New Shift table Algorithm For Multiple Variable Length String Pattern Matching

Punit Kanuga
IEEE Member
punitkanuga@gmail.com

***Abstract*— Multiple string pattern matching is an approach to find all occurrences of a set of patterns in given text. Efficiency of searching technique lies in development of an accurate & precise shift table. In case of a mismatch between text and pattern, shift table determines maximum length of part of text which can be skipped without missing any pattern match. This paper extends Boyer Moore concept to cultivate a shift table algorithm which works on multiple variable length patterns and can cohesively be used with various searching techniques enhancing their speed. Run-time complexity of the presented algorithm is O(N) where N denotes sum of lengths of all variable length patterns.**

***Keywords*— Multiple string pattern matching, Leftover, Join, Mismatch, Bad character heuristics, Good suffix heuristics, Shift table, Shift distance, String matching.**

1. INTRODUCTION

Multiple string pattern matching is one of the open areas in which a lot of research is being done in computer science. It finds its use in variety of fields such as database queries, operating system commands, bioinformatics, network intrusion detection systems [15], matching DNA sequences [12] [13], search engines[14] and many more. It can be summarised as a problem where objective is to simultaneously search presence of a set of strings (called patterns) in larger string set (called text). This text could either be predetermined or may be generating at run time. Thus, our approach determines correlations among patterns depending on characters constituting them. A shift table is a table which reflects maximum number of characters which can be skipped in text when mismatch happens while searching pattern in it ensuring that no possible match is missed. Overall, shift table reduces search time for searching patterns in text.

2. RELATED WORK

Pattern matching is a popular research area with lot of efforts put in by various researchers. Knuth, Morris, Pratt developed an algorithm which could search patterns within a larger string in time proportional to sum of lengths of all patterns [1]. Further, Aho-Corasick merged this idea with use of automata to narrow down search time logarithmically [2]. Boyer Moore used order of occurrences of characters in patterns to reduce run time of searching by bad character and good suffix heuristics [3]. This algorithm also gave an idea to start matching from end of pattern saving unnecessary matches. Commentz-Walter further used Boyer Moore algorithm with Aho-Corasick algorithm to propose string matching algorithm [4].Various forms of data structure and techniques have been used thus far to propose a solution to this problem. For example, Trie is used in SBOM [5] and SDBM [6]. Hashing is used by Rabin Karp [7], Wu-Manber [8], Yang-Xu-Yong [9], Khancome-Boonjing [10] and Kanuga-Chauhan[16].

3. DEFINITIONS

Let string which has to be searched be called *pattern*. Let string in which searching has to be done be called *text*. Character in *pattern* where matching fails be called *mismatch*. Part of *pattern* matched before *mismatch* be called *join* and part of pattern which remained to match after *mismatch* be called *leftover*. Thus, overall pattern will be sum of *leftover*, *mismatch* and *join*.

4. CONCEPT

4.1 Boyer Moore work for single pattern

Bad character heuristics [3] in Boyer Moore algorithm decides the shifting of *text* with respect to *pattern* in case of mismatch. This is done by finding the rightmost occurrence of text's character which causes mismatch in pattern and aligning both of them. However, in case of dynamic text generation text is not available during the generation of shift table. Objective here is to develop a shift table which works well in searching phase whether text is available statically or it appears dynamically as in case of network intrusion detection systems.

Good suffix heuristics [3] aims to search *join* in *leftover*. Best case would be finding a complete match of *join* in *leftover*. Alternatively, a sub-part of *join* can be searched in *leftover* starting from right end of *join*. Worst case is seen when even first character of *join* doesn't appear in *leftover*.

4.2 Extending idea to multiple patterns

For multiple patterns, consider window of *len-1* length, where *len* is the length of smallest pattern across multiple patterns. Window signifies the set of characters within a pattern which are considered currently. The entire pattern is

978-1-4799-7075-9/15/$31.00 ©2015 IEEE

processed in sets of windows. Window is processed from right to left direction to ensure minimum comparisons [3].

Each pattern is processed one by one. During each pattern's processing, shift distance is calculated for each of its characters. Maximum shift value for any character appearing in pattern would be *len*-1 (because that is the length of window and thus it is equal to the maximum skip of characters in *text*). We retain the minimum shift value for each character as we process each pattern. This shift value for each character guarantees the maximum possible shift with which we can skip *text* when a mismatch occurs due to that character. The entire process is explained in detail in Section V.

