V CSIS
SJBIT

Automata Theory and Computability

Programming Tools

* IBM 7090 – assembly level programming language
* Fortran – (1957) made it possible for people to write programs that looked more straightforward like mathematics.
* 1970 - IBM 360 Series of Computers was in widespread use for both business and Scientific Computing. To submit a job, one keyed onto punch cards a set of commands in OS/360 JCL (Job Control Language)
* By 2000, many of them know that the Java method allows the user to select a file, which is read in and parsed using whitespace delimiters.
* APL (A Programming language) has been introduced
* * * There are mathematical properties, both of problems and of algorithms for solving problems, that depend on neither the details of today's technology nor the programming fashion. Automata theory addresses some of these properties.
* Automata theory is useful in two key ways:
  1) It provides a set of abstract structures that are useful for solving certain classes of problems. These abstract structures can be implemented on whatever hardware/software platforms is available.

2) It defines a provable limits to what can be computed, regardless of processor speed or memory size. An understanding of these limits helps us to focus our design effort in areas in which it can pay off.

The focus will be on analyzing problems, rather than comparing solutions to problems. The goal will be to discover fundamental properties of the problems themselves.

1) Is there any computational soln to the problem? If not, is there a restricted but useful variation of the problem for which a soln does exist?
2) If a soln exists, can it be implemented using some fixed amount of memory?
3) If a soln exists, how efficient is it? (space & time)
4) Are these groups of problems that are equivalent in the sense that if there is an efficient soln to one member of the group there is an efficient soln to all the others?

Applications of Theory of Automation.

Some of the applications are:

1) Languages - Enable both machine/machine and person/machine communication. Without them, none of today's applications of computing could exist.

2) Both the design and the implementation of modern programming languages rely heavily on the theory of Context-free languages ~~CFGs are~~ Context-free grammars are used to document the language's Syntax and ~~theory~~ form the basis for the parsing techniques that all Compilers use

3) Programs to manage our words, Check our grammar, Search the world wide web and translate from one language to another also rely on the theory of Context-free languages.

4) Systems as diverse as parity Checkers, Vending machines, Communication protocols and building Security devices Can be straightforwardly described as finite State machines.

5) many interactive Video games are finite State machines.

6) DNA molecules as well as the protiens that they describe are Strings that are made up of Symbols drawn from small alphabets. Computational biologists Exploit many of the Same tools that are based on both finite State nlcs and CFGs.

7) Security is perhaps the most important property of many Computer Systems. These Cannot Exist a general purpose method for automatically verifying arbitrary Security properties of programs. The Complexity serve as the basis for powerful Encryption techniques.

8) Undecidability

9) Graphs

10)

Languages and Strings

* An alphabet is a finite non-Empty Set of Symbols often denoted by Σ.

Ex: (a) English alphabet – $\{a, b, c, \ldots z\}$ Ex: aabc, aaaa

(b) Binary alphabet – $\{0, 1\}$ Ex: 0010, 1100

Strings

~~Strings~~ :

* A String is a finite Sequence, ~~of possibly empty~~ of Symbols Chosen from Some alphabet Ex: 1011

* Given any alphabet Σ, the Shortest String that can be formed from Σ is the Empty String, which is denoted as ϵ.

* The Set of all possible Strings over an alphabet Σ is written as Σ^*. (Klene star operator = *)

Functions on Strings

* The length of a String S, denoted by $|S|$, is the no of Symbols in S. Ex: $|\epsilon| = 0$

$|10001| = 5$

For any Symbol c and String S, the function $\#_c(S)$ defines the no of times the Symbol c occurs in S. Ex: $\#_a(abbaaaa) = 4$.

* Concatenation of two strings s and t, written as st or simply st, is the string formed by appending t to s.

Ex: If x = good and y = bye, then xy = goodbye.

So $|xy| = |x| + |y|$

Empty String ϵ, is the identity for concatenation of strings. So $\forall x (x\epsilon = \epsilon x = x)$.

concatenation of strings is associative, ie $\forall s, t, w$

$((st)w = s(tw))$

* String replication - For each String w and each natural no i, the String w^i is defined as

$$w^0 = \epsilon$$
$$w^{i+1} = w^i w$$

Ex: $a^3 = aaa$, $(bye)^2 = byebye$, $a^0 b^3 = bbb$

* String reversal - For each string w, the reverse of w, written as w^R and is defined as

If $|w| = 0$ then $w^R = w = \epsilon$

If $|w| \geq 1$ then $\exists a \in \Sigma (\exists u \in \Sigma^* (w = ua))$ (ie the last character of w is a)

then $w^R = au^R$

Theorem (1): Concatenation and Reverse of Strings

Statement: If w and x are strings, then $(wx)^R = x^R w^R$

Ex: $(\underbrace{name}_{w}\underbrace{tag}_{x})^R = (tag)^R (name)^R$ = gatemon.

Proof: Proof is by induction on $|x|$

Basis: $|x| = 0$. Then $x = \epsilon$ and $(wx)^R = (w\epsilon)^R$ (LHS)

$= (w)^R$

$= \epsilon w^R$

$= \epsilon^R w^R$

$= x^R w^R$ - RHS

Hypothesis: Assume that $(|x| = n) \rightarrow ((wx)^R = x^R w^R)$

Prove: $\forall n \geq 0$

Induction: Prove that for $|x| = n+1$, $(wx)^R = x^R w^R$

Consider any string x, where $|x| = n+1$, then $x = ua$ for some character a and $|u| = n$.

LHS So $(wx)^R = (w(ua))^R$ rewrite x as ua

$= ((wu)a)^R$ associativity of concatenation

$= a(wu)^R$ defn of reversal

$= a u^R w^R$ Induction Hypothesis

$= (au^R) w^R$ associativity of concatenation

$= (ua)^R w^R$ defn of reversal

$= x^R w^R$ - RHS rewrite ua as x.

Relations on Strings

* A string s is a substring of a string t, iff s occurs contiguously as part of t.

  Ex: aaa is a substring of aaabbbaaa

  aaaaa is not a substring of aaabbbaaa

* A string s is a proper substring of a string t iff s is a substring of t and $s \neq t$.

  Every string is a substring of itself (although not a proper substring). Empty string ϵ, is a substring of every string.

* A string s is a prefix of t iff $\exists x \in \Sigma^* (t = sx)$.
* A string s is a proper prefix of a string t iff s is a prefix of t and $s \neq t$.

  Every string is a prefix (although not a proper prefix) of itself. The empty string ϵ, is a prefix of every string. Ex: Prefixes of abba are: a, ab, abb, abba.

* A string s is a suffix of t iff $\exists x \in \Sigma^* (t = xs)$.
* A string s is a proper suffix of a string t iff s is a suffix of t and $s \neq t$.

  Every string is a suffix (although not a proper suffix) of itself. The empty string ϵ, is a suffix of every string. Ex: Suffixes of abba are: ϵ, a, ba, bba, abba.

Languages

* A language is a (finite or infinite) set of strings over a finite alphabet Σ. We use the notation Σ_L to mean the alphabet from which the strings in the language L are formed

Ex: $\Sigma = \{a, b\}$ $\Sigma^* = \{\epsilon, a, b, aa, bb, ab, ba, aaa, aab, \ldots\}$

Examples of languages over Σ are ϕ, $\{\epsilon\}$, $\{a, b\}$,

- $\{\epsilon, a, aa, aaa, aaaa, aaaaa\}$

  $\{\epsilon, a, aa, aaa, aaaa, aaaaa, \ldots\}$

Techniques for Defining languages

Since languages are sets, we can define them using any of the set-defining techniques: Enumeration & Characteristic function

Enumeration: listing all the strings in a language explicitly

Ex: $L = \{0, 1, 00, 10, 11\}$ over $\Sigma = \{0, 1\}$

Characteristic function:- A predicate that is true of every element in the set and false of everything else

Ex: $L = \{w \in \{a,b\}^* : \text{all } a\text{'s precede all } b\text{'s in } w\}$

Strings ϵ, aa, aabbb and bb are in L.

Strings aba, ba, and abc are not in L.

Ex(2) Let $L = \{x : \exists y \in \{a,b\}^* (x = ya)\}$ - Strings ↑ Ends with a.

Strings a, aa, aaa, bbaa, and ba are in L.

Strings ∈, bab, and bca are not in L

Ex(3): Let $L = \{x\#y : x, y \in \{0,1,2,3,4,5,6,7,8,9\}^*$ and when x and y are viewed as the decimal representations of natural nos, $Square(x) = y\}$.

Strings 3#9, 12#144 are in L.

Strings 3#8, 12, and 12#12#12 are not in L.

Ex(4): Empty language

$L = \{\} = \phi$. L is a language that Contains no Strings.

Ex(5): Language that Contains Empty String

$L = \{\epsilon\}$. Note: This L is different from ϕ.

Ex(6): Using prefix Relation

$L_1 = \{w \in \{a,b\}^* : \text{no prefix of } w \text{ contains } b\}$

$= \{\epsilon, a, aa, aaa, aaaa, aaaaa, \ldots\}$

$L_2 = \{w \in \{a,b\}^* : \text{no prefix of } w \text{ Starts with } b\}$

$= \{w \in \{a,b\}^* : \text{the first Character of } w \text{ is } a\} \cup \{\epsilon\}$

$= \{\epsilon, a, ab, aab, aba, abb, \ldots\}$

$L_3 = \{w \in \{a,b\}^* : \text{Every prefix of } w \text{ Starts with } b\}$

$= \phi$ because ϵ is a prefix of Every String Since ϵ does not Start with b, no Strings meet L_3's requirement.

Ex(7): Using Replication to Define a language

let $L = \{a^n : n \geq 0\}$, $L = \{\epsilon, a, aa, aaa, \ldots\}$

Lexicographic order ($<_L$): Shorter strings precede longer ones of strings that are the same length, sort them in dictionary order

2) Cardinality of a language

Ex: let $L = \{w \in \{a,b\}^* : \text{all } a\text{'s precede all } b\text{'s}\}$

lexicographic order of L is: $\epsilon, a, b, aa, ab, bb, aaa, aab, abb, bbb, aaaa, aaab, aabb, abbb, bbbb$

* The smallest language over any alphabet is ϕ, whose cardinality is 0.

* The largest language over any alphabet Σ is Σ^*.

Suppose that $\Sigma = \phi$, then $\Sigma^* = \{\epsilon\}$ and $|\Sigma^*| = 1$

Theorem(1) ~~Cardinality of Σ^*~~ If $\Sigma \neq \phi$ then Σ^* is countably infinite (A set is countably infinite if its elements can be put in one-to-one correspondance with the set of natural nos)

Proof: The elements of Σ^* can be lexicographically enumerated by a straightforward procedure that

> Enumerates all strings of length 0, then length 1, then length 2 and so forth.

> within the strings of a given length, enumerates them in dictionary order

This enumeration is infinite since there is no longest string in Σ^*. Since there exists an infinite enumeration of Σ^*, it is countably infinite.

Since any language over Σ is a subset of Σ^*, the cardinality of every language(s) atleast 0 and atmost $\aleph_0$.
So, all languages are either finite or countably infinite.

How many Languages Are These?

Let Σ be an alphabet. The set of languages defined on Σ is $\mathcal{P}(\Sigma^*)$, the power set of Σ^* or the set of all subsets of Σ^*.

If $\Sigma = \phi$ then Σ^* is $\{\epsilon\}$ and $\mathcal{P}(\Sigma^*)$ is $\{\phi, \{\epsilon\}\}$

Theorem: If $\Sigma \neq \phi$ then set of languages over Σ is uncountably infinite. (A set is uncountable if its cardinality is larger than that of set of all natural no.)

Proof: The set of languages defined on Σ is $\mathcal{P}(\Sigma^*)$.
By theorem 1, Σ^* is countably infinite. If S is a countably infinite set, $\mathcal{P}(S)$ is uncountably infinite. So, $\mathcal{P}(\Sigma^*)$ is uncountably infinite.

3) Functions on languages

Since languages are sets, all of the standard set operations are well-defined on languages. Following are two set functions applied to languages:

(a) Let $\Sigma = \{a, b\}$

L_1 = { Strings with even no of a's } = { ϵ, b, bb, aab, aabb, aa, aaaa, ... }

L_2 = { Strings with no b's } = { ϵ, a, aa, aaa, ... }

$L_1 \cup L_2$ = { all Strings of just a's + Strings that Contains b's and an even no of a's }. $L_1 \cup L_2$ = { ϵ, aa, b, aa, bb, aab, aaa, ... }

$L_1 \cap L_2$ = { ϵ, aa, aaaa, aaaaaa, ... }

$L_2 - L_1$ = { a, aaa, aaaaa, ... }

$\neg(L_2 - L_1)$ = { Strings with atleast one b } $\cup$ { Strings with an Even no. of a's }

$\Rightarrow \overline{L_2 \cup L_1}$ =

Because languages are sets of Strings, it makes Sense to define operations on them in terms of the operations that we have already defined on Strings. Three useful ones are: Concatenation, Kleene Star, and reverse

Let L_1 and L_2 be two languages defined over Some alphabet Σ. Then their Concatenation, written as L_1L_2 or $L_1 \cdot L_2$

(i) Concatenation

$L_1L_2 = \{ w \in \Sigma^* : \exists s \in L_1 (\exists t \in L_2 (w = st)) \}$.

Example: Let L_1 = { cat, dog, mouse, bird }.

L_2 = { bone, food }

L_1L_2 = { Catbone, Catfood, dogbone, dogfood, mousebone, mousefood, birdbone, birdfood }

The language { ϵ } is the identity for Concatenation of languages. So for all languages L, $L\{\epsilon\} = \{\epsilon\}L = L$.

The language ϕ is a zero for concatenation of languages. So, for all languages L, $L\phi = \phi L = \phi$ (No ways to select a string from the Empty set).

* Concatenation defined on languages is associative. So for all languages L_1, L_2 of L_3: $((L_1L_2)L_3) = L_1(L_2L_3)$.

Note: Be careful when concatenating languages that are defined using replication.

~~Ex: Suppose $L_1 = \{a^n : n \geq 0\}$ means the set of strings composed of zero or more a's~~

Ex: Suppose that $L_1 = \{a^n : n \geq 0\}$ and $L_2 = \{b^n : n \geq 0\}$.

$L_1 = \{\epsilon, a, aa, aaa, aaaa, \ldots\}$ $L_2 = \{\epsilon, b, bb, bbb, bbbb, \ldots\}$

By the defn of language concatenation

$L_1L_2 = \{w : w$ consists of a (possibly empty) a region followed by a (possibly Empty) b region$\}$. $= \{\epsilon, b, bb, bbb, bbbb, \ldots, a, ab, abb, abbb, \ldots\}$

$L_1L_2 \neq \{a^nb^n : n \geq 0\}$, Since Every string in $\{a^nb^n : n \geq 0\}$ must have the same no of b's as a's. The easiest way to avoid confusion is simply to rename conflicting variables before attempting to concatenate the expressions that contain them. So $L_1L_2 = \{a^nb^m : n, m \geq 0\}$.

(ii) Kleene Star: Let L be a language defined over some alphabet Σ. Then the Kleene star of L, written as L^* and is defined as

$L^* = \{\epsilon\} \cup \{w \in \Sigma^* : \exists k \geq 1 \ (\exists w_1, w_2, \ldots, w_k \in L \ (w = w_1 w_2 \cdots w_k))\}$

ie L^* is the set of strings that can be formed by concatening together zero or more strings from L.

Ex 1) Let $L = \{dog, cat, fish\}$

$L^* = \{\epsilon, dog, cat, fish, dogdog, dogcat, \ldots,$
~~fishfish~~, fishdog, $\ldots$, fishcatfish, ~~fishdog~~
fishdogfishcat, $\ldots\}$.

Ex 2) Let $L = \{w \in \{a,b\}^* : \#_a(w)$ is odd and $\#_b(w)$ is Even$\}$.

Then $L^* = \{w \in \{a,b\}^* : \#_b(w)$ is Even$\}$.

The constraint on the no of a's disappears in the descri of L^* because strings in L^* are formed by concatenating together any no of strings from L. If an odd no of string are concatenated together, the result will contain an odd no of a's. If an Even no of strings are concatenated, the result will contain an Even no of a's.

