FEDERAL UNIVERSITY OF TECHNOLOGY, MINNA, NIGERIA

CENTRE FOR OPEN DISTANCE AND e-LEARNING (CODeL)

SCHOOL OF INFORMATION & COMMUNICATION TECHNOLOGY

COURSE TITLE: Automata, Computability And Complexity

COURSE CODE: CPT 323

© 2015

COURSE DEVELOPMENT

CPT 323

Automata, Computability and Complexity

Course Developer/Writers
Mrs. O.A. Abisoye
Dr. J.K. Alhassan
Department of Computer Science
School of Information & Communication Technology,
Federal University of Technology, Minna, Nigeria.

Course Editors
Dr. J.K. Alhassan
Department of Computer Science,
School of Information & Communication Technology,
Federal University of Technology, Minna, Nigeria.

Programme Coordinator
Mrs. Abisoye O.A.
Department of Information & Media Technology
School of Information & Communication Technology,
Federal University of Technology, Minna, Nigeria.

Instructional Designers
Dr. Gambari, Amosa Isiaka
Mr. Falode, Oluwole Caleb
Centre for Open Distance and e-Learning,
Federal University of Technology, Minna, Nigeria.

Editor
Mr. Azeez
Centre for Open Distance and e-Learning,
Federal University of Technology, Minna, Nigeria.

COURSE GUIDE

INTRODUCTION

CPT 323 Automata, Computability and Complexity is a 2 credit unit course for students studying towards acquiring a bachelor of technology in computer science and other related disciplines. The course is divided into 5 module and 10 study units. It will first take a brief review of the Automata Theory. This course will then go ahead to deal with the different classes of automata, the need for finite automata, types of Finite Automata. The course went further to explain how regular expression came about. The course also explains the concept of Computability and Complexity that will enable the reader have proper understanding of Automata, Computability and Complexity

The course guide therefore gives you an overview of what the course. CPT 323 is all about, the textbooks and other materials to be reference, what you expect to know in each unit, and how to work through the course material.

What you will learn in this course

The overall aim of this course, CPT 323 is to introduce you to basic concepts of c in order to enable you to understand the basic element of automata use in core Computer Science and especially its major role in compiler design and parsing.

The course defined the languages recognized by the automaton. In this course of your studies, you will be put through the definition of common terms in relation to Automata, Computability and Complexity theory

Course Aim

This course aim to introduce student to the basic concept of Automata, Computability and Complexity theory and appreciate decision processes especially for critical and cost involving (life, money, etc) endeavours. It will help the reader to understand the level of disparity between outcome and realty and why we require an automaton inn text processing, compilers, and hardware design in the field of artificial intelligence, Cellular automata, Quantum Physics and study of Human Language.

Course objectives

It is important to note that each unit has specific objective. Student should study them carefully before proceeding to subsequently unit. Therefore it may be useful to refer to these objectives in the course of your study of the unit to assess your progress. You should always look at the unit objective after completing a unit. In this way, you can be sure that you have done what is required of you by the end of the unit.

However, below are overall objective of this course. On completing this course, you should be able to:

i. State why is the need for Automata
ii. Define the language recognized by the automaton.
iii. Differentiate between deterministic and non-deterministic automata
iv. State their recognizable languages
v. State the application of automata to various fields of life.
vi. Relate automata with complexity
vii. Explain in detail about Deterministic Finite Automata.
viii. Explain the language of the DFA
ix. Discuss languages of NFA and
x. Explain the concept of Regular Expressions
xi. Discuss the concept of Moore and Mealy machine
xii. The concept of Turing machine
xiii. Explain Non Deterministic Turing machine
xiv. Explain Polynomial- Time Reductions

- Narrate Cook Levin Theorem
- Explain Time complexity
- Explain space complexity
- Explain Probabilistic complexity

Working through this Course

To complete this course, you are required to study all the units, the recommended text books, and other relevant materials. Each unit contains some self assessment exercises and tutor assignments, and at some point in these courses, you required to submit the tutor marked assignments. There is also final examination at the end of these courses. Stated below are the component of these courses and what you have to do.

Course materials

1. Course Guide
2. Study Units
3. Text Books
4. Assignment Files
5. Presentation Schedule

Study unit

There are 10 study unit and 5 modules in this course, They are:

MODULE 1: INTRODUCTION TO AUTOMATA

UNIT 1: AUTOMATA THEORY
UNIT 2: CLASSES OF AUTOMATA

MODULE 2: FINITE AUTOMATA

UNIT 1: INTRODUCTION TO FINITE AUTOMATA
UNIT 2: DETERMINISTIC FINITE AUTOMATA (DFA)
UNIT 3: NON-DETERMINISTIC FINITE AUTOMATA (NFA)

MODULE 3: REGULAR EXPRESSION

UNIT 1 CONCEPTS OF REGULAR EXPRESSION

MODULE 4: COMPUTABILITY THEORY

UNIT 1 TURING MACHINE

MODULE 5: COMPLEXITY THEORY

UNIT 1: TIME COMPLEXITY
UNIT 2: SPACE COMPLEXITY
UNIT 3: PROBABILITY COMPLEXITY

Recommended Texts

- http://cseweb.ucsd.edu/~ccalabro/essays/fsa.pdf
- http://www.hermann-gruber.com/data/icalp08.pdf
- http://dev.perl.org/perl6/doc/design/apo/A05.html.
- http://weblog.fortnow.com/2002/09/complexity-class-of-week-pp.html.
- http://www.scottaaronson.com/talks/postselect.ppt. Retrieved 2009-07-27.
- http://weblog.fortnow.com/2004/01/complexity-class-of-week-pp-by-guest.html.

Assignment File
The assignment file will be given to you in due course. In this file you will find all the detail of the work you must submit to your tutor for marks for marking. The marks you obtain for these assignments will count toward the final mark for the course. Altogether, there are tutor marked assignments for this course.
Presentation schedule
The presentation schedule included in this course guide provides you with importance date for completion of each tutor marked assignment. You should therefore endeavour to meet the deadline.
Assignment
There are two aspects to the assessment of this course. First there are tutor marked assignments: and second the written examination. Therefore you are expected to take note of the fact information and problem solving gathered during the course. The tutor marked assignment must be submitted to your tutor for formal assessments. In accordance to the deadline given the work submitted will count for 40% of your total course mark.
At the end of the course you will need to sit for a final written examination. This examination will account for 60%of your total score.
Tutor Marked Assignment (TMAs)
There are TMAs in this course; you need to submit all the TMAs. The best 10 will therefore be counted. When you have completed each assignment, send them to your tutor as soon as possible and make certain that it gets to your tutor on or before the stipulated deadline. If for any reason you cannot complete your assignment on time, contact your tutor before the assignment is due to discuss the possible of extension. Extension will not be granted after the deadline unless on extraordinary cases.
Final Examination and Grading
Final examination for CPT 323 will be of the last with period of 2hours and have a values 60% of the total course grade. The examination will consist of questions which reflect the self assessment exercise and tutor marked assignment s that you have previously encountered. Furthermore, all areas of the course will be examined. It would be better to use the time between finishing the last unit and sitting for the examination, to revise the entire course. You might find it useful to review your TMAs and comment on them before the examination. The final examination covers information from all parts of the course.
The following are practical strategies for working through this course.

1. Read the course guide thoroughly
2. Organize a study schedule. Refer to the course overview for more details. Note the time you are expected to spend on each unit and how the assignment relates to the units. Important details, e.g. details of your tutorials and the date of the first day of the semester are available. You need to gather all these information in one place such as a diary, a wall chart calendar or an organizer. Whatever method you choose, you should decide on and write in your own dates for working on each unit.
3. Once you have created your own study schedule, do everything you can to stick to it. The major reason that students fail that they get behind with their course works. If you get into difficulties with your schedule, please let your tutor know before it is too late for help.
4. Turn to unit 1 and read the introduction and the objectives for the unit.
5. Assemble the study materials. Information about what you need for a unit is given in the table of content at the beginning of each unit. You will almost always need both the study unit you are working on and one of the materials recommended for further readings, on your desk at the same time.
6. Work through the unit, the content of the unit itself has been arranged to provide a sequence for you to follow. As you work through the unit, you will be encouraged to read from your set books.
7. Keep in mind that you will learn a lot by doing all your assignments carefully. They have been designed to help you meet the objectives of the course and will help you pas the examination.
8. Review the objectives of each study unit to confirm that you have achieved them. If you are not certain about any of the objectives, review the study material and consult your tutor.
9. When you are confident that you have achieved a unit’s objectives, you can start on the next unit. Proceed unit by unit through the course and try to place your study so that you can keep yourself on schedule.

10. When you have submitted an assignment to your tutor for marking, do not wait for its return before starting on the next unit. Keep to your schedule. When the assignment is returned, pay particular attention to your tutor's comments, both on the tutor marked assignment form and also written on the assignment. Consult your tutor as soon as possible if you have and questions or problems.
11. After completing the last unit, review the course and prepare yourself for the final examination. Check that you have achieved the unit objectives (listed at the beginning of each unit) ant the course objectives (listed in this course guide).

Tutors and Tutorials

There are 8 hours of tutorial provide in support of this course. You will be notified of the dates, time and location together with the name and phone number of your tutor as soon as you are allocated a tutorial group. Your tour tutor will mark and comment on your assignments, keep a close watch on your progress and on any difficulties you might encounter and provide assistance to you during the course. You must mail your tutor marked assignment to your tutor well before the due date. At least two working days are required for this purpose. They will be marked by your tutor and returned to you as soon as possible.

Don not hesitates to contact your tutor by telephone, e-mail or discussion board if you need help. The following might be circumstances in which you would find help necessary:

- You don't not understand any part of the study units or the assigned readings;
- You have difficulty with the self test or exercise.
- You have questions or problems with an assignment, with your tutor's comments on an assignment or with the grading of an assignment.

You should endeavour to attend the tutorials. This is the only opportunity to have face to face contract with your tutor and ask questions which are answered instantly. You can raise any problem encountered in the course of your study. To gain the maximum benefit from the course tutorials, have some questions handy before attending them. You will learn a lot from participating actively in discussions.

Good luck!

Module 1

Introduction to Automata

Unit 1 Automata theory
Unit 2 Classes of Automata

Unit 1

Automata Theory

Content

1.0 Introduction
2.0 Learning Outcomes
3.0 Learning Activities
    3.1 Automata theory
    3.2 Automata
        3.2.1 Informal Description
3.2.2 Formal Description
    3.3 Attributes of automata
        3.3.1 Input
        3.3.2 States
        3.3.3 Transition function
        3.3.4 Acceptance condition
4.0 Conclusion
5.0 Summary
6.0 Tutor Marked Assignments and Marking Scheme
7.0 References/ Further readings

1.0 Introduction

The study of automata is an important part of the core of Computer Science. From our previous course CPT 312 we have been exposed to the need of Programming Language Translation. We have explored the enormous essence of translating formal language into the language computer understands. But the study of automata is a vital issue that cannot be left out as it plays a major role in compiler design and parsing.

In this study unit you will learn the need for the study of automata, the meaning of an automata and automaton, formal and informal description of an automaton, define the attributes of an automata such as input, alphabets, word, and transition function, differentiate between final states and accepting states, define the language recognized by the automaton.

2.0 Learning Objectives for Module 1 Study Unit 1

At the end of this unit, you should be able to:

I. State why is the need for the study of Automata theory

II. Define an automaton and automata

III. Differentiate between automaton and automata

IV. Differentiate between final states and accepting states,

V. Differentiate between deterministic and non-deterministic automata

VI. Define the language recognized by the automaton.

3.0 Learning Activities

3.1 Automata theory

In Theoretical Computer Science, **automata theory** is the study of abstract machines (or more appropriately, abstract 'mathematical' machines or systems) and the computational problems that can be solved using these machines. These abstract machines are called automata.

The study of the mathematical properties of automata is automata theory.

A Finite State Machine is known as **automaton**. This automaton consists of states (represented in the figure by circles), and transitions (represented by arrows). As the automaton sees a symbol of input, it makes a *transition* (or *jump*) to another state, according to its *transition function* (which takes the current state and the recent symbol as its inputs).

Automata theory is also closely related to formal language theory, as the automata are often classified by the class of formal languages they are able to recognize. An automaton can be a finite representation of a formal language that may be an infinite set.

Automata play a major role in compiler design and parsing.

Self Assessment Exercise (SAE 1)

What is the purpose of studying automata theory?

Self-Assessment Answer

Answer: Automata theory is the study of abstract computing devices or machines (automata). Automata play a major role in compiler design and parsing.

3.2 Automata

Following is an introductory definition of one type of automata, which attempts to help one grasp the essential concepts involved in automata theory.

3.2.1 Informal Description

An automaton is supposed to *run* on some given sequence or string of *inputs* in discrete time steps. At each time step, an automaton gets one input that is picked up from a set of *symbols* or *letters*, which is called an *alphabet*. At any time, the symbols so far fed to the automaton as input form a finite sequence of symbols, which is called a *word*. An automaton contains a finite set of states. At each instance in time of some run, automaton is *in* one of its states.

At each time step when the automaton reads a symbol, it *jumps* or *transits* to next state depending on its current state and on the symbol currently read. This function in terms of the current state and input symbol is called *transition function*. The automaton *reads* the input word one symbol after another in the sequence and transits from state to state according to the transition function, until the word is read completely.

Once the input word has been read, the automaton is said to have been *stopped* and the state at which automaton has stopped is called *final state*. Depending on the final state, it's said that the automaton either *accepts* or *rejects* an input word. There is a subset of states of the automaton, which is defined as the set of *accepting states*. If the final state is an accepting state, then the automaton *accepts* the word. Otherwise, the

word is *rejected*. The set of all the words accepted by an automaton is called the *language recognized by the automaton*.

Self-Assessment Exercise(s) (SAE 2)

Differentiate between final state and accepting state.

Self-Assessment Answer

Answer: Once the input word has been read, the automaton is said to have been *stopped* and the state at which automaton has stopped is called *final state* while accepting state is the state at which the automaton either *accepts* or *rejects* an input word depending on the final state.

3.2.2 Formal Definition

Automaton

An **automaton** is represented formally by the 5-tuple $\langle Q,\Sigma,\delta,q_0,A\rangle$, where:

i. Q is a finite set of *states*.

ii. Σ is a finite set of *symbols*, called the *alphabet* of the automaton.

iii. δ is the **transition function**, that is, $\delta: Q \times \Sigma \rightarrow Q$.

iv. q_0 is the *start state*, that is, the state which the automaton is *in* when no input has been processed yet, where $q_0 \in Q$.

v. A is a set of states of Q (i.e. $A \subseteq Q$) called **accept states**.

Input word

An automaton reads a finite string of symbols $a_1,a_2,....,a_n$, where $a_i \in \Sigma$, which is called a *input word*. Set of all words is denoted by Σ^*.

Run

A *run* of the automaton on an input word $w = a_1,a_2,....,a_n \in \Sigma^*$, is a sequence of states $q_0,q_1,q_2,....,q_n$, where $q_i \in Q$ such that q_0 is the start state and $q_i = \delta(q_{i-1},a_i)$ for $0 < i \leq n$. In words, at first the automaton is at the start state q_0, and then the automaton reads symbols of the input word in sequence. When the automaton reads symbol a_i it jumps to state $q_i = \delta(q_{i-1},a_i)$. q_n is said to be the *final state* of the run.

Accepting word

A word $w \in \Sigma^*$ is accepted by the automaton if $q_n \in A$.

Recognized language

An automaton can recognize a formal language. The recognized language $L \subset \Sigma^*$ by an automaton is the set of all the words that are accepted by the automaton.

Recognizable languages

The recognizable languages are the set of languages that are recognized by some automaton. For the above definition of automata the recognizable languages are **regular languages**. For different definitions of automata, the recognizable languages are different.

Self-Assessment Exercise(s) (SAE 3)

Which class of formal languages is recognizable by some type of automata?

Answer: The recognized language $L \subset \Sigma^*$ by an automaton is the set of all the words that are accepted by the automaton. The recognizable languages are called **regular languages**.

3.3 Attributes of automata

Automata are defined to study useful machines under mathematical formalism. So, the definition of an automaton is open to variations according to the "real world machine", which we want to model using the automaton. People have studied many variations of automata. The most standard variant, which is described above, is called a deterministic finite automaton. The following are some popular attributes of automata.

3.3.1 Input

i. *Finite input*: An automaton that accepts only finite sequence of symbols. The above introductory definition only accepts finite words.

ii. *Infinite input*: An automaton that accepts infinite words (ω-words). Such automata are called *ω-automata*.

iii. *Tree word input*: The input may be a *tree of symbols* instead of sequence of symbols. In this case after reading each symbol, the automaton *reads* all the successor symbols in the input tree. It is said that the automaton *makes one copy* of itself for each successor and each such copy starts running on one of the

successor symbol from the state according to the transition relation of the automaton. Such an automaton is called tree automaton.

3.3.2 States

I. *Finite states*: An automaton that contains only a finite number of states. The above introductory definition describes automata with finite numbers of states.

II. *Infinite states*: An automaton that may not have a finite number of states, or even a countable number of states. For example, the quantum finite automaton or topological automaton has uncountable infinity of states.

III. *Stack memory*: An automaton may also contain some extra memory in the form of a stack in which symbols can be pushed and popped. This kind of automaton is called a *pushdown automaton*

3.3.3 Transition function

i. *Deterministic*: For a given current state and an input symbol, if an automaton can only jump to one and only one state then it is a *deterministic automaton*.

ii. *Nondeterministic*: An automaton that, after reading an input symbol, may jump into any of a number of states, as licensed by its transition relation. Notice that the term transition function is replaced by transition relation: The automaton *non-deterministically* decides to jump into one of the allowed choices. Such automaton are called *nondeterministic automaton*.

iii. *Alternation*: This idea is quite similar to tree automaton, but orthogonal. The automaton may run its *multiple copies* on the *same* next read symbol. Such automata are called *alternating automaton*. Acceptance condition must satisfy all runs of such *copies* to accept the input.

3.3.4 Acceptance condition

i. *Acceptance of finite words*: Same as described in the informal definition above.

ii. *Acceptance of infinite words*: an *omega automaton* (ω) cannot have final states, as infinite words never terminate. Rather, acceptance of the word is decided by looking at the infinite sequence of visited states during the run.

iii. *Probabilistic acceptance*: An automaton need not strictly accept or reject an input. It may accept the input with some probability between zero and one. For example, quantum finite automaton, geometric automaton and *metric automaton* has probabilistic acceptance.

Different combinations of the above variations produce many variety of automaton.

Self-Assessment Exercise(s) (SAE 4)

Differentiate between deterministic and non-deterministic automata.

Self-Assessment Answer

Answer:
For a given current state and an input symbol, if an automaton can only jump to one and only one state then it is a deterministic automaton while an automaton that, after reading an input symbol, may jump into any of a number of states, as licensed by its transition relation are called nondeterministic automaton

4.0 Conclusion

In this unit you have got the overview of automata, the essence of studying automata theory, the attributes of automata, the formal and informal description of automata; these serves as introduction to this course title Automata, Computability and Complexity.

5.0 Summary

In this Module 1Study Unit 1, the following aspects have been discussed:

1. Automata theory is the study of abstract machines (automata) and the computational problems that can be solved using this machine.
2. A finite state machine is known as automaton.
3. Formal and informal description of an automaton.
4. Define an input, alphabets, word, and transition function.
5. Differences between deterministic and non deterministic automata.
6. Differences between final states and accepting states.
7. Definition of the language recognized by the automaton.

6.0 Tutor Marked Assignments and Marking Scheme

1. Introduce the concept of automata to a lay man
2. Explain these informal descriptions of automata: input, word, alphabet, transition fuction, accepting state and final states.
3. Give the formal definition of automata.
4. What is a regular language?
5. Discuss four (4) attributes of an automaton

7.0 References/ Further readings

John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman (2000). Introduction to Automata Theory, Languages, and Computation (2nd Edition). Pearson Education. ISBN 0-20-44124-1

Paritosh K. Pandya, **Automata: Theory and Practice,** TIFR, Mumbai, India, Unversity of Trento, 24th May, 2005

Unit 2

Classes of Automata

1.0 Introduction

2.0 Learning Outcomes

3.0 Main Content

- 3.1 Types of Automata
- 3.2 Discrete, Continuous and Hybrid Automata
- 3.3 Applications of Automata

4.0 Conclusion

5.0 Summary

6.0 Tutor Marked Assignments and Marking Scheme

7.0 References/ Further readings

1.0 Introduction

From our previous Unit 1 we have said that a different automaton has its own different recognizable languages but the general name given to the recognizable languages are called regular languages. So the need to study these classes of automata is very essential as an automaton is concern.

So study unit describes the classes of automata and their recognizable languages, discrete, continuous and hybrid automata, various applications of automata in computer Science and as it relate to other field of Science and Engineering.

2.0 Learning Objectives for Module 1 Study Unit 2

At the end of this unit, you should be able to:

1. List the types of automata
2. State their recognizable languages
3. Differentiate between discrete and analogue automata
4. State the application of automata to various fields of life.

3.1 Types of Automata.

The following is a list of types of automata.

Automata	**Recognizable language**
Deterministic finite automata (DFA)	regular languages
Nondeterministic finite automata (NFA)	regular languages
Nondeterministic finite automata with ε-transitions (FND-ε or ε-NFA)	regular languages
Pushdown automata (PDA)	context-free languages
Linear bounded automata (LBA)	context-sensitive language
Turing machines	recursively enumerable languages
Timed automata	

Deterministic Büchi automata	ω-limit languages
Nondeterministic Büchi automata	ω-regular languages
Nondeterministic/Deterministic Rabin automata	ω-regular languages
Nondeterministic/Deterministic Street automata	ω-regular languages
Nondeterministic/Deterministic parity automata	ω-regular languages
Nondeterministic/Deterministic Muller automata	ω-regular languages

Self-Assessment Exercise(s) (SAE 1)

State the importance of finite automata in the field of Computer Science

Self-Assessment Answer(S)

Answer: Finite automata are used in text processing, compilers, and hardware design

3.2 Discrete, Continuous, and Hybrid Automata

Normally automata theory describes the states of abstract machines but there are analog automata or continuous automata or hybrid discrete-continuous automata, which use analog data, continuous time, or both.

Self-Assessment Exercise(s) (SAE 2)

Differentiate between discrete and analogue automata

Self-Assessment Answer

Answer: Analog automata uses analog data, continuous time, or both while discrete automata

3.3 Applications

Each model in automata theory plays an imporant role in several applied areas. Finite automata are used in text processing, compilers, and hardware design. Context-free grammar (CFGs) are used in programming languages and artificial intelligence. Originally, CFGs were used in the study of the human languages. Cellular automata are used in the field of biology, the most common example being John Conway's Game of Life.

Some other examples which could be explained using automata theory in biology include mollusk and pine cones growth and pigmentation patterns. Going further, a theory suggesting that the whole universe is computed by some sort of a discrete automaton, is advocated by some scientists. The idea originated in the work of Konrad Zuse, most importantly his 1969 book Rechnender Raum, and gives rise to Digital physics.

Self Assessment Exercise (SAE 3)

What type of language is used in designing programming languages and artificial intelligence.

Self-Assessment Answer

Answer: Context Free Language(Grammar)

Self-Assessment Exercise (SAE 4)

State the importance of Cellular automata in the field of Biological Science

Self-Assessment Answer

Answer: Mollusk and pine cones growth and pigmentation patterns

4.0 Conclusion

In this Unit you have studied the classes of automata and their different recognizable language. Also you have been exposed to the difference between analog and discrete automata and finally studied the applications of the type of automata to different fields of Computer science, Biological science and Engineering technology.

5.0 Summary

In this Module 1Study Unit 2, the following aspects have been discussed:

1. The types of automata and their recognizable languages
2. Differences between discrete and analogue automata
3. Applications of automata to various fields of life.

6.0 Tutor Marked Assignments and Marking Scheme

1. State the application of automata to various fields of life.
2. List the types of automata you know and state their recognizable languages
3. How digital Physics does came about?