4.3 Extensive approach

We will consider every possible situation when a character at any position of window may cause a mismatch and we will calculate shift distance for each case. If mismatch occurs at last character of window with text (assume last character of window to be 'a'), we will check possibility of match with character which is not same as last character of the window moving from right to left within window (first character in window which is not 'a' moving from right to left). In order to align text with that character of window, we consider shift value for mismatch at last character ('a') to be equal to distance between those two characters.

In order words, we exhaustively calculate shift value for each character of *pattern* considering that it will cause mismatch with *text* during actual searching phase. Then, we will take minimum of possible set of shift values for each character. Here we assume all mismatching cases because we do not have *text* while generating shift table. This is further explained in section V.

4.4 Window Length

We take window of *len*-1 length instead of minimum length of pattern. Doing so ensures that in the worst case either beginning or end part of smallest *pattern* is considered as part of window. This guarantees finding of smallest *pattern* in the *text*.

If we take window of length *len* and if we suppose that the last character of pattern creates mismatch and further never appears in leftover again then we would take shift value as *len*. This will rule out any matching of patterns which would start from that particular last character in next *len* number of characters. Specifically, it may lead to missing possible match of pattern of shortest length only. Thus, to avoid this problem, we take window length of *len*-1 and maximum shift distance for all character appearing in any pattern to be *len*-1. This is further explained in Example 1 of section V. Shift value for all characters which do not appear in any pattern will be taken as *len* because there will be no possible match for such character and thus all combination of windows with this character can be solely avoided.

5. SHIFT TABLE ALGORITHM

Consider a shift table *shift* with two columns. First is the character column and second is the value column corresponding to each character. Let value column be initialized by length of smallest pattern (*l*).

Consider *Insert_shift(char, n)* function to enter value n corresponding to character *char* in shift table.

Consider *value(char)* function to give existing shift value of character *char* from shift table.

Consider *window* being the characters being considered in pattern.

Consider *windowlength* = length of smallest pattern-1

Consider matching starts from right to left position inside window, ie, character at last position of window is matched first, followed by last two characters combined and so on.

Consider *join* to be the part of pattern which is assumed to be matched with text till a mismatch occurs.

Consider *leftover* be the part of pattern which is still to be matched after a mismatch occurs.

Consider L to be length of pattern being processed.

Algorithm:

1. For each pattern
2. For 1 to (L - *windowlength* +1)
3. Consider *window* of *windowlength* characters starting from left of *pattern*
4. If (let) mismatch occurs at last character *char* of window
5. Insert_shift(char, distance with first different character)
6. Else if (let) mismatch occurs at any other character *char*
7. X = length of join
8. pos = position of mismatch + 1
9. while (X > 0)
10. find match of string[pos,window end] with any part of leftover moving from right to left
11. if match found
12. calculate Y
13. Where Y is the shift needed to align string with leftover
14. if(Y < value(char))
15. Insert_shift (char, Y)
16. Endif
17. Break out of while loop
18. else
19. X = X – 1
20. pos = pos + 1
21. End if
22. End while
23. end if
24. if(X = 0)
25. Insert_shift (char, *windowlength*)
26. End if
27. Move window by one position towards right till end of pattern is reached
28. End for
29. End for

6. SHIFT TABLE CONSTRUCTION

Consider set of patterns P {*aaba, aabab, aababc, aababcd, aababcde, abcb, zmnd, qope,jmqfm*}.
Minimum length of pattern here is 4. Thus, size of window would be 3. Let us observe how processing of two of the above patterns is done.
Consider pattern *aababc*.

Window 1: aab

If mismatch happens at last character b, then we can shift this window by 1 towards right with respect to text to get nearest different character. Thus we consider shift table as {(b,1)}, ie, shift value of character b is 1.

If mismatch happen at middle character a, join = b and leftover is a. Here we are sure that last character that was matched in text was b. There is no match of any part of join in leftover. So now shift table would be {(a,3), (b,1)}.

If mismatch happens at first character a, join = ab and leftover is NULL. There is no match of any part of join in leftover, neither ab or b alone. Now shift table remains as {(a,3), (b,1)}.

Window 2: aba

If mismatch happens at last character a, shift value for a will be 1 which is smaller than existing value for a in table. Thus, shift table will update as {(a,1), (b,1)}.

If mismatch happen at middle character b, join = a and leftover is a. Last matched character in text was a. Thus, we can shift pattern by 2 position to get align a (of join) with a (of text). But existing shift value for a is 1 which is less than 2. Thus, shift table remains as {(a,1), (b,1)}.

If mismatch happens at first character a, join = ba and leftover = NULL. There is no match of any part of join in leftover. So shift table remains as {(a,1), (b,1)}.

