Automata-based Algorithm for Multiple Word Matching

Majed AbuSafiya
Software Engineering Department
Al-Ahliyya Amman University, Amman, Jordan

***Abstract*—In this paper, an automata-based algorithm that finds the valid shifts of a given set of words *W* in text *T* is presented. Unlike known string matching algorithms, a preprocessing phase is applied to *T* and not to the words being searched for. In this phase, a deterministic finite state automaton (DFA) that recognizes the words in *T* is built and is augmented with their shifts in *T*. The preprocessing phase is relatively expensive in terms of time and space. However, it needs to be done once for any number of words to match in a given text document. The algorithm is analyzed for complexity, implemented and compared with an adjusted version of KMP algorithm. It showed better performance than KMP algorithm for large number of words to match in *T*.**

Keywords—Algorithms; finite state automata; word matching; KMP

I. Introduction

In this paper, a special case of string matching [1] problem is considered that is called *multiple word matching*. Its input is a set of words *W* to match in a text string *T* of length *n*. Its output is a vector of the valid shifts of each word of *W* in *T*. The motivation for this research is that it is common to have a text document that need to be repeatedly searched for single words. Another motivation is the speed illustrated by the proposed algorithm to solve this problem compared with other matching algorithms for large |*W*|.

The proposed solution is based on a preprocessing phase that is applied on *T* not on the words to search for. The idea is based on scanning the words in *T* and incrementally building a deterministic finite automaton (DFA) [2] that recognizes only the words of *T*. Once created, the DFA is used to search for a set of words *W* (repetition of words in *W* is allowed). Although building this DFA is time consuming, it is needed to be built only once for searching any number of words in *T*. The search time for the individual words will be $O(m \times |\Sigma|)$ where *m* is the length of the word searched for and $|\Sigma|$ is the size of the alphabet. This means that search time will be independent of the length of *T*. The algorithm does better than other matching algorithms only in case of a large number of word searches in *T* is needed.

This paper is organized as follows: Section 2 introduces related work. Section 3 presents the proposed algorithm. Section 4 gives a rough complexity analysis of the proposed algorithm. Section 5 shows the experimental study that was conducted to compare the proposed algorithm with KMP string matching algorithm that is adjusted for multiple search words. The paper ends up with a conclusion and a list of references.

II. Related Work

String matching algorithms are well-known class of algorithms that have two inputs: a string to search in of length n called T, and a pattern string to search for of length m called P. Their output is the valid shifts of P in T. The simplest and the most expensive among these algorithms – with complexity O(m - □ is) the Naïve string matching algorithm [1]. In this algorithm, P is compared with every sub-string in T of length m. Many string matching algorithms with better efficiency were invented such as Boyer-Moore[3], Knuth-Morris-Pratt[4], Karp-Rabin [5], Horspool [6], Quick search [7], Shift-Or [8], Raita [9], Berry-Ravendran [10]. Knuth-Morris-Pratt (KMP) algorithm is widely known and proven to be a very efficient and generic. Its complexity is O(n) for small m. It requires computing a prefix-function on P, which costs O(m), prior matching against T. A strong relation between string matching and the theory of finite automata exists, and this was discussed in detail in [11]. A very close work related to our work is the work of Aho-Corasick [12]. Their algorithm searches for a set of words in T by constructing a finite state automaton to recognize these words. This finite state automaton is then used to find the occurrences of these words in T. The main difference between our work and theirs is that in our algorithm, a finite automaton to recognize the words of T and not the words to search for is constructed. This means that Aho-Corasick approach will require O(n) string matching complexity, and our approach will have O(|W|*m) where |W| is the number of words to search for and m is the length of words which is known to be short compared to n in the context of natural languages text. However, our algorithm pays for this shorter search time, by a pre-processing phase that takes longer time. This is because constructing a finite state automaton for T takes longer time. On the other hand, Aho-Corasick algorithm constructs the finite state machine for the words to search for, which is usually much smaller than the set of words of T.

The difference between our algorithm and other string matching algorithms can be summarizes in two points: (1) Ours matches single words. So, m for our problem is relatively short. This means that our algorithm is less generic than other string matching algorithms where a pattern could be a sub-word or multiple words. (2) Ours is directed to solve the multiple word matching problem. The input is a set of words for each to be matched in T. One run of our algorithm will serve multiple search requests. Other string matching algorithms serve a search for one pattern in a single run. However, these string matching algorithms can be simply adjusted to solve the multiple word matching problem by repeatedly applying them on a set of words on the same T.

