CS3231 Theory of Computation notes

Aloysius

August 2025

Contents

1 Preliminaries

Definition 1.1 (Alphabet)**.** An alphabet is a **finite**, non-empty set of symbols, usually denoted by the symbol Σ. e.g. $\{0,1\}, \{A, B, \ldots, Z\}, \{0, A, s\}$

Definition 1.2 (String)**.** A string is a finite sequence of symbols from an Alphabet, e.g. 001011, ABZSFAJF, 0AAAss. The empty string is denoted as ϵ

Definition 1.3 (Language)**.** A Language is a set of Strings over an alphabet.

Definition 1.4 (Powers on an Alphabet)**.** The nth power of an alphabet, Σ^n, is the set of all strings of length n over the alphabet. Note that Σ^* represents all strings of finite length over the alphabet.

Definition 1.5 (Concatenation of Strings)**.** String concatenation operator is $\cdot$, but it can be dropped.

Definition 1.6 (Operators on Languages)**.**

1. $L_1 \cdot L_2 = L_1 L_2 = \{xy : x \in L_1, y \in L_2\}$
2. $L^* = \{x_1 x_2 \ldots x_n : x_1, x_2, \ldots, x_n \in L, n \in \mathbb{N}\} = \epsilon \cup L \cup LL \cup LLL \cup \ldots$
3. $L^+ = \{x_1 x_2 \ldots x_n : x_1, x_2, \ldots, x_n \in L, n \geq 1\} = L \cup LL \cup LLL \cup \ldots$

2 DFA, NFA, Regular expressions

2.1 DFA

A Deterministic Finite automata is defined by 5 things

1. A finite set of states, denoted as Q
2. A finite set of symbols from an alphabet Σ
3. A transition function, that takes a state from Q and a symbol from Σ, and returns a state from Q, denoted as δ
4. A starting state, q_0
5. A set of final/accepting states, denoted as F

For example, a DFA can be constructed to accept any string containing an odd number of bs

- $\delta(q_0, a) = q_0$
- $\delta(q_0, b) = q_1$

- $\delta(q_1, a) = q_1$
- $\delta(q_1, b) = q_0$

$\hat{\delta}$ is a function that takes in an initial state and a string, and accepts it if the DFA accepts the string.

$$\begin{aligned}&\text{Base case:}\\&\hat{\delta}(q, \epsilon) = q\\&\text{Inductive case:}\\&\hat{\delta}(q, wa) = \delta(\hat{\delta}(q, w), a)\end{aligned}$$

Definition 2.1 (Dead state)**.** A dead state is a state from which a final state can not be reached.

Definition 2.2 (Unreachable state)**.** An unreachable state is a state q for which $\forall w \in \Sigma^*, \hat{\delta}(q_0, w) \neq q$

2.2 NFA

A Non Deterministic Finite Automata is almost like a DFA, except it's transition function maps to a subset of states in Q. In practice, this means a state in a NFA diagram can have multiple arrows with the same symbol attached to it. This allows for more compact diagrams.

2.3 Equivalence of DFA and NFA

Given any NFA, we can treat every possible subset of Q as a state in the DFA, such that $Q_D = \{S | S \subset Q\}$. The set of final states is the set of subsets of Q where there is at least one final state, $F_D = \{S | S \subset Q \wedge S \cap F \neq \emptyset\}$. We can then define a transition function for the DFA as:

$$\delta_D(S, a) = \cup_{q \in S} \delta_N(q, a).$$

Since an NFA can handle multiple states at once, this definition of a transition function basically simulates an NFA by finding all the possible states the NFA can be in. This is known as the Powerset construction.

Powerset construction. Claim: For any string w, $\hat{\delta_D}(\{q_0\}, w) = \hat{\delta_N}(q_0, w)$

$$\begin{aligned}&\text{Base case:}\\&\delta_D(\{q_0\}, \epsilon) = \{q_0\} = \delta_N(q_0, \epsilon)\\&\text{Inductive case:}\\\hat{\delta_D}(\{q_0\}, wa) &= \delta_D(\hat{\delta_D}(\{q_0\}, w), a)\\&= \delta_D(\hat{\delta_N}(q_0, w), a)\\&= \cup_{q \in \hat{\delta_N}(q_0, w)} \delta(q, a)\\&= \hat{\delta_N}(q_0, wa)\end{aligned}$$

□

Hence any NFA can be converted into a DFA, and it follows that an NFA can not be constructed to accept a Language that a DFA can not accept.

2.4 ϵ-NFA

An ϵ-NFA is a NFA with a transition function that accepts empty strings, essentially it allows some states to have optional transitions.

Definition 2.3 (Empty closure)**.** The empty closure of a state in an ϵ-NFA is the set of states reachable by taking any amount of ϵ transitions

1. $q \in Eclose(q)$
2. $p \in Eclose(q) \implies \forall k \in \delta(p, \epsilon), k \in Eclose(q)$
3. Iterate step 2 until not more ϵ transitions can be taken

The $\hat{\delta}$ of a ϵ-NFA is defined as follows:

Base case:

$\hat{\delta}(q, \epsilon) = Eclose(q)$

Inductive case:

$\hat{\delta}(q, wa)$

First, find all the set of all possible ending states:

$R = \cup_{p \in \hat{\delta}(q,w)} \delta(p, a)$

Then, the string transition function is just the empty closure of the ending states

$\hat{\delta}(q, wa) = \cup_{p \in R} Eclose(p)$

Naturally, ϵ-NFAs are equivalent to NFAs, which mean they are equivalent to DFAs as well. To convert an ϵ-NFA to a NFA, $\delta_N(q, a) = \cup_{p \in Eclose(q)} Eclose(\delta_\epsilon(p, a))$.

2.5 Regular expressions

A regular expression is equivalent to a DFA, given an alphabet Σ, $L(r)$ gives the set of strings accepted by the regular expression. The grammar for a regular expression is as follows:

1. ϵ, where $L(\epsilon) = \{\epsilon\}$
2. $\emptyset$, where $L(\emptyset) = \emptyset$
3. a, where $a \in \Sigma$ and $L(a) = \{a\}$
4. $r_1 + r_2$, given two regular expressions, their union is represented by $+$. $L(r_1 + r_2) = L(r_1) \cup L(r_2)$
5. $r_1 \cdot r_2$, given two regular expressions, their concatenation is represented by $\cdot$. $L(r_1 \cdot r_2) = \{xy | x \in L(r_1), y \in L(r_2)\}$
6. r^*, given a regular expression, its kleene star is represented by $*$. $L(r^*) = \{x_1 x_2 \dots x_k | \forall 1 \leq i \leq k, x_i \in L(r)\}$. Note that k can be 0, so $\emptyset \in L(r^*)$
7. (r), a regular expression wrapped around a paranthesis

