Real time Aho-Corasick Implementation of String Matching technique suitable for smart IOT

Mohammad Equebal Hussain
Department of Computer Science
Suresh Gyan Vihar University
Jaipur, Rajasthan, India
mdequebal.60508@mygyanvihar.com

Rashid Hussain
Professor and HoD
Suresh Gyan Vihar University
Jaipur, Rajasthan, India
rashid.hussain@mygyanvihar.com

Abstract — **With growing technology, high speed internet, ease of access of hand held devices at affordable cost and many more such reasons, data is growing exponentially. Almost all application require cost effective, fast and efficient real time search, modify and replace feature within large volume of raw data, which is nothing but streams of characters. In this article we will propose a model for "Real time Aho-Corasick Implementation of String Matching (RACISM)" technique suitable for low cost internet enabled smart IOT device as an extension to virtual wireless sensor network (vWSN). Aho-Corasick (AC) is one of the best multiple string matching algorithm also suitable for IOT application. AC is having linear running time complexity of O(n) , where 'n' is length of input stream. The beauty of this algorithm is that, once the AC automaton is ready then computation time doesn't depend on the size of test data set, dictionary or pattern. It can match the pattern on the fly within continuous stream of data. The downside of Aho-Corasick algorithm is that it requires more memory to store AC automata as a TRIE data structure, which is an extension of deterministic finite automata (DFA). The focus of this paper is to propose lightweight TRIE based implementation of AC algorithm suitable for IOT devices.**

Keywords— Aho-Corasick, dictionary, pattern, Finite State Machine, Deterministic Finite Automata, String searching, Internet of Things, virtual wireless sensor network

I. INTRODUCTION

In order to provide a smart service or solution requires combination of infrastructure, framework, protocol, intelligent algorithm and continuous innovation. Wireless sensor network (WSN) as IOT when combined with real time processing capabilities add lots of value to take real time decision by solving complex and challenging problems to make this world a better place to live. There is a basic difference between IOT and WSN. The former can directly exchange data over internet using sensors and API but WSN is a group of sensor nodes where individual sensor consists of power module, microcontroller and wireless transceiver. WSN connects to the internet through a gateway. Individual sensor nodes powered by batteries, forms a connected network using star, tree or mesh topology. WSN is a revolutionary information gathering method to detect physical phenomenon useful across all industry including but not limited to health, environment, agriculture, traffic management etc [17].

String matching is one of the basic requirements for solving problems such as text, image, speech, natural language processing (NLP) to search certain relevant information from continuous stream of data produced by smart devices like IOT based wireless sensor network. The result thus produced is consumed by other application for further processing. It is commonly used to filter out bad words, sensitive information like any personally identifiable information (PII) etc in order to achieve data loss prevention (DLP). Hence there is always a requirement for simple, cost effective with improved performance of multiple string matching technique.

Taking the advantage of latest hardware which support large memory, advance computing capabilities, and multicore processors. Such hardware also need efficient algorithm to produce real time results. Although the string search algorithm is not new but research activities has increased significantly in last few years, which indicates that requirement of efficient solution is very high [16]. Many researches been done to make string searching as much fast as possible either using hardware approach like FPGA [4] or software approach using multiprocessor [5] and multithreaded application. Boyer and Moore [2] worst case linear time string searching algorithm by skipping large portion of text during search. Wu and Manber is another string search technique by checking multiple characters simultaneously [3]. Aho-Corasick (AC) multiple string search is the most commonly used algorithm [1]. Since AC algorithm is already implemented using parallel pipelined approach [6] or using multicore cpu [7], our focus is to present a model useful for IOT based virtual wireless sensor network (vWSN) devices by proposing combined approach.

Going forward, in this paper, we will discuss about how Aho-Corasick automata (RACE) model is different from deterministic finite automata, followed by implementation of AC using forward and failure link, preparation of transition table, proposed algorithm, comparison of results and performance evaluation followed by proposed RACE model architecture for an end to end design of IOT model. Finally we conclude the paper.