7.0 References/ Further readings

John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman (2000). Introduction to Automata Theory, Languages, and Computation (2nd Edition). Pearson Education. ISBN 0-201-44124-1

Paritosh K. Pandya, **Automata: Theory and Practice,** TIFR, Mumbai, India, Unversity of Trento, 24th May, 2005

Module 2

Finite Automata

Unit 1: Introduction to Finite Automata

Unit 2: Deterministic Finite Automata (DFA)

Unit 3: Non Deterministic Finite Automata (NFA)

Unit 1

Introduction to Finite Automata

1.0 Introduction
2.0 Learning Outcomes/Objectives
3.0 Learning Activities
    3.1 Structural Representation
        3.1.1 Grammar
        3.1.2 Regular Expressions
    3.2 Automata and Complexity
    3.3 Alphabets
        3.3.1 Powers of an Alphabet
    3.4 Strings
        3.4.1 Empty String
        3.4.2 Length of a string
        3.4.3 Concatenation
    3.5 Languages
    3.6 Problems
4.0 Conclusion
5.0 Summary
6.0 Tutor Marked Assignments and Marking Scheme
7.0 References/ Further readings

Finite automata are a useful model for many important kinds of hardware and software. Let us list some of the most important kinds:

1. Software for designing and checking the behavior of digital circuits.
2. The "lexical analyzer" of a typical compiler, that is, the compiler component that breaks the input text into logical units, such as identifiers, keywords and punctuations.
3. Software for scanning large bodies of text, such as collections of web pages, to find occurrences of words, phrases, or other patterns.
4. Software for verifying systems of all types that have a finite number of distinct states, such as communications protocols or protocols for secure exchange of information.

There are many systems or components, contains finite number of "states". The purpose of a state is to remember the relevant portion of the system's history. The advantage of having only a finite number of states is that we can implement the system with a fixed set of resources.

Example

The simplest finite automaton is an on/off switch. The device remembers whether it is the "on" state or the "off" state, and it allows the user to press a button whose effect is different, depending on the state of the switch. That is, if the switch is in the off state, then pressing the button changes it to the on state, and if the switch is in the on state, then pressing the same button turns it to the off state.

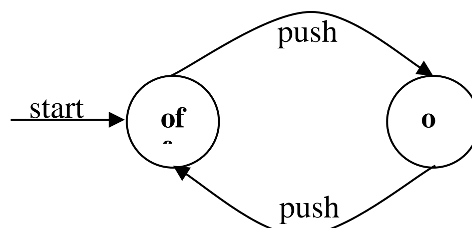


Fig 1: A finite automaton modeling an on/off switch

As for all finite automata, the states are represented by circles; in this example, we have named the states on and off. Arcs between states are labeled by "inputs", which represent external influences on the system. Here, both arcs are labeled by the input Push, which represents a user pushing the button. The intent of the two arcs is that whichever state the system is in, when the push input is received it goes to the other state. The state in which the system is placed initially is called as the "start state".

In our example, the start state is off, and we conveniently indicate the start state by the word Start and an arrow leading to that state.

2.0 Learning Outcomes

At the end of this unit, you should be able to:

1. State why is the need for Finite Automata
2. Explain the structural representation of automata
3. Relate automata with complexity
4. Explain briefly about Alphabets, Strings, Languages and Problems.
5. Define operations on strings and alphabets

3.1 Structural Representations

There are two important notations that are not automaton-like, but play an important role in the study of automata and their applications.

3.1.1 Grammars

1. These are useful models when designing software that processes data with a recursive structure.
2. The best known example is a "parser", the component of a compiler that deals with the recursively nested features of the typical programming language, such as expressions-arithmetic, conditional and so on.
3. For instance, a grammatical rule like $E \Rightarrow E + E$ states that an expression can be formed by taking any two expressions and connecting them by a plus sign; this rule is typical of how expressions of real programming languages are formed.

3.1.2 Regular Expressions

These also denote the structure of data, especially text strings. The style of these expressions differs significantly from that of grammars. The UNIX-style regular expression '[A-Z] [a-z] * [] [A-Z] [A-Z]' represents capitalized words followed by a space and two capital letters. This expression represents patterns in text that could be a city and state, e.g., Ithaca NY. It misses multiword city names, such as Palo alto CA, which could be captured by the more complex expression

‘([A-Z] [a-z] * [] [A-Z] [A-Z])’

When interpreting such expressions, we only need to know that [A-Z] represents a range of characters from capital “A” to capital “Z” and [] is used to represent the blank character alone. Also, the symbol * represents “any number of” the preceding expression. Parentheses are used to group components of the expression; they do not represent characters of the text described.

Self-Assessment Exercise(s) (SAE 1)

Discuss the application of finite automata to computer science

Self-Assessment Answer(S)

Answer: Mollusk and pine cones growth and pigmentation patterns

3.2 Automata and complexity

Automata are essential for the study of the limits of computation. There are two important issues:

i. What can a computer do at all? This study is called “decidability”, and the problems that can be solved by computer are called “decidable”.

ii. What can a computer do efficiently? This study is called “intractability”, and the problems that can be solved by a computer using no more time than some slowly growing function of the size of the input are called “tactable”.

The most important definitions include the “alphabet” (a set of finite symbols), “strings”(a list of symbols from an alphabet) and “language” (a set of strings from the same alphabet).

Self-Assessment Exercise(s) (SAE 2)

Differentiate between decidable problems and non-intractable problems.

Self-Assessment Answer(S)

The problems that can be solved by computer are called "decidable problems" while the problems that can be solved by a computer using no more time than some slowly growing function of the size of the input are called "tractable problems

3.3 Alphabets

An alphabet is a finite, nonempty set of symbols. We use the symbol ∑ for an alphabet. Common alphabets include:

1. ∑ = {0, 1}, the binary alphabet

2. ∑ = { a, b, ………..,z}, the set of all lower-case letters.

3. The set of all ASCII characters, or the set of all printable ASCII characters.

3.3.1 Powers of an Alphabet

If ∑ is an alphabet, the set of all strings of a certain length from that alphabet is expressed by using an exponential notation. We define $\sum^k$ to be the set of strings of length k, each of whose symbols is in ∑.

Eg: Note that $\sum^0 = \{\varepsilon\}$, regardless of what alphabet ∑ is. That is, ε is the only string whose length is 0.

If $\sum = \{0, 1\}$,

then $\sum^1 = \{0, 1\}$,

$\sum^2 = \{00,01,10,11\}$,

$\sum^3 = \{ 000, 001,010,011,100,101,110,111\}$ …..

There is a slight confusion between ∑ and $\sum^1$. The former is an alphabet; its members 0 and 1 are symbols. The latter is a set of strings; its members are the strings 0 and 1, each of which is of length 1.

4. The set of all strings over an alphabet Σ is denoted by Σ^*.

. That is $\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \ldots$

5. The set of nonempty strings from alphabet $\sum$ is denoted by $\sum^+$.

6. Thus the equivalences are

7. $\sum^+ = \sum^1 \cup \sum^2 \cup \sum^3 \cup \ldots.$

8. $\sum^* = \sum^+ \cup \{\varepsilon\}$

3.4 Strings

A string (or sometimes word) is a **finite sequence of symbol**s chosen from some alphabet. For eg., 01101 is a string from the binary alphabet $\sum = \{0, 1\}$. The string 111 is another string chosen from this alphabet.

3.4.1 The Empty String

The empty string is the string with zero occurrences of symbols. This string, denoted ε, is a string that may be chosen from any alphabet whatsoever.

3.4.2 Length of a String

It is often useful to classify strings by their length, that is, the number of positions for symbols in the string. For instance, 01101 has length 5. The standard notation for the length of a string w is $|w|$.

For eg., $|011| = 3$ and $|\varepsilon| = 0$.

3.4.3 Concatenation of Strings

Let x and y be strings. Then xy denote the concatenation of x and y, that is, the string formed **by making a copy of x and following it by a copy of y.** If x is the string composed of i symbols

$$x = a_1a_2\ldots\ldots\ldots\ldots a_i$$

and y is the string composed of j symbols

$$y = b_1b_2\ldots\ldots\ldots\ldots b_j,$$

then xy is the string of length i+j;

$$xy = a_1a_2\ldots\ldots a_ib_1b_2\ldots\ldots b_j.$$

Eg: Let x = 01101 and y = 110. Then xy =01101110 and yx = 11001101.

For any string w, the equations εw = wε = w

Self-Assessment Exercise(s) (SAE 3)

Fill the gap below with the correct answer and state the particular operation on it

i. |10111| = --------------------------

ii. ε^2 = ----------------------

iii. $\{0,1\}^*$ =----------------------

iv. **{ ε ,01,10,0011,0011,0101,1001,…………..}**

i. Answer: 5, Length of a string

ii. Answer: 2, Power of an alphabet

iii. Answer: {ε, 0, 1, 00, 01, 10, 11, 000, …. }, the set of all string over an alphabet 0's and 1's

iv. Answer: the set of all string with an equal no of each

3.5 Languages

i. A set of strings chosen from some ∑*, is called a language, denoted by L.

ii. The language can be expressed as $L \subseteq \sum^*$.

iii. For any programming language, the legal programs are a subset of the possible strings that can be formed from the alphabet of the language. This alphabet is a subset of the ASCII characters.

iv. For example consider the following languages:

i. The language of all strings consisting of n 0's followed by n 1's, for some n ≥ 0;

ii. {ε, 01,0011,000111,………}.

iii. The set of binary numbers whose value is a prime: {10, 11, 101, 111, 1011, ……….}

iv. ∑* is a language for any alphabet ∑.

v. ϕ, the empty language, is a language over any alphabet.

a. {ε}, the language consisting of only the empty string, is also a language over any alphabet. Notice that ϕ = {ε}; the former has no strings and the latter has one string.

3.6 Problems

In automata theory, a problem is the question of deciding whether a given string is a member of some particular language. A "problem" can be expressed as a membership in a language. If ∑ is an alphabet, and L is a language over ∑, then the problem L is:

i. Given a string w in ∑*, decide whether or not w is in L.

ii. **Eg:** The problem of testing can be expressed by the language L_p consisting of all binary strings whose value as a binary number is a prime. That is, given a string of 0's and 1's say "yes" if the string is the binary representation of a prime and say "no" if not.

iii. One potentially unsatisfactory aspect of our definition of "problem" is that one commonly thinks of problems not as decision questions but as requests to compute or transform some input. The technique of showing one problem hard by using its supposed efficient algorithm to solve efficiently another problem that is already known to be hard is called a "reduction" of the second problem to the first.

Self-Assessment Exercise(s) (SAE 4)

Given a string w in ∑*, decide whether or not w is in L

Self-Assessment Answer

Answer:

The problem of testing can be expressed by the language L_p consisting of all binary strings whose value as a binary number is a prime. That is, given a string of 0's and 1's say "yes" if the string is the binary representation of a prime and say "no" if not.

4.0 Conclusion

In this unit you have got the essence of studying automata for designing software for checking the behaviour of digital circuit, scanning large bodies of text and a lexical analyzer. Also, the representation of automata in

alphabets, strings, languages and problems were discussed; these serves as the central concept of finite automata in introductory aspect.

5.0 Summary

In this Module 2 Study Unit 1, the following aspects have been discussed:

i. The need of finite automata for modeling any kind of hardware and software

ii. The structural representation of automata, such as length of string, powers of an alphabet, concatenation of strings, and languages

iii. Define Operations on strings, alphabets and languages

iv. Relate automata with complexity

v. The concept of automata problem

6.0 Tutor Marked Assignment

i. Discuss two important notations that are not automaton-like, but play an important role in the study of automata and their applications.

ii. What is a string and state the standard notation for the set of string?

iii. Perform the three operations on string on these set of word 110001 and 1000001

iv. State the operations on the equation below and fill the gap with correct answer

   i. $\varepsilon^* =$ ------------------------

   ii. $\varepsilon^2 =$ ------------------------

v. How can you define a language over an alphabet ∑ ?

vi. The set of strings of 0's and 1's with an equal number of each is?

M. Mukund, Finite State Automata over Infinite Inputs,*Technical Report, TCS-96-2, Chenna Mathematical Institute*, 1996. (http://www.cmi.ac.in/techreps/html/tcs-96-2.html)

W. Thomas, Automata on Infinite Objects, in *Handbook of Theoretical Computer Science, Volume B*,North-Holland, 1990.

Unit 2

Deterministic Finite Automata (DFA)

1.0 Introduction

2.0 Learning Outcomes

3.0 Learning Activities

- 3.1 Formal Definitions of DFA
- 3.2 Languages of DFA
- 3.3 Simpler Notations for DFA
  - 3.3.1 Transition Diagram
  - 3.3.2 Transition Tables

4 Conclusion

5 Summary

6 Tutor Marked Assignments and Marking Scheme

7 References/ Further readings

1.0 Introduction

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. A deterministic finite automaton consists of:

1. A finite set of states, often denoted Q.
2. A finite set of input symbols, often denoted $\sum$.
3. A transition function that takes as arguments a state and an input symbol and returns a state.(δ)
4. A start state, one of the states in Q.(q_o)
5. A set of final or accepting states F. The set F is a subset of Q.

2.0 Learning Outcomes

At the end of this unit, you should be able to:

1. State the meaning of the word "deterministic"
2. Explain in detail about Deterministic Finite Automata.
3. Give the formal definition of deterministic automata(DFA)
4. Explain the language of the DFA

3.0 Learning Activities

3.1 Formal Definition of DFA

A Deterministic Finite Automaton will often be referred to by its acronym: DFA. DFA in "five-tuple" notation is given as,

$$A = (Q, \sum, \delta, q_o, F)$$

Where A is the name of the DFA, Q is its set of states, $\sum$ is its input symbols, δ its transition function, q_o its start state, and F its set of accepting states.

Self Assessment Exercise (SAE 1)

Discuss about Deterministic Finite Automata.

Self-Assessment Answer

Answer

A Deterministic Finite Automaton will often be referred to by its acronym: DFA. DFA in "five-tuple" notation is given as,

$$A = (Q, \sum, \delta, q_o, F)$$

Where A is the name of the DFA, Q is its set of states, $\sum$ is its input symbols, δ its transition function, q_o its start state, and F its set of accepting states.

3.2 Language of the DFA

1. The "language" of the DFA is the set of all strings that the DFA accepts.
2. Suppose $a_1,a_2 \ldots\ldots\ldots\ldots a_n$ is a sequence of input symbols. We start out with the DFA in its start state, q_o.
3. The transition function δ, say $\delta(q_o, a_1) = q_1$ to find the state that the DFA enters after processing the first input symbol a_1.
4. Then process the next input symbol a_2, by evaluating $\delta(q_1, a_2)$; continue in this manner, finding states $q_3,q_4 \ldots\ldots\ldots q_n$ such that $\delta(q_{i-1}, a_i) = q_i$ for each i.
5. If q_n is a member of F, then the input $a_1,a_2 \ldots\ldots\ldots\ldots a_n$ is accepted, and if not then it is "rejected".

Example:

Let us formally specify a DFA that accepts all and only the strings of 0's and 1's that have the sequence 01 somewhere in the string. We can write this language as:

L = {w / w is of the form x01y for some strings x and y consisting of 0's and 1's only }

Another equivalent description, using parameters x and y to the left of the vertical bar, is:

{x01y/x are any strings 0's and 1's}

Examples of strings in the language include 01, 11010, and 100011. Examples of strings not in the language include ε, 0 and 111000.

Self-Assessment Exercise(s) (SAE 2)

Define the set of string on the language L = {w / w is of the form x11y for some strings x and y consisting of 0's and 1's only}

Self-Assessment Answer(S)

Answer

{11, 0011, 1100, 0110, 0111, 01110, 01111………………..}

3.3 Simpler Notations for DFA's

Specifying a DFA as a five-tuple with a detailed description of the δ transition function is both tedious and hard to read. There are two preferred notations for describing automata:

1. A transition diagram which is a graph
2. A transition table, which is a tabular listing of the δ functions, which by implication tell us the set of states and the input alphabet.

3.2.1. Transition Diagrams:

A transition diagram for a DFA A = (Q, ∑, δ, q_0, F) is a graph defined as follows:

1. For each state in Q there is a node

2. For each state q in Q and each input symbol a in ∑, let δ(q, a) = p. then the transition diagram has an arc from node q to node p, labeled a. If there are several input symbols that cause transitions from q to p, then the transition diagram can have one arc, labeled by the list of these symbols.

3. There is an arrow into the start state q_o, labeled start. This arrow does not originate at any node.

4. Nodes corresponding to accepting states are marked by a double circle. States not in F have a single circle.

Example

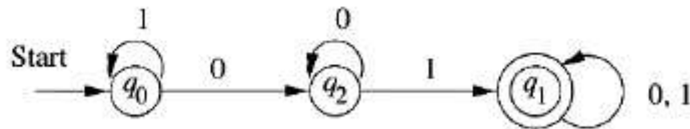


Fig 2: The transition diagram for the DFA accepting all strings with a substring 01

In the figure., given above the three nodes that corresponds to the three states. There is a start arrow entering the start state, q_o, and the one accepting state, q_1, is represented by a double circle. Out of each state is one arc labeled 0 and one arc labeled 1.

3.2.2 Transition tables

A transition table is a conventional, tabular representation of a function like δ that takes two arguments and returns a value. The rows of the table correspond to the states, and the columns correspond to the inputs. The entry for the row corresponding to state q and the column corresponding to input a is the state δ(q, a).

Example

The two features of a transition table marked below with the start state being marked with an arrow, and the accepting states are marked with a star. Since we can deduce the sets of states and input symbols by looking at the row and column heads, we can now read from the transition table all the information we need to specify the finite automaton uniquely.

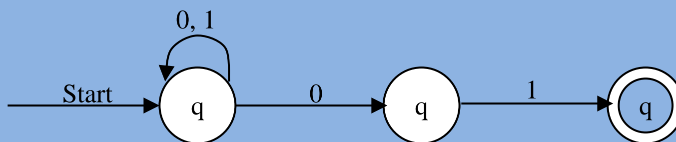
	0	1
→ q_0	q_2	q_0
* q_1	q_1	q_1
q_2	q_2	q_1

Table 1: The transition table for the DFA accepting all strings with a substring 01

Self Assessment Exercise(s) (SAE 3)

Draw the transition diagram of the transition table below

	0	1
→ q_0	q_2	q_0
* q_2	q_2	q_2
q_1	q_1	q_2



4.0 Conclusion

In this unit you have studied the concepts of DFA, the languages of DFA, the simpler notations for DFA in form of transition diagrams and transition tables; these serves as the representation of terms to be used for designing deterministic finite automata as we move ahead in this course.

5.0 Summary

In this Module 2 Study Unit 2, the following aspects have been discussed:

a. An automaton is a said to be deterministic if its transition from its current state tis one and only one state.

b. DFA is a "five-tuple" notation given as $A = (Q, \sum, \delta, q_o, F)$

c. Where A is the name of the DFA, Q is its set of states, $\sum$ is its input symbols, δ its transition function, q_o its start state, and F its set of accepting states.

d. The "language" of the DFA is the set of all strings that the DFA accepts.

e. The two simpler notations for DFA: transition diagram and transition table

6.0 Tutor Marked Assignments and Marking Scheme

i. When do we say that an automaton is deterministic?

ii. Explain the following with appropriate diagram

i. transition table

ii. transition diagram

iii. Define the set of string on the language L = {w / w is of the form x10y for some strings x and y consisting of 0's and 1's only }

7.0 References/ Further readings

Hopcroft, JE.; R. Motwani, JD Ullman (2001). *Introduction to Automata Theory, Languages and Computation*. Addison Wesley. ISBN 0-201-44124-1.

McCulloch, W. S.; Pitts, E. (1943). "A logical calculus of the ideas imminent in nervous activity". *Bulletin of Mathematical Biophysics*: 541–544.

Rabin, M. O.; Scott, D. (1959). "Finite automata and their decision problems.". *IBM J. Res. Develop.*: 114–125.

Michael Sipser, *Introduction to the Theory of Computation*. PWS, Boston. 1997. ISBN 0-534-94728-X. Section 1.1: Finite Automata, pp. 31–47. Subsection "Decidable Problems Concerning Regular Languages" of section 4.1: Decidable Languages, pp. 152–155.4.4 DFA can accept only regular language

Unit 3

Non Deterministic Finite Automata (NFA)

1.0 Introduction

2.0 Learning Outcomes/Objectives

3.0 learning Activities

- 3.1 Formal Definitions of NFA
- 3.2 The Extended transition function
- 3.3 Languages of NFA
- 3.4 Formal Notation for ϵ- DFA

4.0 Conclusion

5.0 Summary

6.0 Tutor Marked Assignments and Marking Scheme

7.0 References/ Further reading

A non-deterministic finite automaton (NFA) has the power to be in several states at once. This ability is often expressed as an ability to "guess" something about its input. For instance, when the automaton is used to search for certain sequences of characters (eg., keywords) in a long text string, it is helpful to "guess" that we are at the beginning of one of those strings and use a sequence of states to do nothing but check that the string appears, character by character.

An Informal view of NFA:

Like DFA, NFA has a finite set of states, a finite set of input symbols, one start state and a set of accepting states. It also has a transition function, which we shall commonly call δ. The difference between the DFA and NFA is in the type of δ. For the NFA, δ is a function that takes a state and input symbol as arguments, but returns a set of zero, one or more states.

Example

A non-deterministic finite automaton, whose job is to accept all and only the strings of 0's and 1's that end in 01. State q_0 is the start state and we think of the automaton as being in state q_0 whenever it has not yet "guessed" that the final 01 has begun. It is always possible that the next symbol does not begin the final 01, even if that symbol is 0. Thus, state q_0 may transition to itself on both 0 and 1.

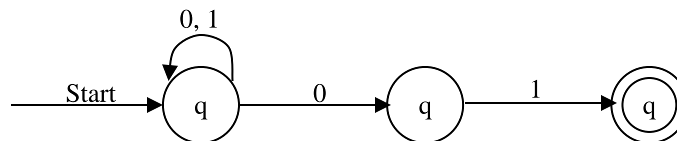


Fig (3) An NFA accepting all strings that end in 01

However, if the next symbol is 0, this NFA also guesses that the final 01 has begun. An arc labeled 0 thus leads from q_0 to state q_1. Notice that there are two arcs labeled 0 out of q_0. The NFA has the option of going either to q_0 or to q_1 and in fact it does both. In state q_1, the NFA checks that the next symbol is 1, and if so, it goes to state q_2 and accepts.

Notice that there is no arc out of q_1 labeled 0, and there are no arcs at all out of q_2. In these situations, the thread of the NFA's existence corresponding to those states simply "dies" although other threads may continue to exist.

Then, the second 0 is read. State q_0 may again go to both q_0 and q_1. However, state q_1 has no transition on 0, so it "dies". When the third input, a1, occurs, we must consider transitions from both q_0 and q_1. We find that q_0 goes only to q_0 on 1, while q_1 goes only to q_2. Thus, after reading 001, the NFA is in states q_0 and q_2. Since q_2 is an accepting state, the NFA accepts 001.However, the input is not finished. The fourth input, a0, causes q_2's thread to die, while q_0 goes to both q_0 and q_1.

The last input, a1, sends q_0 to q_0 and q_1 to q_2. Since we are again in an accepting state, 00101 is accepted.

***Please insert relevant images/graphics**

2.0 Learning Outcomes

At the end of this unit, you should be able to:

1. Give Formal Definitions of NFA
2. Explain the extended transition function
3. Discuss languages of NFA and
4. Explain formal Notation for ϵ- DFA

3.0 Learning Activities

3.1 Formal Definition of Nondeterministic Finite Automata:

An NFA is represented essentially like a DFA:

$$A = (Q, \Sigma, \delta, q_0, F)$$

Where:

1. Q is a finite set of states
2. Σ is a finite set of input symbols
3. q_0 a member of Q, is the start state
4. F, a subset of Q, is the set of final (or accepting) states.
5. δ, the transition function is a function that takes a state in Q and an input symbol in Σ as arguments and returns a subset of Q.

Example

The NFA of fig(a) can be specified formally as

$$(\{ q_0, q_1, q_2 \}, \{ 0,1 \}, \delta, q_0, \{ q_2 \})$$

where the transition function δ is given by the transition table as follows:

	0	**1**
$\rightarrow q_0$	$\{ q_0, q_1 \}$	$\{ q_0 \}$
q_1	ϕ	$\{ q_2 \}$
$* q_2$	ϕ	ϕ

Table 2: Transition Table for the transition function δ of fig 2

Notice that the transition tables can be used to specify the transition function for an NFA as well as for a DFA. The only difference is that each entry in the table for the NFA is a set, even if the set is a singleton (has one member). When there is no transition at all from a given state on a given input symbol, the proper entry is ϕ, the empty set.

