International Journal of Applied Information Systems (IJAIS) – ISSN : 2249-0868
Foundation of Computer Science FCS, New York, USA
Volume 9 – No.8, October 2015 – www.ijais.org

String Searching with DFA-based Algorithm

Preye Ejendibia
Department of Computer Science
University of Port Harcourt

Barileé B. Baridam
Department of Computer Science
University of Port Harcourt

ABSTRACT
Searching for information in large depositories or the Internet employs the concept of string searching. With the world-wide web expanding with databases from diverse fields it has become a growing concern for database curators to find an efficient searching algorithms for the task. In comparative terms, the power of an algorithm over another is in its time-complexity and efficiency of operation. A lot of algorithms have been designed for the task of string searching. Also, some of the fast string searching algorithms were developed based on the Deterministic Finite Automata (DFA), which prompts the need to thoroughly research and investigate how this principle is applied. This paper analyses and compares the searching power of DFA and brute-force searching algorithms. The DFA approach is used to overcome the problem of backtracking, which is faced with the brute-force approach thereby improving the time complexity, the speed and efficiency of search based on results obtained.

General Terms
Pattern Searching, Algorithms, Automata.

Keywords
Brute-force, DFA, Algorithms, String Searching.

1. INTRODUCTION
One of the most common problems involving strings is that of searching for occurences of a given pattern as a substring of a larger text string [5]. Text is one of the most widely used media datatypes [6], in research and development for information retrieval and data mining; because of the wealth of work done in the area of searching patterns in text files.

When information or data is been searched in large depositories or the internet, the string searching concept is said to have been employed. String searching algorithms are algorithms that are designed to search for occurrences of strings in a larger body of strings and these algorithms differ in their time-complexities and efficiency of operation.

There are quite a number of string searching algorithms[10] and they are of importance in areas such as data processing, information retrieval, text-editing, word-processing, linguistic analysis, and also in areas of molecular biology such as genetic sequence analysis. Before the advent of the classical string searching algorithms such as Knutt-Morris-Pratt, Boyer-Moore, Karp-Rabin etc. The Brute force algorithm was in use and was found to be very slow and inefficient as timing constraints is a major consideration when talking about algorithms. The advantage an algorithm has over another is in its time-complexity and efficiency of operation. There was a need to deduce more efficient algorithms which work in linear-order time as compared to the quadratic-order of time the Brute force algorithm employed.

This project provides the computational method of automata-based string searching algorithm and gives a comprehensive analysis of how strings are searched, by building and running automata as many of the efficient algorithms used in searching for information, create finite automata to effectively search for strings.

2. RELATED WORK
According to S. Mitra and T. Acharya [6], String searching is a very important area of research for successful development of data mining systems, particularly for text databases and in mining of data through the Internet by a text-based search engine. G. A. Stephens [5] defined string searching as a process of seeking a set of string (substring or subsequence) within a larger body of string. String matching algorithms with linear-order computational complexity are very useful in many practical text-based applications such as edit, search and retrieval of text, and development of search engine, and therein lies its possible influence in text data mining. The essence of developing a linear-order string matching algorithm with finite automata, to tackle the problem of buffering due to backtracking in the text string when a mismatch is encountered.

In a brute force manner [5][6], the string matching algorithm compares a pattern character by character in each and every location of the text. Starting at the beginning of the text string, the characters of the pattern is compared one after another with the corresponding characters in the text, until a mismatch is found or the complete pattern is searched exhaustively. If the pattern is exhausted, then, a match is said to have been found at the beginning of the text. If a mismatch of character is detected before the pattern is exhausted, then the pattern does not occur at the beginning of the text. The matching is started all over again at the next character in the text, and the same procedure is continued repeatedly. The brute force approach requires the input text string to be buffered, because the text needs to be backtracked whenever there is an unsuccessful match with a character in the pattern. The computational complexity of the algorithm is $O(m.n)$ in the worst case.