Window 3: bab

If mismatch occurs at last character b, shift will be 1. If mismatch occurs at a, shift will be 2 to align leftover b with matched text character b.

If mismatch occurs at first character b, join ab cannot be found. Further, join b cannot be found. So shift will be 3. Thus, shift table will be {(a,1), (b,1)}.

Window 4: abc

If mismatch occurs at last character c, shift will be 1. If mismatch occurs at b, join will be c and leftover will be a. We cannot find join in leftover. So shift value will be 3.

If mismatch occurs at a, join will be bc and leftover will be NULL. So shift value will be 3. Thus, shift table will be {(a,1), (b,1),(c,1)}.

Consider pattern *jmqfm*.

Window 1: jmq

If mismatch occurs at last character q, shift will be 1. If mismatch occurs at m, join will be q and leftover will be j. We cannot find join in leftover. So shift value will be 3.

If mismatch occurs at j, join will be mq and leftover will be NULL. So shift value will be 3. Thus, shift table will be {(q,1), (m,3),(j,3)}.

Window 2: mqf

If mismatch occurs at last character f, shift will be 1. If mismatch occurs at q, join will be f and leftover will be m. We cannot find join in leftover. So shift value will be 3. But this is greater than existing (q,1). So no change will be made in shift table.

If mismatch occurs at m, join will be qf and leftover will be NULL. So shift value will be 3. Thus, shift table will be {(q,1), (m,3),(j,3),(f,1)}.

Window 3: qfm

If mismatch occurs at last character m, shift will be 1. This is smaller than (m,3). So it will change. If mismatch occurs at f, join will be m and leftover will be q. We cannot find join in leftover. So shift value will be 3 but already lesser shift value is assigned.

If mismatch occurs at q, join will be fm and leftover will be NULL. So shift value will be 3. Thus, shift table will be {(q,1), (m,1),(j,3),(f,1)}.

This illustrates the calculation procedure for various conditions that appear during the process. Shift table for complete pattern set P can be calculated in similar manner and is given in next section in table III.

Example 1: This example demonstrates need of taking window length equal to one short of smallest pattern length.
Let there be a pattern *zmnd* and z appears only here in this pattern together will all other patterns. Let shift value of z be 4 (assuming we take maximum shift distance equal to *l* instead of *l-1*). Let there be a text (…aabab*cdez*mndjmq…) with current window consisting of *cdez*. Shift value for z is 4. Thus, next window will be *mndj*. Thus, we can see that pattern *zmnd* is missed. Thus, we take maximum shift value for any character appearing across all patterns to be *l-1*.

7. RESULTS AND DISCUSSIONS

7.1 Implementation Result

Proposed shift table algorithm can be used along with various multiple pattern matching algorithms which use shift tables at their core. On such algorithm is carved by Khancome and Boonjing [11]. Taking case study mentioned in this paper, we implement proposed shift table algorithm to search set of patterns P = {*aaba, aabab, aababc, aababcd, aababcde, abcb, zmnd, qope, jmqfm*}. Table I shows the shift values obtained. '*' represents all other characters of alphabet set except those mentioned.

TABLE I. PROPOSED ALGORITHM'S SHIFT VALUES

Character	Experimental shift value
a	1
b	1
c	1
d	1
e	1
f	1
m	1
n	1
o	3
p	1
q	1
j	3
z	3
*	4

Using these values of shift table, we search pattern set P in text T = *aababcdezmndjmqfmaababcd* using two hashing table technique [11]. We are able to search all patterns successfully. Time taken by this algorithm can be further reduced by use of fast access data structures such as hash table during generation of shift table. It will reduce order of time taken because matching of *join* with *leftover* will take O(1) time. This algorithm works well for variable size of patterns as illustrated in the example above.

7.2 Effect of order of characters in patterns

The advantage of this algorithm is that it takes into account number of occurrence as well as relative order of appearance of characters. Thus, it successfully differentiates between patterns *abba* and *abab*. It takes into account the length of individual patterns and thus processing of each character is done depending on its position and neighbourhood.

Additionally, it aims at finding biggest length match between *join* and *leftover* to ensure maximum shift distance possible. This enables it to reduce searching time which is crucial is situations where text is of unknown length or is generated dynamically at run time.

7.3 Dynamic Text

The shift table is generated taking into account various scenarios of mismatch of patterns with text. Text is not actually present when we are carving shift table. This can be leveraged in cases where we are sure of what to be searched but field of search is defined only at the run-time.

One of the many similar areas is of intrusion detection system. Signatures of various viruses are pre-registered and shift table can be generated on basis of them. Network is monitored at run-time.