II. BACKGROUND AND PROBLEM STATEMENT

A Deterministic Finite Automata is a finite state machine defined as 5 tuple represented by a directed graph G(V,E) having set of edges $E \in \sum$, set of vertex $V \in Q$. It always starts from initial state q_0. Based on the input character and current state, DFA can either make a transition to next state or stop if transition is not defined or reach to final state. An AC automaton is little different from regular DFA because it has an option to follow the failure link if next move is not defined for the given alphabet.

Throughout this article, pattern and dictionary will be used interchangeably. Let $P = \{p_1, p_2, . p_n\}$ is set of patterns. S represents input text and |P| represents sum of length of all patterns [1] where $|P| = k = \sum_{i=1..n} k_i$ then problem statement is to find $\forall P_i \in P$ from S.

Pao and Liu [9] characterize AC automata as a state graph G = (Q, E), where E is set of edge defined by the transition function δ, i.e. $E = \{(u, w, v) | u \in Q \wedge w \in \sum \wedge v =$

δ(u, w))} where u, w and v are current state, input character and next state respectively. Initial state q_0 correspond to an empty string ε. It also discuss about various approaches to reduce memory cost in AC implementation using state graph reduction, Lookup table compression and rule set partitioning. Since the AC graph is constructed from pre populated dictionary hence the number of failure link increases as dictionary grows, and the relation between dictionary size and number of failure link is not linear. The majority of memory (≈ 99%) is used to store failure edge in AC state graph.

String matching algorithm can be put into various categories [12]. It is mainly divided into exact and approximate matching which can be further sub divided into software and hardware based exact and multiple matching algorithm. Automata based approach is under software multiple matching category.

FPGA based hardware implementation [13]-[15] of Aho-Corasick multiple string matching algorithm with support of software based state transition lookup table (LUT) doesn’t require hardware reconfiguration when pattern gets changed. Hence same hardware can be used multiple times.

S Faro [16] proposed fast skip-search algorithm which is based on fingerprint technique in which each substring is converted into a numeric value within certain range, which is used for searching.

III. Realtime Aho-Corasick Engine (RACE) model

A. AC Automata

Let M(C) represents the RACE model defined as 6 tuple Finite State Machine (FSM) instead of 5 tuple regular DFA. M(C) = (Q, ∑, q_0, δ, Δ, F) represents Finite State Machine which will be extended to Aho-Corasick automata with few change. Each state will be represented using a circle with label, Final state as labeled double circle, valid transition δ using solid arrow from current state to next state with input character as label, failed transaction Δ as dotted arrow from current state to next state without label as shown in Figure 1. We have used JFLAP software [10] to draw DFA and other automata and draw.io online tool [11] for constructing diagrams and charts throughout this document.

Q = Set of state

∑ = Set of characters, ε represents empty string, $\varepsilon \in \sum^*$

q0 is initial state, q0 ϵ Q

δ represents valid transition function i.e. $Q \times \sum \rightarrow Q$

Δ represents failure transition function i.e. $Q \times \sum^* \rightarrow Q$

F is set of final state. $F \subseteq Q$

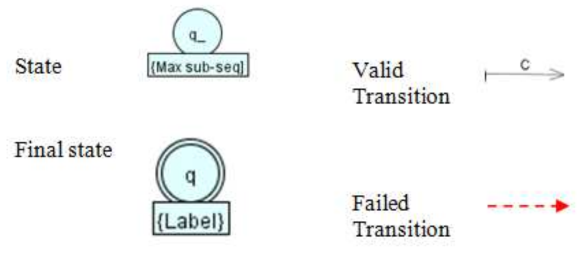

Fig. 1. Diagram to represents RACE model [10]

B. Building labelled AC TRIE

Aho-Corasick is an efficient dictionary matching algorithm which matches each elements of dictionary or set of words within an input stream. The algorithm constructs a TRIE using FSM with extra links called failure links pointing to some node within the TRIE in order to traverse for non matching elements from dictionary. For simplicity let’s assume a dictionary of “m” words each of length “κ” to find from a sequence of input string of length “n”. In order to apply Aho-Corasick algorithm, the first step is to construct AC automata from the dictionary.