KMP algorithm was chosen to evaluate the performance of our proposed algorithm. This algorithm is among the best and most generic known string matching algorithms. KMP is adjusted slightly to do multiple word search and hence can be used to study the performance of our algorithm. Through this comparison, the circumstances where the proposed algorithm out-performs other string matching algorithms is explored.

III. Proposed Algorithm

MULTIPLE-WORD-MATCHING algorithm is shown in (Fig. 1). The input of the algorithm is a text to search *T* in and a set of words *W* to search for. The multiple searches for words in *T* is passed as an input to the algorithm. However, our algorithm may also be applied in the context where repeated search requests (for words in *T*) successively arrive in the same session. A *word* is to be a sequence of characters that does not contain spaces nor white characters. It is the same known concept of "word" in the context of natural languages. Our algorithm will only match single words in *T*. So, patterns that are sub-words or multiple words will not be matched by our algorithm. For example, if *T*=<*abc abd*>, our algorithm will assume that the only words existing in *T* are *abc* and *abd*. It will assume the strings "*ab*" and "*abc abd*" do not exist in *T*. This assumption is considered for simplicity. The output of the algorithm will be a vector of the valid shifts (in *T*) for each *w* in *W*. The first step is to build a DFA that recognizes the words of *T*. Then, GET-SHIFTS(DFA,*w*) is called for every *w* in *W* and the valid shifts are returned. It is assumed that a *w* has an attribute called *shiftVector* that will be set by the shift vector that is returned by GET-SHIFTS(DFA,*w*).

The algorithm for BUILD-DFA(T) is shown in Fig. 2. In this algorithm, the DFA is initialized where the start state is created and its name field is set to the empty string. Each state s in the DFA will be augmented with a name field which corresponds to the string that takes the DFA from the start state to this state s. The loop will get the words of T one at a time and then add them to the DFA along with their shifts in T. ADD-TO-DFA will be called once for every word in T. The shift variable is updated to contain the shift of the next word by adding the shift of the current word, its length plus 1. For simplicity, T is assumed to be normalized. This means that T contains only words and these words are separated by single spaces. Additional processing may be needed to do this normalization. This assumption eases the calculation of the shifts of the words in T.

ADD-TO-DFA algorithm (Fig. 3) will set the currentState to be start state of the DFA. The loop gets the letters of the word, one at a time. In each iteration, the nextLetter of the word is taken. A transition with nextLetter from the currentState is checked. If no such transition was found, nextState will be null. This requires that a new state (is called nextState) to be created with a transition from the currentState to nextState and is labeled with nextLetter. The name field of the nextState will be set to be the concatenation of name field of the currentState with the nextLetter. The currentState is set to be the nextState. This should be done in each iteration, whether if nextState was created or found. Once the loop terminates, all the letters of the word are consumed. The algorithm will set the current State to be a final state and shift is added to the shift vector of this final state. Note that only the final states are augmented with shifts vector. This is because augmenting all the states with shift vectors will result in too large shift vectors, especially for these states that are shallow in the DFA.

To illustrate this algorithm with an example (Fig. 4), assume that T=<ab ac a>. Adding the word ab to the DFA will be done by calling ADD-TO-DFA("ab", 0). The name field is shown for all states. For example, s_2.name is the word ab which corresponds to the prefix of the word ab that takes the DFA from the start state s_0 to s_2. The name field of the start state (s_0) is the empty string. Only s_2 has the attribute shiftVector since it is a final state. Final states are distinguished with a different color.

The next call will be ADD-TO-DFA("ac", 3) because the next word in T is ac with shift equals 3. The currentState is set to s_0.The word length is 2. So the loop will iterate twice. In the first iteration, nextLetter will be the letter 'a'. The algorithm finds a transition from s_0 with 'a'. A nextState is found which is s_1.The currentState becomes s_1. In the second iteration, no transition from s_1 with letter 'c' is found. So, a new state is created which is s_3. When the loop terminates, the s_3 is set to be a final state and the shift is added to the shift vector of s_3. The DFA will be as shown in Fig. 5.

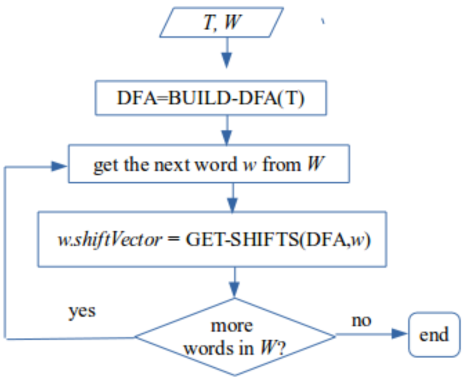

Fig. 1. Multiple-Word-Matching Algorithm.

