Notes on

CSE-103: Computational Models

Mann Malviya
mannmalviya15@gmail.com

These notes cover the theory of computational models. The primary sources used to write these notes were:

- CSE 103: Computational Models — UCSC, Spring 2025 taught by Assistant Prof. Daniel Fremont
- Textbook: *Introduction to the Theory of Computation* by Michael Sipser

Contents

Introduction **3**
- Textbooks . . . 3
- What is this Doc? . . . 3

Lecture 1 **4**
- Overview of the Course . . . 4
- Learning Objectives . . . 4
- Outline of the Course . . . 4

Lecture 2 **5**
- What is Computation? . . . 5
- Input . . . 5
- Output . . . 7
- Procedure . . . 7
- Exercise Solutions . . . 8

Lecture 3 **9**
- Finite Automata . . . 9
- Examples of Finite Automata . . . 10
- Deterministic Finite Automata (DFA) . . . 12

Lecture 4 **13**
- DFAs continued . . . 13
- A proof using PMI . . . 13
- Extended transition function . . . 13

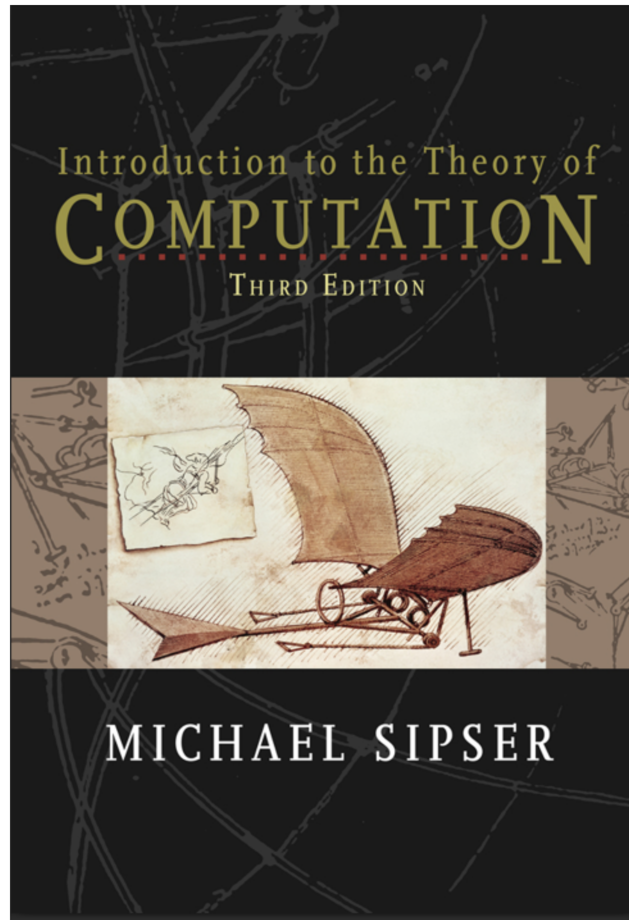
Lecture 5 **14**
- Extended transition function continued . . . 14
- Some Definitions . . . 14
- Nondeterministic Finite Automata (NFA) . . . 14

Lecture 6 **15**

Introduction

Textbooks

1. Introduction to the Theory of Computation by Michael Sipser



Introduction to the Theory of Computation
by Michael Sipser

What is this Doc?

This document shall contain my notes for the class CSE-103: Computational Models offered at UCSC, taught by Assistant Prof. Daniel Fremont. This document will contain notes from the lectures(possibly verbatim?) and may contain some additional information, either from the text or other sources that I find useful.

Lecture 1

Overview of the Course

- This course is called **Computational Models** what does that actually mean?
    - SW

Learning Objectives

After taking this course, you will be able to:

1. Interpret and design finite automata (DFAs and NFAs) and regular expressions.
2. Interpret and design context-free grammars (CFGs) and pushdown automata.
3. Prove basic properties of regular and context-free languages.
4. Interpret and design Turing machines(TMs).
5. Prove basic languages are decidable or Turing-recognizable.
6. Construct reductions between problems and apply such reductions to establish undecidability of problems
7. Construct polynomial-time algorithms/verifiers and polynomial-time reductions and use them to show languages are in P, NP, or are NP-complete.

Outline of the Course

Lecture 2:

We are building up to a simple model of computation: The Finite Automaton.

What is Computation?

- A computation is some kind of procedure that takes some input and produces some output.

$$\text{Input} \longrightarrow \text{Procedure} \longrightarrow \text{Output}$$

A Computation

- We will talk about how we can model each of the 3 parts, the **input**, the **procedure**, and the **output**.