Dictionary Đ = {“ACC”, “ATC”, “CAT”, “GCG”}
Input string Ṡ = “GCATCG”

ε Represents no input or empty string

Below is the two step process to construct Aho-Corasick automata M(C) from dictionary Đ.

1) Construct the Basic TRIE ß shown in Figure 2 represented by Table I and
2) Construct failure adjacency node Ậ which is longest suffix for each node represented by Table II.

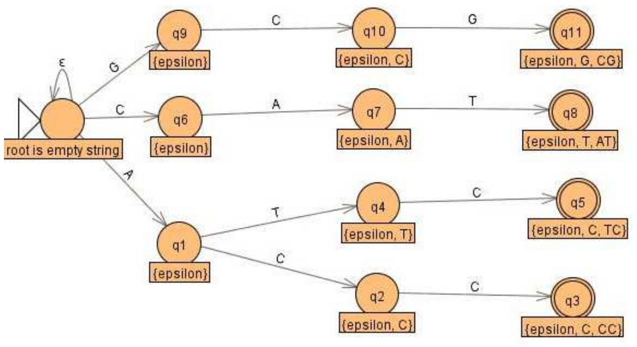

Fig. 2. Aho-Corasick basic TRIE ß for the dictionary Đ [10], [11]

C. Building Aho-Corasick transition table

Two separate AC transition table will be constructed as below.

1) Constructing valid AC transition table δ from given set of dictionary.
2) Constructing failure AC transition table Δ from longest proper suffix when transition is not defined by δ.

TABLE I VALID TRANSITION FOR DICTIONARY Đ

Current State	Input Character	Next State
<Root>	A	$<q_1>$
	C	$<q_6>$
	G	$<q_9>$
$<q_1>$	C	$<q_2>$
	T	$<q_4>$
$<q_6>$	A	$<q_7>$
$<q_9>$	C	$<q_{10}>$
$<q_2>$	C	$<q_3>$ (Final)
$<q_4>$	C	$<q_5>$ (Final)
$<q_7>$	T	$<q_8>$ (Final)
$<q_{10}>$	G	$<q_{11}>$ (Final)

TABLE II AC TRANSITION Δ FOR FAILURE LINKS

Current State	Next State	Longest suffix
<Root>	<Root>	ε
<q_1>	<Root>	ε
<q_2>	<q_6>	C
<q_3>	<q_6>	C
<q_4>	<Root>	ε
<q_5>	<q_6>	C
<q_6>	<Root>	ε
<q_7>	<q_1>	A
<q_8>	<q_4>	AT
<q_9>	<Root>	ε
<q_{10}>	<q_6>	C
<q_{11}>	<q_9>	G

D. Building complete AC TRIE with failure links

Once the transition table is ready, construct final TRIE as below so that input string can be processed in one pass.

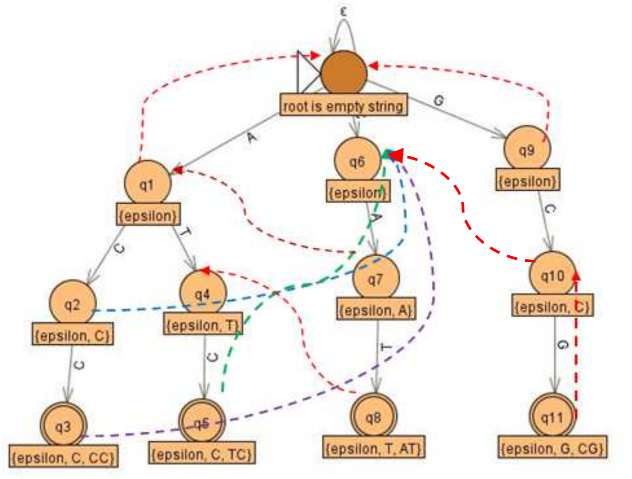

Fig. 3. AC automata representing failure links for dictionary {ACC, ATC, CAT, GCG} [10], [11]