Self Assessment Exercise (SAE 1)

Discuss about Non Deterministic Finite Automata.

Self-Assessment Answer

Answer

An NFA is represented essentially like a DFA: $A = (Q, \Sigma, \delta, q_0, F)$

Where: Q is a finite set of states,Σ is a finite set of input symbols, q_0 a member of Q, is the start state, F, a subset of Q, is the set of final (or accepting) states and δ, the transition function is a function that takes a state in Q and an input symbol in Σ as arguments and returns a subset of Q.

3.2 The Extended Transition Function

We need to extend the transition function δ of an NFA to a function **δ** that takes a state q and a string of input symbolsω, and returns the set of states that the NFA is in if it starts in state q and processes the stringω. **δ**(q,

ω) is the column of states found after reading ω, if q is the lone state in the first column. Formally we define $\hat{\delta}$ for an NFA's transition function δ by:

BASIS $\hat{\delta}(q, \varepsilon) = \{q\}$. That is, without reading any input symbols, we are in the state we began in.

INDUCTION Suppose **w** is of the form w = xa, where a is the final symbol of w and x is the rest of w. Also suppose that $\hat{\delta}(q, x) = \{p_1, p_2, \ldots\ldots p_k\}$. Let

$$\bigcup_{i=1}^{k} \delta(p_i, a) = \{r_1, r_2, \ldots\ldots\ldots\ldots r_m\}$$

Then $\hat{\delta}(q, w) = \{r_1, r_2, \ldots\ldots\ldots\ldots r_m\}$. Less formally, we compute $\hat{\delta}(q,w)$ by first computing $\hat{\delta}(q, x)$, and by then following any transition from any of these states that is labeled a.

Self-Assessment Exercise(s) (SAE 2)

What is the difference between DFA and NFA?

Self-Assessment Answer

Answer

The difference between the DFA and NFA is in the type of δ. For the NFA, δ is a function that takes a state and input symbol as arguments, but returns a set of zero, one or more states.

3.3 The Languages of an NFA

An NFA accepts a string ω if it is possible to make any sequence of choices of next state, while reading the characters of ω, and go from the start state to any accepting state. The fact that other choices using the input symbols of ω lead to a non accepting state, or do not lead to any state at all(i.e., the sequence of states "dies"), does not prevent ω from being accepted by the NFA as a whole. Formally, if $A = (Q, \Sigma, \delta, q_0, F)$ is an NFA, then

$$L(A) = \{w / \hat{\delta}(q_0, w) \cap F \neq \phi\}$$

That is, $L(A)$ is the set of strings ω in Σ* such that $\delta(q_0, \omega)$ contains at least one accepting state.

Every language that can be described by some NFA can also be described by some DFA. The DFA has about as many states as the NFA. Although it often has more transitions. In the worst case, however, the smallest DFA can have 2^n states while the smallest NFA for the same language has only n states.

The proof that DFA's can do whatever NFA's can do involves an important "construction" called the subset construction because it involves constructing all subsets of the set of states of the NFA.

The subset construction starts from an NFA $N = (Q_N, \Sigma, \delta_N, q_0, F_N)$. Its goal is the description of a DFA $D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ such that $L(D) = L(N)$. The input alphabets of the two automata are the same, and the start state of D is the set containing only the start state of N. The other components of D are constructed as follows.

a. Q_D is the set of subsets of Q_N ; i.e., Q_D is the power set of Q_N. Note that if Q_N has n states, then Q_D will have 2^n states. Note all these states are accessible from the start state of Q. Inaccessible states can be "thrown away", so effectively, the number of states of D may much smaller than 2^n.

b. F_D is the set of subsets S of Q_N such that $S \cap F_N \neq \phi$. That is, F_D is all sets of N's states that include at least one accepting state of N.

c. For each set $S \subseteq Q_N$ and for each input symbol a in Σ,

$$\delta_D(S, a) = \bigcup_{p \text{ in } S} \delta_N(p, a)$$

That is, to compute $\delta_D(S, a)$ we look at all the states p in S, see what states N goes from p on input a, and take the union of all those states.

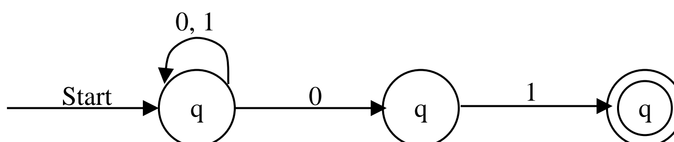


Fig 4: Transition diagram to compute $\delta_D(S, a)$

Example

Let N be the automaton of the above figure that accepts all strings that end in 01. Since N's set of states is $\{q_0, q_1, q_2\}$, the subset construction produces a DFA with $2^3 = 8$ states, corresponding to all the subsets of these three states. The following figure shows the transition table for these eight states.

This transition table belongs to a deterministic finite automaton. Even though the entries in the table are sets, the states of the constructed DFA are sets. We can invent new names for these states eg., A for ϕ, B for $\{q_0\}$ etc., The DFA transition table defines exactly the same automaton but makes clear the point that the entries in the table are single states of the DFA.

***Please the Figures should be numbered accordingly.**

	0	**1**
ϕ	ϕ	ϕ
$\rightarrow\{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_1\}$	ϕ	$\{q_2\}$
$*\{q_2\}$	ϕ	ϕ
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$

*{q_0, q_2}	{q_0, q_1}	{q_0}
*{q_1, q_2}	ϕ	{q_2}
*{q_0, q_1, q_2}	{q_0, q_1}	{q_0, q_2}

Subset Construction from NFA

Table 3: Transition table to compute δ_D(S, a) showing Subset Construction from NFA

Renaming the States

	0	**1**
A	A	A
→ B	**E**	**B**
C	A	D
*** D**	A	A
E	E	F
*** F**	**E**	**B**
*** G**	A	D
*** H**	E	F

Renaming the States

Renaming the States

Table 4: Transition table to compute δ_D(S, a) showing Renaming the of the States

Of the right states in table-2 starting in the start state B, we can only reach states B, E and F. The other five states are inaccessible from the start state. Those states are removed and the table is reconstructed and the DFA diagram is drawn with that table as shown in the figure.

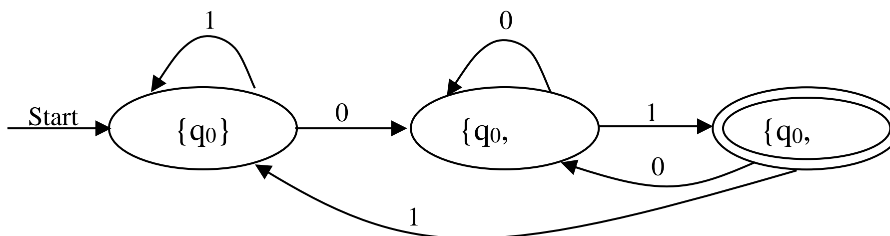


Fig 5: Transition diagram of the Transition Table 4

3.4 The formal Notation for an ε-NFA:

We may represent an ε-NFA exactly as we do an NFA, with one exception: the transition function must include information about transitions on ε. Formally, we represent an ε-NFA A by A = (Q, Σ, δ, q_0, F), where all components have their same interpretation as for NFA, except that δ is now a function that takes as arguments:

1. A state in Q, and
2. A member of Σ ∪{ε}, that is, either an input symbol, or the symbol ε. We require that ε, the symbol for the empty string, cannot be a member of the alphabet Σ, so no confusion results.

Self-Assessment Exercise(s) (SAE 3)

How does an extended transition function of NFA came about?

Self-Assessment Answer

Answer

Extended transition function of NFA came about because of the transition function **δ** that is different from DFA because of its argument.

Self-Assessment Exercise(s) (SAE 4)

Discuss what we mean by ε-NFA

Answer:

An ε-NFA A can be represented by A = (Q, Σ, δ, q_0, F), where all components have their same interpretation as for NFA, except that δ is now a function that takes as arguments: A state in Q, and a member of **Σ** ∪{**ε**}, that is, either an input symbol, or the symbol **ε**.

Example

The above ε-NFA is formally represented as

$$E = (\{q_0, q_1, \ldots\ldots q_5\}, \{., +, -, 0, 1\ldots.9\}, \delta, q_0, \{q_5\})$$

Where δ is defined by the transition table in figure

	ε	**+,-**	**.**	**0,1,…9**
q_0	$\{q_1\}$	$\{q_1\}$	ϕ	ϕ
q_1	ϕ	ϕ	$\{q_2\}$	$\{q_1, q_4\}$
q_2	ϕ	ϕ	ϕ	$\{q_3\}$
q_3	$\{q_5\}$	ϕ	ϕ	$\{q_3\}$
q_4	ϕ	ϕ	$\{q_3\}$	ϕ
q_5	ϕ	ϕ	ϕ	ϕ

Table 5: Transition table that define δ

4.0 Conclusion

In this unit you have studied the concepts of NFA, the extended transition function, the languages of DFA, the difference between DFA and NFA and finally the simpler notations for an **ε**-NFA; these serves as the representation of terms to be used for designing Non Deterministic Finite Automata as we move ahead in this course.

5.0 Summary

In this Module 2 Study Unit 3, the following aspects have been discussed:

i. The difference between the DFA and NFA is in the type of δ. For the NFA, δ is a function that takes a state and input symbol as arguments, but returns a set of zero, one or more states.

ii. An NFA is represented essentially like a DFA: $A = (Q, \Sigma, \delta, q_0, F)$

Where: Q is a finite set of states,Σ is a finite set of input symbols, q_0 a member of Q, is the start state, F, a subset of Q, is the set of final (or accepting) states and δ, the transition function is a function that takes a state in Q and an input symbol in Σ as arguments and returns a subset of Q.

iii. The extended transition function δ of an NFA to a function **δ** takes a state q and a string of input symbols ω, and returns the set of states that the NFA is in if it starts in state q and processes the string ω. **δ**(q**,** ω) is the column of states found after reading ω, if q is the lone state in the first column. Formally we define **δ** for an NFA's transition function δ by: **δ**(q, ε) = {q}.

iv. The Languages of an NFA is defined by: L(A) = {w/ **δ**(q_0, w) ∩ F ≠ ϕ} where A = (Q, Σ, δ, q_0, F) is an NFA, then L(A) is the set of strings ω in Σ* such that **δ**(q_0, ω) contains at least one accepting state.

v. An ε-NFA A can be represented by A = (Q, Σ, δ, q_0, F), where all components have their same interpretation as for NFA, except that δ is now a function that takes as arguments: A state in Q, and A member of Σ ∪{ε}, that is, either an input symbol, or the symbol ε.

6.0 Tutor Marked Assignments and Marking Scheme

1. Give the formal description of NFA
2. Discuss about Extended Transition function of NFA.
3. When do we have an ε-NFA

7.0 References/ Further readings

M. O. Rabin and D. Scott, "Finite Automata and their Decision Problems", *IBM Journal of Research and Development*, **3**:2 (1959) pp. 115–125.

Paritosh K. Pandya, **Automata: Theory and Practice,** TIFR, Mumbai, India, Unversity of Trento, 24th May, 2005

Michael Sipser, *Introduction to the Theory of Computation*. PWS, Boston. 1997. *(see section 1.2: Nondeterminism, pp.47–63.)*

John E. Hopcroft and Jeffrey D. Ullman, Addison-Wesley Publishing, Reading Massachusetts, 1979. *(See chapter 2.)*

^ Rabin, M. O.; Scott, D. (April 1959). (PDF, IEEE Xplore access required). *IBM Journal of Research and Development* **3** (2): 114–125. Retrieved 2007-03-15.

FOLDOC Free Online Dictionary of Computing,

Module 3

Regular Expressions

Unit 1

Overview of Regular Expression

Contents

1.0 Introduction

2.0 Learning Outcomes/Objectives

3.0 Learning Activities

3.1 Operations on Regular Expressions

3.2 Precedence of Regular Expression Operators

3.3 Building Regular Expression

3.4 Finite Automata with Regular Expression

3.5 From DFA's to Regular Expression

3.5.1 Construction of Regular Expression from DFA

3.6 Conversion of Regular Expression to Automata

3.7 The Algebraic Laws of Regular Expression.

3.8 Context free Grammar

3.9 Context free Language

4 Conclusion

5 Summary

6 Tutor Marked Assignments and Marking Scheme

7 References/ Further readings

1.0 Introduction

The algebraic description of the regular language is called as "regular expressions". Regular expressions offer a declarative way to express the strings we want to accept. Thus, regular expressions serve as the input language for many systems that process strings. Examples include:

1. Search commands
2. Lexical analyzer generators

2.0 Learning Outcomes

At the end of this lesson you should be able to

1. Define Operations on Regular Expressions
2. Cite Precedence of Regular Expression Operators
3. Discuss Finite Automata with Regular Expression
4. Convert DFA's to Regular Expression
5. Construct Regular Expression from DFA
6. Convert Regular Expression to Automata
7. Get the Algebraic Laws of Regular Expression.
8. Discuss Context free Grammar and Context free Language

3.1 Operations on Regular Expressions

Regular expressions denote languages. For a simple example, the regular expression $01^* + 10^*$ denotes the language consisting of all strings that are either a single 0 followed by any number of 1's or a single 1 followed by any number of 0's. Before describing the regular-expression notation, we need to learn the three operations on languages that the operators of regular expressions represent. These operations are:

1. The **union** of two languages L and M, denoted $L \cup M$, is the set of strings that are in either L or M, or both. For eg., if L={001,10,111} and M={ε,001}, then $L \cup M$ = {ε,10,001,111}.

2. The **concatenation** of languages L and M is the set of strings in L and concatenating it with any string in M. This operator is denoted either with a dot or with no operator at all.

   For eg., if L={001,10,111} and M={ε,001}, then

   L.M, or just LM is {001,10,111,001001,10001,111001}.

   The first three strings in LM are the strings in L concatenated with ε. Since ε is the identity for concatenation, the resulting strings are the same as the strings of L. However, the last three strings in LM are formed by taking each string in L and concatenating it with the second string in M, which is it with the second string in M, which is 001.For instance, 10 from L concatenated with 001 from M gives us 10001 for LM.

3. The **closure** of a language L is denoted L^* and represents the set of those strings that can be formed by taking any number of strings from L, possibly with repetitions and concatenating all of them. For instance, if L={0,1}, then L^* is all strings of 0's and 1's. If L={0,11}, then L^* consists of those strings of 0's and 1's such that the 1's come in pairs. e.g., 011,11110 and ε but not 01011 or 101.

Self-Assessment Exercise(s) (SAE 1)

Define regular expression with example.

Self-Assessment Answer

Answer

The algebraic description of the regular language is called as "regular expressions". Thus, regular expressions serve as the input language for many systems that process strings. Examples include:

I. Search commands

II. alyzer generators

3.2 Precedence of Regular-Expression operators:

Like other algebras, the regular expression operators have an assumed order of "precedence" which means that operators are associated with their operands in particular order. For instance, we know that xy+z groups the product xy before the sum, so it is equivalent to the parameterized expression (xy)+z and not to the expression x(y+z). For regular expressions, the following is the order of precedence for the operators.

1. The star (closure) operator is of highest precedence.
2. Next in precedence comes the concatenation or "dot" operator. After grouping all stars to their operands, we group concatenation operators to their operands. Concatenation is an associative operator, evaluates the expression from the left. For instance, 012 is grouped (01)2.
3. Finally, all unions (+ operators) are grouped with their operands. Since union is also associative, it is also evaluates from the left.

Self-Assessment Exercise (SAE 2)

State the precedence of regular expression operators

Self-Assessment Answer

Answer

1. The star (closure) operator is of highest precedence.
2. Next in precedence comes the concatenation or "dot" operator. After grouping all stars to their operands, we group concatenation operators to their operands. Concatenation is an associative operator, evaluates the expression from the left. For instance, 012 is grouped (01)2.
3. Finally, all unions (+ operators) are grouped with their operands. Since union is also associative, it is also evaluates from the left.

Algebras of all kinds start with some elementary expressions, usually constants or variables. Algebras then allow us to construct more expressions by applying a certain set of operators to these elementary expressions and to previously constructed expressions. We can describe the regular expressions recursively, as follows. In this definition we not only describe what the legal regular expressions are, but for each regular expression E, we describe the language it represents, which we denote L(E).

Basis

The basis consists of three parts.

1. The constants ε and ϕ are regular expressions, denoting the languages $\{\varepsilon\}$ and ϕ respectively. That is, $L(\varepsilon) = \{\varepsilon\}$, and $L(\phi) = \phi$
2. If 'a' is any symbol, then 'a' is a regular expression. This expression denotes the language {a}. That is, L(a) ={a}.
3. A variable usually capitalized and italic such as *L*, is a variable, representing any language.

Induction

There are four parts to the inductive step, one for each of the three operators and one for the introduction of parentheses.

1. If E and F are regular expressions, then E+F is a regular expression denoting the union of L(E) and L(F). That is, $L(E+F) = L(E) \cup L(F)$.
2. If E and F are regular expressions then EF is a regular expression denoting the concatenation of L(E) and L(F). That is L(EF) = L(E)L(F).
3. If E is a regular expression, then E^* is a regular expression, denoting the closure of L(E). That is, $L(E^*) = (L(E))^*$.
4. If E is a regular expression, then (E), a parenthesized E, is also a regular expression, denoting the same language as E. Formally, L((E))=L(E).

Self-Assessment Exercise (SAE 3)

Discuss the inductive step of regular expression

Self-Assessment Answer

Answer:

There are four parts to the inductive step, one for each of the three operators and one for the introduction of parentheses.

1. If E and F are regular expressions, then E+F is a regular expression denoting the union of L(E) and L(F). That is, $L(E+F) = L(E) \cup L(F)$.

2. If E and F are regular expressions then EF is a regular expression denoting the concatenation of L(E) and L(F). That is $L(EF) = L(E)L(F)$.

3. If E is a regular expression, then E^* is a regular expression, denoting the closure of L(E). That is, $L(E^*) = (L(E))^*$.

4. If E is a regular expression, then (E), a parenthesized E, is also a regular expression, denoting the same language as E. Formally, $L((E))=L(E)$.

3.4 Finite automata with regular expressions?

While the regular-expression approach to describe languages is fundamentally different from the finite-automaton approach, these two notations turn out to represent exactly the same set of languages, which we termed the "regular languages". In order to show that the regular expressions define the same class, we must show that:

1. Every language defined by one of these automata is also defined by a regular expression. For this proof, we assume the language is accepted by some DFA.

2. Every language defined by a regular expression is defined by one of these automata. For this part of the proof, the easiest is to show that there is an NFA with ε-transitions accepting the same language.

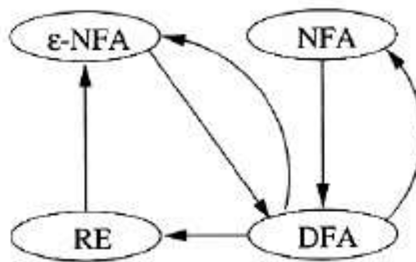


Figure 6: NFA with ε-transitions accepting the same language.

The above figure shows all the equivalences. An arc from class X to class Y means that we prove every language defined by class X is also defined by class Y. Since the graph is strongly connected we see that all four classes are really the same.

3.6 From DFA's to Regular Expressions:

We build expressions that describe sets of strings that label certain paths in the DFA's transition diagram. However, the paths are allowed to pass through only a limited subset of the states. In an inductive definition of these expressions, we start with the simplest expressions that describe paths that are not allowed to go through any state; i.e., the expressions we generate at the end represent all possible paths.

Example: Let us convert the DFA to a regular expression. This DFA accepts all strings that have at least one 0 in them. To see why, note that the automaton goes from the start state 1 to accepting state 2 as soon as it sees an input 0. The automaton then stays in state 2 on all input sequences.

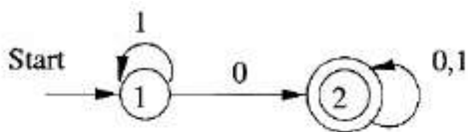


Fig 7: The transition diagram that shows basis expression from DFA

$R_{11}^{(0)}$	$\epsilon + \mathbf{1}$
$R_{12}^{(0)}$	$\mathbf{0}$
$R_{21}^{(0)}$	$\emptyset$
$R_{22}^{(0)}$	$(\epsilon + \mathbf{0} + \mathbf{1})$

Table 6: The transition Table for the transition diagram(Fig 7)

from the DFA, the basis expressions can be constructed as shown in the table.

1. For instance, $R_{11}^{(0)}$ has the term ε because the beginning and ending states are the same, state 1. It has the term 1 because there is an arc from state 1 to state 1 on input 1.

2. As another example, $R_{12}^{(0)}$ is 0 because there is an arc labeled 0 from state 1 to state 2. There is no ε term because the beginning and ending states are different.

3. For a third example, $R_{21}^{(0)}$ = ϕ, because there is no arc from state 2 to state 1.

Now, construct the expressions by considering the state 1. The rule for computing the expressions $R_{ij}^{(1)}$ are

$$R_{ij}^{(1)} = R_{ij}^{(0)} + R_{i1}^{(0)} \; R_{11}^{(0)} * R_{1j}^{(0)}$$

The following table gives first the expressions computed by direct substitution into the above formula, and then a simplified expression that we can show to represent the same language as the more complex expression.

	By direct substitution	Simplified
$R_{11}^{(1)}$	$\epsilon + 1 + (\epsilon + 1)(\epsilon + 1)^*(\epsilon + 1)$	1^*
$R_{12}^{(1)}$	$0 + (\epsilon + 1)(\epsilon + 1)^*0$	1^*0
$R_{21}^{(1)}$	$\emptyset + \emptyset(\epsilon + 1)^*(\epsilon + 1)$	$\emptyset$
$R_{22}^{(1)}$	$\epsilon + 0 + 1 + \emptyset(\epsilon + 1)^*0$	$\epsilon + 0 + 1$

Table 6: The transition Table showing expressions computed by direct substitution and its simplified expression of the latter equation.

To understand the simplification, note the general principle that

1. If R is any regular expression, then $(\varepsilon + R)^* = R^*$.

2. In our case, we have $(\varepsilon+1)^* = 1^*$; both expressions denote any number of 1’s.

3. Also that $(\varepsilon+1)1^* = 1^*$,denotes any number of 1’s.

4. Thus, the original expression $R_{12}^{(1)}$ is equivalent to 0+1*0. This expression denotes the language containing the string 0 and all strings consisting of a 0 preceded by any number of 1's. This language is also expressed by the simpler expression 1*0.

5. For any regular expression R: $\phi R = R\phi = \phi$. And $\phi + R = R + \phi = R$. That is, ϕ is the identity for union; it results in the other expression whenever it appears in a union.

Let us compute the expressions $R_{ij}^{(2)}$. The inductive rule applied with k=2 give us:

$$R_{ij}^{(2)} = R_{ij}^{(1)} + R_{i2}^{(1)} \; R_{22}^{(1)} * \; R_{2j}^{(1)}$$

The expressions are given in the following table.

	By direct substitution	Simplified
$R_{11}^{(2)}$	$1^* + 1^*0(\epsilon + 0 + 1)^*\emptyset$	1^*
$R_{12}^{(2)}$	$1^*0 + 1^*0(\epsilon + 0 + 1)^*(\epsilon + 0 + 1)$	$1^*0(0+1)^*$
$R_{21}^{(2)}$	$\emptyset + (\epsilon + 0 + 1)(\epsilon + 0 + 1)^*\emptyset$	$\emptyset$
$R_{22}^{(2)}$	$\epsilon + 0 + 1 + (\epsilon + 0 + 1)(\epsilon + 0 + 1)^*(\epsilon + 0 + 1)$	$(0+1)^*$

Table 7: The transition Table showing expressions computed by direct substitution and its simplified expression of the given equation above.

The final regular expression equivalent to the DFA is constructed by taking the union of all the expressions where the first state is the start state and the second state is accepting. In this eg., with 1 as the start state and 2 as the only accepting state, we need only the expression $R_{12}^{(1)}$.

This expression is $\mathbf{1^*0(0+1)^*}$.

It is simple to interpret this expression. Its language consists of all strings that begin with zero or more 1's, then have a 0, and then any string of 0's and 1's. In another way, the language is all strings of 0's and 1's with at least one 0.

3.6.1 Construction of regular expression from DFA

The regular expressions can be constructed by eliminating the states of DFA.