In 1956, just a few years after the invention of ENIAC, Stephen C. Kleene proved [8] the equivalence between finite automaton and regular expressions, which lead to solving the string searching problem in time linear in order, $O(m + n)$. For a long time, many string pattern recognition problems were formulated in terms of finite automata. This approach reduced the string matching problem to a language recognition problem.

In 1970, Morris and Pratt, [9] came up with the first linear-time algorithm to solve the string matching problem which preprocessed the pattern in $O(m)$ time and searched the text string in $O(n+m)$ time. This algorithm is able to skip comparisons by studying the internal structure of the pattern. Seven years later , in 1977, Knuth, Morris and Pratt [9]

enhanced that algorithm. Although they achieved the same time complexity, their algorithm works much better in practice. Their algorithm is the oldest and one of the most popular classical algorithms for string matching. During the search process, all the characters in the text are read forward sequentially one after another.

In 1977, around the same time that Knuth, Morris, and Pratt came out with their algoritmn, Boyer and Moore [11] proposed an algorithm that preprocessed the pattern in $O(m + |\sum|)$ where $|\sum|$ is the alphabet size, it used n/m number of comparisons and searched the text in $O(m + n)$ in the worst case. During the sarch operation of this algorithm the pattern symbols are matched starting from the last symbol which allows the pattern to shift in large jumps through the information gained and in most cases not all of the first text symbols are inspected. This algorithm is very much used in practice because of it's good performance.

The key insight of the Boyer-Moore algorithm is that some of the characters in the text can be skipped entirely without comparing them with the pattern, because it can be shown that they can never contribute to an occurrence of the pattern in the text. In Boyer-Moore algorithm, although the text is scanned left to right, comparisons of the pattern and the text are done backwards right to left along the search window while reading the longest suffix of the search window that is also a suffix of the pattern. This is a significant performance improvement as compared to prefix comparison-based Knuth-Morris-Pratt algorithm. In 1990, Sunday suggested using the symbol in the text immediately following the one that caused a mismatch to address the occurrence-heuristic table of the Boyer-Moore algorithm [14][15]. Using this approach, three variants, such as the Quick Search (QS), the Maximal Shift (MS), and the Optimal Mismatch (OM) algorithms were developed, with differences in the manner the order of the symbol comparisons between the pattern and the current text-substring is determined in each case. The QS algorithm, performes comparisons from left to right, while the MS algorithm orders the comparisons such that the distance to the next pattern position in the event of a mismatch is maximised. And the OM algorithm compares statistically rarer symbols first.

In 1980, Horspool [12] brought forward an algorithm which showed that search speed can be enhanced by comparing first the character in the search pattern that occurs least frequently. Standard search algorithms use the first letter in the pattern to compare first and the Boyer-Moore algorithm starts comparing with the last letter in the search pattern employing only a single auxiliary table indexed by the mismatching text-symbols and results in performance comparable to that of the original version.

In 1987, Karp and Rabin [13] published an algorithm that ameliorates the comparison step by computing finger prints of the pattern and the text. Their approach is similar to that of brute force searching, but rather than directly comparing the pattern symbol strings at successive text positions, their respective signatures are compared. In Karp-Rabin algorithm, instead of directly comparing the pattern characters with the text characters, the text is first pre-processed (with a preprocessing time of $O(m)$) to map into a sequence of integers. Here each character position in the text is mapped into an integer, and this sequence of numbers is then compared with a fixed integer representing the pattern. The algorithm is not restricted to string matching and may be extended to multi-dimensional pattern matching. It's worst-case running time is quadratic $O(m \cdot n)$, but when set up properly, its average case is linear $O(m + n)$.

B. Wellner and M. Dant wrote [1] that 'Grep'- a Unix utiity is an application used to look for a string search pattern in one or more text files that also displays the lines that contain the desired pattern. The first grep was developed by Ken Thompson which used a non-deterministic finite automaton. In 1976, Al Aho implemented a more powerful (in terms of search patterns) grep and called it 'Egrep' which utilized a deterministic finite automaton [1].