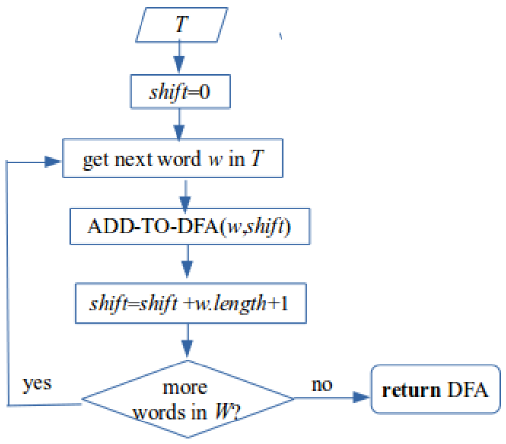

Fig. 2. BUILD-DFA Algorithm.

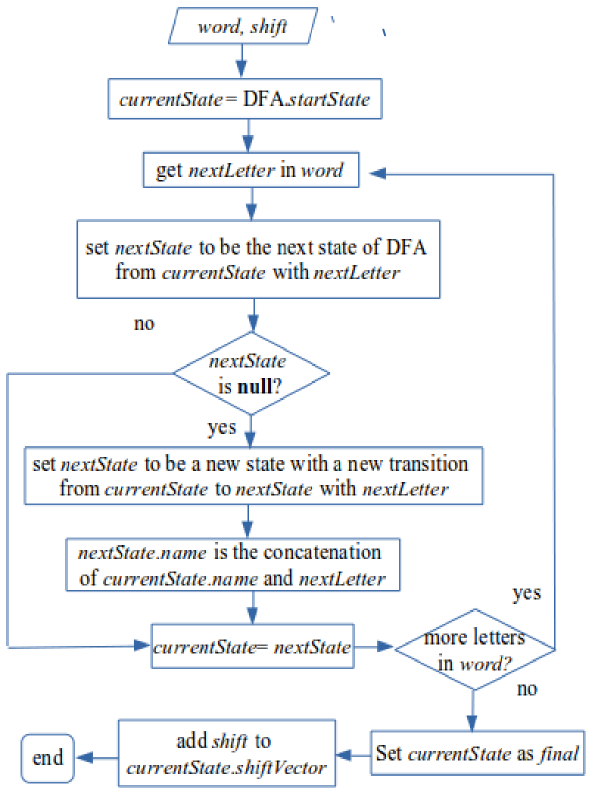
word, shift
currentState= DFA.startState
get nextLetter in word
set nextState to be the next state of DFA from currentState with nextLetter
no
nextState is null?
yes
set nextState to be a new state with a new transition from currentState to nextState with nextLetter
nextState.name is the concatenation of currentState.name and nextLetter
currentState= nextState
more letters in word?
yes
no
Set currentState as final
add shift to currentState.shiftVector
end

Fig. 3. ADD-TO-DFA Algorithm.

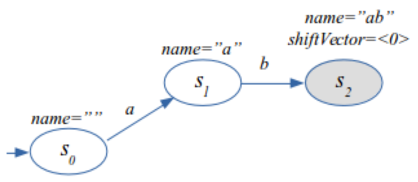

Fig. 4. DFA after ADD-TO-DFA(*"ab"*, *0*) Call.

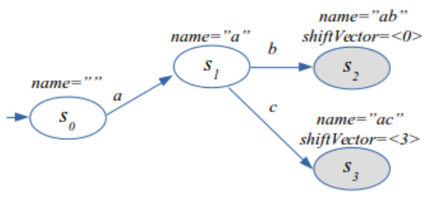

Fig. 5. DFA after ADD-TO-DFA(*"ac"*, 3).

ADD-TO-DFA will be called for the third word in *T* which is "*a*" ending with DFA in Fig. 6. Note that s_1 became a final state and a shift vector is set.

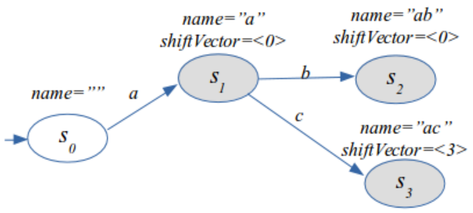

Fig. 6. DFA for *T*=<ab ac a>.