Note that the precedence rule is $* > \cdot > +$

2.5.1 DFA to Regular Expression

A DFA can be converted to a regular expression very easily. Given the DFA $A = (Q, \Sigma, \delta, q_0, F)$, and assume that $Q = \{1, 2, \dots n\}$ and $q_0 = 1$. Let $R^k_{i,j}$ be the regular expression that accepts the set of strings that moves the DFA from the state i to the state j, only using intermediary states numbered $\leq k$.
Base case, $k = 0$.

$$R^0_{i,j} = \begin{cases} \sum_{\delta(i,a_i)=j} a_i, & \text{if } i \neq j \\ \epsilon + \sum_{\delta(i,a_i)=i} a_i, & \text{if } i = j \end{cases}.$$

Induction case

$$R^{k+1}_{i,j} = R^k_{i,j} + R^k_{i,k+1}(R^k_{k+1,k+1})^* R^k_{k+1,j}.$$

The construction is simple, you take the regular expression for $R^k_{i,j}$, then union it with the regular expression that takes i to $k+1$ ($R^k_{i,k+1}$), concatenated with any number of useless transitions ($(R^k_{k+1,k+1})^*$), concatenated with the regular expression that takes $k+1$ to j ($R^k_{k+1,j}$).
Hence a regular expression equivalent to the DFA would be

$$\sum_{j \in F} R^n_{1,j}.$$

2.5.2 Regular Expression to ϵ-NFA

A regular expression can be converted to an ϵ-NFA by applying some simple recursive transformations, which leads to an ϵ-NFA with the following properties:

1. It has only one final state
2. There is no transition into the final state
3. There is no transition out of the final state
4. The starting and final states are different

The transformation rules are as follows, note that if a state does not have a transition, it goes to a dead state.

1. $\emptyset$: $A = (\{q_0, q_f\}, \Sigma, \delta, q_0, \{q_f\})$, where q_0 is a dead state.
2. ϵ: $A = (\{q_0, q_f\}, \Sigma, \delta, q_0, \{q_f\})$, where $\delta(q_0, \epsilon) = q_f$ is the only transition.
3. a: $A = (\{q_0, q_f\}, \Sigma, \delta, q_0, \{q_f\})$, where $\delta(q_0, a) = q_f$ is the only transition.
4. $r_1 + r_2$: Let $A_1 = (Q_1, \Sigma, \delta_1, q_0^1, q_f^1)$ and $A_2 = (Q_2, \Sigma, \delta_2, q_0^2, q_f^2)$ represent the ϵ-NFA for r_1 and r_2 respectively. Our new ϵ-NFA combines these two, where $A = (\{q_0, q_f\} \cup Q_1 \cup Q_2, \Sigma, \delta, q_0, \{q_f\})$ and
   - $\delta(q_0, \epsilon) = \{q_0^1, q_0^2\}$
   - $\delta(q_f^1, \epsilon) = q_f$
   - $\delta(q_f^2, \epsilon) = q_f$

   Additionally, δ contains all the transitions of δ_1 and δ_2
5. $r_1 \cdot r_2$: Let $A_1 = (Q_1, \Sigma, \delta_1, q_0^1, q_f^1)$ and $A_2 = (Q_2, \Sigma, \delta_2, q_0^2, q_f^2)$ represent the ϵ-NFA for r_1 and r_2 respectively. Our new ϵ-NFA combines these two, where $A = (\{q_0, q_f\} \cup Q_1 \cup Q_2, \Sigma, \delta, q_0, \{q_f\})$ and
   - $\delta(q_0, \epsilon) = \{q_0^1\}$
   - $\delta(q_f^1, \epsilon) = \{q_0^2\}$
   - $\delta(q_f^2, \epsilon) = \{q_f\}$

   Additionally, δ contains all the transitions of δ_1 and δ_2
6. r^*: Let $A_1 = (Q_1, \Sigma, \delta_1, q_0^1, q_f^1)$ represent the ϵ-NFA for r. Our new ϵ-NFA applies the kleene star rule, where $A = (\{q_0, q_f\} \cup Q_1, \Sigma, \delta, q_0, \{q_f\})$ and
   - $\delta(q_0, \epsilon) = \{q_0^1, q_f\}$
   - $\delta(q_f, \epsilon) = \{q_0^1, q_f\}$

   Additionally, δ contains all the transitions of δ_1

2.6 Minimization of DFA

Definition 2.4 ($\equiv_L$)**.** Given a language L, $u \equiv_L w \iff \forall x \in \Sigma^*, ux \in L \iff wx \in L$, meaning $ux \in L$ and $wx \in L$, or $ux \notin L$ and $wx \notin L$

$\equiv_L$ forms an equivalent relation, meaning it is reflexive, symmetric, and transitive. Let $equiv(w)$ denote the equivalence class of w formed from $\equiv_L$. If $|\{equiv(w) | w \in \Sigma^*\}|$ is finite, then it forms a DFA $(Q, \Sigma, \delta, q_0, F)$ where

- $Q = \{equiv(w) | w \in \Sigma^*\}$
- $q_0 = equiv(\epsilon)$
- $F = \{equiv(x) | x \in L\}$
- $\delta(equiv(w), a) = equiv(wa)$

In this DFA, we are basically grouping strings into different classes based on their behaviour, any two strings in the same class will lead to the same state given the same continuation. The DFA formed is minimal, meaning there is no other DFA for L with a smaller number of states. We can prove this by showing that a DFA for L must have at least the number of equivalence classes under $\equiv_L$.
Claim: Every DFA for L must have at least the number of equivalent classes formed under $\equiv_L$. This is equivalent to the statement $u \not\equiv_L w \implies \hat{\delta}(q_0, u) \neq \hat{\delta(q_0}, w)$

Proof. Suppose otherwise, that $u \not\equiv_L w \implies \hat{\delta}(q_0, u) = \hat{\delta}(q_0, w)$. This means that there exists a smaller DFA that can place two strings into the same state even though they were from different classes.
Then, by definition of $\equiv_L$, there exists some string x such that $ux \in L$ and $wx \notin L$ or vice versa. However, $\delta(\hat{q_0, ux}) = \delta(\hat{q_0, wx})$, meaning ux and wx must either be both in L, or not, hence a contradiction. □

The intuition behind why this works is because we are finding the minimal number of classes of strings that behave differently, if two equivalent classes behaved the same way, they would not be separate. For example, given $\Sigma = \{a, b\}$ and $L = \{w | w \text{ ends with } a\}$, we have two equivalent classes, the set of strings ending in a and the set of strings ending in b. $a \not\equiv_L b$ because $a \cdot \epsilon \in L$ but $b \cdot \epsilon \notin L$, hence our DFA only needs two states as $equiv(\epsilon) = equiv(b)$.

