Summer of Science 2025

Midterm Report

Name: Sumit

Roll No: 23B3313

Project ID: CS39(Theory of Computation)

Mentor: Arvind Yadav

1 Introduction

Finite Automata : Finite automata involve states and transitions among states in response to inputs. They are useful for building several different kinds of software including the lexical analysis component of a compiler and systems for verifying the correctness of circuits or protocols, for example.

Alphabets: An alphabet is a finite nonempty set of symbols. Conventionally, we use the symbol Σ for an alphabet Common alphabets include:

- $\Sigma = \{0, 1\}$
- $\Sigma = \{$a, b, ..., z$\}$
- The set of all ASCII characters or the set of all printable ASCII characters.

Strings: A string is a finite length sequence of symbols.

Languages and Problems: A language is a (possibly infinite) set of strings all of which choose their symbols from some one alphabet. When the strings of a language are to be interpreted in some way the question of whether a string is in the language is sometimes called a problem.

2 Finite Automata

2.1 Deterministic Finite Automaton

Definition:

A *deterministic finite automaton* consists of:

1. A finite set of *states*, often denoted Q.
2. A finite set of *input symbols*, often denoted Σ.
3. A *transition function* that takes as arguments a state and an input symbol and returns a state. The transition function will commonly be denoted δ.
4. A *start state*, one of the states in Q.
5. A set of *final* or *accepting* states F. The set F is a subset of Q.

A deterministic finite automaton will often be referred to by its acronym: **DFA**. The most succinct representation of a DFA is a listing of the five components above. In proofs we often talk about a DFA in "five-tuple" notation:

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

where A is the name of the DFA, Q is its set of states, Σ its input symbols, δ its transition function, q_0 its start state, and F its set of accepting states.

Extending the transition function to strings:

The extended transition function is a function that takes a state q and a string w and returns a state p — the state that the automaton reaches when starting in state q and processing the sequence of inputs w. We define $\hat{\delta}$ by induction on the length of the input string, as follows:

BASIS: $\hat{\delta}(q, \epsilon) = q$. That is, if we are in state q and read no inputs, then we are still in state q.

INDUCTION: Suppose w is a string of the form xa; that is, a is the last symbol of w, and x is the string consisting of all but the last symbol. For example, $w = 1101$ is broken into $x = 110$ and $a = 1$. Then

$$\hat{\delta}(q, w) = \delta(\hat{\delta}(q, x), a) \tag{2.1}$$

To compute $\hat{\delta}(q, w)$, first compute $\hat{\delta}(q, x)$, the state that the automaton is in after processing all but the last symbol of w. Suppose this state is p; that is, $\hat{\delta}(q, x) = p$. Then $\hat{\delta}(q, w)$ is what we get by making a transition from state p on input a, the last symbol of w. That is, $\hat{\delta}(q, w) = \delta(p, a)$.

The Language of a DFA

Now, we can define the *language* of a DFA $A = (Q, \Sigma, \delta, q_0, F)$. This language is denoted $L(A)$, and is defined by

$$L(A) = \{w \mid \hat{\delta}(q_0, w) \text{ is in } F\}$$

That is, the language of A is the set of strings w that take the start state q_0 to one of the accepting states. If L is $L(A)$ for some DFA A, then we say L is a *regular language.*

2.2 Nondeterministic Finite Automaton

Definition:

A *Nondeterministic finite automaton* consists of:

1. A finite set of *states*, often denoted Q.
2. A finite set of *input symbols*, often denoted Σ.
3. A *transition function* that takes as arguments a state and an input symbol and returns a subset of Q. The transition function will commonly be denoted δ.
4. A *start state*, one of the states in Q.
5. A set of *final* or *accepting* states F. The set F is a subset of Q.

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

where A is the name of the NFA, Q is its set of states, Σ its input symbols, δ its transition function, q_0 its start state, and F its set of accepting states.

Extending the transition function to strings:

We define $\hat{\delta}$ for an NFA's transition function $\hat{\delta}$ by:

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

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

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

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

The Language of an NFA

If $A = (Q, \Sigma, \delta, q_0, F)$ is an NFA, then

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

That is, $L(A)$ is the set of strings w in Σ^* such that $\hat{\delta}(q_0, w)$ contains at least one accepting state.