Next, how the DFA is used to get the shifts of a word w in T need to be defined. This is done by calling GET-SHIFTS(DFA, w) as shown in Fig. 7. Initially, the variable currentState contains the start state. It will change to represent the state that is reached while scanning the letters of w. The letters of w are taken one by one. A check for a next state from the currentState with letter is made. If not found, this means that w does not exist in T and an empty shift vector is returned. However, if a next state is found, the currentState is updated to be the nextState. The loop will break either when (1) a null state is reached which means that w does not exist in T or (2) all the letters of the word where consumed. In case all the letters of w were consumed ending in a final state, then w is in T and the shift vector (augmented in the reached final state) is returned. An empty shifts vector is returned when (1) w could not be completely consumed because of reaching a null state, or (2) if w was completely consumed but a non-final state was reached.

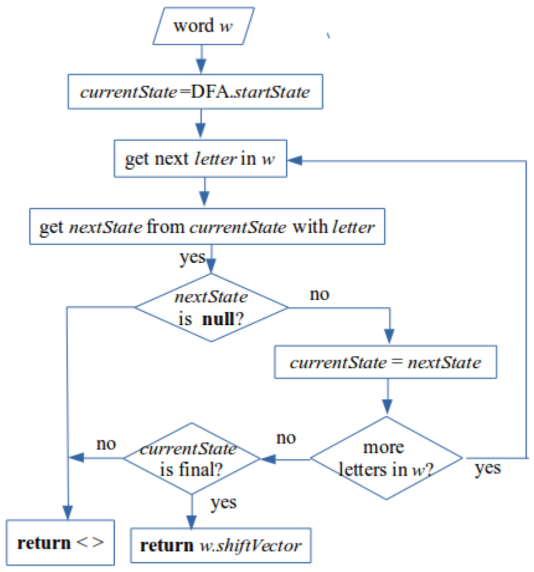

Fig. 7. Get-Shifts(DFA,*w*).

For example, to search for the word "ac" in T using the DFA in Fig. 6, GET-SHIFTS(DFA,"ac") is called. The nextLetter will be "a" and next state will be found which will be s_1 which is not null. This will result in another loop iteration where next letter will be "c". The next state will be s_3 which is not null. The loop will break and since all w's letters were consumed. Since the reached state (s_3) is a final state, the shift vector will be returned which is <3>.

IV. Analysis of the Proposed Algorithm

To analyze the time complexity of the MULTIPLE-WORD-MATCHING algorithm, for each step, the following need to be found (1) its time complexity for a single run and (2) the number of times it's executed. These two values are multiplied and added up for all the steps. A bottom-up approach is taken, where we start analyzing the supporting algorithms and then find the complexity of the main algorithm.

Starting with ADD-TO-DFA, each statement is executed only once except for the statements within the loop which will be run m times in worse case where m is the length of the word. All the steps take constant time to execute except for the step of finding the nextState for currentState with letter. It requires scanning the next states of the currentState to find a transition with nextLetter label. An upper bound on the number of the next states for a currentState is the size of the alphabet of the text language $|\Sigma|$. This is a loose upper bound, because in the words in natural languages, not all letters may appear next to a given letter. Adding the complexities of the statements, it is found that the complexity of ADD-TO-DFA algorithm is $O(m\times|\Sigma|)$ which is constant and is independent of the size of T.

For a BUILD-DFA call, the loop will iterate a number of times equals to the number of the words in *T*. All the steps within the loop are of constant time complexity except for the step (ADD-TO-DFA call) which is $O(m\times|\Sigma|)$. The time complexity of BUILD-DFA will be $O(n\times m\times|\Sigma|)$. A tighter bound can be given, since the number of words multiplied by *m* will be roughly equal to *n*. That is, it can be said that the time complexity of BUILD-DFA will be $O(n\times|\Sigma|)$.

The statements of GET-SHIFTS will run only once, each with constant time complexity, except for the loop statements. The statements of the loop will run in the worst case *m* times where *m* is the length of the word that is searched for. The steps within the loop all take constant time expect for getting the *nextState* step which will take $O(|\Sigma|)$ to search for the next state for a given letter. So the time complexity of GET-SHIFTS will be $O(m\times|\Sigma|)$.