1. When a state is eliminated, all the paths that went through s no longer exist in the automaton.

2. Once a state is eliminated, the language will also change. So an arc must be included to connect its predecessor and successor.

3. This arc will contain a string as a label.

4. Regular expression is used to represent the string.

5. Finally the language of the automaton is the union over all paths from the start state to an accepting state of the language formed by concatenating the languages of the regular expressions along that path.

The following figure(7) shows a generic state s about to be eliminated. The state s has predecessor states $q_1,q_2,\ldots\ldots,q_k$ and successor states $p_1,p_2,\ldots\ldots p_m$.

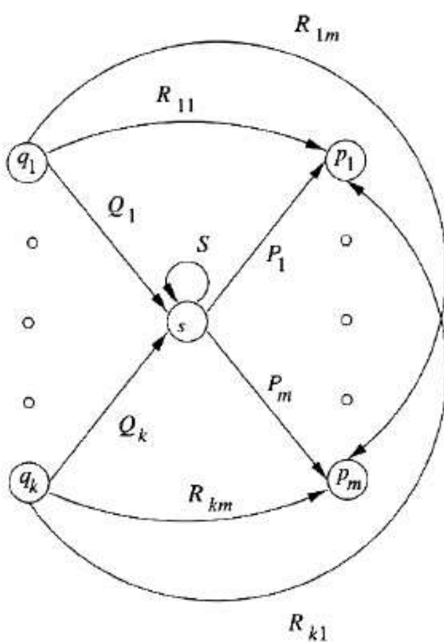


Figure7: A generic state s about to be eliminated.

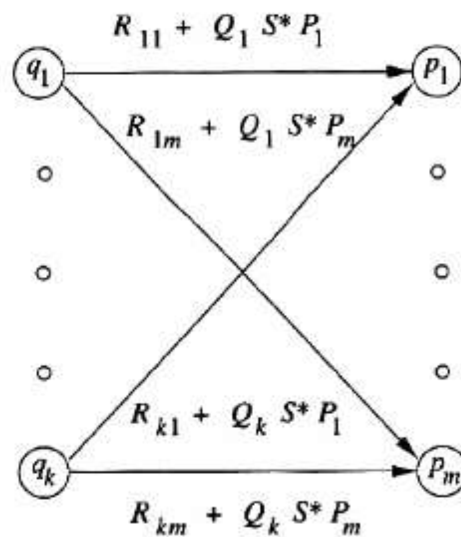


Figure 8 shows what happens when we eliminate state s.

All arcs involving state s are deleted.

1. To compensate, we introduce, for each predecessor q_i of s and each successor p_j of s, a regular expression that represents all the paths that start at q_i, go to s, perhaps loop around s zero or more times.

2. The expression for these paths is $Q_i s^* P_j$. This expression is added to the arc from q_i to p_j.

3. If there was no arc $q_i \rightarrow p_j$, then first introduce one with regular expressionϕ.

The strategy for constructing a regular expression from a finite automaton is as follows:

1. For each accepting state q, apply the above reduction process to produce an equivalent automaton with regular expression labels on the arcs.
2. Eliminate all states except q and the start state q_0.
3. If $q \neq q_0$, then we shall be left with a two-state automaton that looks like as in the figure.
4. The regular expression for the accepted strings can be described in various ways.
5. One is (R+SU*T)* SU*.
    a. In explanation, we can go from the start state to itself any number of times, by following a sequence of paths whose labels are in either L(R) or L(SU*T).
    b. The expression SU*T represents paths that go to the accepting state via a path in L(S), perhaps return to the accepting state several times using a sequence of paths with labels in L(U), and then return to the start state with a path whose label is in L(T).
    c. Then we must go to the accepting state, never to return to the start state, by following a path with a label in L(S).
    d. Once in the accepting state, we can return to it as many times as we like, by following a path whose label is in L(U).

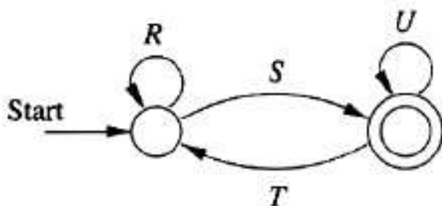


Fig 9: Diagram showing that the start state is also an accepting state

6. If the start state is also an accepting state, then we must also perform a state-elimination from the original automaton that gets rid of every state but the start state .When we do so, we are left with a one-state automaton that looks like in the following figure. The regular expression is R^*.

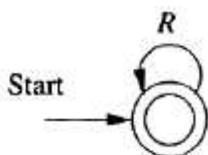


Fig 10: a one-state automaton(R^*)

7. The desired regular expression is the sum of all the expressions derived from the reduced automata for each accepting state, by rules (2) and (3).

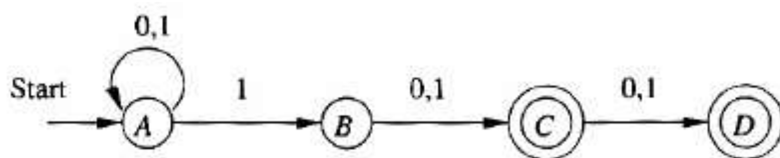


Fig 11: Transition diagram showing NFA that accepts all strings of 0's and 1's

Eg: Let us consider the NFA in the above figure that accepts all strings of 0's and 1's such that either the second or third position from the end has a 1. Our first step is to convert it to an automaton with regular expression labels. Since no state elimination has been performed, all we have to do is replace the labels "0,1" with the equivalent regular expression 0+1. The result is shown in the following figure.

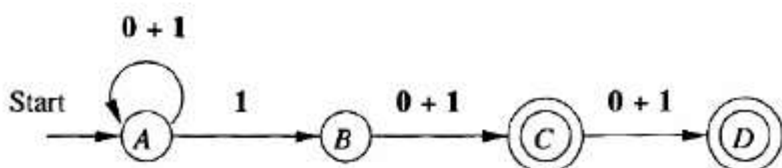


Fig 12: Equivalent expression diagram for the transition diagram figure 10

Let us first eliminate state B. Since this state is neither accepting nor the start state, it will not be in any of the reduced automata. State B has one predecessor, A, and one successor, C.

1. As a result, the expression on the new arc from A to C is $\phi+1\phi^*(0+1)$.
2. To simplify, we first eliminate the initialϕ, which any be ignored in a union. The expression thus becomes $1\phi^*(0+1)$. Note that the regular expression ϕ^* is equivalent to the regular expressionε.
3. Thus, $1\phi^*(0+1)$ is equivalent to $1(0+1)$.

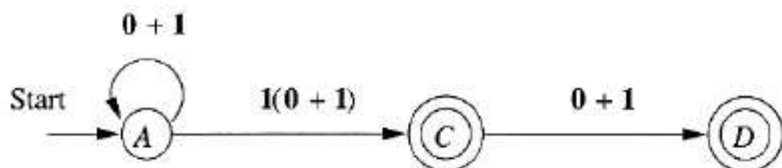


Fig 13: The resulting automaton after eliminating State B

Now, we must branch, eliminating states C and D in separate reductions. To eliminate state C, and the resulting automaton is as shown in the figure.

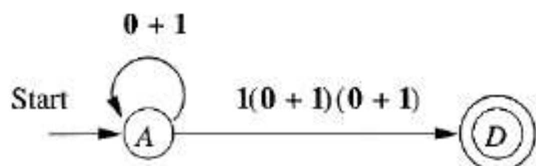


Fig 14: The resulting automaton after eliminating State C

1. In terms of the generic two-state automaton, the regular expressions are: R= 0+1, S=1(0+1)(0+1), T=ϕ, and U=ϕ. The expression U^* can be replaced by ε.
2. Also, the expression SU^*T is equivalent to ϕ.
3. The generic expression $(R+SU^*T)^*SU^*$ thus simplifies in this case to R^*S, or $\mathbf{(0+1)^*1(0+1)(0+1)}$. In informal terms, the language of this expression is any string ending in 1, followed by two symbols that are each either 0 or 1.
4. That language is one portion of the strings accepted by the original automaton. those strings whose third position from the end has a 1.

Now we start again and eliminate state D instead of C. since D has no successors. The resulting two-state automaton is shown in the figure.

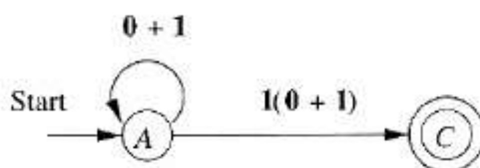


Fig 15: The resulting automaton after eliminating State D instead of C

Thus, we can apply the rule for two-state automata and simplify the expression to get $\mathbf{(0+1)^*1(0+1)}$. This expression represents the other type of string the automaton accepts: those with a 1 in the second position from the end. All that remains is to sum the two expressions to get the expression for the entire automaton. This expression is

$\mathbf{(0+1)^*1(0+1) + (0+1)^*1(0+1)(0+1)}$

3.7 Conversion of regular expression into automata?

The automata can be constructed for single symbols ε, and ϕ. And they can be combined into larger automata that accept the union, concatenation, or closure of the language accepted by smaller automata. All of the automata we construct are ε-NFA's with a single accepting state.

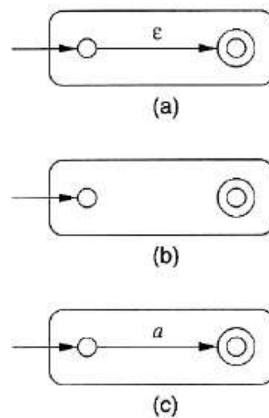


Fig 16(a.b,c): Showing the Conversion of regular expression into automata

Theorem: Every language defined by a regular expression is also defined by a finite automaton.

Proof: Suppose L=L(R) for a regular expression R. We show that L=L(E) for some ε-NFA E with:

1. Exactly one accepting state
2. No arcs into the initial state
3. No arcs out of the accepting state

BASIS

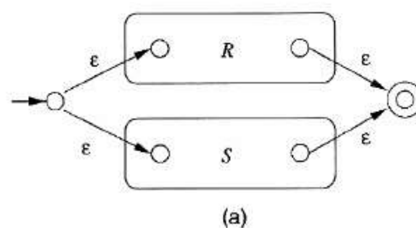
There are three parts to the basis as shown in the figure.

1. Part (a) shows to handle the expressionε. The language of the automaton is easily seen to be {ε}, since the only path from the start state to an accepting state is labeledε.
2. Part (b) shows the construction forϕ. Clearly there are no paths from start state to accepting state, so ϕ is the language of this automaton.
3. Finally, part (c) gives the automaton for a regular expression a. the language of this automaton consists of the one string a, which is also L (a). It is easy to check that these automata all satisfy conditions (1), (2) and (3) of the inductive hypothesis.

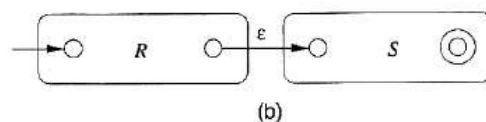
Induction

The three parts of the induction are shown in the figure. It is assumed that the statement of the theorem is true for the immediate sub expressions of a given regular expression; that is, the languages of these sub expressions are also the languages of ε-NFA's with a single accepting state. The four cases are:

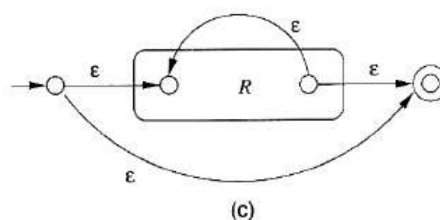
1. The expression id R+S for some smaller expressions R and S. Then the automaton of fig(a) serves. That is, starting at the new start state, we can go to the start state of either R or S. We then reach the accepting state of one of these automata, following a path labeled by some string in L(R) or L(S). Once we reach the accepting state of the automaton for R or S, we can follow one of the ε-arcs to the accepting state of the new automaton. Thus, the language of the automaton in fig(a) shows L(R)∪L(S).



(a)



(b)



(c)

Fig 17(a.b,c): Showing the three parts of the induction

2. The expression is RS for some smaller expression R and S. The automaton for the concatenation is shown in fig(b). Note that the start state of the first automaton becomes the start state of the whole, and the accepting state of the second automaton becomes the accepting state of the whole. The idea is that the only paths from the start state to accepting state go first through the automaton for R, where it must follow a path labeled by a string in L(R), and then through the automaton for S, where it follows a path labeled by a string in L(S). Thus, the paths in the automaton of fig(b) are all and only those labeled by strings in L(R) L(S).

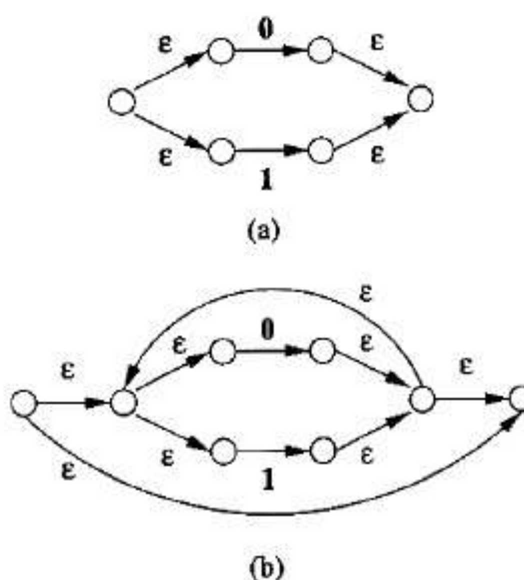
3. The expression is R^* for some smaller expression R. Then we use the automaton of fig(c). That automaton allows us to go either.

    1. Directly from the start state to the accepting state along a path labeled ε which is in L(R)* no matter what R is.

    2. To the start state of the automaton for R through that automaton one or more times, and then to the accepting state. This set of paths allows us to accept in L(R), L(R) L(R), L(R) L(R) L(R), and so on, thus covering all strings in L(R)* except ε which was covered by direct arc to the accepting state mentioned in 3(a).

4 The expression is (R) for some smaller expression R. The automaton of R also serves as the automaton for (R) since the parenthesis do not change the language defined by the expression.

Fig 18: The Resulting Automaton



Conclusion: It is a simple observation that the constructed automata satisfy the three conditions given in the inductive hypothesis. One accepting state, with no arcs into the initial state or out of the accepting state.

Example

Let us convert the regular expression $(0+1)^*1(0+1)$ to an ε-NFA.

1 Our first step is to construct an automaton for 0+1.

2 Next, we apply closure operation.

3 The third automaton in the concatenation is another automaton for 0+1.

4 The step by step construction is shown in the figure.

Self-Assessment Question(s) (SAE 4)

How can u convert a regular expression to an automata?

Self-Assessment Answer

Answer

Proof: Every language defined by a regular expression is also defined by a finite automaton.

Suppose L=L(R) for a regular expression R. We show that L=L(E) for some ε-NFA E with:

1. Exactly one accepting state
2. No arcs into the initial state
3. No arcs out of the accepting state

3.8 The Algebraic laws for regular expressions.

Like arithmetic expressions, the regular expressions have a number of laws that work for them. Many of these are similar to the laws for arithmetic, if we think of union as addition and concatenation as multiplication.

Commutativity

Commutativity is the property of where the order of its operands can be switched and gets the same result. An example for arithmetic is given as $x+y = y+x$.

1. $L+M = M+L$. This law, the commutative law for union, says that we may take the union of two languages in either order.

Associativity

Associativity is the property of an operator that allows us to group the operands when the operator is applied twice. For eg., the associative law of multiplication is $(x \times y) \times z = x \times (y \times z)$.

2. $(L+M)+N = L+(M+N)$. The union of three languages either by taking the union of the last two initially, or taking the union of the last two initially.

3. $(LM)N = L(MN)$. This law, the associative law for concatenation, says that we can concatenate three languages by concatenating either the first two or the last two initially.

Identities

An identity for an operator is a value such that when the operator is applied to the identity for addition, since $0+x = x+0=x$, and 1 is the identity for multiplication, since $1 \times y = y \times 1 = y$.

1. $\phi + L = L + \phi = L$. This law asserts that ϕ is the identity for union.

2. $\varepsilon L = L\varepsilon = L$. This law asserts that ε is the identity for concatenation.

Annihilators

An annihilator for an operator is a value such that when the operator is applied to the annihilator and some other value, the result is the annihilator. For instance, 0 is an annihilator for multiplication, since $0 \times y = y \times 0 = 0$. There is no annihilator for addition.

3. $\phi L = L\phi = \phi$. This law asserts that ϕ is the annihilator for concatenation.

Distributive Laws:

A distributive law involves two operators, and asserts that one operator can be pushed down to be applied to each argument of the other operator individually. The most common example from arithmetic is the distributive law of multiplication over addition, that is $x \times (y + z) = x \times y + x \times z$. Two types are

1. $L(M+N) = LM + LN$. This law, is the left distributive law of concatenation over union.
2. $(M+N)L = ML + NL$. This law, is the right distributive law of concatenation over union.

The Idempotent Law:

An operator is said to be idempotent if the result of applying it to two of the same values as arguments is that value. The common arithmetic operators are not idempotent; $x + x \neq x$ in general and $x \times x \neq x$ in general. However, union and intersection are common examples of idempotent operators. Thus, for regular expressions,

1. $L+L=L$ This law, the idempotent law for union, states that if we take the union of two identical expressions, we can replace them by one copy of the expression.

Laws involving closures:

There are a number of laws involving the closure operators as given below

2. $(L^*)^*$ This law says that closing an expression that is already closed does not change the language. That is $(L^*)^*$ L^*
3. $\phi^* = \varepsilon$. The closure of ϕ contains only the string ε.
4. $\varepsilon^* = \varepsilon$. It is easy to check that the only string that can be formed by concatenating any number of copies of the empty string is the empty string itself.

$L^+ = LL^* = L^*L$

$L^* = L^+ + \varepsilon.$

3.9 Context Free Grammar (CFG)

There are four important components in a grammatical description of language:

1. There is a finite set of symbols that form the strings of the language being defined. This set was {0,1}.We call this alphabet the terminals or terminal symbols.
2. There is a finite set of variables also called sometimes non-terminals or syntactic categories. Each variable represents a language that is a set of strings.
3. One of the variables represents the language being defined; it is called the start symbol. Other variables represent auxiliary classes of strings that are used to help define the language of start symbol.
4. There is a finite set of productions or rules the represents the recursive definition of a language. Each production consist of
   a. A variable that is being defined by the production and it is being defined by the production and it is called as the head of the production.
   b. The production symbol→.
   c. A string of zero or more terminals and variables. This string called the body of the production represents one way to form strings in the language of the variable of the head.

The CFG G can be represented by its 4 components that is G=(V,T,P,S)

V → set of variables

T → terminals

P → set of productions

S → start symbol.

Example-1: Consider the context free grammar for palindromes.

1. $P \to \varepsilon$

2. $P \rightarrow 0$

3. $P \rightarrow 1$

4. $P \rightarrow 0P0$

5. $P \rightarrow 1P1$

The grammar G_{pal} for palindrome is represented by: G_{pal}=({p},{0,1},A,P)

where A represents the set of 5 production as seen above.

Example-2: Consider the set of identifiers only with the letters a, b and the digits 0 and 1. Every identifier must begin with a or b, which may be followed by any string in {a,b,0,1} . We need two variables in this grammar one we call E, represents the expressions. It is the start symbol and represents the language of expressions we are defining. The other variable I represents the identifier. The regular expression for the language is (a+b) (a+b+0+1)*. This expression can be converted into a CFG as follows:

1. $E \rightarrow I$
2. $E \rightarrow E + E$
3. $E \rightarrow E * E$
4. $E \rightarrow (E)$
5. $I \rightarrow a$
6. $I \rightarrow b$
7. $I \rightarrow Ia$
8. $I \rightarrow Ib$
9. $I \rightarrow I0$
10. $I \rightarrow I1$

The grammar for expressions is started finally as G = ({E,I},T,P,E) where T is the set of symbols {+,*,(,)a,b,0,1} and P is the set of productions shown above.

I. Rule (1) is the basis rule for expressions. It says that expressions can be a single identifier.

II. Rule (2) through(4) describe the inductive case fro expressions.

III. Rule(2) say that an expression can be two expressions connected by a plus sign,

IV. Rule (3)says the same with a multiplication sign.

V. Rule (4) says that if we take any expression and put matching parentheses around it, the result is also an expression.

VI. Rules(5) through(10) describe identifiers. I.

VII. The basis is rules(5)and (6); they say that a and b are identifiers. The remaining 4 rules are the inductive case, They say that if we have any identifier, we can follow it by a,b,0(or)1, and the result will be another identifiers.

3.9.1 Context Free Language?

If G(V,T,P,S) is a CFG, the language of G, denoted L(G), is the set of terminal strings that have derivations from the start symbol. That is

$$L(G) = \{ w \text{ in } T \mid S \overset{*}{\underset{G}{\Rightarrow}} w \}$$

If a language L is the language of some context free grammar, then L is said to be a context free language or CFL. For instance, we asserted that the grammar of the language of palindromes over alphabet {0,1}. Thus the set of palindrome is a context free language.

Ex: Context free grammar for palindromes.

1. $P \rightarrow \epsilon$
2. $P \rightarrow 0$
3. $P \rightarrow 1$
4. $P \rightarrow 0P0$
5. $P \rightarrow 1P1$

Self-Assessment Question(s) (SAE 5)

Explain Context Free Language?

Self-Assessment Answer

Answer

If G(V,T,P,S) is a CFG, the language of G, denoted L(G), is the set of terminal strings that have derivations from the start symbol. That is L(G) = { w in T | $S \overset{*}{\underset{G}{\Rightarrow}} w$ }

4.0 Conclusion

In this unit we have discussed the Operations on Regular Expressions, Precedence of Regular Expression Operators, how to build Regular Expression, relate Finite Automata with Regular Expression. Also construction of Regular Expression from DFA, conversion of Regular Expression to Automata and the Algebraic Laws of Regular Expression were discussed. Finally the concept of Context free Grammar and Context free Language were studied. All these are the rudimentary of Regular Expression.

5.0 Summary

In this Module 3 Study Unit 1, the following aspects have been discussed:

i. The algebraic description of the regular language is called “regular expressions”.

ii. The Operations on Regular Expressions are union, concatenation and closure.

iii. The order of precedence for the regular expression operators is :The star (closure) operator is of highest precedence. Next in precedence comes the concatenation or “dot” operator. Finally, all unions (+ operators) are grouped with their operands. Since union is also associative, it is also evaluates from the left.

iv. For each regular expression E, we describe the language it represents, which we denote L(E).

v. Every language defined by one of these automata is also defined by a regular expression. For this proof, we assume the language is accepted by some DFA.

vi. The regular expressions can be constructed by eliminating the states of DFA.

vii. The automata can be constructed for single symbols ε, and ϕ. And they can be combined into larger automata that accept the union, concatenation, or closure of the language accepted by smaller automata.

viii. Every language defined by a regular expression is also defined by a finite automaton.

ix. Many of the law of arithmetic are the Algebraic laws for regular expressions.

x. A CFG G can be represented by its 4 components that is G=(V,T,P,S) where V is the set of variables, T is the terminals, P is the set of productions and S is the start symbol.

xi. A Context Free Language G(V,T,P,S) is the language of G, denoted L(G), is the set of terminal strings that have derivations from the start symbol.

6.0 Tutor Marked Assignments and Marking Scheme

i. if L={10,011,100} and M={ε,101} Define the Union, concataenation and closure operation on the set of string.

ii. Explain finite automata with regular expressions?

iii. Proof that every language defined by an automaton is also defined by a regular expression.

iv. Discuss the Algebraic laws for regular expressions.

7.0 References/ Further Readings

Cox, Russ (2007). "Regular Expression Matching Can Be Simple and Fast". http://swtch.com/~rsc/regexp/regexp1.html

Forta, Ben (2004). *Sams Teach Yourself Regular Expressions in 10 Minutes*. Sams. ISBN 0-672-32566-7.

Friedl, Jeffrey (2002). *Mastering Regular Expressions*. O'Reilly. ISBN 0-596-00289-0. http://regex.info/.

Gelade, Wouter; Neven, Frank (2008). "Succinctness of the Complement and Intersection of Regular Expressions". *Proceedings of the 25th International Symposium on Theoretical Aspects of Computer Science (STACS 2008)*. pp. 325–336. http://drops.dagstuhl.de/opus/volltexte/2008/1354

Gruber, Hermann; Holzer, Markus (2008). "Finite Automata, Digraph Connectivity, and Regular Expression Size". *Proceedings of the 35th International Colloquium on Automata, Languages and Programming (ICALP 2008)*. **5126**. pp. 39–50. doi:10.1007/978-3-540-70583-3_4.

