Finite Automata and Regular Languages

FUNDAMENTALS

Alphabet: alphabet is a finite non-empty set of symbols
Σ denotes an alphabet

String:

- a string over an alphabet ‘A’ is a finite ordered sequence of symbols from ‘A’. The length of the string is the number of symbols in string with repetitions counted
- an empty string denoted by ‘ε’, is the (unique) string of length zero.
- If S and T are sets of strings then ST = {xy | x є S and y є T }
- Given an alphabet A,
  - $A^o = \{ ε\}$
  - $A^{n+1} = A.A^n$
  - . . .
  - $A^* = U(n = 0 \text{ to infinity}) A^n$

LANGUAGES

- A language ‘L’ over Σ is any finite or infinite set of strings over Σ.
- The elements in L are strings – finite sequences of symbols
- A language which does not contain any elements is called an ‘empty language’
- Empty language resembles empty set => { } = Φ ≠ ε
- A language L over an alphabet A is subset of A^* i.e. L (A^*
- An empty language is a language that does not accept any strings including ε. (->O)
- A language which only accepts ε (->OO (concentric))

OPERATIONS

Operations on strings

1. **Concatenation**: Combines two strings by putting one after the other (a.b)
   - Concatenation of empty string with any other string gives the string itself

2. **Substring:** If ‘w’ is a string, then ‘v’ is a substring of ‘w’ if there exists string x and y such that w = xvy. ‘x’ is called the prefix and ‘y’ is called the suffix of w.

3. **Reversal:** if ‘w’ is a string, then w^R is reversal of string spelled backwards.
   - $x=(x^R)^R$
   - $(xz)^R = z^R.x^R$

4. **Kleen star operation:** Let ‘w’ be a string w^* is set of strings obtained by applying any number of concatenations of w with itself, including empty string.
   - Example: a^* = {ε, a, aa, aaa, ...}

Operations on Langauges

1. **Union:** Given some alphabet Σ, for any two languages, L_1, L_2 oven Σ, the union L_1 union L_2 of L_1 and L_2 is the language L_1 union L_2 = {w є $Σ^*$ | w є L_1 or w є L_2 }
2. **Intersection:** Given some alphabet Σ, for any two languages, L_1, L_2 oven Σ, the intersection $L_1 \cap L_2$ of L_1 and L_2 is the language $L_1 \cap L_2$ = {w є $Σ^*$ | w є L_1 and w є L_2 }

3. **Difference / Relative Complement:** Given some alphabet Σ, for any two languages, L_1, L_2 oven Σ, the differnece $L_1 - L_2$ of L_1 and L_2 is the language $L_1 - L_2 = \{w \in \Sigma^* \mid w \in L_1$ and w does not belong to $L_2 \}$
4. **Concatenation:** Given some alphabet Σ, for any two languages, L_1, L_2 oven Σ, the concatenation $L_1 L_2$ of L_1 and L_2 is the language $L_1 L_2 = \{w \in \Sigma^* \mid \exists\, u \in L_1, \exists v \in L_2$ and $w = uv\}$
   - **Properties**
   - $L\emptyset = \emptyset = \emptyset L$
   - $L\{\varepsilon\} = L = \{\varepsilon\} L$
   - $(L_1 \cup \{\varepsilon\}) L_2 = L_1L_2 \cup L_2$
   - $L_1(L_2 \cup \{\varepsilon\}) = L_1L_2 \cup L_1$
   - $L^nL = LL^n = L^{n+1}$
     - $L_1L_2 \neq L_2L_1$
5. **Kleen * Closure (L^*)**: Given an alphabet Σ, for any language L over Σ, the * closure L^* of L is language, $L^* = \bigcup_{n>=0} L^n$
6. **Kleen + Closure (L^+)**: Given an alphabet Σ, for any language L over Σ, the kleen $^+$ closure L^+ of L is language, $L+ = \bigcup_{n>=1} L^n$

   Properties
   - $\emptyset^* = \{\varepsilon\}$
   - $L^+ = L^*L$
   - $(L^*)^* = L^*$
   - $L^*L^* = L^*$

FINITE STATE MACHINES

FSM is the simplest computational model of limited memory computers. It is designed to solve decision problems, i.e, to decide whether the given input satisfies certain conditions. The next state and output of FSM is a function of input and the current state.

