AUTOMATA

PROJECT SUBMITTED IN PARTIAL FULFILMENT OF THE REQUIREMENT FOR

THE **BACHELOR OF SCIENCE DEGREE IN MATHEMATICS**
2017-2020

BY

JOSEPH NAVEEN P.S

Reg No. 170021032414

ASHIQUE K.S

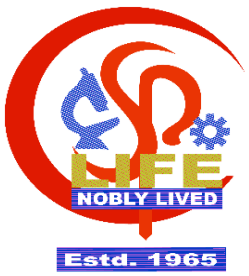
Reg No. 170021032401

SUMITH D.S

Reg No. 170021032437

UNDER THE GUIDANCE OF

Dr. MANJU K MENON



DEPARTMENT OF MATHEMATICS

ST. PAUL'S COLLEGE, KALAMASSERY

(AFFILIATED TO M.G. UNIVERSITY, KOTTAYAM)

2017-2020

CERTIFICATE

This is to certify that the project report entitled "AUTOMATA" is a bonafide record of studies undertaken by JOSEPH NAVEEN P.S (Reg no. 170021032414) , ASHIQUE K.S (Reg no. 170021032401) , SUMITH D.S (Reg no. 170021032437) in partial fulfilment of the requirements for the award of B.Sc. Degree in Mathematics at Department of Mathematics, St. Paul's College, Kalamassery, during the academic year 2017-2020.

Dr MANJU K MENON	**Dr. SAVITHA K.S**
Project supervisor	**Head of the Department**
Assistant Professor	**Assistant Professor**
Department of Mathematics	**Department of Mathematics**

Examiner

DECLARATION

We, JOSEPH NAVEEN P.S (Reg no. 170021032414) , ASHIQUE K.S (Reg no. 170021032401) , SUMITH D.S (Reg no. 170021032437) hereby declare that this project entitled "AUTOMATA" submitted to Department of Mathematics of St. Paul's college, Kalamassery in partial requirement for the award of B.Sc Degree in Mathematics, is a work done by us under the guidance and supervision of Dr MANJU K MENON**,** Department of Mathematics, St. Paul's college, Kalamassery during the academic year 2017-2020

We also declare that this project has not been previously presented for the award of any other degree, diploma, fellowship, etc.

KALAMASSERY

JOSEPH NAVEEN P.S

ASHIQUE K.S

SUMITH D.S

ACKNOWLEDGEMENT

We express our heartfelt gratitude to our project supervisor Dr MANJU K MENON, Department of Mathematics, for providing us necessary stimulus for the preparation of this project.

We would like to acknowledge our deep sense of gratitude to Dr. SAVITHA K.S, Head of the Department of Mathematics and all other teachers of the department and classmates for their help at all stages.

We also express our sincere gratitude to Ms. VALENTINE D'CRUZ, Principal, St. Paul's College, Kalamassery for the support and inspiration rendered to us in this project report.

KALAMASSERY

JOSEPH NAVEEN P.S

ASHIQUE K.S

SUMITH D.S

CONTENT

INTRODUCTION

We shall start this unit with the introduction of basic concepts and terms required to study the theory of automata and formal languages.We shall study what is language and basic operations to manipulate strings of a language.Then we shall see the classification of formal languages made by Chomsky and relationship among them.There are different mechanism to express formal languages,we shall start with regular expression which have been widely used in many real life applications such as Unix.We shall also learn how we can Next we shall proceed to graphical representation,of the languages.deterministic finite automata and non-deterministic finite automata are two popular and simple way to represent languages pictorially.These two devices beautifully explain the way the words of a languages are generated.Then we shall study the grammar behind the generation of words in regular language.These regular languages have many interesting properties such as closure properties and decision properties.We shall have in-depth exploration of these properties.After having introduction of all these different mechanism to express regular languages,we shall see that all of these representations are equivalent to each other and we shall establish this equivalence.At the end of this unit,we shall reach at the conclusion that regular languages are not everything in theory of computer science.There are many languages which are not regular.We shall conclude the unit with the study of two equivalent but different model of computation called Moore model and mealy model.

CHAPTER 1

INTRODUCTION TO FINITE AUTOMATA

In this chapter we are going to study a class of abstract machines called finite automata. In the theory of computation, a finite automata is an abstract machine that has only a finite, constant amount of memory. The internal states of the machine carry no further structure. This kind of model is very widely used in the study of computation and languages. Finite automata are computing devices that accept regular languages and are used to model operations of many systems we find in practice.

Let us consider the operation of an automated teller machine (ATM), a popular machine, used for the transaction of the money. Initially our machine is waiting for a customer to come. It is in waiting-for-customer state. When you put the ATM card in machine, the machine changes its state and asks for the pin code, (pc) that is, machine is in another state say main window state. From there you have many options to choose from. We assume only three options are available namely balance enquiry, withdrawal and deposit.