Habibi, Mehran (2004). *Real World Regular Expressions with Java 1.4*. Springer. ISBN 1-59059-107-0.

Hopcroft, John E.; Motwani, Rajeev; Ullman, Jeffrey D. (2000). *Introduction to Automata Theory, Languages, and Computation* (2nd ed.). Addison-Wesley.

Kleene, Stephen C. (1956). "Representation of Events in Nerve Nets and Finite Automata". In Shannon, Claude E.; McCarthy, John. *Automata Studies*. Princeton University Press. pp. 3–42

Kozen, Dexter (1991). "A Completeness Theorem for Kleene Algebras and the Algebra of Regular Events". *Proceedings of the 6th Annual IEEE Symposium on Logic in Computer Science (LICS 1991)*. pp. 214–225

Sipser, Michael (1998). "Chapter 1: Regular Languages". *Introduction to the Theory of Computation*. PWS Publishing. pp. 31–90. ISBN 0-534-94728-X.

Stubblebine, Tony (2003). *Regular Expression Pocket Reference*. O'Reilly. ISBN 0-596-00415-X.

Wall, Larry (2002). "Apocalypse 5: Pattern Matching".

Goyvaerts, Jan; [Jan Goyvaerts], [Steven Levithan] (2009). *Regular Expressions Cookbook*. [O'reilly]. ISBN 978-0-596-52068-7.

Module 4

Computability Theory

Unit 1

Turing Machine

Contents

1.0 Introduction

2.0 Learning Outcomes/Objectives

3.0 Learning Activities

3.1 Introduction:

3.1.1 Moore and

3.1.2 Mealy Machine

3.2 Turing Machine

3.2.1 Notation for Turing Machine

3.2.2 Instantaneous Description

3.2.3 Transition Diagram

3.2.4 Language of Turing Machine

3.2.5 Storage in the State

3.3 Non Deterministic Turing machine

3.4 Counter and Multi- Stack Machine

4.0 Conclusion

5.0 Summary

6.0 Tutor Marked Assignments and Marking Scheme

7.0 References/ Further readings

1.0 Introduction

One limitation of the finite automaton as we have defined it is that its output is limited to a binary signal: "accept" / "don't accept." Model in which the output is chosen form some other alphabet have been considered .There are two distinct approaches; the output may be associated with the state (called a Moore machine) or with the transition (called a Mealy machine). We shall define each formally and then show that the two machine types produce the same input-output mappings.

2.0 Learning Outcomes/Objectives

At the end of this lesson you should be able to:

i. Discuss the concept of Moore and Mealy machine
ii. The need for Turing machine
iii. The concept of Turing machine
iv. Explain Non Deterministic Turing machine
v. Explain Counter and Multi- Stack Machine

3.0 Learning Activities

3.1 Introduction

3.1.1 Moore machines

A Moore machine is a six-tuple (Q, Σ, Δ, λ, õ,q0) , where Q, Σ, õ and q0 are as in the DFA. Δ is the output alphabet and λ is a mapping from Q to Δ giving the output associated with each state. The output of M in response to input a1,a2 .... an , n≥0, is λ (q0) λ (q1) …. λ (qn), where q0,q1, ….. qn is the sequence of states such that õ (qi-1,a1) = qi for 1≤ i ≤ n. Note that any Moore machine gives output λ (q0) in response to input ε. The DFA may be viewed as a special case of a Moore machine where the output alphabet is {0,1} and state q is "accepting" if and only if λ (q)=1.

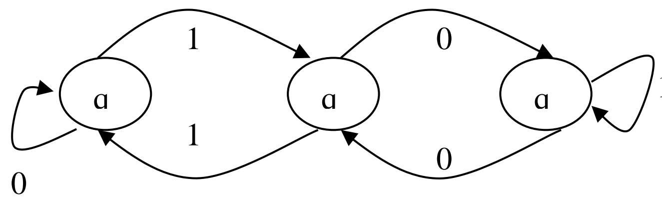


Fig 19: A Moore machine calculating

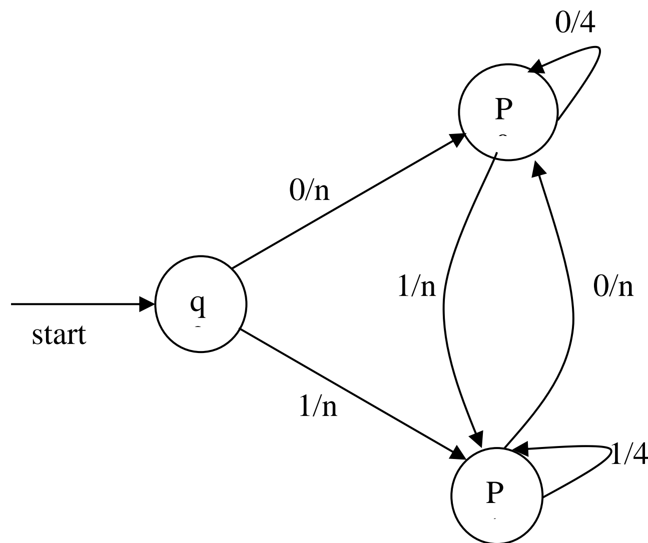
Example :

Suppose we wish to determine the residue mod 3 for each binary string treated as a binary integer. To begin, observe that if I written in binary is followed by a 0, the resulting string has value 2i, and if I in binary is followed by a remainder of 2i/3 is 2p mod 3. If p = 0, 1, or 2, then 2p mod3 is 0, 2, or1, respectively. Similarly, the remainder of (2i+1)/3 is 1, 0, or 2, respectively. It suffices therefore to design a Moore machine with three states, q0, q1, and q2, where qj is entered if and only if the input seen so far the residue j. We define λ(qj) = j for j = 0, 1, and 2 .

In the above transition diagram, where outputs label the states. The transition function T is designed to reflect the rules regarding calculation of residues described above. On input 1010 the sequence of states entered is q0,q1,q2,q2,q1, giving output sequence 01221. That is, E (which has "value"0) has residue 0, 1 has residue 1,2 (in decimal) has residue 2 , and 10 (in decimal) has residue 1.

3.1.2 Mealy machines

A Mealy machine is also a six-tuple M = (Q, Σ, Δ, λ, õ, q0), where all is as in the Moore machine, except λ that maps Q * Σ to Δ. That is, λ(q,a) gives the output associated with the transition from state q on input a, The output of M in response to input a1,a2 …. An is λ (q0,a1) λ (q1,a2) …. λ (qn-1,an), where q0,q1, ….. qn is the sequence of states such that õ (qi-1,a1) = qi for $1 \leq i \leq n$. Note that this sequence has length n rather than length n + 1 as for the Moore machine, and on input ε a Mealy machine gives output ε.



A Mealy

Example

Even if the output alphabet has only two symbols, the Mealy machine model can save states when compared with a finite automation. Consider the language (0 + 1)*(00 + 11) of all strings of 0's and 1's whose last two symbols are the same. In the next chapter we shall develop the tools necessary the show that this language is accepted by no DFA with fewer than five states. However, we may define a three-state Mealy machine that uses its state to remember the last symbol read, emits output y whenever the current input matches the precious one, and emits n otherwise.

The sequence of y's and n's emitted by the Mealy machine corresponds to the sequence of accepting and non accepting states entered by a DFA on the same input; however, the Mealy machine does not make an output prior to any input, while the DFA rejects the string E, as its initial state is nonfinal.

The Mealy machine M = ({q0,p0,p1},{0,1},{y,n}, δ̃, λ,q0) is shown in below figure. We use the label a/b on an arc from state p to state q to indicate that δ̃ (p,a) = q and λ (p , a) = b. The response of M to input 01100 is nnyny, with the sequence of states entered being q0p0p1p1p0p0. Note how p0 remembers a zero and p1 remember a one. State q0 is initial and "remembers" that no in

Self-Assessment Exercise(s) (SAE 1)

Differentiate between Moore and Mealy machines?

Self-Assessment Answer

Answer:

A Moore machine is a six-tuple (Q, Σ, Δ, λ, õ,q0) , where Q, Σ, õ and q0 are as in the DFA. Δ is the output alphabet and λ is a mapping from Q to Δ giving the output associated with each state. While A Mealy machine is also a six-tuple M = (Q, Σ, Δ, λ, õ, q0), where all is as in the Moore machine, except λ that maps Q * Σ to Δ. That is, λ(q,a) gives the output associated with the transition from state q on input a,

3.5 Turing Machine

A problem that can't be solved by computer is called as "undecidable" problems. Those problems can be solved using a new computing device called the Turing Machine.

3.2.1 Notation for the Turing Machine

The Turing machine consists of a finite control which can be in any of a finite set of states. There is a tape divided into squares or cells; each cell can hold any one of a finite number of symbols.

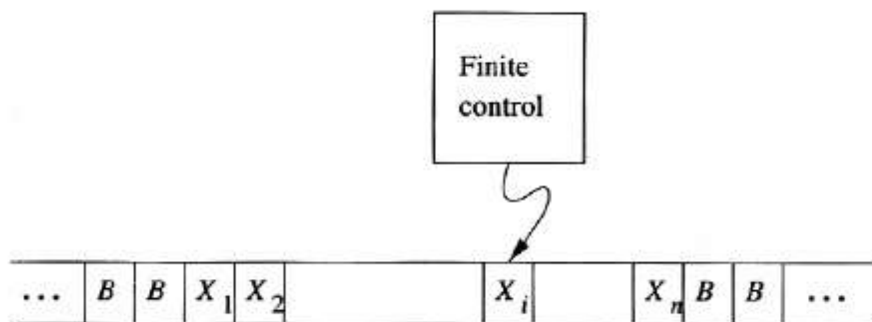


Fig 20: A Turing machine

1. Initially, the input string of symbols chosen from the input alphabet is placed on the tape.
2. All other tape cells initially hold a special symbol called the blank.
3. The blank is a tape symbol, but not an input symbol.

4. There is a tape head that is always positioned at one of the tape cells.
5. The Turing machine is said to be scanning that cell.
6. Initially the tape head is at the leftmost cell that holds the inputs.
7. A move of the TM is a function of the state of the finite control and the tape symbol scanned.
8. In one move, the Turing machine will:
   i. Change state. The next state optionally may be the same as the current state.
   ii. Write a tape symbol in the cell scanned.
   iii. Move the tape head left or right.

The formal notation we shall use for a Turing machine(TM) is similar to that used for finite automata or PDA's . We describe a TM by the 7-tuple

$M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$

Whose components have the following meanings?

Q The finite set of state of the finite control.

Σ The finite set of input symbols.

Γ The complete set of tape symbols.

δ The transition function. The value of $\delta(q, X)$ if it is defined, is a triple (p, Y, D), where :

p is the next state in Q, Y is the symbol, in Γ, D is a direction, either "left" or "right".

q_0 The start state, a member of Q, in which the finite control is found initially.

B The blank symbol.

F The set of finite or accepting state, a subset of Q.

Self Assessment Exercise(s) (SAE 2)

Explain the concepts of Turing Machine

Self-Assessment Answer

Answer

The formal notation for Turing machine(TM) is similar to that used for finite automata or PDA's . We describe a TM by the 7-tuple

$M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$

Whose components have the following meanings:

Q The finite set of state of the finite control.

Σ The finite set of input symbols.

Γ The complete set of tape symbols.

δ The transition function. The value of $\delta(q, X)$ if it is defined, is a triple (p, Y, D), where :

p is the next state in Q, Y is the symbol, in Γ, D is a direction, either "left" or "right".

q_0 The start state, a member of Q, in which the finite control is found initially.

B The blank symbol.

F The set of finite or accepting state, a subset of Q.

The ability of a TM is denoted by

1. Storage in the State
2. Multiple tracks
3. Subroutines

3.2.2. Instantaneous Descriptions

The string $X_1 X_2 \ldots X_{i-1}\, q\, X_i X_{i+1} \ldots X_n$ is used to describe the transition in Turing machines

1. q is the state of the Turing machine.

2. The tape head is scanning the ith symbol from the left.

3. $X_1X_2..X_n$ is the portion of the tape between the leftmost and the rightmost nonblank.

4. The moves of the Turing machine are described by notation. And, or will be used to indicate zero, one, or more moves of the TM.

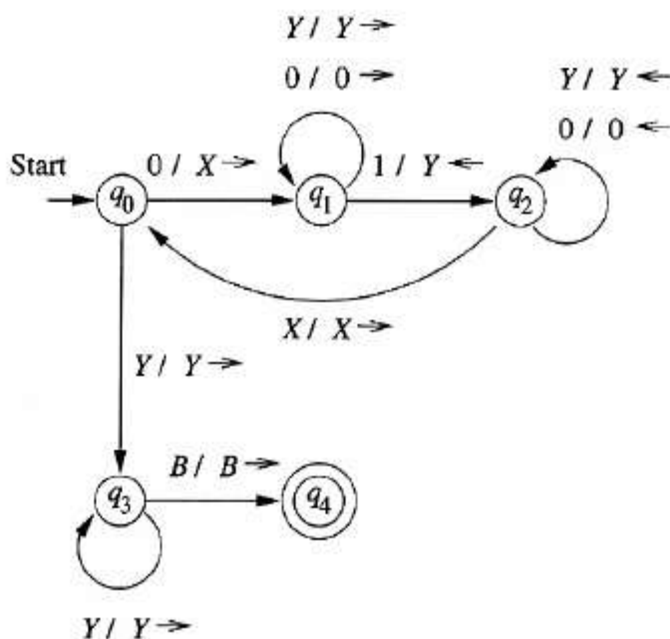
Example:

Let us design a Turing machine will accept the language $\{0^n1^n \mid n>=1\}$. Initially, it is given a finite sequence of 0's and 1's on its tape, preceded and followed by infinity of blanks. Alternately, the TM will change a 0 to an X and then a 1 to a Y, until all 0's and 1's have been matched.

In more details, starting at the left end of the input, it repeatedly changes a 0 to an X and moves to the right over whatever 0's and Y's it sees, until it comes to a 1.It changes the 1 to a Y, and moves left, over Y's and 0's , until it finds an X. At that point, it looks for a 0 immediately to the right, and if it finds one, changing it to X and repeats the process, changing a matching 1 to a Y.

The formal specification of the TM is M=({q_0,q_1,q_2,q_3,q_4},{0,1},{0,1,X,Y,B}, δ, q0, B, {q4}), Where δ is given by the table in figure 21

Fig 21: Sowing the Formal Specification of TM M=({q_0,q_1,q_2,q_3,q_4},{0,1},{0,1,X,Y,B}, δ, q0, B, {q4})



State	0	1	Symbol X	Y	B
q_0	(q_1, X, R)	–	–	(q_3, Y, R)	–
q_1	$(q_1, 0, R)$	(q_2, Y, L)	–	(q_1, Y, R)	–
q_2	$(q_2, 0, L)$	–	(q_0, X, R)	(q_2, Y, L)	–
q_3	–	–	–	(q_3, Y, R)	(q_4, B, R)
q_4	–	–	–	–	–

Table 8: Sowing the formal specification of Fig 21 (TM)

3.2 Transition Diagrams

A transition diagrams consists of a set of nodes corresponding to the states of the TM. An arc from state q to state p is labeled by one or more items of the form X / Y D, where X and Y are tape symbols, and D is a direction, either L or R. Start state is represented by the word "start" and an arrow entering that state. Accepting states indicated by double circles. This, the only information about the TM one cannot read directly from the diagram is the symbol used for the blank. We shall assume that symbol is B unless we state otherwise. The transition diagram for the previous example is given below.

Example

In this example we shall show how a Turing machine might compute the function —— ,which is called minus or proper subtraction and is defined by m —— n =max(m-n,0) that is m —— n is m-n if m ≥ n and 0 if m<n.

A TM that performs this operation is specified by

M=({q_0,q_1,…….q_6},{0,1},{0,1,B},δ,q_0,B)

Note that, since this TM is not used to accept inputs, we have omitted the seventh component, which is the set of accepting states. M will start with a tape consisting of 0^m 10^n surrounded by blanks. M halts with 0^{m-n} on its tape, M repeatedly finds its leftmost remaining 0 and replaces it by a blank. It then searches right, looking for a 1. After finding a 1, it continues right, until it comes to a 0, which it replaces by a 1. M then returns left, seeking the leftmost 0, which it identifies when it first meets a blank and then moves one cell to the right. The repetition ends if either:

1. Searching right for a 0, M encounters a blank. Then the n 0's in 0m 10^n have all been changed to 1's and m+1 of the m 0's have been changed to B. M replaces the n+1 1's by one 0 and n B's leaving m-n 0's on the tape. Since m>=n in this case, m-n=m-n.
2. Beginning the cycle, M cannot find a 0 to change to a blank, because the first m 0's already have been changed to B. Then n>=m, so m-n=0. M replaces all remaining 1's and 0's by B and ends with a completely blank tape.

3.2.4 Language of Turing machine and its Halting

The input string is placed on the tape, and the tape head begins at the leftmost input symbol. If the TM eventually enters an accepting state, then the input is accepted, and otherwise not. Let M = (Q, Σ,δ_0 ,B,F) be a Turing machine. Then L(M) is the set of strings w in such that for some state p in F and any tape strings α and β. This definition was assumed when we discussed the Turing machine of Example which accepts strings of the form 0^n 1^n. The set of languages we can accept using a Turing machine is often called the recursively enumerable languages or RE languages.

There is another notation of "acceptance" that is commonly used for Turing machines acceptance by halting. We say a TM halts if it enters a state q, scanning a tape symbol x, and there is no move in this situation. i.e., δ(q,X) is undefined.

The ability of a TM is denoted by

1. Storage in the State
2. Multiple tracks
3. Subroutines

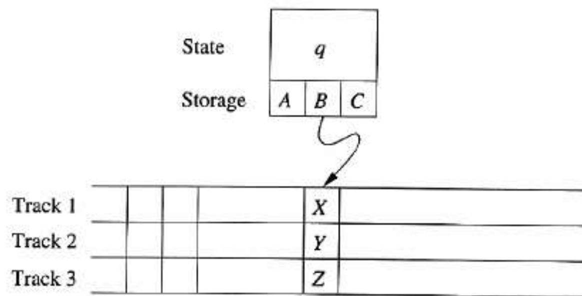


Fig 22: Showing the Storage in the State, Multiple tracks, Subroutines

3.2.5 Storage in the State

The finite control is used not only to represent a position in the "program" of the Turing machine, but to hold a finite amount of data. Here we see the finite control consisting of not only a "control" state q but three data elements A, B, and C. The technique requires no extension to the TM model and is considered as a tuple. We shall design a TM

$$M=(Q,\{0,1\},\{0,1,B\},\delta,[q_0,B],\{[q_1,B]\})$$

That remembers in its finite control the first symbol that it sees, and checks that it does not appear elsewhere on its input. Thus, M accepts the language 01*+10*. Accepting regular languages such as this one does not stress the ability of Turing machines but it will serve as a simple demonstration.

The set of states Q is $\{q_0,q_1\}\times\{0,1,B\}$. T at is the states may be thought of as pairs with two components.

A. A control portion q_0 or q_1 that remembers what the TM is doing control state q_0 indicates that M has not yet read its first symbol while q1 indicates that it has read the symbol, and is checking that it does not appear elsewhere by moving right and hoping to reach a blank cell.

B. A data portion ,which remembers the first symbol seen, Which means be 0 the transition function δ of M is as follows:

1. $\delta([q_{0,B}],a)=([q_1,a],a,R)$ for a=0 or a=1. Initially, q_0 is the control state ,and the data portio0n the is B., the symbol scanned is copied into the copied into the second component of the state, and M moves right, entering control state q_1 as it does so.

2. $\delta(q_1,a],a) = ([q_1,a],a,R)$ where a is the "complement" of a, that is, 0 if a=1 and 1 if a=0. In state q1, M skips over each symbol 0 or 1 that is different from the one it has stored in its state, and continuous moving right.

3. $\delta([q_1,a],B) =([q_1\ B],B,R)$ for a=0 or a=1. If M reaches the first blank, it enters the accepting state$[q_{1,}B]$.

3.3 Nondeterministic Turing Machines

A nondeterministic Turing machine differs from the deterministic variety we have been studying by having a transition function such that for each state q and tape symbol x, δ(q,X) is a set of triples

$\{(q1,Y1,D1),(q2,Y2,D2,),\ldots,(q_{k,},Y_k,D)\}$where k is any finite integer.

The language accepted by an NTM M is defined in the expected manner, in analogy with the other nondeterministic devices, such as N FA's and PDA's that we have studied. That is M accepts an input w if there is any sequence of choices of move that leads from the initial ID with w as input, to an ID with an accepting state.

The NTM's accept no languages not accepted by a deterministic TM. The proof involves showing that for every NTM M_N we can construct a DTM M_D that explores the ID's that MN can reach by any sequence of its choices.If M_D finds one that has an accepting state, then M_D enters an accepting state of its own. M_D must be systematic, putting new ID's on a queue rather than a stack, so that after some finite time M_D has simulated all sequences up to moves of M_N for k=1,2,….

Self Assessment Exercise(s) (SAE 3)

Outline the differences between deterministic and non deterministic Turing machine

Self-Assessment Answer

Answer

A nondeterministic Turing machine differs from the deterministic variety we have been studying by having a transition function such that for each state q and tape symbol x, δ(q,X) is a set of triples

{(q1,Y1,D1),(q2,Y2,D2,),….,($q_{k,}$,Y_k,D)}where k is any finite integer.

The language accepted by an NTM M is defined in the expected manner, in analogy with the other nondeterministic devices, such as N FA's and PDA's that we have studied. That is M accepts an input w if there is any sequence of choices of move that leads from the initial ID with w as input, to an ID with an accepting state.

The NTM's accept no languages not accepted by a deterministic TM. The proof involves showing that for every NTM M_N we can construct a DTM M_D that explores the ID's that MN can reach by any sequence of its choices.If M_D finds one that has an accepting state, then M_D enters an accepting state of its own.

M_D must be systematic, putting new ID's on a queue rather than a stack, so that after some finite time M_D has simulated all sequences up to moves of M_N for k=1,2,….

3.4 Multi stack machine and Counter machine

We now consider a class of machines called "counter machines." These machines have only the ability to store a finite number of integers ("counter"), and to make different moves depending on which if any of the counters are currently 0. The counter machine can only add or subtract one from the counter, and cannot tell two different nonzero counts from each other in effect, a counter is like a stack on which we can place only two symbols: a bottom-of-stack marker that appears only at the bottom and one other symbol that may be pushed and popped from the stack.

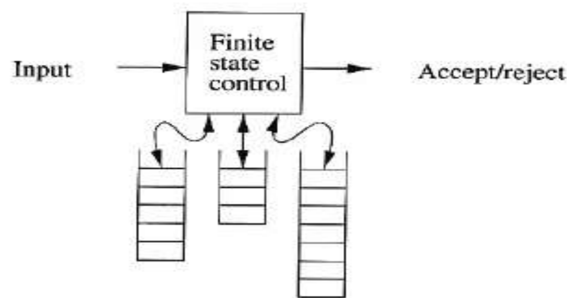


Fig 23: **Multi stack machine and Counter machine**

A k-stack machine is a deterministic PDA with k stacks. It obtains its input, like the PDA does, from an input source, rather than having the input placed on tape or stack, as the TM does. The multi stack machine has a finite control, which is in one of a finite set of states. It has a finite stack alphabet, which it uses for all its stacks. A move of the multi stack machine is based on:

1. The state of the finite control.
2. The input symbol read, which is chosen from the finite input alphabet. Alternatively, the multi stack machine can make a move using ε input, but to make the machine deterministic, there cannot be a choice of an ε-move or a non-ε-move in any situation.
3. The top stack symbol on each of its stacks.

In one move, the multistack machine can:

1. Change to a new state.
2. Replace the top symbol of each stack with a string of zero or more stack symbols. There can be (and usually is) a different replacement string for each stack.

Thus, a typical transition rule for a k-stack machine looks like:

$\delta(q,a,X1,X2\ldots X_k)=(p,Y1,Y2\ldots\ldots,Y_k)$

The interpretation of the rule is that state q, with X_i on top of the ith stack, for i=1,2….,k, the machine may consume a (either an input symbol or Є) from is input, go to state p, and replace X_i on top of the ith stack by string Y_i for each i=1,2,…,k. The multistack machine accepts by entering a final state.