L^* always contains an infinite no of strings as long as L is not Equal to Either $\emptyset$ or $\{\epsilon\}$. If $L = \emptyset$, then $L^* = \{\epsilon\}$

If $L = \{\epsilon\}$, then L^* is also $\{\epsilon\}$

It is sometimes useful to require that atleast one Element of L be selected. So we define:

$L^+ = LL^*$

L^+ — closure of L under Concatenation.

Note: $L^+ = L^* - \{\epsilon\}$ iff $\epsilon \notin L$.

Ex(3): Let $L = \{0,1\}^+$ be the set of binary strings. L does not include ϵ.

3) Let L be a language defined over some alphabet Σ. Then the reverse of L, written as L^R and is defined as

$L^R = \{w \in \Sigma^* : w = x^R \text{ for some } x \in L\}$.

$\Sigma = \{a, b\}$, $L = \{ab, abb, baa, aab\}$, $L^R = \{ba, bba, aab\}$

ie L^R is the set of strings that can be formed by taking some string in L and reversing it.

Theorem: If L_1 and L_2 are languages, then $(L_1 L_2)^R = L_2^R L_1^R$

Proof: If x and y are strings, then $\forall x (\forall y ((xy)^R = y^R x^R))$ — Theorem (1)

$(L_1 L_2)^R = \{(xy)^R : x \in L_1 \text{ and } y \in L_2\}$ — Defn of Concatenation of languages

$= \{y^R x^R : x \in L_1 \text{ and } y \in L_2\}$ — ~~Line 1 & 2~~ Theorem (1)

$= L_2^R L_1^R$ — Defn of Concatenation of Languages

Assigning Meaning to the Strings of a language:

A Language Hierarchy

Language Recognition: Assume that we are given

- The definition of language L
- A string w

Then we must answer the question: "Is w in L?"
A decision problem is simply a problem that requires a yes or no answer."

The Power of Encoding

Automata Theory answers the question "Is w in L?". It also answers the problems like multiplying nos, sorting lists and retrieving values from a database? and the real world problems like air traffic control and inventory management by Encoding with an appropriate Encoding, other kinds of problems can be recast as the problem of deciding whether a string is in a language.
Examples divided into two Categories:

1) Problems that are already stated as decision problems. For these, all we need to do is to encode the inputs as strings and then define a language that contains Exactly the set of inputs for which the desired answer is yes.

2) Problems that are not already stated as decision problems. These problems may require results of any type. For these, we must first reformulate the problem as a decision problem and then encode it as a language recognition task.

1) Everything is a String

- We will use the notation $\langle X \rangle$ to mean a String Encoding of some object X.
- $\langle X, Y \rangle$ to mean the Encoding into a single String of the two objects X and Y.

Ex(1): Pattern matching on the Web

- Problem: Given a search string w and a web document d, do they match? ie Should a Search Engine, on input w, consider returning d?
- The language to be decided: $\{\langle w, d \rangle : d$ is a candidate match for the query $w\}$.

Ex(2): Does a program Always Halt?

- Problem: Given a program P, written in some Standard programming language, is P guaranteed to halt on all inputs?
- The language to be decided: $HP_{ALL} = \{P : P$ halts on all inputs$\}$

Casting Problems as Decision Questions

Ex(1): Casting Addition as Decision

- Problem: Given two nonnegative integers, Compute their Sum.
- Encoding of the Problem: Transform the problem of adding two nos into the problem of checking to see whether a third no is the Sum of the first two.
- Language to be decided:

  INTEGERSUM: {w of the form: <integer1> + <integer2> = <integer3>, where each of the substrings <integer1>, <integer2>, and <integer3> is an Element of {0, 1, 2, ... 9} and <integer3> is the Sum of integer1 and integer2}.

Ex(2): Casting Sorting as Decision

- Problem: Given a list of integers, Sort it.
- Encoding of the Problem: Transform the problem of Sorting a list into the problem of Examining a pair of lists and deciding whether the Second corresponds to the sorted version of the first.
- The language to be decided:

  $L = \{w_1 \# w_2 : \exists n \geq 1$ {w_1 is of the form $int_1, int_2, \ldots, int_n$ and w_2 is of the form $int_1, int_2, \ldots, int_n$ and w_2 Contains the Same objects as w_1 and w_2 is sorted}.

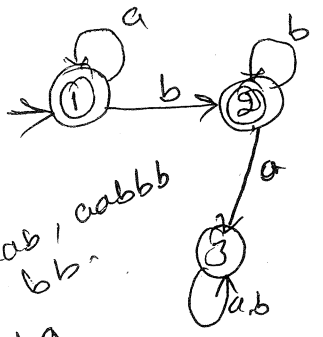
- Example of a string in L: 1, 5, 3, 9, 6 #1, 3, 5, 6, 9
- Example of a string not in L: 1, 5, 3, 9, 6 #1, 2, 3, 4, 5, 6, 7

A Machine-Based Hierarchy of Language Classes

A brief introduction to a hierarchy of computational models and the language hierarchy that goes along with it.

1. Regular Languages

Consider the first computational model: finite state machine (FSM).



A simple FSM that accepts strings of a's and b's, where all a's come before all b's.

accepting strings: aab, aabbb, bb, ba
rejecting:

The i/p to FSM is a string, which is fed to it one character at a time, left to right. The FSM has a start state, shown with an unlabelled arrow leading to it and some number (zero or more) of accepting states, which are shown with double circles. The FSM starts in its start state. As each character is read, the FSM changes state based on the transitions shown in the figure. If an FSM is in accepting state after reading the last character of some input string s, then M accepts s. Otherwise it rejects it.

Ex: FSM stays in state 1 as long as it is reading a's. When it sees a b, it moves to state 2, where it stays as long as it continues seeing b's. Both states 1 & 2 are accepting states. If in state 2, it sees a, it goes to state 3 - a nonaccepting state.

The Class of languages that can be accepted by Some FSM are called regular.

The Context-Free languages

There are useful languages that are not regular.

Ex(1): (()), and ()(). ()))(, ~~Not Correct~~

Ex(2): $A^nB^n = \{a^nb^n : n \geq 0\}$ – All a's Come first and the no of a's Equals the no of b's. We could try to build an FSM to accept A^nB^n.

Problem: How Can we Count the a's So that we Can Compare them to b's. It is not possible to build an FSM to accept A^nB^n.

But languages like Bal and A^nB^n are important.

FSM is able to Solve this problem?

Ans: Yes. Suppose that we add one thing Stack.

We will Call any machine that Consists of an FSM, plus a Single Stack, a pushdown automata or PDA.

We can Easily build a PDA M to accept A^nB^n.

The idea is that, Each time it Sees an a, M will push it onto the Stack. Then, Each time it Sees a b, it will pop an a from the Stack. If it runs out of input and Stack at the Same time and it is in an accepting

accepting state, it will accept. Otherwise, it will reject.

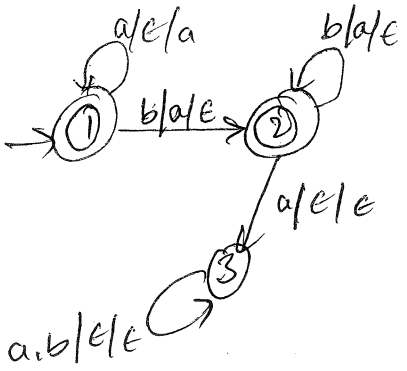


Fig: A simple PDA that accepts A^nB^n

~~A simple PDA that accepts A^nB^n.~~

An arc label of the form x/y/z means, "If the input is an x, and it is possible to pop y off the stack, then take the transition, do the pop of y and push z. If the middle argument is ε, then don't bother to check the stack. If the 3rd argument is ε, then don't push anything.

Now, we can build a PDA to accept Ral and other languages. Class of languages accepted by PDA are called Context-free languages

3) The Decidable and Semidecidable Languages

But these are useful languages that are not context-free.

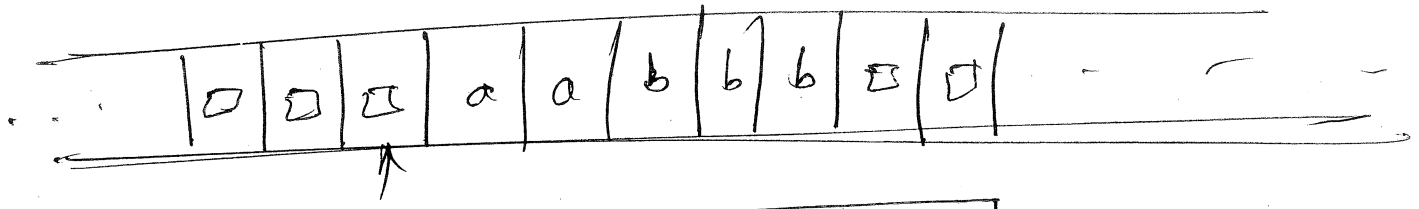
Ex: Let $A^nB^nC^n = \{a^nb^nc^n : n \geq 0\}$.

We could try to build a PDA to accept $A^nB^nC^n$.

We could use the stack to count the a's, just as we did for A^nB^n. We could pop the stack as the b's come in and compare them to the a's. But what shall we do about the c's? We have lost all information about the a's and b's. If they matched, the stack will be empty. It is not possible to build a PDA to accept $A^nB^nC^n$.

To meet this need, we will introduce a third kind of machine that is stronger than the PDA by replacing stack with an infinite tape. The tape will have a single read/write head. Only the tape square under the read/write head can be accessed (for reading or for writing). The read/write head can be moved one square in either direction on each move. The resulting machine is called a Turing machine (TM).

Also change the way that input is given to the machine. Instead of streaming it, one character at a time, the way did for FSMs and PDAs, we will simply write the input string onto the tape and then start the machine with the read/write head just to the left of the first input character.



Finite State Controller
$s, q_1, q_2, \ldots h_1, h_2$

I/p: Current state
Symbol under read/write head

O/p: Next state, ~~Char~~ Symbol to be written on P/W head, move P/W head L or R

Fig: Structure of a Turing machine

At each step, a TM considers its current state and the character that is on the tape directly under its

read/write head. Based on these two things, it chooses its next state, chooses a character to write on the tape under the read/write head and chooses whether to move the read/write head one square to the right or one square to the left. A finite segment of M's tape contains the input string. The rest is blank, but M may move the read/write head off the input string and write on the blank squares of the tape.

There exists a simple TM that accepts $a^n b^n c^n$. It marks off the leftmost a, scans to the right to find b, marks it off, continues scanning to the right, finds a c, and marks it off. Then it goes back to the left, marks off the next a, and so forth. When it runs out of a's, it makes one final pass to the right to make sure that there are no extra b's or c's. If that check succeeds, the machine accepts. If it fails, or if at any point the machine failed to find a required b or c, it rejects.

We will use the TM to define two new classes of languages:

- A language L is decidable iff there exists a TM M that halts on all inputs, accepts all strings that are in L and rejects all strings that are not in L. I.e. M can always say yes or no, as appropriate

- A language L is semidecidable iff there exists a TM that accepts all strings that are in L and fails to ~~accept~~ reject every string that is not in L. Given a string that is not in L, it may reject or it may loop forever (it may not know when it should give up looking for a soln and say no).

The Computational Hierarchy

We have now defined four language classes.

1. Regular languages, which can be accepted by some finite state machines
2. Context-free languages, which can be accepted by some pushdown automata.
3. Decidable (or simply D) languages, which can be decided by some TM that always halts
4. Semidecidable (or SD) languages, that can be semi-decided by some TM that ~~always~~ halts on all strings in the language.

Each of these classes is a proper subset of the next class, as illustrated in the figure below.

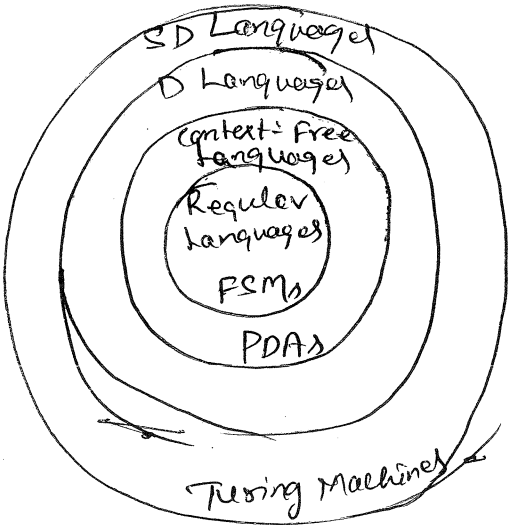


Fig: A hierarchy of language classes.

As we move outward in the language hierarchy, we have access to tools with greater and greater expressive power. Expressiveness generally comes at a price. The price may be:-

1) Computational ~~Power~~ Efficiency:

- FSMs run in time that is linear in the length of the input string.
- A general Context-free parser based on the idea of PDA requires time that grows as the cube of the length of the input string.
- A TM may require time that grows exponentially (or faster) with the length of the input string.

2) Decidability : we have to answer many useful questions about finite state machines. Ex (i) Does an FSM accept some particular string?

(ii) Is an FSM minimal (ie is it the simplest machine that does the job it does?)

(iii) Are two FSMs identical?

A subset of these questions can be answered for PDA. None of them can be answered for TMs.

3) Clarity : These exists tools that enable designers to draw and analyze finite state machines. Every regular language can also be described using the regular expression pattern language

- Every context-free language, in addn to recognizable by some PDA, can be described with a context-free grammar (CFG)

For many important kinds of languages, CFGs are sufficiently natural that they are commonly used as documentation tools.

No corresponding tools exists for the broader class of decidable and semidecidable languages.

Finite State machines

The simplest and most Efficient Computational model is finite State machine (FSM)

Ex: A Vending Machine (VM)

Consider the problem of deciding when to dispense a drink from a VM. Assume that it is possible to buy a drink for $.25. The VM Controller will receive a sequence of inputs, each of which corresponds to one of the following Events:

> A Coin is deposited into the machine. we can use the Symbols N - Nickle, D - Dime and Q - Quarter. to represent these Events.

> The coin return button is pushed. we can use the Symbol R (for return) to represent this Event.

> A drink button is pushed and a drink is dispensed we can use the Symbol S (for Soda) for this Event

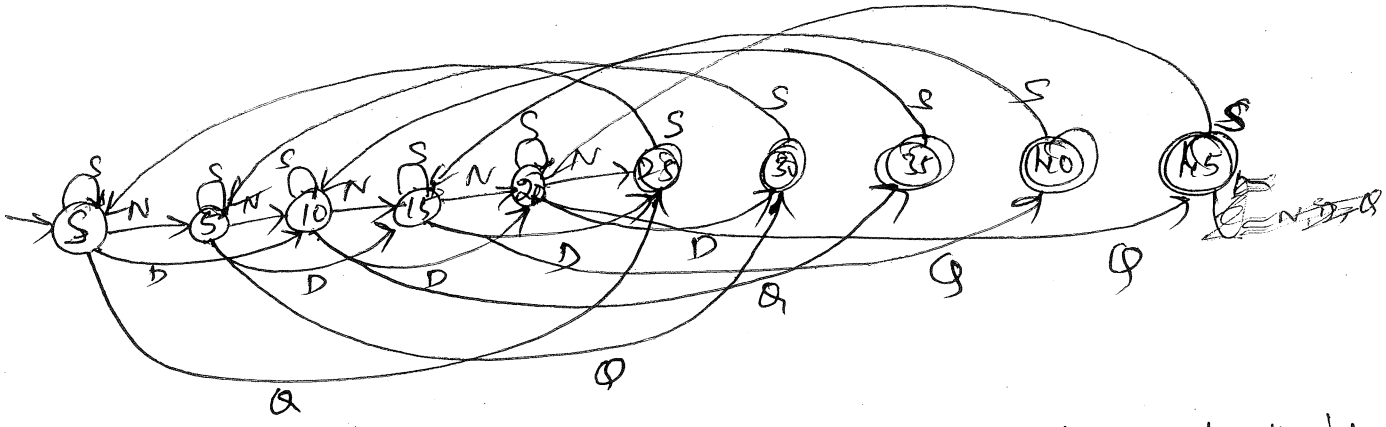
After any finite sequence of inputs, the Controller will be in Either:

> A dispensing State, in which it is willing to dispense a drink if a drink button is pressed.