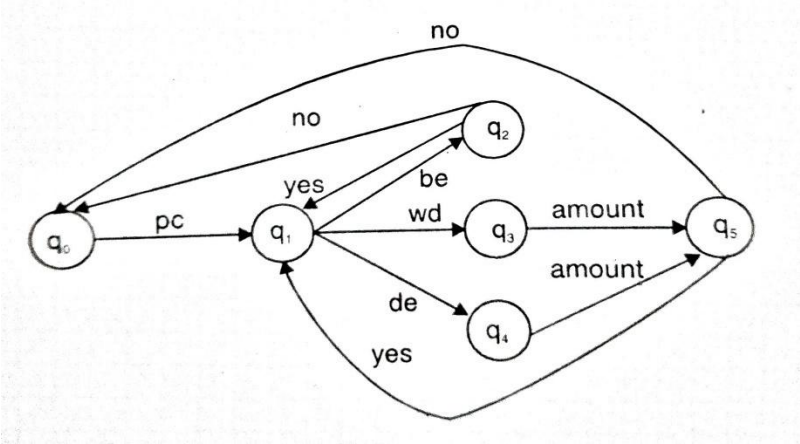
If you press balance enquiry button a form will be displayed where your balance will be displayed. Then you will be asked whether you want any other transaction. If you press "yes" then you will return to main window otherwise machine will return to waiting-for-customer state.

If you press withdrawal (wd) then a form will be displayed where you will be asked for the amount to be withdrawn. After typing amount you may be asked whether you want any other transaction. If you press yes then you will return to main window otherwise machine will return to waiting-for-customer state.

If you press deposit (de) then another form will be displayed where you will be asked for amount to be deposited. After typing amount you

may be asked whether you want a receipt or not. If you press yes then you will get a receipt (not shown in diagram), otherwise you will asked whether you want any other transaction. If you press yes then you will return to main window otherwise machine will return to waiting-for-customer state.

The states and the transitions between states for this automated teller machine can be represented with a diagram. In this figure 1.1.1, circles represent states and arrows represent state transitions.



Symbols on arrows represent input from user

q_0 waiting_for_customer state
q_1 main_window state
q_2 balance_display state
q_3 withdrawal state
q_4 deposite_amount state
q_5 any_other_transaction

FIGURE 1.1.1 :AN ATM REPRESENTED THROUGH FINITE AUTOMATA

In this example automated teller machine have gone through (transitions between) a number of states responding to the inputs from the customer. An automated teller machine looked at this way is an example of finite automaton.

- **DEFINITION OF DETERMINISTIC FINITE AUTOMATA**

Here we are going to formally define finite automata, in particular deterministic finite automata and see some examples. Finite automata recognize regular languages and, conversely, any language that is recognized by a finite automaton is regular. There are other types of finite automata such as nondeterministic automata and nondeterministic automata withε-transitions and they will be studied in later chapters.

Let us now formally define deterministic finite automaton. Let

Q be a finite set of states,

Σ be a finite set of symbols,

δ be a function from Q. × Σ to Q. δ is called transition function. It takes one state and one alphabet symbol as arguments and produces another state.

q0 be a state in Q. It is called the initial state.

F be a subset of Q. F is the set of accepting states (or also called final states).

Then a deterministic finite automaton is a 5-tuple < Q, Σ, δ, q0, F >.

The term `deterministic' refers to the fact that on each input there is one and only one state to which the automaton can transition from its current state. Deterministic finite automaton is generally abbreviated as DFA.

Before we proceed to designing of DFA let us first understand, what does the phrase "acceptance of a string " and "rejection of a string " mean in context of DFA. When we say that a string is accepted by a DFA, it means that if DFA begins from its initial state, proceeds processing the string by applying transition functions, one symbol at a time, and finally when all symbols of the strings get processed, DFA is in one of the final states. Otherwise it is considered rejected.

DFA's can be represented in three different ways. The first method to represent the DFA is to enumerate all the transition functions required to describe the automata. This is the basis for all other methods. We will explain this method through an example.

Example 1.1

Suppose we want to build a finite automata for the language of strings that contains only one string {a}. DFA for this language is

$$A= (Q, \Sigma, \delta, q0, F)$$

$$Q = \{q_0, q_1, q_2\}$$
$$\Sigma = \{a,b\}$$
$$F = \{q_1\}$$

and δ is defined as follow :

$$\delta\ (q_0, a) = q_1$$
$$\delta\ (q_0, b) = q_2$$
$$\delta\ (q_1, a) = q_2$$
$$\delta\ (q_1, b) = q_2$$
$$\delta\ (q_2, a) = q_2$$
$$\delta\ (q_2, b) = q_2$$

Where

Finite automata can be represented in tabular form also. In a transition table rows represent transition from a state for different input symbols. Columns represent transition for a particular symbol. Entry in the ith row and j th column represents the state where the automata goes from state qi on getting symbol aj. We have given below tabular representation of the DFA in example 1.1. State preceded with an arrow is initial state and that with *is final state.