We add one capability that simplifies input processing by this deterministic machine: we assume there is a special symbol $, called the end-marker that appears only at the end of the input and is not part of that input. The presence of the end marker allows us to know when we have consumed all the available input, we shall see in the next theorem how the end marker makes it easy for the multistack machine to simulate a Turing machine. Notice that the conventional TM needs no special end marker, because the first blank serves to mark the end of the input.

Counter Machines

A counter machine may be thought of in on of two ways:

1. The counter machine has the same structure as the multistack machine, but in place of each stack is a counter. Counters hold any nonnegative integer, but we can only distinguish between zero and nonzero counters. That is, the move of the counter machine depends on its states, input symbol, and which, if any, of the counters are zero. In one move the counter machine can:

   a. Change state.

   b. Add or subtract 1 from any of its counters independently. However, a counter is not allowed to become negative, so it cannot subtract 1 from a counter that is currently 0.

2. A counter machine may also be regarded as a restricted multistack machine. The restrictions are as follows:

   a. There are only tow stack symbols, which we shall refer to as Z_0 (the bottom-of-stack marker), and X.

   b. Z_0 is initially on each stack.

   c. We may replace Z_0 only by a string of the form $X^i Z_0$ for some i>=0.

   d. We may replace X only by X^i for some i>=0. That is, Z_0 appears only on the bottom of each stack, and all other stack symbols, if any, are X.

We shall use definition (1) for counter machines, but the two definitions clearly define machines of equivalent power. The reason is that stack $X^i Z_0$ can be identified with the count k. In definition (2), we can tell count 0 from other counts, because for count 0 we see Z_0 on top of the stack, and otherwise we see X.

The Power of Counter Machines

The observations about the languages accepted by counter machines are as follows:

1 Every language accepted by a counter machine is recursively enumerable. The reason I so that a counter machine is a special case of a stack machine, and a stack machine is a special case of a multi tape Turing machine, which accepts only recursively enumerable languages.

2 Every language accepted by a non-counter machine is a CFL. Note that a counter, in point-of-view (2), is a stack, so a one-counter machine is a special case of one-stack machine, i.e., a PDA. In fact, the languages of one-counter machines are accepted by deterministic PDA's, although the proof is

surprisingly complex. The difficulty in the proof stems from the fact that the multistack and counter machines have an end marker at the end of their input.

Comparing the Running Times of Computers and Turing Machines

The running time for the Turing machine that simulates a computer are as follows:

a. The issue of running time is important because we shall use the TM not only to examine the question of what can be computed at all, but when can be compared with enough efficiency.

b. The dividing line between the tractable – that which can be solved efficiently—from the intractable – problems that can be solve, but not fast enough for the solution to be usable –is generally held to be between what can be computed in polynomial time and what requires more than any polynomial running time.

c. Thus, we need to assure ourselves that if a problem can be solved in polynomial time on a typical computer, then it can be solved in polynomial time by a Turing machine, and conversely.

Thus the TM described above can simulate n steps of a computer in $0(n^3)$ time, we need to confront the issue of multiplication as a computer instruction.

Self Assessment Exercise(s) (SAE 4)

Differentiate between Counter and Multistack machine

Self-Assessment Answer

Answer

1. The counter machine has the same structure as the multistack machine, but in place of each stack is a counter. Counters hold any nonnegative integer, but we can only distinguish between zero and nonzero counters. That is, the move of the counter machine depends on its states, input symbol, and which, if any, of the counters are zero.

2. A counter machine may also be regarded as a restricted multistack machine.

5.0 Conclusion

In this unit we have explored the need of Moore Mahine, Mealy Machine and Turing machine. The concept of turing machine such its notation, transition diagram, language of Turing machine and storage in the state

were explored. The difference between non deterministic and deterministic Turing machine, counter and multi stack machine were highlighted. Conclusively the Reducibility theorem were explored

6.0 Summary

In this Module 4 Study Unit 1, the following aspects have been discussed:

1. The concept of Moore and Mealy machine
2. The need for Turing machine
3. The concept of Turing machine
4. Non Deterministic Turing machine
5. Counter and Multi- Stack Machine

6.0 Tutor Marked Assignments and Marking Scheme

1. Explain the concepts of Moore and Mealy machine
2. What is Turing Machine? Explain its Components.
3. Outline the techniques behind storage in the state
4. Explain the programming technique of Turing machine

7.0 References/Further Reading

Boolos, George; John Burgess, Richard Jeffrey, (2002). *Computability and Logic* (4th ed.). Cambridge UK: Cambridge University Press. ISBN 0-521-00758-5 (pb.).Some parts have been significantly rewritten by Burgess. Presentation of Turing machines in context of Lambek "abacus machines" (cf Register machine) and recursive functions, showing their equivalence.

Taylor L. Booth (1967), *Sequential Machines and Automata Theory*, John Wiley and Sons, Inc., New York. Graduate level engineering text; ranges over a wide variety of topics, Chapter IX *Turing Machines* includes some recursion theory.

Davis, Martin; Ron Sigal, Elaine J. Weyuker (1994). *Computability, Complexity, and Languages and Logic: Fundamentals of Theoretical Computer Science* (2nd ed.). San Diego: Academic Press, Harcourt, Brace & Company. ISBN 0-12-206382-1.

Hennie, Fredrick (1977). *Introduction to Computability*. Addison–Wesley, Reading, Mass.. On pages 90–103 Hennie discusses the UTM with examples and flow-charts, but no actual 'code'.

John Hopcroft and Jeffrey Ullman, (1979). *Introduction to Automata Theory, Languages and Computation* (1st ed.). Addison–Wesley, Reading Mass. ISBN 0-201-02988-X..A difficult book. Centered around the issues of machine-interpretation of "languages", NP-completeness, etc.

Hopcroft, John E.; Rajeev Motwani, Jeffrey D. Ullman (2001). *Introduction to Automata Theory, Languages, and Computation* (2nd ed.). Reading Mass: Addison–Wesley. ISBN 0-201-44124-1.Distinctly different and less intimidating than the first edition.

Stephen Kleene (1952), *Introduction to Metamathematics*, North–Holland Publishing Company, Amsterdam Netherlands, 10th impression (with corrections of 6th reprint 1971). Graduate level text; most of Chapter XIII *Computable functions* is on Turing machine proofs of computability of recursive functions, etc.

Zohar Manna, 1974, *Mathematical Theory of Computation*. Reprinted, Dover, 2003. ISBN 978-0-486-43238-0

Marvin Minsky, *Computation: Finite and Infinite Machines*, Prentice–Hall, Inc., N.J., 1967. See Chapter 8, Section 8.2 "Unsolvability of the Halting Problem." Excellent, i.e. relatively readable, sometimes funny.

Christos Papadimitriou (1993). *Computational Complexity* (1st ed.). Addison Wesley. ISBN 0-201-53082-1.Chapter 2: Turing machines, pp. 19–56.

Michael Sipser (1997). *Introduction to the Theory of Computation*. PWS Publishing. ISBN 0-534-94728-X.Chapter 3: The Church–Turing Thesis, pp. 125–149.

Stone, Harold S. (1972). *Introduction to Computer Organization and Data Structures* (1st ed.). New York: McGraw–Hill Book Company. ISBN 0-07-061726-0.

Module 5

Complexity Theory

Unit1: Time Complexity

Unit 2: Space Complexity

Unit 3: Probabilistic Complexity (PP)

Study Unit 1

Time Complexity

Contents

1.0 Introduction

2.0 Learning Outcomes

3.0 Learning Activities

3.1 Non Deterministic Time Complexity

3.2 P and NP problem

3.3 NP Completeness II

3.4 Polynomial- Time Reductions

3.5 Cook Levin Theorem

4.0 Conclusion

5.0 Summary

6.0 Tutor Marked Assignments and Marking Scheme

7.0 References/Further Reading

1.0 Introduction

A Turing machine M is said to be of time complexity T(n) moves, regardless of whether or not M accepts. This definition applies to any function T(n),such as $T(n)=50n^2$ or $t(n)=3^n+5n^4$;we shall be interested predominantly in the case where T(n) is a polynomial in n. we say a language L is in class p if there is some polynomial T(n) such that L=L(M) for some deterministic Tm M of time complexity T(n)**.**

2.0 Learning Outcomes/Objectives

At the end of this lesson you should be able to:

1. Describe when a Turing Machine is Non Deterministic Time Complexity
2. Explain P and NP problem
3. Discuss NP Completeness II
4. Explain Polynomial- Time Reductions
5. Narrate Cook Levin Theorem

3.0 Learning Activities

3.1Nondeterministic Time Complexity/Polynomial time

Formally, we say a language L is in the class NP. if there is a nondeterministic TM and when M is given an input of length n, there are no sequence of more than T(n) moves of M. Our first observation is that, since every deterministic TM is a nondeterministic TM that happens never to have a choice of moves, $P \subseteq NP$. However, it appears NP contains many problems not in p. The intuitive reason is that a NTM running in polynomial time has the ability to guess an exponential number of possible solutions to a problem and check one in polynomial time, “in parallel”.

However it is one of the deepest open questions of Mathematics whether p=NP.Whether in fact everything that can be done in polynomial time by a NTM can in fact are done by DTM in polynomial time, perhaps with a higher-degree polynomial.

The input to TSP is the same as to MWST, a graph with integer weights on the edges such as that of fig and a weight limit W. the question asked is whether the graph as a “Hamilton circuit” of total weight at most W. A Hamilton circuit is a set of edges with each node appearing exactly once. Note that the number of edges on a Hamilton circuit must equal the number of nodes in the graph.

Example: The graph of fig actually has only one Hamilton circuit the cycle(1,2,4,3,1). The total weight of the cycle is 15+20+18+10=63. Thus, if W is 63 or more, the answer is “yes”, and if W<63 the answer is “no”.

However, the TSP on four-node graph is deceptively simple, since there can never be more than two different Hamilton circuits once we account for the different nodes at which the same cycle can start, and for the direction in which we traverse the cycle. In m-node graphs, the number of distinct cycles grows as O(m!). the factorial of m, which is more than 2cm for any constant c.

It appears that all ways to solve the TSP involve trying essentially all cycles and computing their total weight. By being clever, we can eliminate some obviously bad choices. But it seems that no matter what we do, we must examine an exponential number of cycles before we can conclude that there is none with the desired weight limit W, or to find one if we are unlucky in the order in which we consider the cycles.

On the other hand , if we had a nondeterministic computer, we could guess a permutation of the nodes, and compute the total weight for the cycle of nodes in that order. If there were a real computer that was nondeterministic, no branch. A similar amount of time. Thus, a single –tape NTM can solve the TSP in O (n4) time at most . We conclude that the TSP is in NP.

Self Assessment Exercise(s) (SAE 1)

When do we say a language L is in the class NP?

Self-Assessment Answer

Answer

We say a language L is in the class NP. if there is a nondeterministic TM and when M is given an input of length n, there are no sequence of more than T(n) moves of M. Our first observation is that, since every deterministic TM is a nondeterministic TM that happens never to have a choice of moves, P $\subseteq$ NP. However, it appears NP contains many problems not in p

3.2 P and NP Problem

We introduce the basic components of intractability theory: the classes P and NP of problems solvable in polynomial time by deterministic and nondeterministic TM’s, respectively, and the technique of polynomial time reduction.

An example: kruskal's algorithm: Many problems that have efficient solutions; perhaps you studied some in a course on data structures and algorithms. These problems are generally in P. we shall consider one such problem finding there. Informally we think of graphs as diagram such as that of fig 24

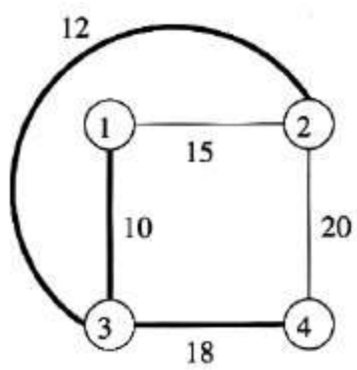


Fig 24: A spanning tree Graph of the above example

There are nodes, which are numbered 1-4 in this example graph. And there are edges between some pairs of nodes. Each edge has a weight, which is an integer. A spanning tree is a subset of the edges such that all nodes are connected through these edges. Yet there are no cycles. An example of a spanning tree is shown by bold edges. A minimum-weight spanning tree has the least possible total edge weight of all spanning trees.

There is a well-known "greedy" algorithm, called Kruskal's Algorithm, for finding a MWST. Here is an informal outline of the key ideas:

1. Maintain for each node the connected component in which the node appears, using whatever edges of the tree have been selected so far. Initially, no edges are selected, so every node is in it's a connected component by itself.
2. Consider the lowest-weight edge that has not yet been considered; break ties any way you like. If this edge connects two nodes that are currently in different connected components then:

   A Select that edge for the spanning tree, and

   B Merge the two connected components involved.

3. Continue considering edges until either all edges have been considered, or the number of edges selected for the spanning tree is one less that the number of nodes. Note that in the latter case, all nodes must be in one connected component and we can stop considering edges.

Example:

In the graph we first consider the edge(1:3) because it has the lowest weight, 10. Since 1 and 3 are initially in different components, we accept this edge and make 1 and 3 have the same component number say "component 1". The next edge in order of weights (2,3), with weight 12, since 2 and 3 are in the different components, we accept this edge and merge node 2 into "component 1".

The third edge is (1,2), with weight 15. However, 1 and 2 are now in the same component, so we reject this edge and proceed to the fourth edge, (3,4). Since 4 is not in "component 1". We accept this edge, now, we have three edges for the spanning tree of a 4-node graph, and so may stop.

3.3 NP-Completeness

NP-Complete Problems

Let L be a language (problem) in NP we say L is NP –complete if the following statements are true about L.L is in NP. For every language L1 in NP there is a polynomial-time reduction of D to L. An example of an NP-complete problem, as we shall see, is the Traveling salesman problem, which we introduced already. Since it appears that P≠NP, and in particular, all the NP-complete problems are in NP-P, we generally view a proof of NP-completeness for a problem as a proof that the problems is not in P.

We shall prove our first problem, called SAT (for Boolean satisfiability) to be NP-complete by showing that the language of every polynomial-time NTM has a polynomial –time reduction to SAT. However, once we have some NP-complete by reducing some known NP=complete problem to it, using a polynomial-time reduction.

This problem whether a Boolean expression is satisfiable—is proved NP-complete by explicitly reducing the languages of any nondeterministic, polynomial-time TM to the satisfiability problem.

The satisfiability problem

The Boolean expression is built from:

1. Variables whose values are Boolean: i.e., they either have the value 1(true)or 0 false.
2. Binary operators Λ and V standing for the logical AND or OR of the two expressions.
3. Unary operator –standing for logical negation.
4. Parentheses to group operators and operands, if necessary at alter the default precedence of operators:- highest, then Λ. and finally v.

***Please insert relevant images/graphics**

Example

An example of a Boolean expression is x Λ-(y vz). The sub expression y v z is true whenever either variable y or variable z has the value true, but the sub expressions is false whenever both y and z are false. The larger sub expression –(y V z) is true exactly when y V z is false, that is , when both y and z are false. If either y or z or both are true, then-(y V z) is false.

Finally, consider the entire expression. Since it is the logical AND of two sub expressions, it is true exactly when both sub expressions are true. That is, x Λ –(y V z) is true exactly when x is true, y is false, and z is false. A truth assignment for a given Boolean expression E assigns either true or false

to each of the variables mentioned in E. The value of expression E given a truth assignment T, denoted E(t) is the result of evaluation E with each variable x replaced by the value T(x) (true or false) that T assigns to x. A truth assignments T satisfies Boolean expression E if E(T)=1; the truth assignment t makes expression E true A Boolean expression E is said to be satisfiable if there exists at least one truth assignment T that satisfies E.

***Please insert relevant images/graphics**

Self-Assessment Exercise(s) (SAE 2)

Describe the NP complete problem?

Self-Assessment Answer

Answer

This problem whether a Boolean expression is satisfiable—is proved NP-complete by explicitly reducing the languages of any nondeterministic, polynomial-time TM to the satisfiability problem.

The satisfiability problem

The Boolean expression is built from:

1. Varibles whose values are Boolean: i.e., they either have the value 1(true)or 0 false.

2. Binary operators Λ and V standing for the logical AND or OR of the two expressions.

3. Unary operator –standing for logical negation.

4. Parentheses to group operators and operands, if necessary at alter the default precedence of operators:-highest, then Λ. and finally v.

Example:

An example of a Boolean expression is x Λ-(y vz). The sub expression y v z is true whenever either variable y or variable z has the value true, but the sub expressions is false whenever both y and z are false. The larger sub expression –(y V z) is true exactly when y V z is false, that is , when both y and z are false. If either y or z or both are true, then-(y V z) is false.

Finally, consider the entire expression. Since it is the logical AND of two sub expressions, it is true exactly when both sub expressions are true. That is, x Λ –(y V z) is true exactly when x is true, y is false, and z is false. A truth assignment for a given Boolean expression E assigns either true or false

to each of the variables mentioned in E. The value of expression E given a truth assignment T, denoted E(t) is the result of evaluation E with each variable x replaced by the value T(x) (true or false) that T assigns to x.

A truth assignments T satisfies Boolean expression E if E(T)=1; the truth assignment t makes expression E true A Boolean expression E is said to be satisfiable if there exists at least one truth assignment T that satisfies E.

Representing SAT instance

The symbols in a Boolean expression are Λ,V,- the left and right parentheses, and symbols representing variables. The satisfiability of an expression does not depend on the names of the variables, only on whether two occurrences of variables are the same variables are or different variables. Thus, we may assume that the variables are x1,x2……, although in examples we shall continue to use variable names like y or z, as well as x' s . we shall also assume that variables renamed as we use the lowest possible subscripts for the variables .

For instance, Since there are an infinite number of symbols that could in principle, appear in a Boolean expression, we have a familiar problem o having to device a code with a fixed, finite alphabet to represent expressions with arbitrarily large number of variables. Only then can we talk about SAT as a "problem", that is, as a language over a fixed alphabet consisting of the codes for those Boolean expressions that are satisfiable. The code we shall use is as follows.

1. The symbols Λ, V, ¬. (, and) are represented by themselves.

2. The variable xi is represented by the symbol x followed by 0's and 1's that represent I in binary.

Thus, the alphabet for the SAT problem/language has only eight symbols. All instances of SAT are strings in this fixed, finite alphabet.

Example

consider the expression x Λ ¬(y V z) form example 10.6 our first step in coding it is to replace the variables by subscripted x's. Since there are three variables. We must use x1, x2,and x3. we have freedom regarding which of x, y , and z is replaced by each of the xi's and to be specific let x=x1 , y=x2 and z=x2. Then the expression becomes xi Λ ¬ (x2 V x3). The code for this expression is X1 Λ ¬ (x 10 V x11)

Notice that the length of a coded Boolean expression is approximately the same as the number of positions in the expression, counting each variable be current as i. the reason for the difference is a that if the expression has m position it can have O(m) variables, so variables may take O(log m) symbols to code. Thus, an expression whose length is m positions can have a code as long as n=O(m log m) symbols.

However, the difference between m and m log m is surely limited by a polynomial. Thus, as long as we only deal with the issue of whether or not a problem can be solved in time that is polynomial in its input length, there is no need to distinguish between the length of an expression's code and the number of positions in the expression itself.

***Please insert images/graphics**

NP-Completeness of the SAT problem

We now prove "cook's Theorem", the fact that SAT is NP-complete. To prove a problem is NP-complete, we need first to show that it is in NP. Then, we must show that every language in NP reduces to the problem in question. In general, we show the second part by offering a polynomial – time reduction from some other

NP-complete problem at right now; we don't know any NP-complete problems to reduce to SAT. Thus, the only strategy available is to reduce absolutely every problem in NP to Sat.

Self Assessment Exercise(s) (SAE 3)

Explain the Complements of Languages in NP?

Self-Assessment Answer

Answer

The class of languages P is closed under complementation for a simple argument why, let L be in P and let M be a TM for L. Modify M as follows, to accept L. Introduce a new accepting state q and have the new TM transition to q whenever M halts in a state that is not accepting. Make the former accepting states of M be non accepting. Then the modified TM accepts L, and runs in the same amount of time that M does, with the possible addition of one move. Thus, L is in P if L is.

It is not known whether NP is closed under complementation. It appears not, however, and in particular we expect that whenever a language L is NP complete, then its complement is not in NP.

The class of languages P is closed under complementation for a simple argument why, let L be in P and let M be a TM for L. Modify M as follows, to accept L. Introduce a new accepting state q and have the new TM transition to q whenever M halts in a state that is not accepting. Make the former accepting states of M be non accepting. Then the modified TM accepts L, and runs in the same amount of time that M does, with the possible addition of one move. Thus, L is in P if L is.

It is not known whether NP is closed under complementation. It appears not, however, and in particular we expect that whenever a language L is NP complete, then its complement is not in NP.

The Class of Languages Co-NP

Co-NP is the set of languages whose complements are in NP. We observed at the beginning of section that every language in P has its complement also in P, and therefore in NP. On the other hand, we believe that none of the NP-complete problems have their complements in NP, and therefore no NP-complete problem is in co-NP. Likewise, we believe the complements. However, we should bear in mind that, should P turn out to equal NP, then all three classes are actually the same.

Example:

Consider the complement of the language SAT which is surely a member of ‘Co-NP we shall refer to this complement as USAT include all those that code Boolean expressions that are not satisfiable. However, also in USAT are those strings that do not code valid Boolean expressions , because surely none of those strings that do not code valid Boolean expressions, because surely none of those strings are in SAT. We believe that USAT is not in NP but there is no proof.

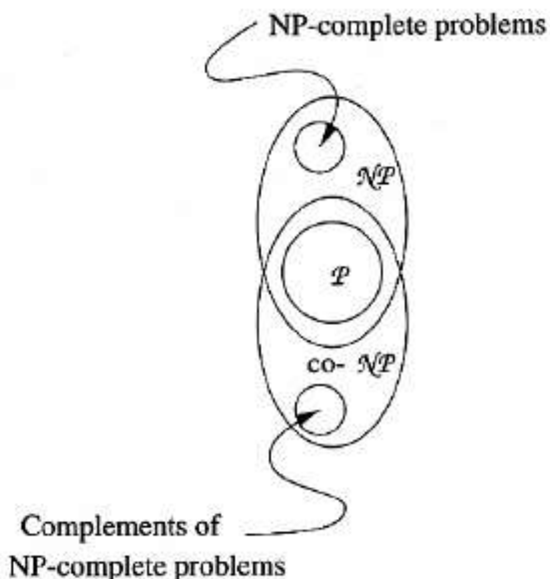


Fig 25: NP-Complete Problems and Co-NP

NP-Complete Problems and Co-NP

Let us assume that P= NP. It is still possible that the situation regarding co-NP is not exactly as suggested by figure. Because we could have NP and are in NP, and yet not be able to solve them in deterministic polynomial time. However, the fact that we have not been able to find even one NP complete problem whose complement whose complement is in NP is strong evidence that NP= co-NP, as we prove in the next theorem.

Refer the theorem NP=co-NP if and only if there is some NP-complete problem whose complement is in NP.

Self Assessment Exercise(s) (SAE 4)

What is Co-NP?

Self-Assessment Answer

Answer:

Co-NP is the set of languages whose complements are in NP

3.4 Polynomial –Time Reductions

Our principal methodology for proving that a problem P2 cannot be solved in polynomial time (i.e. P2 is not in P) is the reduction of a problem P1, which is known, not be in P, to P2.2. the approach was suggested in fig which we reproduce here as

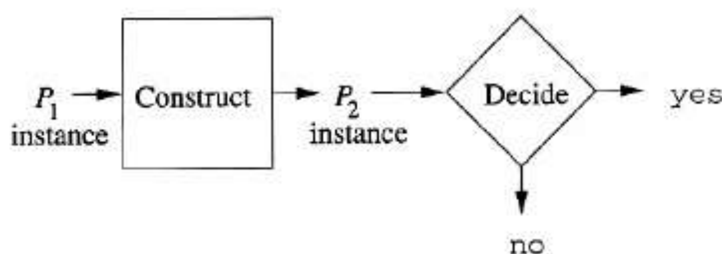


Fig 26: Reprise of the picture of reduction

Suppose we want to prove the statement: if P2 is in P, then so is P1". Since we claim that P1 is not in P, either. However, the mere existence of the algorithm labeled "Construct" in fig 10.2 is not sufficient to prove the desired statement.