> A nondispensing State, in which not Enough money had been inserted into the machine.

Assumption: Controller will simply reject any money that comes in after the amount required to buy a drink has been recorded and before a drink has actually been dispensed. Build a MC that will accept upto $.45. If more than the necessary $.25 is inserted before a dispensing button is pushed, our machine will remember the difference and leave a "credit" in the machine.

The Controller that is being described needs 10 states, corresponding to the possible values of the credit that the Customer has in the machine: 0, 5, 10, 15, 20, 25, 30, 35, 40, 45 cents. The main structure of the Controller is



S - Start state. As coins are deposited, the controller's state changes to reflect the amount of money that has been deposited. When the drink button is pushed (indicated as S in the diagram) and the Customer has a credit of less than $.25, nothing happens. Machine state does not change.

If the drink button is pushed and the customer has a credit of $.25 or more, the credit is decremented by $.25

and a drink is dispensed. The drink-dispensing states namely those that correspond to "Enough money" can be thought of as goal or accepting states - shown with double circles.

It is difficult to show all transitions. we must add the following ones:

- From Each ~~State~~ of the accepting states, a transition back to itself labeled with each coin value. These transitions correspond to our decision to reject additional coins once the machine has been fed the price of the drink.

N, n, D

- From each state, a transition back to the start state labeled R. These transitions will be taken whenever the customer pushes the coin return button (R). They correspond to the machine returning all of the money that it had accumulated since the last drink was dispensed.

Deterministic Finite State machines

Def: A FSM is a computational device whose input is a string and whose output is one of two values: ~~Reject~~ Accept and Reject. FSMs are sometimes called finite ~~state~~ automata or FSAs.

- If M is a FSM, then input string is fed to M one character at a time, left to right. Each time it receives a character, M considers its current state and the new character and chooses a next state.

- one or more states of M may be marked as accepting states. If M runs out of input and is in an accepting state, it accepts. If M runs out of input and is not in an accepting state, it rejects.

- The no of steps that M executes on input w is exactly equal to |w|, so M always halts and either accepts or rejects.

Classes of FSMs: ~~1) Deterministic FSM~~

1) Deterministic FSM: These is always exactly one move that can be made at each step and is determined by the current state and the next input character

» Nondeterministic FSMs: These may, at various points in the Computation, be more than one move from which the machine may choose.

1) Deterministic FSM (DFSM): DFSM is a quintuple $(K, \Sigma, \delta, s, A)$ where

K - a finite of States

Σ - input alphabet

$s \in K$ - Start state

$A \subseteq K$ - Set of accepting States, and

δ - transition function. It maps from

$K \times \Sigma$ to K
State input alphabet State

> A Configuration of a DFSM M is an Element of $K \times \Sigma^*$. It captures two things that can make a difference of M's future behavior.

- Its Current State
- The input that is still left to read

> The initial Configuration of a DFSM M on input w, is (s_M, w), where s_M - Start State of M

> The transition function δ defines the operation of a DPSM M one step at a time. We can use it to define the sequence of configurations that M will enter. We start by defining the relation, yields-in-one-step, written $\vdash_M$. ~~related configurations~~

Yields-in-one-step related configuration$_1$ to configuration$_2$ iff M can move from configuration$_1$ to configuration$_2$ in one step.

Let c be any element of Σ and let w be any element of Σ^*. Then,

$(q_1, cw) \vdash_M (q_2, w)$ iff $(q_1, c, q_2) \in \delta$

> ~~The relation yields, written as~~

> The configuration C_1 yields configuration C_2 if M can go from C_1 to C_2 in zero or more steps and is written as $C_1 \vdash_M^* C_2$.

> A computation by M is a finite sequence of configurations $C_0, C_1, \ldots, C_n$ for some $n \geq 0$ such that

- C_0 — initial configuration
- C_n is of the form (q, ϵ), for some state $q \in K_M$ (i.e. the entire input string has been read) and
- $C_0 \vdash_M C_1 \vdash_M C_2 \vdash_M \cdots \vdash_M C_n$.

Let w be an element of Σ^*. Then we will

Say that:

- M accepts w iff $(s,w) \vdash_M^* (q,\epsilon)$ for some $q \in A_M$.

  Any configuration (q,ϵ) for some $q \in A_M$ is called an accepting configuration of M

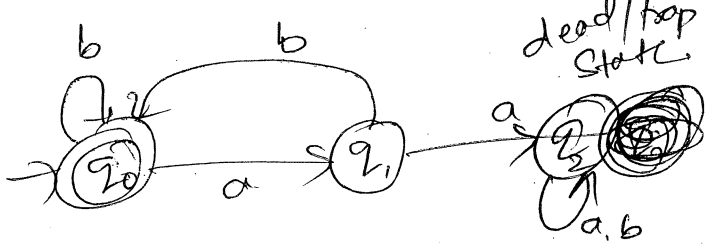
- M rejects w iff $(s,w) \vdash_M^* (q,w)$ for some $q \notin A_M$.

  Any configuration (q,ϵ) for some $q \notin A_M$ is called an rejecting configuration of M.

M halts whenever it encounters either an accepting or a rejecting configuration. It will do so immediately after reading the last character of its input.

The language accepted by M, denoted by $L(M)$ is the set of all strings accepted by M.

Ex(1): Let $L = \{w \in \{a,b\}^* : \text{every } a \text{ is immediately followed by a } b\}$.

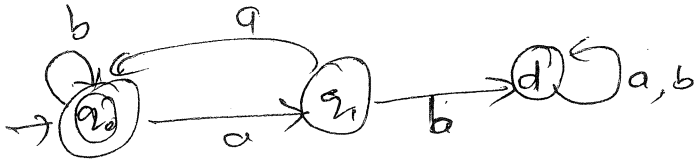


$\delta = \{((q_0, a) = q_1)$
$((q_0, b) = q_2)$
$((q_1, a) = q_2)$
$((q_1, b) = q_2)$
$((q_2, a) = q_2)$
$((q_2, b) = q_2)\}$

$M = (\{q_0, q_1, q_2\}, \{a,b\}, \delta, q_0, \{q_2\})$.

Ex(2): Even length Regions of a's

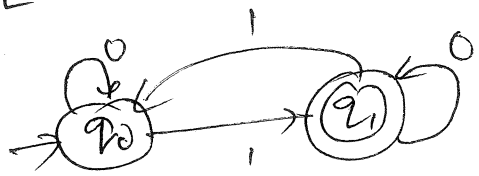
$L = \{w \in \{a,b\}^* :$ Every region of a in w is of even length$\}$.



Ex(3): Checking for odd parity.

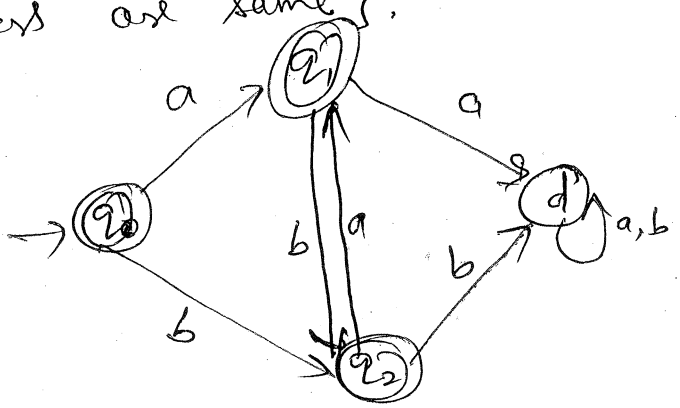
let $L = \{w \in \{0,1\}^* :$ w has odd parity. A binary string has odd parity iff the no of 1's in it is odd.

S: L can be accepted by DFSM M.



Ex(4): let $L = \{w \in \{a,b\}^* :$ w contains no more than one b$\}$. L is regular because it can be accepted by DFSM M.

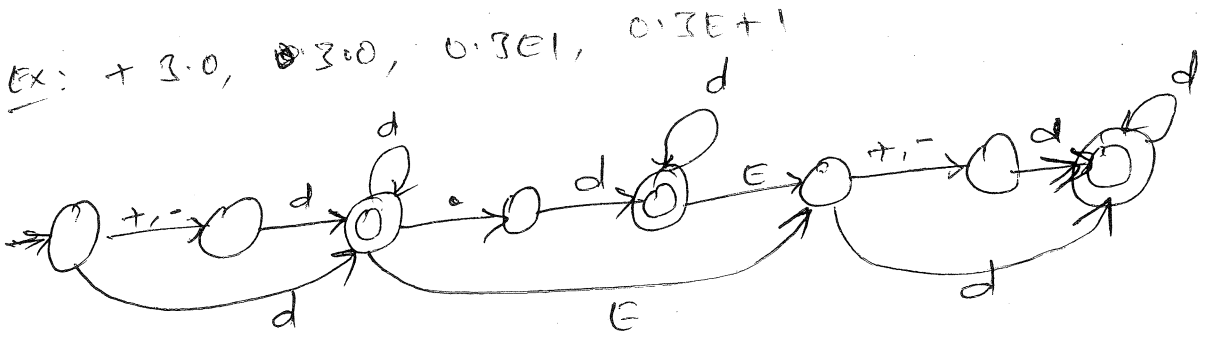
Ex(5): Let $L = \{w \in \{a,b\}^* :$ no two consecutive characters are same$\}$.



Ex 6 : Floating point numbers.

let FLOAT = {w : w is the string representation of a floating

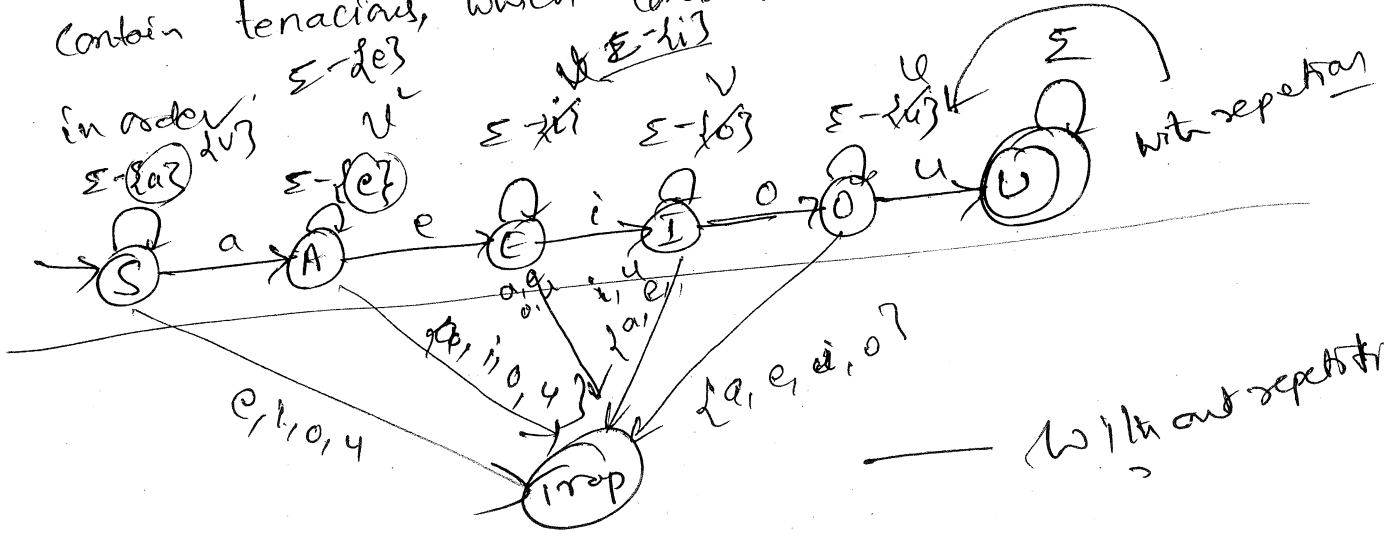
A floating point no



d - Stand for any of the decimal digits (0-9)

Ex 7 : All the vowels in alphabetical order

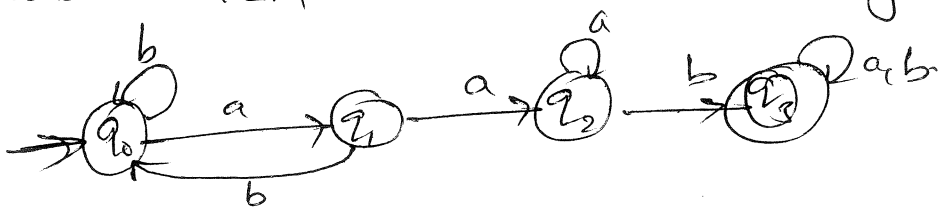
Let L = {w ∈ {a-z}* : all five vowels a, e, i, o, u occur in w in alphabetical order}. So L contains words like abstemious, facetious, and sacrilegious, but does not contain tenacious, which contains all the vowels but not in order.



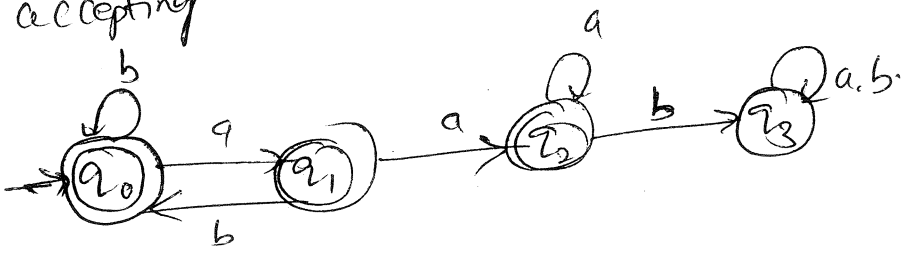
Ex① A Substring that doesnot occur

Let $L = \{w \in \{a,b\}^* : w$ does not contain the substring $aab\}$

S1: Build the FSM that accepts substring aab:



S2: Build a machine to accept L by building the machine to accept $\sim L$ by Converting above M/C by making ~~above M/C into~~ States q_0, q_1, q_2 as accepting and state q_3 as non-accepting



Dr. Krishna A N BE, ME, Ph.D.
Professor, Dept. of CSE,
SJB Institute of Technology,
BGS Health and Education City,
Kengeri, Bangalore - 560 060.
Mob: 9481189899

Nondeterministic FSMs (NDFSMs)

A nondeterministic FSM (NDFSM) M is a quintuple $(K, \Sigma, \Delta, s, A)$ where

K - finite set of States

Σ - Alphabet

$s \in K$ - Start State

$A \subseteq K$ - Set of final States

Δ - transition relation. It is a finite subset of

$$(K \times (\Sigma \cup \{\epsilon\})) \times K$$

i.e each Element of Δ contains a (State, input symbol or ϵ) pair and a new State.

Let w be an element of Σ^*. Then

* M accepts w iff atleast one of its Computations accepts.
* M rejects w iff none of its Computations accepts.
* The language accepted by M, denoted L(M) is the set of all strings accepted by M.

Two key differences between DFSM and NDFSM: In ~~Every DFSM~~ Configuration, a DFSM can make Exactly one move. that is not true for an NDFSM. Instead

1) An NDFSM M may Enter a Configuration in which there are still input Symbols left to read but from which no moves are available. i.e (State, input) pairs for which no next state is defined.

Any sequence of moves that leads to such a configuration cannot ever reach an accepting configuration, M will simply halt without accepting.

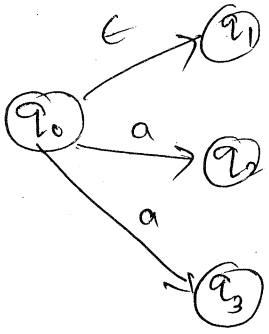
2) An NDFSM M may enter a configuration from which two or more competing moves are possible. This competition is because of either or both of the following properties:

(i) An NDFSM may have one or more transitions that are labelled ϵ, rather than being labeled with a character from Σ.

(ii) Out of some state q, there may be more than one transition with a given label.

These competing transitions give M another way to guess at a correct path.