State (q)	Input(a)	Next State (δ (q, a))
$\rightarrow q_0$	a	q_1
q_0	b	q_2
$*q_1$	a	q_2
q_1	b	q_2
q_2	a	q_2
q_2	b	q_2

TABLE 1.1

DFAs are often represented by digraphs called **state transition diagrams** or simply **transition diagrams.** The vertices (denoted by single or available circles) of a transition diagram represent the states of the DFA and the arcs labeled with an input symbol correspond to the transitions. An arc (p, q) from vertex p to vertex q with label **a** represents the transition δ (p, a) = q. Double circles indicate the accepting states. An arrow precedes start state. We will use transition diagrams throughout the book as they give more clear understanding of the language compared to transition functions and transition tables.

A state transition diagram for DFA in example 1.1 is given below in figure 1.2.

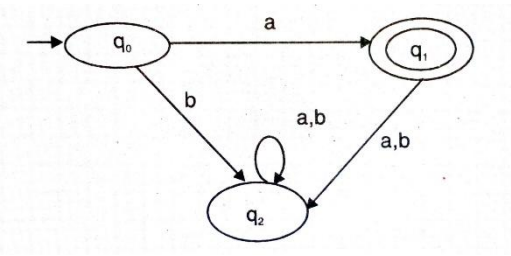


FIGURE 1.2

EXAMPLE 1.2

DFA for strings consisting of only a's.

Q = {q0, q1}, Σ = {a, b}, F ={q0}, the initial state is q0 and δ is as shown in the following table.

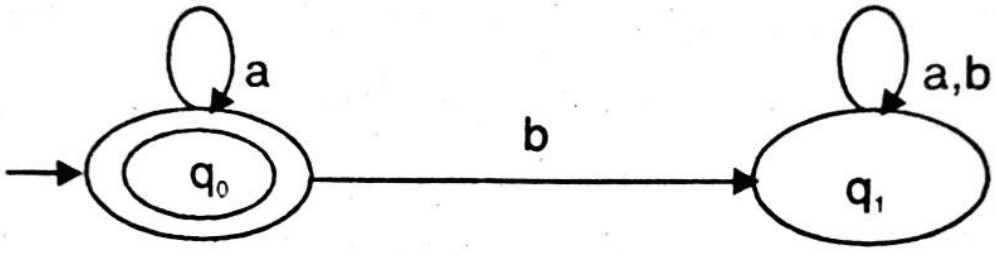
State (q)	Input(a)	Next State (δ (q, a))
→ * q_0	a	q_0
q_0	b	q_1
q_1	a	q_1
q_1	b	q_1

TABLE 1.2

A state transition diagram for this DFA is given below.

FIGURE 1.3

- **OPERATION OF FINITE AUTOMATA**



Let us see how an automaton operates when it is given some inputs. As an example let us consider the DFA of example 1.2. Initially it is in state q0. When zero or more a's are given as an input to it, it stays in state q0 while it reads all the a's. Since the state q0 is also the accepting state, when all the a's are read, the DFA is in the accepting state. Thus this automaton accepts any string of a's. If b is read while it is in state q0 (initially or after reading some a's), it moves to state q1. Once it gets to state q1, then no matter what symbol is read, this DFA never leaves state q1. Hence when b appears anywhere in the input, it goes into state q1 and the input string is not accepted by the DFA. For example strings aaa, aaaaaa, etc are accepted.

CHAPTER 2

NONDETERMINISTIC FINITE AUTOMATA

A Nondeterministic Finite Automaton (NFA) is a finite state machine where for each pair of state and input symbol there may be more than one possible nest States. In general, NFAs contain less number of states compared to DFAs and are simpler to design. NFAs are useful in designing applications based on text search.

- FORMAL DEFINITION

An NFA is a 5 – tuple $(Q, \sum, \delta, q_0, F)$, consisting of a finite set of symbols of states (Q)

a finite set called the alphabet ($\sum$),

a transition function ($\delta: Q \times \sum \rightarrow P(Q)$),

a start($q_0 \in Q$), and

a set of accepting states (F)

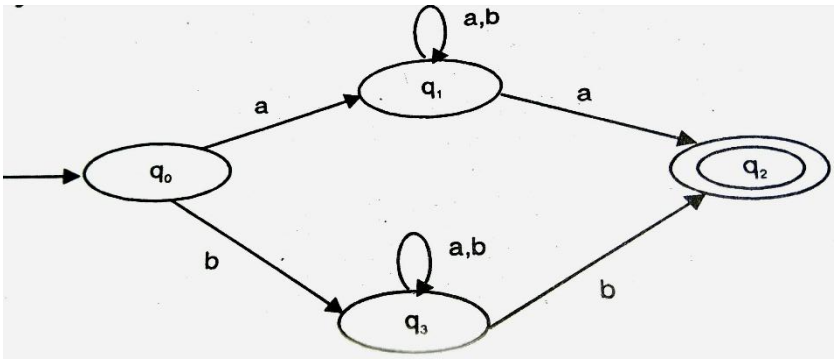
Where P(Q) is the power set of S.

As clear from the definition of NFA, output of $\delta(q , a)$ is a set of states whose cardinality may vary from 1 to n is the number of states in NFA.

EXAMPLE 2.1

Design an NFA for language of strings that start and end with same letter.

Solution :



We assume that all strings are of length greater than or equal to two. Our NFA must remember that what was the first symbol it read and accordingly there are two different branches in automata, one each for a and b. Once it has remembered first symbol (say a), and reached in intermediate state (say q1), other symbols (in this case b) read are insignificant for changing state. It uses the nondeterminism to reach the final state upon reading the same symbol. The final transition diagram has been given in figure 2.1

NFA for example 2.1

- NFA WITH ε – TRANSITIONS

NFA with ε – transitions (also called ε – NFA) are same as NFA with one exception: the transition function must include information about transitions on ε.

Formally,

A ε – NFA 5 – tuple, $(Q, \sum, \delta, q_0, F)$, consisting pf

a finite set of states (Q),

a finite set called the alphabet ($\sum$),

a transition function ($\delta: Q \times (\sum U \varepsilon) \rightarrow P(Q))$,

a start state ($q_0 \in Q$), and

a set accepting states (F) .

Where P(Q) is the power set of S and ε is the empty string.

EXAMPLE 2.2

The following ε – NFA M, with a binary alphabet, determines if the

	0	1	ε
q_0	ϕ	ϕ	$\{q_1, q_3\}$
q_1	$\{q_2\}$	$\{q_1\}$	ϕ
q_2	$\{q_1\}$	$\{ q_2 \}$	ϕ
q_3	$\{q_3\}$	$\{ q_4\}$	ϕ
q_4	$\{ q_4\}$	$\{q_3\}$	ϕ

input contains an even number of 0s or an even number of 1s.

M = (Q, ∑, δ, q_0, F) where

Q = {q0,q1,q2,q3,q4},

ε= {0,1}

q0 = q0,

F = { q1,q3} and

The transition function δcan be defined by state transition table shown in table,

The transition diagram for M is :

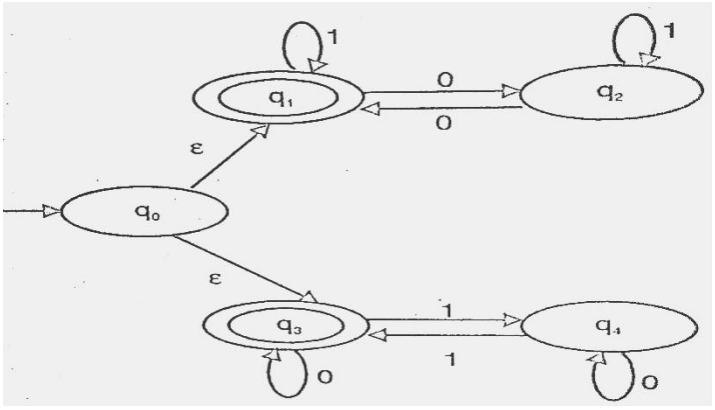


Figure 2.2

M can be viewed as the union of two DFAs: one with staes { q1,q2 } and the other with states {q3,q4}. The first counts whether number of 0's is even and second counts for number of is to be even.

- EPSILON_CLOSURE

In ε -NFA representation of a regular language, starting from one state, machine may go to many other states without getting any single input symbol. To find these states we use the concept of ε -close of state q is a set, member of which can be computed by following the ε -arcs and including the states reached thus so far. We can recursively compute ε -close(q) as follow

Basis Clause : q is in ε -close (q)

Inductive Clause : If state P is ε –close(q) and there ε -arc from state P to state r, than r is also in ε -close(q).

CHAPTER 3

EQUIVALENCE OF FINITE AUTOMATA

We have studied five different method to describe regular languages. In this chapter we shall prove all of them, though may appear different in appearance, are equivalent. We shall start with the conversion $\varepsilon - NFAs$ to equivalent, then NFAs to DFAs so that nondeterminisms are removed and computer programs can be written for simulating them. Then we shall also establish equivalence between regular expression and $\varepsilon - NFA$ and between NFA to regular expression in the form of Kleen's theorem. Finally we are going to learn that the DFA that recognize a regular language can be transformed into a unique DFA by minimizing by number of their states. After we finish these conversions, we shall be in a position to claim that all forms of these machines describing regular languages are in fact equivalent as suggested in figure 3.1.1

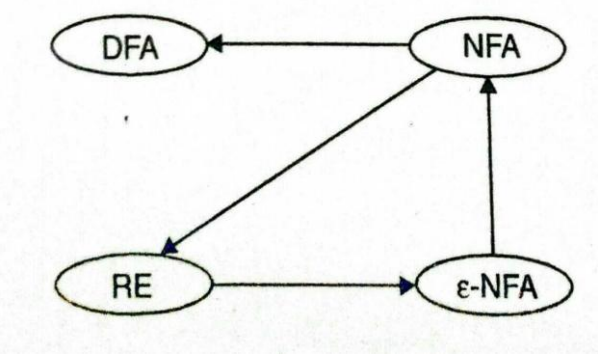


FIGURE 3.1.1 : EQUIVALENCE BETWEEN FA'S AND REGULAR EXPRESSIONS

KLEENE'S THEOREM

Kleene's theorem states that any regular language is accepted by an FA and conversely that any language accepted by an FA is regular.

Theorem : Any regular language is accepted by a finite automaton

The proof below also provides a mechanism to convert regular expression into a $\varepsilon - NFA$

Proof : This is going to be proven by induction following the recursive definition of regular language

Basic Step : As shown below the languages $\emptyset, \{\varepsilon\}$ and $\{a\}$ for any symbol a in Σ are accepted by an FA.

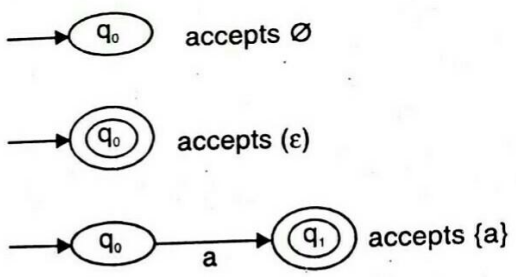


FIGURE 3.1.2 : FA'S ACCEPTING LANGUAGES $\emptyset$, ε AND a

Inductive Step : We are going to show that for any languages $L_1 and\ L_2$ if they are accepted by FA's, then $L_1 \cup L_2$, $L_1\ L_2$, are accepted by FAs. Since any regular language is obtained from $\{\varepsilon\}$ and $\{a\}$ for any symbol a in Σ by using union, concatenation and Kleene star operations, that together with the basic step would prove the theorem.

Suppose that $L_1 and\ L_2$ are accepted by FAs $M_1 = (Q_1, \Sigma, \delta_1, q_{10}, F_1)$ and $M_2 = (Q_2, \Sigma, \delta_2, q_{20}, F_2)$, respectively. We can without

loss of generality ,assume that $Q_1 \cap Q_2 = \emptyset$ since states can be renamed if necessary.

Then $L_1 \cup L_2$, , L_1 L_2, and L_1^* are accepted by FAs $M_u = (Q_u, \Sigma, \delta_u, q_{u0}, F_u)$, $M_c = (Q_c, \Sigma, \delta_c, q_{c0}, F_c)$ and $M_k = (Q_k, \Sigma, \delta_k, q_{k0}, F_k)$, respectively. We are describing each construction one by one.

Construction of M_u : Union of two finite automata M_1 *and* M_2 will be described by

$$M_u = (Q_u, \Sigma, \delta_u, q_{u0}, F_u)$$

Where

$$Q_u = Q_1 \cup Q_2 \cup \{q_{u0}\},$$

Here q_{u0} is a state which is neither in Q_1 nor in Q_2.

$$\delta_u = \delta_1 \cup \delta_2 \cup \{(q_{u0}, \varepsilon) = \{ q_{10}, q_{20} \}\}.$$

Note that $\delta_u(q_{u0}, a) = \emptyset$ for all a in Σ.

Construction of M_u has been illustrated in the figure 3.1.3

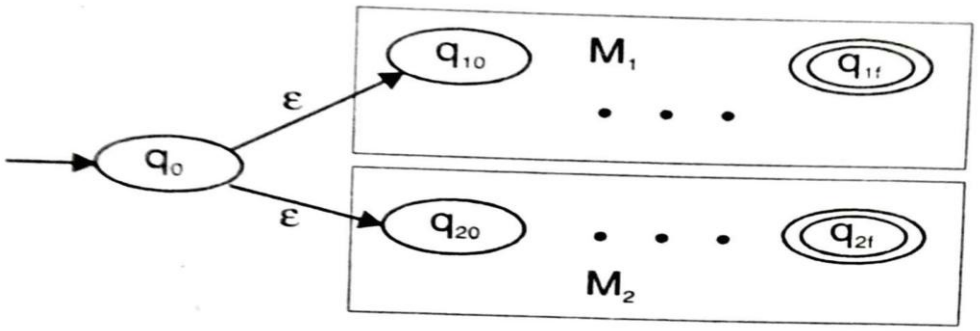


FIGURE 3.1.3 : NFA FOR M_u

Construction of M_c: Construction of two finite automata M_1 and M_2 will be described by

$$M_c = (Q_c, \Sigma, \delta_{c0}, F_c)$$

Where

$$Q_c = Q_1 \cup Q_2 ,$$

$$\mathrm{q_{c0}} = q_{10},$$

$$\delta_c = \delta_1 \cup \delta_2 \cup \{(q, \varepsilon, \{q_{20}\}) | q \epsilon F_1 |\}, \text{ and}$$

$$F_c = F_2$$

Construction of M_c has been illustrated in the figure 3.1.4

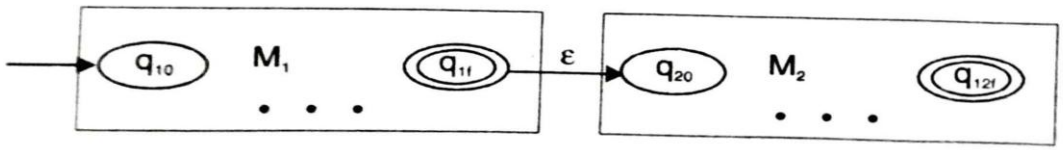


FIGURE 3.1.4 : CONCATENTION OF TWO FA'S M_1 AND M_2

Construction of $\mathbf{M_k}$: Kleen's closure of a finite automata $M_1 w$ill be described by

$$M_k = (Q_k, \Sigma, \delta_k, \mathrm{q_{k0}}, F_k)$$

Where

$$Q_k = Q_1 \cup \{\mathrm{q_{k0}}\},$$

Here $\mathrm{q_{k0}}$ is a state which is not in Q_1.

$\delta_k = \delta_1 \cup \{(\mathrm{q_{k0}}, \varepsilon) = \{q_{10}, q_{kf}\}\} \cup \{(q, \varepsilon) = q_{10} | q_{10} \epsilon F_1 |\}$,

$F_k = \{q_{kf}\}$.

Construction of M_k *has been* illustrated in the figure 3.1.5

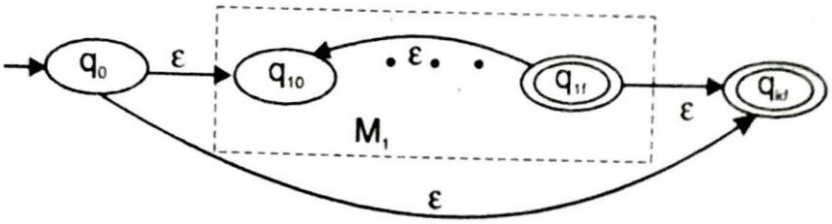


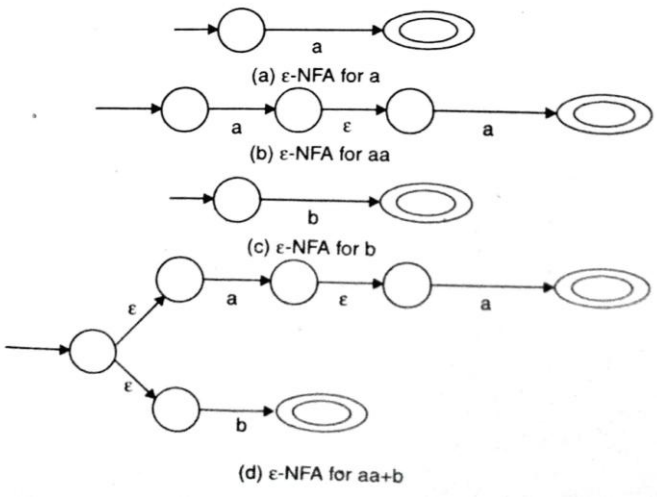
FIGURE 3.1.5 : KLEEN'S CLOSURE OF FA

It can been proven , though we omit proofs , that these $\varepsilon - NFAs,\ M_u,$ M_c and M_k, in fact accept $L_1 \cup L_2, , L_1\ L_2$, and $L_1^*, respectively.$

Let us illustrate the proof of part 1 of kleen's theorem by an example.

Example :

An $\varepsilon - NFAs$ that accepts the language represented by the regular expression (aa+b)* can be constructed as follows using the operations given above



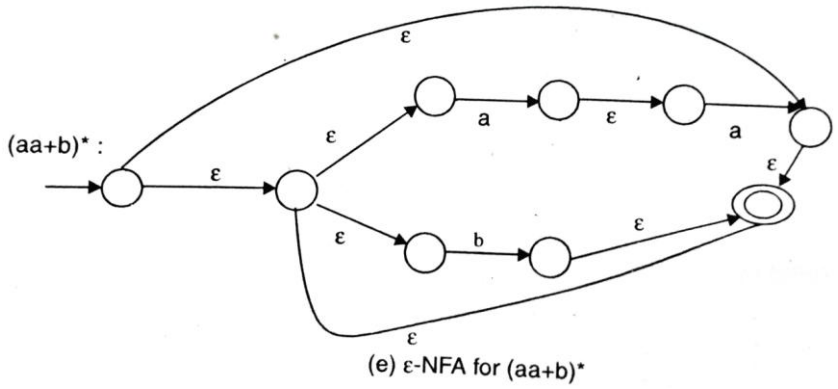


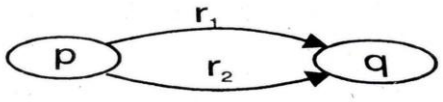
FIGURE 3.1.6 : CONSTRUCTION OF ε – NFAs (aa+b)*

CONVERSION FROM DFA TO REGULAR EXPRESSION

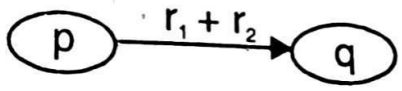
We know that both DFA and regular expressions are used for describing a regular language. Both are just different ways to describe the same language and are in fact equivalent. We assume that DFA has been presented using transition diagrams. This method is known as **state elimination method.**

Basic idea behind the state elimination method is to convert some part of DFA into regular expression using one of the rules mentioned below. Then we eliminate edges and state of the transition diagram using the rules iteratively. Finally, we shall have a transition diagram with to states connected by an edge.

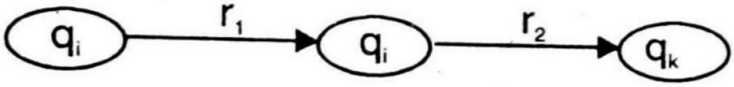
RULE 1: Suppose two states are connected by more than one edge going in the same direction like



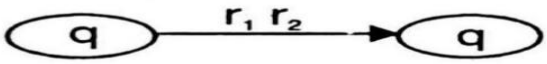
When $r_1 and\ r_2$ are each regular expressions. We can replace this with a single edge that is labeled with a regular expression :



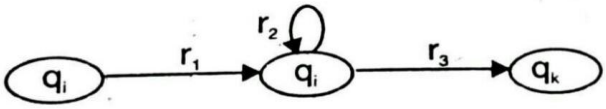
RULE 2a : If we have three states in a row connected by edges labeled With regular expression, we can eliminate the middle state by concatenating the two regular expressions and going to third state directly. Thus the DFA below



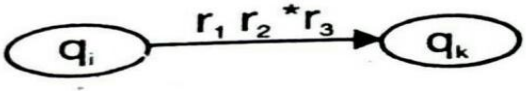
Is equivalent to



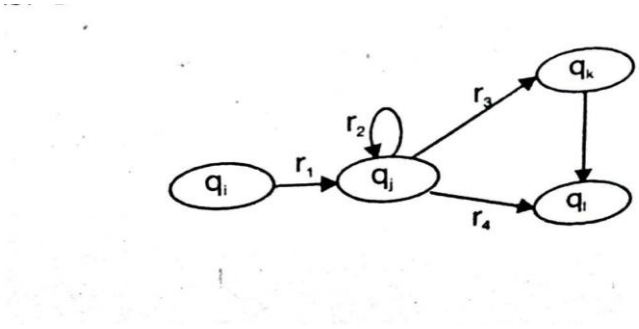
RULE 2b: If there is self loop on q_J as in the example below



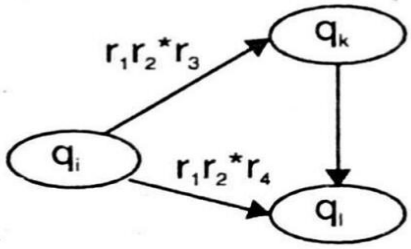
Then equivalent DFA will be



RULE 2c: Before complete eliminating the middle state**,** make sure that labels on all outgoing edges from the state have been changed into regular expressions. Consider the transition diagram below**.**



It can be redrawn as



Let us collect all the pieces together and illustrate the above rules with an example

Example:

Let us construct a regular expression for the language accepted by DFA in the figure 3.2.1

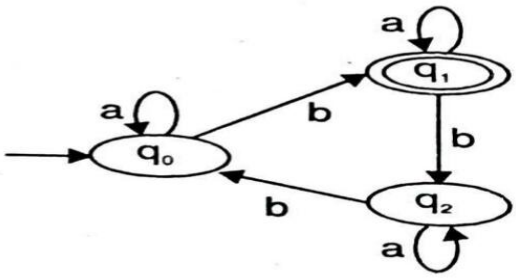
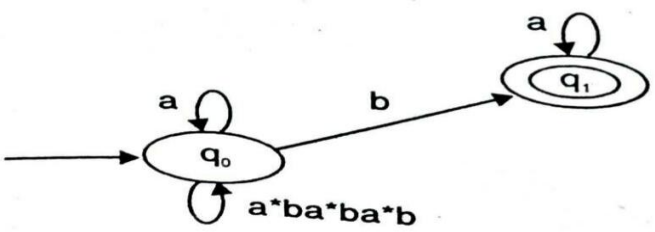