Types of FSM
- Mealy machine
- Moore machine

Finite Automata
- FA is a state machine that comprehensively captures all possible states and transitions that a machine can take while responding to a stream (sequence) of input symbols.
- FA is recognizer of ‘regular languages’

State Machine
- Finite state machines
  - Mealy machine
  - Moore machine
- Finite automata
  - DFA
  - NFA
  - epsilon – NFA

Types of Finite Automata

1. **Deterministic Finite Automata**
   - DFA can exist in only one state at a time
   - DFA is defined by 5-tuple : {Q, Σ, q_0, F, δ}
     - Q → Finite number of states (elements)
     - Σ → Finite set of symbols (alphabets)
     - q_o → Start/Initial state
     - F → Set of final states
     - δ → Transition function, which is a mapping between
       - δ: Q X Σ → Q
   - How to use DFA

Transition Diagram
State machines are represented by directed graphs called transition (state) diagrams.
Vertices : states
Arcs : transition
Double concentric circles: Final states

Transition Table
Transition function can be represented by tables.
Note: minimum number of states for k-divisibility is k-states

2. Non-deterministic Finite Automata
- The machine can exist in multiple states at the same time
- Each transition function maps to a set of states
- NFA is defined by 5-tuple: {Q, Σ, q_0, F, δ}
  - Q → Finite number of states (elements)
  - Σ → Finite set of symbols (alphabets)
  - q_o → Start/Initial state
  - F → Set of final states
  - δ → Transition function, which is a mapping between
    - δ: Q X Σ → 2^Q
- How to use NFA

Difference between NFA and DFA

DFA	**NFA**
1. All transitions are deterministic i.e., each transition leads to exactly one state	1. Transitions could be non-deterministic i.e., a transition could lead to a subset of states
2. For each state, the transition on all possible symbols should be defined	2. For each state, not all symbols necessarily have to be defined
3. Accepts input if the last state is in F	3. Accepts input if one of the last states is in F.
4. Practical implementation is feasible	4. Practical implementation has to be deterministic. It needs conversion from NFA to DFA

Relation between DFA and NFA

1. A language L is accepted by a DFA if and only of it is accepted by NFA
2. DFA is a special case of NFA
3. Every language accepted by NFA is also accepted by a DFA. $D_f = N_f$

NFA WITH Є – MOVES

- Є-transitions in finite automata allows a state to jump to another state without consuming any input symbol

Conversion and equivalence:
Є – NFA → NFA → DFA

NFA without Є – moves:

- Two FA, $N_Є$ and N are said to be equivalent, if $L(N_Є) = L(N)$ i.e., any language described by some $N_Є$ there is a N that accepts the same language.
- For $N_Є = \{Q, \Sigma, q_0, F, \delta\}$ and $N = \{Q, \Sigma`, q_0, F`, \delta`\}$, find
- $\delta`(q, a) = Є$ – closure (δ (Є-closure(q), a))
- $F` = \{F \cup (q_0\}\}$, if Є – closure (q_0) contains a member of F=F, otherwise
- *Note: when transforming Ne to N, only transitions are required to be changes and states remains same.*

CONVERSION OF NFA TO DFA

Let NFA be defined as $N = \{Q_N, \Sigma, q_0, F_N, \delta_N\}$
The equivalent DFA, $D = \{Q_D, \Sigma, q_0, F_D, \delta_D\}$

Step 1: $Q_D = 2^{QN}$, i.e., Q_D is set of all subsets of Q, i.e., it is power set of Q_N
Step 2: F_D is set of subsets S of Q_N such that $S \cap F_N \neq Ø$, i.e., F_D is all sets of N's states that include once accepting state of N.
Step 3: For each set, $S <= Q_N$ and for eacg input symbol a in
Σ : $\delta_D(S,a) = U_{P\,ЄS}\ \delta_N(P, a)$
That is to compute $\delta_D(S, a)$, look at all states P in S, see what states N goes to starting from P on input a, and take union of all those states.

Note: For any NFA, N with 'n' states, the corresponding DFA can have 2^n states.

MINIMIZATION OF DFA

Minimization of DFA means that we are minimizing the representation of a FA to it's least form. Here basically we replace multiple states with a single state without disturbing the representation. In a DFA states p and q are equivalent when:

$\delta(p, w)$ Є F => $\delta(q, w)$ Є F and $\delta(p, w)$ not Є F => $\delta(q, w)$ not Є F
if len(w) = 0 and p and q follow the above property: q and p are 0-equivalent
...
if len(w) = n and p and q follow the above property: p and q are n-equivalent

Therefore, instead of having two states, we can combine them into one state and decrese the number of states in the final answer.

We are using Partitioning Method here.