In 2012, N. Singla and D. Garg wrote [4] about applications of string searching algorithms and their areas of optimal performance and The algorithm of choice for text editors, digital library and search engines is the Boyer-Moore algorithm.

The Boyer-Moore-Horspool [12] algorithms achives best results when used with medical tests. The most preferred algorithm for multimedia and computational biology is the Needleman Wunsch and Smith Waterman algorithm.

According to Eric Gribkoff [3], DFA is used in protocol analysis, video game character behavior, text parsing, security analysis, natural language processing, CPU control units, and speech recognition. Additionally, many mechanical devices are frequently designed and implemented using DFAs. Example of such are elevators, vending machines, and traffic-sensitive traffic lights. System which must maintain an internal definition of state naturally uses DFAs.

3. THE STRING SEARCHING PROBLEM

Given a pattern $p = p_1p_2...p_m$ of length m and a text $t = t_1t_2...t_n$ of length n are two strings formed over the same finite alphabet $\sum$ such that $m < n$. The pattern p occurs in text t at the beginning of text location k if $1 \le k \le n - m$ and $t_{k+i-1} = p_i$ for $1 \le i \le m$. The string matching problem is the problem of finding all the text locations where the given pattern p occurs in the given text t.

For example, let text $t =$ *'c b b a b a b a a b a b a c a b a'* and a pattern $p =$ *'b a b a'* over the finite alphabet $\sum = \{a, b, c\}$. The pattern *'b a b a'* can be found in text locations *3, 5,* and *10,* respectively. A string searching algorithm has the task to efficiently locate the pattern within the text in minimal time with minimal usage of storage.

3.1 Brute Force String Searching Algorithm

The brute force string matching algorithm compares a pattern symbol by symbol in each and every location of the text. Starting at the beginning of the text string, we compare the symbols of the pattern one after another with the corresponding symbols in the text, until a mismatch is found or the complete pattern is exhausted. If the pattern is exhausted, we claim to have found a match at the beginning of the text. If a mismatch of symbol is detected before the pattern is exhausted, then the pattern does not occur at the beginning of the text. Then matching is started all over again at the next symbol in the text, and continue the same procedure.

The Algorithm

Compare pattern to text while pattern symbols is less than text symbols.

1. If first symbol of pattern is same as first symbol of text,

i. Increment pattern and text to next symbol of pattern and next symbol of text respectively and check again whether they match.

ii. Continue 1(i) as long as successive increment of pattern matches successive increment of text until end of pattern is reached and announce match success.

2. If first symbol of pattern is not same as first symbol of text ,

i. Shift pattern down text to the right by one position so that first symbol of pattern is aligned with next symbol of text and check again whether they match.

ii. Continue 2(i) as long as subsequent shifts of pattern does not match successive increment of text until text is exhausted and announce match failure.

3.2 DFA String Searching Algorithm

A finite automata search algorithm follows certain steps such as, constructing a DFA for the pattern, performing the search - each character in the text is examined just once, in sequential order; when searching is done, a state transition table for the automaton is created to represent the state transition function[2][7][16].

Given a text $t = t_1t_2t_3...t_n$ where i represents the index of the text symbols such that $1 \leq i \leq n$ and pattern $p = p_1p_2p_3...p_m$ where j represents index of pattern symbols such that $1 \leq j \leq m$ ($m < n$ where m and n are the lengths of pattern and text respectively).

The Algorithm

1. Construct a DFA for the pattern:

i. A DFA constructed for the pattern will be one state longer than the length of pattern so the DFA takes $m+1$ states which ranges from $0,1,2...m$. (Let k represent the index of states such that $0 \leq k \leq m$).

ii. Initialize start state to be 0 and final state to be m.

iii. Initialize start and last index of text to be 1 and n respectively (Let i represent the index of text such that $1 \leq i \leq n$).

iv. Initialize start and last index of pattern to be 1 and m respectively (Let j represent the index of pattern such that $1 \leq j \leq m$).