E. Processing input string

Figure 3 represents AC TRIE for dictionary Ḋ containing four patterns "ACC", "ÁTC", "CAT" and "GCG". For each word the final state is represented by state q_3, q_5, q8 and q_{11} respectively using double circle. TABLE I represents the valid transition and Table-1b represents the failure transition. At any point of time for a given input character and present state, valid transition table will be looked up first. If the entry is available then move to the next state else failure transition table will be looked up for the next state for alternate option using back-tracking. During traversal, whenever a final state is reached, it represents a match from dictionary. All the matched string can be put as a list in the output.

TABLE III PROCESSING INPUT STRING "GCATCG"

Current state	Input character	Transition using Table I and Table II			
		Next state	Table	Remarks	Input character consumed
<Root>	G	<q_9>	I	→START	Yes
<q_9>	C	<q_{10}>	I	-	Yes
<q_{10}>	A	<q_6>	II	Failure link	No
<q_6>	A	<q_7>	I	-	Yes
<q_7>	T	<q_8>	I	Final state Match found	Yes
<q_8>	C	<q_4>	II	Failure link	No
<q_4>	C	<q_5>	I	Final state Match found	Yes
<q_5>	G	<q_6>	II	Failure link	No
<q_6>	G	<Root>	II	Failure link	No
<Root>	G	<q_9>	I	STOP	Input exhausted

From above transition two matches were found i.e. output list contains {"CAT", "ATC"}.

IV. IMPLEMENTATION AND ALGORITHM

As an initial implementation of AC TRIE, we used "multifast-v2.0.0" [8]. It provides AC library with header only implementation in C programming language. It accepts two inputs, i.e. an array of finite string called pattern and input string. It returns array of matched results as output. Each entry in the output set also determines position of the found occurrence and pattern that was matched. It also accepts both inputs as files. IN order to achieve case insensitive search, every input string need to be converted to lower case. Version 2.0.0 also support replace functionality. Steps are summarized as below.

Algorithm. Search using multifast AC library API.
Input 1. Pattern or dictionary as array of string or file.
Input 2. File containing string to be searched for pattern.
Output. Matched patterns with position in the input string.
Method.

```
Begin
    Create a new trie - ac_trie_create ()

    do
        Add pattern to automata – ac_trie_add ()
    while (no more patterns)

   /* Finalize automata (no more patterns will be added) */
    Finalize the trie - ac_trie_finalize ()

    Display (optional) – ac_trie_display ()
    Set the input string – ac_trie_settext ()

    do
        Find the match – ac_trie_findnext ()
    while (not end of search)

    Release the automata – ac_trie_release ()
End
```

V. RESULT AND PERFORMANCE EVALUATION

Our result is based on comparison between naïve search methods vs. AC algorithm for 1000000 patterns each of random length of maximum 256 characters long, containing lower and upper case alphabets, numbers and special characters within a given input string. The test ran for 9 iterations. Figure 4 in Table 4 shows the variation in running time for both naïve and AC search.

TABLE IV RUNTIME COMPARISION BETWEEN NAÏVE AND AC SEARCH

Cycle	Running time in millisecond (ms)	
	Naive	*AC*
1	1145	10
2	1151	10
3	1153	11
4	1158	10
5	1174	11
6	1222	16
7	1248	10
8	1261	17
9	1644	16

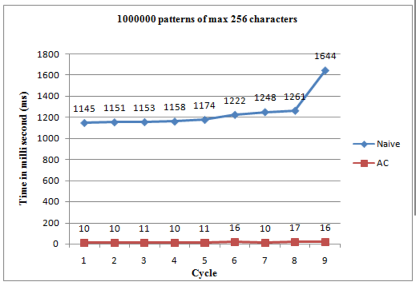

Fig. 4. Run time comparison of naïve and AC search for random pattern size < 256 characters.

We also evaluated the AC algorithm to compare the performance based on three parameters i.e. number of patterns |P|, length of input string |S| and size of individual words in pattern file |k|.