Step1: Identify the start and final state
Step2: If there is any state which is unreachable from the initial state, delete it.
Step3: draw the state transition table
Step4: Find the 0-equivalent sets. Separate non-final states from final states
Step5: Find the 1-equivalent sets. Separate the non equivalent from equivalent
Step5: Keep finding the n-equivalent sets until the next equivalent calculation is identical to the previous one.
Remove the dead states

<u>EQUIVALENCE BETWEEN NFA AND DFA</u>

There is a DFA for any NDA, i.e.,
L(D) = L(N).

Construction:

- In DFA or NFA, whenever an arrow is followed, there is a set of possible states. This set of states is a subset of Q.
- Track the information about subsets of states that can be reached from the initial state after following arrows.
- Consider each subset of states of NFA as a state of DFA nad every subset of states containing a final state as a final state of DFA.

Equivalence of Finite Automata

- Two automata A and B are said to be equivalent if both accept exactly the same set of input strings
- If two automata M_1 and M_2 are equivalent then
    - If there is a path from the start state of M_1 to a final state of M_1 labeled as a_1, a_2 ... a_k then there is a path from the start state of M_2 to the final state of M_2 labeled as a_1, a_2, ... a_k
    - If there is a path from the start state of M_2 to a final state of M_1 labeled as b_1, b_2 ... b_k then there is a path from the start state of M_1 to the final state of M_2 labeled as b_1, b_2, ... b_k

Union: The union of languages L and M is the set of strings that are in both L and M
Concatenation: The concatenation of languages L and M is the set of strings that can be formed by taking any string in L nad concatenating it with any string in M.

Closure, star or Kleen star of Language L:
Kleen star is denoted as L^*. It represents the set of strings that can be formed by taking any number of strings from L with repetition and concatenating them. It is a Unary operator.
L_0 is the set; we can make selecting zero strings from L.
L_0 = {epsilon}
L_1 is the language consisting of selecting one string from L.
L_2 is the language consisting of concatenations selecting two strings from L.
...
L^* is the union of L_0, L_1, ... Linf
Ex: L = {0, 10}
L^* = {0, 00, 000, 10, 010, ...}

Intersection:
Let two DFAs M_1 and M_2 accept the languages L_1 and L_2.
$M_1 = \{Q_1, \Sigma, q_0^1, F_1, \delta_1\}$
$M_2 = \{Q_2, \Sigma, q_0^2, F_2, \delta_2\}$
The intersection of M_1 and M_2 can be given as
$M = \{Q, \Sigma, q_0, F, \delta\}$
Q = Pairs of states, one from M_1 and one from M_2 i.e.,
$Q = \{(q_1, q_2) \mid q_1 \in Q_1$ and $q_2 \in Q_2\}$
$Q = Q_1 \times Q_2$
$q_0 = (q_0^1, q_0^2)$
$\delta((q_i^1, q_j^2), x) = (\delta_1(q_i^1, x), \delta_2(q_j^2, x))$
$F = \{(q_1, q_2) \mid q_1 \in F$ and $q_2 \in F\}$

Union:
Let two DFAs M_1 and M_2 accept the languages L_1 and L_2.
$M_1 = \{Q_1, \Sigma, q_0^1, F_1, \delta_1\}$
$M_2 = \{Q_2, \Sigma, q_0^2, F_2, \delta_2\}$
The union of M_1 and M_2 can be given as
$M = \{Q, \Sigma, q_0, F, \delta\}$
Q = Pairs of states, one from M_1 and one from M_2 i.e.,
$Q = \{(q_1, q_2) \mid q_1 \in Q_1$ and $q_2 \in Q_2\}$
$Q = Q_1 \times Q_2$
$q_0 = (q_0^1, q_0^2)$
$\delta((q_i^1, q_j^2), x) = (\delta_1(q_i^1, x), \delta_2(q_j^2, x))$
$F = \{(q_1, q_2) \mid q_1 \in F$ or $q_2 \in F\}$

Difference:
The difference of L1 and L2 can be given as:
L1 – L2 with $M = \{Q, \Sigma, q_0, F, \delta\}$
$Q = Q_1 \times Q_2$
$\delta((q_i^1, q_j^2), x) = (\delta_1(q_i^1, x), \delta_2(q_j^2, x))$
$F = \{(q_1, q_2) \mid q_1 \in F$ and q_2 does not $\in F\}$

Reversing a DFA:
- M is a DFA which recognizes the language L
- M^R will accept the language L^R
- To construct M^R
    - Reverse all transitions
    - Turn the start state to the final state
    - Turn the final states to the start state
    - Merge states and modify the FA such that the resultant contains a single start state.