2. Performing the search:

i. Starting at state k, read each text symbol against first occurences of pattern symbol to check for match. If match, increment k to k_{next} and read the next text symbol. (The DFA will be in *kth* state if *ith* symbol(s) of text have been matched with *jth* symbol(s) of pattern).

a. If the next *ith* text symbol matches the corresponding *jth* pattern symbol, then the automaton will transit to the next *kth* state i.e $\delta(k_{current}, t_i = pj) = k_{next}$.

b. Otherwise it remains in the current *kth* state, moves to the start state or other states not exceeding the current *kth* state i.e $\delta(k_{current}, t_i \neq pj) = \{ 0 \leq k_{current} \}$.

ii. If all *jth* pattern symbols were found in sequential order in any *ith* location of text (i.e $k = m$, final state has been reached), then announce match success.

iii. If all *jth* pattern symbol couldn't be found in any *ith* location of text (i.e text is exhausted), then announce match failure.

3. Creating the transition table:

i. Create table for all states $(0 \leq k \leq m)$ from the transition function. i.e $\delta(k_{current}, t_i = p_i) = k_{next}$ for each and every element $t_i = p_i$ in the $\sum$.

Linear-order string matching algorithms are usually constructed with finite automata. This is mostly because finite-automata string searching helps to avoid the problem of buffering due to backtracking in the text to be searched.

4. EXPERIMENTAL RESULTS AND ANALYSIS

4.1 The Brute Force Algorithm

Given the text $t =$ *"ababbaabaaab"* and pattern $p =$ *"abaa"*.

Initially, the pattern is alligned with the text from text postion 1-4. The first three symbols of pattern matches with the text at positions 1, 2 and 3 and a mismatch occurs at the position 4. At this point we shift the pattern by one position to the right of text and check for match, the first pattern symbol does not match the text at position 2 so we shift the pattern again. At the second shift, the first two pattern symbols matches with the text at positions 3 and 4 and mismatches at position 5. We shift again the third time and the first pattern symbol mismatches the text at position 4 and so we carry out the fourth shift. At this shift, the first pattern does not match the text at postion 5. Shifting for the fifth time, the first pattern symbol matches with the text at position 6 and second pattern symbol does not match the text at position 7. At the sixth shift, we see that all pattern symbols matches the text from position 7-10 which implies a successful match of pattern has occured . We continue shifting, at the seventh shift, first pattern symbol does not match text at position 8 and shifting for the 8th and last time, the first pattern symbol matches and mismatches the text at position 9 and 10 respectively. At this point, the text has been exhausted and searching is terminated. The brute force search procedure is shown in Table 1.

The underlined symbols in the text column as indicated in Table 1 shows where test symbols match pattern symbols while the italicised symbols in any given position indicate that pattern does not match text at that position. Bold and underlined symbols at the 7th row in text column represents successful match. Running the brute-force algorithm on the sample data above generates 9 steps with 19 pattern-text comparisons. The result is found at the 7^{th} position of the sample text.

A comparison of this output will be made with that of the DFA.

4.2 The DFA Algorithm

Given the *text =* *"ababbaabaaab"* and *pattern =* *"abaa"*. The DFA is defined by the quintuple: $(Q, \sum, \Delta, q_0, F)$, where $Q = \{ 0, 1, 2, 3, 4 \}$, $\sum = \{ a, b \}$, $q_0 = 0$, $F = 4$, $\Delta = Q \times \sum \rightarrow Q$: $\delta(k_{current}, \sigma) = k_{next.}$ k and σ represent state and transition (input symbol) respectively.