Input

$$\boxed{\text{Input}} \longrightarrow \text{Procedure} \longrightarrow \text{Output}$$

- The input will always be a finite sequence of symbols, e.g. ”001101” or ”abca”.

> **Definition:**
>
> The set of allowed symbols is called the **alphabet** (by analogy to natural language) and is denoted by Σ.

Example: The binary alphabet $\{0,1\}$

- The only thing we will assume about the **alphabet** Σ is that it's finite.

> **Definition:**
>
> A **string** or **word** over Σ is a finite sequence of symbols from Σ.

Example: 00, 101, 000 are words over the binary alphabet.

- These words/strings are going to be the inputs to the algorithms we will talk about in this class, one could imagine there are plenty of algorithms that operate on other kinds of input, like images, sounds, videos etc. but you can find ways of encoding them in binary. Then you can treat any arbitrary kind of input as being a finite sequence of 0's and 1's.
- If w is a string, $|w|$ is the no. of symbols in w.

Example: $|110| = 3$

- The empty string is denoted by ε and has a length zero i.e., $|\varepsilon| = 0$.

- When given a string, w if you want to refer to an individual symbol within the sting then you do so by w_i, for any i between 1 and $|w|$ (i.e., $1 \leq i \leq |w|$), w_i is the ith symbol of w. We are indexing starting from 1, this is just a convention.

  Example: $w = 110$ then $w_1 = 1$, $w_2 = 1$, $w_3 = 0$.

- Another very common operation we will need to use on strings is **concatenation** which is taking 2 strings and joining them together. We write concatenation as multiplication, so $w = xy$ means w consists of the symbol of x followed by the symbols of y.

  Example: $x = 001$ and $y = 10$ then $xy = 00110$ and $yx = 10001$. Notice that concatenation is not commutative.

  Definition:

  For any non-negative(≥ 0) integer k, i.e., $k \in \mathbb{Z}^+$, Σ^k is the set of all strings over Σ of length k.

  Example: $\{0,1\}^2 =$ all length 2 binary strings $= \{00, 01, 10, 11\}$

- We write $\Sigma^{\leq k}$ for the set of strings over Σ of length $\leq k$.

  Example: if $\Sigma = \{0,1\}$ then,

$$\begin{aligned}\Sigma^{\leq 2} &= \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \\ &= \{\varepsilon\} \cup \{0,1\} \cup \{00,01,10,11\} \\ &= \{\varepsilon, 0, 1, 00, 01, 10, 11\}\end{aligned}$$

  Note: For any alphabet Σ, Σ^0 is the set of words of length zero, of which there is exactly one: the empty string ε.

$$\Sigma^0 = \{\varepsilon\} \neq \varnothing$$

- We write Σ^*(sigma-star) for the set of all strings of any length over Σ. Formally,

$$\Sigma^* = \bigcup_{k \geq 0} \Sigma^k$$

  This is an infinite set.

  Example: $\{0,1\}^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, \ldots\}$

This is going to suffice for us to talk about the input to our algorithms, because we will just assume some kind of standardized encoding of all other kinds of inputs into binary strings. The details of how we do the encoding will not be important at least in this class.

We will assume a standardized binary encoding of non-string datatypes, so that we can treat all inputs as binary strings. This shouldn't be too hard to believe because on real computers, everything is stored as 0 and 1's anyways.

Exercise:

1. How many elements are in the set $\{a,b,c\}^3$?
2. How many elements are in the set $\{a\}^3$?

Output

- We will make a somewhat restrictive assumption, we are only going to look at algorithms who's answer is YES/NO or TRUE/FALSE[1]. These are called **decision problems**. In these kinds of problems we are simply trying to say YES or NO, we are not going to deal with problems that need a string output.

> **Definition:**
>
> Problems for which every possible inputs output is either YES or NO are called **decision problems**.

Example: "Does a binary string contain an even number of 1's" is a Decision problem

$$011 \to \text{YES}$$
$$0100 \to \text{NO}$$
$$\varepsilon \to \text{YES}$$

> **Definition:**
>
> To fully specify a decision problem, it's enough to identify the strings with answers "YES", this set is called the **language**[2] of the decision problem. A language $\mathscr{L}$ over an alphabet Σ is a subset, $\mathscr{L} \subseteq \Sigma^*$.

- The decision problem for $\mathscr{L}$ is to decide whether a given string $w \in \Sigma^*$ is in $\mathscr{L}$, i.e. $w \in \mathscr{L}$?

