UNION-FREE DECOMPOSITION OF REGULAR LANGUAGES

Thesis submitted in partial fulfillment of the requirements for the award of degree of

Master of Engineering
in
Computer Science and Engineering

Submitted By
Sukhpal Singh Ghuman
Roll No: 801032027

Under the supervision of:
Mr. Ajay Kumar Loura
Assistant Professor

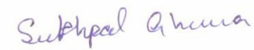
COMPUTER SCIENCE AND ENGINEERING DEPARTMENT
THAPAR UNIVERSITY
PATIALA – 147004

June 2012

Certificate

I hereby certify that the work which is being presented in the thesis entitled, "*Union-free decomposition of regular languages*", in partial fulfillment of the requirements for the award of degree of Master of Engineering in *Computer Science and Engineering* submitted in Computer Science and Engineering Department of Thapar University, Patiala, is an authentic record of my own work carried out under the supervision of *Ajay Kuman Loura* and refers other researcher's work which are duly listed in the reference section.

The matter presented in the thesis has not been submitted for award of any other degree of this or any other University.



Signature:

(Sukhpal Singh Ghuman)

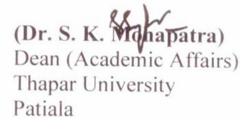
This is to certify that the above statement made by the candidate is correct and true to the best of my knowledge.

(Mr. Ajay Kumar Loura)

Assistant Professor
Computer Science and
Engineering Department
Thapar University
Patiala

Countersigned by

(Dr. Maninder Singh)
Head
Computer Science and Engineering Department
Thapar University
Patiala



(Dr. S. K. Mohapatra)
Dean (Academic Affairs)
Thapar University
Patiala

Acknowledgement

No volume of words is enough to express my gratitude towards my guide, **Mr. Ajay Kumar Loura** Assistant. Professor, Computer Science and Engineering Department, Thapar University, who has been very concerned and has aided for all the material essential for the preparation of this thesis report. He has helped me to explore this vast topic in an organized manner and provided me with all the ideas on how to work towards a research-oriented venture.

I am also grateful to **Dr. Maninder Singh**, Head of Department, CSED and **Mr. Karun Verma**, P.G. Coordinator for the motivation and inspiration that helped me for the thesis work.

I would also like to thank the staff members and my class mates who were always there in the need of the hour and provided with all the help and facilities, which I required, for the completion of my thesis.

Most importantly, I would like to thank my parents and the Almighty for showing me the right direction out of the blue, to help me stay calm in the oddest of the times and keep moving even at times when there was no hope.

Sukhpal Singh Ghuman
(801032027)

Abstract

Regular expressions are well known in the field of computer science. They are commonly used and well-applicable in theory as well as in practice. The regular expressions are used in field of compilers, programming languages, pattern recognition, protocol conformance testing etc.

There exists an algorithm for union-free decomposition of regular language. The algorithm uses a set of all maximal finite concatenations of languages. To construct a union-free decomposition, the algorithm examines all the subsets of maximal finite concatenations of languages and chooses the subset containing a minimum number of languages. By decomposing the regular language into an equivalent union-free regular language also helps in determining the union complexity of regular language. The union-complexity of a language is one if and only if it is union-free regular.

An algorithm is proposed in this thesis report, for determining whether a regular language is union-free or not. The same is implemented in .NET. For decomposition of a regular expression into an equivalent union-free regular expression, an algorithm is proposed. Size relationship between non-union-free and equivalent union-free regular expression is also discussed.

Table of Contents

Certificate……………………………………………………………………………….i

Acknowledgement…………………………………………………………………...ii

Abstract……………………………………………………………………………...iii

Table of Contents……………………………………………………………………iv

List of Figures and Tables…………………………………………………………..vi

Chapter 1: Introduction

1.1 Automata……………………………………………………………………………1

1.2 Definitions and Notations…………………………………………………………..3

1.2.1 Alphabet and Language……………………………………………......3

1.2.2 Deterministic Finite Automata…………………………….....................4

1.2.3 Non-deterministic Finite Automata…………………………………………5

1.2.4 Computation...…..6

1.2.5 Regular Expression…………………………………………………….....6

1.2.6 Applications of Regular Expressions……………………………………......7

1.3 Operations on Regular Languages………………………………………………......7

1.4 Thesis Outline…………………………………………………………………………9

Chapter 2: Union-free Regular Languages

2.1 Union of Regular Languages………………………………………………………...10

2.2 Component of a Regular Language…………………………………………….....11

2.3 Union Width of a Regular Language………………….…………………………..11

2.4 Cycle-Free Path in an Automata…………………………………………………..12

2.5 Union-Free Regular Languages…………………………………………………...13

2.6 Union-Free Decomposition of a Regular Language……………………………....14

2.7 Properties of Union-Free Languages……………………………………………...16

2.8 Star Height of a Regular Expression…………………………………………………17

2.9 Algorithm for Union-Free Decomposition of Regular Language……………………18

Chapter 3: Problem Statement

3.1 Problem Statement…………………………………………..……………20

3.2 Objectives and Methodology……………………………….………………20

3.3 Motivation………………………………………………….……………..20

3.4 Union-Complexity of Regular languages……..…………………….………………21
3.4.1 Algorithm for Decomposition of the Tree Form……………………...…....25
3.4.2 Application of Normal Form in Syntactical Description…………………..25
3.4.3 Union Complexity and Finite Automata…………………………………..26
Chapter 4: Proposed Solution
4.1 Algorithm 1……………………………………..………………………….…..28
4.2 Algorithm 2.…………………………………………………………………….30
4.3 Algorithm 3……………………………………………………………………..30
4.4 Tool to determine whether the given regular expression is union-free or not……….37
4.5 Size Relationship between Union-free and Non-union-free Regular Languages……44
Chapter 5: Conclusions and Future Work
References……………………………………………………………………………..47
List of Publications……………………………………………………………………50

List of Figures and Tables

Figure No. Figure title **Page No.**

Figure 1.1 Representation of a general automaton………………………………...….. 2

Figure 1.2 Automata accepting the string "ab"……………………………………….....3

Figure 1.3 Transition diagram of regular expression accepting even number of ones….4

Figure 1.4 Transition diagram of non-deterministic finite automata……..…….………5

Figure 2.1 Automata for $L_{1+}L_2$.............…………..…………………….....................10

Figure 2.2 Automata for $L_1 + L_2$……………………………………………………11

Figure 2.3 Automata without any cycle…………………………………......................12

Figure 2.4 Automata without a cycle………………………………………………..12

Figure 2.5 Automata with a cycle…..………………………………………………13

Figure 2.6 Automata with a cycle…..………………………………………………13

Figure 2.7 Representation of an automata M with q4 as final state……………………15

Figure 2.8 Representation of an automata S1 with q0,q2,q3,q4 as final states……......15

Figure 2.9 Representation of an automata S2 with q0 as final state…………………..15

Figure 2.10 Representation of an automata S3 with q0, q1 as final states……………...16

Figure 3.1 Three basic operations to compose flow diagram………………………....22

Figure 3.2 Syntax graph representation of the regular expression ab………………....22

Figure 3.3 Syntax graph representation of the regular expression a+b+c……………....23

Figure 3.4 Syntax graph representation of the regular expression a*…………………...23

Figure 3.5 Alternative representations of iteration operation……………………………23

Figure 3.6 A possible rewriting of regular expressions to a union-free form……………24

Figure 3.7 Syntax graph for the regular expression 0(0*) + 1(1*)…………………….....26

Figure 3.8 Representation of an automaton with one final state……………………......26

Figure 3.9 Representation of an automaton with two final states………………………..27

Figure 4.1 Entering a Regular Expression ………………………………………………38

Figure 4.2 Operation analysis of the regular expression.……………..……................38

Figure 4.3 Operation Analysis of the regular expression ……………………………....39

Figure 4.4 Operation analysis of the regular expression………………………………..39

Figure 4.5 Operation conversion of the entered regular expression…………………....40

Figure 4.6 Operation analysis of the entered regular expression……………………….40
Figure 4.7 Operation conversion of the entered regular expression…………………...41
Figure 4.8 Operation analysis of the regular expression……………………………….41
Figure 4.9 Operation conversion of the regular expression……………………………42
Figure 4.10 Operation analysis of the regular expression………………………………42
Figure 4.11 Operation conversion of the regular expression…………………………...43
Figure 4.12 Operation analysis of the regular expression…………………...……….43
Figure 4.13 Operation conversion of the regular expression…...………………………44

Table No. Table Title Page No.

Table 1.1 Transition Table Representing Transition Function of DFA………………...5
Table 1.2 Transition Table representing transition function of NFA ……………..……6

Chapter 1
Introduction

This chapter gives an introduction to automata and regular language and operations on regular languages.

1.1 Automata

The theory of computation is the branch of computer science and mathematics which deals with the problems that can be solved on a model of computation and how efficiently they can be solved [18]. In theoretical computer science, an automata is an abstract model of a digital computer. Automata theory [12, 26] is closely related to formal languages. Automata are often classified by the class of formal languages, which they recognize. Study of automata is an important part of core Computer Science. Automata is used in digital circuits (designing and checking the behavior), lexical analysis(phase of compiler construction), software for scanning large bodies of text and software for verifying systems of all types that have finite number of distinct states [25]. Automata include some essential features. It has a mechanism of reading input for which, the input is written on input file. This can be only be read by the automata but cannot be changed by it. The input file is divided into cells, each cell holding one symbol.

Automata can produce output with the help of a temporary storage device, which consists of an unlimited number of cells. The automata can read and change the contents of the storage cells. Finally, it has a control unit, which can be in any one of a finite number of internal states. The state can be changed in some defined manner. An automata is assumed to operate in a discrete time frame. At any given time, the control unit is in some internal state, and the input mechanism scans a particular symbol on the input file. The internal state of the control unit at the next time step is determined by the transition function. The transition function gives the next state in terms of current state, the current input symbol, and the information currently in the temporary storage. Figure 1.1 shows a systematic representation of general automata [18]. An automata is represented by

directed edges and circles. The directed edges are labeled with the input alphabets and each state is labeled with the state number.

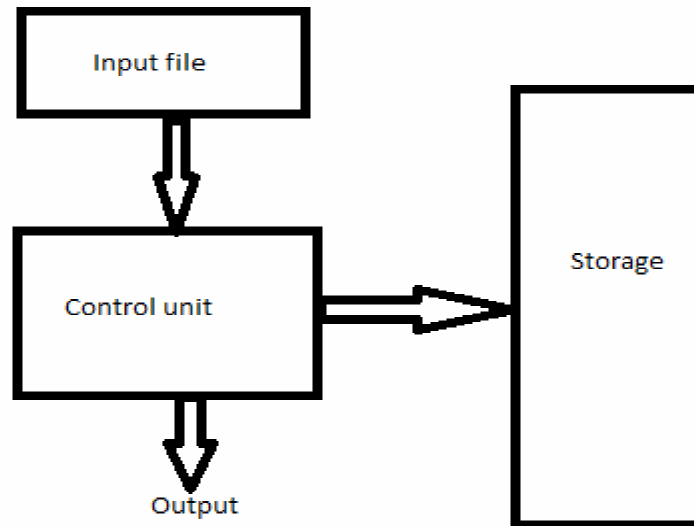


Figure 1.1: Representation of a general automata.