2.6.1 Minimization algorithm

First we need to find all distinguishable pairs of states. We say that (p, q) is distinguishable iff there exists a w such that

$$\hat{\delta}(p, w) \in F \iff \hat{\delta}(q, w) \in F.$$

1. Initialise all pairs (p,q) where $p \in F$ and $q \notin F$, where they are all distinguishable pairs
2. For any $a \in \Sigma$, if $\delta(p, a)$ and $\delta(q, a)$ form a distinguisable pair, then p and q are distinguishable
3. Continue step 2 until no more distinguishable pairs can be formed, then the remaining pairs are indistinguishable

Once we have all the indistinguishable pairs, they form their own equivalence classes, if $\delta(p, a) = q$, then $\delta(Ep, a) = Eq$, where Ep and Eq are the equivalence classes for p and q.
Initial state is the equivalence class containing the start state, final states are all equivalence classes containing final states.

2.7 Properties of Regular languages

Theorem 2.7.1 (Pumping lemma)**.**

Let L be a regular language, then there exists a constant n for L such that for any string $w \in L$ where $|w| \geq n$, then we can break w into 3 strings $w = xyz$ such that

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

Proof: Consider the DFA for a regular language L which has n states and a string w of length at least n, let q_0 be the initial state and $q_1, q_2, \ldots q_n$ be the subsequent states after accepting the next n characters of w. Since the DFA only has n states but we have visited $n + 1$ states, by pigeonhole principle there must be at least 1 state repeated. Lets call this repeated state q_s, let x be the substring that ends up in the first occurence of q_s, let y be the next substring that ends up in the repeated occurence of q_s. Then let z be the rest of the string w. Since $q_s = \hat{\delta}(q_0, x) = \hat{\delta}(q_0, xy)$, then $\forall k \in \mathbb{N}, q_s = \hat{\delta}(q_0, xy^k)$

2.7.1 Decision problems

1. To decide if L is the empty set, check if all the final states are unreachable
2. To decide if L is Σ^*, take the complement of the DFA and check if all the final states are unreachable

2.8 Parallel simulation of DFA

Given two DFAs, $A = (Q, \Sigma, \delta, q_0, F)$ and $A' = (Q', \Sigma, \delta', q_0', F')$, we can do a parallel simulation to run both DFAs.
Construct another DFA $A'' = (Q \times Q', \Sigma, \delta'', (q_0, q_0', F''))$, where $\delta''((q, q'), a) = (\delta(q, a), \delta'(q', a))$, and F'' depends on the need.
For example, if we are trying to find the intersection of two languages, then $F'' = \{(q, q') : q \in F \wedge q' \in F\}$. If we are trying to find the union of two languages, then $F'' = \{(q, q') : q \in F \vee q' \in F\}$.

2.9 Homomorphism

Suppose Σ and Γ are two alphabets. Suppose h is a mapping from Σ to Γ^*. Extend h to strings as follows.
$h(\epsilon) = \epsilon$.
$h(aw) = h(a) \cdot h(w)$, for any $a \in \Sigma, w \in \Sigma^*$.
Above h is called a homomorphism. If L is regular then h(L) = $\{h(w) | w \in L\}$ is also regular. Likewise if $h(L)$ is irregular, then L is irregular.

2.10 Summary of proven statements

- Number of strings over any fixed finite alphabet Σ is countable
- Number of languages over any non-empty alphabet is uncountable
- DFA is equivalent to NFA is equivalent to ϵ-NFA is equivalent to RegLangs
- RegLang properties:
  - M+N = N+M
  - L(M+N) = L(M) + L(N)
  - L + L = L
  - $(L^*)^* = L^*$
  - $\emptyset^* = \epsilon$
  - $\epsilon^* = \epsilon$
  - $L^+ = LL^* = L^*L$
  - $L^* = \epsilon + L^+$
  - $(L + M)^* = (L^*M^*)^*$
- If a language has n equivalence classes, it must have at least n states
- Closure properties of RegLangs:
  1. If L_1 and L_2 are regular, then so is $L_1 \cup L_2$
  2. If L_1 and L_2 are regular, then so is $L_1 \cdot L_2$
  3. If L_1 and L_2 are regular, then so is $L_1 \cap L_2$
  4. If L_1 and L_2 are regular, then so is $L_1 - L_2$
  5. If L_1 and L_2 are regular, then so is $L_1 \cdot \overline{L_2}$
  6. If L, is regular, then so is $\overline{L}$
  7. If L, is regular, then so is L^R
  8. If L, is regular, then so is $h(L)$
  9. If $h(L)$ is irregular, then L is irregular.

2.11 Languages known to be regular or irregular

Regular:

- Strings that contain odd number of bs over $\{a, b\}$
- strings that contain 00 as a substring
- $L = \{w|$ n-th symbol in w from the end is 1$\}$.
- An integer that is $(+, -, \epsilon)$ followed by digits
- $\{w|$ number of a's in w is of form 3i + 1, for some natural number i$\}$ (Tutorial 1 Q4a)
- $\{w|w$ has abaab as a substring$\}$ (Tutorial 1 Q4b)
- $\{w|w$ does not contain ababba as a substring$\}$ (Tutorial 1 Q4c)
- (Odd number of a's and no b's) or (any number of a's followed by a b followed by even number of a's) (Tutorial 1 Q5)
- $\{w|w = a_1b_1a_2b_2 \dots a_nb_n,$ for some n, where $a_i, b_i \in \{0, 1\}$ and $a_1a_2 \dots a_n > b_1b_2 \dots b_n\}$
- Set of strings that end in bba
- For a string x, let x_i denote the i-th character in x. That is, x = $x_1x_2x_3 \dots x_n$, where n is the length of x and each $x_i \in \Sigma$. Suppose L is regular. Then $\{x : \exists r \in N, |x| = 2r \wedge x_1x_3x_5 \dots x_{2r-1} \in L\}$ is also regular.
- $\{wxw^R|w, x \in \{a, b\}^+\}$

- Half(L) is regular

Irregular:

- $\{a^m b^m | m \geq 1\}$
- $\{a^i b^j | i < j\}$
- $\{a^p | pisprime\}$
- $\{wcw^R | w \in \{a, b\}^*\}$ (Tutorial 3 q3a)
- $\{ww | w \in \{a, b\}^*\}$ (Tutorial 3 q3b)
- $\{a^m : m > 0$ and binary representation of m has even number of bits$\}$

3 Context Free Grammars

3.1 CFG

3.1.1 Definition

A CFG is a 4 tuple $G = (V, T, P, S)$ where

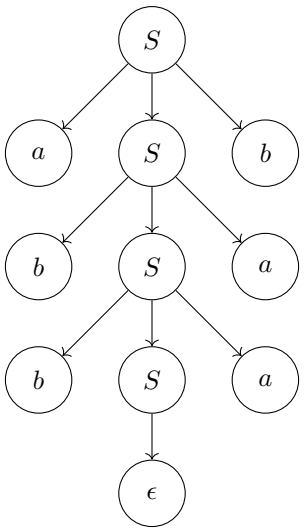
- V is a finite set of non terminals
- T is a finite set of terminals
- P is a finite set of production rules $A \to \gamma$ where $A \in V$ and $\gamma \in (VUT)^*$
- S is the starting symbol, and $S \in V$

3.1.2 Derivation

$\alpha A \beta \Rightarrow \alpha\gamma\beta$ if there exists a production rule of the form $A \to \gamma$. $\alpha \Rightarrow_G^* \beta$ if there is a series of zero or more production rules that derives beta from alpha.
If $S \Rightarrow_G^* \alpha$, then α is a sentential form. Basically any string of terminals and non terminals that can be reached from S is a sentential form.