Example: $\mathscr{L}_{\text{PRIME}} =$ "binary string encoding prime numbers"

$$10 \in \mathscr{L}_{\text{PRIME}}$$
$$101 \in \mathscr{L}_{\text{PRIME}}$$
$$1001 \notin \mathscr{L}_{\text{PRIME}}$$

Procedure

[1] For the purposes of this class, it's sufficient for us to just deal with TRUE/FALSE questions, not too much interesting new stuff happens if you consider more complicated things, so we will not worry about that here.

[2] because it's a set of words, an analogy from natural language

- In a decision problem, you take in a finite string as input and you need to output either YES or NO, one question is how do you make the decision? Thats where the computational model is going to come in, you can think of the **procedure** as just a way of computing the mapping from inputs to outputs.

> Definition:
>
> The **procedure** is a mapping from the inputs to the outputs.

- You can model the procedure mathematically as just a function, because thats what a function does, it tells you for every possible input, what the output is.

- In general the function can be written as,

$$f : \Sigma^* \to \{\underset{\text{no}}{0}, \underset{\text{yes}}{1}\}$$

The language $\mathscr{L}$ for f would be the set of all inputs x such that $f(x) = 1$,

$$\mathscr{L} = \{x \in \Sigma^* | f(x) = 1\}$$

Note: A well defined function need not have an actual algorithm for computing $f(x)$ from x.

Example:

$$f(x) = \begin{cases} 1 & \text{if x encodes a python program that terminates} \\ 0 & \text{otherwise} \end{cases}$$

Later we will see that this function is **not** computed by any algorithm.

Next lecture we will define **finite automata** as a restricted class of such functions.

Exercise Solutions

1. 3 symbols to pick since we want to find $|\Sigma^3|$ which only contains words of length 3, we have 3 options fore each symbol(a,b or c) so by the multiplication principle from combinatorics, we have $3 \cdot 3 \cdot 3 = 3^3 = 27$ words of length 3.
2. $|\{a\}^3| = 1^3 = 1$

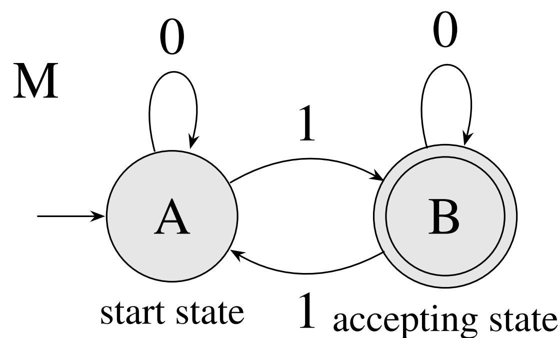
Lecture 3:

Last Lecture we spoke about the process of going from an input to an output(the procedure) we going to talk about several such models, the first being **Finite Automata**.

Finite Automata

- We will first cover the intuition for finite automata and then cover the formal definition.
- Intuition: Think of finite automata as "computers" with limited memory.
- Programming languages like C/C++ provide mechanisms to allocate new memory, but in finite automaton you cannot do that, you have a fixed amount of memory which is set once and for all and it's independent of the size of the input given.
- There's a convenient way to draw finite automata as directed graphs, representing each state as a node and the arrows/edges between them as transitions, with each edge labeled with the input symbol that causes the transition.
- Example:
  - Let M be a machine that reads an input string one symbol at a time.
  - The machine M has a **finite** number of modes or states.
  - The machine M remembers things by being in different states.
  - The machine **transitions** to the next state based on the input symbol and the current state.
  - After reaching all symbols, M **accepts**(answer YES) if we end up in an "accepting state" otherwise it **rejects**(answer NO).

    Let M be a finite state machine with alphabet $\Sigma = \{0, 1\}$.



Lets run the automaton on the strings 001 and 110 and see whether `M` accepts or rejects them.

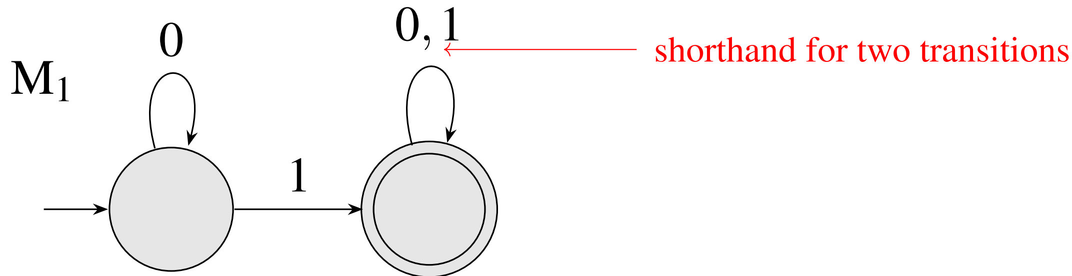
string	0	0	1	
path	A	A	A	B
string	1	1	0	
path	A	B	A	A