There are different types of states used in the representation of an automata. The states can be classified as initial state, intermediate state and final state. Initial state is represented by a circle and directed edge towards the circle from no other state. There are directed edges from other states towards initial state. Intermediate state is represented by directed edges towards that state from other states. Final state is represented by two concentric circles and directed edges towards that state from other states. There can be more than one final state but a single initial state. It is necessary to indicate the initial state and final state of finite automata, in order to determine the sequence of input symbols accepted by finite automata. For any sequence to be accepted by the automata, it needs to start from the initial state, go through the set of intermediate states and reach the final state for acceptance in the end.

Example 1.1: The automata in the figure 1.2 accepts the string "ab" using three different types of states.

As shown in the figure 1.2, the finite automata has three states (q0, q1, q2) and there is a transition from one state to another with a particular input symbol. This finite automata accepts the string "ab" and each state of finite automata represents the different position

of string that has been reached till that state. The acceptance of the string starts from the initial state, reaches to the state q1 after a transition from state q0. Finally reaches at state q2 and gets accepted at state q2.

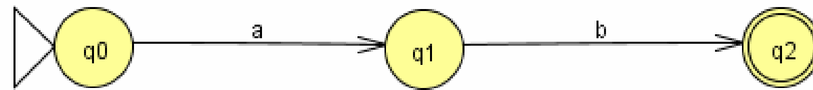


Figure 1.2: Automata accepting the string “ab”.

Finite automata can be classified in two categories, deterministic finite automata (DFA) and non-deterministic finite automata(NFA). The language accepted by a DFA and NFA is a regular language. Every regular language which can be represented by a non-deterministic finite automata (NFA) can also be described by a deterministic finite automata (DFA) equivalently.

1.2 Definitions and Notations

In this section, various definitions and notations regarding regular languages are discussed.

1.2.1 Alphabet and Language

Def. 1: Alphabet is a finite non-empty set of symbols, on which a language is defined. Alphabet is denoted by Σ [18].

Example 1.2: Given an alphabet Σ ={a , b}. The strings corresponding to the alphabet can be a, b, ab, abab, aaabbb. The strings corresponding to an alphabet depend on the language over that alphabet.

Def. 2: Language is defined as a subset of Σ^* (alphabet). Empty string and null language are denoted by ε and ∅ respectively [13]. There are various kinds of formal languages which can be classified as regular, context free, context sensitive and recursive language. Regular language can be described by a regular expression or finite automata (Deterministic finite automata or Non-deterministic finite automata). The representation of the regular language by the two automata is described in the following sections.

Example 1.3: Let a language L be defined over an alphabet Σ ={0, 1}. Language consisting of all the strings having length zero, one or two includes the strings L = {ε,0,1,00,01,10,11}.

1.2.2: Deterministic Finite Automata

A deterministic finite automata(DFA) [13] is defined by the quintuple M= (Q, Σ, δ, q_0, F) where:

Q is a finite set of internal states,

Σ is a finite set of symbols called input alphabet,

δ: Q X Σ → Q is called as transition function,

q_0 is the initial state,

F⊆Q is a set of final states.

Deterministic finite automata has rules for transitions from one state to another represented by transition function which depends on the present state and present input alphabet. Deterministic finite automata can be represented by a transition diagram or transition table, shown in the figure 1.3 and table 1.1 respectively.

Example 1.4: The DFA shown in figure 1.3 is represented by ({q0, q1}, {0, 1}, δ, {q0}, {q1}) where, δ is transition function, generating the transition from one state to another. The figure shows the transition diagram depicting the transition from one state to other based on the state and the input alphabet.

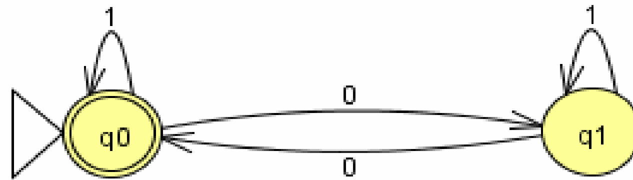


Figure 1.3: Transition diagram of a regular expression accepting even number of 1's

Table 1.1 represents the transition table corresponding to the above transition diagram accepting even number of ones. In the table, a transition from one state to another is depicted in tabular form. Transitions from one state to another, depends upon the input symbol and the current state. Second and third column of the table represent the next state.

Table 1.1: Transition Table Representing Transition Function of DFA

Previous State	Next State δ(Q,0)	Next State δ(Q,1)
q0	q1	q0
q1	q0	q1

1.2.3: Non-Deterministic Finite Automata

A non-deterministic finite automata or NFA is defined by the quintuple M= (Q, Σ, δ, q_0, F) where Q, Σ, q_0, F are same as defined for deterministic finite automata, except the transition function δ: Q X {Σ U {ε}} → 2^Q [13].

NFA has internal states, rules for transitions from one state to another represented by transition function and some input. NFA can also be represented by a transition diagram or a transition table. Non-determinism refers a choice of moves for automata. In contrast, deterministic finite automata prescribe a unique move in each combination of present state and present input alphabet defined by the transition function. Equivalent deterministic finite automata can be constructed corresponding to each non deterministic finite automata. Figure 1.4 represents a non deterministic finite automata corresponding to language (a+b)*ab. The choice of moves can be clearly seen from the figure. The choice occurs at q0 state for input alphabet ‘a’. On readung ‘a’, state can be changed to q1 or the control is again passed to state q0. Hence, there is a choice of moves, in non deterministic finite automata.

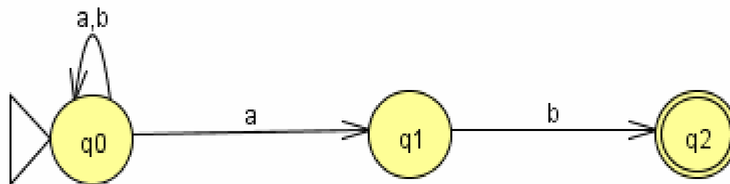


Figure 1.4: Transition diagram of non-deterministic finite automata.

The table 1.2 represents the transition table corresponding to the transition diagram in figure 1.4, accepting the regular expression (a+b)*ab. Table 1.2 represents the tabular representation of the transition from one state to the other based on present state and input alphabet.

Table 1.2: Transition Table Representing Transition Function of NFA

Previous State	Next State δ(Q, a)	Next State δ(Q, b)
0	0	0
1	---	2
2	---	---

1.2.4 Computation

Let M = (Q, ∑, δ,q_0, F) be a finite automata and let w = $w_1 w_2 \dots w_n$ be a string where each w_i is a member of alphabet ∑ . Automata accept w, if a sequence of states $r_0 r_1 \dots r_n$ in Q exists with three conditions [18]:

1. $r_0 = q_0$
2. $\delta(r_i, w_{i+1}) = r_{i+1}$ for i=0, … , n-1
3. $r_n \in F$

M recognizes a language A if A = {w | M accepts w}.In other words, a language is a set of all of those strings that are accepted by the finite automata.

1.2.5 Regular Expression

One way of describing regular languages is through the notation of regular expressions (RE). The notation involves a combination of strings of symbols from some alphabet. It also consists of parenthesis and the operators +, **.** , * which are the union, concatenation and star operators respectively . Language consisting of string {a}, can be denoted by the regular expression "a" [18, 26] . Regular expression for an alphabet is represented by itself. The empty string and null language are regular expressions, denoting the language {ε} and {∅ } respectively. Other regular languages are represented by the combination of operators and the input alphabets. In other words, a regular expression over input alphabets Σcan be defined as:

1. Every input alphabet can be represented by itself.
2. Null language and null string represent themselves.
3. If r_1 and r_2 are regular expressions representing the languages l_1 and l_2 respectively, then:

   3.1 Union of r_1 and r_2 is represented by $r_1 + r_2$..

   3.2 Kleene closure of the regular expression is represented by $(r_1)^*$.

3.3 Concatenation of r_1 and r_2 is represented by r_1r_2.

4. Rule 3 can be defined recursively.

Example 1.5: $a+b^*$ denotes { ε, a, b, bb, bbb, ...} means single 'a' and zero or more number of b's. $(a+b)^*$ denotes the set of all strings consisting of a and b, including the empty string: { ε, a, b, aa, ab, ba, bb, aaa,...}. The RE $ab^*(c|ε)$ denotes the set of strings starting with a, then zero or more b's and optionally c: {a, ac, ab, abc, abb, abbc, ...}.

1.2.6 Applications of Regular Expressions: Regular expressions are well known in the field of computer science. They are commonly used and well-applicable in theory as well as in practice. The regular expressions are used in field of compilers, programming languages, pattern recognition, protocol conformance testing etc. Two are discussed as follows:

1. Lexical analyzers: The tokens of the programming language can be expressed using regular expressions. The lexical analyzer scans the input program and separates the tokens.

Example 1.6: Identifier can be expressed as a regular expression : (letter)(letter+digit)* . The letter in RE is {A,B,C,……….Z,a,b,c….z} and digit is {0,1,…9}. If anything in the source language matches with this regular expression then it is recognized as an identifier.

2. Text editors: These are the programs used for processing the text. In UNIX text editors any regular expression is converted to a NFA with ϵ transitions and this NFA can be then simulated directly.

1.3 Operations on Regular Languages

Various operations such as intersection, union, concatenation, closure, complement and reversal, can be performed on regular languages and are described as follows:

1. **Intersection of Regular Languages:** The "∩" operator is used for intersection operation. Suppose L_1 and L_2 are two regular languages, then intersection of these regular languages is given by $L_1 \cap L_2$. The strings which are common between the two languages are the result of this operation.

Example 1.7: Given two regular languages $L_1=\{a,b,c\}$ and $L_2=\{a\}$ then $L_1 \cap L_2=\{a\}$.

2. **Union of Regular Languages:** The "U" operator is used for union operation. If L_1 and L_2 are two regular languages, then union of these regular languages is represented by $L_1 U L_2$. It gives the resultant language that accepts all string from L_1 and L_2.

Example 1.8: If A and B are regular expression, then A+B is a regular expression denoting the union of L(A) and L(B).

3. **Concatenation of Regular Languages:** The "." operator is used for union operation. If A and B are the regular expressions, then A.B is a regular expression denoting the concatenation of L(A) and L(B). The concatenation of languages L_1 and L_2 is represented by $L_1 L_2 = \{xy \mid x \varepsilon L_1, y \varepsilon L_2\}$. It denotes all strings that are formed by concatenation of a string x from L_1 with a string y from L_2 [25] .

Example 1.9: If $L_1=\{a, b\}$ and $L_2=\{ab, abb\}$ then the concatenation of the language L_1 and L_2 is $L_1 L_2=\{aab,aabb,bab,babb\}$.

4. **Closure or Kleene Closure of Regular Languages:** The kleene closure of a given regular language L is the collection of all possible finite-length strings generated from the symbols in L including null string. It is denoted by L*[13].