Now, the main algorithm needs to be analyzed. BUILD-DFA step will run once and its complexity is $O(n\times|\Sigma|)$. The loop will iterate a number of times equals to the size of word set *W* to be searched for (i.e $|W|$). GET-SHIFTS will be run $|W|$ times with $O(m\times|\Sigma|)$. The total complexity of GET-SHIFTS will be $O(|W|\times m\times|\Sigma|)$. So the total time complexity of the main algorithm will be $O(n\times|\Sigma|)+O(|W|\times m\times|\Sigma|)$ which will be $O(n\times|\Sigma|)+ O(|W|\times m\times|\Sigma|)$. Since we know that the length of the words *m* and $|\Sigma|$ in natural languages are relatively small constants, we can roughly say that the complexity of the MULTIPLE-WORD-MATCHING is $O(n)+O(|W|)$ for very large *n* and $|W|$.

To compare our algorithm with KMP, it was slightly adjusted to solve our multiple word matching problem (Fig. 8).

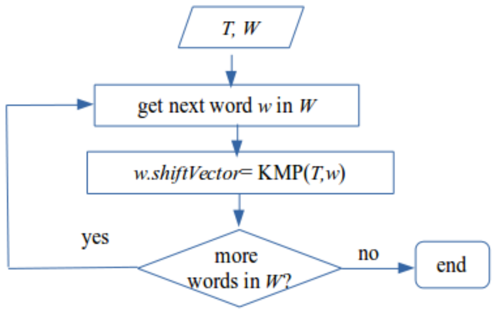

Fig. 8. Adjusted KMP.

The loop will iterate |W| times. KMP(T,w) is the same as KMP in [1] but adjusted to build a shift vector instead of printing the shifts. From [1] we know that KMP(T,w) is of O(n)+O(m) complexity. This is because, O(m) is needed to build the prefix function for w and O(n) is needed to scan T for w. KMP(T,w) will be called |W| times. So, the total complexity of the Adjusted KMP will be $O(n\times|W|)+O(m\times|W|)$. Knowing that m is relatively small constant in natural languages, it can be said that its complexity is $O(n\times|W|)+O(|W|)$ which will be $O(n\times|W|)$ for very large n and |W|.

For space complexity, our algorithm needs O(n) space. However a tighter analysis may be considered. It was found that the number of states of the DFA is linear with the set of prefixes of the words in T. Repeated words in T means less number of states. Repetition of words, is a common feature in natural language text documents. On the other hand, Adjusted KMP needs only O(m) space to store the prefix function of the current word being searched for. A comparison between MULTIPLE-WORD-MATCHING and Adjusted KMP is shown in Table I.

TABLE I. COMPARISON BETWEEN MWM AND ADJUSTED KMP

MWM	***Adjusted KMP***	***Comparison Facet***
T	*W*	Preprocessing phase is applied to
O(n□|□|)	*O(m□|W|)*	Preprocessing phase complexity for *W*
O(m□|□|)	*O(n)*	Search phase complexity for a single word
O(n)+O(|W|)	*O(n□|W|)*	Search phase complexity for a |*W*| words (|*W*| large)
O(n)	*O(m)*	Space Complexity
large W	*Small W*	Better for

V. EXPERIMENTAL STUDY

Both algorithms: MULTIPLE-WORD-MATCHING (MWM) and ADJUSTED-KMP were implemented. We chose *T* to be the text of the Holy Quran which is composed of 78,245 Arabic words. Fig. 9 shows the algorithm that was written to compare the two algorithms.

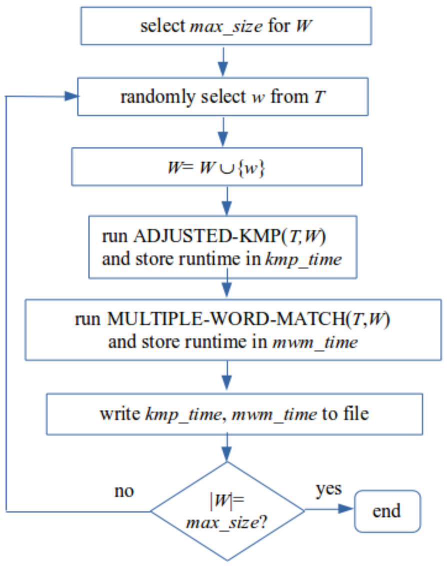
select max_size for W
randomly select w from T
W= W ∪ {w}
run ADJUSTED-KMP(T,W) and store runtime in kmp_time
run MULTIPLE-WORD-MATCH(T,W) and store runtime in mwm_time
write kmp_time, mwm_time to file
no
|W|= max_size?
yes
end

Fig. 9. Comparison Algorithm.