FIGURE 3.2.1: DFA FOR THE EXAMPLE

We can redraw the figure as



From this diagram it is obvious that equivalent regular expression is (a+a*ba*ba*b)* ba *

EQUIVALENCE OF TWO DFAS

As discussed earlier, there may be more than one DFA for the same regular language. Then one may wonder how to decide that whether the two DFA are equivalent. Luckily we have a very easy method to decide this problem

Two DFA are equivalent if for all strings both of them either accept the string or reject it. It implies that for same string both of automata either reached into final state or both fail to do so. Does it give a clue? Yeah, two automata re equivalent if their start states are equivalent. So we have following algorithm to decide the equivalence of two DFAs.

Step 1: Visualize the two DFAs as combined one.

Step 2: Find equivalent class of states**.**

Step 3: If start sate of both automata fall into same class then both are equivalent or not

Example:

Consider two DFAs

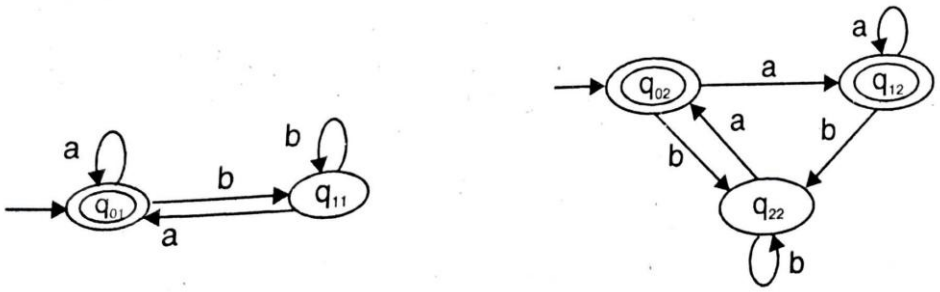


Figure 3.3.1

If we visualize the two automata as one, we find the following two equivalent classes of state: $\{q_{01}, q_{02}, q_{12}\}$ and $\{ q_{11}, q_{22}\}$.

Since start state of both DFAs fall into same classes, these two automata are required

CONCLUSION

In this thesis novel concepts of normal automata, fan automata And rewriting cyclic normal automata are introduced. RCNA has been Used to implement signal processing operations with Markov normal Algorithms. An attempt was made to perform signal processing Operations like arithmetic operations, shift operations, reverse operation Using normal algorithm and RCNA. Processing signals using numeric computers is possible because Algorithms are written for various operations, hence those algorithms Can Be converted into programs using any one of the high level programming Language. Symbolic computation can be realized with numeric Computers, so users will get benefits of both computations. Markov normal algorithms are best suitable for writing algorithms For signal operations with productions and rules. Examples which are Given in the thesis will help to write new algorithm for various Operations. Users can apply normal algorithms in the place context free Grammar, context sensitive grammar, unrestricted grammar and regular Grammar. The form of writing rules and derivation of strings belong to Languages in normal algorithms are quite similar to grammars in automata theory and compiler design, hence researchers those who are Familiar with those grammars can easily learn and apply normal Algorithms which are more powerful tool to design any new languages Also. The operations which are designed using normal algorithms and RCNA can be implemented as software. So we can eradicate the Problem of non availability of software to perform advanced signal Processing operations. Other operations of signal processing also Accomplished in the same way. Signals are represented as strings before They processed by normal algorithms. In any programming language we Have data type character or string hence

having signal as input also no Problem at all. Strings can be stored in a memory for further processing So we indirectly achieved modelling signals in computer process able Format. Most of the programming languages or software has number of String processing operations which can be applied to signals because Signals are in the form of strings. Time complexity of algorithm is very Important factor when it‘s implemented as a program. Even though time Complexity is not as expected but it is at infant stage so we can Compromise on performance later efforts can be made to improve it. Those who have knowledge in concepts like finite state automata, Transition diagrams, grammars and derivations can easily follow, learn and use novel concepts normal automata, fan automata and rewriting cyclic automata and apply them to solve problems in any field wherever automata theory is applicable.

APPLICATION FOR FINITE AUTOMATA

Finite Automata (FA) – For the designing of lexical analysis of a compiler. For recognizing the pattern using regular expressions.

Push Down Automata (PDA) – For designing the parsing phase of a compiler (Syntax Analysis).

Linear Bounded Automata (LBA) – For implementation of genetic programming.

Turing Machine

REFERNECE:

FORMAL LANGUAGES AUTOM ATA THEORY