword set size. Computational complexity refers to the worst-case time that an algorithm can take in order to solve a given problem [2]. Average-case and best-case times provide valuable information of the algorithmic performance in specific cases but we are usually most interested in the worst-case case because it clearly defines the limitations of the algorithm.

II. SINGLE-KEYWORD PATTERN MATCHING ALGORITHMS

Single keyword pattern matching algorithms are used to find all occurrences of a specific keyword in a given input. Due to the size of the keyword set is one these types of algorithms will be very limited in modern Network Security Systems. Therefore the use of single string algorithms is limited to instances such as data processors, text editors, search engines, images analyzers, and parsers [1].

III. BRUTE FORCE ALGORITHM

The basic method of pattern matching is the Brute Force (BF) algorithm. This technique is very simple. Let's assume we have pattern (keyword) P with size m and a text (input) T with length n. we start by comparing the pattern to the text, scanning left to right, one character at a time, until there are no more matching characters[4]. If a mismatch occurs, the algorithms shift the pattern one character to the right. Here is the generic pseudo-code of the algorithm:

Algorithm I Brute Force

Step 1 Align pattern at beginning of text

Step 2 Moving from left to right and compares each character of pattern to the corresponding character in text until:

- All characters are found to match (successful search); or
- A mismatch is detected

Step 3 while pattern is not found and the text is not yet exhausted, pattern is realigns at one position to the right and repeat Step 2

In the example given below the pattern P "come" is to be found in the text T. The algorithm will search to see if the first character of the pattern - "c" – occurs anywhere in the text. If there is a match the algorithm begins comparing the second character of the pattern – "o" to the next position of the text. If this time a mismatch is occur we label this instance as a false start. The process continues until we reach the end of the string.

T: may I come in sir?

P: come

First we have a *for loop* with a variable i that keeps track of the difference between the lengths of the text and the string. This is because even if a character match is found there might not be enough characters at the end of the string to account for a complete pattern match. The *while loop* starts matching symbols of the patterns to the text, one at a time. If there is a character match it bumps the index j to the next symbol until no more matches are found.

The *for loop* is called at most n-m+1 times and the while loop is executed m times for every iteration of the outer loop so the worst-case performance of Brute Force is O(mn). In the average cases it is close to O(n+m), especially when the alphabet is large. This is because in large-alphabet languages we are much more likely to encounter repetitive strings. If the alphabet is of a smaller size the performance significantly decreases.

IV. KNUTH-MORRIS-PRATT ALGORITHM

This algorithm was introduced by Don Knuth, Jim Morris, and Vaughan Pratt in 1977. However we use the information from the previously compared characters in order to determine the maximum possible shift of the pattern to the right. This idea avoids comparisons with elements from the text T that have previously been compared with some elements of the pattern P[5]. In order to achieve this task, KMP preprocesses find matches of prefixes of the pattern with the pattern itself. The pre-calculation is done in time O(m) and is called the next function F[j] [13], which is an array that represents the size of the largest prefix of P[0…j] which is also a suffix of P[1…j][5]. The KMP algorithm states that the most we can shift the pattern in order to avoid redundant comparisons.

Algorithm IInextFunction (P)
F[0] = 0
i = 1
j = 0
while i < m //we have a continued substring in p from the start of p
if P[i] == P[j]
F[i] = j + 1
i++
j++
else if j > 0 then //use nextFunction to shift P
j = F[j - 1]
else
F[i] = 0 //no match, we weren't in a substring
i++

It uses a simple while loop to find repeated substrings of the pattern itself which are prefixes of the pattern. This way the function generates knowledge about how the pattern matches against shifts to itself. And the results are logged in a table which will be used during the second stage of the algorithm – the string matching. The first table values of F[j] are always fixed – namely "-1" and "0" because if we have a mismatch in the very first character we simply slide the pattern one position to the right, so the construction of the table actually begins at position j[2].

Based on the pattern "aabcaabbaa", below table shows the final computation of the next function. indexes of P and j will be represented by the variable i that will be the indexes of the text T. Objective of F[j] is to find the largest prefix of the pattern P[0…j] that is also a suffix of P[1…j] .

TABLE I

PRE-COMPUTE VALUE OF THE NEXT FUNCTION DEFINED IN THE KMP ALGORITHM

j	0	1	2	3	4	5	6	7	8	9
P[j]	a	a	b	c	a	a	b	b	a	a
F[j]	0	1	0	0	1	2	3	0	1	2

After that, we will assume an example below and identify the working of the algorithm.

T = aaabaabcabaabcaabbaab

P = aabcaabbaa

We start matching the characters left to right. The first mismatch occurs at T[2] ≠ P[2]. Now we use the pre-computed value of j in Table 1 to find out what is the maximum amount of shifting, instead of shifting the pattern by one. We calculate the value of j by using the formula j = j + i – F[j]. In this case we have j = 0 + 2 – 1 = 1. The initial position of j is 0 and the position where the mismatch occurred is i = 2. The index F[i] is taken from the pre-computed table by the next function. From this formula we see that the new value of j is j=1 which means we slide the pattern across to j=1 and we reset i=0. This process is continuing till pattern match is found or all text is scanned.