3.1.3 Parsing

Using left most derivation, we apply the production rule on the left most non terminal. Likewise for right most derivation, we apply the production rule on the right most non terminal.
A parse tree is generated when generating a string of non terminals from the starting symbol S. For example:



3.1.4 Linear grammar

A right linear grammar is a grammar where all production rules are of the form

$$A \to wB, B \in V, w \in T^* \text{ or}$$
$$A \to w, w \in T^*$$

Likewise a left linear grammar is a grammar where all production rules are only to strings of terminals, or a non terminal followed by a string of terminals.
Linear grammars are equivalent to regular expressions.

3.1.5 Ambiguous grammars

A grammar is ambiguous if there exists two distinct parse trees to get to the same string of non terminals. Some grammars can be disambiguated by changing the production rules, but some grammars are inherently ambiguous.

3.2 PDA

A push down automata is like a finite state machine but with a stack. More formally it is defined by the 7 tuple $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, where

- Q, Σ, q_0, F are like in a DFA
- Γ is the stack alphabet
- Z_0 is the only and initial symbol on the stack
- δ is a transition function from $Q \times \Sigma \times \Gamma$ to $\mathcal{P}(Q \times \Gamma^*)$

$(p, \gamma) \in \delta(q, a, X)$ indicates that at state q, after reading input character a, and popping the top of the stack to get X, the machine can travel to state p, and push γ onto the stack. $\gamma \in \Gamma^*$, and we push characters from right to left. Note that we must always pop something from the top of the stack, we cannot progress if the stack is empty.

3.2.1 Instantaneous description

We say that $(q, aw, X\alpha) \vdash (p, w, \beta\alpha)$ if $(p, \beta) \in \delta(q, aw, X)$

3.2.2 Language acceptance

We can either accept by final state, if

$$\{w | (q_0, w, Z_0) \vdash^* (q_f, \epsilon, \alpha) \text{ for some } q_f \in F\}.$$

or by empty stack, if

$$\{w | (q_0, w, F) \vdash^* (q, \epsilon, \epsilon) \text{ for some } q \in Q\}.$$

Acceptance by final state or by empty stack is equivalent in a non deterministic PDA

3.2.3 Deterministic PDA

A deterministic PDA is just like a NPDA, but there is only one possible next move at each state, input character, and top of stack symbol. So the transition function is from $Q \times \Sigma \times \Gamma$ to $Q \times \Gamma^*$. And if there is an epsilon transition at state q and top of stack X, then there can be no other transitions for any other input symbol $a \in \Sigma$.
A deterministic PDA is strictly weaker than a NPDA, and the language it accepts it forms the class of deterministic context free languages. Acceptance by final state is also more powerful than acceptance by empty stack. Final state acceptance can accept all regular languages, but empty stack can only accept a subset of final state, namely languages where for any $x, y \in L$, x is not a prefix of y.

3.3 Procedures

There are a few procedures that transforms grammars into simpler grammars but still express the same language

3.3.1 Generating symbols

A symbol A is generating if $A \Rightarrow_G^* w$ for some $w \in T^*$
Non generating symbols in a grammar is useles, so lets get rid of them.

1. Set all terminal symbols in the grammar to be generating
2. If there exists a production of the form $A \to w$, where every symbol in w is generating, then A is generating.
3. Repeat step 2 until no more symbols can be added, then we can get rid of non generating symbols.

3.3.2 Reachable symbols

A symbol A is reachable if $S \Rightarrow_G^* \alpha A \beta$, for some $\alpha, \beta \in (V \cup T)^*$
Even if a symbol is generating, it may not be reachable from the starting symbol, such symbols are also useless, so lets get rid of them.

1. S is reachable
2. If $A \to \alpha$ and A is reachable, then all symbols in α are reachable.
3. Repeat step 2 until no more symbols can be added, then we can get rid of non reachable symbols.

3.3.3 Useless symbols

A symbol is useful only if it is reachable and generating. To get rid of useless symbols, first get rid of non generating symbols, then get rid of non reachable symbols.

3.3.4 ϵ productions

An ϵ production is a production rule that can lead to the empty string. We want to eliminate such productions
First we need to identify nullable symbol. A nullable symbol is one where $A \Rightarrow_G^* \epsilon$.

1. For all $A \to \epsilon$, set A as nullable
2. If $A \to \alpha$ and every symbol in α is nullable, then A is nullable
3. Repeat step 2 until no more nullable symbols can be added

Then to remove all ϵ productions, we do the following:

1. First, designate all non terminals A where $A \Rightarrow_G^* \epsilon$ as nullable.
2. Then, we want to remove all such productions that can lead to an empty string. For each production $B \to \alpha$, where α contains one or more nullable non terminals, replace it with all possible productions $B \to \alpha'$, which can be formed by removing some nullable non terminals in α

If S is nullable, then the new language accepts the original language, except the empty string.

3.3.5 Unit productions

If $A \Rightarrow_G^* B$, and $B \in V$, then $A \Rightarrow_G^* B$ is a unit production.
To get rid of such a unit production, we need to find all non unit productions $B \to \gamma$, and add $A \to \gamma$, and remove all productions of the form $A \to C$ where $C \in V$.
To do this, we need to find all unit pairs (A, B) where $A \Rightarrow_G^* B$

1. $\forall A \in V, (A, A)$ is a unit pair
2. If (A, B) is a unit pair, and $B \to C$, then (A, C) is a unit pair.

All unit pairs are unit productions

3.3.6 Chomsky normal form

Every grammar can be converted to a chomsky normal form, where all production rules are of the form $A \to BC$ or $A \to a$, where $a \in T$ and $A, B, C \in V$
To convert a grammar to chomsky normal form:

1. Remove all ϵ productions
2. Remove all unit productions
3. Convert all productions of length greater than 2 to length 2 (only non terminals in RHS) or 1 (only terminal).

Given a production of length greater than 2, $A \to X_1 X_2 \dots X_n$, we can convert it to the following set of productions:
$A \to Z_1 B_2$
$B_2 \to Z_2 B_3$
$\dots$
$B_{n-1} \to Z_{n-1} Z_n$
$Z_i \to X_i$ if $X_i \in T$
$Z_i = X_i$ if $X_i \in V$

3.4 Properties

3.4.1 Size of parse tree

Given a grammar in chomsky normal form, if the length of the longest path from root to a node is n, then the size of the string generated is at most 2^{n-1}

3.4.2 Pumping Lemma

Let L be a CFL. Then there exists a constant n such that, if z is any string in L such that $|z| \geq n$, then we can write $z = uvwxy$ such that:

1. $|vwx| \leq n$
2. $vx \neq \epsilon$, meaing only at most 1 of v or x can be ϵ
3. $\forall i \geq 0, uv^i wx^i y \in L$

Proof of Pumping Lemma for CFL Given L is a CFL, WLOG, assume $L \neq \emptyset$ and $L \neq \{\epsilon\}$. There exists a Chomsky Normal Form $G = (V, T, P, S)$ for $L - \{\epsilon\}$.
Let $m = |V|$ and $n = 2^m$, Given a string $z \in L$ of size at least 2^m, there exists a path from root to leaf of at least $m + 1$ due to the binary tree nature of the parse tree. Consider one such path, there are m+1 non terminals, hence by pigeonhole principle, we have at least 1 non terminal repeated along this path.
Now consider a string $z = uvwxy$, where $S \underset{G}{\overset{*}{\Rightarrow}} uAy \underset{G}{\overset{*}{\Rightarrow}} uvAxy \underset{G}{\overset{*}{\Rightarrow}} uvwxy$ Note that A represents the subpath that starts and ends at the repeated non terminal, so upon encountering the same non terminal at the end, we can repeatedly loop it.
Hence $A \underset{G}{\overset{*}{\Rightarrow}} v^i w x^i$

3.4.3 Ogdens lemma

Let L be a CFL, then there exists some constant n such that for any string z where $|z| \geq n$ and $z \in L$, we can arbitrarily mark any n characters in z to be distinguished, and split z into $uvwxy$ such that the following will always hold:

- vwx has at most n distinguished positions
- vx has at least 1 distinguished position
- $\forall i \in \mathbb{N}$, $uv^i wx^i y \in L$

3.4.4 Closure

- Closure under substitution.

  Given a mapping between each terminal a to a CFL L_a, where $s(a) = L_a$, define $s(w)$ as follows:

  $s(\epsilon) = \{\epsilon\}$
  $s(wa) = s(w) \cdot s(a)$

  Then $\cup_{w \in L} s(w)$ is a CFL

  The idea is that for each terminal a, we have a grammar for L_a which is $G_a = (V_a, T_a, P_a, S_a)$. Now whenever a is encountered in the derivation of our new language, we replace it with S_a.

  Formally, let the grammar for our new language be $G' = (V', T', P', S)$, where

  - V' is $V \cup (\cup_{a \in T} V_a)$, that is all the new non terminals including our original ones.
  - T' is $U_{a \in T} T_a$, that is all the new terminals, excluding the old terminals as they are being replaced with non terminals.
  - P' is $P_{new} \cup (\cup_{a \in T} P_a)$ where P_new is formed by replacing any terminals a in the production rules in the original grammar with S_a.

- Closure under reversal

  $L^R = \{w^R : w \in L\}$, where if L is a CFL, then L^R is also a CFL.

  This one is quite intuitive, just reverse the production rules.

- Closure under intersection with regular language

  Given L is a CFL and R is a regular language, then $L \cap R$ is a CFL.

  This one is also quite intuitive, just do a parallel simulation between the PDA for L and the DFA for R. We only need 1 stack for the PDA.

- Not closed under intersection with another CFL

  Again pretty intuitive, to do a parallel simulation, you need to keep track of two stacks, this becomes a turing machine.

3.4.5 CYK

The CYK (Cocke–Younger–Kasami) algorithm is a DP algo used to test membership in the CFL
Here is the main idea:

1. Consider the chomsky normal form grammar for our language L
2. Given the string we want to test, $w = a_1 a_2 \dots a_n$, we determine the set of non terminals $X_{i,j}$ that generate the string $a_i a_{i+1} \dots a_j$
3. Base case: $X_{i,i}$ is just the set of non terminals $A \to a_i$
4. Inductive case: $X_{i,j}$ contains all nonterminals A such that $A \to BC$ where $B \in X_{i,k}$ and $C \in X_{k+1,j}$ for $i \leq k < j$, so B generates $a_i a_{i+1} \dots a_k$ and C generates $a_{k+1} a_{k+2} \dots a_j$.
5. Now we can find $X_{1,n}$
6. If $X_{1,n}$ contains S, then $w \in L$

For $i = 1$ to n do
Let $X_{i,i} = \{A : A \to a_i\}$.
EndFor
For $s = 1$ to n - 1 do
    For $i = 1$ to $n - s$ do
        Let $j = i + s$.
        Let $X_{i,j} = \{A : A \to BC, B \in X_{i,k}, C \in X_{k+1,j}, i \leq k < j\}$
    EndFor
EndFor
Note that in the above algorithm, $X_{i,k}$ and $X_{k+1,j}$ are already computed by the time $X_{i,j}$ is computed, since k - i and j - (k + 1) are both $<$ j - i.

4 Turing Machines

4.1 Definitions

The basic model of turing machine used in this module is a 7 tuple $(Q, \Sigma, \Gamma, \delta, q_0, B, F)$ where:

- Q, Σ, Γ are the states, input alphabet and tape alphabet respectively
- δ is a function $Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$
- q_0 is the initial state
- B is a blank symbol where $B \in \Gamma - \Sigma$
- F is a set of final states.

The input to the turing machine is usually given without any blanks. The head of the machine starts at the leftmost non blank symbol, at every step it must write some symbol to the current cell before moving left or right.
The instantaneous description of a TM is given as the values of all cells written out, with the current state to the left of the head, e.g. $x_0x_1 \dots x_{n-1}qx_nx_{n+1} \dots x_m$. Blank symbols are not shown on either ends unless the head is among the blanks.
A turing machine accepts an input on the tape if

$$q_0x \vdash^* \alpha q_f \beta.$$

where $q_f \in F, x \in \Sigma^*$ and $\alpha, \beta \in \Gamma^*$.

4.1.1 Languages and functions

A function can be computed by a turing machine if it halts on all inputs where $f(x)$ is defined. There are many ways to define the output of $f(x)$, can be taken to be the non blank symbols on the tape, or there can be a specific output tape to be written to.

- A language is recursively enumerable (RE) if a turing machine can accept the language (it does not need to halt on inputs not in the language). It can also be shown that if a turing machine can print out all elements of the language (regardless of order) then the language is RE.
- A language is recursive if a turing machine can accept the language and halt on inputs not in the language. It can also be shon that if a turing machine can print out all elements of the language in lexicographic order, then the language is recursive.
- A function is partially recursive (partially computable) if a turing machine halts on all inputs where f is defined on and computes the output, but does not halt on other inputs
- A function is recursive (computable) if a function can compute it and halts on all inputs

4.1.2 Equivalent models

There are many modifications you can make to the basic model that remains equally as powerful

1. Stay at the current cell with an S move
2. Subroutines, like function calling
3. Multiple tapes can be simulated by using Γ^n, where each dimension represents one tape.
4. Right only infinite tape, cannot move past left boundary. A two way infinite machine can still be simulated using two tapes, where one tape represents the right direction and another represents the left direction.
5. Non determinismm can be simulated through BFS by storing IDs separated by a special character # on a separate tape as a queue.