2.3 Equivalence of Deterministic and Nondeterministic Finite Automata

Theorem: If $D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ is the DFA constructed from NFA $N = (Q_N, \Sigma, \delta_N, q_0, F_N)$ by the subset construction, then $L(D) = L(N)$.

Proof: What we actually prove first, by induction on $|w|$, is that

$$\hat{\delta}_D(\{q_0\}, w) = \hat{\delta}_N(q_0, w)$$

Notice that each of the $\hat{\delta}$ functions returns a set of states from Q_N, but $\hat{\delta}_D$ interprets this set as one of the states of Q_D (which is the power set of Q_N), while $\hat{\delta}_N$ interprets this set as a subset of Q_N.

Basis: Let $|w| = 0$; that is, $w = \epsilon$. By the basis definitions of $\hat{\delta}$ for DFA's and NFA's, both $\hat{\delta}_D(\{q_0\}, \epsilon)$ and $\hat{\delta}_N(q_0, \epsilon)$ are $\{q_0\}$.

Induction: Let w be of length $n + 1$, and assume the statement for length n. Break w up as $w = xa$, where a is the final symbol of w and x is the rest of w. By the inductive hypothesis, $\hat{\delta}_D(\{q_0\}, x) = \hat{\delta}_N(q_0, x)$. Let both these sets of N's states be $\{p_1, p_2, \ldots, p_k\}$.

The inductive part of the definition of $\hat{\delta}$ for NFA's tells us

$$\hat{\delta}_N(q_0, w) = \bigcup_{i=1}^{k} \delta_N(p_i, a) \tag{2.2}$$

The subset construction, on the other hand, tells us that

$$\delta_D(\{p_1, p_2, \ldots, p_k\}, a) = \bigcup_{i=1}^{k} \delta_N(p_i, a) \tag{2.3}$$

Now, let us use (2.3) and the fact that $\hat{\delta}_D(\{q_0\}, x) = \{p_1, p_2, \ldots, p_k\}$ in the inductive part of the definition of $\hat{\delta}$ for DFA's:

$$\hat{\delta}_D(\{q_0\}, w) = \delta_D(\hat{\delta}_D(\{q_0\}, x), a) = \delta_D(\{p_1, p_2, \ldots, p_k\}, a) = \bigcup_{i=1}^{k} \delta_N(p_i, a) \tag{2.4}$$

Thus, Equations (2.2) and (2.4) demonstrate that $\hat{\delta}_D(\{q_0\}, w) = \hat{\delta}_N(q_0, w)$.

When we observe that D and N both accept w if and only if $\hat{\delta}_D(\{q_0\}, w)$ or $\hat{\delta}_N(q_0, w)$, respectively, contain a state in F_N, we have a complete proof that $L(D) = L(N)$. $\square$

Theorem: A language L is accepted by some DFA if and only if L is accepted by some NFA.

Proof: (If) The "if" part is the subset construction and above theorem.

(Only-if) This part is easy; we have only to convert a DFA into an identical NFA. Put intuitively, if we have the transition diagram for a DFA, we can also interpret it as the transition diagram of an NFA, which happens to have exactly one choice of transition in any situation. More formally, let $D = (Q, \Sigma, \delta_D, q_0, F)$ be a DFA. Define $N = (Q, \Sigma, \delta_N, q_0, F)$ to be the equivalent NFA, where δ_N is defined by the rule:

- If $\delta_D(q, a) = p$, then $\delta_N(q, a) = \{p\}$.

It is then easy to show by induction on $|w|$, that if $\hat{\delta}_D(q_0, w) = p$, then

$$\hat{\delta}_N(q_0, w) = \{p\}$$

3 Regular Expressions and Languages

3.1 Regular Expressions

A **regular expression** over an alphabet Σ defines a regular language and is built using the following base cases and operations:

Base Cases:

- $\emptyset$ denotes the empty language.
- ϵ denotes the language $\{\epsilon\}$.
- For any $a \in \Sigma$, a denotes the language $\{a\}$.

Operations:

- $r + s$ is the union of languages described by r and s.
- rs is the concatenation of languages.
- r^* is the Kleene star of the language of r.