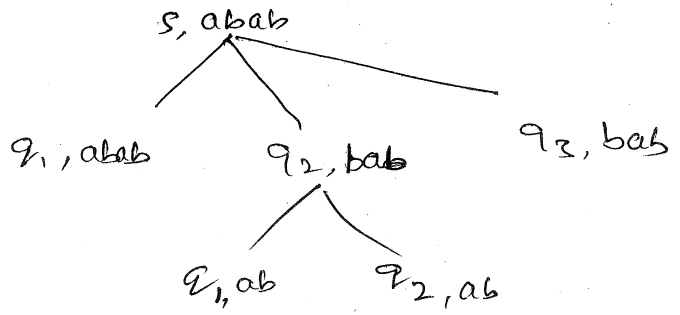
Ex:



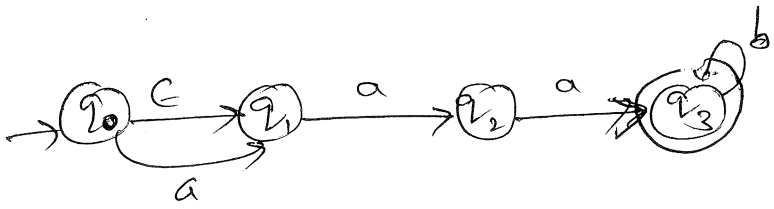
If M is in state q_0 and the next i/p char is 'a'. Then there are 3 moves that M could make.

An NDFSM with two kinds of nondeterminism

One way to depict the operation of M is as a tree

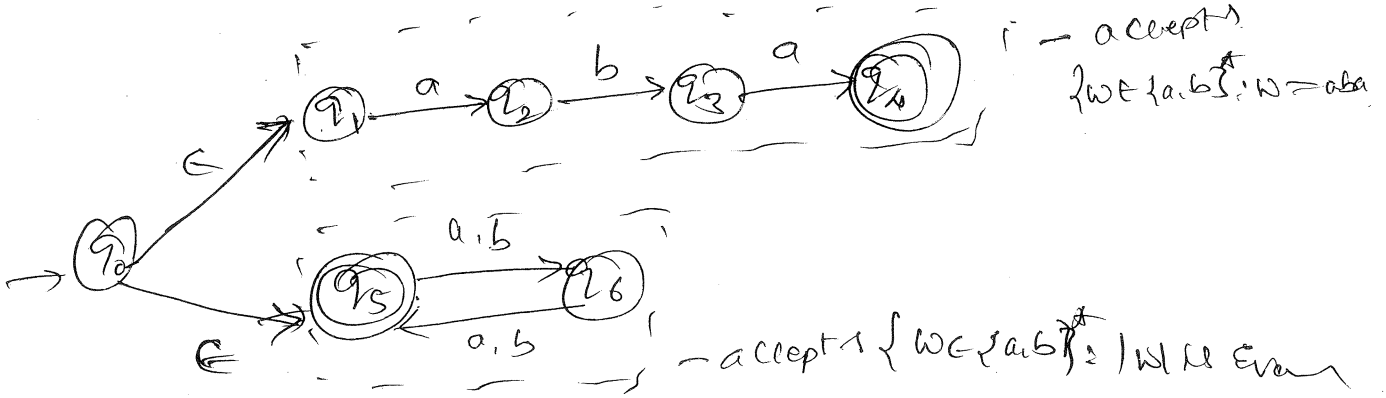


Ex(1) Let $L = \{w \in \{a,b\}^* : w$ is made up of an optional a followed by aa followed by zero or more b's$\}$.



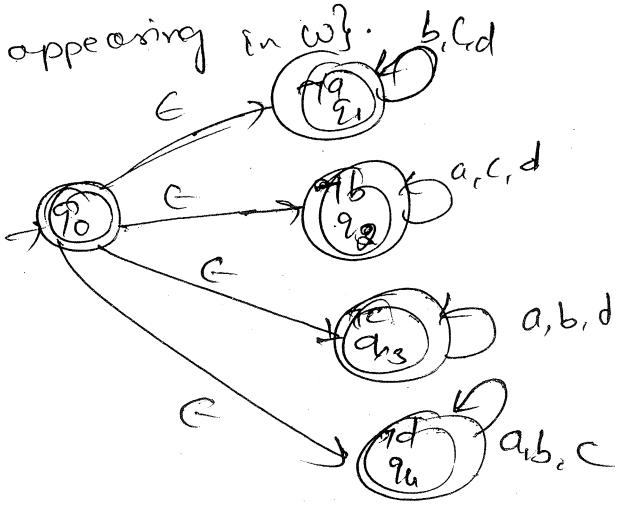
Ex(2) Let $L = \{w \in \{a,b\}^* : w = aba$ or $|w|$ is Even$\}$.

An easy way to build an FSM to accept this language is to build FSMs for each of the individual sublanguages and then "glue" them together with ε-transitions.



Ex(3): missing letter language

Let $\Sigma = \{a, b, c, d\}$. $L_{missing} = \{w :$ there is a symbol $a_i \in \Sigma$ not appearing in $w\}$.



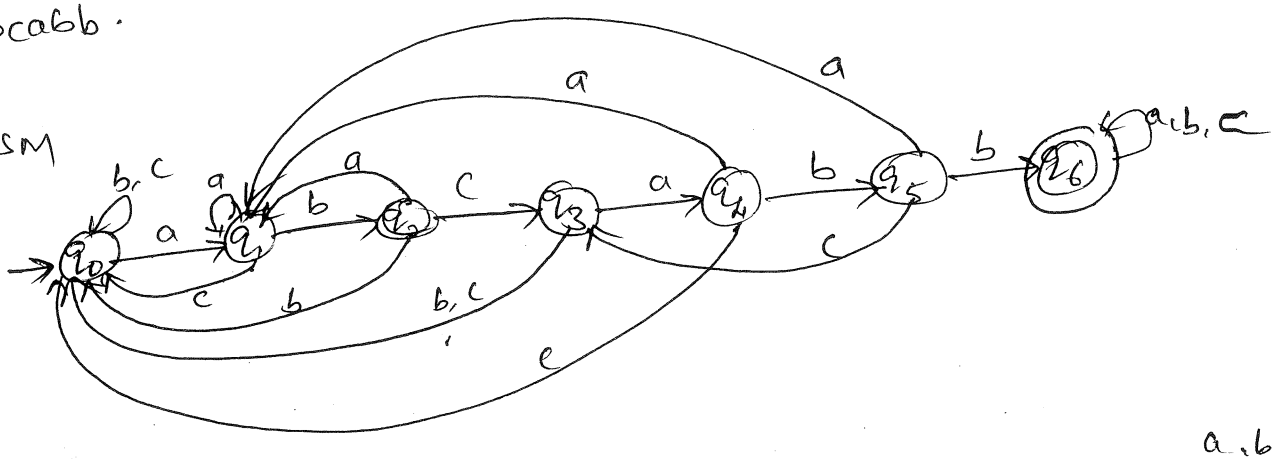
NDFSMs for pattern or Substring Matching

NDFSMs are Effective to define simple machines to search a text string for one or more patterns or substrings.

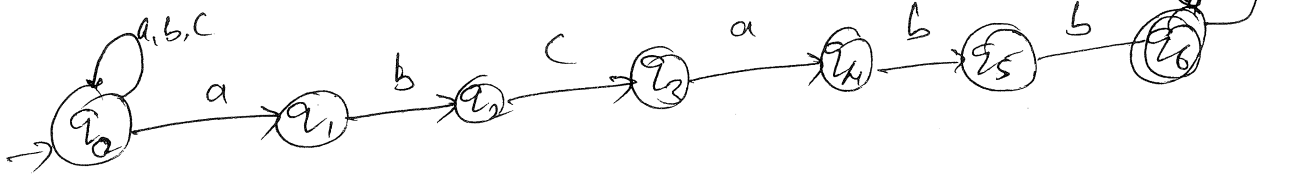
Ex(1): Let $L = \{w \in \{a,b,c\}^* : \exists x,y \in \{a,b,c\}^* (w = x\,abcabb\,y)\}$.

ie w must contain atleast one occurrence of the substring abcabb.

DFSM



NDFSM

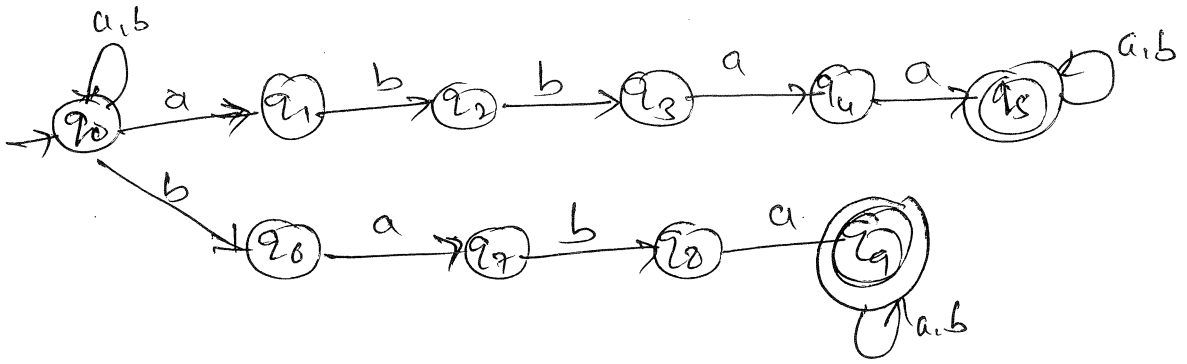


* Easy to build NDFSM Compared to DFSM.
* String Searching is a fundamental op^n in every word processing

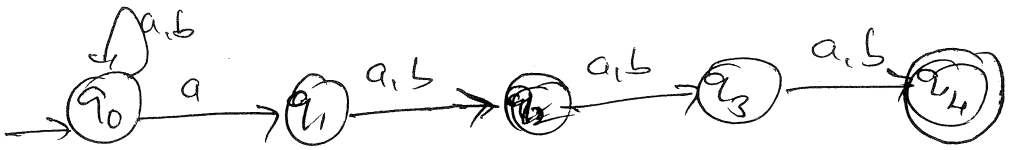
Ex(2): Multiple keywords

Let $L = \{w \in \{a,b\}^* : \exists x,y \in \{a,b\}^* ((w = x\,abbaa\,y) \vee (w = x\,baba\,y))\}$.

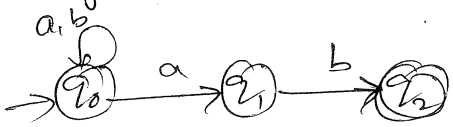
ie w Contains atleast one occurrence of the Substring abbaa or the Substring baba



Ex(3): $L = \{w \in \{a,b\}^* :$ the fourth character from the last is $a\}$.



Ex(4): Construct an NFA to accept strings of a's & b's ending with ab.

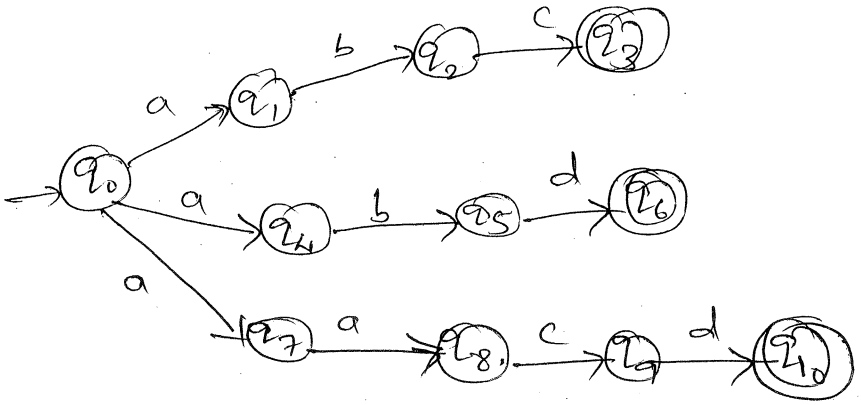


Jan 06, (5) Design an NFA for the following languages

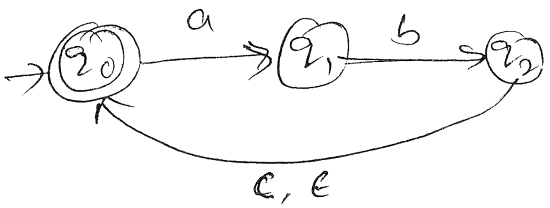
(i) abc, abd, aacd, where $\Sigma = \{a, b, c, d\}$

(ii) $\{ab, abc\}^*$, where $\Sigma = \{a, b, c\}$

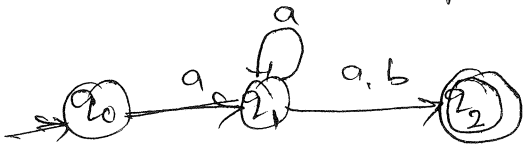
Soln (i)



(ii)

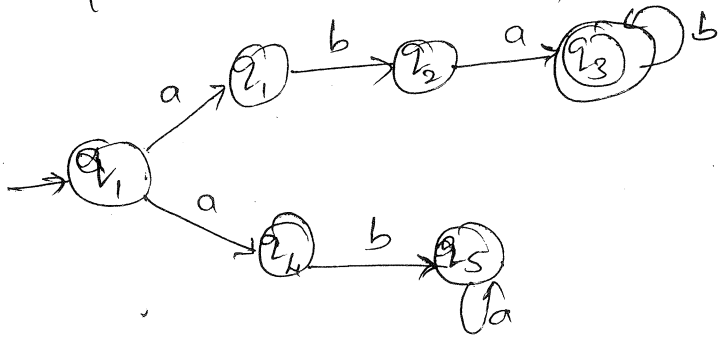


Jan 10 4PM (6) Design an NFA to accept the language $\{aa^*(a+b)\}$

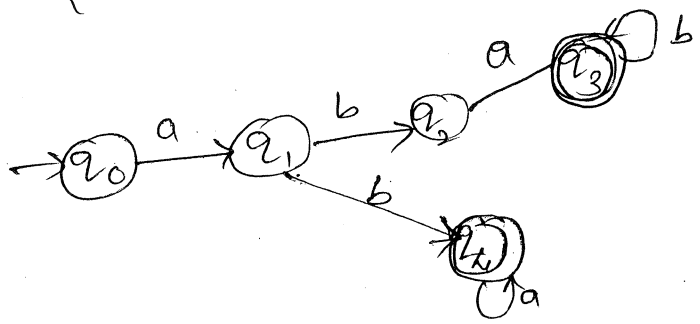


(7) obtain an NFA to accept the following language

$L = \{w \mid w \in abab^n \text{ or } ab^n, n \geq 0\}$



(8) Design an NFA with no more than 5 States for the set $\{abab^n : n \geq 0\} \cup \{ab^n : n \geq 0\}$



Analyzing Nondeterministic FSMs

Handling ϵ-Transitions

eps(q), where q is some state in M, to be the set of states of M that are reachable from q by following zero or more ϵ-transitions.

Formally $eps(q) = \{p \in K : (q, w) \vdash_M^* (p, w)\}$

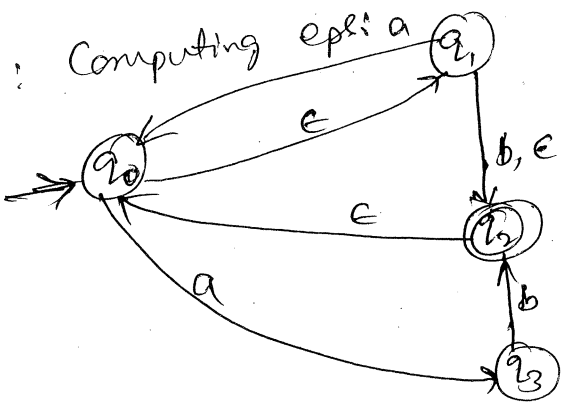
Alternatively, eps(q) is the closure of {q} under the relation {(p, r): there is a transition $(p, \epsilon, r) \in \Delta$}.

Algorithm: To Compute eps:

eps(q: state) =

1. result = {q}
2. While there exists some $p \in$ result and some $r \notin$ result and some transition $(p, \epsilon, r) \in \Delta$ do: Insert r into result.
3. Return result.