Firstly, a DFA is constructed for the given pattern as in Figure
The remaining steps then follow:

Table 1. Tabular representation of Brute force search on the sample data(pattern)

Steps	Text	Match/mismatch position	Number of pattern-text comparison
1	aba*b*baabaaab	Mismatch: 4th position	4
2	a*b*abbaabaaab	Mismatch: 2nd position	1
3	abab*b*aabaaab	Mismatch: 5th position	3
4	aba*b*baabaaab	Mismatch: 4th position	1
5	abab*b*aabaaab	Mismatch: 5th position	1
6	ababba*a*baaab	Mismatch: 7th position	2
7	ababba**abaa**ab	Match: 7th position (Success)	4
8	ababbaa*b*aaab	Mismatch: 8th position	1
9	ababbaaba*a*ab	Mismatch: 10th position	2

Performing the search

In the automaton diagram shown in Figure 1, pattern preprocessing is carried out as each text symbol is scanned once against the pattern symbols. The text is scanned from the shortest prefix*(λ)* of the pattern through to the longest prefix *(abaa)*. Unless we get to state 4 the pattern is yet to be found. Here, the automaton remains in state 0 which is the empty string(*λ*) prefix of the pattern. At state 0, we input the first symbol of text *a* and it matches pattern prefix of length one *a*, so we can move to state 1. If we put *b* as the input, the automaton would still remain in state 0. At state 1, we input the next text symbol *b* and it matches the pattern prefix of length two *ab* and it moves to state 2 or remains in state 1 if *a* is entered as input. At state 2, *a* which is the next text symbol is entered and it matches the pattern prefix of length three *aba* and it moves to state 3. If *b* is eneterd it moves back to state 0. At state 3, the next text symbol *b* does not match the pattern prefix of length four so the automaton moves back to state 2. At state 2, we enter *b* which is the next symbol of text and it moves to state 0 since it still does not match pattern prefix of length two. Again at state 0, we input *a* and the automaton moves to state 1 and still remains in state 1 since the next text symbol entered is *a*. Now, we enter *b*, *a* and *a* from the text symbols which allows the automaton to transit to state 2, state 3 and state 4 respectively. At state 4 where pattern prefix of length four ***abaa*** has been matched, we can announce match success since the final state has been reached but searching continues until the whole text symbols are exhausted. At state 4 again, if we input *a* the autoamton moves to state 1 and if we input the next symbol *b*, the automaton moves to state 2. This whole process continues until the text is exhaustively searched. Table 2 illustrates the automaton search explained above. A transition table is created for the data as is required in DFA searching algorithm (Table 3).

In Table 2 the underlined symbols in the text column represent text symbols that match pattern prefixes while the italicised symbols indicate the current text symbol that is being compared with the pattern. The bold and underlined symbols at the 11th row in the text column signify where match is successful.

Applying the DFA on the same sample data as the Brute-force algorithm as indicated, there were 12 steps involved with 12 pattern-text comparison.

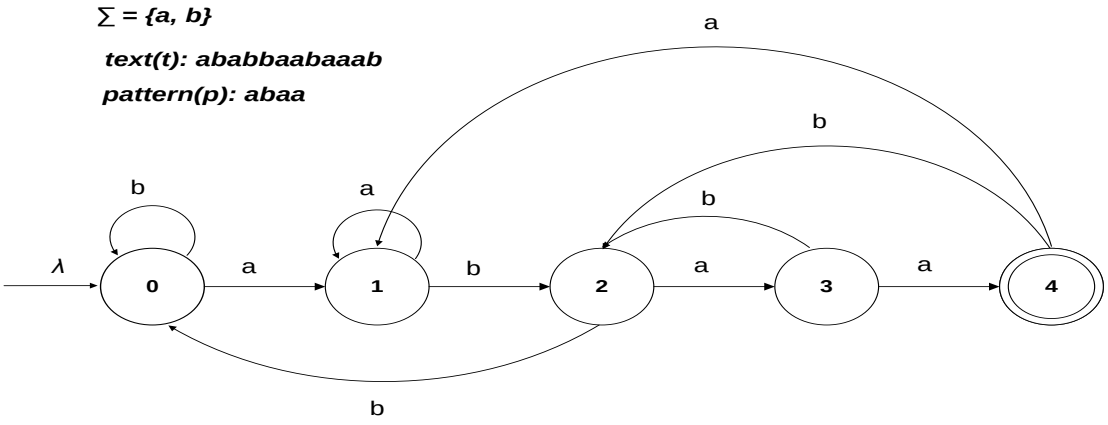

Fig 1: String matching automaton for pattern 'abaa'