1) Figure 5. Shows the result for |P| = 100000 and 300000 respectively, |S| = 256 and |k| = 8.

2) Figure 6. Shows the result for |P| = 100000, |S| = 256 and |k| = 16.

3) Figure 7. Shows the result for |P| = 300000, |S| = 512 and |k| = 8.

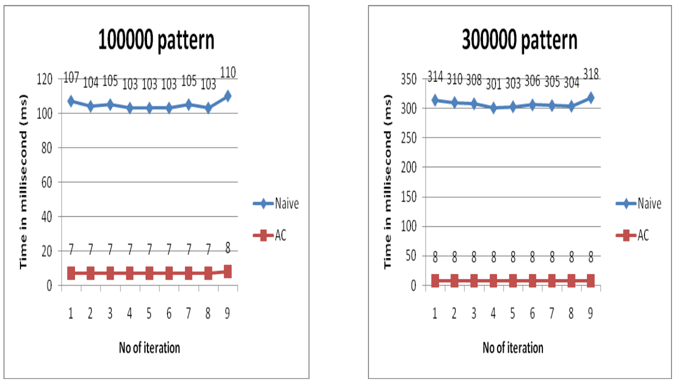

Fig. 5. Result: |P| = 100000 and 300000 respectively, |S| = 256 and |k| = 8

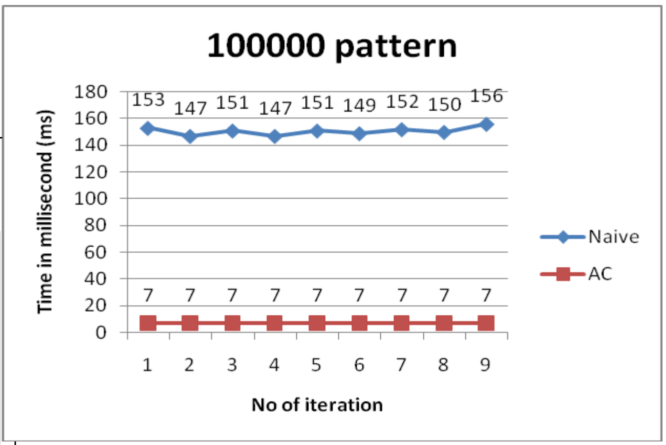

Fig. 6. result for |P| = 100000, |S| = 256 and |k| = 16

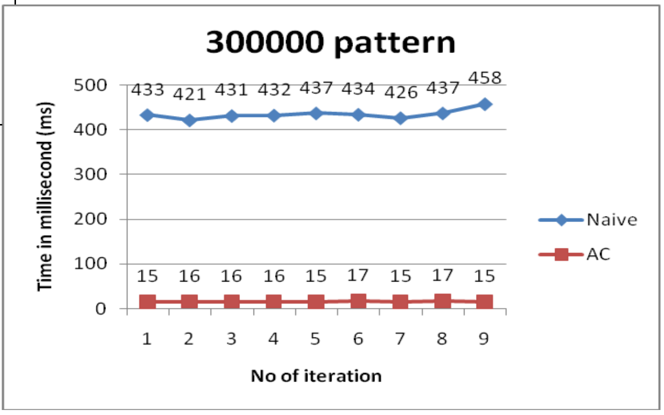

Fig. 7. result for |P| = 300000, |S| = 512 and |k| = 8

Hence from the test result it is proved that AC always performs better compared to naïve method.

VI. PROPOSED DESIGN AND ARCHITECTURE

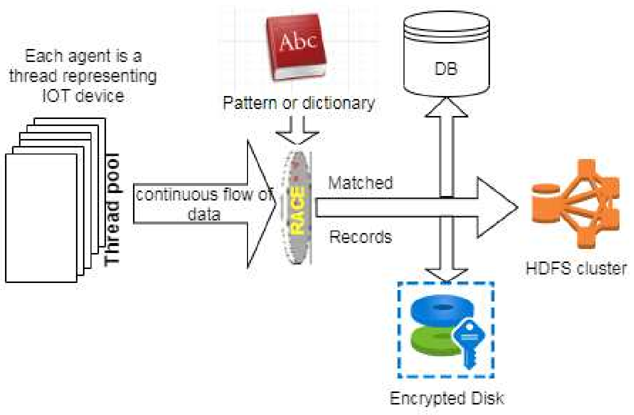

Fig. 8. Real time AC (RACE) model

Major components of proposed design represented in Figure 8.