Ex 6.1: Computing eps: a



To compute eps(q0), initially result = {q0}. Then q1 is added, producing {q0, q1}. Then q2 is added, producing {q0, q1, q2}. There is an ϵ-transition from q2 to q0, but q0 is already in result. So computation of eps(q0) halts.

So $eps(q_0) = \{q_0, q_1, q_2\}$

$eps(q_1) = \{q_1, q_2\}$

$eps(q_2) = \{q_0, q_1, q_2\}$

$eps(q_3) = \{q_3\}$

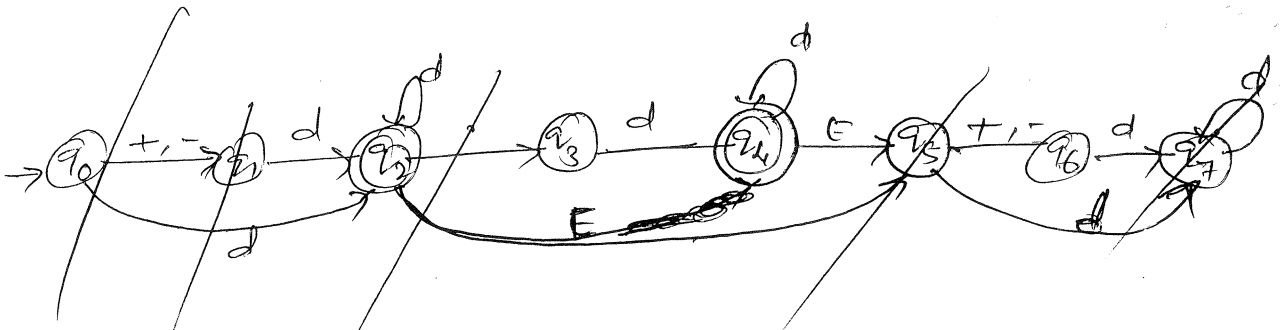
A Simulation Algorithm

With eps function, we can now define an algorithm for tracing all paths in parallel through an NDFSM M:

ndf Simulate (M: NDFSM, w: String) =

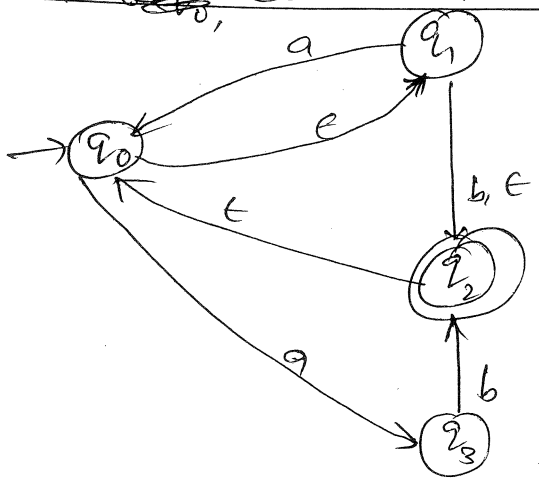
1. current-state = eps(s)
2. While any input symbols in w remain to be read do:
   - 2.1 c = get-next-symbol(w)
   - 2.2 next-state = $\emptyset$
   - 2.3 For each state q in current state do:
     - For Each state p such that $(q, c, p) \in \Delta$ do.
       - next-state = next-state $\cup$ eps(p).
   - 2.4 Current-State = next-state
3. If current-state contains any state in A, accept else reject.

Ex:



Compute $\delta(q_0, 5.6)$ for the above NFA

Soln: 1. ~~$\delta(q_0,$~~ Current State = $eps(s)$ =



$eps(q_0) = \{q_0, q_1, q_2\}$

$eps(q_1) = \{q_0, q_1, q_2\}$

$eps(q_2) = \{q_0, q_1, q_2\}$

$eps(q_3) = \{q_3\}$

Compute $\hat{\delta}(q_0, bab)$

Soln:

Step1: Current-State = $eps(q_0) = \{q_0, q_1, q_2\}$

Step2: Compute δ', Δ

active-states = $\{\{q_0, q_1, q_2\}\}$.

Consider $\{q_0, q_1, q_2\}$

$\Delta(q_0, b) = \Delta(\delta'(q_0, \epsilon), b)$

$= \Delta(\{q_0, q_1, q_2\}, b)$

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

$\Delta(q_0, b) = \{q_2\}$

$\delta'(q_0, b) = eps(q_2) = \{q_0, q_1, q_2\}$

~~$\Delta(q_0, b) = eps(\Delta(q_0, \epsilon), b)$~~

~~$= eps(\{q_0, q_1, q_2\}, b)$~~

Step 3 $\delta(q_0, ba) = \delta(\hat{\delta}(q_0, b), a) = \delta(\{q_0\}, a) = \phi$

$= \delta(\{q_0, q_1, q_2\}, a) = \delta(q_0, a) \cup \delta(q_1, a) \cup \delta(q_2, a)$

$\delta(q_0, ba) = \{q_0, q_3\}$

$\hat{\delta}(q_0, ba) = eps(q_0) \cup eps(q_3)$

$= \{q_0, q_1, q_2\} \cup \{q_3\}$

$\hat{\delta}'(q_0, ba) = \{q_0, q_1, q_2, q_3\}$

Step 4 $\hat{\delta}(q_0, bab) = \hat{\delta}(\hat{\delta}(q_0, ba), b)$

$= \hat{\delta}(\{q_0, q_1, q_2, q_3\}, b)$

$= \hat{\delta}(q_0, b) \cup \delta(q_1, b) \cup \hat{\delta}(q_2, b) \cup \hat{\delta}(q_3, b)$

$\hat{\delta}(q_0, bab) = \{q_2\}$

$\hat{\delta}'(q_0, bab) = eps(q_2) = \{q_0, q_1, q_2\}$

The Equivalence of Nondeterministic and Deterministic FSMs

Theorem: Given an NDFSM $M = (K, \Sigma, \Delta, s, A)$ that accepts some language L, there exists an equivalent DFSM that accepts L.

Proof: The proof is by construction of an equivalent DFSM M'. The construction is based on the function eps and on the simulation algorithm. The states of M' will correspond to sets of states in M. So

$M' = (K', \Sigma, \delta', s', A')$ where

K' - contains one state for each element of $\mathcal{P}(K)$

s' - $eps(s)$

$A' = \{Q \subseteq K : Q \cap A \neq \emptyset\}$

$\delta'(Q, c) = \cup (eps(p) : \exists q \in Q ((q, c, p) \in \Delta))$

Note

(1) There is one state in K' for each element of $\mathcal{P}(K)$. However, in most cases, many of those states will be unreachable from s' and thus unnecessary. So, construction algorithm creates states only as it needs to.

(2) To decide whether a state in K' is an accepting state, we see whether it corresponds to an element of $\beta(K)$ that contains atleast one element of A i.e one accepting state from K.

(3) M' accepts whenever it runs out of input and is in a state that contains atleast one accepting state of M.

(4) δ' corresponds to step 2.3 of the simulation algorithm

Following algorithm computes M' from M

ndfsm to dfsm(M: NDFSM) =

1. For each state q in K do:
   Compute eps(q) /* These values will be used below.

2. $s' = eps(s)$

3. Compute δ':
   a. active-states = $\{s'\}$
   b. $\delta' = \emptyset$
   c. while there exists some element Q of active-states for which δ' has not yet been computed do:
      - For each character c in Σ do:
        new-state = $\emptyset$
        For each state q in Q do:
          For each state p such that $(q, c, p) \in \Delta$ do:
            new-state = new-state $\cup$ eps(p).

Add the transition (Q, C, new-state) to δ'

If new-state ∉ active-states then insert it into active-states.

4. K' = active-states

5. A' = {Q ∈ K' : Q ∩ A ≠ ∅}.

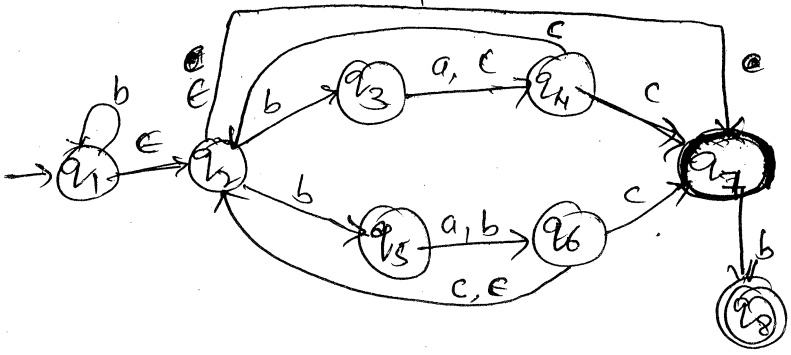
The algorithm ndfsmtodfsm halts on all inputs and constructs a DFSM M' that accepts exactly L(M), the language accepted by M.

The algorithm ndfsmtodfsm is important for two reasons:

1. It proves the theorem that, for every NDFSM there exists an equivalent DFSM

2. It lets us use nondeterminism as a design tool, even though we may ultimately need a deterministic machine. We can build a NDFSM to solve our problem, ndfsmtodfsm can easily construct an equivalent DFSM.

Ex: Use ndfsmtodfsm to build a deterministic FSM



NOT an accepting state.

1. Compute eps(q) for each state q in K_M

$eps(q_1) = \{q_1, q_2, q_7\}$, $eps(q_2) = \{q_2, q_7\}$, $eps(q_3) = \{q_3\}$

$eps(q_4) = \{q_4\}$, $eps(q_5) = \{q_5\}$, $eps(q_6) = \{q_2, q_6, q_7\}$

$eps(q_8) = \{q_8\}$

2. $s' = eps(s) = \{q_1, q_2, q_7\}$

3. Compute δ':

active-states = $\{\{q_1, q_2, q_7\}\}$.

Consider $\{q_1, q_2, q_7\}$:

$$\delta'(\{q_1, q_2, q_7\}, a) = eps(\delta(\{q_1, q_2, q_7\}, a))$$
$$= eps(\phi) = \phi$$

$$\delta'(\{q_1, q_2, q_7\}, b) = eps(\delta(\{q_1, q_2, q_7\}, b)$$
$$= eps(\{q_1, q_3, q_5, q_8\})$$
$$= \{q_1, q_2, q_3, q_5, q_7, q_8\}$$

$$\delta'(\{q_1, q_2, q_7\}, c) = eps(\delta(\{q_1, q_2, q_7\}, c)$$
$$= eps(\{\}) = \phi$$

active-states = $\{\{q_1, q_2, q_7\}, \phi, \{q_1, q_2, q_3, q_5, q_7, q_8\}\}$

Consider ϕ:

$\delta'(\phi, a) = \phi$

$\delta'(\phi, b) = \phi$

$\delta(\phi, c) = \phi$

/* ϕ is a dead state and will generally omit it.

Consider $\{q_1, q_2, q_3, q_5, q_7, q_8\}$

$\delta'(\{q_1, q_2, q_3, q_5, q_7, q_8\}, a) = eps(\delta(\{q_1, q_2, q_3, q_5, q_7, q_8\}, a)$

$= eps(\{q_4, q_6\})$

$= \{q_4, q_6, q_7\}$

$\delta'(\{q_1, q_2, q_3, q_5, q_7, q_8\}, b) = eps(\delta(\{q_1, q_2, q_3, q_5, q_7, q_8\}, b)$

$= eps(\{q_1, q_3, q_5, q_6, q_8\})$

$= \{q_1, q_2, q_3, q_5, q_6, q_7, q_8\}$

$\delta'(\{q_1, q_2, q_3, q_5, q_7, q_8\}, c) = eps(\delta(\{q_1, q_2, q_3, q_5, q_7, q_8\}, c))$

$= eps(\{q_4\}) = \{q_4\}$

active-states = $\{\{q_1, q_2, q_7\}, \emptyset, \{q_1, q_2, q_3, q_5, q_7, q_8\}, \{q_2, q_4, q_6, q_7\}, \{q_1, q_2, q_3, q_5, q_6, q_7, q_8\}, \{q_4\}\}$.

Consider $\{q_2, q_4, q_6, q_7\}$

$\delta'(\{q_2, q_4, q_6, q_7\}, a) = eps(\delta(\{q_2, q_4, q_6, q_7\}, a)$

$= eps\{\} = \phi$

$\delta'(\{q_2, q_4, q_6, q_7\}, b) = eps(\delta(\{q_2, q_4, q_6, q_7\}, b)$

$= eps(\{q_3, q_5, q_8\}) = \{q_3, q_5, q_8\}$ //

$\delta'(\{q_2, q_4, q_6, q_7\}, c) = eps(\delta(\{q_2, q_4, q_6, q_7\}, c))$

$= eps(\{q_2, q_7\}) = \{q_2, q_7\}$

active-states = {{q1, q2, q7}, ϕ, {q1, q2, q3, q5, q7, q8}
{q2, q4, q6, q7}, {q1, q2, q3, q5, q6, q7, q8}, {q4}
{q3, q5, q8}, {q2, q7}}

$\delta'(\{q_1, q_2, q_3, q_5, q_6, q_7, q_8\}, a) = eps(\delta(\{q_1, q_2, q_3, q_5, q_6, q_7, q_8\}, a)$

$= eps(\{q_4, q_6\}) = \{q_4, q_4, q_6, q_7\}$ //

$\delta'(\{q_1, q_2, q_3, q_5, q_6, q_7, q_8\}, b) = eps(\{q_1, q_2, q_3, q_5, q_6, q_7, q_8\}, b)$

$= eps(\{q_1, q_3, q_6, q_8\}) = \{q_1, q_2, q_3, q_5, q_6, q_7, q_8\}$

$\delta'(\{q_1, q_2, q_3, q_5, q_6, q_7, q_8\}, c) = eps(\{q_4, q_7\}) = \{q_2, q_4, q_7\}$

new state

active-states = {

$\delta'(\{q_4\}, a) = eps(\{\}) = \phi$

$\delta'(\{q_4\}, b) = eps(\{\}) = \phi$

$\delta'(\{q_4\}, c) = eps(\{q_2, q_7\}) = \{q_2, q_7\}$ // — not a new state

$\delta'(\{q_3, q_5, q_8\}, a) = eps(\{q_4, q_6\}) = \{q_4, q_4, q_6, q_7\}$; not new

$\delta'(\{q_3, q_5, q_8\}, b) = eps(\{q_8\}) = \{q_2, q_6, q_7\}$ // — new

$\delta'(\{q_3, q_5, q_8\}, c) = eps(\{q_4\}) = \{q_4\}$ // — not new

$\delta'(\{q_2, q_7\}, a) = eps(\{\}) = \phi$

$\delta'(\{q_2, q_7\}, b) = eps(\{q_3, q_5, q_8\}) = \{q_3, q_5, q_8\}$ ← not new

$\delta'(\{q_2, q_7\}, c) = eps(\{\}) = \phi$

Consider $\{q_2, q_4, q_7\}$

$\delta'(\{q_2, q_4, q_7\}, a) = eps(\{\}) = \phi$

$\delta'(\{q_2, q_4, q_7\}, b) = eps(\{q_3, q_5, q_8\}) = \{q_3, q_5, q_8\}$ — not new

$\delta'(\{q_2, q_4, q_7\}, c) = eps(\{q_2, q_7\}) = \{q_2, q_7\}$ — not new

Consider $\{q_2, q_6, q_7\}$

$\delta'(\{q_2, q_6, q_7\}, a) = eps(\{\}) = \phi$

$\delta'(\{q_2, q_6, q_7\}, b) = eps(\{q_3, q_5, q_8\}) = \{q_3, q_5, q_8\}$ — not new

$\delta'(\{q_2, q_6, q_7\}, c) = eps(\{q_2, q_7\}) = \{q_2, q_7\}$ — not new

No New States generated

$Q' = \{\{q_1, q_2, q_7\}, \phi, \{q_1, q_2, q_3, q_5, q_7, q_8\}, \{q_2, q_4, q_6, q_7\},$
$\{q_1, q_2, q_3, q_5, q_6, q_7, q_8\}, \{q_4\}, \{q_3, q_5, q_8\}, \{q_2, q_7\},$
$\{q_2, q_4, q_7\}, \{q_2, q_6, q_7\}\}$

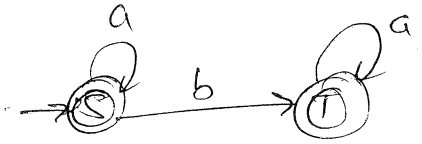
Accepting States: $\{\{q_1, q_2, q_3, q_5, q_7, q_8\}\{q_1, q_2, q_3, q_5, q_6, q_7, q_8\}\}$
$\{q_3, q_5, q_8\}$

From FSMs to Operational Systems

* An FSM is an abstraction. FSM Solves a problem without worrying about many kinds of implementation details.
* FSMs for real problems can be turned into operational systems in any of a no. of ways:
  1. Any FSM can be translated into a circuit design and implemented directly in hardware
     Ex: Implementing the parity Checking FSM in HW.
  2. An FSM can be simulated by a general purpose interpreter. A simulation can be used to check the design before it is translated into hardware
  3. An FSM can be used as a Specification for some critical behavior of a complex System. The Specification can then be implemented in software.