Table 2. Tabular representation of DFA search for the pattern

Steps	Text	Transition	Number of pattern-text comparisons
1	*a*babbaabaaab	State 0 → state 1	1
2	a*b*abbaabaaab	State 1 → state 2	1
3	ab*a*bbaabaaab	State 2 → State 3	1
4	aba*b*baabaaab	State 3 → State 2	1
5	abab*b*aabaaab	State 2 → State 0	1
6	ababb*a*abaaab	State 0 → State 1	1
7	ababba*b*baaab	State 1 → State 1	1
8	ababbaa*b*aaab	State 1 → State 2	1
9	ababbaab*a*aab	State 2 → State 3	1
10	ababbaaba*a*ab	State 3 → State 4	1
11	ababba**abaa***a*b	State 4 → State 1	1
12	ababba**abaa***a*b	State 1 → State 2	1

Table 4 clearly shows that the DFA algorithm performs a search with lesser number of pattern-text comparisons.

However, the time required to perform the comparisons actually tell which algorithm performs better. That brings to question the computational complexity of the algorithms.

4.3 The Computational Complexity

4.3.1 Brute-Force Algorithm

From the experiment performed, the brute force approach requires the input text string to be backtracked whenever there is an unsuccessful match with a symbol in the pattern. This problem of backtracking raises the computational time of the algorithm. Backtracking is the phenomenon where text string jumps to the next position whenever a mismatch occurs at the current text position. Whenever the pattern is shifted to the right as a result of a mismatch, it can otherwise be said that it is the text that has been shifted to the left so that the next text symbol can be compared with the pattern. Evidently, the computational complexity of the algorithm is $O(m.n)$ in the worst case. This is a quadratic-order time implying that the brute force algorithm is slow.

4.3.2 DFA Algorithm

On construction of the state diagram (or the state transition table) of the finite automaton of a pattern, we can scan the text to search for the pattern by comparing each text symbol only once not requiring any backtrack when there is a mismatch. Hence we can find all the occurrences of the pattern in the text of length n in $O(n)$ time. This is the significant difference of the DFA string searching method compared to the brute force approach. Although, there is an overhead for preprocessing the pattern which requires $O(m.|\sum|)$ time to:

1. Construct the state diagram or the state transition table for the pattern and
2. Store the table in the memory for pattern matching.

Hence total computational complexity for string matching using the the DFA method becomes $O(n + (m.|E|))$. However, m is usually much smaller compared to n. Therefore for small alphabet $\sum$ the computational complexity, on the average, becomes linear in order.

Table 5 gives a summary of the complexity of both algorithms.

Table 3. State-transition table for the sample data

State \ Transition	a	b
0	1	0
1	1	2
2	3	0
3	4	2
4	1	2

Table 4. Results obtained between Brute force and DFA search

Observations \ Algorithm	Brute-force	DFA
Text	ababbaabaaab	
Pattern	abaa	
Text position of match success	7	
Number of steps employed	9	12
Number of pattern-text comparisons	**19**	**12**

Table 5. Tabular comparison between Brute force and DFA algorithms

Algorithm	Computational complexity	
	Preprocess time	Search time
Brute-force	No preprocessing	O(m.n)
DFA	O(m.|∑|)	O(n)

5. CONCLUSIONS

It has been shown from the experiments in this paper that the brute-force approach to string searching is very slow and inefficient. This approach when employed reduces the searching speed when users make use of text processing applications. The brute-force performs search in *O(m.n)* time which is a great setback in comparison to most efficient algorithms available for accessing information in a quick and efficient manner. It therefore hinders the usability and functionality of applications designed to seek for contents from a larger database. Also, backtracking is one major constraint facing the brute-force algorithm. Although, the brute-force method is still utilized in a case where the length of pattern and text are relatively short and also where the alphabet has relatively less number of distinct elements, it falls flat where the pattern and the text are longer, as in today databases.