We can see that the automaton M accepts the string 001 and rejects the string 110, because 001 ends in state A(which isn't accepting) while 110 ends in state B(which is accepting).

$$\mathscr{L}(\mathrm{M}) = \text{"all strings with an odd number of 1s"}$$

Examples of Finite Automata

- **Example 1:**
  $\Sigma = \{0,1\}$



We can notice that M_1 rejects ε, 0, 00, 000, etc. and accepts 1, 101, etc.

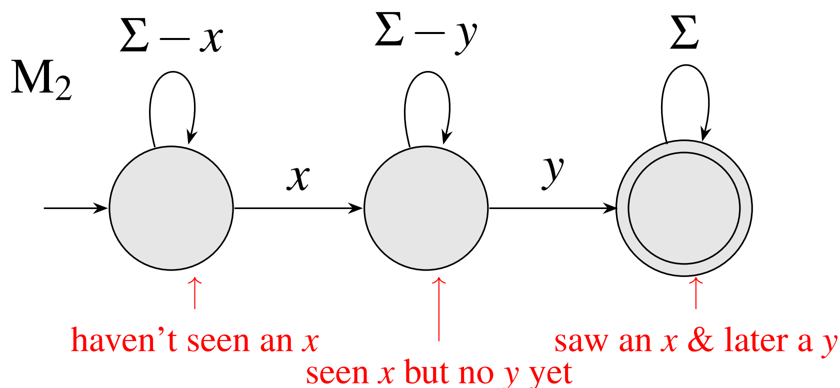
In this DFA the states are acting like memory as they keep track of whether we've seen a 1 or not.

The **first state** represents: we haven't seen a 1 yet.

The **second state** represents: we have now seen a 1.

$$\mathscr{L}(M_1) = \text{"all strings containing a 1"}$$

- **Example 2:**
  $\Sigma = \{w,x,y,z\}$



M_2 rejects yx, xw, etc. and accepts xy, xyy, $xwzy$, etc.

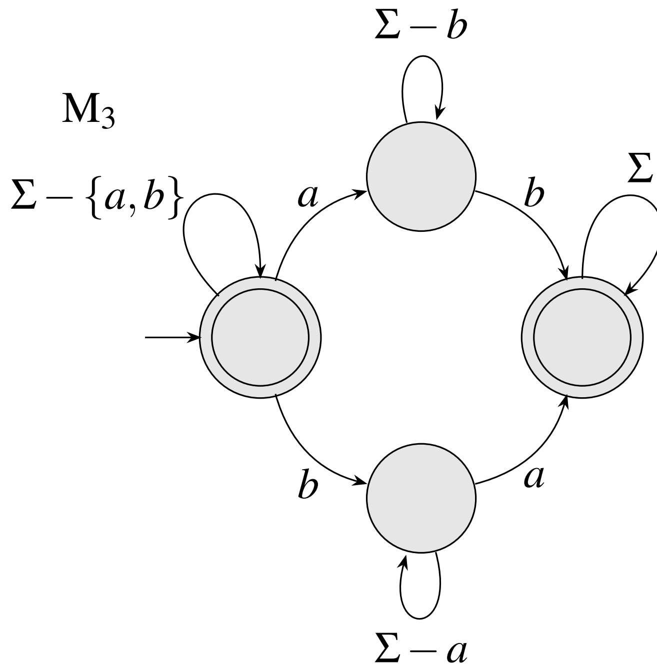
The **first state** represents: we haven't seen an x yet(because we stay in the state until we see an x).

The **second state** represents: we have seen an x but since then we have not seen y.

The **third state** represents: we have seen an x and then we later saw a y.