4.1.3 Church Turing hypothesis

The church turing hypothesis is merely saying that any effective method on the naturals can be done by a turing machine, so far there has not been an example of a useful algorithm that cannot be computed by a turing machine.

4.1.4 Codings of turing machines

A string x over $\{0,1\}^*$ can be given the number $1x$ - 1 in binary, assigning a unique natural number to every possible string. This can be generalised to any alphabet size.
Using this coding of strings, we can assign every possible turing machine a natural number.

- Number every state, $q_1, q_2, \ldots$ where q_1 is the start state.
- Number every tape symbol $X_1, X_2, \ldots$
- Assign a number to the directions, where L is D_1 and R is D_2.
- For every transition $\delta(q_i, X_j) = (q_k, X_l, D_m)$, map it to the binary string $0^i, 10^j10^k10^l10^m$.
- Assign a unique natural number to every transition $C_1, C_2, \ldots$, then the turing machine can be encoded as $C_111C_211\ldots C_n$

Then the turing machine M_i can refer to the turing machine with the number i using the string coding mentioned above, where a leading 1 is placed in front of the code for the machine. The use of this encoding allows us to refer to machines by number, and is very useful in proofs.

4.2 Properties of languages

- If L is recursive then $\overline{L}$ is recursive. Since L is recursive, there is a machine M that halts on all inputs, and either accepts or rejects. We can construct M', that also halts on all inputs, and accepts iff M rejects
- L is recursive iff L and $\overline{L}$ is RE. The forward direction is trivial, for the other direction, given that L and $\overline{L}$ is RE, there exists M and M' that halts on inputs in L and inputs not in L respectively. One can then construct M'', that runs M and M' simultaneously, alternating between the two machines step by step. If M ever accepts, then M'' accepts. If M' accepts, then M'' rejects.
- RE langs are closed under union, given $L_1 \cup L_2$, we can perform a similar simulation as above, that simultaneously simulates both machines until one of them accepts.
- RE langs are closed under intersection. Given $L_1 \cap L_2$, we accept $\iff$ both machines accept. In this scenario, we don't have to do a parallel simulation.
- Recursive langs are closed under union. Given $L_1 \cup L_2$, simulate M_1, then if it rejects then simulate M_2, we are safe since they are guaranteed to halt.
- Recursive langs are closed under intersection. Given $L_1 \cap L_2$, simulate M_1 then if it accepts, simulate M_2 and accept if M_2 accepts. Reject in all other cases.
- If L_1 is recursive and L_2 is RE, then $L_2 - L_1$ must be RE since $L_2 - L_1 = L_2 \cap \overline{L_1}$. For $L_1 - L_2$, we can only say that its complement is RE.
- Every finite language is recursive, since you can just brute force
- Every co finite language ($\overline{L}$ is finite) is recursive, since you can still brute forceA
- If L is recursive and D is finite, then $L \triangle D$ is recursive, where $L \triangle D = L - D \cup D - L$
- If L is RE and not recursive, then there must be infinitely many inputs for M which do not halt.

4.3 Universal Turing Machine

$L_u = \{(M, w) : M \text{ accepts } w\}$.
This is a turing machine that can simulate any other turing machine. It uses 4 tapes: the input tape, a tape for M, a tape to store the state of M, and a scratch tape for temporary storage. WLOG we assume $w \in \{0,1\}^*$
The state of M is stored as 0^s, the tape for M is stored using the string encoding mentioned above, with 10 for zero and 100 for one. The head of the simulation always lies on the 1 before 0^j
Then this machine can simulate M on input w. At every step it searches for the transition of the form $0^i10^j10^k10^l10^m$, where $i = s$ and j is the value of the current cell (1 for zero, 2 for one, 3 for blank). The tape for the state of M can then be changed to 0^k.
To write 0^l to the tape for M, first mark the head with a special character $*$, then copy the tape for M to the scratch tape. Upon seeing $*$, write $*0^l$ instead, and copy the rest. Then copy this back into the tape for M, and set the head back to the $*$, and write 1 to it. Lastly, move to the left or right depending on 0^m.

4.4 Non RE langs

Let $L_d = \{w_i : w_i \notin L(M_i)\}$. This language is not RE, it is similar to russels paradox.

Proof. Suppose by way of contradiction that L_d was RE, then there exists M where $L(M) = L_d$. For any w, suppose $w \in L(M)$, then $w \notin L_d$ by definition. Suppose $w \notin L(M)$, then $w \in L_d$ by definition. In both cases, $L(M) \neq L_d$, hence by contradiction there does not exist a turing machine that accepts L_d and it is not RE □

$\overline{L_u}$ is not RE. $\overline{L_u} = \{(M, w) : w \notin L(M)\}$

Proof. Suppose by way of contradiction that $\overline{L_u}$ was RE, then we have a machine M that accepts $\overline{L_u}$. Using M, we can construct M' that accepts L_d.
M' on input w_i extracts the code i from w_i, then runs M on (M_i, w_i). M' accepts iff M accepts.
M' will accept any w_i that is not accepted by M_i, however this is a contradiction as there cannot exist a turing machine that accepts L_d, hence $\overline{L_u}$ cannot be RE. □

4.5 Reductions

P_1 reduces to P_2, if some recursive function f behaves as follows

$$x \in P_1 \iff f(x) \in P_2.$$

We say that $P_1 \leq_m P_2$. Here P is a language, but it can be thought of as a "problem", where the solutions to the problem is itself a language.

1. If P_2 is recursive, then so is P_1. Intuitively you can just use the machine for P_2 to decide if $x \in P_1$
2. If p_2 is RE, then so is P_1. Same reasoning as above.
3. If P_1 is undecideable, then so is P_2. Undecideable means not recursive. This follows from 1, if P_2 was recursive then P_1 must be recursive.
4. If P_1 is not RE, then so is P_2. Same reasoning as above.

So if we want to show some language P_2 is "hard" (not RE), find a P_1 which is known to be not RE and map every element in P_1 to some element in P_2.