A solid conclusion can be made from the analysis in this paper that the most efficient solutions for the string matching problem are based on finite automata. The ability of the finite automata to eliminate the probem of the text backtracking has influenced the development of the first linear-order string searching algorithm (the Knuth-Morris-Pratt algorithm) because it also preprocesses pattern in *O(m)* time and runs a search in *O(n)* time.

The DFA is today used as a tool for string searching because of its efficiency. Among these are the Knuth-Morris-Pratt (mentioned above) and Boyer-Moore algorithms which are based on DFAs.

6. REFERENCES

[1] Ben Wellner (471 13 0453 7) and Michael Dant (390 80 0003 1) The Unix "GREP" utility, "CS520 – Introduction to Formal Models" http://pages.cs.wisc.edu/~mdant/cs520_4.html

[2] J. Kaur, B. Chauhan and J. K. Korepal "Implementation of Query Processor Using Automata and Natural Language Processing" International Journal of Scientific and Research Publications, ISSN: 2250-3153 Volume 3, Issue 5, pp. 1- 5, May 2013.

[3] E. Gribkoff "Applications of Deterministic Finite Automata" ECS 120 UC Davis, Spring 2013.

[4] N. Singla, D. Garg "String Matching Algorithms and their Applicability in various Applications" International Journal of Soft Computing and Engineering (IJSCE) ISSN: 2231-2307, Volume 1, Issue 6, pp. 218 – 222, January 2012.

[5] G. A. Stephen, "String Searching Algorithms" Singapore: World Scientific, Chapter 2, pp. 5-37, 2001.

[6] S. Mitra, T. Acharya "Data Mining: Multmedia, Soft Computting and Bioinformatics" New Jersey: Wiley-Interscience, Chapter 4, pp. 143-169, 2003.

[7] J. D. Ullman, Video Lecture "Informal Introduction to finite automata" Stanford University, May 2012, www.coursera.com

[8] J. E. Hopcroft, R. Motwani and J. D. Ullman "Finite Autmata and Regular Expression" Introduction to automata theory, languages and computation, New York: Addison Wesley, pp. 13-45, 2006.

[9] D. Knuth, J. Morris and V. Pratt "Fast pattern matching in strings" SIAM journal of Computting, Volume 6, pp. 323-350, 1977.

[10] A.V. Aho, M.J. Corasick "Efficient string matching: an aid to bibliographic search" Communications of the ACM, Volume 18, No. 6, pp. 333-340, June 1975.

[11] R. S. Boyer, J. S. Moore "A fast string searching algorithm" Communications of the ACM, Volume 20, No. 10 pp.762-772, October 1977.

[12] R. N. Hoorspool "Practical fast searching in strings" Software – Practice and Experience, Volume 10 No. 6, pp. 501-506, 1980.

[13] R. M. Karp and M. O. Rabin "Efficient randomized pattern matching algorithms" IBM Journal of Research and Development, Volume 31, No 6, pp. 249-260, 1987.

[14] D. M. Sunday "A very fast substring searchalgorithm" Communications of the ACM, Volume 33, No. 8, pp. 132-142, August 1990.

[15] A. V. Aho and J. D. Ullman "Patterns, Automata and Regular expressions" Foundations of Computer Science, New York: W. H. Freeman & Company, pp. 530-571.

[16] M. V. Lawson "Introduction to finite automata; Non-deterministic automata; Kleene's Theorem" Finite Automata, 1 ed. New York: Chapman and Hall/CRC, pp.1-14, pp. 53-60, 2003.