Simulators for FSMs

1. Simulating Deterministic FSMs : one approach is to think FSM as the Specification for a simple, table-driven program and then proceed to write the code:
   Ex: Consider the FSM that accepts the language
   $L = \{w \in \{a,b\}^* : w$ contains no more than one $b\}$



We can view M as a specification for the following program:

Until accept or reject do:

S: s = get-next-symbol
If s = end-of-file then accept
Else if s = a then goto S
Else if s = b then goto T

T: s = get-next-symbol
If s = end-of-file then accept
Else if s = a then goto T
Else if s = b then reject

End

* Given a FSM M with states K, this approach will create a program of length = $2 + (|K| \cdot (|\Sigma| + 2))$.
* The time required to analyze an input string w is $\Theta(|w| \cdot |\Sigma|)$.

~~Problem~~ Drawback: We must generate new code for every FSM that we wish to run.

Instead we can build an interpreter that executes the FSM directly.

A simple interpreter for a deterministic FSM $M = (K, \Sigma, \delta, s, A)$:

dfsmsimulate(M: DFSM, w: string) =

1. st = s.
2. Repeat:
   2.1. c = get-next-symbol(w).
   2.2. If c ≠ end-of-file then:
      2.2.1. st = δ(st c)

   until c = end-of-file
3. If st ∈ A then accept else reject.

The algorithm dfsmsimulate runs in time approximately $O(|w|)$, if we assume that the step 2.2.1 can be implemented in constant time.

2. Simulating Nondeterministic FSMs

Suppose that we want to execute an NDFSM M. One way is:

ndfsmconvertandsimulate(M: NDFSM) =
   dfsmsimulate(ndfsmtodfsm(M))

Converting an NDFSM to a DFSM can be very inefficient in terms of both time and space. If M has K states, it could take time and space equal to $O(2^K)$ just to do the conversion. Simulation after the conversion, would take time equal to $O(|w|)$.

So, we need an algorithm that directly simulates an NDFSM M without converting it to a DFSM.

The idea is to simulate being in sets of states at once. But, instead of generating all of the reachable sets of states right away, an ndfsm to dfsm does, it generates them as the fly, as they are needed, being careful not to get stuck chasing ϵ-loops. Given NDFSM $M = (K, \Sigma, \Delta, s, A)$, running an input string w.

ndfsmsimulate(M: NDFSM, w: string) =

1. Declare the set st. /* st will hold the current state (set of states from K)
2. Declare the set st1. /* st1 will be built to contain the next state.
3. st = eps(s) /* Start in all states reachable from s via only ϵ-transitions.
4. Repeat:
   c = get-next-symbol(w)
   If c ≠ end-of-file then do:
   st1 = ∅.
   For all q ∈ st do:
   For all r: (q, c, r) ∈ Δ do:
   st1 = st1 ∪ eps(r).
   st = st1
   If st = ∅ then exit.
   Until c = end-of-file
5. If st ∩ A ≠ ∅ then accept else reject.

Minimizing FSMs

* To solve a real problem with a FSM, we may need to find the smallest one that does the job.
* We say that a DFSM M is minimal iff there is no other DFSM M' such that $L(M) = L(M')$ and M' has fewer states than M does.

Building a minimal DFSM for a language

* we will say that strings x and y are indistinguishable with respect to L, and we write as $x \approx_L y$ iff
  $\forall z \in \Sigma^*$ (Either both xz and $yz \in L$ or neither is)

  Ex: If $L = \{a\}^*$, then $a \approx_L aa \approx aaa$.

  If $L = \{w \in \{a,b\}^* : |w| \text{ is Even}\}$, then $a \approx_L aaa$, but it is not the case that $a \approx_L aa$ because if $z = a$, we have $aa \in L$ but $aaa \notin L$.

* Strings x and y are distinguishable with respect to L, iff they are not indistinguishable.
  So, if x and y are distinguishable, then there exists atleast one string z such that one but not both of xz and yz is in L.

Note: $\approx_L$ is an Equivalence relation because it is

* Reflexive: $\forall x \in \Sigma^* (x \approx_L x)$, because $\forall x, z \in \Sigma^* (xz \in L \leftrightarrow xz \in L)$

> Symmetric: $\forall x, y \in \Sigma^* (x \approx_L y \rightarrow y \approx_L x)$, because $\forall x, y, z \in \Sigma^*$
$(xz \in L \leftrightarrow yz \in L) \leftrightarrow (yz \in L \leftrightarrow xz \in L))$

> Transitive: $\forall x, y, z \in \Sigma^* ((x \approx_L y) \wedge (y \approx_L w)) \rightarrow (x \approx_L w)$,
because $\forall x, y, z \in \Sigma^* (((xz \in L \leftrightarrow yz \in L) \wedge (yz \in L \leftrightarrow wz \in L))$
$\rightarrow (xz \in L \leftrightarrow wz \in L))$

Three notations to describe the Equivalence classes of $\approx_L$

1. [1] [2] etc will refer to Explicitly numbered classes.
2. [x] describes the Equivalence class that contains the string x.
3. [Some logical Expression P] describes the Equivalence class of strings that Satisfies P.

Since $\approx_L$ is an Equivalence relation, its Equivalence class constitute a partition of the set Σ^*. So

1. No Equivalence class of $\approx_L$ is Empty, and
2. Every string in Σ^* is in Exactly one class of $\approx_L$.

Equivalence classes of $\approx_L$ correspond Exactly to the States of the minimum DFSM that accepts L. Every string in Σ^* will drive the DFSM to Exactly one state.

Given some language L, how can we determine $\approx_L$?

Ans: Any pair of strings x and y are related via $\approx_L$ unless there exists some z that could cause one to be in L and the other not to be.

one way to start the process is to begin lexicographically enumerating the strings in Σ^* and see whether they are distinguishable or not.

Ex(i): Let $\Sigma = \{a, b\}$. Let $L = \{\omega \in \Sigma^* :$ Every a is immediately followed by a b$\}$. Determine the equivalence classes of $\approx_L$.

Ans: start by creating a first class [1] and assign ϵ to it. Now consider a. It is distinguishable from ϵ since $\epsilon ab \in L$ but $aab \notin L$. So create a new equivalence class [2] and put a in it.

Now consider b. $b \approx_L \epsilon$. They are both in L as long as their continuation doesnot violate the rule. If their continuation violate the rule, they are both out. So b goes into [1].

Next we try aa. It is distinguishable from the strings in [1] because the strings in [1] are in L but aa is not. Also noticed that aa is distinguishable from a and so cannot be in [2], because a still has a chance to become in L if it is followed by a string that starts with a b. But aa is out, no matter what comes next. So create a new class [3] and put aa in it.

ba bba

Continue in this fashion until we discover the property that holds of each equivalence class.

[1] {ϵ, b, ab, bb, abb, - -} [all strings in L]

[2] {a, ba, aba, - - }

[3] {aa, aaa, aab, - - }

Key ~~point~~ observations

1) No equivalence class can contain both strings that are in L and strings that are not.

2) If there are strings that would take a DFSM for L to the dead state, then there will be one equivalence class of $\approx_L$ that corresponds to the dead state.

3) Some equivalence class contains ϵ. It will correspond to the start state of the minimal machine that accepts L.

Building a minimal DFSM from $\approx_L$.

1) The equivalence classes of $\approx_L$ becomes the states of M.

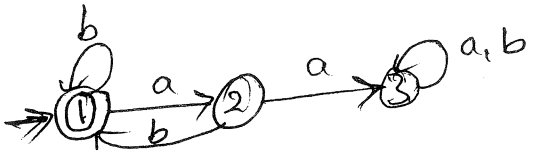
2) The start state is [ϵ] = [1]

3) The accepting states are all equivalence classes that contain strings in L, namely [1]

2) $\delta([x], a) = [xa]$ ✓

Ex: Equivalence class [1] contains the string ϵ. If the character a follows ϵ, the resulting string, a, is in equivalence class [2]. So, we create a transition from [1] to [2] labeled a

Equivalence class [2] contains the string a. If the character b follows a, the resulting string, ab is in equivalence class [1]. So we create a transition from [2] to [3] labeled b. And so forth.



Ex (2): Let $\Sigma = \{a, b\}$. Let $L = \{w \in \{a,b\}^* : \text{no two adjacent characters are the same}\}$.

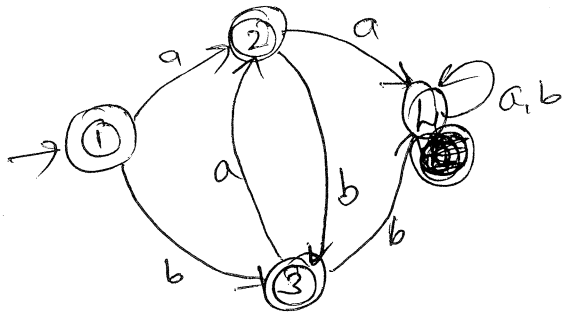
Sol: Equivalences classes of $\approx_L$ are,

- [1] [ϵ]
- [2] [a, ba, aba, ...]
- [3] [b, ab, ...]
- [4] [aa, bb, abb, ...]

> Equivalence classes of $\approx_L$ become the states of M

> Start state is $[\epsilon] = [1]$

> Accepting states are all equivalence classes that contain strings in L, namely [1], [2] and [3].



Minimizing an Existing DFSM

Suppose that we ~~have~~ already have a DFSM M that accepts L. We can construct a minimal DFSM to accept L by starting with M rather than with $\approx_L$. Two approaches to constructing a minimization algorithm:

1. Begin with M and collapse redundant states, getting rid of one at a time until the resulting machine is minimal.
2. Begin by ~~over~~ over clustering the states of L into just two groups, accepting and nonaccepting. Then iteratively split those groups apart until all the distinctions that L requires have been made.

Algorithm Based on the Second Approach

The goal is to end up with a minimal machine in which all equivalent states of M have been collapsed. We will use the following ~~defn~~:

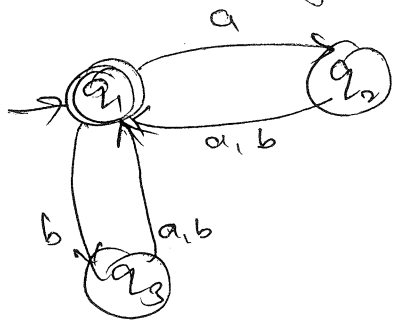
Defn: We will say that two states q and p in M are

Equivalent, and we write $q \equiv p$ iff for all strings $w \in \Sigma^*$, either w drives M to an accepting state from both q and p or to a rejecting state from both q and p.

Ex: A nonminimal DFSM with two equivalent states

Let $\Sigma = \{a, b\}$. Let $L = \{w \in \Sigma^* : |w| \text{ is Even}\}$. $L = \{\epsilon, aa, ab, ba, bb$

Consider the following FSM that accepts L:



Here $q_2 \equiv q_3$

We can define a series of equivalence relations $\equiv^n$, for values of $n \geq 0$. For any two states p and q, $p \equiv^n q$ iff p and q yield the same outcome for all strings of length n. So:

- $p \equiv^0 q$ iff they behave equivalently when they read ϵ. (ie if they are both accepting or both rejecting states).
- $p \equiv^1 q$ iff they behave equivalently when they read any string of length 1. It is equivalent to saying that any single character sends them to states that are $\equiv^0$ to each other.
- $p \equiv^2 q$ iff they behave equivalently when they read any string of length 2. It is equivalent to saying that any single character sends them to states that are $\equiv^1$ to each other. and so forth.

Formally, for all $p, q \in K$

- $p \equiv^0 q$ iff they are both accepting or both rejecting states
- For all $n \geq 1$, $q \equiv^n p$ iff:
  - $q \equiv^{n-1} p$, and
  - $\forall a \in \Sigma \quad (\delta(p,a) \equiv^{n-1} \delta(q,a))$

minDFSM, a minimization algorithm takes as its input a DFSM $M = (K, \Sigma, \delta, s, A)$. minDFSM will construct a minimal DFSM M' that is equivalent to M. It begins by constructing $\equiv^0$, which divides the states of M into almost two equivalence classes, corresponding to A and $K-A$. If M has no accepting states or if all of its states are accepting, then there will be only one non-empty equivalence class and we can quit since there is a one-state machine that is equivalent to M. Therefore, we consider only those cases where both A and $K-A$ are non-empty.

minDFSM executes a sequence of steps, during which it constructs the sequence of equivalence relations $\equiv^1, \equiv^2, \ldots$. To construct $\equiv^{k+1}$, minDFSM begins with $\equiv^k$. Then it splits equivalence classes of $\equiv^k$ whenever it discovers some pair of states that do not behave equivalently. minDFSM halts when it discovers that $\equiv^n$ is the same as $\equiv^{n+1}$.

minDFSM(M: DFSM) ≡

1. Classes = {A, K−A} /* Initially, just two classes: accepting and rejecting
2. Repeat until a pass at which no change to classes has been made:
   2.1 newclasses = ∅. /* At each pass, we build a new set of classes, splitting old ones as necessary. Then this new set becomes the old set, and the process is repeated.
   2.2 For each Equivalence Class e in classes, do:
   If e contains more than one state, see if it needs to be split:
   For each state q in e do:
   For each character c in Σ do:
   Determine which element of classes q goes to if c is read.

If there are any two states p and q such that there is any character c such that, when c is read, p goes to one element of classes, and q goes to another, then p and q must be split. Create as many new classes as are necessary so that no state remains in the same class with a state whose behavior differs from its. Insert those classes into newclasses.

If there are no states whose behavior differs, no splitting is necessary. Insert e into newclasses.

2.3 Classes = newclasses.

/* States of the minimal machine will correspond exactly to the elements of classes at this point */.

3. Return $M' = (\text{classes}, \Sigma, \delta, [s_M], \{[q]: \text{the elements of } q \text{ are in } A_M\})$, where δ_M is constructed as follows:

If $\delta_M(q, c) = p$, then $\delta'_M([q], c) = [p]$

Clearly, no class that contains a single state can be split.

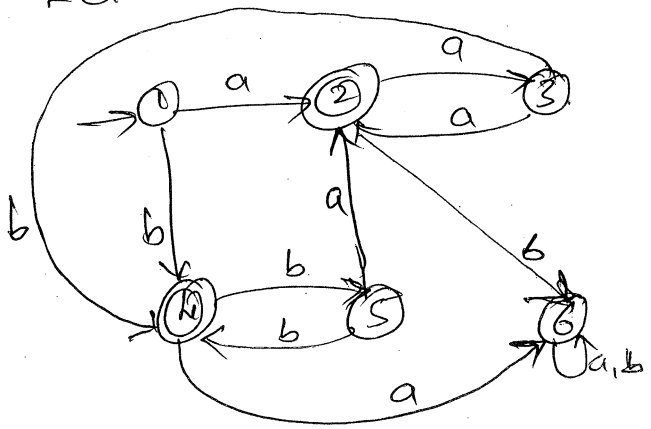
So if $|K|$ is k, then maximum no of times that minDFSM can split is $k-1$. Thus minDFSM must halt in a finite no of steps. M' is minimal DFSM that is eqt to M

Since:

- M' is minimal: It splits classes and thus creates new states only when necessary to simulate M, and
- $L(M') = L(M)$

Example: Find a minimal DFSM from the given DFSM

Let $\Sigma = \{a, b\}$. Let $M =$



Soln: Initially, Classes = {[2,4], [1,3,5,6]}

Step 1:

((2,a), [1,3,5,6]) ((4,a), [1,3,5,6])

((2,b), [1,3,5,6]) ((4,b), [1,3,5,6])

No splitting required here

((1,a), [2,4]) ((3,a), [2,4]) ((5,a), [2,4]) ((6,a), [1,3,5,6])

((1,b), [2,4]) ((3,b), [2,4]) ((5,b), [2,4]) ((6,b), [1,3,5,6])

Two different patterns, so we must split into two classes, [1,3,5] and [6].