Example 1.10: If the regular language L contains alphabets $\Sigma=\{a, b\}$ then $L^*=\{\varepsilon, a, b, ab, ba, aa, \ldots\}$ i.e. all the strings containing a and b.

5. **Complement of Regular Languages:** The complement of a regular language is defined with respect to Σ^*. The complement of a regular language L is given by L $= \Sigma^* - L$ [18].

Example 1.11: Suppose $\Sigma = \{a\}$ and the language L consists of all non empty string of a's. L={a, aa , aaa , aaaa , aaaaaa …….}. Then the complement of the language L is given by $L=\Sigma^* - L = \{\varepsilon\}$i.e. L consists only of a null string.

6. **Reversal of Regular Languages:** Given an NFA A (Q, Σ, δ, s, F) for a regular language L, then a NFA A^R (Q, Σ, δ, f, s) for L^R can be constructed by flipping the directions of all transitions and interchanging the start state and the final state [18].

1.4 Thesis Outline

Thesis is organized into five chapters. Chapter 1 gives the description about the basic concepts of automata, language and operation on regular languages. Chapter 2 describes the union-free regular languages and some basic concepts regarding them. Chapter 3 describes the motivation behind the thesis, discusses the problem statement, objectives and methodology and explains the union complexity of regular languages. In chapter 4, an algorithm that determines whether a regular expression is union-free or not is discussed. It also includes an algorithm for converting a regular expression into an equivalent union-free regular expression and tools for the same are represented. Chapter 5 summarizes the conclusions drawn from the thesis along with the directions regarding the future work.

Chapter 2

Union-free Regular Languages

This chapter describes the union of regular languages. It also introduces the concept of component, union width, star height of regular languages and some properties of union free languages. An algorithm is presented in this chapter, which determines whether a regular language is union free or not is discussed.

2.1 Union of Regular Languages

A language is regular if it can be recognized by some finite automata irrespective of its property of determinism. Suppose L_1 and L_2 are two regular languages, then union of these regular languages is the combination of the strings which are contained in either of the two languages that is L_1, L_2. Union of L_1 and L_2 is represented by L_1UL_2.The union operation of regular languages results into strings accepted by either L1 or L_2. If R_1 and R_2 are two regular expressions, then R_1+ R_2 is a regular expression denoting the union of $L(R_1)$ and $L(R_2)$ [18].

Example 2.1: Given two regular languages A and B then,AUB= {xl x belong to A or x belong to B}.

The figure 2.1 shows two automata that accepts a and b respectively. First automata accepts terminal ‘a’ and the second automata accepts the terminal ‘b’. The union of the two automata is represented in the figure 2.2, accepting a+b, using the null moves.

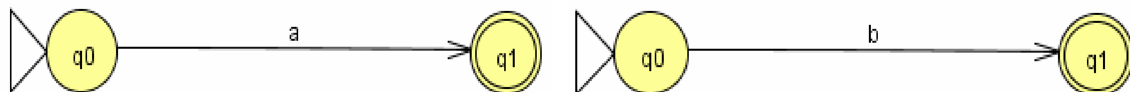


Figure 2.1a: Automata for L_1 **Figure 2.1b:** Automata for L_2

Figure 2.1: Automata for $L_1 + L_2$

The null move is used for applying union operation on two automata. An additional initial and one final state is used for the union operation creating the choice of two paths. Using the initial state, any path can be followed. The final state is used for accepting any of the two strings.

The figure 2.2 represents the union operation of two regular languages with the help of automata.

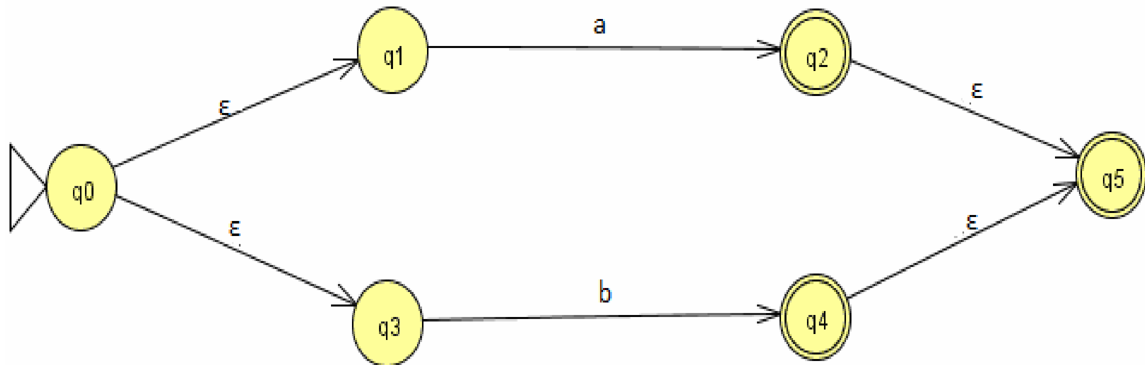


Figure 2.2: Automata for L1 + L2

2.2 Component of a Regular Language

Component of a regular expression is the individual string which is represented by an empty string or by the combination of alphabets, concatenation operators and the star operators. A language is called regular component splittable if it can be expressed as a disjoint union of regular components and a finite set [20].

Example 2.2: If a language L can be represented by a regular expression = {a + ab* +a*b}, then the language has three components.

2.3 Union Width of a Regular Language

The minimum number of components in the representation of a regular expression is called the union width of a regular language. More the number of components more is the union width of a regular expression. It is simple to calculate the union width of a regular expression by simply counting the total number of components in that regular expression [20]. Union width gives an idea about the total length of a regular expression. More the union width, more is the length of a regular expression.

Example 2.3: The regular expression {ε} + a*ba* + b*ab* has 3 components and hence the union width is 3.

2.4 Cycle-Free Path in Automata

A sequence of transitions from one state to another for the string w which belongs Σ^* is called a path iff there is an ordered list of states $\{q_1,\ldots\ldots\ldots\ldots,q_m\}$, $w=a_1\ldots\ldots a_{m-1}$ and $\delta(q_i,a_i)=q_{i+1}$. A cycle in the path occurs if a string starts from one state goes through sequence of at least three states and returns to the starting state. A path in an automata is called cycle free iff it starts from an initial state q_0, ends at a final state q_f belongs to F and does not contain any cycle. Existence of a cycle in an automata does not determine the union-freeness of a language corresponding to that automata. Even if there is no cycle in automata, the corresponding regular language is not necessarily union free [14]. If there is a cycle in automata then the corresponding language can be union free. Following figures shows the relationship between the cycle free path in an automata and the union freeness of regular language corresponding to that automata. Figure 2.3 represents the automata without any cycle and the corresponding regular expression is (a+b)*ab. The regular expression is not union free. Hence depicting that non existence of a cycle does not leads to the union freeness of regular language. Language is non union-free regular because of occurrence of two terminals on the self loop in the initial state.

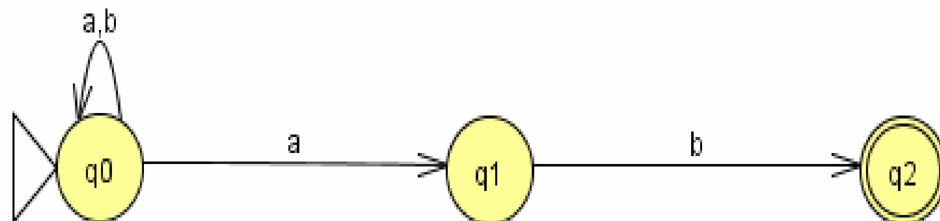


Figure 2.3: Automata without cycle and the corresponding language is not union-free.

Figure 2.4 represents the automata without a cycle and the corresponding regular expression comes out to be b*ab*a. The regular expression is union free. From this figure it is evident that automata without any cycle can have union-free regular expression.

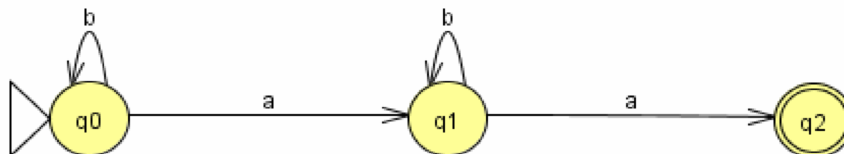


Figure 2.4: Automata without a cycle and the corresponding language is union-free.

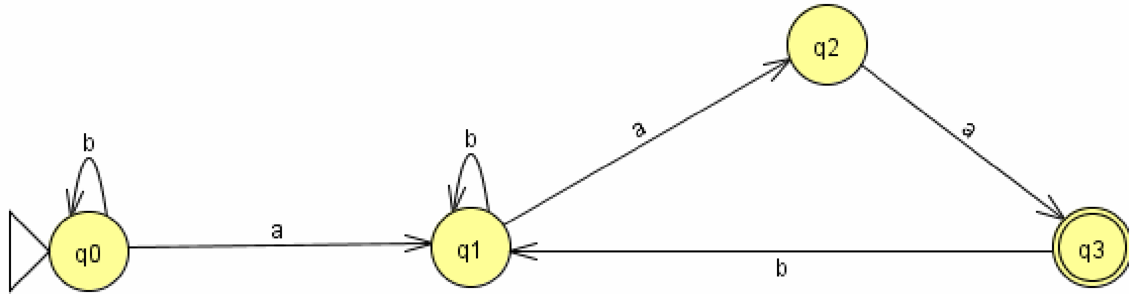


Figure 2.5: Automata with a cycle and the corresponding language is union-free.

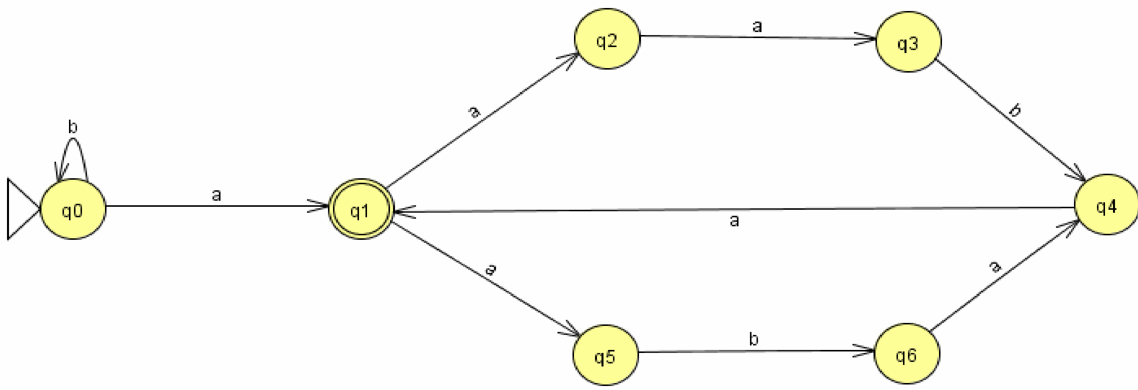


Figure 2.6: Automata with a cycle and the corresponding language is not union-free.