Example: The regular expression $(a + b)^* abb$ denotes the set of strings over $\{a, b\}$ ending with abb.

3.2 Equivalence of Regular Expressions and Finite Automata

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

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

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

The proof is by structural induction on R, following the recursive definition of regular expressions.

3.3 Finite Automata and Regular Expressions

To convert a finite automaton into a regular expression:

1. Add a new start state q_s and new final state q_f.
2. Add ϵ-transitions from q_s to the old start state and from all old final states to q_f.
3. Use state elimination: remove intermediate states one by one, updating the regular expressions labeling edges.

Example: DFA accepting strings over $\{a, b\}$ that end in ab can be converted into RE $(a+b)^*ab$.

4 Properties of Regular Languages

4.1 The pumping lemma for regular languages

Theorem: Let L be a regular language. Then there exists a constant n (which depends on L) such that for every string $w \in L$ such that $|w| \geq n$, we can break w into three strings, $w = xyz$, such that:

1. $y \neq \epsilon$.
2. $|xy| \leq n$.
3. For all $k \geq 0$, the string xy^kz is also in L.

That is, we can always find a nonempty string y not too far from the beginning of w that can be "pumped"; that is, repeating y any number of times, or deleting it (the case $k = 0$), keeps the resulting string in the language L.

PROOF: Suppose L is regular. Then $L = L(A)$ for some DFA A. Suppose A has n states. Now, consider any string w of length n or more, say $w = a_1a_2 \cdots a_m$, where $m \geq n$ and each a_i is an input symbol. For $i = 0, 1, \ldots, n$, define state p_i to be $\delta(q_0, a_1a_2 \cdots a_i)$, where δ is the transition function of A, and q_0 is the start state of A. That is, p_i is the state A is in after reading the first i symbols of w. Note that $p_0 = q_0$.

By the pigeonhole principle, it is not possible for the $n+1$ different p_i's for $i = 0, 1, \ldots, n$ to be distinct, since there are only n different states. Thus, we can find two different integers i and j, with $0 \le i < j \le n$, such that $p_i = p_j$. Now, we can break $w = xyz$ as follows:

1. $x = a_1 a_2 \cdots a_i$.
2. $y = a_{i+1} a_{i+2} \cdots a_j$.
3. $z = a_{j+1} a_{j+2} \cdots a_m$.

That is, x takes us to p_i once; y takes us from p_i back to p_i (since p_i is also p_j), and z is the balance of w. The relationships among the strings and states are suggested by Fig. 4.1. Note that x may be empty, in the case that $i = 0$. Also, z may be empty if $j = n = m$. However, y can not be empty, since i is strictly less than j.

$$\begin{tikzcd}
p_0 \arrow[r, bend left, "x = a_1 \cdots a_i"] & p_i \arrow[loop above, "y = a_{i+1} \cdots a_j"] \arrow[r, bend left, "z = a_{j+1} \cdots a_m"] & p_j
\end{tikzcd}$$

Figure 4.1: Every string longer than the number of states must cause a state to repeat

Now, consider what happens if the automaton A receives the input xy^kz for any $k \ge 0$. If $k = 0$, then the automaton goes from the start state q_0 (which is also p_0) to p_i on input x. Since p_i is also p_j, it must be that A goes from p_i to the accepting state shown in Fig. 4.1 on input z. Thus, A accepts xz.

If $k > 0$, then A goes from q_0 to p_i on input x, circles from p_i to p_i k times on input y^k, and then goes to the accepting state on input z. Thus, for any $k \ge 0$, xy^kz is also accepted by A; that is, xy^kz is in L. $\square$

4.2 Closure Properties of Regular Languages

1. The union of two regular languages is regular.
2. The intersection of two regular languages is regular.
3. The complement of a regular language is regular.
4. The difference of two regular languages is regular.
5. The reversal of a regular language is regular.
6. The closure(star) of a regular language is regular.
7. The concatenation of regular languages is regular.
8. A homomorphism (substitution of strings for symbols) of a regular language is regular.
9. The inverse homomorphism of a regular language is regular.

Theorem: If L and M are regular languages, then so is $L \cup M$.