Another similar application is firewall system where shift table can be generated on the basis of formats or keywords which need to be filtered. Traffic across the network can be scanned at run-time to filter any suspected transfer.

8. COMPLEXITY ANALYSIS

For a single pattern with,
P: Length of single pattern
W: Size of window
Mathematically runtime of this algorithm on single pattern will be O(P-W+1). Effectively, runtime complexity will be O(P) considering pattern to be much larger than window.

For a set of variable length patterns with,
X: Number of pattern
N: Sum of lengths of all patterns
M: Average pattern length
L: Smallest pattern length, thus W=L-1

Mathematically runtime of generating shift table by presented algorithm is O(N/M*(M-W+1)). Considering, N is much larger in comparison to L, effectively runtime complexity for presented algorithm can be given as O(N).

9. CONCLUSION

A shift table algorithm which is able to solve problem of variable length multiple string pattern matching is presented. This is inspired by Boyer Moore concept for single pattern matching. It further successfully enhances that concept to gain promising results. With use of hash table to search occurrence of join in leftover this algorithm takes O(|P|) time where |P| defines sum of all pattern lengths.

As shown in results section, it produces shift table which works well with existing multiple string pattern matching algorithms. This shift table algorithm ensures that we are able to pre-process pattern set for successful search in case of both static as well as dynamic text.

REFERENCES

[1] D.E. Knuth, J.H. Morris, V.R Pratt, "Fast pattern matching in strings", SIAM Journal on Computing 6(1), 1997, pp.323-350.
[2] A. V. Aho, and M. J. Corasick, "Efficient string matching: An aid to bibliographic search", Comm. ACM, 1975, pp.333-340.
[3] R.S. Boyer, and J.S. Moore, "A fast string searching algorithm", Communications of the ACM, 20(10), 1977, pp.762-772.
[4] B. Commentz-Walter, "A string matching algorithm fast on the average", In Proceedings of the Sixth International Collogium on Automata Languagees and Programming, 1979, pp.118-132.
[5] C. Allauzen, and M. Raffinot, "Factor oracle of a set of words", Technical report 99-11, Institute Gaspard Monge, Universitě de Marne-la-Vallěe, 1999.
[6] G. Navarro, and M. Raffinot, "Flexible Pattern Matching in Strings", The press Syndicate of The University of Cambridge. 2002.
[7] K. M. Karp, and M.O. Rabin, "Efficient randomized pattern matching algorithms" IBM Journal of Research and Development, 31(2), 1987, pp.249-260.
[8] S. Wu, and U. Manber, "A fast algorithm for multi-pattern searching", Report tr-94-17, Department of Computer Science, University of Arizona, Tucson, AZ, 1994.

[9] Y. Hong, D. X. Ke, and C. Yong, "An improved Wu-Manber multiple patterns matching algorithm", Performance, Computing, and Communications Conference, 2006. IPCCC 2006. 25th IEEE International 10-12, 2006, pp.675-680.

[10] C. Khancome and V. Boonjing, "New Hashing-Based Multiple String Pattern Matching Algorithms", 2012 Ninth International Conference on Information Technology- New Generations, (ITNG 2012), LasVegas, USA, 2-4 April 2012, pp.195-200.

[11] C. Khancome, V. Boonjing and P. Chanvarasuth, "A Two-Hashing Table Multiple String Pattern Matching Algorithm", 2013 Tenth International Conference on Information Technology- New Generations (ITNG 2013), LasVegas, USA, 15-17 April 2013, pp.696-701.

[12] M. Makula and L. Benuskova "Interactive visualisation of oligomer frequency in DNA", Computing and Informatics, Vol. 28, No. 5, 2009, pp. 695-710.

[13] S. Kuruppu, B. Beresford-Smith, T. Conway, and J. Zobel "Iterative dictionary construction for compression of large DNA data sets", IEEE/ACM transactions on Computational Biology and Bioinformatics, Vol. 9, No. 1, 2012, pp. 137- 149.

[14] P. Lu., Y. Che, and Z. Wangk, "UMDA/S: An effective iterative compilation algorithm for parameter search", Computing and Informatics, Vol. 29, No. 6+, 2010, pp. 1159-1179.

[15] P.C. Bosnjak, and S. M. Cisar, "EWMA based threshold algorithm for intrusion detection", Computing and Informatics, Vol. 29 No. 6+, 2010, pp. 1089-1101.

[16] P. Kanuga, A. Chauhan, "Adaptive Hashing Based Multiple Variable Length Pattern Search Algorithm for Large Data Sets", International Conference on Data Science and Engineering, (ICDSE 2014), Cochin, 26-28 August 2014, pp. 130-135.