2.5 Union-Free Regular Languages

A regular language is said to be union-free [20] if it can be represented by a regular expression without using the union operation or union of union-free regular languages. **Example 2.4:** the expression (a + b*)* is not union-free. On the other hand, the regular expression a* + b* is union-free. Every regular expression r can be transformed into a regular expression r' in which union operations appear only on the "top level" of the expression, i.e., r' can be of the form: $r' = r_1 + \ldots + r_m$, whose regular expressions $r_1, \ldots, r_m$ does not contain the "+" operator [20]. This means that every regular language can be represented as a finite union of union-free languages. But this decomposition is not necessarily unique.

Example 2.5: (a + b)* can be written as (a*b*)* or it can be also be written as {ε} + a*ba* + b*ab*, and these are two different union-free decompositions of the language (a+b)*.Another example can be (a + b)a* can be written as aa* + ba* , $(a + b)^3$ can be

equal to (a + b) (a + b) (a + b) or equal to aaa + abb + aab + aba + bbb + bab + baa + bba.

2.6 Union-Free Decomposition of a Regular Language

Union free decomposition of a regular language corresponds to the decomposition, which can either be represented by a union-free regular expression or can be represented by a finite union of union free regular expressions.

The problem of constructing union-free decompositions of regular languages also has practical applications. In particular, regular languages can be used for a description of the syntactic structure of a programming language. The concatenation operation corresponds to the sequential continuation, the Kleene star corresponds to loops, and the union operation corresponds to branching. In this context, union-free languages represent sequences of operators that do not contain any conditional transitions. Minimal union-free decompositions of regular languages may be useful for simplifying and normalizing such descriptions [20].

Definition 2.1:

Let L be a regular language, then $L = L_1 \cup L_2 \cup \ldots\ldots \cup L_k$ is called a union-free decomposition of L iff L_i is a union-free language for all i = 1, . . . , k. The decomposition is called minimal iff there is no other union-free decomposition of L with fewer elements [9].

Definition 2.2:

A language W □ Σ^* is called a star language iff W = V* for some V □Σ^*. A language L is called union-free iff it can be represented by a regular expression that contains the star and concatenation operations only, it takes the following form [11]:

$$L = S_{01}^* S_{02}^* \cdots S_{0k_0}^* u_1 S_{11}^* \cdots S_{1k_1}^* u_2 \cdots S_{l-1,1}^* \cdots S_{l-1,k_{l-1}}^* u_l S_{l,1}^* \cdots S_{l,k_l}^*$$

where Sij are regular languages, $u_1, \ldots\ldots\ldots\ldots, u_l$ are non-empty strings, and l>= 0. The above equation can be called as a general form of a union-free language and is denoted as GF(L) [20].

The figure 2.7 represents the example of union free language. It is difficult to recognize a union free language by just looking at the automata. The language is represented by

$M=S_1{}^*bS_2{}^*aS_3{}^*$ where M,S_1,S_2,S_3 are shown in the figures 2.7, 2.8, 2.9, 2.10 respectively. Let L be a union free language. If u and v are shortest strings in L then u=v. General form of a union-free regular language is represented by the above equation (L), v $\in$ L and length of v is equal to that of u, $v_i=u_i$ for i= 1,………,l hence u=v.

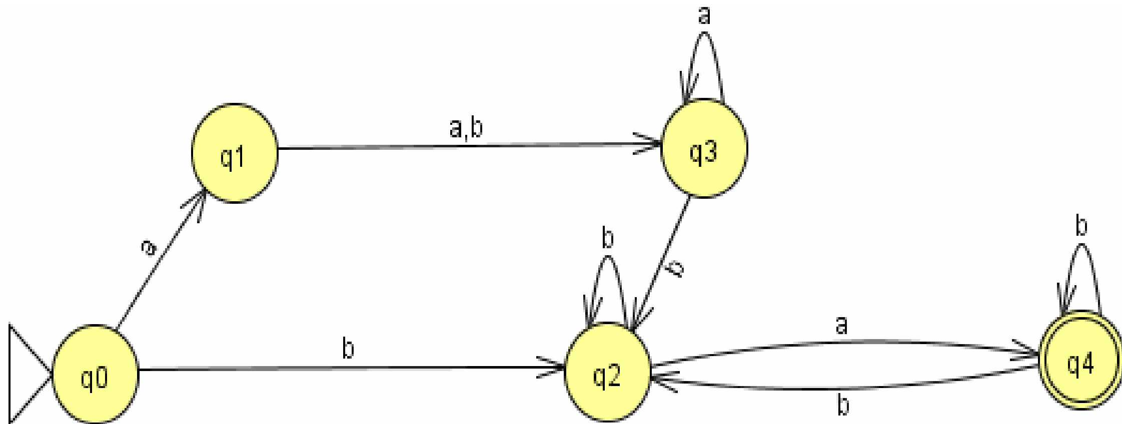


Figure 2.7: Representation of an automata M with q4 as final state [20].

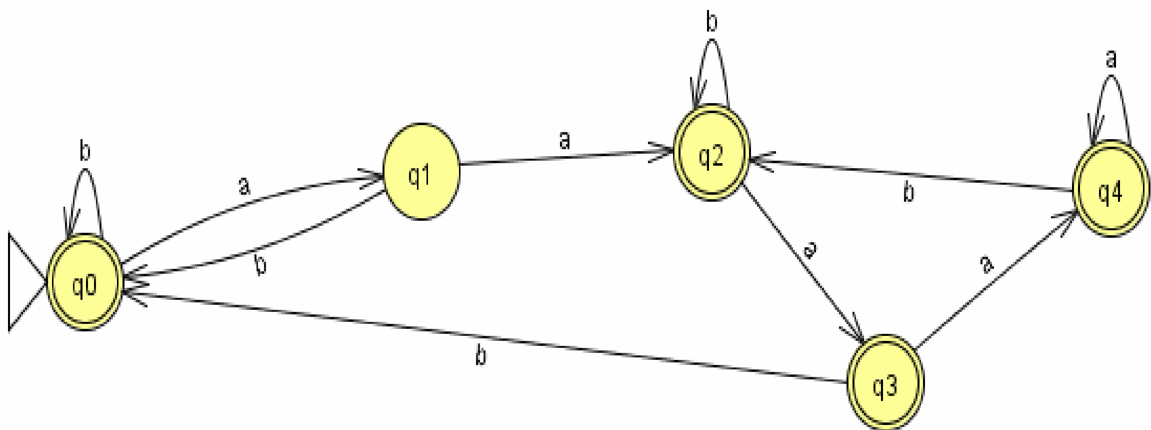


Figure 2.8: Representation of an automata S1 with q0,q2,q3,q4 as final states [20].

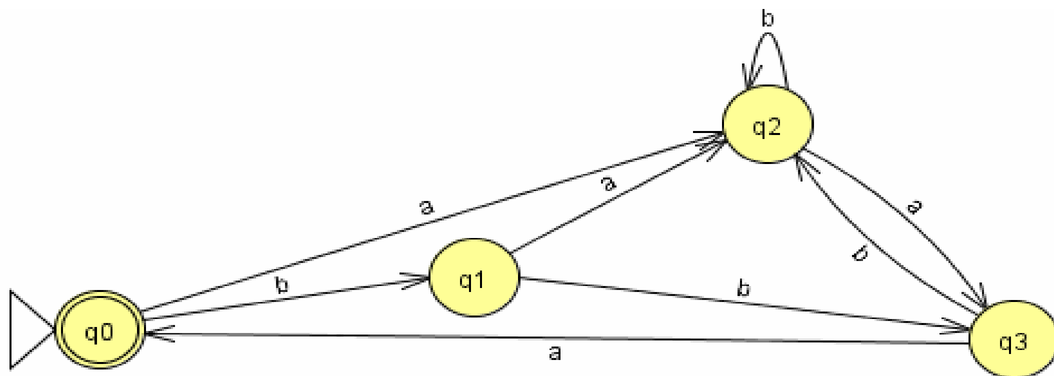


Figure 2.9: Representation of automata S2 with q0 as final state [20].

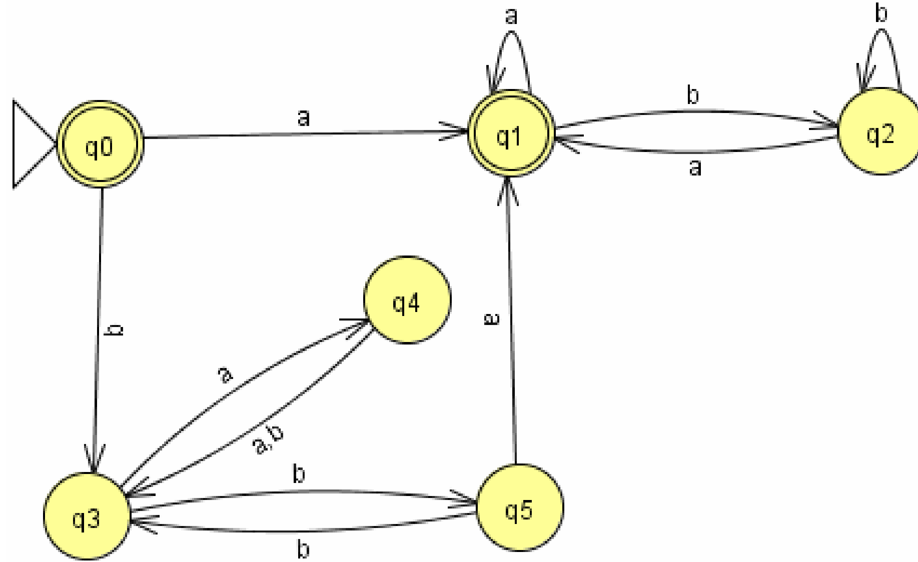


Figure 2.10: Representation of automata S3 with q0, q1 as final states [20].

2.7 Properties of Union-Free Languages

In this section some common properties of union free languages are described.

1. Reversal of union-free regular languages: The class of union free regular languages is closed under reversal [9]. Given a union-free regular language L, the reversal of the language L is a union-free regular language.

Example 2.4: Let ba* represents a union-free regular language. The reversal of the language is a*b. a*b is a union-free regular expression. Union-free regular languages are closed under reversal because even by reversing the regular expression there is no occurrence of the union operator.

Concatenation of union-free regular languages: The union-free languages are closed under concatenation [17]. Given two regular languages L_1 and L_2, the resultant regular language L_1L_2 is union-free.

Example 2.5: Let a*b and (ab)* represents two union-free regular languages. Then the concatenation of these two regular languages become equals to a*b(ab)*. The resultant regular language is also a union-free regular language.

2. Kleene closure of union-free regular languages: The union-free languages are not closed under Kleene-star. For every union-free regular language L, kleene

operation is represented by L*, which may not result into union-free regular language.

Example 2.6: Given a union-free regular language L=a+b, the kleene closure of language L will result into the language L*= (a+b)* which is not union-free.

3. Union of union-free regular languages: The class of union free regular languages is closed under union.

Example 2.7: Given two union-free regular languages L_1 and L_2, where L_1 = a*ba* and L_2 = b*ab*. The union of two languages result into the language L_1+ L_2=a*ba* + b*ab*, which is union-free.