1) **Thread pool** – it is a pre-configured pool of agents. Each agent is nothing but software representation of physical IOT device. It accepts connection from real device and then sends the data over socket to dedicated RACE engine. The

synchronization between thread will be managed by the thread pool manager. Agent is initialized and resources were allocated at the time of initialization. It is returned back to the pool when complete its job. Agent never destroyed due to an expensive operation.

2) **Real time Aho-Corasick engine (RACE)** - It accepts the stream of characters from agent, and a dictionary to build AC automata, parse it as per AC algorithm, generate matched result from the real time stream generated by IOT agent. The output is then saved into Database, encrypted disk or HDFS cluster for further processing by various application possibly a cloud based DLP.
3) **Radis** - in memory database.
4) **HDFS uploader** - Its role is to upload parsed file or records to Hadoop cluster.
5) **Socket connection** – TCP//IP wherever required
6) **Dockerfile –** It is a text file containing series of instruction to build docker image. Few important commands [17], [18] are FROM, RUN, COPY, ADD, CMD, ENTRYPOINT, WORKDIR, USER, VOLUME build 'race' using below instruction set.

```
  COPY docker/*.sh /usr/local/bin/
  docker build -t="race" . # create docker image
  docker run  -it  race  /bin/bash
  docker      build      -t="race"      --build-arg
DEVEL_TOOLS=1 .
  RUN set -ex \
  && ./configure  --enable-ac \
  && make clean \
  && make
```

VII. CONCLUSION AND FUTURE WORK

Due to anywhere and everywhere high speed internet availability, cloud computing, artificial intelligence based wireless sensor network IOT devices etc are generating huge data in which some of them are sensitive in nature. In order to prevent loss of sensitive information, a real time multiple string search technique is needed. Many research has already been done and in progress but very few of them is focused on IOT and WSN. With the help of performance result it is evident that AC algorithm is simple and much faster compared to normal search technique. In this article we proposed a simple and elegant design and architecture in which RACE is placed as an individual dedicated component. This component could be running either as a process or separate docker pod within virtual wireless sensor network environment. Docker and vWSN are beyond the scope of this paper. A major issue in AC algorithm is memory requirement to maintain failure links but issue already been addressed [9] using pipelined processing. The fundamental property of RACE model is that it stores complete dictionary or pattern before constructing the model. This is done using finalize () method as mentioned in section IV. Addition, deletion or modification of pattern is not allowed once finalize is called. AC is quite interesting algorithm where lots of scope exists for the improvement to make regular expression based search instead of using static dictionary. Since a regular expression can be represented as a finite automata either deterministic or non deterministic (DFA or NDFA). Therefore it is a challenging task to design AC TRIE from regular expression rather than fixed set of patterns. An approach is discussed [19] but at very initial stage. If this is achieved then AC will be directly used to match any pattern. For example let's assume that a sensitive information like credit card number, CVV and expiry date can be represented using a regular expression, then searching for a sensitive information like any credit card detail using Aho-Corasick algorithm will add wings to multi search process. Regular expression along with dictionary is gaining popularity for DLP application. In future study we will explore support for regular expression in RACE model as well as hashing and fingerprint techniques.

ACKNOWLEDGMENT

I would like to thank Dr. Rashid Hussain for his encouragement and support during the research.

REFERENCES

[1] Alfred V. Aho and Margaret J. Corasick, Efficient String Matching: An Aid to Bibliographic Search, Comm. ACM vol. 18, 1975, pp 333-340.

[2] R. S. Boyer and J. S. Moore, "A fast string searching algorithm", Communications of the ACM, vol. 20, no. 10, (1977), pp. 62-72.