Note: [6] has the same behaviour as [2,4] after reading a single character, it cannot be combined with [2,4] because they do not share the behaviour after reading no character.

Classes = {[2,4], [1,3,5], [6]}

Step 2:

((2,a), [1,3,5]) ((4,a), [6])

((2,b), [6]) ((4,b), [5])

There two must be split

((1,a), [2,4]) ((3,a), [2,4]) ((5,a), [2,4])

((1,b), [2,4]) ((3,b), [2,4]) ((5,b), [2,4])

No splitting required

Classes = {[2], [4], [1,3,5], [6]}

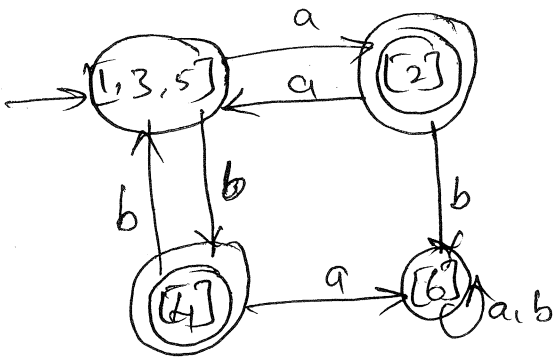
Step 3

((1,a),[2]) ((3,a),[2]) ((5,a),[2])

((1,b),[4]) ((3,b),[4]) ((5,b),[4])

No splitting required

So minDFSM returns M' =



Ex(2): Minimize the following DFSM

Same as a/M

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

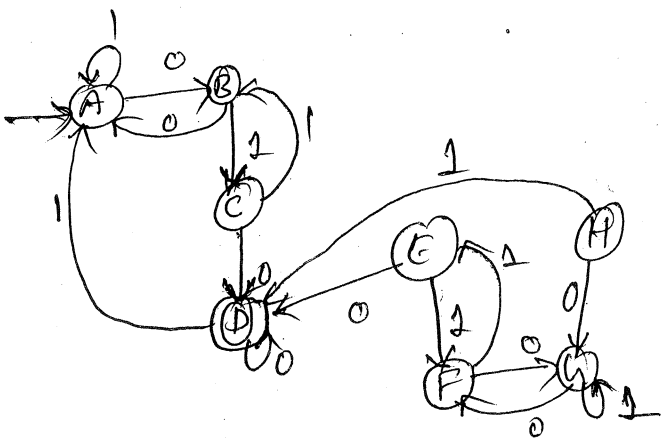
Soln Initially, class = {[A], [B, C, D, E, F, G, H]}

Step 1: ((B,0),[A]) ((C,0),[B,C,D,E,F,G,H])

((B,1),[B,C,D,E,F,G,H]) ((C,1),[B,C,D,E,F,G,H])

((D,0),[B,C,D,E,F,G,H]), ((D,1),[A])

((E,0),[B,C,D,E,F,G,H]), ((E,1),[B,C,D,E,



Soln: Initially, Classes = {[D], [A, B, C, E, F, G, H]}

Step1: ((D,0), [D])

((D,1), [A,B,C,E,F,G,H])

((A,0), [A,B,C,E,F,G,H]) ((B,0), [A,B,C,E,F,G,H])

((A,1), [A,B,C,E,F,G,H]) ((B,1), [A,B,C,E,F,G,H])

((C,0), [D]), ((E,0), [D])

((C,1), [A,B,C,E,F,G,H]) ((G,1), [A,B,C,E,F,G,H])

((F,0), [A,B,C,E,F,G,H]) ((G,0), [A,B,C,E,F,G,H])

((F,1), [A,B,C,E,F,G,H]) ((G,1), [A,B,C,E,F,G,H])

((H,0), [A,B,C,E,F,G,H])

((H,1), [D])

classes = {[D], [A,B,F,G] [C,E], [H]}

Step2:

((A,0), [A,B,F,G]) ((B,0), [A,B,F,G]) ((F,0), [A,B,F,G]) ((G,0), [A,B,E,

((A,1), [A,B,F,G]) ((B,1), [C,E]) ((F,1), [C,E]) ((G,1), A,B,F,G

((C,0), [D]) ((E,0), [D])

((C,1), [A,B,F,G]) ((E,1), [A,B,F,G])

Dr. Krishna A N BE, ME, Ph.D.
Professor, Dept. of CSE,
SJB Institute of Technology,
BGS Health and Education City,
Kengeri, Bangalore - 560 060.
Mob: 9481189899

Classes = { [D], [A,G], [B,F], [C,E], [H] }

NA

Step3:

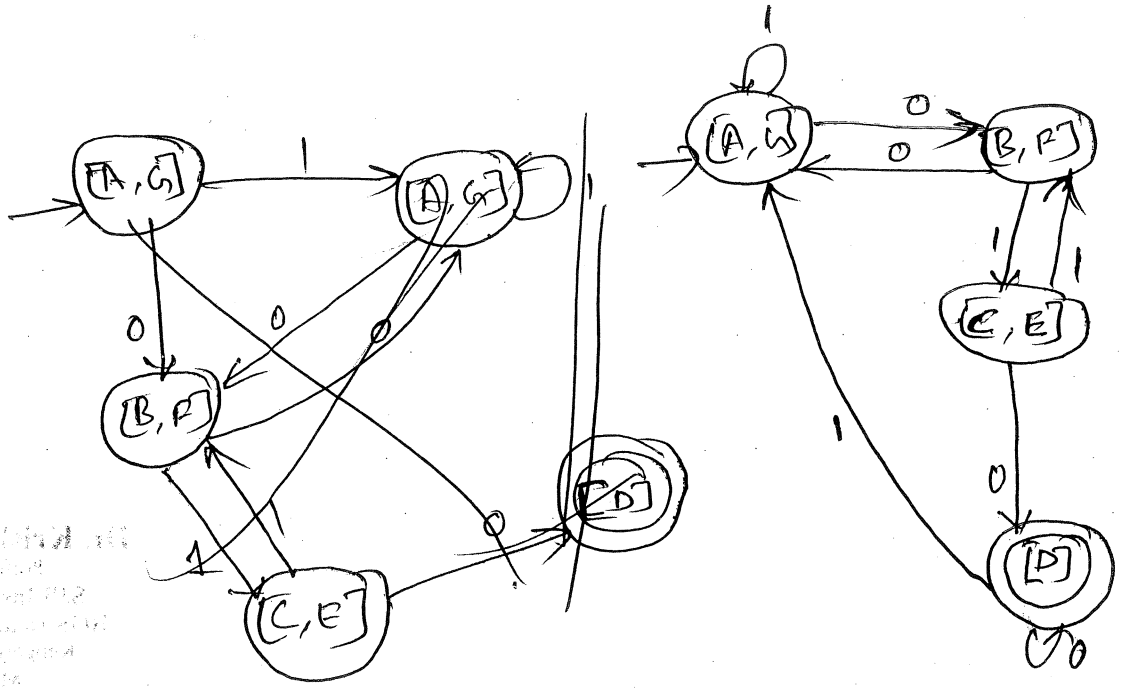
((A,0), [B,E]) ((G,0), [B,F])
((A,1), [A,G]) ((G,1), [A,G]) } No Splitting

((B,0), [A,G]) ((F,0), [A,G])
((B,1), [C,E]) ((F,1), [C,E]) } No Splitting

((C,0), [D]) ((E,0), [D])
((C,1), [B,F]) ((E,1), [B,F]) } No Splitting

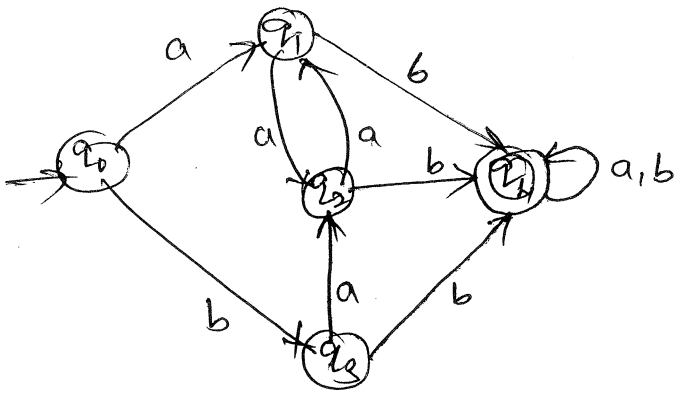
Classes = { [D], [A,G], [B,F], [C,E], [H] }

M' =



July 2018
6M

Minimize the following finite automata



Soln Initially, classes = $\{[q_0, q_1, q_2, q_3], [q_4]\}$

Step 1

$((q_0, a), [q_0, q_1, q_2, q_3])$ $((q_1, a), [q_0, q_1, q_2, q_3])$ -

$((q_0, b), [q_0, q_1, q_2, q_3])$ $((q_1, b), [q_4])$ - splitting is required

$((q_2, a), [q_0, q_1, q_2, q_3])$ $((q_3, a), [q_0, q_1, q_2, q_3])$ -

$((q_2, b), [q_4])$ $((q_3, b), [q_4])$ -

classes = $\{[q_0], [q_1, q_2, q_3], [q_4]\}$

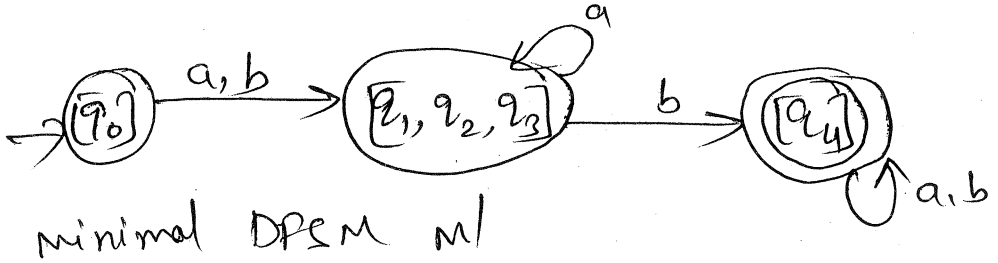
Step 2

$((q_1, a), [q_1, q_2, q_3])$ $((q_2, a), [q_1, q_2, q_3])$ $((q_3, a), [q_1, q_2, q_3])$

$((q_1, b), [q_4])$ $((q_2, b), [q_4])$ $((q_3, b), [q_4])$

No splitting required

classes = $\{[q_0], [q_1, q_2, q_3], [q_4]\}$



minimal DFSM //

Canonical form for Regular Languages

→ A canonical form for some set of objects in C assigns exactly one representation to each class of "equivalent" objects in C. ie two objects in C have the same representation if they are "equivalent". ~~in the sense of~~

- Two FSMs have a canonical form if they accept the same language

Ex- we could test whether two FSMs are equivalent (ie they accept the same language).

It is sufficient to construct the canonical form for each of them and test whether the two forms are identical →

- All minimal ~~M's~~ machines for L(M) are identical except possibly for state names. So, we could define a standard way to name states, we could define a canonical machine to accept L(M).
- Following algorithm does this by using the state-naming convention

buildFSMCanonicalform(M: FSM)

1. M' = ndfsmtodfsm(M)
2. M# = minDFSM(M')
3. Create a unique assignment of names to the states of M# as follows:

3.1. Call the start state q_0.

3.2. Define an order on the Elements of Σ.

3.3. Until all States have been named do:

Select the lowest numbered named State that has not yet been Selected. Call it q.

Create an ordered list of the transitions out of q by the order imposed on their labels.

Create an ordered list of the as yet Unnamed States that those transitions Enter by doing the following:

If the first transition is (q, c_1, p_1), then put p_1 first.

If the 2nd transition is (q, c_2, p_2) and p_2 is not already on the list, put it next. If it is already on the list, Skip it. Continue until all ~~transitions~~ transitions have been Considered. Remove from the list any States that have already been named.

Name the States on the list that was just created: Assign to the first one the name q_k, where k is the Smallest index. Assign the next name to the next State and So forth until all have been named.

4. Return $M\#$.

Given two FSMs M_1 and M_2, buildFSMCanonicalform(M_1) = buildFSMCanonicalform(M_2) iff $L(M_1) = L(M_2)$. It provides the basis for a Simple way to test whether an FSM accepts any Strings or whether two FSMs are Equivalent.

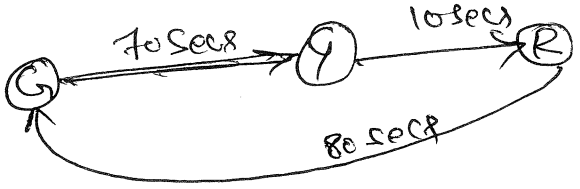
Finite State Transducers

* So far we have used finite state machines as language recognizers. We have analyzed a machine M whether or not it ends in accepting state.
* It is simple to augment finite state model to allow for output at each step of a machine operation.
* Many finite state transducers are loops that simply run for ever processing inputs.
* one simple kind of finite state transducer associates an output with each state of a machine M. That output is generated whenever M enters the associate state. Deterministic finite state transducers of this sort are called Moore machines.
* A moore machine M is a seven-tuple $(K, \Sigma, O, \delta, D, s, A)$ where
  - K - finite set of states
  - Σ - input alphabet
  - O - output
  - $s \in K$ - start state
  - $A \subseteq K$ - set of accepting states
  - δ - transition function. It is a function from $(K \times \Sigma)$ to (K) and
  - D - Display or output function. It is a function from (K) to (O^*).

A moore machine M computes a function $f(w)$ if, when it reads the input string w, its output sequence is $f(w)$.

Ex(1): Typical ~~United States~~ Traffic light.

Consider the following ~~the~~ controller for a single direction of a very simple traffic light (which ignores, time of day traffic, the need to let Emergency vehicles through, etc).



* > States correspond to light's colors: Green, Yellow & Red
  > Definition of the start state is arbitrary
  > Three inputs, all of which are Elapsed time.

* Mealy machine – Deterministic finite state transducer permits each machine to output any finite sequence of symbols as it makes each transition (ie as it reads each symbol of its input). A mealy machine M is a 6-tuple $(K, \Sigma, O, \delta, s, A)$, where

  K – finite set of states

  Σ – input alphabet

  O – output –"–

  $s \in K$ – start state

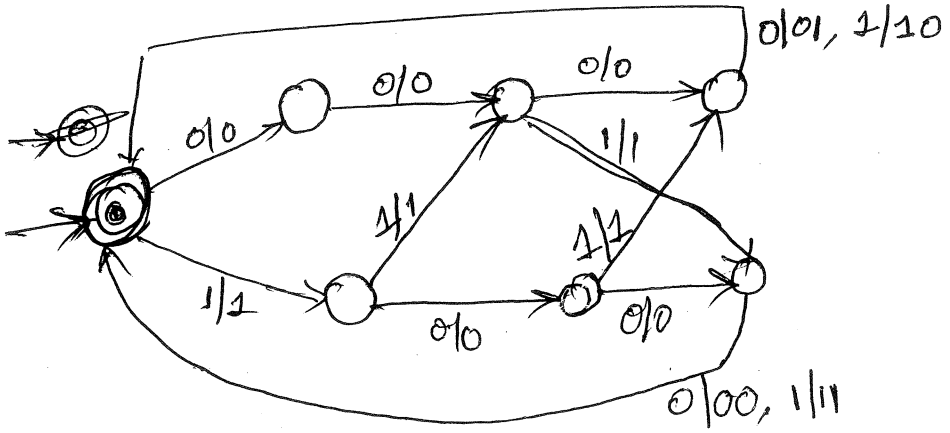
  $A \subseteq K$ – set of accepting states

  δ – transition function. from $(K \times \Sigma)$ to $(K \times O^*)$

A mealy m/c M Computes a function $f(w)$ of, when it reads the input string w, its output sequence is $f(w)$.