MEALY AND MOORE MACHINES

Moore Machine
A Moore machine is a finite state machine, where outputs are determined by current state alone. A moore machine associates an output symbol with each state and each time a state is entered, an output is obtained simultaneously. So, first output always occurs as soon as the machine starts.

Moore machine is defined by 6-tuple:
$\{Q_1, \Sigma, \delta_1, q_0, \Delta, \lambda\}$

- $Q \rightarrow$ Finite number of states
- $\Sigma \rightarrow$ Finite set of input symbol
- $\Delta \rightarrow$ It is an output alphabet
- $\delta \rightarrow$ Transition function, $Q \times \Sigma \rightarrow Q$ (state function)
- $\lambda \rightarrow$ Output function, $Q \rightarrow \Delta$ (machine function)
- $q_o \rightarrow$ Initial state of machine

Note: The output symbol at a given time depends only on present state of moore machine.

Mealy Machine
- A Mealy machine is a FSM, where outputs are determined by current state and input.
- It associates an output symbol with each transition and the output depends on current input
- Mealy machine is defined as 6-tuple $\{Q_1, \Sigma, \delta_1, q_0, \Delta, \lambda\}$ where,
    - $Q \rightarrow$ Finite number of states
    - $\Sigma \rightarrow$ Finite set of input symbol
    - $\Delta \rightarrow$ It is an output alphabet (finite set of output symbols)
    - $\delta \rightarrow$ Transition function, $Q \times \Sigma \rightarrow Q$ (state function)
    - $\lambda \rightarrow$ Output function, $Q \rightarrow \Delta$ (machine function)
    - $q_o \rightarrow$ Initial state of machine ($q_0 \in Q$)

Note: In Moore machine, for input string of length n, the output sequence consists of (n+1) symbols
In Mealy machine, for input sequence of length n, the output sequence also consists of 'n' symbols

Equivalence of Moore and Mealy machine

(a) Mealy machine equivalent to Moore Machine
If M1 = $\{Q, \Sigma, \delta, q_0, \Delta, \lambda\}$ is a Moore machine, then there is a Mealy machine equivalent to M1.

Proof: Let M2 = $\{Q_1, \Sigma, \delta, q_0, \Delta, \lambda^1\}$ and define $\lambda^1(q, a)$ to be $\lambda(\delta(q, a))$ for all states q and input symbol 'a'. Then, M1 and M2 enter the same sequence of state on the same input, and with each transition M2 emits the o/p that M1 associates with the state entered.

Steps:
1. Always start with the initial state
Note: during the conversion of mealy to moore, the number of states may increase. If mealy machine has N states and m outputs, the number of states in the equivalent moore machine may become Nxm

(b) Moore machine equivalent to Mealy machine
The number of states in the conversion of moore machine to mealy machine will remain same.

REGULAR LANGUAGES

Introduction

The examples that we have seen; set of all binary numbers divisible by 2, 3 etc are nothing bur regular languages.

FA are ***acceptors*** of regular languages. Theyare some mathematical represntation to which if you give a string, they will accept it or not. FA (with and without output) are acceptors of regular languages. Acceptors accept “strings” from a language and tell us if the string is accepted or not.

The ***generators*** of regular languages are regular grammars. They generate the language. They are of two types (Right Linear Grammar and Left Linear Grammar).

A ***representator*** is a mathematical expression that represents a Regular Langauge. The RL are mathematically representated by expressions called Regular Expressions.

Let’s talk about RE. They are the representations of the languages that are exactly accepted by the FA. Or if we take any language which is accepted by FA, then we can represent it using Regular expressions.

Operations on Regular Expressions
1. Union (+)
2. Concatenation (●)
3. Kleene Closure (*)

Examples:

Regular Expression	Regular Language
Ø	{ }
ε	{ε}
a	{a}
a*	{ε, a, aa, aaa, ...}
a^+	a.a*/a*.a/{a, aa, aaa, ...}
(a+b)*	{ε, a, b, aa, ab, ba, bb, aaa, aab, aba, ...} (set of all strings possible over (a,b))

Points to remember:
1. If the language is finite, there is a finite automata and regular expression that accept and represent that lanaguage.