[3] Wu, S. and Manber, U. 1992. Fast text searching: Allowing errors. *Commun. ACM 35*, 10, 81–93.

[4] S. M. Vidanagamachchi, S. D. Dewasurendra, R. G. Ragel and M. Niranjan, "Tile optimization for area in FPGA based hardware acceleration of peptide identification," 2011 6th International Conference on Industrial and Information Systems, 2011, pp. 140-145, doi: 10.1109/ICIINFS.2011.6038056.

[5] D. Herath, C. Lakmali and R. Ragel, "Accelerating string matching for bio-computing applications on multi-core CPUs," 2012 IEEE 7th International Conference on Industrial and Information Systems (ICIIS), 2012, pp. 1-6, doi: 10.1109/ICIInfS.2012.6304784.

[6] W. Lin and B. Liu, "Pipelined Parallel AC-Based Approach for Multi-String Matching," 2008 14th IEEE International Conference on Parallel and Distributed Systems, 2008, pp. 665-672, doi: 10.1109/ICPADS.2008.47.

[7] S. Arudchutha, T. Nishanthy and R. G. Ragel, "String matching with multicore CPUs: Performing better with the Aho-Corasick algorithm," 2013 IEEE 8th International Conference on Industrial and Information Systems, 2013, pp. 231-236, doi: 10.1109/ICIInfS.2013.6731987.

[8] Available online at https://sourceforge.net/projects/multifast/files/multifast-v2.0.0/multifast-v2.0.0.tar.gz/

[9] Derek Pao, Wei Lin, and Bin Liu. 2010. A memory-efficient pipelined implementation of the aho-corasick string-matching algorithm. <i>ACM Trans. Archit. Code Optim.</i> 7, 2, Article 10 (September 2010), 27 pages. DOI:https://doi.org/10.1145/1839667.1839672.

[10] Software for drawing DFA and other automata http://www.jflap.org/

[11] Online drawing tool http://draw.io/

[12] S. I. Hakak, A. Kamsin, P. Shivakumara, G. A. Gilkar, W. Z. Khan and M. Imran, "Exact String Matching Algorithms: Survey, Issues, and Future Research Directions," in IEEE Access, vol. 7, pp. 69614-69637, 2019, doi: 10.1109/ACCESS.2019.2914071

[13] J. A. Joseph, K. Reeba, and S. Salivahanan, "Efficient string matching FPGA for speed up network intrusion detection," Appl. Math. Inf. Sci., vol. 12, no. 2, pp. 397-404, Mar. 2018.

[14] M. Aldwairi, Y. Flaifel, and K. Mhaidat, "Efficient WU-Manber pattern matching hardware for intrusion and malware detection," in

Proc. Int. Conf. Elect., Electron., Comput., Commun., Mech. Comput. (EECCMC), Tamil Nadu, India, Jan. 2018, pp. 1-6.

[15] X. Wang and D. Pao, "Memory-Based Architecture for Multicharacter Aho–Corasick String Matching," in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 26, no. 1, pp. 143-154, Jan. 2018, doi: 10.1109/TVLSI.2017.2753843.

[16] Faro, S.. "A Very Fast String Matching Algorithm Based on Condensed Alphabets." *AAIM* (2016).

[17] Hussain M.E., Hussain R. (2021) Bluetooth 5 and Docker Container: Together We Can Move a Step Forward Towards IOT. In: Misra R., Kesswani N., Rajarajan M., Bharadwaj V., Patel A. (eds) Internet of Things and Connected Technologies. ICIoTCT 2020. Advances in Intelligent Systems and Computing, vol 1382. Springer, Cham. https://doi.org/10.1007/978-3-030-76736-5_19

[18] https://docs.docker.com/engine/reference/builder/

[19] G. Bhamare and S. Banait, "Faster Multipattern Matching System on GPU Based on Aho-Corasick Algorithm [J]", *International Journal of Computer Science & Mobile Computing*, vol. 3, no. 7, 2014.