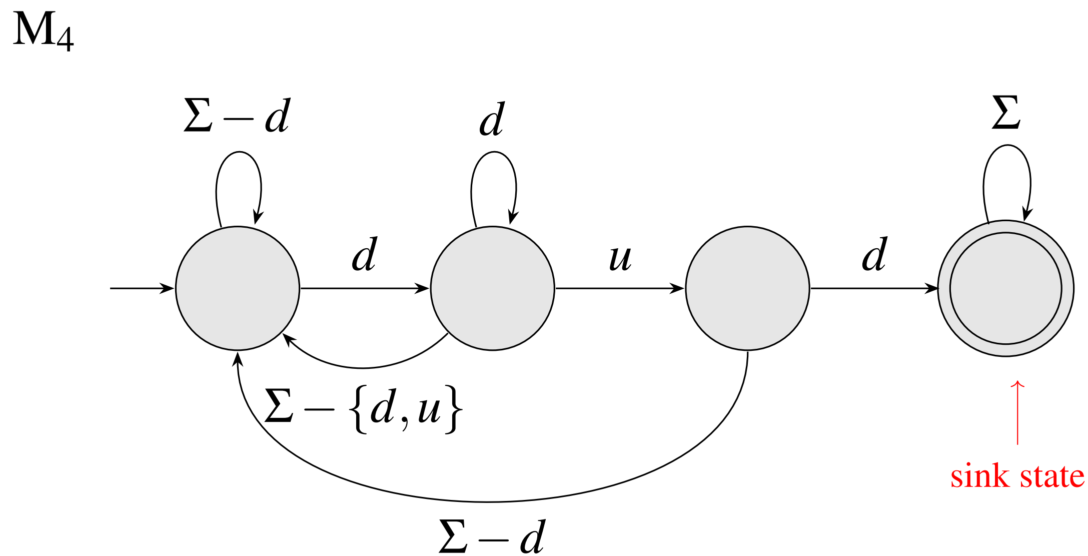
$$\mathscr{L}(M_2) = \text{"all strings which have an } x \text{ eventually followed by a } y\text{"}$$

- **Example 3:**
  $\Sigma = \{a, b, c, \ldots, z\}$



$\mathscr{L}(M_3) =$ "all strings containing either both a and b or neither a nor b"

- **Example 4:**
  $\Sigma = \{a, b, c, \ldots, z\}$



The accepting state in M_4 is a sink state, meaning that once we reach it, we can never leave it.

$\mathscr{L}(M_4) =$ "all strings with a "dud" substring"

Deterministic Finite Automata (DFA)

Definition:

A **DFA**(Deterministic Finite Automata) is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where:

- Q is a finite set of states
- Σ is a finite alphabet
- $\delta : Q \times \Sigma \to Q$ is the **transition function**
- $q_0 \in Q$ is the start (or initial state)
- $F \subseteq Q$ is the set of **accepting states** (or final states)

So we have now covered what a DFA is, but we still need to define how a DFA runs ob a given input string.

Definition:

A DFA M **accepts** a string $w = w_1 w_2 ... w_n$ iff there exists an **accepting path** for w: a sequence of states $r_0, r_1, ..., r_n$ such that:

1. $r_0 = q_0$ (start in the start state)
2. $r_n \in F$ (end in an accepting state)
3. For every $i \in \{0, 1, ..., n-1\}$, $\delta(r_i, w_{i+1}) = r_{i+1}$ (path follows transitions)[3]

Definition:

The **language** $\mathscr{L}(M)$ of M is $\{x \in \Sigma^* \mid M \text{ accepts } x\}$

---

[3]We need to say that you follow transitions properly through the automaton, making sure you didn't just jump around to random states

Lecture 4:

DFAs continued

A proof using PMI

Extended transition function

Lecture 5:

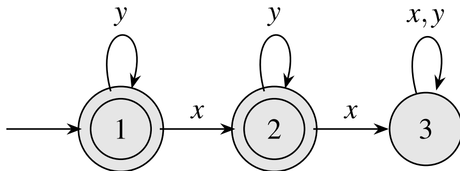
Extended transition function continued

Some Definitions

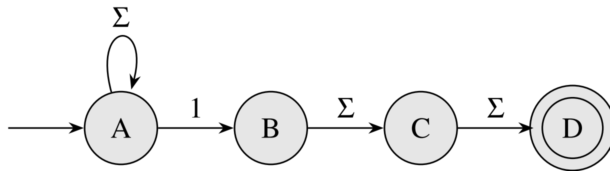
Nondeterministic Finite Automata (NFA)

The following is all the state machines drawn in this lecture:

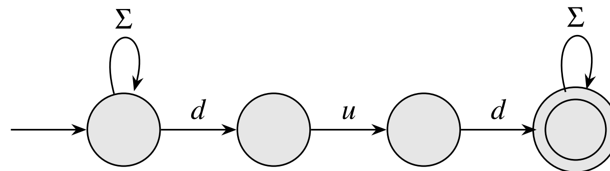
M



M



NFA:



Lecture 6: