B. TECH. PROJECT REPORT

On

Performance Evaluation of Multi Pattern Matching Algorithms for Network Applications

BY

Ghanshyam Bairwa

160001022

DISCIPLINE OF COMPUTER SCIENCE AND ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY INDORE

December 2019

Performance Evaluation of Multi Pattern Matching Algorithms for Network Applications

A PROJECT REPORT

Submitted in partial fulfillment of the
requirements for the award of the degrees

of
BACHELOR OF TECHNOLOGY
in
COMPUTER SCIENCE AND ENGINEERING

Submitted by:
Ghanshyam Bairwa
160001022

Guided by:
Dr. Neminath Hubballi

INDIAN INSTITUTE OF TECHNOLOGY INDORE
December 2019

CANDIDATE'S DECLARATION

We hereby declare that the project entitled "Performance Evaluation of Multi Pattern Matching Algorithms for Network Applications" submitted in partial fulfillment for the award of the degree of Bachelor of Technology in Computer Science and Engineering completed under the supervision of Dr. Neminath Hubballi, IIT Indore is an authentic work. Further, I declare that I have not submitted this work for the award of any other degree elsewhere.

Signature and name of the student(s) with date

CERTIFICATE by BTP Guide(s)

It is certified that the above statement made by the students is correct to the best of my knowledge.

Dr. Neminath Hubballi

Associate Professor

Acknowledgments

I wish to thank my supervisor, Dr. Neminath Hubballi for his kind support and valuable guidance. It is his help and support, due to which the project design and report came to life.

The opportunity to work on a project under his guidance was a great chance for learning and professional development. In spite of being busy with his duties, he always spared time to listen to my problems, clear my doubts and guide me at every step of my project.

Ghanshyam Bairwa
160001022
B.Tech. IV Year
Discipline of Computer Science and Engineering
IIT Indore

Abstract

Multi-pattern matching algorithms find all occurrences of patterns in a given input text. We implement two multi pattern matching algorithms namely Aho-Corasick and Wu-Manber and compare the performance using their execution time. We used different set of keywords taken from network application and search them in packet payloads and evaluate the algorithms time complexity. We compare the running time of both algorithms with different implementations.

Table of Contents

Chapter 1

Multi Pattern Matching Algorithms Implementation and Performance Comparison

1.1 Introduction

For multi-pattern matching algorithms find occurrences of multiple patterns in a given text. In this work, we implemented two multi-pattern matching algorithms namely Aho-Coarasick and Wu-Manber algorithms. Aho-Coarasick uses the concept of finite automata. Wu-Manber uses the mismatching method used in Boyer-Moore algorithm. We compared these two algorithms by implementing them in both Java and C++.

1.2 Aho-Coarasick Algorithm

Aho-Corasick algorithm consists of two parts. First part is used to construct a finite state pattern matching machine from the given set of keywords whereas the second part uses the pattern matching machine to process the text string in a single pass. The time required for the construction of the pattern matching machine is proportional to the sum of the lengths of the keywords. Aho-Coarasick algorithm uses *goto, failure,* and *output* functions which are explained subsequently.

1.2.1 Preprocessing and Scanning

Consider following set of keywords *{he, she, his, hers}.* The state transition machine built using the *goto* function is shown in Fig. 1.1. In this figure, double circles denote a keyword has been identified in the input text. Algorithm 1 provides the procedure to construct *goto* and partially *output* function. In some cases, a keyword contains other keywords in its substring, for e.g., if we consider a keyword *she* from the above set it also contains a keyword *he* as a suffix which is also a keyword.

These types of cases are handled using *failure* function. The *failure* function provides the next state of a failed *goto* function. Algorithm 2 defines the *failure* function and *output* function. The start state or 0th state will not have any failed transitions. Algorithm 2 uses breadth first search to reach all states and then find the failure states. The result of *failure* function for the given example is shown in Table 1.1. Similarly, the *output* function for the given example is shown in Table 1.2.

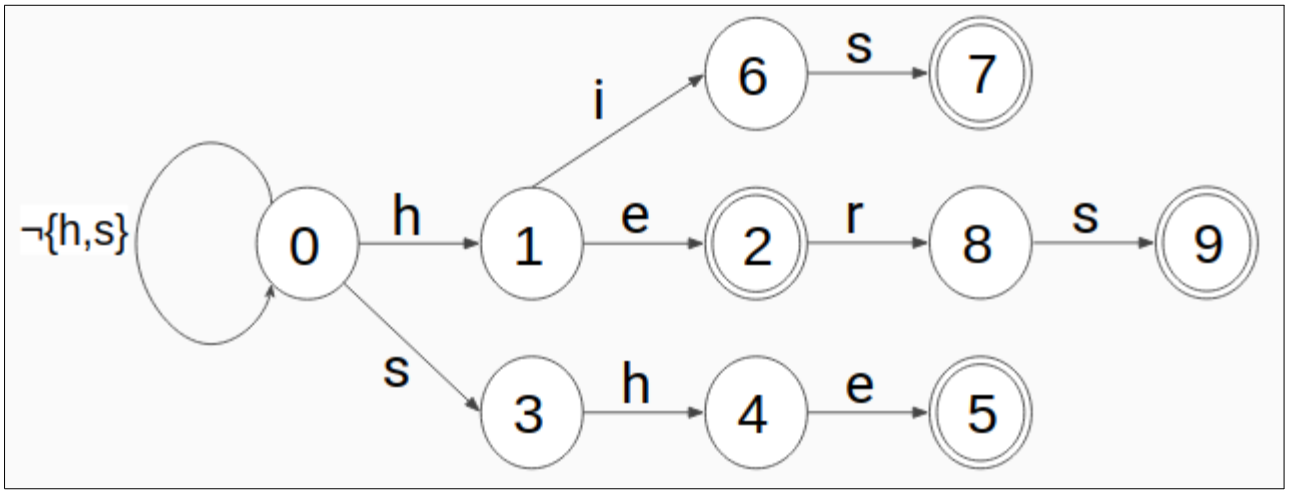

Fig. 1.1 *goto* function (finite automata)

State no. - i	1	2	3	4	5	6	7	8	9
failure(i)	0	0	0	1	2	0	3	0	3

Table 1.1 *failure function*

State no. - i	1	2	3	4	5	6	7	8	9
output(i)	{}	{he}	{}	{}	{she, he}	{}	{his}	{}	{hers}

Table 1.2. *output function*

Algorithm 3 is used for scanning the text and searching the keywords. Reading each character results in the transition between the states of finite state automata. If the next transition of *goto* function fails then *failure* function is used to identify the next state. For every state, it checks whether the state contains the keywords or not.

1.2.2 Implementation Ways

goto and *failure* functions can either be stored as arrays or can be implemented using linked lists. We used both types of data structures in our implementation. It is worth noting that accessing an element in an array takes constant time, while in linked list it takes linear time. In contrast, linked lists are dynamic and flexible and can expand and contract their size if more patterns need to be added or some are to be removed. It is advisable to use arrays if number of keywords are less, while lists can be used if there are a large number of patterns.

1.2.3 Time Complexity

In algorithm 3, the time taken for searching a keyword is twice the length of text transition. As the algorithm has to scan the whole text reading each character which requires one transition. The transition for fail cases are also needed to be considered. Algorithm 1 and 2 require linear time which is proportional to the sum of the length of keywords. Thus, the total time complexity is $n+l$, where n size of the text and l is the sum of the length of the keywords.

1.2.4 Algorithms

Algorithm 1. Construct *goto* function.

Input. keywords $K = \{y_1, y_2 \ldots\ldots y_k\}$.

Output. partially computed output function *output and* g*oto* function *g*.

Method. Initially *output(s) = NULL* and *g(s,a) = null*

begin

newstate $\leftarrow 0$

for $i \leftarrow 1$ **until** k **do** *entry*(y_i)

for $\forall$ a, such that $g(0,a) = null$ **do** $g(0, a) \leftarrow 0$

end

function *entry*$(a_1\ a_2 \ldots\ldots a_k)$

begin

$currentstate \leftarrow 0;\ j \leftarrow 1$

while $g(currentstate, a_j) \neq null$ **do**

$currentstate \leftarrow g(currentstate, a_j)$

$j \leftarrow j+l$

endwhile

for $p \leftarrow j$ **until** k **do**

$newstate \leftarrow newstate + 1$

$g(currentstate, a_p) \leftarrow newstate$

$currentstate \leftarrow newstate$

endfor

$output(currentstate) \leftarrow \{a_1\ a_2 \ldots\ldots a_k\}$

endfunction

Algorithm 2. Construct *failure* function.

Input - output of Algorithm 1

Output - output function *output and* Failure function *f* .

Method

begin

$queue \leftarrow empty$

for each a such that $g(0, a) = s \neq 0$ **do**

$queue \leftarrow queue \cup \{s\}$

$f(s) \leftarrow 0$

endfor

while $queue \neq empty$ **do**

let r be the next state in *queue*

$queue \leftarrow queue - \{r\}$

for each a such that $g(r, a) = s \neq fail$ **do**

$queue \leftarrow queue \cup \{s\}$

$state \leftarrow f(r)$

while $g(state, a) = fail$ **do**

$state \leftarrow f(state)$

endwhile

$f(s) \leftarrow g(state, a)$

$output(s) \leftarrow output(s) \cup output(f(s))$

endfor

endwhile

end

Algorithm 3. Pattern matching machine.

Input - A text string ***str*** = $a_1 a_2 \ldots\ldots a_n$ and a pattern matching machine *M*.

Output - Positions of keywords in **str**.

Method

begin

$state \leftarrow 0$

for $i \leftarrow 1$ **until** n **do**

while $g(state, a_i) = fail$ ***do***

$state \leftarrow f(state)$

endwhile

$state \leftarrow g(state, a_i)$

if $output(state) \neq empty$ **then**

print i

print $output(state)$

endif

endfor

end

1.3 Wu-Manber Algorithm

This algorithm also consists of two parts preprocessing and searching. It compares the last characters of the keywords with the text and shifts search portion if the characters do not match. The maximum shift depends on the size of the block and the size of the minimum length keyword.

1.3.1 Preprocessing Stage

First, we will find the minimum length keyword in the set, let's assume its length is *m. V*ariable *B* is used to represent block and its value should be 2 or 3. Three tables are constructed namely *Shift table, Hash table,* and *Prefix table. Shift table* is used to find the shift amount for a particular hash value. *Hash table* is used to map the shift value with the block of characters to match. *Prefix* table is used to enhance the performance of the algorithm. Algorithm 4 is for the preprocessing stage. Implementation of this algorithm requires a hash function which must be time and memory efficient. Shift value can vary from *0* to *m-B+1*.

1.3.2 Scanning Stage

In this stage, algorithms scans or read the text and search for the keywords. On scanning, it calculates the hash value for the current *B* size substring of the text with the help of the hash function. From this hash value, it calculates the shift value from the *Shift* table. Shift value has two possibilities. If the shift value is 0 then it indicates that this substring matched with the last *B* characters of at least one keyword. Therefore, we can calculate the hash value for the *Prefix* table. If this hash value gets any keyword from the *Prefix* table then we need to search for this whole keyword. If we find whole keywords matched with the substring of the text then we can print the keyword. If it fails to match then we will shift the current position by 1. The second possibility is greater than *0* shift value. It means the current sub-string is not matching with any of the keywords. So we can shift the current position according to the shift value which varies from 1 to *m-B+1*. Algorithm 5 is used for the scanning stage.

1.3.3 Complexity

The complexity of the algorithm is less than the linear time for the linear size text it is due to shifting in blocks. Let *n* number of patterns, *m* is the average size of a pattern, *l* be the size of the text, *L=mn* the total size of the patterns. *Shift* table can be formed in the O*(L).* For shift value *0* it takes *O(1)* time. But for shift value more than *0*, it uses the *Hash table* and the *Prefix table*. Therefore, the time complexity can be calculated by the following equation:

*Time complexity = O(log(2L)*l / m)*

1.3.4 Algorithms

Algorithm 4. Preprocessing

Input. Set of keywords $K = \{y_1, y_2 \ldots\ldots y_k\}$, $B = 3$(it can be 2).

Output - Prefix table *prefix,* Shift table *shift,* Hash table *hash*.

Method

begin

$m \leftarrow$ length of smallest keyword

for each keyword y_i **do** // $y_i = a_1\, a_2 \ldots\ldots a_p$

begin

for $j \leftarrow m$ **until** B **do**

Begin

$hash \leftarrow hashFunctoin(a_{j-B+1}\ldots\ldots a_{j-1}a_j)$

$shift(hash) \leftarrow minimum(m-B+1, m-j)$

end

$prefix \leftarrow hashFunction(a_1a_2)$

end

end

Algorithm 5. Scanning

Input. Text *str*, Prefix table *prefix,* Shift table *shift,* Hash table *hash*.

Output - Locations at which keywords occur in ***str***. // $str = a_1\, a_2 \ldots\ldots a_p$

Method

begin

$i \leftarrow m$

while i $\leq$ *length of str* ***do***

begin

hv1 ← *hashFunction (* a_{i-B+1}*…..*$a_{i-1}a_i$ *)*

shiftValue ← *shift (hash (hv1))*

if *shiftValue > 0* **then do** *i* ← *i + shiftValue*

else

begin

prefixValue ← *hashFunction (* $a_{i-m+1}a_{i-m+2}$ *)*

for prefix value of each keyword k_j **do**

begin

if *prefixValue = prefix(* k_j *)* **then**

scan the text from a_{i-m+1} *and keyword* k_j

if *keyword* k_j *matched* **then**

print (*j* **)**

print (k_j **)**

endif

endif

end

i ← *i + 1*

end

end

end

1.4 Result and Analysis

We performed experiments using our implementations of Aho-Coarasick and Wu-Manber algorithms on a network traffic dataset generated by collecting network packets. Keywords are searched in the payloads of reconstructed bidirectional flows. A flow is identified by 5 tuple source port, destination port, source IP, destination IP, and transport protocol type. Bidirectional flows are generated using *Jnetpcap library*. Similarly *Netinet* library is used to implement the same in C++.

We used a set of keywords to search in payload of flows. Fig 1.2 is shows the comparison of running time of WM and AC for different sizes of data. For preprocessing, AC took around 5.1 ms and WM took around 696 ms. Running time comparison for different size dataset is tabulated in Table 1.3. This result is the execution time from implementation of C++. Java implementations both the algorithms took 5 times more time than their C++ counterparts.

S.No.	Data (MB)	Aho-Corasick (sec)	Wu-Manber (sec)
1	1.3	0.032636	0.0324286
2	1.1	0.038713	0.038027
3	7.4	0.172994	0.1745674
4	14.7	0.4632056	0.4963688
5	25.8	0.5015116	0.5028964
6	47.4	1.204218	1.111128
7	49.1	1.371124	1.356192
8	76.1	2.194196	2.294584
9	292.2	6.633694	6.641762
10	387	9.10905	9.19906

Table 1.3 Time taken by AC and WM

1.5 Conclusion

In this thesis, we described performance comparison of two multi-pattern matching algorithms. Aho-Corasick and Wu-Manber to find the occurrence of patterns in the text. We then performed an experiment on a dataset using sets of keywords from network traffic. Using the results of the experiment we finally compared time complexity of both algorithms to conclude that Aho-Corasick algorithm is faster compared to Wu-Manber.

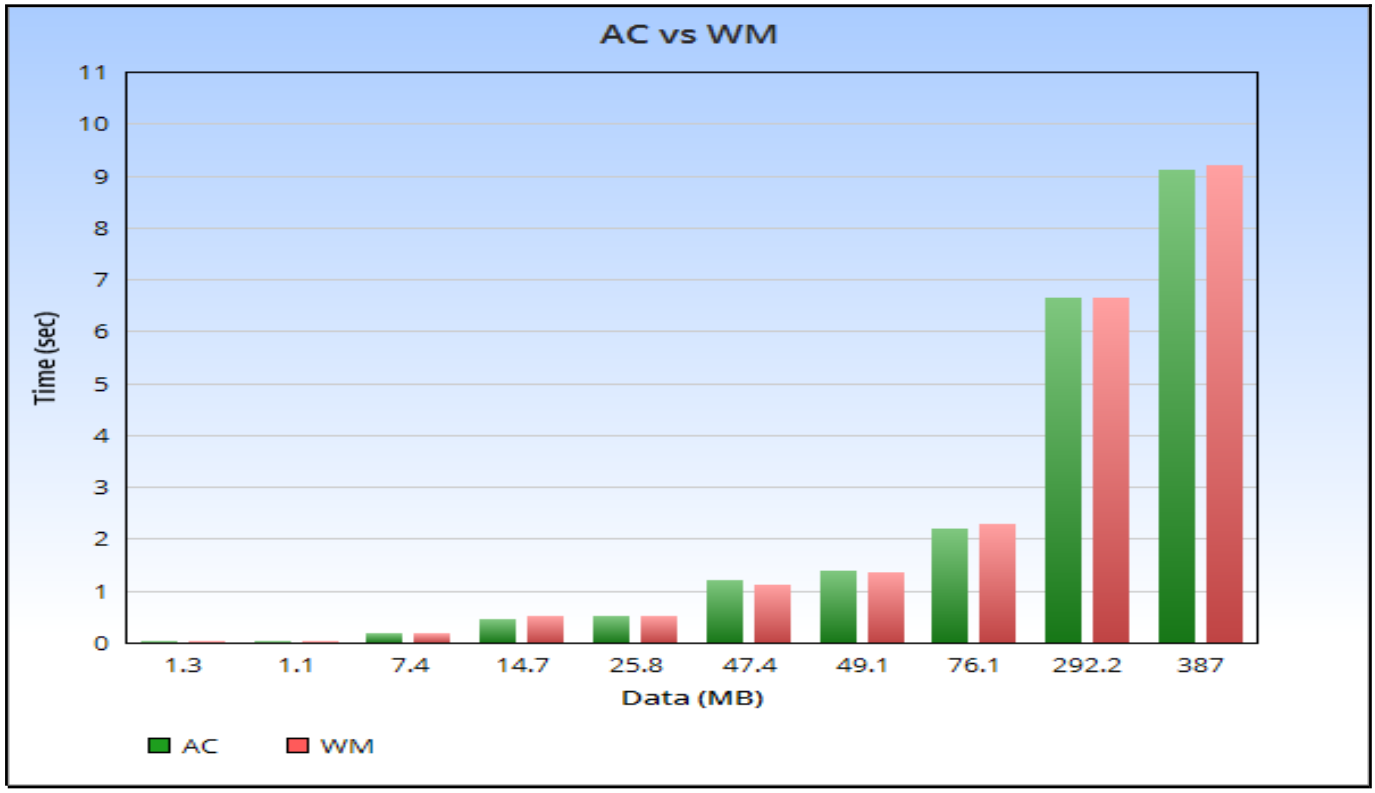

Fig 1.2 AC vs WM for scanning time comparison

1.6 Future Work

Current implementations is limited to only TCP/UDP protocols we would like to implement our algorithms to process other protocols like DCCP, FCP, SST, SPX, etc.

References

[1] Aho, Alfred V. & Corasick, Margaret J. (1975). Efficient string matching: an aid to bibliographic search. Communications of the ACM, 18, 333-340.

[2] Commentz-Walter, Beate. (1979). A string matching algorithm fast on the average. Automata Languages and Programming, 6, 118-132

[3] Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001-09-01). "The Rabin–Karp algorithm". Introduction to Algorithms (2nd ed.). Cambridge, Massachusetts: MIT Press. pp. 911–916.

[4] Baeza-Yates, R. A. (1989). ‘Improved string searching, Software - Practice and Experience, 19, 257-271.

[5] Wu, Sun & Manber, Udi. (1994). A fast algorithm for multi-pattern searching. Technical Report TR-94-17, University of Arizona.