PROOF: This proof is simple. Since L and M are regular, they have regular expressions; say $L = L(R)$ and $M = L(S)$. Then $L \cup M = L(R + S)$ by the definition of the $+$ operator for regular expressions. $\square$

Homomorphisms

A string *homomorphism* is a function on strings that works by substituting a particular string for each symbol.

Example: The function h defined by $h(0) = ab$ and $h(1) = \epsilon$ is a homomorphism. Given any string of 0's and 1's, it replaces all 0's by the string ab and replaces all 1's by the empty string. For example, h applied to the string 0011 is $abab$. □

Formally, if h is a homomorphism on alphabet Σ, and $w = a_1a_2 \cdots a_n$ is a string of symbols in Σ, then

$$h(w) = h(a_1)h(a_2) \cdots h(a_n).$$

That is, we apply h to each symbol of w and concatenate the results, in order.

Theorem: If L is a regular language over alphabet Σ, and h is a homomorphism on Σ, then $h(L)$ is also regular.

PROOF: Let $L = L(R)$ for some regular expression R. In general, if E is a regular expression with symbols in Σ, let $h(E)$ be the expression we obtain by replacing each symbol a of Σ in E by $h(a)$. We claim that $h(E)$ defines the language $h(L(E))$.

The proof is an easy structural induction that says whenever we take a subexpression E of R and apply h to it to get $h(E)$, the language of $h(E)$ is the same language we get if we apply h to the language $L(E)$. Formally,

$$L(h(E)) = h(L(E)).$$

BASIS: If E is ϵ or $\emptyset$, then $h(E)$ is the same as E, since h does not affect the string ϵ or the language $\emptyset$. Thus, $L(h(E)) = L(E)$. However, if E is 0 or 1, then $L(E)$ contains either no strings or a string with no symbols, respectively. Thus $h(L(E)) = L(E)$ in either case. We conclude $L(h(E)) = L(E) = h(L(E))$.

The only other basis case is if $E = a$ for some symbol $a \in \Sigma$. In this case, $L(E) = \{a\}$, so $L(h(E)) = \{h(a)\}$. Also, $h(E)$ is the regular expression that is the string of symbols $h(a)$. Thus, $L(h(E)) = \{h(a)\}$, and we conclude

$$L(h(E)) = h(L(E)).$$

INDUCTION: There are three cases, each of which is simple. We shall prove only the union case, where $E = F + G$. The way we apply homomorphisms to regular expressions assures us that

$$h(E) = h(F + G) = h(F) + h(G).$$

We also know that $L(E) = L(F) \cup L(G)$, and

$$L(h(E)) = L(h(F) + h(G)) = L(h(F)) \cup L(h(G)) \tag{4.1}$$

by the definition of what "+" means in regular expressions. Finally,

$$h(L(E)) = h(L(F) \cup L(G)) = h(L(F)) \cup h(L(G)) \tag{4.2}$$

because h is applied to a language by application to each of its strings individually. Now we may invoke the inductive hypothesis to assert that $L(h(F)) = h(L(F))$ and $L(h(G)) = h(L(G))$. Thus, the final expressions in (4.1) and (4.2) are equivalent, and therefore so are their respective first terms; that is,

$$L(h(E)) = h(L(E)).$$

We shall not prove the cases where expression E is a concatenation or closure; the ideas are similar to the above in both cases. The conclusion is that $L(h(R))$ is indeed $h(L(R))$; i.e., applying the homomorphism h to the regular expression for language L results in a regular expression that defines the language $h(L)$. $\square$

4.3 Minimizing Deterministic Finite Automata:

We can partition the states of any DFA into groups of mutually indistinguishable states. Members of two different groups are always distinguishable. If we replace each group by a single state we get an equivalent DFA that has as few states as any DFA for the same language.

5 Context-Free Grammars and Languages

5.1 Introduction

Context-free grammars (CFGs) are formal systems that generate context-free languages. They play a crucial role in describing the syntax of programming languages, natural languages, and various computation models.

5.2 Definition of CFG

A CFG is a 4-tuple:

$$G = (V, T, P, S)$$

where:

- V is a finite set of variables (non-terminal symbols)
- T is a finite set of terminal symbols, $T \cap V = \emptyset$
- P is a finite set of productions of the form $A \to \alpha$ where $A \in V$, $\alpha \in (V \cup T)^*$
- $S \in V$ is the start symbol

5.3 Derivations using a Grammar

A derivation in a CFG is a sequence of rule applications starting from S.

Leftmost Derivation: In each step, the leftmost non-terminal is replaced.

Rightmost Derivation: In each step, the rightmost non-terminal is replaced.

5.4 Language of a CFG

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

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

If a language L is the language of some context-free grammar, then L is said to be a *context-free language*, or CFL.

5.5 Constructing Parse Trees

Let us fix on a grammar $G = (V, T, P, S)$. The *parse trees* for G are trees with the following conditions:

1. Each interior node is labeled by a variable in V.
2. Each leaf is labeled by either a variable, a terminal, or ϵ. However, if the leaf is labeled ϵ, then it must be the only child of its parent.
3. If an interior node is labeled A, and its children are labeled

$$X_1, X_2, \ldots, X_k$$

   respectively, from the left, then $A \to X_1 X_2 \cdots X_k$ is a production in P. Note that the only time one of the X's can be ϵ is if that is the label of the only child, and $A \to \epsilon$ is a production of G.

6 Pushdown Automata

6.1 Introduction to Pushdown Automata

A Pushdown automata (PDA) is a nondeterministic finite automata coupled with a stack that can be used to store a string of arbitrary length. The stack can be read and modified only at its top.

A PDA is formally a 7-tuple:

$$M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$$

where:

- Q is a finite set of states
- Σ is the input alphabet
- Γ is the stack alphabet
- $\delta : Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \to 2^{Q \times \Gamma^*}$ is the transition function
- $q_0 \in Q$ is the start state
- $Z_0 \in \Gamma$ is the initial stack symbol
- $F \subseteq Q$ is the set of accepting states

6.2 Instantaneous Descriptions of a PDA

We shall represent the configuration of a PDA by a triple (q, ω, γ) where:

- q is the state
- ω is the remaining input
- γ is the stack contents.

Let $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ be a PDA. Define $\vdash_P$, or just $\vdash$ when P is understood, as follows. Suppose $\delta(q, a, X)$ contains (p, α). Then for all strings w in Σ^* and β in Γ^*:

$$(q, aw, X\beta) \vdash (p, w, \alpha\beta)$$

This move reflects the idea that, by consuming a (which may be ε) from the input and replacing X on top of the stack by α, we can go from state q to state p. Note that what remains on the input, w, and what is below the top of the stack, β, do not influence the action of the PDA; they are merely carried along, perhaps to influence events later.

We also use the symbol $\vdash_P^*$, or $\vdash^*$ when the PDA P is understood, to represent zero or more moves of the PDA. That is:

Basis: $I \vdash^* I$ for any ID I.

Induction: $I \vdash^* J$ if there exists some ID K such that $I \vdash K$ and $K \vdash^* J$.

That is, $I \vdash^* J$ if there is a sequence of ID's $K_1, K_2, \ldots, K_n$ such that $I = K_1$, $J = K_n$, and for all $i = 1, 2, \ldots, n-1$, we have $K_i \vdash K_{i+1}$.

6.3 Language Acceptance by PDA

There are two modes of acceptance:

- **Acceptance by final state:** Input is accepted if PDA enters a final state after consuming the input.
- **Acceptance by empty stack:** Input is accepted if stack becomes empty after consuming input.

Theorem: The class of languages accepted by PDAs by final state is the same as that accepted by empty stack.

6.4 Example: PDA for $\{a^n b^n : n \geq 0\}$

- Push a symbols to the stack
- For each b, pop an a
- Accept if stack is empty

Transitions:

$$\delta(q_0, a, Z_0) = \{(q_0, aZ_0)\}, \quad \delta(q_0, a, a) = \{(q_0, aa)\}$$

$$\delta(q_0, b, a) = \{(q_1, \epsilon)\}, \quad \delta(q_1, b, a) = \{(q_1, \epsilon)\}, \quad \delta(q_1, \epsilon, Z_0) = \{(q_f, Z_0)\}$$

6.5 Deterministic Pushdown Automata:

We define a PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ to be *deterministic* (a deterministic PDA or DPDA), if and only if the following conditions are met:

1. $\delta(q, a, X)$ has at most one member for any q in Q, a in Σ or $a = \epsilon$, and X in Γ.
2. If $\delta(q, a, X)$ is nonempty, for some a in Σ, then $\delta(q, \epsilon, X)$ must be empty.