4.5.1 TM accepting empty set

Let $L_e = \{M : L(M) = \emptyset\}$ and $L_{ne} = \{M : L(M) \neq \emptyset\}$. We show that L_{ne} is RE and L_e is not RE.
To show that L_{ne} is RE, we can construct a machine that accepts L_{ne}. Our machine M on input M' uses dovetailing to simulate every possible input on M' until it accepts.
For $t = 0$ to ∞
    For $i = 0$ to t
        If $M(w_i)$ accepts within t steps, then accept

To show that L_e is not RE, we can reduce $\overline{L_u}$ to L_e. Given any machine word pair (M, w), we construct M'
$M'(x)$
For $t = 0$ to ∞
    If $M(w)$ accepts within t steps then accept.
This mapping is obviously recursive.
If $w \notin M$, then $M(w)$ never accepts and M' also will never accept any input. Hence $(M, w) \in \overline{L_u} \iff L(M') = \emptyset)$

4.6 Rice's theorem

Given a non trivial property P about RE langs, $\{M : L(M)$ satisfies $P\}$ is undecideable. A property is non trivial if there exists at least one RE language which does not satisfy the property and one RE language that satisfies the property. In other words, a property is non trivial iff it is not true/false for all turing machines.
Proof: We reduce L_e to L_P. WLOG assume $L = \emptyset$ satisfies P. (Otherwise switch P and $\overline{P}$). Suppose L is an RE language that does not satisfy P (Meaning it is not $\emptyset$), let M'' accept L. Define $f(M) = M'$ as follows:
$M'(x)$:
For $t = 0$ to ∞:
    For $i = 0$ to t:

If $M(w_i)$ accepts within t steps and $M''(x)$ accepts within t steps then accept x.
If $L(M) = \emptyset$ (meaning $M \in L_e$), then $L(M') = \emptyset$ (meaning $M' \in L_P$)
If $L(M) \neq \emptyset$ (meaning $M \notin L_e$), then $L(M') = L(M'') = L$ (meaning $M' \notin L_P$)
Hence $L_e \leq_m L_P$, since L_e is not recursive, we have that L_P is not recursive aka undecideable.

4.7 Undecideable problems

4.7.1 Post Correspondence Problem

The Post Correspondence Problem is an undecideable problem that states: Given two lists of strings as input, $\alpha_1, \alpha_2, \ldots, \alpha_k$ and $\beta_1, \beta_2, \ldots, \beta_k$, does there exists a list of indices $1 \leq i_n \leq k, 1 \leq n \leq m$ for some $m > 0$, such that $\alpha_{i_1}, \alpha_{i_2}, \ldots, \alpha_{i_m} = \beta i_1, \beta i_2, \ldots, \beta i_m$.
For example, given $[a, ab, bba]$ and $[baa, aa, bb]$, a solution would be $[3, 2, 3, 1]$, since $bba\,ab\,bba\,a = bbaabbbaa = bb\,aa\,bb\,baa$.
The Modified Post Correspondence Problem restricts the first string in the solution to some particular pair of α_i and β_i (WLOG assume α_1 and β_1). This problem asks if there exists a list of indices $1 \leq i_n \leq k, 1 \leq n \leq m$ for some $m > 0$, such that $\alpha_1 \alpha_{i_1}, \ldots, \alpha_{i_m} = \beta_1, \beta i_1, \ldots, \beta i_m$.
We show that $L_u \leq_m MPCP \leq_m PCP$

Proof. $MPCP \leq_m PCP$ □

To show that $MPCP \leq_m PCP$, we must show that if some input has a solution, then there exists a transformation on the inputs that have a solution in PCP. We must also show that if some transformed input has a solution in PCP, then the untransformed input has a solution in MPCP.
Suppose we had $\alpha_1, \alpha_2, \ldots, \alpha_k$ and $\beta_1, \beta_2, \ldots, \beta_k$. Then form

1. α'_i, by adding $*$ after each symbol in α_i
2. β'_i, by adding $*$ before each symbol in β_i
3. $\alpha'_0 = *\alpha'_1, \beta'_0 = \beta'_1$
4. $\alpha'_{k+1} = \$$
5. $\beta'_{k+1} = *\$$

If α', β' has a solution in PCP, then we can just take the untransformed input as a solution, and drop α'_{k+1} and β'_{k+1}. This works because now $\alpha'_0 = \alpha_1$ and $\beta'_0 = \beta_1$.
If α and β has a solution in $MPCP$ then when we transform the input, we are forced to use α'_0 and β'_0 at the start of our solution, and α'_{k+1} and β'_{k+1} at the end of our solution. Then we can freely insert the solution to the MPCP problem.

Proof. $L_u \leq MPCP$ □

For this proof, we will use machines that can not move to the left of the leftmost symbol and never writes a blank. we use symbols X, Y, Z to mean any symbol in Γ, likewise we use p, q to mean any state in Q. Given any $M\#w$ in L_u, we produce the following input lists:

List α	List β	
$\#$	$\#q_0w\#$	
X	X	For all $X \in \Gamma$, to construct the tape that does not surround the head
$\#$	$\#$	To divide each ID
qX	Yp	if $\delta(q, X) = (p, Y, R)$
ZqX	pZY	if $\delta(q, X) = (p, Y, L)$
$q\#$	$Yp\#$	if $\delta(q, B) = (p, Y, R)$
$Zq\#$	$pZY\#$	if $\delta(q, B) = (p, Y, L)$
XqY	q	if q is an accepting state
Xq	q	if q is an accepting state
qX	q	if q is an accepting state
$q\#\#$	$\#$	if q is an accepting state

We basically encode (M, w) as an input to MPCP. For each (α, β) pair in the center 4 rows, β represents the next state of α. The solution, if it exists, will contain the instantaneous descriptions of the turing machine for each step until the machine halts. It should be trivial to see that $(M, w) \in L_u \iff (\alpha, \beta)$ has a solution for $MPCP$

4.7.2 Ambiguous grammars

Suppose input to PCP is $A = w_1, \ldots, w_k$ and $B = x_1, \ldots, x_k$. Let $a_1, \ldots, a_k$ be symbols of a grammar not in A or B.
$S \to A|B$
$A \to w_i A a_i | w_i a_i, 1 \leq i \leq k$
$B \to x_i B a_i | x_i a_i, 1 \leq i \leq k$
The use of a_i is to keep track of the indices used, the fact that they are in reverse order does not matter. Now it is intuitive that if A and B can generate the same string, then there is a solution to PCP. However if they can generate the same string, then the grammar is ambigious.
Hence G is ambiguous iff $L(G_A) \cap L(G_B) \neq \emptyset$ iff PCP has a solution to A and B.
Note that $\overline{L(G_A)}$ and $\overline{L(G_B)}$ are context free. We show that $\overline{L(G_A)}$ is context free, which also shows $\overline{L(G_B)}$ is context free.
Recall that G_A generates strings of the form

$$w_{i_1} w_{i_2} \ldots w_{i_m} a_{i_m} \ldots a_{i_2} a_{i_1}.$$

Where a sequence of words is followed by a sequence of indexes in reverse order. Let us build a grammar that covers all cases which breaks this pattern.
$S \to \epsilon | C | D | E | F.$
Let C cover the cases where a word follows an index, breaking the pattern.
$C \to (\Sigma \cup I)C | C(\Sigma \cup I) | I\Sigma$
Where Σ represents the alphabet of the original PCP problem, and I represents the set of indices in that problem.
Then, let D cover the cases where the index do not match up with the word it should represent.
$D \to w_i D a_i | A_i' D_1 a_i$
$D_1 \to \Sigma D_1 | D_1 I | \epsilon$
Where A_i' generates all strings that are not prefixes of w_i and of length at most $|w_i|$. This causes the mismatch, and after the mismatch, we can generate any string and index we want.
Then, let E cover the cases where there are excess elements in Σ
$E \to w_i E a_i | E_1$
$E_1 \to \Sigma E_1 | \Sigma$
Lastly, let F cover the cases where there are excess indices in I.
$F \to w_i F a_i | A_i F_1 a_i$
$F_1 \to I F_1 | \epsilon$