The comparison algorithm is based on measuring the running times for both algorithms for the growing sizes of *W*. Initially *W* is empty. The experiment was conducted by randomly selecting 200 words from the *T*. In each iteration, the newly selected word *w* is added to *W*. We record the start time, call the adjusted KMP algorithm for *W* and record the end time. The same is applied for MWM. The size of *W* and run time for both algorithms for this *W* is wrote into a file. The file is charted as shown in Fig. 10. The x-axis represents the growing |*W*| and the y-axis shows the run time needed by each algorithm. We have two graphs for each algorithm where a point (*x*,*y*) in any of these graphs means that searching for the *x* randomly selected words took *y* milliseconds.

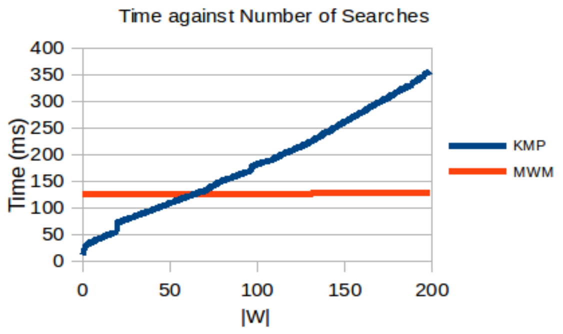

Fig. 10. Comparing MWM with Adjusted KMP.

The observations can be summarized as follows. The first search operation took too long time for MWM compared to the adjusted KMP algorithm. This is expected because of time needed to construct the DFA. However, our algorithm outperforms the adjusted KMP when |*W*| reaches 65 words. Although this number is not a fixed value, it gives a notion when our algorithm will out-perform the adjusted KMP for the given *T*. Note also that the accumulated time for MWM looks as if it is constant. However, it is increasing, but with very small value. The line has very small slope.

VI. Conclusions

In this paper, we proposed a multiple word matching algorithm. The proposed algorithm showed competitive performance only in case of a large number of word matchings is to be applied on *T*. However, it is really very expensive if small number of word matchings is required on *T*. Preprocessing of *T* may open new horizons for better text search algorithms. As future work, we wish to work on optimizations on our algorithm so that it shows better performance than the adjusted KMP on lower |*W*|. We will relax the restriction of word matching so that the algorithm can be used to search for any pattern and not only for single words.

References

[1] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, Introduction to Algorithms, 3rd ed., Massachusetts: MIT Press, 2009.

[2] J. Hopcroft, J. and Ullman, Introduction to Automata Theory, Languages and Computation, 1st ed., New York: Edison Wesley, 1979.

[3] R. Boyer, and J. Moore, “A fast string searching algorithm,” Communications of the ACM, vol. 20, pp. 762-772, 1977.

[4] D. Knuth, J, Morris, and V. Pratt, “Fast pattern matching in strings,” SIAM journal on Computing, vol. 6 No.2, pp. 323-350, 1977.

[5] R. Karp, and M. Rabin, “Efficient randomized pattern matching algorithms,” IBM journal of Research and Development, vol. 31, no. 2, pp. 249-260, 1987.

[6] R. Horspool, “Practical fast searching in strings, “Software: Practice and Experience,” vol. 10 no. 6, pp. 501-506, 1980.

[7] D. Sunday, “A very fast substring search algorithm,” Communications of the ACM, vol. 33, no. 8, pp. 132-142, 1990.

[8] R. Baeza-Yates and G. Gonnet, “A new approach to text searching,” Communications of the ACM, vol. 35, no. 10, pp. 74-82, 1992.

[9] T. Ratia, “Tuning the Boyer-Moore-Horspool string searching algorithm,” Software: Practice and Experience, vol. 22, no. 10, pp. 879-884, 1992.

[10] T. Berry, and S. Ravindran (1999), “A fast string matching algorithm and experimental results,” Proceedings of the Prague Stringology Club Workshop, Prague, Czech Republic, pp. 16-28, 1999.

[11] A. Aho, J. Hopcroft, and D. Ullman, The Design and Analysis of Computer Algorithms (1st ed.). Massachusetts: Addison Wesley. 1974.

[12] A. Aho, and M. Corasick, “Efficient string matching: an aid to bibliographic search,” Communications of the A vol. 18, no. 6, pp.333-340, 1975.