Acceptance by Deterministic Pushdown Automata: The two modes of acceptance—final state and empty stack—are not the same for DPDA's. Rather, the languages accepted by empty stack are exactly those of the languages accepted by final state that have the prefix property: no string in the language is a prefix of another word in the language.

The Languages Accepted by DPDA's: All the regular languages are accepted (by final state) by DPDA's, and there are nonregular languages accepted by DPDA's. The DPDA languages are context-free languages, and in fact are languages that have unambiguous CFG's. Thus, the DPDA languages lie strictly between the regular languages and the context-free languages.

6.6 Non-Determinism in PDA

Unlike DFAs, nondeterminism increases the power of PDAs. There are context-free languages that cannot be accepted by deterministic PDAs.

Example: $L = \{a^n b^n c^n : n \geq 1\}$ is not context-free. No PDA can use a single stack to match three different symbol sequences.

6.7 Theorem: Equivalence of PDA and CFG

Theorem: A language is context-free if and only if there exists a PDA that accepts it.

Proof. We prove this in two directions:

(1) Every Context-Free Grammar (CFG) has an Equivalent PDA

Let $G = (V, \Sigma, R, S)$ be a context-free grammar. We construct a PDA P such that $N(P) = L(G)$.

Construction:

We define a PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ as follows:

- $Q = \{q\}$ (a single state),
- $\Gamma = V \cup \Sigma$,
- $q_0 = q$, $Z_0 = S$, $F = \emptyset$ (we use acceptance by empty stack),
- The transition function δ includes:
  - For every production $A \to \alpha$ in R:

$$\delta(q, \varepsilon, A) \ni (q, \alpha)$$

– For every terminal $a \in \Sigma$:

$$\delta(q, a, a) \ni (q, \varepsilon)$$

We shall prove that w is in $N(P)$ if and only if w is in $L(G)$.

(If)

Suppose $w \in L(G)$. Then w has a leftmost derivation

$$S = \gamma_1 \Rightarrow_{lm} \gamma_2 \Rightarrow_{lm} \cdots \Rightarrow_{lm} \gamma_n = w$$

We show by induction on i that $(q, w, S) \vdash_P^* (q, y_i, \alpha_i)$, where y_i and α_i are a representation of the left-sentential form γ_i. That is, let α_i be the tail of γ_i, and let $\gamma_i = x_i\alpha_i$. Then y_i is the string such that $x_iy_i = w$, i.e., it is what remains when x_i is removed from the input.

Basis:

For $i = 1$, $\gamma_1 = S$. Thus $x_1 = \varepsilon$, and $y_1 = w$. Since $(q, w, S) \vdash_P^* (q, w, S)$ by 0 moves, the basis is proved.

Induction:

Now we consider the case of the second and subsequent left-sentential forms. We assume:

$$(q, w, S) \vdash_P^* (q, y_i, \alpha_i)$$

and prove

$$(q, w, S) \vdash_P^* (q, y_{i+1}, \alpha_{i+1})$$

Since α_i is a tail, it begins with a variable A. Moreover, the step of the derivation $\gamma_i \Rightarrow \gamma_{i+1}$ involves replacing A by one of its production bodies, say β. Rule (1) of the construction of P lets us replace A at the top of the stack by β, and rule (2) then allows us to match any terminals on top of the stack with the next input symbols. As a result, we reach the ID:

$$(q, y_{i+1}, \alpha_{i+1})$$

which represents the next left-sentential form γ_{i+1}.

To complete the proof, we note that $\alpha_n = \varepsilon$, since the tail of γ_n (which is w) is empty. Thus, $(q, w, S) \vdash_P^* (q, \varepsilon, \varepsilon)$, which proves that P accepts w by empty stack.