Identities of RE
1. Ø + R = R + Ø = R
2. Ø.R = R.Ø = Ø
3. ε.R = R.ε = R
4. ε* = ε
5. Ø * = ε
6. ε + RR* = R*R + ε = R*
7. (a + b)* = (a* + b*)*
   = (a*b*)*
   = (a* +b)* = (a+b*)* = a*(ba*)* = b*(ab*)*

The set of regular languages over an alphabet Σ is defined recursively as below. Any language belonging to this set is a regular language over Σ.

Definition of set of regular languages

- **Basis clause**: Ø, {ε}, {a} for any symbol a Є Σ, are regular languages.
- **Inductive clause**: If L_r and L_s are regular language, then L_r U L_s, L_r ● L_s, L_r^* are regular languages.
- **External clause**: Nothing is a regular language, unless it is obtained from above two clauses.

Regular Language: Any language represented by regular expression(s) is called a regular language.
Ex: the regular expression a^* denotes a language which has {ε, a, aa, aaa, ...}

Regular Expression

- Regular expressions are used to denote regular languages
- The set of regular expressions over an alphabet Σ is defined recursively as below. Any element of that set is a regular expression.
- **Basis clause**: Ø, ε, a are regular expressions corresponding to Ø, {ε}, {a} respectively where a is an element of Σ.
- **Inductive clause:** If r and s are regular expressins corresponding to languages L_r and L_s then (r+s), (rs) and (r^*) are regular expressions corresponding to the languages L_r U L_s, L_r ● L_s, L_r^* respectively.
- **External clause:** Nothing is a regular expression unless it is obtained from the above two clauses.

Closure property of RE: The iteration or closure of a regular expression R, written as R^* is also a regular expression.

CONVERSION OF RE TO FA: Just look at the RE given to you and bitch just make that FA.

CONVERSION OF FA TO RE: The methods used is state elimination method.
Step 1: The initial state should not have any incoming edge. If it is there, create a new initial state.
Step 2: Final state should not have outgoing edges. If it is there, create a new final state. There should be only one final state
Step 3: Eliminate the states other than initial and final states.

IS THE LANGUA GE REGULAR OR NOT?
To check if the language is regular or not.
1. If langauge is finite, it is a regular language
2. If the language is infinite, the FA should have a loop and inside the loop there should be a pattern. If there is no such pattern then no FA will be possible tpo recognize that language. This is what pumping lemma says.
If an infinite language has to be accepted by a finite automata then there should be a loop inside the finite automata. Therefore Pumping Lemma is a negative test. This means that pumping lemma says that the languge is not regular if you don’t find a pattern but it doesn’t say that the language IS regular if you find a pattern. Pumping lemma doesnt gaurantee that. Therefore, if you fail the pumping lemma, the language is NOT REGULAR but if you pass it you can’t say IT IS REGULAR.

To find if a given language is regular or not, make a RE or a FA out of the language, if it is possible to get, language is regular. In some cases you may need to find that the language does not take infinite memory.

CLOSURE PROPERTIES OF REGULAR SETS
1. **Union:** If L and M are regular languages, LUM is regular language closed under union.
2. **Concatenation and Kleene Closure**: If L and M are regular languages, L.M is regular language and L* is also regular.
3. **Intersection:** L intersection M is regular, if L and M are regular languages
4. **Difference**: L – M contains strings in L but not in M, wgere L and M are regular languages
5. **Complementation**: The complement of language L is Σ* - L.
Since Σ* is surely regular, the complement of a regular langugae is always regular. Where Σ* is a universal language.
6. **Homomorphism:** If L is a regular language, h is homomorphism on its alphabet then h(L) = {h(w) | w is in L} is also a regular langugae.

<u>GRAMMAR</u>
A grammar is generally represented by {V, T, P, S} where
V = set of all vertices
T = set of all terminals
P = set of all productions
S = start symbol

Ex. A grammar is given as:
$S \rightarrow aSB$
$S \rightarrow aB$
$B \rightarrow b$

for this V = {S, B}, T = {a, b}, P = {production formulas given above}, S = start symbol. So if this is the grammar, let's find out what is the language generated by this grammar.
Getting a string from this grammar is called *derivation.*

S => aSB (sentential/sequential form)(in every step, only one variable will be substituted by the production formula.)
If you start replacing the leftmost symbol first, that is called leftmost derivation. If you start replacing the rightmost symbol first, it is called rightmost derivation.
Let's do the leftmost derivation first.

S => aaBB (sentential/sequential form)
S => aabB(sentential/sequential form)
S => aabb (derivation)

Derivation Tree: The yield of the parse/derivation tree is gonna be the string.