For instance, suppose that when given an instance of P1 of length m , the algorithm produced and output string of length 2m, which it fed to the hypothetical polynomial –time algorithm for P2. if that decision algorithm ran in ,, say, time O(n k), then on an input of length 2m it would run in time O(2km), which is exponential in m. Thus, the decision algorithm for Pq takes, when given an input of length m, time that is exponential in m. these facts are entirely consistent with the situation where P2 is in P and P1 is not in P.

Even if the algorithm that constructs a P2 instance from a P2 instance from a P1 instance always produces an instance that is polynomial in the size of its input, we can fail to reach our desired conclusion. For instance, suppose that the instance of P2 constructed is of the same size, m, as the P1 instance, but the construction algorithm for P2 that takes polynomial time $O(n_k)$ on input of length n only implies that there is a decision algorithm for P1 that takes time O(2m+mk) on input of length m. this running time bound takes into account

the fact that we have to perform the translation to P2 as well as solve the resulting P2 instance. Again it would be possible for P1 to be in P and P2 not.

Thus, in the theory of intractability we shall use polynomial –time reductions only. A reduction from p1 t0 p2 is polynomial –time if it takes time that is some polynomial in the length of ht ep1 instance. Note that as a consequence, the p2 instance will be of a length that is polynomial in the length of the p1 instance.

***Please insert relevant images/graphics**

3.5 Cook Levin theorem

In computational complexity theory, the **Cook–Levin theorem**, also known as **Cook's theorem**, states that the Boolean satisfiability problem is NP-complete. That is, any problem in NP can be reduced in polynomial time by a deterministic Turing machine to the problem of determining whether a Boolean formula is satisfiable.

An important consequence of the theorem is that if there exists a deterministic polynomial time algorithm for solving Boolean satisfiability, then there exists a deterministic polynomial time algorithm for solving *all* problems in NP. Crucially, the same follows for any NP complete problem. A decision problem is *in NP* if it can be solved by a non-deterministic algorithm in polynomial time.

An *instance of the Boolean satisfiability problem* is a Boolean expression that combines Boolean variables using Boolean operators.An expression is *satisfiable* if there is some assignment of truth values to the variables that makes the entire expression true.

The question of whether such an algorithm exists is called the P versus NP problem and it is widely considered the most important unsolved problem in theoretical computer science. The theorem is named after Stephen Cook and Leonid Levin.

***Please insert relevant images/graphics**

THM: Suppose we are given a NTM N and a string w of length n which is decided by N in $f(n)$ or fewer nondeterministic steps. Then there is an explicit cnf formula f of length $O(f(n)^3)$ which satisfiable iff N accepts w. In particular, when $f(n)$ is a polynomial, f has polynomial length in terms of n so that every language in **NP** reduces to CSAT in polynomial time. Thus CSAT is **NP**-hard. Finally, as CSAT is in **NP**, CSAT is **NP**-complete.

The most confusing thing about the proof of the Cook-Levin Theorem are all the indices. We use the following conventions:

a. a –index for letters in tape alphabet

b. q –index for NTM states

c. i –index for tape cells

d. t –index for computation step (time)

e. k –index for branching choice

Thus, any NTM that runs in time $f(n)$ can have it's inputs of size n converted into size $O(f(n)^3)$ instances of CSAT. So if $f(n)$ is a polynomial, the reduction runs in polynomial time on a RAM (since the cube of a polynomial is a polynomial). This proves that CSAT is **NP**-complete.•

Self-Assessment Exercise(s) (SAE 5)

How many choices for each index?

Self-Assessment Answer

Answer:

The number of choices are

1. a : |G| the size of tape alphabet
2. q : $|Q|$ the number of states
3. i : $f(n)$, since ranges from 1 to $f(n)$
4. t : $f(n)+1$, since ranges from 0 to $f(n)$
5. k : b, since ranges from 1 to branching factor b

4.0 Conclusion

In this unit we have explored the concepts of Non Deterministic Time Complexity, P and NP problem,NP Completeness II, Polynomial- Time Reductions of Turing Machine. The essence of Cook Levin Theorem was envisaged. All these terms are used for Turing machine when based on complexity theory.

5.0 Summary

In this Module 5 Study Unit 1, the following aspects have been discussed:

1. The concept of Non Deterministic Time Complexity

2. P and NP problem of Turing Machine.
3. NP Completeness II of Turing Machine.
4. Polynomial- Time Reductions of Turing Machine.
5. The essence of Cook Levin Theorem

6.0 Tutor Marked Assignment

1. Write short notes on the complements of languages in NP?
2. Explain the concept of intractability theory based on polynomial time?
3. Self Assessment Exercise
4. Describe the classes P and NP problem using intractability theory

7.0 References/Further Reading

Cook, Stephen (1971). "The complexity of theorem proving procedures". *Proceedings of the Third Annual ACM Symposium on Theory of Computing*. pp. 151–158. http://portal.acm.org/citation.cfm?coll=GUIDE&dl=GUIDE&id=805047.

Karp, Richard M. (1972). "Reducibility Among Combinatorial Problems". In Raymond E. Miller and James W. Thatcher (editors). *Complexity of Computer Computations*. New York: Plenum. pp. 85–103. ISBN 0-306-30707-3. http://www.cs.berkeley.edu/~luca/cs172/karp.pdf.

T. P. Baker; J. Gill, R. Solovay (1975). "Relativizations of the P = NP question". *SIAM Journal on Computing* **4** (4): 431–442. doi:10.1137/0204037.

Dekhtiar, M. (1969). "On the impossibility of eliminating exhaustive search in computing a function relative to its graph". *Proceedings of the USSR Academy of Sciences* **14**: 1146–1148.**(Russian)**

Levin, Leonid (1973). "Universal search problems (Russian: Универсальные задачи перебора, Universal'nye perebornye zadachi)". *Problems of Information Transmission (Russian: Проблемы передачи информации, Problemy Peredachi Informatsii)* **9** (3): 265–266.**(Russian)**, translated into English by Trakhtenbrot, B. A. (1984). "A survey of Russian approaches to *perebor* (brute-

force searches) algorithms". *Annals of the History of Computing* **6** (4): 384–400. doi:10.1109/MAHC.1984.10036.

Garey, Michael R.; David S. Johnson (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman. ISBN 0-7167-1045-5.

Unit 2

Space Complexity

Contents

1.0 Introduction

2.0 Learning Outcomes

3.0 Learning Activities

- 3.1 Space complexity
- 3.2 Low Space complexity classes
- 3.3 Gaps and Speed Ups
- 3.4 Space and time hierarchy
- 3.5 Relationship between complexity classes.

4.0 Conclusion

5.0 Summary

6.0 Tutor Marked Assignments and Marking Scheme

7.0 References/Further Reading

1.0 Introduction

So far, we have only studied decision problems with respect to their computability. In this section we will look at the problem of how much space and/or time it takes to solve certain decision problems, and whether there are space and time hierarchies of decision problems.

Complexity classes correspond to bounds on resources, one such resource is space: the number of tape cells a TM uses when solving a problem

2.0 Learning Outcomes / Objectives

a. To define space complexity

b. To define the Low Space complexity classes

c. Explain Gaps and Speed Ups

d. Explain space and time hierrrchy

e. Give relationship between complexity classes

3.0 Learning Activities

3.1 Space Complexity

For any function $f:N \rightarrow N$, we define: SPACE(f(n))={ L : L is decidable by a deterministic O(f(n)) space TM}

NSPACE(f(n))={ L : L is decidable by a non-deterministic O(f(n)) space TM}

3.2 Low Space Classes

Definitions (logarithmic space classes):

1. L = SPACE(logn)
2. NL = NSPACE(logn)

Self-Assessment Exercise(s) (SAE 1)

Explain Space Complexity of a Turing Machine

Self-Assessment Answer

Answer

For any function f:N→N, we define: SPACE(f(n))={ L : L is decidable by a deterministic O(f(n)) space TM}   NSPACE(f(n))={ L : L is decidable by a non-deterministic O(f(n)) space TM}

3.3 Gaps and Speed-ups

First, we show that in certain situations there can be large gaps between complexity classes.

Theorem 3.1 *For every computable function f* : IN ! IN *there are monotonically increasing*

functions s; t : IN ! IN *with* DSPACE(s(n)) = DSPACE(f(s(n)) *and* DTIME(t(n)) = DTIME(f(t(n))) *:*

The theorem implies that there are functions $t; s$ with

DSPACE(s(n)) = DSPACE(22s(n)) andDTIME(t(n)) = DTIME(22t(n))

At first glance, this seems to be quite surprising. However, it has been shown, for instance, that DSPACE(o(log = DSPACE(1), which explains why such gaps can occur. We will see that for well behaved" functions s and t it is not possible to create such gaps.

Another phenomenon is that it is quite easy to achieve constant improvements in space or time.

Theorem 3.2 *If L can be decided by an s*(n) *space-bounded Turing machine, then L can be also decided by an s*(n)=2 *space-bounded Turing machine.*

Proof. Let M be any s(n) space-bounded Turing machine that decides L, and let ¡ be the

tape alphabet of M. In order to obtain an $s(n)=2$ space-bounded Turing machine for L, simply extend the alphabet by ¡£¡. This will allow to encode two cells of M in one cell and therefore to reduce the space requirement by a factor of two. *Ut*

Self-Assessment Exercise(s) (SAE 2)

Proof that *If L can be decided by a* $t(n)$ *time-bounded Turing machine, then L can be also decided by an* $n + t(n)=2$ *time-bounded Turing machine.*

Self-Assessment Answer

Proof.

Let M be any $t(n)$ time-bounded Turing machine that decides L, and let ¡ be the tape alphabet of M. In order to obtain an n+ $t(n)=2$ space-bounded Turing machine for L, simply extend the alphabet by ¡£¡. This will allow to encode two cells of M in one cell and therefore to reduce the time requirement by a factor of two and increase time complexity t(n) by n. *Ut*

Theorem 3.3 *If L can be decided by a* $t(n)$ *time-bounded Turing machine, then L can be also decided by an* $n + t(n)=2$ *time-bounded Turing machine.*

Proof. The proof will be an assignment. *Ut*

Next we will show that there are hierarchies of complexity classes. Recall that we defined the complexity classes over multi-tape Turing machines.

***Please insert relevant images/graphics**

3.4 A space hierarchy

First, we prove the existence of a space hierarchy. For this we will need the following lemma.

Lemma 3.4 *For any function* s : IN ! IN, *any* $O(s(n))$ *space bounded multi-tape Turing machine can be simulated by a single-tape Turing machine with* $O(s(n))$ *space.*

Proof. Let *M* = (*Q;§; ¡; ±; q*0*; F*) be an arbitrary *k*-tape Turing machine. Our aim will be to simulate *M* by a single-tape Turing machine *M0* = (*Q0;§0; ¡0; ±0; q0*0*; F0*) that uses (asymptotically) the same amount of space. To achieve this, we choose a tape alphabet of ¡*0* = (¡*[f"g*)2*k*(\"" is a symbol that is not in ¡). This is possible, because *k* is a constant. The alphabet allows us to view the tape of *M0* as consisting of 2*k* tracks.

Let these tracks be numbered from 1 to 2*k*. The purpose of the odd tracks is to store the contents of the tapes of *M*, and the purpose of the even tracks is to store the positions of the heads of *M*:

track 1 contents of tape 1

track 2 *"* (position of head 1)

track 3 contents of tape 2

track 4 *"* (position of head 2)

: : : : : :

Let *Q0* = *Q £* ¡*k*. This set of states allows *M0* to store the current state of *M* plus the *k* symbols that are read by the heads of *M*. In this case, a single step of *M* can be simulated by *M0* in two phases. In the first phase, the head of *M0* searches for the positions of the heads of *M* and stores the symbols that are read by the heads in its state. In the second phase, *M* replaces the symbols and moves the head pointers according to the transition function ± and stores in its state the next state of *M*.

Since the space required by *M0* is obviously bounded by the space required by *M*, the lemma follows. *Ut*

Given a function *s* : IN *!* IN, the language *Ls* is defined as:

Ls = *fhMiw j M* is a single-tape TM that started with *hMiw* uses at most *s*(*jhMiwj*) cells*g :*

The next theorem gives a lower bound on the amount of space necessary to decide *Ls*.

Theorem 3.5 *For any function s* : IN *!* IN*, Ls 62* DSPACE(*o*(*s*(*n*))).

Proof. Suppose on the contrary that there is a deterministic Turing machine that decides

Ls with *o*(*s*(*n*)) space. Then, according to Lemma 3.4, there is also a deterministic single-tape Turing machine that decides *Ls* with *o*(*s*(*n*)) space. Let ~*M* be any one of these machines. We use ~*M* to construct a \mean" single-tape Turing machine *M* that works as follows:

simulate ~*M* on the given input ² if ~*M* accepts, then use infinitely many tape cells otherwise, halt

***Please insert relevant images/graphics**

Next we show that, when enough space is available, *Ls* can be decided. We start with an auxiliary result.

Lemma 3.6 *If a single-tape Turing machine M started on input x uses at most s*(*jxj*) *tape cells for more than jQj ¢* (*s*(*jxj*) + 1) *¢ j¡js*(*jxj*) *time steps, then it uses at most s*(*jxj*) *tape cells for all time steps.*

Proof. We count the number of configurations that *M* started with *x* can reach. If we restrict ourselves to configurations (*q;w*1*;w*2) with *jw*1*w*2*j · s*(*jxj*), then

there are at most *jQj* many states,

there are at most *j¡js*(*jxj*) many ways of choosing *w*1*w*2, and

there are at most *s*(*jxj*) + 1 many ways of separating *w*1*w*2 into *w*1 and *w*2 (and thereby determining the head position).

Therefore, there can be at most *N* = *jQj ¢* (*s*(*jxj*) + 1) *¢ j¡js*(*jxj*) different configurations. If *M* started with *x* uses at most *s*(*jxj*) many cells for more than *N* steps, then it must have visited a configuration at least twice. Since *M* is deterministic, this means that *M* will be in an infinite loop, and thus will never use more than *s*(*jxj*) tape cells. *ut*

A function *s*(*n*) is said to be *space constructible* if there is an *O*(*s*(*n*)) space-bounded Turing machine *M* that on input *x* computes the binary representation of *s*(*jxj*). It is not difficult to check that all standard functions such as log *n*, *nk*, and 2*n* are space constructible. Using this definition, we can prove the following result.

Theorem 3.7 *For any space constructible function s* : IN *!* IN *with s*(*n*) = -(*n*)*,*

Ls 2 DSPACE(*s*(*n*)).

Proof. The following *O*(*s*(*n*)) space-bounded 2-tape Turing machine *~M*

decides *Ls*.

1. Test whether the input *x* on tape 1 is of the form *hMiw*, where *M* is a single-tape Turing machine. (This can be done with *O*(*jxj*) = *O*(*s*(*jxj*)) space.) If not, reject; otherwise

   continue.

2. Compute *s*(*jxj*). (Since *s* is space constructible, this can be done with *O*(*s*(*jxj*)) space.)

3 Compute *T* = *jQj* ¢ (*s*(*jxj*)+1) ¢ *j*¡*js*(*jxj*) on tape 1, where *Q* is the set of states and ¡ is the tape alphabet of *M*. (This can be done with *O*(*s*(*jxj*)) space, because log *T* = *O*(*s*(*jxj*)).)

4 Simulate on tape 2 *M* started with *hMiw* for *T* +1 steps. Accept if and only if *M* uses at most *s*(*jxj*) many cells during these steps.

Obviously, the space needed by ~*M* is bounded by *O*(*s*(*jxj*)). For the correctness we see that *hMiw 2 Ls ,* *M* started with *hMiw* uses at most *s*(*jhMiwj*) cells

, *M* started with *hMiw* uses at most *s*(*jhMiwj*) cells during the first

T + 1 steps *,* *hMiw* is accepted by ~*Mut*

Combining Theorem 3.5 and Theorem 3.7 we obtain the following result.

Corollary 3.8 *For any space constructible function S* : IN *!* IN *with S*(*n*) = -(*n*) *and any*

function s : IN *!* IN *with s*(*n*) = *o*(*S*(*n*)) *it holds that*

DSPACE(*s*(*n*)) ½ DSPACE(*S*(*n*)) *:*

Hence, we have an infinite space hierarchy of decision problems.

3.5 A time hierarchy

Next we prove the existence of a time hierarchy. For this we need the following lemma.

Lemma 3.9 *For any function t* : IN *!* IN*, any O*(*t*(*n*)) *time bounded multi-tape Turing machine*

*M*1 *can be simulated by a* 2*-tape Turing machine M*2 *in O*(*t*(*n*) log *t*(*n*)) *time.*

Proof. The first storage tape of *M*2 will have two tracks for each storage tape of *M*1. For convenience, we focus on two tracks corresponding to a particular tape of *M*1. The other tapes of *M*1 are simulated in exactly the same way. The second tape of *M*2 is used only for scratch, to transport blocks of data on tape 1. One particular cell of tape 1, known as *B*0, will hold the storage symbols read by each of the

heads of *M*1. Rather than moving head markers as in the proof of Lemma 3.4, *M*2 will transport

data across *B*0 in the direction opposite to that of the motion of the head of *M*1 being simulated. This enables *M*2 to simulate each move of *M*1 by looking only at *B*0. To the right of *B*0 will be blocks *B*1*;B*2*;*

: : of exponentially increasing length; that is, *Bi* is of length 2*i*¡1. Likewise, to the left of *B*0 are blocks *B*¡1;*B*¡2; : : : with *B*¡*i* having a length of 2*i*¡1. The markers between blocks are assumed to exist, although they will not actually appear until the block is used.

Recall that we concentrate on a particular head of *M*1. Let *a*0 be the contents initially scanned by this head. The initial contents of the cells to the right of this cell are *a*1; *a*2; : : :, and those to the left are *a*¡1; *a*¡2; : : :. At the beginning, the upper track of *M*2 is empty, while the lower tracks holds all the *ai*. Hence, the two tracks look as follows.

: : : : : :

: : : *a*¡7 *a*¡6 *a*¡5 *a*¡4 *a*¡3 *a*¡2 *a*¡1 *a*0 *a*1 *a*2 *a*3 *a*4 *a*5 *a*6 *a*7 : : :

B¡3 *B*¡2 *B*¡1 *B*0 *B*1 *B*2 *B*3

After the simulation of each move of *M*1, the following has to hold:

1 For any *i* > 0, either *Bi* is full (both tracks) and *B*¡*i* is empty, or *Bi* is empty and *B*¡*i* is full, or the bottom tracks of both *Bi* and *B*¡*i* are full, while the upper tracks are empty.

2 The contents of any *Bi* or *B*¡*i* represent consecutive cells on the tape of *M*1. For *i* > 0, the upper track represents cells to the left of those of the lower track, and for *i* < 0, the upper track represents cells to the right of those of the lower track.

3 For *i* < *j*, *Bi* represents cells to the left of those of *Bj* .

4 *B*0 always has only its lower track filled, and its upper track is specially marked. To see how data is transferred, imagine that the tape head of *M*1 in question moves to the left. Then *M*2 must shift the corresponding data right. To do so, *M*2 moves the head of tape 1 from *B*0 and goes to the right until it finds the first block, say *Bi*, that does not have both tracks full. Then *M*2 copies all the data of *B*0; : : : ;*Bi*¡1 onto tape 2 and stores it in the lower track of *B*1;*B*2; : : : ;*Bi*¡1 plus the lower track of *Bi*, assuming that the lower track of *Bi* is not already ¯filled. If the lower track of *Bi* is already filled, the upper track of *Bi* is used instead. In either case, there is just enough room to distribute the data. Note that the operations described can be performed in time proportional to the length of *Bi*.

Next, in time proportional to the length of *Bi*, *M*2 can find *B*¡*i* (using tape 2 to measure the distance from *Bi* to *B*0 makes this easy). If *B*¡*i* is completely full, *M*2 picks up the upper track of *B*¡*i* and stores it on tape 2. If *B*¡*i* is half full, the lower track is put on tape 2. In either case, what has been copied to tape 2 is next copied to the lower tracks of *B*¡(*i*¡1);*B*¡(*i*¡2); : : : ;*B*0. (By rule 1, these tracks have to be empty, since *B*1; : : : ;*Bi*¡1 were full.) Again, note that there is just enough room to store the data, and all the above operations can be carried out in time proportional to the length of *Bi*.

We call all that we have described above a *Bi-operation*. The case in which the head of $M1$ moves to the right is analogous. Note that for each tape of $M1$, $M2$ must perform a *Bi*-operation at most once per 2*i¡*1 moves of $M1$, since it takes this long for *B*1*;B*2*; : : : ;Bi¡*1 (which are half empty after a *Bi*-operation) to ¯ll. Also, a *Bi*-operation cannot be performed for the first

time until the 2*i¡*1th move of $M1$. Hence, if $M1$ operates in time $t(n)$, $M2$ will perform only *Bi*-operations with i · log $t(n)$ + 1.

***Please insert images/graphics**

We have seen that $M2$ needs at most $O(2i)$ steps to perform a *Bi*-operation. If $M1$ runs for $t(n)$ steps, $M2$ needs at most $t1(n)$ = logX$t(n)$+1

i=1 $O(2i)$ ¢ $t(n)$ 2*i¡*1 = $O(t(n)$ ¢ log $t(n))$ steps for the simulation of one tape of $M1$. Since the simulation of k tapes only increases the simulation time by a factor of k, the lemma follows. *Ut* Given a function t : IN *!* IN, the language *Lt* is defined as *Lt = fhMiw j M* is a 2-tape TM that started with *hMiw* halts after at most *t*(*jhMiwj*)*=jhMij* time steps*g :* We again start with a negative result.

Theorem 3.10 *For any function t* : IN *!* IN *with* $t(n)$ = -(n)*, Lt 62* DTIME($o(t(n)$= log $t(n)$)).

Proof. Suppose on the contrary that there is a deterministic Turing machine that decides *Lt* in $o(t(n)$= log $t(n)$) time steps. Then, according to Lemma 3.9, there is a deterministic 2-tape Turing machine that decides *Lt* in $o(t(n))$ time steps. Let ~*M* be any one of these machines. We use ~*M* to construct a \mean" 2-tape Turing machine *M* that works as follows: ² simulate ~*M* on the given input ² if ~*M* accepts, then run forever otherwise, halt;

What does *M* when started with *hMiw*? In order to analyze this, we distinguish between two

cases.

1. ~*M* accepts *hMiw*. Then *M* started with *hMiw* should halt after at most *t*(*jhMiwj*)*=jhMij* time steps. However, *M* will run in this case forever.

2. ~*M* does not accept *hMiw*. Then, according to *Lt*, *M* started with *hMiw* should run for more than *t*(*jhMiwj*)*=jhMij* time steps. However, since ~*M* only uses *o*(*t*(*jhMiwj*)) time steps, there must be an *n*0 such that for all *w* with *jwj* ¸ *n*0, the time used by ~*M* is less than *t*(*jhMiwj*)*=jhMij*. In this case, *M* will use at most *t*(*jhMiwj*)*=jhMij* time steps.

Since for both cases we arrive at a contradiction, the theorem follows. *ut*

Next we prove an upper bound on the time necessary to decide *L*. A function *t* : IN *!* IN is called *time constructible* if there is a *t*(*n*) time-bounded Turing machine that for an input *x* computes the binary representation of *t*(*jxj*). With this definition we can show the following result.

Theorem 3.11 *For any time constructible function t* : IN *!* IN *with t*(*n*) = -(*n*)*,*

Lt 2 DTIME(*t*(*n*)).

Proof. The following *O*(*t*(*n*)) time-bounded multi-tape Turing machine ~*M* decides *Lt*.

1 Test whether the input *x* on tape 1 is of the form *hMiw*. (This can be done in *O*(*jxj*) = *O*(*t*(*jxj*)) time steps.) If not, reject. Otherwise continue.

2 Compute *t*(*jxj*)*=jhMij* on tape 2. (Since *t* is time constructible, this can be done in *O*(*t*(*jxj*)) time steps.)