(Only-if)

We need to prove something more general: that if P executes a sequence of moves that has the net effect of popping a variable A from the top of its stack, without ever going below A on the stack, then A derives, in G, whatever input string was consumed from the input during this process. Precisely:

- If $(q, x, A) \vdash_P^* (q, \varepsilon, \varepsilon)$, then $A \Rightarrow_G^* x$.

The proof is an induction on the number of moves taken by P.

Basis:

One move. The only possibility is that $A \to \varepsilon$ is a production of G, and this production is used in a rule of type (1) by the PDA P. In this case, $x = \varepsilon$, and we know that $A \Rightarrow \varepsilon$.

Induction:

Suppose P takes n moves, where $n > 1$. The first move must be of type (1), where A is replaced by one of its production bodies on the top of the stack. The reason is that a rule of type (2) can only be used when there is a terminal on top of the stack. Suppose the production used is $A \to Y_1Y_2 \cdots Y_k$, where each Y_i is either a terminal or a variable.

The next $n-1$ moves of P must consume x from the input and have the net effect of popping each of $Y_1, Y_2, \ldots$ from the stack, one at a time. We can break x into $x_1x_2 \cdots x_k$, where x_1 is the portion of the input consumed until Y_1 is popped off the stack (i.e., the stack first is as short as $k-1$ symbols). Then x_2 is the next portion of the input that is consumed while popping Y_2 off the stack, and so on.

Formally, we can conclude that:

$$(q, x_ix_{i+1} \cdots x_k, Y_i) \vdash_P^* (q, x_{i+1} \cdots x_k, \varepsilon)$$

for all $i = 1, 2, \ldots, k$. Moreover, none of these sequences can be more than $n-1$ moves, so the inductive hypothesis applies if Y_i is a variable. That is, we may conclude $Y_i \Rightarrow^* x_i$.

If Y_i is a terminal, then there must be only one move involved, and it matches the one symbol of x_i against Y_i, which are the same. Again, we can conclude $Y_i \Rightarrow x_i$; this time, zero steps are used.

Now we have the derivation:

$$A \Rightarrow Y_1Y_2 \cdots Y_k \Rightarrow x_1Y_2 \cdots Y_k \Rightarrow \cdots \Rightarrow x_1x_2 \cdots x_k$$

That is, $A \Rightarrow^* x$.

To complete the proof, we let $A = S$ and $x = w$. Since we are given that w is in $N(P)$, we know that $(q, w, S) \vdash_P^* (q, \varepsilon, \varepsilon)$. By what we have just proved inductively, we have $S \Rightarrow^* w$; i.e., $w \in L(G)$.

(2) Every PDA has an Equivalent CFG

Let $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ be a PDA. We construct a CFG G such that $L(G) = N(P)$.

Assumption:

Assume P accepts by empty stack. If P accepts by final state, we can convert it into an equivalent PDA that accepts by empty stack.

Construction:

We define a CFG $G = (V, \Sigma, R, S)$ where:

- V contains variables of the form $[qAr]$ for all $q, r \in Q$ and $A \in \Gamma$,
- $S = [q_0Z_0q_f]$ for some $q_f \in Q$,

- Rules are defined to simulate the behavior of P from q to r with A on the stack.

The main idea is to generate rules of the form:

$$[qAr] \to a[q_1 B q_2][q_2 C r]$$

corresponding to transitions like:

$$\delta(q, a, A) \ni (q_1, BC)$$

Also include:

$$[qAr] \to a$$

if a transition consumes input a and pops A.

Intuition:

The grammar G generates exactly those strings that the PDA accepts by emptying its stack, mimicking its stack behavior through variables. Proof follows through induction which I would not be able to show here .

□

7 Properties of Context-Free Languages

7.1 Normal Forms for Context-Free Grammars

To simplify CFGs and enable algorithms like parsing and equivalence checking, we often convert CFGs into standard formats called **normal forms**.

7.1.1 Eliminating Useless Symbols:

A variable can be eliminated from a CFG unless it derives some string of terminals and also appears in at least one string derived from the start symbol. To correctly eliminate such useless symbols we must test whether a variable derives a terminal string and eliminate those that do not, along with all their productions. Only then do we eliminate variables that are not derivable from the start symbol.