KMP algorithm successfully finds all the positive comparison until the end of the text and reports that the pattern is found.

Algorithm III KMPMatch (T, P)
F = nextFunction (P)
i = 0
j = 0
while i < n
if T[i] = P[j]

```
            if  j = m - 1
                return  i – j   //match
            else
                i++
                j++
        else
            if  j> 0
                    j = F[j - 1]
            else
                    i++
    return  -1     // no match
```

The construction of the table T takes O(m) time and the string matching process takes O(n) time since the complexity of KMP is O(m+n). The performance of KMP decreases as the alphabet size increases.

V. KARP- RABIN (KR) ALGORITHM

In 1987, Michael Rabin and Richard Karp proposed an algorithm for pattern matching. They used a completely different approach than the single-keyword methods. The key idea is that it involves mathematical computations, instead of using comparisons. It extends to the notion of hashing. The application of hashing has always been a useful approach when it comes down to string matching. If both words have different hash values then we can be certain they are different [3]. But if their hash values are the same we cannot say that they are the same string and will have to perform further comparisons (usually we use Brute Force algorithm for compression). Because it is impossible to give a unique value to every single substring derived from a 256-character alphabet. We assume that hash collisions will rarely occur. We define these collisions with h(x1) == h(x2), where x1! = x2. We can significantly decrease the number of hash values used by applying the modulo operation (mod) [3].

The hash function implies an algorithmic complexity of O(nm), calculating hash values for each substring of the text. For this we can represent each characters with their decimal values – “a” = 97, “b” = 98, “c” = 99 and so on. The Karp-Rabin algorithm states that for a pattern P with length m we need to calculate the hash function of every possible m-character substring in the text T, and compare if it is equal to the hash value of the pattern itself [3]. The hash function can be represented by h(k) = k (mod q) where q is a large prime integer. Each next hash value is then calculated based on the hash value of the substring in the previous position hence for each substring, the computation is done in time O(1).

For example text is “aabbcaadcabd” and pattern is “cab” then we calculate the hash values as following

T: aabbcaadcabd

P: cab

h(cab) = (99 + 97 + 98) % 3 = 294 % 3 = 0

h(aab) = (97 + 97 + 98) % 3 = 292 % 3 = 1

Both hash values are different so both strings are not same. This process is continued till a match is found. For hash value for next three characters in text we use the previous hash value as: we remove “a” and add “b” to the calculated hash value of previous hash value.

1 + 98 – 97 = 2 and 2%3 = 2 this is also not same with pattern. So we move further.

Algorithm IV Karp-Rabin-Matcher(T,P,d,q)

```
 n = length(T)
 m = length(P)
 h = dm-1  mod q
 p = 0
t0  = 0
for i = 1 to m                          //preprocessing
     p = (d*p + P[i]) mod q          //checksum of P
     t0   = (d*t0   + T[i])  mod  q          //checksum  of
T[1…m]
for s = 0 to n-m                          //matching
         if p = ts
            if P[1..m] = T[s+1..s+m]
// Checksums match.
//Now test for false positive.
          print “Pattern occurs with shift” s
             if s < n-m
                 ts+1   = (d*(ts-T[s+1]*h) + T[s+m+1])
mod q
 // Update checksum for T[s+1..s+m] using checksum
```

T[s..s+m-1]

The complexity of algorithm's is O(n) for the average case because there is only one comparison needed per m-substring. But for the worst-case scenario which is O(nm) if the prime integer is small.

VI. COMPARISON

TABLE II

COMPLEXITY OF DIFFERENT ALGORITHM

Algorithms		**Brute Force**	**Knuth-Morris-Pratt**	**Karp-Rabin**
Time Complexity	**Type**	Linear	Linear	linear on average
	Average case	O(n+m)	O(n+m)	O(n)
	Worst case	O(mn)	O(mn)	O(mn)
Search Type		Linear search	Prefix	Prefix
Key Ideas		Comparing the pattern to the text, scanning left to right.	Use the information from the previously compared characters in order to determine the maximum possible shift of the pattern to the right.	Compare the text and the patterns from their hash functions.
Approach		Linear search.	Find matches of prefixes of the pattern with the	Hashing-based.

VII. CONCLUSION

There are many algorithms that have been developed but not every algorithm is suitable. Factors such as handling large patterns, large signature sets, and variable alphabets are must take into consideration.

VIII. REFERENCES

[1]. B. Watson. *"The performance of single-keyword and multiple-keyword pattern matching algorithms"*. Eindhoven University of Technology. Department of Mathematics and Computing Science. 1994.

[2]. Lance Fortnow, Steve Homer. "*A Short History of Computational Complexity"*. NEC Research Institute. Computer Science Department. November 2002.

[3]. Plaxton, Greg. "*String Matching: Rabin-Karp Algorithm"*. Theory in Programming Practice. University of Austin, Texas. Fall 2005.

[4]. en.wikipedia.org/wiki/Brute_force

[5]. *en.wikipedia.org/wiki/Knuth–Morris–Pratt_algorithm.*