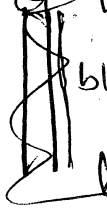
Ex(1) Generating Parity Bits

The following mealy m/c adds an odd parity bit after every four digits that it reads. We will use the notation a/b on an arc to mean that the transition may be followed if the input character is a. If it is followed, then the string b will be generated.



Ex(2): A Bar Code Reader

A bar code system that encodes just binary nos. Bar code composed of columns, each of the same width. A column can be either white or black. The job of the white columns is to delimit the black ones. A single black column encodes 0. A double black column encodes 1.



We can build a finite state transducer to read such a bar code and output a string of binary digits.

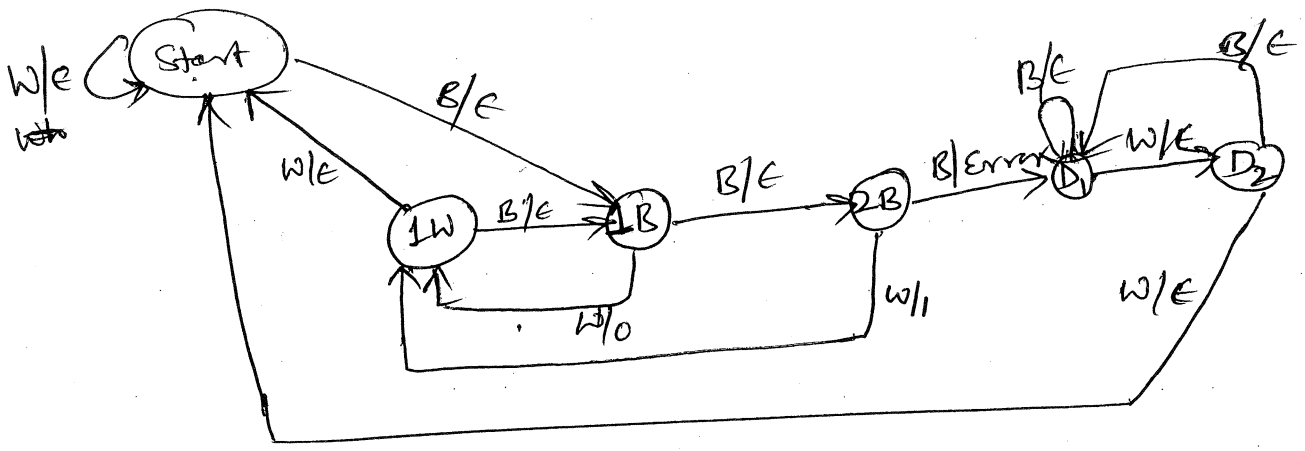
Represent a black bar with the Symbol B

— white ——— W

The input to the transducer will be a sequence of those Symbols, Corresponding to reading the barcode left to right.

Assume that Every correct bar code Starts with a black column, so white space ahead of the first black column is ignored.

Also assume that after Every Caplete bar code there are atleast two white columns. So, the reader Should at that point, reset to be ready to read the next code. If the reader Sees three or more black columns in a row, it must indicate an Error and Stay in its Error State until it is reset by Seeing two white columns.



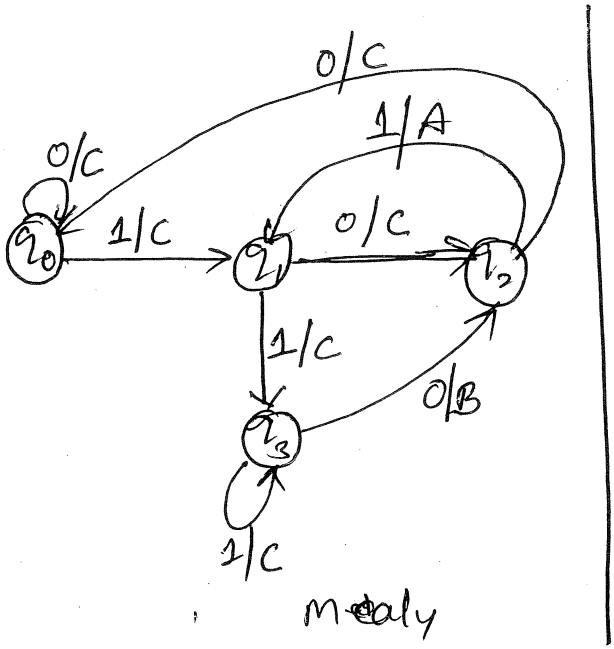
July 18
6M

Construct the Mealy M/c for the following:

(i) Design a mealy M/c for a binary input sequence s.t if it has a substring 101, the M/c o/p A. If input has substring 110, the M/c outputs B. Otherwise output C.

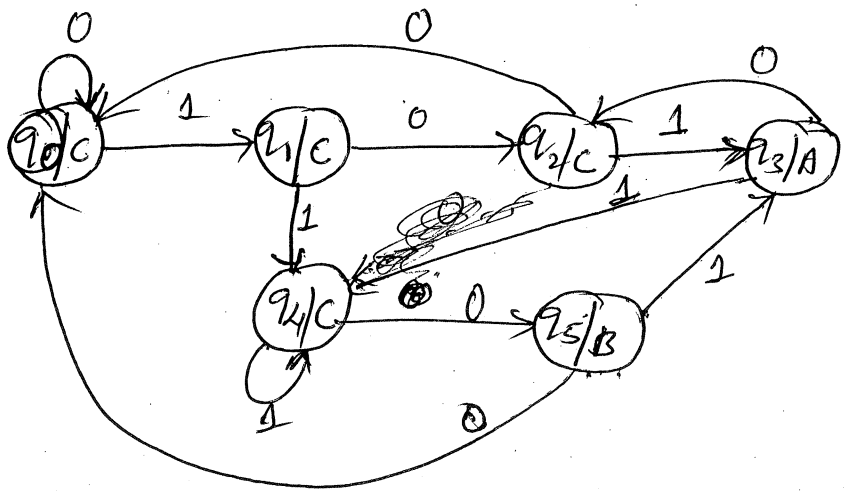
(ii) Design a Mealy M/c that takes binary no as input and produces 2's complement of that no. as output. Assume that string is read from LSB to MSB and end carry is discarded

Soln (i)



1100

(ii) Moore M/c



10001

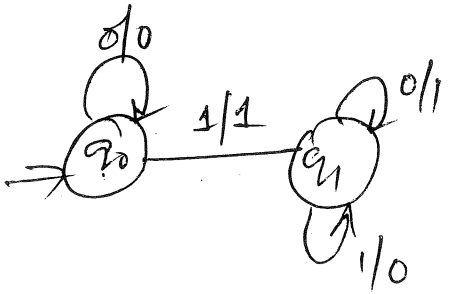
(ii) Assume that input is read from LSB to MSB keep the binary as it is until we read first 1. Keep that 1 as it is then change remaining 1's by 0's and 0's by 1's.

Eg) let binary be 1011 ← ... (read from LSB)
0101

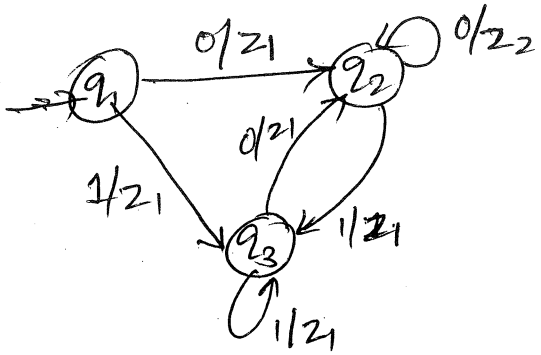
1011
1's 0100
2's 1
0101

(2) ——— 111
← ...
001

111
1's 000
2's 1
001



June 2018
4M Convert the following mealy m/c to moore m/c.



Procedure: 1) Construct the transition table for mealy m/c

Present State	NXT State I/p = 0	output	NXT State I/p = 1	output
q_1	q_2	z_1	q_3	z_1
q_2	q_2	z_2	q_3	z_1
q_3	q_2	z_1	q_3	z_1

Step 2: Find out those states which have more than 1 outputs associated with them

q_2 associated with both the outputs z_1 & z_2

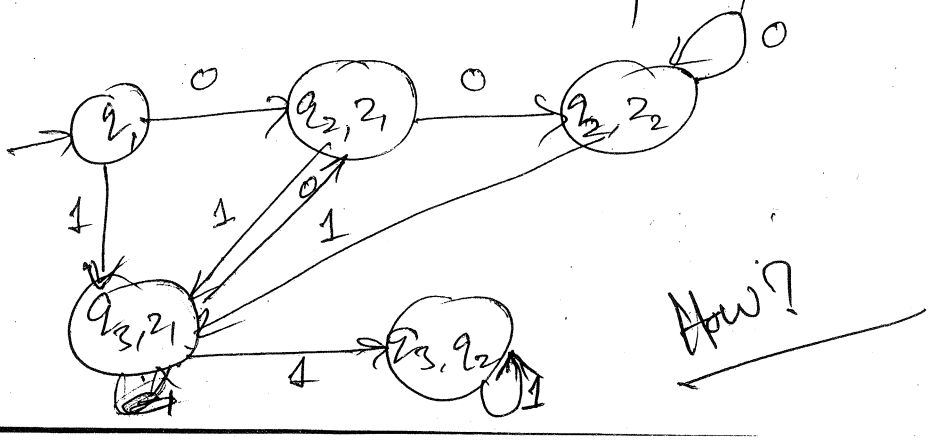
Step 3: Create two states for this. (~~q2 stated with o/p z1 and~~ z_2)

ie (q_2, z_1) and (q_2, z_2)

~~(q_3, z_1) q_1~~ q_3 - only z_1 o/p - return

~~Step 4: Create an Empty mose M/C with new generated State, Split both State~~

Present State	Next state, Input 0	Next state, Input 1	output 0	output 1
q_1	q_2, z_1	q_3, z_1	z_1	z_1
q_2, z_1	q_2, z_2	q_3, z_1	z_2	z_1
q_2, z_2	q_2, z_2	q_3, z_1	z_2	z_1
q_3, z_1	q_2, z_1	q_3, z_1	z_2	z_1
q_3, z_2	q_2, z_1	q_3, z_1	z_1	z_1



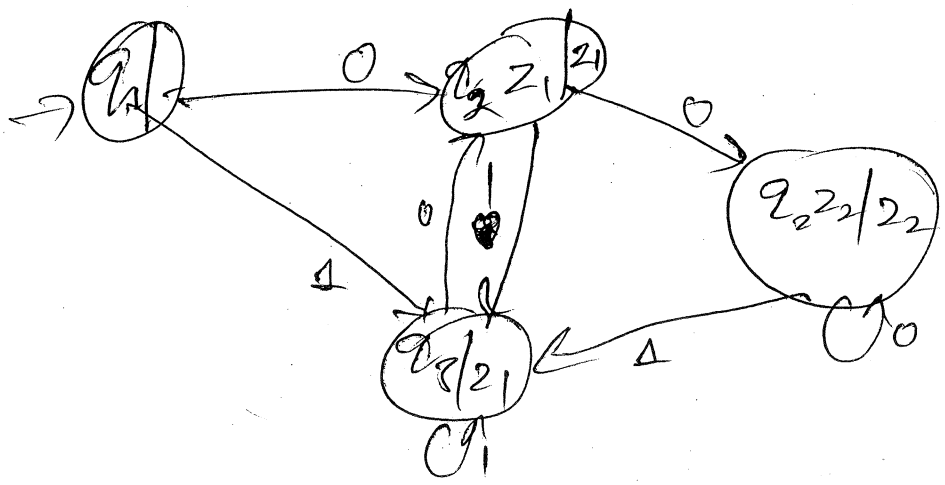
How?

Present State	Input a=0 State	a=0 o/p	a=1 State	a=1 o/p
q_1	$q_2 z_1$	z_1	q_3	z_1
$q_2 z_1$	$q_2 z_2$	z_2	q_3	z_1
$q_2 z_2$	$q_2 z_2$	z_2	q_3	z_1
q_3	$q_2 z_1$	z_1	q_3	z_1

by merge NLC, Copy all the States and Common o/p, two o/ps of the next state are Common.

Moore m/c Transition Table

$\otimes_1$ c	Input a=0	Input a=1	output
q_1	$q_2 z_1$	q_3	-
$q_2 z_1$	$q_2 z_2$	q_3	z_1
$q_2 z_2$	$q_2 z_2$	q_3	z_2
q_3	$q_2 z_1$	q_3	z_1



Conversion from Mealy to Moore

Ex.

Present State	Next State a=0		Next State a=1	
	State	o/p	State	o/p
→ q_0	q_3	0	q_1	1
q_1	q_0	1	q_3	0
q_2	q_2	1	q_2	0
q_3	q_1	0	q_0	1

1. Determine the no. of different output associated with q_i in the next state column.
2. Split q_i into different states according to the o/p associated with it. In this example.
   q_1 is associated with 0 and 1
   q_2 ——— 0 and 1
3. So Split both states i.e. into q_{10}, q_{11} and q_{20}, q_{21}. Now the transition table becomes

	a=0		a=1	
→ q_0	q_3	0	q_{11}	1
q_{10}	q_0	1	q_3	0
q_{11}	q_0	1	q_3	0
q_{20}	q_{21}	1	q_{20}	0
q_{21}	q_{21}	1	q_{20}	0
q_3	q_{10}	0	q_0	1

Now in moore M/C, we copy all the states and common output because in moore M/C, the o/p's of two next state are common.

Pr. State	Next State a=0	Next State a=1	output
→ q_0	q_3	q_{11}	1
q_{10}	q_0	q_3	0
q_{11}	q_0	q_3	1
q_{20}	q_{21}	q_{20}	0
q_{21}	q_{21}	q_{20}	1
q_3	q_{10}	q_0	0

$(a-b)/n$

$a \equiv b \bmod n$

$7 \equiv 1 \bmod 3$

$7-1/3$

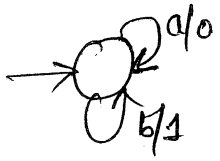
Bidirectional Transducers

* A process that reads an input string and constructs a corresponding output string can be described using bidirectional transducers. It describes the relationship between inputs and outputs. This model can be run in two directions.

Ex(1): Letter Substitution

When we define a regular language, it doesnot matter what ~~alphabet~~ alphabet we use. Any thing that is true of a language L defined over the alphabet {a,b} will also be true of the language L' that contains exactly two strings in L except that Every a has been replaced by a 0 and Every b has been replaced by 1.

We can build a simple ~~dtr~~ bidirectional transducer that can convert strings in L to strings in L' and vice versa.



Ex(2): A way to find similar sounding names

People change the spelling of their names. Sometimes the spelling was changed for them when they immigrated to a country with a different language, a different set of sounds and maybe a different writing system. For various reasons one might want to identify other people to ~~whom~~ whom one is

related. But, because of Spelling Changes, it is not Sufficient to look for people with exactly the same last name. The Soundex System is a Solution to this problem.

The System maps any name to a four character code that is derived from the original name. So, find related names, one can run the Soundex transducer in one direction, from a starting name to its Soundex code and then in the other direction, ~~from the~~ from the code to the other names that share that code.

Ex: If we start with the Kaylor, it will produce the Soundex code K460. If we then use that code and run the transducer backwards, we can generate the names Kahler, Kaler, Kaylor, Keeler, Keller, Kelleher, Keller, ...

The Soundex System is described by the following set of rules for mapping from a name to a Soundex code.

1. If two or more adjacent letters (including the first in the name) would map to the same no. If rule 3.1 were applied to them, remove all but the first in the sequence.
2. The first character of the Soundex code will be the first letter of the name.
3. For all other letters of the name do:
   3.1 Convert the letters using the following correspondences.
   B, P, F, V = 1

C, S, G, J, K, Q, X, Z = 2

D, T = 3

L = 4

M, N = 5

R = 6

3.2 Delete all instances of the letters A, E, I, O, U, Y, H & W.

4. If the string contains more than ~~three~~ 3 nos. delete all but the leftmost three.

5. If the string contains fewer than 3 nos, pad with 0's on the right to get three.