7.1.2 Eliminating ϵ and Unit productions:

Given a CFG, we can find another CFG that generates the same language except for string ϵ, yet has no ϵ productions (those with body ϵ) or unit productions (those with a single variable as the body).

7.1.3 Chomsky Normal Form (CNF)

A CFG is in Chomsky Normal Form if every production is of the form:

- $A \to BC$ where B, C are non-terminals (not start symbol), or
- $A \to a$ where a is a terminal, or

- $S \to \epsilon$ if $\epsilon \in L(G)$

Theorem: Every context-free language (excluding $\{\epsilon\}$) can be generated by a CFG in CNF.

Sketch. By eliminating null productions, unit productions, and useless symbols, and then converting all rules to the required binary or terminal form. □

7.1.4 Greibach Normal Form (GNF)

A CFG is in GNF if all production rules are of the form:

$$A \to a\alpha$$

where a is a terminal and α is a (possibly empty) string of non-terminals.

Theorem: Every context-free language can be generated by a grammar in GNF.

- GNF is useful for proving theorems related to parsing and pushdown automata.

7.1.5 Comparison of CNF and GNF

- CNF is widely used in parsing (e.g., CYK algorithm)
- GNF ensures every derivation step introduces a terminal

Example: Convert the CFG

$$S \to aSb \mid \epsilon$$

into CNF.

Step 1: Eliminate ϵ-productions: $S \to aSb \mid ab$

Step 2: Introduce new variables for terminals: $A \to a$, $B \to b$

Final: $S \to ASB \mid AB$

7.2 The Pumping Lemma for CFL

Theorem: Suppose we have a parse tree according to a Chomsky-Normal-Form grammar $G = (V, T, P, S)$, and suppose that the yield of the tree is a terminal string w. If the length of the longest path is n, then $|w| \leq 2^{n-1}$.

PROOF: The proof is a simple induction on n.

BASIS: $n = 1$. Recall that the length of a path in a tree is the number of edges, i.e., one less than the number of nodes. Thus, a tree with a maximum path length of 1 consists of only a root and one leaf labeled by a terminal. String w is this terminal, so $|w| = 1$. Since $2^{n-1} = 2^0 = 1$ in this case, we have proved the basis.

INDUCTION: Suppose the longest path has length n, and $n > 1$. The root of the tree uses a production, which must be of the form $A \to BC$, since $n > 1$; i.e., we could not start the tree using a production with a terminal. No path in the subtrees rooted at B and C can have length greater than $n - 1$, since these paths exclude the edge from the root to its child labeled B or C. Thus, by the inductive hypothesis, these two subtrees each have yields of length at most 2^{n-2}. The yield of the entire tree is the concatenation of these two yields, and therefore has length at most

$2^{n-2} + 2^{n-2} = 2^{n-1}$. Thus, the inductive step is proved. $\square$

The Pumping Lemma for CFL: Let L be a CFL. Then there exists a constant n such that if z is any string in L such that $|z|$ is at least n, then we can write $z = uvwxy$, subject to the following conditions:

1. $|vwx| \leq n$. That is, the middle portion is not too long.
2. $vx \neq \epsilon$. Since v and x are the pieces to be "pumped," this condition says that at least one of the strings we pump must not be empty.
3. For all $i \geq 0$, $uv^iwx^iy \in L$. That is, the two strings v and x may be "pumped" any number of times, including 0, and the resulting string will still be a member of L.

8 Future Plan of Action

Week 5: Turing Machines

- Reference: Hopcroft's book
- Chapter 8: Turing Machines
- Exercises: Designing simple TMs, simulation exercises

Week 6: Lectures

- Go through CS760 IITB Course, lectures 1 to 10
- Reference: Hopcroft's book
- Chapter 10: Intractable Problems
- Exercise: Approximation Algorithms for TSP

Week 7: Research Paper Exploration

- Paper: *On the Satisfiability of Context-free String Constraints with Subword-Ordering*
- If time permits: Skim follow-up paper *Satisfiability of Context-free String Constraints with Subword-Ordering and Transducers*

Week 8: Project

- Complete project and end-term report.

9 Reference:

Book *Introduction to Automata Theory, Languages, and Computation* by Hopcroft et al., and additional research materials.