3 Simulate on tapes 3 and 4 *M* started with *hMiw* for *t*(*jxj*)*=jhMij* time steps. Accept if and only if *M* halts at one of these time steps. (Since the lookup of the next transition in *hMi* takes *O*(*jhMij*) steps and therefore the simulation of a single step of *M* takes *O*(*jhMij*) steps, the simulation of *t*(*jxj*)*=jhMij* time steps of *M* takes *O*(*t*(*jxj*)) steps.)

From the comments above it follows that the time needed by ~*M*

is bounded by *O*(*t*(*jxj*)). The correctness is straightforward. *Ut* Combining Theorem 3.10 and Theorem 3.11 we obtain the following result.

Corollary 3.12 *For any time constructible function T* : IN *!* IN *with T*(*n*) = -(*n*) *and any function t* : IN *!* IN *with t*(*n*) = *o*(*T*(*n*)) *it holds that*

DTIME(*t*(*n*)= log *t*(*n*)) ½ DTIME(*T*(*n*)) *:*

Hence, there is also an infinite time hierarchy of decision problems. Using involved methods, one can even prove the following result.

Theorem 3.13 *For any time constructible function T* : IN *!* IN *with T*(*n*) = -(*n*) *and any function t* : IN *!* IN *with t*(*n*) = *o*(*T*(*n*)) *it holds that* DTIME(*t*(*n*)) ½ DTIME(*T*(*n*)) *:*

Self-Assessment Exercise(s) (SAE 3)

What does *M* when started with *hMiw*? In order to find out, we distinguish between two cases.

Self-Assessment Answer

a. ~*M* accepts *hMiw*. Then *M* started with *hMiw* should use at most *s*(*jhMiwj*) space.

However, *M* will use in this case infinitely many tape cells.

b. ~*M* does not accept *hMiw*. Then *M* started with *hMiw* should use more than *s*(*jhMiwj*)

space. However, since ~*M* only uses *o*(*s*(*jhMiwj*)) space, there must be an *n*0 such that for all *w* with *jwj* ¸ *n*0, the space used by ~*M* is less than *s*(*jhMiwj*). In this case, also *M* will use less than *s*(*jhMiwj*) tape cells.

3.6 Relationships between complexity classes

We start with a definition of one more complexity class. The class EXP is defined as

EXP = [*k*¸1DTIME(2*nk*) *:*

The following theorem gives some easy relationships.

Theorem 3.14 *For any function f* : IN *!* IN *it holds:*

1. If L 2 DTIME(*f*(*n*))*, then L* 2 DSPACE(*f*(*n*)).

2. If L 2 NTIME(*f*(*n*))*, then L* 2 DSPACE(*f*(*n*)).

3. If L 2 DSPACE(*f*(*n*)) *and f*(*n*) ¸ log *n, then there is a constant c (depending on L) such*

that L 2 DTIME(*cf*(*n*)).

Proof. Proof of 1): If a Turing machine *M* runs for no more than *f*(*n*) time steps, then it cannot scan more than *f*(*n*) + 1 cells on any tape.

Proof of 2): Given a non-deterministic Turing machine *M* that decides *L* in time *O*(*f*(*n*)),we construct a deterministic Turing machine *M0* that decides *L* in the following way: go in a depth first-search manner through all the computational paths of *M*. If an accepting path is found, then accept. Otherwise reject. The space necessary to store the directions taken at any of the *O*(*f*(*n*)) time steps is at most *O*(*f*(*n*)), since *M* only has a constant number of choices for each time step. Furthermore, since *M* only runs for *O*(*f*(*n*)) steps, it follows from 1) that it only uses *O*(*f*(*n*)) space. Hence, the total amount of space needed by *M0* to simulate *M* is *O*(*f*(*n*)).

Proof of 3): From Lemma 3.6 we know that if an *f*(*n*) space-bounded Turing machine *M* that decides *L* can run for at most *jQj ¢ f*(*n*) *¢ j¡jf*(*n*) time steps on any input of size *n* (otherwise, it would run in an in¯nite loop). Since *f*(*n*) ¸ log *n*, there is some constant *c* such that for all *n* ¸ 1, *cf*(*n*) ¸ *jQj ¢ f*(*n*) *¢ j¡jf*(*n*). Thus, the runtime of *M* can be bounded by *cf*(*n*). *ut*

Theorem 3.14 and Assignment 1 together yield the following relationships.

Corollary 3.15

P *µ* RP *µ* BPP *µ* PSPACE *and*RP *µ* NP *µ* PSPACE *µ* EXP *:*

Next we will study the relationship between PSPACE and NPSPACE.

Theorem 3.16 (Savitch) *Let s* : IN *!* IN *be space constructible. Then* NSPACE(*s*(*n*)) *µ* DSPACE(*s*(*n*)2) *:*

Proof. Let *M* be an *s*(*n*) space-bounded nondeterministic Turing machine. W.l.o.g. *M* has only one accepting configuration ~ *C*1 and one rejecting configuration ~ *C*2. We first describe a recursive algorithm that does the following: given an input (*C*1*;C*2*; `; t*) with *`; t 2* IN and *C*1 and *C*2 being configurations of length at most `, the algorithm TEST checks, whether *C*1 *·t! C*2 using at most ` tape cells. (W.l.o.g. we will a assume that *t* is a power of 2.)

Function TEST(*C*1*;C*2*; `; t*):boolean

if *t* = 1 then *Test* = (*C*1 *!* *C*2) _ (*C*1 = *C*2)

else

*C*3 = (*q*0*; ²;B : : :B*) (` many *B*'s)

Test = *false*

repeat

Test = TEST(*C*1*;C*3*; `; t*=2) ^ TEST(*C*3*;C*2*; `; t*=2)

*C*3 = lexicographically next configuration of *C*3

until *Test* is true or no more configurations possible for *C*3

return *Test*

It is not difficult to check that the algorithm above computes the truth value of *C*1 ·*t! C*2.

Lemma 3.17 *If C*1 *and C*2 *have a length of at most `, then TEST*(*C*1*;C*2*; `; t*) *can be computed by a deterministic Turing machine using at most* (3` + log *t*)(log *t* + 1) *tape cells.*

Proof. The trick to keep the space low is to save space by processing the recursive calls on the same part of the tape. We prove the space bound by induction on *t*. *t* = 1: we have to test whether *C*1 *!* *C*2. This can obviously be done with at most 3` · (3` + log *t*)(log *t* + 1) cells. *t* > 1: the tape is used in the following way:

*C*1 *C*2 *C*3 bin(*t*) place *R* for recursive calls The Turing machine for TEST works as follows:

1. Copy *C*1, *C*3, and *t*=2 to the begin of *R*.
2. Process TEST(*C*1*;C*3*; `; t*=2) on *R*.
3. Also process TEST(*C*3*;C*2*; `; t*=2) on *R*.
4. If both Tests return \true", then return \true".
5. Otherwise, replace *C*3 by the lexicographically next configuration and go to 1).

This results in a space requirement for TEST(*C*1*;C*2*; `; t*) of at most

3` + log *t* + (3` + log(*t*=2))(log(*t*=2) + 1) | {z }

hypothesis

= 3` + log *t* + (3` + log *t* ¡ 1)((log *t* ¡ 1) + 1)

= 3` + log *t* + (3` + log *t* ¡ 1) log *t*

· (3` + log *t*)(1 + log *t*) *:*

The algorithm uses recursive operations. However, it is also possible to construct an iterative algorithm with essentially the same space requirement. *ut*

In order to prove the theorem, we use the result of Lemma 3.6, adapted to non-deterministic Turing machines: if on input *x* there is an accepting computation of *M* with space `, then there is an accepting computation with space ` and a length of at most *jQj ¢* (` + 1) *¢ j¡j*` = 2*O*(`).

Hence, only 2*O*(`) steps have to be checked by TEST to ¯nd an accepting computation. Now we are ready to construct an *O*(*s*(*n*)2)-space bounded Turing machine *M0* for *M*. Let *s0*(*n*) = *O*(*s*(*n*)) be a function so that the space used by *M* on any input *x* is at most *s0*(*jxj*).(Such as function must clearly exist.) *M0* works as follows when started with input is *x*.

a. Set *T* to the power of 2 closest from above to *jQj ¢* (*s0*(*jxj*) + 1) *¢ j¡j*`.

b. Execute TEST((*q*0*; ²; x*)*; ~ C; s0*(*jxj*)*; T*).

c. If TEST returns *true* then accept and otherwise reject.

From Theorem 3.14(3) it follows that *M0* accepts input *x* if and only if *M* has an accepting computation using space at most *s0*(*jxj*), which is true if and only if *x 2 L*(*M*). Hence,*L*(*M0*) = *L*(*M*). Furthermore, Lemma 3.17 implies that the space needed by *M0* is *O*(*s0*(*jxj*)2) = *O*(*s*(*jxj*)2). *ut*

***Please insert images/graphics**

The theorem has the following important consequence.

Corollary 3.18

PSPACE = NPSPACE *:*

4.0 Conclusion

In this unit we have defined Space Complexity, Low Space Complexity. Several theorems on Proofs of Gaps and Speed Ups have been proofed, explanation on time and space hierarchy have been given and finally give the Relationship between complexity classes. All these are the concepts of Space Complexity

5.0 Summary

In this Module 5 Study Unit 2, the following aspects have been discussed:

i. Introduction to a new way to classify problems: according to the space needed for their computation.

ii. Definition of Low Space complexity classes:

iii. Proofs of Gaps and Speed Ups

iv. Explanation of space and time hierarchy

v. Relationship between complexity classes.

5.0 Tutor Marked Assignments and Marking Scheme

i. Proof that *If L can be decided by an* $s(n)$ *space-bounded Turing machine, then L can be also decided by an* $s(n)=2$ *space-bounded Turing machine.*

ii. Proof that for *every computable function* f : IN ! IN *there are monotonically increasing functions* $s; t$: IN ! IN *with* DSPACE($s(n)$) = DSPACE($f(s(n)$) *and* DTIME($t(n)$) = DTIME($f(t(n))$)

7.0 References/Further Reading

J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, Reading, 1979.

W.J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computers and System Sciences* 4:177-192, 1970.

Unit 3

Probabilistic Complexity (PP)

Content

1.0 Introduction

2.0 Learning Outcomes

3.0 Learning Activities

3.1 Probabilistic Complexity (PP)

3.2 PP vs BPP

3.3 PP compared to other complexity classes

3.4 Complete problems and other properties

3.5 Proof that PP is closed under complement

3.6 Other equivalent complexity classes

3.6.1 PostBQP

3.6.2 PQP

4.0 Conclusion

5.0 Summary

6.0 Tutor Marked Assignments and Marking Scheme

7.0 References/Further Reading

1.0 Introduction

In complexity theory, **PP** is the class of decision problems solvable by a probabilistic Turing machine in polynomial time, with an error probability of less than 1/2 for all instances. The abbreviation **PP** refers to probabilistic polynomial time. The complexity class was defined by Gill in 1977. If a decision problem is in **PP**, then there is an algorithm for it that is allowed to flip coins and make random decisions. It is guaranteed to run in polynomial time.

If the answer is YES, the algorithm will answer YES with probability more than 1/2. If the answer is NO, the algorithm will answer YES with probability less than or equal to 1/2. In more practical terms, it is the class of problems that can be solved to any fixed degree of accuracy by running a randomized, polynomial-time algorithm a sufficient (but bounded) number of times.

An alternative characterization of **PP** is the set of problems that can be solved by a nondeterministic Turing machine in polynomial time where the acceptance condition is that a majority (more than half) of computation paths accept. Because of this some authors have suggested the alternative name *Majority-P*.[2]

2.0 Learning Outcomes/Objectives

At the end of this lesson you should be able to:

1. Compare Probabilistic Complexity (PP) to other complexity classes
2. Describe Complete problems and other properties
3. Proof that PP is closed under complement
4. Explain Other equivalent complexity classes

3.0Learning Activities

3.1 Probabilistic Complexity(PP)

A language L is in **PP** if and only if there exists a probabilistic Turing machine M, such that

1. M runs for polynomial time on all inputs
2. For all x in L, M outputs 1 with probability strictly greater than 1/2
3. For all x not in L, M outputs 1 with probability less than or equal to 1/2.
4. Alternatively, **PP** can be defined using only deterministic Turing machines. A language L is in **PP** if and only if there exists a polynomial p and deterministic Turing machine M, such that
5. M runs for polynomial time on all inputs
6. For all x in L, the fraction of strings y of length $p(|x|)$ which satisfy $M(x,y) = 1$ is strictly greater than 1/2
7. For all x not in L, the fraction of strings y of length $p(|x|)$ which satisfy $M(x,y) = 1$ is less than or equal to 1/2.

In both definitions, "less than or equal" can be changed to "less than" (see below), and the threshold 1/2 can be replaced by any fixed rational number in (0,1), without changing the class.

Self-Assessment Exercise(s) (SAE 1)

Explain Probabilistic Complexity (PP)

Answer

A language L is in **PP** if and only if there exists a probabilistic Turing machine M, such that

1. M runs for polynomial time on all inputs
2. For all x in L, M outputs 1 with probability strictly greater than 1/2
3. For all x not in L, M outputs 1 with probability less than or equal to 1/2.

3.2 PP vs BPP

BPP is a subset of **PP**; it can be seen as the subset for which there are efficient probabilistic algorithms. The distinction is in the error probability that is allowed: in **BPP**, an algorithm must give correct answer (YES or NO) with probability exceeding some fixed constant c greater than 1/2, such as 2/3 or 501/1000. If this is the case, then we can run the algorithm a polynomial number of times and take a majority vote to achieve any desired probability of correctness less than 1, using the Chernoff bound. This number of repeats increases if c becomes closer to 1/2, but it does not depend on the input size n.

The important thing is that this constant c is not allowed to depend on the input. On the other hand, a **PP** algorithm is permitted to do something like the following:

1. On a YES instance, output YES with probability $1/2 + 1/2^n$, where n is the length of the input.
2. On a NO instance, output YES with probability $1/2 - 1/2^n$.

Because these two probabilities are so close together, especially for large n, even if we run it a large number of times it is very difficult to tell whether we are operating on a YES instance or a NO instance. Attempting to achieve a fixed desired probability level using a majority vote and the Chernoff bound requires a number of repetitions that is exponential in n.

3.3 PP compared to other complexity classes

PP contains BPP, since probabilistic algorithms described in the definition of BPP form a subset of those in the definition of PP. PP also contains NP. To prove this, we show that the NP-complete satisfiability problem belongs to PP. Consider a probabilistic algorithm that, given a formula $F(x_1, x_2, ..., x_n)$ chooses an assignment $x_1, x_2, ..., x_n$ uniformly at random.

Then, the algorithm checks if the assignment makes the formula F true. If yes, it outputs YES. Otherwise, it outputs YES with probability 1/2 and NO with probability 1/2. If the formula is unsatisfiable, the algorithm will always output YES with probability 1/2. If there exists a satisfying assignment, it will output YES with probability more than 1/2 (exactly 1/2 if it picked an unsatisfying assignment and 1 if it picked a satisfying assignment, averaging to some number greater than 1/2).

Thus, this algorithm puts satisfiability in PP. As SAT is NP-complete, and we can prefix any deterministic polynomial-time many-one reduction onto the PP algorithm, NP is contained in PP. Because PP is closed under complement, it also contains co-NP. Furthermore, PP contains MA[3], which subsumes the previous two inclusions.

***Please insert relevant images/graphics**

PP also contains BQP, the class of decision problems solvable by efficient polynomial time quantum computers. In fact, BQP is low for PP, meaning that a PP machine achieves no benefit from being able to

solve BQP problems instantly. The class of polynomial time on quantum computers with postselection, PostBQP, is equal to PP (see #PostBQP below).

Furthermore, PP contains QMA, which subsumes inclusions of MA and BQP.

A polynomial time Turing machine with a PP oracle (P^{PP}) can solve all problems in PH, the entire polynomial hierarchy. This result was shown by Seinosuke Today in 1989 and is known as Toda's theorem. This is evidence of how hard it is to solve problems in PP. The class #P is in some sense about as hard, since $P^{\#P} = P^{PP}$and therefore $P^{\#P}$ contains PH as well.

PP strictly contains TC^0, the class of constant-depth, unbounded-fan-in uniform boolean circuits with majority gates. (Allender 1996, as cited in Burtschick 1999).

PP is contained in PSPACE. This can be easily shown by exhibiting a polynomial-space algorithm for MAJSAT, defined below; simply try all assignments and count the number of satisfying ones.

PP is not contained in SIZE(n^k) for any k (proof).

Self-Assessment Exercise(s) (SAE 2)

Compare _Probabilistic Complexity (PP) with other complexity classes

Self-Assessment Answer

PP contains BPP, since probabilistic algorithms described in the definition of BPP form a subset of those in the definition of PP.

PP also contains NP. To prove this, we show that the NP-complete satisfiability problem belongs to PP.

Furthermore, PP contains MA, which subsumes the previous two inclusions.

PP also contains BQP, the class of decision problems solvable by efficient polynomial time quantum computers. In fact, BQP is low for PP, meaning that a PP machine achieves no benefit from being able to solve BQP problems instantly. The class of polynomial time on quantum computers with postselection, PostBQP, is equal to PP (see #PostBQP below).

Furthermore, PP contains QMA, which subsumes inclusions of MA and BQP.

A polynomial time Turing machine with a PP oracle (P^{PP}) can solve all problems in PH, the entire polynomial hierarchy. This result was shown by Seinosuke Today in 1989 and is known as Toda's theorem. This is evidence of how hard it is to solve problems in PP. The class #P is in some sense about as hard, since $P^{\#P} = P^{PP}$and therefore $P^{\#P}$ contains PH as well.

PP strictly contains TC^0, the class of constant-depth, unbounded-fan-in uniform boolean circuits with majority gates. (Allender 1996, as cited in Burtschick 1999).

PP is contained in PSPACE. This can be easily shown by exhibiting a polynomial-space algorithm for MAJSAT, defined below; simply try all assignments and count the number of satisfying ones.

3.4 Complete problems and other properties

Unlike **BPP**, **PP** is a syntactic, rather than semantic class. Any polynomial-time probabilistic machine recognizes some language in **PP**. In contrast, given a description of a polynomial-time probabilistic machine, it is undecidable in general to determine if it recognizes a language in **BPP**.

PP has natural complete problems, for example, **MAJSAT**. **MAJSAT** is a decision problem in which one is given a Boolean formula F. The answer must be YES if more than half of all assignments $x_1, x_2, \ldots, x_n$ make F true and NO otherwise.

3.5 Proof that PP is closed under complement

Let *L* be a language in **PP**. Let L^c denote the complement of *L*. By the definition of **PP** there is a polynomial-time probabilistic algorithm *A* with the property that $x \in L \Rightarrow \Pr[A \text{ accepts } x] > 1/2$ and $x \notin L \Rightarrow \Pr[A \text{ accepts } x] \leq 1/2$. We claim that without loss of generality, the latter inequality is always strict; once we do this the theorem can be proved as follows. Let A^c denote the machine which is the same as *A* except that A^c accepts when *A* would reject, and vice-versa. Then $x \in L^c \Rightarrow \Pr[A^c \text{ accepts } x] > 1/2$ and $x \notin L^c \Rightarrow \Pr[A^c \text{ accepts } x] < 1/2$, which implies that L^c is in **PP**.

Now we justify our without loss of generality assumption. Let $f(|x|)$ be the polynomial upper bound on the running time of *A* on input *x*. Thus *A* makes at most $f(|x|)$ random coin flips during its execution. In particular, $x \in L \Rightarrow \Pr[A \text{ accepts } x] \geq 1/2 + 1/2^{f(|x|)}$ since the probability of acceptance is an integer multiple of $1/2^{f(|x|)}$. Define a machine *A′* as follows: on input *x*, *A′* runs *A* as a subroutine, and rejects if *A* would reject; otherwise, if *A* would accept, *A′* flips $f(|x|) + 1$ coins and rejects if they are all heads, and accepts otherwise. Then

$x \notin L \Rightarrow \Pr[A' \text{ accepts } x] \leq 1/2 \cdot (1 - 1/2^{f(|x|)+1}) < 1/2$ and

$x \in L \Rightarrow \Pr[A' \text{ accepts } x] \geq (1/2 + 1/2^{f(|x|)}) \cdot (1 - 1/2^{f(|x|)+1}) > 1/2$. This justifies the assumption (since *A′* is still a polynomial-time probabilistic algorithm) and completes the proof.

David Russo proved in his 1985 Ph.D. thesis [5] that **PP** is closed under symmetric difference. It was an open problem for 14 years whether **PP** was closed under union and intersection; this was settled in the affirmative

by Beigel, Reingold, and Spielman.[6] Alternate proofs were later given by Li[7] and Aaronson (see #PostBQP below).

Self-Assessment Exercise(s) (SAE 3)

Explain complete problems and other properties

Self-Assessment Answer

Answer

PP is a syntactic, rather than semantic class. Any polynomial-time probabilistic machine recognizes some language in **PP**. In contrast, given a description of a polynomial-time probabilistic machine, it is undecidable in general to determine if it recognizes a language in **BPP**.

PP has natural complete problems, for example, **MAJSAT**. **MAJSAT** is a decision problem in which one is given a Boolean formula F.

3.6 Other Equivalent Complexity Classes

***Please introduce the Other Equivalent Complexity Classes**

3.6.1 Post BQP

The quantum complexity class **BQP** is the class of problems solvable in polynomial time on a quantum Turing machine. By adding postselection, a larger class called **PostBQP** is obtained. Informally, postselection gives the computer the following power: whenever some event (such as measuring a qubit in a certain state) has nonzero probability, you are allowed to assume that it takes place.[8] Scott Aaronson showed in 2004 that **PostBQP** is equal to **PP**.[4][9] This reformulation of **PP** makes it easier to show certain results, such as that **PP** is closed under intersection (and hence, under union), that **BQP** is low for **PP**, and that **QMA** is contained in **PP**.

3.6.2 PQP

PP is also equal to another quantum complexity class known as **PQP**, which is the unbounded error analog of **BQP**. It denotes the class of decision problems solvable by a quantum computer in polynomial time, with an error probability of less than 1/2 for all instances.

4.0 Conclusion

In this unit we have came out with the comparism of _Probabilistic Complexity (PP) with other complexity classes, describe _complete problems and other properties, _proof that PP is closed under complement and explain other equivalent complexity classes. All these are the concepts of Probabilistic Complexity (PP).

5.0 Summary

In this Module 5 Study Unit 3, the following aspects have been discussed:

1. Comparism of _Probabilistic Complexity (PP) to other complexity classes
2. _Complete problems and other properties
3. _Proof that PP is closed under complement
4. Other equivalent complexity classes

6.0 Tutor Marked Assignments and Marking Scheme

1. How does BPP came about?
2. Proof that PP is closed under complement.
3. Explain other equivalent complexity classes.

7.0 References/Further Reading

Gill, "Computational complexity of probabilistic Turing machines." *SIAM Journal on Computing*, 6 (4), pp. 675–695, 1977.

Lance Fortnow. Computational Complexity: Wednesday, September 4, 2002: Complexity Class of the Week: PP.

N.K. Vereshchagin, "On the Power of PP" [1]

Aaronson, Scott (2005). "Quantum computing, postselection, and probabilistic polynomial-time". *Proceedings of the Royal Society A* **461** (2063): 3473–3482. DOI:10.1098/rspa.2005.1546.. Preprint available at [2]

David Russo (1985). "Structural Properties Of Complexity Classes". *Ph.D Thesis* (University of California, Santa Barbara).

R. Beigel, N. Reingold, and D. A. Spielman, "PP is closed under intersection", *Proceedings of ACM Symposium on Theory of Computing 1991*, pp. 1–9, 1991.

Lide Li (1993). "On the Counting Functions". *Ph.D Thesis* (University of Chicago).

Aaronson, Scott. "The Amazing Power of Postselection".

Aaronson, Scott (2004-01-11). "Complexity Class of the Week: PP". *Computational Complexity Weblog*

Papadimitriou, C. (1994). "chapter 11". *Computational Complexity*. Addison-Wesley..

Allender, E. (1996). "A note on uniform circuit lower bounds for the counting hierarchy". *Proceedings 2nd International Computing and Combinatorics Conference (COCOON)*. Lecture Notes in Computer Science. **1090**. Springer-Verlag. pp. 127–135..

Burtschick, Hans-Jörg; Vollmer, Heribert (1999). *Lindström Quantifiers and Leaf Language Definability*. ECCC TR96-005.