2.8 Star Height of a Regular Expression

The star height sh(r) [3] of a regular expression r is defined syntactically as the maximum number of nested stars that r contains. The star height sh(r) of a regular language L is the least natural number n such that sh(r) = n for some regular expression r that represents L. Following are some results regarding the star height of regular expressions:

1. Every regular language over a single-letter alphabet is of star height one at most.

2. There are regular languages with any preassigned star height over any alphabet containing at least two letters.

3. There exists an algorithm for computing the star height of a regular language given by a regular expression.

The star height of a regular expression E over a finite alphabet A is inductively defined as follows

h()=0 , h(□)=0 and h(a)=0 for all the alphabet symbols a in A.

h(AB)=h(A|B)=max(h(A),h(B))

h(A*)=h(A) + 1

Where ε is the special regular expression denoting the empty string. While computing the star height of a regular expression is easy, determining the star height of a language can be sometimes tricky. For illustration, the regular expression (ab* + b)*a*b* . Over the alphabet A = {a,b} has star height 2. Thus the language can also be described by the expression (a + b)*a which is only of star height 1.

2.9 Algorithm for Union-Free Decomposition of Regular Language

In this section an Algorithm for union free decomposition of regular expression is discussed.

Theorem 2.2. Let L be a regular language. Then there exists an algorithm that results in a minimal union-free decomposition of L where $L = L1 \cup L2 \cup \ldots\ldots\ldots\ldots \cup L_k$ [20].

Proof.[20] To construct a minimal union-free decomposition, all the subsets are examined of C(L) and choose the subset containing a minimum number of languages which being added up are equal to L. A non union- free languages can be decomposed into a number of subsets. From these subsets few are selected such that they are union free and union of them is equal to the original language. Thus obtaining a decomposition of L into languages from C(L). The final step is to show that the decomposition is obtained and there exists no other decomposition containing fewer elements. Suppose there is such decomposition $L = N_1 \cup N_2 \cup \ldots \cup N_p$, p<k. Then the language Ni (i = 1, . . . , p) and getting a union-free language $C_{N_i} \cup C(L)$ such that Ni is a subset of C_{N_i}. Thus the new decomposition $L = C_{N_1} \cup C_{N_2} \cup \ldots\ldots \cup C_{N_p}$, p< k and every language C_{N_i} belongs to the set C(L). But since all the subsets of C(L) are already examined, and the subset $\{C_{N_1}, \ldots, C_{N_p}\}$ has also been examined , and this subset must have been chosen for the minimal decomposition. This contradiction completes the proof. Algorithm for constructing a minimal union-free decomposition of a given regular language L is computationally expensive since it requires checking all the subsets of the set C(L), which can contain up to $c|Q|(2^{|Q|})|^{Q|-1}$ elements , where c is the number of cycle-free paths in the automata associated with L .

A promising way of constructing minimal union-free decompositions is as follows. Let L be a regular language. The equation L = X*L is proved to have the unique maximal solution X_0 Moreover, the equation $L = (X_0)^* Y$ is proved to have the unique minimal solution Y_0. To construct a minimal union-free decomposition of L it is required to solve these two equations and obtain the language Y_0. Then the same procedure is applied to the language Y_0 and get the minimal language Y_1 such that $L = X_0{}^* X_1{}^* Y_1$. If the process ends the result is resulting either obtaining the

$$L = X_0^* X_1^* \cdots X_m^* Y_{m_1} \cup X_0^* X_1^* \cdots X_m^* Y_{m_2} \cup \cdots \cup X_0^* X_1^* \cdots X_m^* Y_{m_n}$$

language $Y_m = \{\varepsilon\}$ (and thus the union-free decomposition $L = X_0^* X_1^* \ldots\ldots\ldots\ldots X_m^*$) or get a language Y_m such that the equation $Y_m = X^*Y_m$ has no non-trivial solutions. The latter case can be checked whether all the strings in the language Y_m start with the same letter. If it is the case and, for example, all the strings in Y_m start with the a, then $Y_{m=a}Y_m'$,$L_=$ $X_0^* X_1^* \ldots X_m Y_m$ and apply the procedure described above to the language $Y'm$ (by solving the equation $Y'_m = X^*Y'_m$ etc.). If it is not, and there are strings in Y_m that start with different letters, e.g. $a_1, \ldots\ldots\ldots, a_n$, and Y_m can be $Y_m = Y_{m1} \cup \ldots\ldots\ldots\ldots \cup Y_{mn}$ so that every language $Y_{m1}, \ldots\ldots\ldots, Y_{mn}$ contains only strings starting with the same letter a_i, $1 <= i <= n$. Then writing the following equation and applying the above procedure to every language $Y_{m1}, \ldots\ldots\ldots\ldots, Y_{mn}$. By the end of the process the union-free decomposition of the language L is obtained.

Chapter 3
Problem Statement

This chapter includes the problem statement followed by the motivation behind the thesis and the union-complexity of regular languages.

3.1 Problem Statement

The given problem relates to the development of an efficient algorithm for determining whether the given regular language is union-free or not . Development of a tool is required for the same. Moreover, an efficient algorithm and a corresponding tool is required to convert the regular expression into an equivalent regular expression which is union-free. Some approaches are available for finding the union freeness of regular expressions and its corresponding union-free decomposition into union-free regular expression. But the purpose is to develop an efficient algorithm which is easily understandable and solves the given problem.

3.2 Objectives and Methodology

Following points represent the main objectives for the work done in this thesis report:

1. Development of an efficient algorithm to determine whether the given regular expression is union-free or not.
2. An algorithm design to decompose the given regular expression into an equivalent union-free regular expression.
3. Development and implementation of a tool corresponding to the algorithms.
4. Developing a relationship between the lengths of the non union-free regular expression and its corresponding union-free regular expression.

3.3 Motivation

There exists an algorithm [21] for union-free decomposition of regular language. The algorithm uses a set of all maximal finite concatenations of languages. To construct a union-free decomposition, the algorithm examines all the subsets of maximal finite

concatenations of languages and choose the subset containing a minimum number of languages. It should be noted that there is at least one subset containing languages which when added up are equal to language, because there exists at least one union-free decomposition of language thus obtaining a decomposition of language into languages from maximal finite concatenations of languages.

By decomposing the regular language into an equivalent union-free regular language also helps in determining the union complexity of regular language. The union-complexity of a language is 1 if and only if it is union-free regular. Moreover, by the decomposition of regular language, the normal form of union-free regular languages can be described. The normal form represents the finite union of union-free regular languages. The normal forms are used in various areas of computer science. For instance, in logic, conjunctive and disjunctive normal forms are used. Every regular expression can be transformed to a normal form by using equivalences of the language or decomposing it into an equivalent union-free regular language.

An algorithm to find whether a regular language is union free or not, can be developed. Based upon the cycle in an automata some results can be derived about the union-freeness of a regular language. A user friendly tool corresponding to the algorithm can help to easily and quickly determine the union-freeness of a regular language. Another algorithm for converting a regular language into the union-free regular language is presented in this thesis. In order to make it easy to implement and use, a corresponding tool has been developed which converts the regular expression entered by the user into the union-free regular expression.

3.4 Union-Complexity of Regular languages

Regular expressions can be represented by expression tree and by flow diagram (syntax graph) [17]. A possible normal form [17] of regular expressions can be represented using union-free expressions. Based upon this form, union-complexity is defined. The normal form is equivalent to the finite union of union-free regular languages. The regular expressions can also be represented by a directed graph consisting of nodes and directed edges. Diagrammatic representation of regular expressions is called as flow diagram or syntax graph. The figure consists of nodes and edges. For simple regular expressions

there are simple figures having in degree and out degree one. The figures which are not simpler consist of combinations of edges and nodes. These arrows connect the simple syntax graphs into the desired order. The graphs can be used for representation of terminals. Three operations on regular languages (concatenation, union and iteration (star operation)) can also be represented. The graphical form of these operations can be seen in the following figure. The terminals are ellipses shown in the figure 3.1. The first part of the figure represents the concatenation operation on the symbols. The second part of the figure represents the union operation on the symbols. The last part of the figure represents the iteration operation of the symbols.

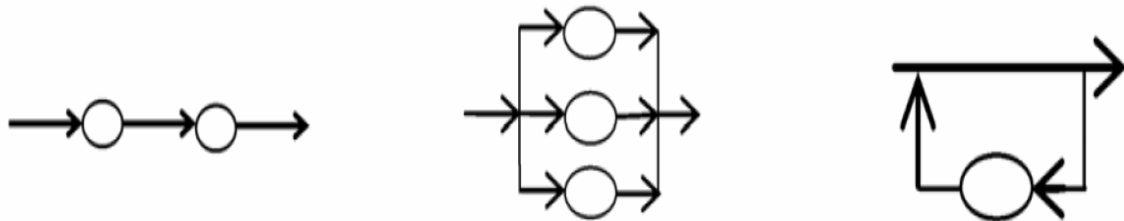


Figure 3.1: Basic operations (concatenation (left), union (middle) and iteration (right)) of flow diagram [17].

Example 3.1: A regular expression a.b consists of the concatenation operation between a and b. It can be represented by the following figure

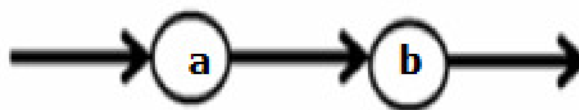


Figure 3.2: Syntax graph representation of the regular expression ab [17].

Example 3.2: A regular expression a+b+c is consists of the union operation between a, b and c. It is represented in the figure 3.3. Any of the three paths can be followed for accepting any of the three terminals. There is a choice for the acceptance of the terminals due to the union operation.

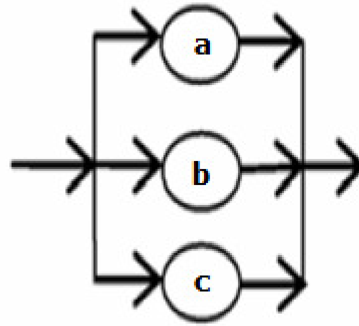


Figure 3.3: Syntax graph representation of the regular expression a+b+c [17].

Example 3.3: A regular expression a* consists of the iteration operation. As a result of this, a can be repeated any number of times as shown in the figure 3.4

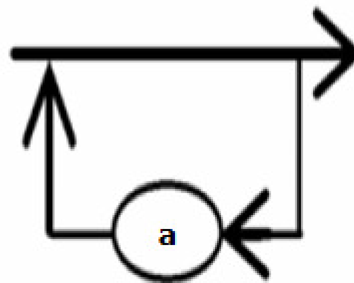


Figure 3.4: Syntax graph representation of the regular expression a*[17].

Example 3.4: The iteration operation can be represented by the either of the two figures in figure 3.5. The iteration operation can be represented, either by a loop on the upper part or the lower part of the syntax diagram of the regular expression.

Figure 3.5: Alternative representations of iteration operation [17].

Definition 3.1: A regular expression is said to be in normal form if it can be represented by union of union-free regular languages.

Normal form of regular expressions can be represented using union-free regular expressions. Based on the normal form or the union freeness of the regular expression the union-complexity is defined. The regular expression can be represented by a tree diagram [17]. The root and children in the tree diagram consists of the operators and terminals. A

regular expression which is not union-free can be represented by the tree diagram and its corresponding decomposition is also represented by omitting the union operator of union-free regular expression. In the tree diagram represented by figure 3.6, left tree represents the regular expression which is not union-free. In Figure 3.6b an equivalent union-free regular expression is represented. Although the depth of the tree increases after decomposition, but the union complexity decreases as the union complexity can be determined from a tree by counting the sub trees having the union operator.

Let a and b be arbitrary regular expressions. (a+ b)* can be written in the form (a*b*)*.The figure 3.6 represents the tree description of the both the regular expressions.

The following equivalences can be used in regular expressions to easily represent the decomposition of regular expressions [17].

Let a, b, c and d are arbitrary regular expressions.

1. (a + b)c is equivalent to (ac + bc),
2. a(b +c) is equivalent to (ab + ac),
3. s(a+b)(c+d) is equivalent to (ac+ad)+(bc+bd).

In the figure 3.6, it can be clearly seen that the left tree represents the regular expression which is not union-free. In order to draw a tree for an equivalent union-free regular expression, the union operator is replaced with star and concatenation operators.

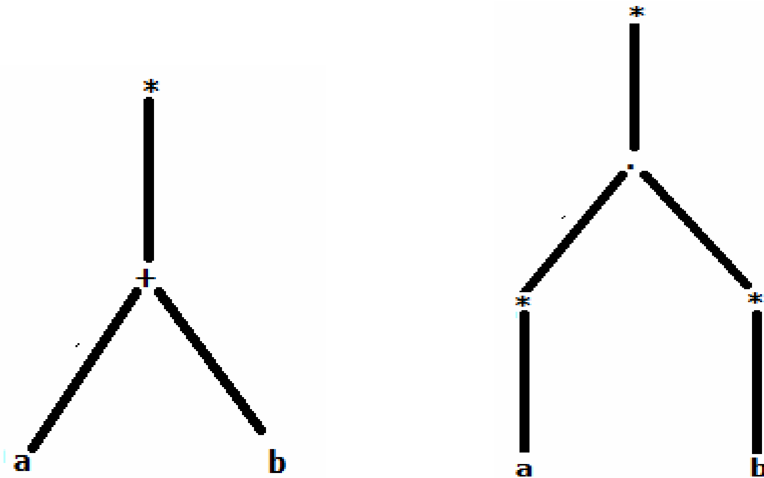


Figure 3.6a: Non union-free expression **Figure 3.6b** Union-free expression

Figure 3.6: A possible rewriting of regular expressions to a union-free form [17].

The depth of the tree is increased, but the union complexity is decreased, since there is no union operator in the right tree. Using the above mentioned equivalences (2, 3, 4) the

union can be substituted or moved upward in the tree of the regular expression. The following algorithm explains the decomposition of the tree form into an equivalent union-free regular expression.

3.4.1 Algorithm [17] for decomposition of the tree form into an equivalent tree form (union-free regular expression)

1. Input: The tree form of a regular expression.
2. Output: A tree describing the same language in which operator + cannot occur below other operator.
3. If there is no other operation in a higher level than the union then exit.
4. Else repeat following steps until there is other operation higher then union.
   4.1 If there is a union which is immediately under a Kleene star then convert it into normal form.
   4.2 If there is a union which is under a concatenation then use one of the above three equivalences (2)-(4) according to the places and numbers of unions.

Example 3.5: Let a normal form of a language is given by the expression: a + bc(a*b*)* + a(a*b*)*. Clearly, it is a proper normal-from.

3.4.2 Application of Normal Form in Syntactical Description

Normal form can be used for regular expressions for representing them in syntactical description [17]. The normal form of regular expressions gives a normal form for BNF expressions and syntax graphs as well. For the regular expression 0(0*) + 1(1*) the syntax graph is represented as shown in the figure 3.7. Firstly, there is a decision to be made between the two paths, since there is a union operator separating the two sub regular expressions. After this, we have no choice to be made. We need to follow only previously selected path.

Example 3.6: For the regular expression 0(0*) + 1(1*), there are two paths. Suppose a path starting with 0 is chosen. After this only the iterative part is to be traversed, depending upon the number of iterations to be traversed.

The normal form means making the decision at the beginning, because the first operation (and only this) is an alternative. After this, what is to be decided is only the number of iterations given on the chosen branch.

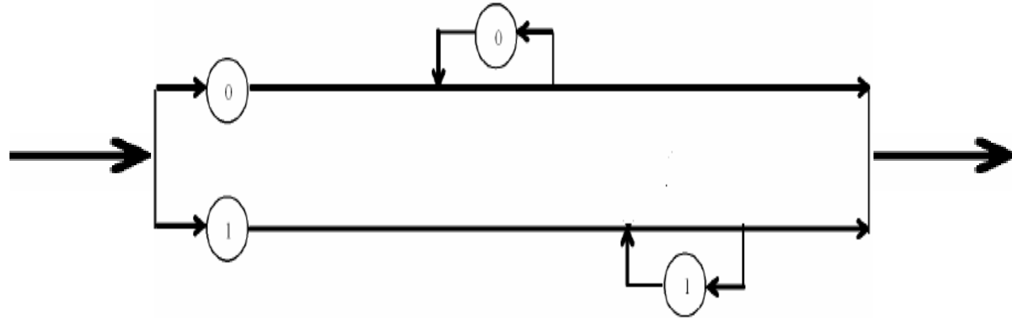


Figure 3.7: Syntax graph for the regular expression 0(0*) + 1(1*) [17].

The union-complexity of the language means how complex this first decision can be. Since, in the graph, the decision is to be taken at the first step, hence the union complexity is 1. The union-complexity gives a measure of the parallelism built in the regular expressions.

Proposition 4.1: The union-complexity of a language is one if and only if it is union-free regular [17].

3.4.3 Union Complexity and Finite Automata

It is well known that regular languages can be accepted by non-deterministic finite automata. There are several non deterministic finite automata accepting the same language. A non-deterministic finite automata accepting a language can have more than one final state. From the example 3.7 some results concerning the union-complexity of languages and the number of final states of their finite automata are discussed. The automata for the regular expression a*(ba)*ba is represented by the figure 3.8. In this automata, 0 and 2 are final states. As the regular expression is union-free, hence the union complexity is 1. So there is no relation between the union complexity and the final states of an automata [17].

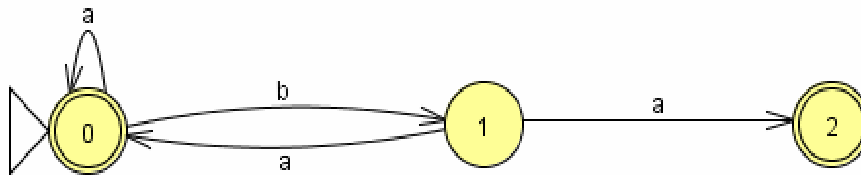


Figure 3.8: Representation of automata with one final state

Example 3.7: Let L be a finite regular language, L= {a, ab}.It is clearly evident from the corresponding automata in the figure 3.9, that there are two final states. The number of

final states is more than the union complexity which is one for the language L.

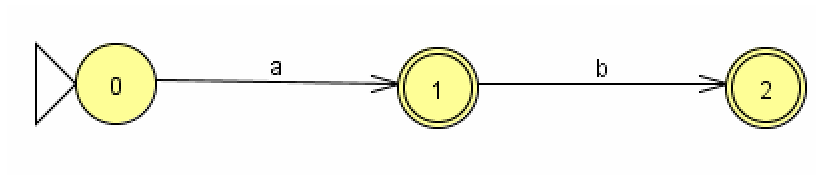


Figure 3.9: Representation of automata with two final states

Hence, it is clearly seen, from the examples that there is no relation between the number of final states and the union complexity of a regular language.

Chapter 4
Proposed Solution

This chapter gives the description of the algorithm proposed for determining whether the given regular expression is union-free or not. Furthermore, a tool has been designed in .NET which takes the regular expression as input and gives the output whether the regular expression entered is union-free or not. In addition to that, another algorithm has been proposed, which converts a regular expression into the union-free regular expression. The same is implemented in .NET, which converts a non-union-free regular expression into an equivalent union-free regular expression.

4.1 Algorithm 1: Union-freeness of a Regular Expression(r)

Given regular expression r, the algorithm determines, whether the language is union-free or not. Position will points to current character of regular expression. Initially points to rightmost symbol of regular expression.

1. Counter =0 and Plus=0
2. Scan symbol from right to left until leftmost symbol is scanned

```
If Counter=0 then
    If symbol[Position]= star operator
        Position= Position-1
        If (symbol [position]= right parenthesis)
            Counter= Counter +1
             Position= Position-1
        Endif
    Else
        Position= Position-1
    Endif
Else
    If  symbol[Position]=left parenthesis
        Counter= Counter-1
```

```
            Position= Position-1
        Else If (symbol [Position]= right parenthesis)
               Counter= Counter +1
               Position= Position-1
             Else If (symbol [Position]= plus)
                     Print regular expression is not union-free
                      Exit
                  Else
                       Position= Position-1
                 Endif
             Endif
        Endif
    Endif
3.  If Position= -1
       Print regular expression is union-free
    Endif
```

Example 4.1: Scan the regular expression (a(cb)*+a*)ab from right to left while star operator occurs (4th position from right). Right parenthesis will not occur at 5th position, hence counter remains zero. Next star operator occurs at 7th position from right and right parenthesis occurs at 8th position from right. Counter is set to one, but no plus sign will appear from 9th position from right to leftmost position. Hence it is a union-free regular expression.

Example 4.2: For the regular expression (ab+c)(a+cd), the algorithm scans it from right to left. The algorithm finds the first occurrence of the star operator. The algorithm finds no occurrence of the star operator. Hence the regular expression is union-free.

Example 4.3: For the regular expression (a+b)*ab, the algorithm scans it from right to left. The algorithm finds the first occurrence of the star operator which is at 3rd position. After this, the algorithm finds the right parenthesis at 4th position. Counter is set to one. Then algorithm finds a plus operator which appears at 6th position. Since, counter is more than one and there is occurrence of a plus sign. Hence the regular expression is not union-free.

It is not a simple task to recognize a union-free language by looking at the automata. In order to find out whether the language corresponding to automata is union free or not, the automata can be converted to the corresponding regular expression.

4.2 Algorithm 2: Union-free decomposition of a regular expression r into an equivalent union-free regular expression.

Given regular expression r, the algorithm decomposes the given regular expression into an equivalent union-free regular expression. Initially it points to rightmost symbol of regular expression.

1. Input the regular expression to be decomposed.
2. Call algorithm 3.
3. If the regular expression is union-free then go to step 7.
4. Else
   4.1. Consider the pair of braces due to which the regular expression is union-free.
   4.2. Check the number of components in the sub regular expression.
   4.3. Put all the components in the pair of left and right parenthesis.
   4.4. Replace all the plus operators with star operators.
   4.5. Put the above obtained regular expression into the main pair of parenthesis.
5. Go to step 1 for again scanning the regular expression, for checking if any non union-free sub regular expression occurs.
6. If the end of the regular expression occurs then go to step 7.
7. exit.

Generalized form of regular expression is represented by $(r_2+r_3+r_4+...r_n)^*r_1(r_2+r_3+r_4+...r_n)^*r_1'(r_2+r_3+r_4+...r_n)^*$ where r_2, r_3... can be any combination of star operator, concatenation operator and one or more alphabets and r_1, r_1' ... can be any combination of star operator, concatenation operator and zero or more alphabets

4.3 Algorithm 3: To convert the regular expression r into generalized form

Given regular expression r, the following algorithm converts the regular expression r into generalized form of $(r_1+r_2+r_3+\ldots.r_n)^*$ in order to convert it into an equivalent union-free

regular expression by Algorithm 2. Position points to current character of regular expression. Initially, algorithm points to left most symbol of regular expression. Position [terminal], position [left parenthesis], position [right parenthesis], position [star operator] gives the positions of a terminal, left parenthesis, fight parenthesis, star operator respectively. Algorithm divides the given regular expression into following four procedures which are based on the relative positions of symbols in regular expression. Then, algorithm calls the procedures according to positions of symbols.

1. Scan the regular expression from left to right.
2. Case 1: If position [terminal] +1 = position [left parenthesis] //eg. $(r_1+ r_2(r_3+r_4)+r_5)^*$

    B11= position [left parenthesis]

    Repeat till position != position [plus operator]

    position= position-1

    R1= position // for position 5 (ab*+b*c(ab+c))*, R1 = 5
3. Call Procedure 1(r, position)
4. Case 2: If position [terminal] -1 = position [right parenthesis] //eg.$(r_1+ (r_2+r_3)r_4+r_5)^*$

    B12= position [right parenthesis]

    Repeat till position != position [plus operator]

    position= position+1

    R2= position // for position 13 (ab*+(ab+c)bc+ac)*, R2 = 14, B12= 11
5. Call Procedure 2(r, position)
6. Case 3: If position [right parenthesis] +1 = position [left parenthesis] //$((r_1+ r_2)(r_3+r_4))^*$

    B11= position [right parenthesis]

    B12= position [right parenthesis] +1
7. Call Procedure 3(r, position)
8. Case 4: If position [star operator] +1 = position [right parenthesis] // $((r_1+ r_2^*(r_3+r_4))^*$

    B11 = position[star operator] +1

    Repeat till position != position [plus operator]

    position= position-1

    R1= position // for position 4 (ab*+ b*(ab+c))*, R1 = 5
9. Call Procedure 4(r, position)
10. Call algorithm 3.

Procedure 1

```
1 Push symbols on stack in direction left to right starting from position B11.
2 Repeat following two steps until counter2 becomes 0
      If (symbol [position] = left parenthesis)
        counter2 = counter2 + 1
        position= position + 1
      If (symbol [position] = right parenthesis)
        counter2 = counter2 - 1
        position= position + 1
3 B12 = position [right parenthesis at top of stack].
4 Call algorithm 1 for sub regular expression, between positions B11 and B12 +1.
5 If the output is not union-free,
      go to step 1 of Algorithm 3. Initialize all variables to zero, scan the RE from the
      position B12 + 1 and empty the stack.
  End if.
  Else
    If B12 + 1 = position [terminal]
        Pop symbols form the stack.
        Repeat for i = 1 to n
            P1i = position [plus operators popped]
        Repeat following two step till position !=B12
                    Put all the symbols (between R1 and B11), at position between
                    B11 and B11+1.
            Put all the symbols (between R1 and B11), after all P1i.
    End if
    Else
        Pop symbols form the stack.
        Repeat for i = 1 to n
            P1i = position [plus operators popped]
        Repeat following two step till position !=B12
                    Put all the symbols (between R1 and B11), at position B11 and
                    remove left parenthesis at B11.
```

Put all the symbols (between R1 and B11), after all P1is.

Remove right parenthesis at position B12.

6 Initialize all variables to zero, empty the stack.

7 Call algorithm 3.

8 Exit.

Procedure 2

1 Push symbols on the stack in direction right to left starting from position B12.

2 Repeat following two steps until counter2 becomes 0

If (symbol [position] = right parenthesis)

counter2 = counter2 + 1

position= position + 1

If (symbol [position] = left parenthesis)

counter2 = counter2 - 1

position= position + 1

3 B11= left parenthesis [at the top of stack]

4 Pop symbols form the stack.

Repeat for i = 1 to n till

P1i = position [plus operators popped]

5 Remove the right parenthesis at B12, replace it with the symbols (between positions B12 and R2.

6 Put the symbols (between positions B12 and R2) before the symbols at P1is.

7 Initialize all variables to zero, empty the stack.

8 Call algorithm 3.

9 Exit.

Procedure 3

1 Push symbols on the stack in direction left to right starting from position B12.

2 Repeat following two steps until counter2 becomes 0

If (symbol [position] = left parenthesis)

counter2 = counter2 + 1

position= position + 1

If (symbol [position] = right parenthesis)

counter2 = counter2 - 1

position= position + 1

3 B121 = position [right parenthesis at top of stack]

4 If B121 + 1= position [star operator]

5 Call algorithm 1 for sub regular expression, between positions B11 and B12 +1.

6 If the output is not union-free,

Go to step 1 of Algorithm xyz. Initialize all variables to zero, scan the RE from the position B12 + 1 and empty the stack.

7 8 Pop symbols form the stack (used in .previous step)

Repeat for i = 1 to n till

P2i = position [plus operators popped]

8 Repeat till position ! = B11

Push the symbols on another stack in the direction right to left

Repeat following two steps until counter2 becomes 0

If (symbol [position] = right parenthesis)

counter2 = counter2 + 1

position= position + 1

If (symbol [position] = left parenthesis)

counter2 = counter2 - 1

position= position + 1

B111=position [left parenthesis at the top of the stack].

9 Pop symbols form the stack (used in .previous step)

Repeat for i = 1 to n till

P1i = position [plus operators popped]

10 Repeat till position ! = B111

position=position-1

11 Put all the symbols (between B111 and B11 separated by P1is) at B12 and one positions next to P2is.

12 Remove parenthesis at B11 and B12 respectively.

13 Initialize all variables to zero, empty the stack.

14 Call algorithm 3.

15 Exit.

Procedure 4

1 Push symbols on stack in direction left to right starting from position B11.

2 Repeat following two steps until counter2 becomes 0

If (symbol [position] = left parenthesis)

counter2 = counter2 + 1

position= position + 1

If (symbol [position] = right parenthesis)

counter2 = counter2 - 1

position= position + 1

3 B12 = position [right parenthesis at top of stack].

4 Call algorithm 1 for sub regular expression, between positions B11 and B12 +1.

5 If the output is not union-free,

Go to step 1 of Algorithm xyz. Initialize all variables to zero, scan the RE from the position B12 + 1 and empty the stack.

End if.

Else

If B12 + 1 = terminal [position]

Pop symbols form the stack.

Repeat for i = 1 to n

P1i = position [plus operators popped]

Repeat following two step till position !=B12

Put all the symbols (between R1 and B11), at position between B11 and B11+1.

Put all the symbols (between R1 and B11), after all P1i.

End if

Else

Pop symbols form the stack.

Repeat for i = 1 to n

P1i = position [plus operators popped]

Repeat following two step till position !=B12

Put all the symbols (between R1 and B11), at position B11 and remove left parenthesis at B11.

Put all the symbols (between R1 and B11), after all P1is.

Remove right parenthesis at position B12.

6 Initialize all variables to zero, empty the stack.

7 Call algorithm 3.

8 Exit.

Example 4.4: For the regular expression (ab(a+c)+a)*, the algorithm detects the occurrence of a terminal b before a left parenthesis. Then it checks that whether the pair of parenthesis at the position 4th and 8th , are the reason for the non union-freeness of the regular expression or not. Since, the regular expression in the pair of parenthesis is union free. The algorithm replaces the left parenthesis at 4th position by ab and also after the plus operator at the 6th position, resulting into the regular expression (aba+abc+a)*. Then the algorithm again scans the regular expression for checking any possibility of concatenation. At last, the algorithm 3 converts the regular expression into the union-free regular expression, by placing the terminals into the ()* combination and finally resulting into the union-free regular expression ((aba)*(abc)*(a)*)*.

Example 4.5: For the regular expression ((a+c)ab*+c)*, the algorithm detects the occurrence of a terminal after the right parenthesis. Then it opens the inner pair of braces and concatenates the outer sub regular expression 'ab*', resulting into the regular expression (aab*+cab*+c)*. Then the algorithm again scans the regular expression for checking any possibility of concatenation. At last, the algorithm 3 converts the regular expression into the union-free regular expression, by placing the terminals into the ()* combination and finally resulting into the union-free regular expression ((aab*)*(cab*)*(c)*)*.

Example 4.6: For the regular expression ab(a+c)*a*+c, the algorithm detects the occurrence of a terminal at the position 3rd position before the left parenthesis. Then it checks whether the sub regular expression between the positions 4 and 9 is union-free or not. The sub regular expression is not union-free. Then algorithm 3 converts this sub regular expression into union-free regular expression, resulting into the decomposed union-free regular expression ab((a)*(c)*)*a*+c.

Example 4.7: For the regular expression ((a+b)(b+c))*, the algorithm detects the occurrence of a right parenthesis at the 6th position before the left parenthesis. Then it

checks whether the sub regular expression between the positions 7 and 11 is union-free or not. The sub regular expression is union-free. Then it simply the regular expression into the generalized form (ab+ac+bb+bc)*. After this, regular expression is converted into an equivalent union-free regular expression ((ab)*+(ac)*+(bb)*+(bc)*)* by algorithm 3.

4.4 Tool to determine whether the given regular expression is union-free or not

The tool for determining whether the given regular expression is union-free or not, is developed in .NET. After applying the logic for development of the software a GUI has been created. It takes the input from the user in the form of regular expression. The tool evaluates the input regular expression then it gives the output corresponding to the input entered by the user. The output shows, whether the regular expression is union-free or not union-free. For the generalized non-union-free regular expression, we convert the regular expression into an equivalent union-free regular expression. Following are the steps which are carried out to analyze the union-freeness of regular expression.

1. Enter the regular expression into the textbox which is in front of the label input expression.
2. Click on the Analysis button for evaluating the regular expression entered in first step.

Figure 4.1 shows the operation of entering a regular expression into the text box in front of the input expression.

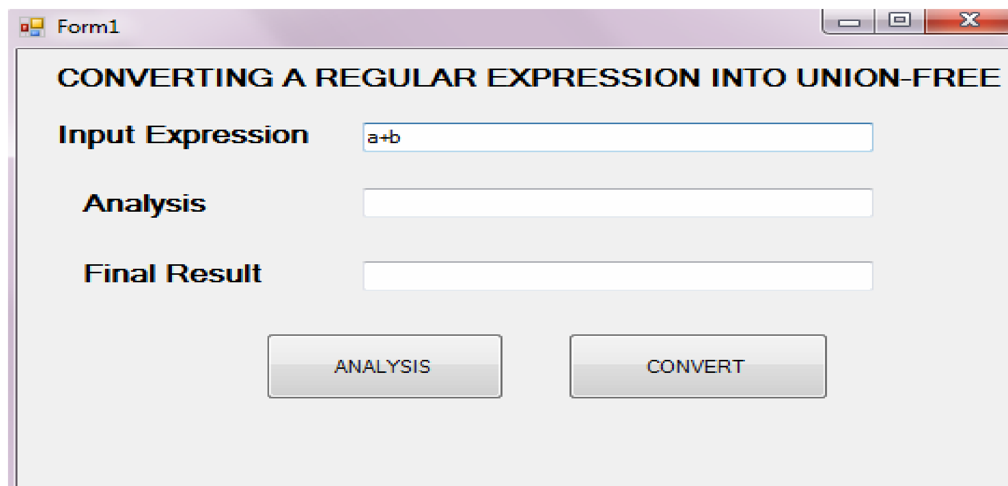


Figure 4.1: Entering a Regular Expression

For the regular r = a+b, we have found that it is union-free as shown in figure 4.2.

Form1

CONVERTING A REGULAR EXPRESSION INTO UNION-FREE

Input Expression a+b

Analysis Union Free

Final Result

ANALYSIS CONVERT

Figure 4.2: Operation analysis of the regular expression.

For a regular expression r = (a(ac)*+a*)ab, we have found that it is union-free as shown in figure 4.3.

Form1

CONVERTING A REGULAR EXPRESSION INTO UNION-FREE

Input Expression (a(ac)*+a*)ab

Analysis Union Free

Final Result

ANALYSIS CONVERT

Figure 4.3: Operation analysis of the regular expression.

For the regular expression r = ((ab+c)*+c)ab, we have found that it is union-free as shown in figure 4.4

Figure 4.4: Operation analysis of the regular expression.

Figure 4.5 represents the conversion of the regular expression ((ab+c)*+c)ab into a union free regular expression (((ab)*(c)*)*+c)ab.

Figure 4.5: Operation conversion of the entered regular expression.

For the regular expression r =((a+b)*+ab)(ab)*, we have found that it is not union-free as shown in figure 4.6.

Form1

CONVERTING A REGULAR EXPRESSION INTO UNION-FREE

Input Expression ((a+b)*+ab)(ab)*

Analysis NOT Union Free

Final Result

ANALYSIS CONVERT

Figure 4.6: Operation analysis of the entered regular expression.

Figure 4.7 represents the decomposition of regular expression ((a+b)*+ab)(ab)*, into an equivalent regular expression (((a)*(b)*)*+ab)(ab)*.

Form1

CONVERTING A REGULAR EXPRESSION INTO UNION-FREE

Input Expression ((a+b)*+ab)(ab)*

Analysis NOT Union Free

Final Result (((a)*(b)*)*+ab)(ab)*

ANALYSIS CONVERT

Figure 4.7: Operation conversion of the entered regular expression.

For the regular expression r =((ab+c+ba)* we have found that it is not union-free as shown in figure 4.8.

Figure 4.8: Operation analysis of the regular expression.

Figure 4.9 represents the decomposition of regular expression ((ab+c+ba)*, into an equivalent regular expression ((ab)*(c)*(ba)*)*.

Figure 4.9: Operation conversion of the regular expression.

For the regular expression r =(a+b)*ac(b+c)* we have found that it is not union-free as shown in figure 4.10.

Figure 4.10: Operation analysis of the regular expression.

Figure 4.11 represents the decomposition of regular expression (a+b)*ac(b+c)*, into an equivalent regular expression ((a)*(b)*)*ac((b)*(c)*)*.

Figure 4.11: Operation conversion of the regular expression.

For the regular expression r =(ab)*+b(ab+c)* we have found that it is not union-free as shown in figure 4.12.

Form1

CONVERTING A REGULAR EXPRESSION INTO UNION-FREE

Input Expression (ab)*+b(ab+c)*

Analysis NOT Union Free

Final Result

ANALYSIS CONVERT

Figure 4.12: Operation analysis of the regular expression.

Figure 4.13 represents the decomposition of regular expression=(ab)*+b(ab+c)* into an equivalent regular expression (ab)*b((ab)*(c)*)*.

Form1

CONVERTING A REGULAR EXPRESSION INTO UNION-FREE

Input Expression (ab)*+b(ab+c)*

Analysis NOT Union Free

Final Result (ab)*+b((ab)*(c)*)*

ANALYSIS CONVERT

Figure 4.13: Operation conversion of the regular expression

4.5 Size Relationship between Union-free and Non-union-free Regular Languages

Given a non-union-free regular language L, the regular expression corresponding to the language is R. Suppose the length of the sub regular expression L_{11}, due to which the non-union-freeness is occurring be 'm'. Length is calculated by counting the occurrences of terminals, star operators, union operators and concatenation operators.

For each of the above symbol encountered, the length of the sub regular expression is incremented by one. As stated above, length of the sub regular expression is 'm', the length of the equivalent regular expression which is obtained by the union-free decomposition of the regular expression L_{11} is 'm+1'.

Example 4.7: Given a regular expression (ab+cd+c)*. Length (m) of this regular expression comes out to be 8. The union-free decomposition of regular expression of regular expression is ((ab)*(cd)*c*)*. The length of the decomposed regular expression is 9. Which is equal to 8+1, means m+1.

Example 4.8: Given a regular expression ab*+c(c+d)*. Length (m) of this sub regular expression (c+d)* due to which the non-union-freeness is occurring comes out to be 4. The union-free decomposition of the above regular expression is ab*+(c*d*)*. The length of the sub regular expression after decomposition ((c)*(d)*)* is 5.

Example 4.9: Given a regular expression (a+b)*(c+d)*. Length (m) of two sub regular expressions ((a)*(b)*)* and ((c)*(d)*)*, due to which the non-union-freeness is occurring comes out to be 8. The union-free decomposition of the above regular expression is (a*b*)*(c*d*)*. The length of the sub regular expression after decomposition is 10.

Chapter 5

Conclusions and Future Work

In this thesis report, concept of union-free regular languages and union complexity is presented. An algorithm is proposed for determining whether a given regular language is union-free or not. A tool is designed for the same. Another algorithm and a tool are proposed for converting regular language into an equivalent union-free regular language. Following are some open problems which can be considered regarding the seminar report:

1. There is a scope of algorithm for minimal union free decomposition of a regular language.
2. Another open problem is concerned with the star height of a regular language. Given a star height of a regular language, is it possible to construct a minimal union-free decomposition that consists of languages of the same star height.
3. Development of a tool to convert any regular expression into an equivalent union-free regular expression.

References

1. Afonin, S., Khazova, E.: Membership and finiteness problems for rational sets of regular languages. International Journal of Foundations of Computer Science 17(3), 493–506, 2006.
2. Brzozowski J. A.: Canonical regular expressions and minimal state graphs for definite events Proc. Symp. on Mathematical Theory of Automata, Microwave Research Inst. Symp. Ser., Brooklyn, N. Y.: Polytechnic Press, 12, 529-561, 1962.
3. Brzozowski, J.: Open problems about regular languages. In: Book, R.V. (ed.) Formal Language Theory, CA, Univ. of CA at Santa Barbara, 23–47, Santa Barbara 1980.
4. Brzozowski, J., Cohen, R.: On decompositions of regular events. Journal of the ACM 16(1), 132–144, 1969.
5. Campeanu C., Culik K. II, Salomaa K., and Yu S.: State complexity of basic operations on finite languages. In Proceedings of WIA '99, Lecture Notes in Computer Science 2214, 60-70, 2001.
6. Ehrenfeucht, A., Zeiger, P.: Complexity measures for regular expressions. In: STOC 1974: Proceedings of the Sixth Annual ACM Symposium on Theory of Computing, ACM, 75–79, New York 1974.
7. Ellul K., Krawetz B., Shallit J., and Wang M.: New results and open problems. Journal of Automata, Languages and Combinatorics, 10(4), 407–437, 2005.
8. Hashiguchi, K.: Representation theorems on regular languages. Journal of Computer and System Sciences, 27, 101–115, 1983.
9. Hermann Gruber Stefan Gulan: Simplifying Regular Expressions: A Quantitative Perspective, IFIG Research Report 0904, 2009.
10. Hricko M., Jiraskova G., and Szabari A.: Union and intersection of regular languages and descriptional complexity. In Proceedings of DCFS'05, 170-181, 2005.
11. Leung, H., Podolskiy, V: The Limitedness problem on distance automata: Hashiguchi's method revisited. Theoretical Computer Science 310(1–3), 147–158, 2004.

12. Mcnaughton R. and Yamada H.: Regular expressions and state graphs for automata, IRE Trans. Electronic Computers, 9, 39-47, 1960.
13. Mishra K.L.P.& N. Chandrasekaran: Theory of Computer Science (Automata Language and. Computation), PHI, Second edition, 1998.
14. Nagy, B.: A normal form for regular expressions. In: Eighth International Conference on Developments in Language Theory, CDMTCS Technical Report 252, 51–60, Auckland 2004.
15. Nagy, B.: Union-free languages and 1-cycle-free-path-automata. Publicationes athematicae Debrecen 68, 183–197,2006.
16. Nagy, B: Syntactic Description of the Elements of the Programming Languages in a Normal Form Conference on Informatics in Higher Education, 2005.
17. Nagy, B.: On Union-complexity of Regular Languages.11th IEEE International Symposium on Computational Intelligence and Informatics, 18–20, November 2010.
18. Peter Linz: An introduction to formal languages and automata Narosa publishers fourth edition, 2009.
19. Paz, A., Peleg, B.: On concatenative decompositions of regular events. IEEE Transactions on Computers 17(3), 229–237, 1968.
20. Sergey Afonin and Denis Golomazov: On Minimal Union-Free Decompositions of Regular Languages. Third International conference, 83-92, Spain 2009.
21. Shyr and Yu : Finite regular component representable language is regular component splittable, Discrete Appl. Math. 82 -219, 1998.
22. Sinisa Crvenkovic, Igor Dolinka and Zoltan Esik: On Equations for Union-Free Regular Languages, Information and Computation 164, 152-172, 2001.
23. Yu S.: State complexity of regular languages. Journal of Automata, Languages and Combinatorics, 6(2), 221-234, 2001.
24. Yu S., Zhuang Q. and Salomaa K.: The state complexities of some basic operations on regular languages. Theoretical Computer Science, 125(2), 315-328, 1994.
25. Ullman, J., J. E. Hopcroft and R. Motwani: Introduction to Automata Theory, Languages, and Computation. Pearson Education Inc, ISBN 0-201-44124-1. Addison Wesley, 2001.

List of Publications

ACCEPTED

Sukhpal Singh Ghuman and Ajay Kumar, "Review Paper on Union-free Regular Languages", Paper ID ICCSE-23JU12-028, International Conference on Computer Science and Engineering (CSE) 24th June 2012, Ahmedabad.

COMMUNICATED

Sukhpal Singh Ghuman and Ajay Kumar, "Union-freeness of Regular Languages", International Journal of Computer Applications.