4.7.3 Further undecideable grammars

1. $L(G_1) \cap L(G_2) = \emptyset$, for CFGs, obvious by letting $G_1 = G_A$ and $G_2 = G_B$
2. Universality problem: Deciding if $L(G) = \Sigma^*$. Let $G = \overline{L(G_A)} \cup \overline{L(G_B)} = \Sigma^* - (L(G_A) \cap L(G_B))$. Proving if $L(G) = \Sigma^*$ involves proving whether $L(G_A) \mathbb{N} L(G_B) = \emptyset$.
3. $L(G_1) = L(G_2)$, let G_1 be any context free grammar and let G_2 generate Σ^*.
4. $L(G) = L(R)$ where R is a regular expression, consequent of the universality problem.
5. $L(G_2) \subseteq L(G_1)$, as this can allow you to decide $L(G_1) = L(G_2)$.
6. $L(R) \subseteq L(G)$, as this can allow you to decide $L(R) = L(G)$.

4.8 Unrestricted grammar

An unrestricted grammar is kinda like a string rewriting system, however note that this module does not allow the left hand side of a production to only contain terminals, there must be at least one non terminal. This is easily fixed by using temporary non terminals to represent terminals, then having a single production rules mapping the temp non terminals to the terminals they represent.
Formally, an unrestricted grammar is a 4 tuple (N, Σ, S, P), almost the same as a CFG but only difference is that the production rules can be of the form

$$\alpha \to \beta.$$

where $\alpha \in (N \cup \Sigma)^* N (N \cup \Sigma)^*$, and $\beta \in (N \cup \Sigma)^*$.
If we restrict $|\alpha| \leq |\beta|$, then the grammar is context sensitive, which is one level below turing machines in the chomsky hierarchy, with space linear bounded turing machines being the automata equivalent.
Unrestricted grammars are strictly as powerful as turing machines, due to them essentially being string rewriting systems, any machine word pair can be represented by an unrestricted grammar, by letting the start symbol map to the ID of the initial state of the turing machine, and using production rules to simulate the transitions. Hence if G is an unrestricted grammar, then $L(G)$ is RE.

5 Complexity

For notions of complexity, we assume turing machines with fixed but arbitrary number of tapes with finite tape alphabet, using begin and end markers $ surrounding the input.

5.1 Time Space complexity

5.1.1 Time complexity

We define the time complexity of a machine on an input as $Time_M(x)$, representing the number of steps used by machine M on input x before halting.
For non deterministic machines, use the maximum time of any path, regardless if that path is accepting.
We define M to be $T(n)$ time bounded if $\forall x \in \Sigma^*, Time_M(x) \leq T(|x|)$. We usually assume $T(n) \geq n$.

5.1.2 Space complexity

We define the space complexity of a machine on an input as $Space_M(x)$, representing the maximum number of unique cells visited by machine M on input x across all its work tapes before halting, not counting the input/output tape.
If the machine does not halt, it is taken to be ∞
We define M to be $S(n)$ time bounded if $\forall x \in \Sigma^*, Space_M(x) \leq S(|x|)$.

5.1.3 Time Space classes

- $DSPACE(S(n)) = \{L|$ some $S(n)$ space bounded deterministic machine accepts $L\}$.
- $DTIME(S(n)) = \{L|$ some $T(n)$ time bounded deterministic machine accepts $L\}$.
- $NSPACE(S(n)) = \{L|$ some $S(n)$ space bounded non deterministic machine accepts $L\}$.
- $NTIME(S(n)) = \{L|$ some $T(n)$ time bounded non deterministic machine accepts $L\}$.

5.2 P and NP

P $= \{L|$ some deterministic turing machine time bounded by a polynomial function accepts $L\}$
NP $= \{L|$ some non deterministic turing machine time bounded by a polynomial function accepts $L\}$
coNP $= \{L|\overline{L} \in \mathbf{NP}\}$
Suppose $L \in NP$, then we say that we can check whether $x \in L$ given a "certificate" in deterministic polynomial time. Basically solutions to NP problems are easy to verify, but hard to solve.
For example, given a partially filled sudoku grid, it is NP hard whether there exists a solution, but given a solution to the sudoku, it is trivial to verify whether it is a correct solution.

Proof. Suppose N is a $q(n)$ time bounded NDTM accepting L, where q is a polynomial.
WLOG, assume at each state, N has exactly 2 choices to branch out to. This can be done by "normalizing" the choices, kinda like CNF, and N is still polynomial time bounded.
Define a computable polynomial predicate $P(x, y)$, where x is a solution to the problem and hence in L, and y is a binary string of length at most $q(|x|)$.
$P(x, y)$
    Let $y = y_1 y_2 \dots y_m$.
    If $m > q(|x|)$ then reject.
    Otherwise simulate N on x, at each step i choosing the first or second choice based on whether y_i is 0 or 1.
    Return 1 iff N accepts.
Hence $\exists y$ where $|y| \leq q(|x|)$ such that $P(x, y) = 1 \iff N(x)$ has an accepting path.. □

We call $P(x, y)$ a "certificate" or "proof" that $x \in L$, since y lays out the path for N to take, it is no longer non deterministic, hence we can consider **NP** as a class of languages where proofs for membership can be easily verified in deterministic polynomial time.

5.2.1 Reducibility

Reducibility is similarly defined for NP problems, we have the added caveat that the function doing the reduction is computable in polynomial time.

- $L_1 \leq_m^p L_2$ means that there exists a poly time computable function f such that $x \in L_1 \iff f(x) \in L_2$
- $L_1 \leq_m^{\log \text{ space}} L_2$ means that there exists a log space bounded computable function f such that $x \in L_1 \iff f(x) \in L_2$

  Note that $\leq_m^p$ is reflexive by identity function, and transitive by function composition.

5.2.2 NP Completeness

A set L is said to be **NP**-complete iff it is in **NP** and $\forall L' \in NP, L' \leq_m^p L$. If the latter criteria is satisfied, then the problem is said to be **NP** hard.
Hence showing that an **NP** complete problem is actually solvable in polynomial time by a deterministic TM proves that $P = NP$.
Another result is that given an **NP** complete problem L, and an **NP** problem L', showing that $L \leq_m^p L'$ is sufficient to show that L' is **NP** complete.

5.2.3 Some known NP Complete problems

1. Satisfiability

   Given an instance of variables U, and a collection of clauses C over U, is there an assignment of boolean values to each variable such that every clause is true?

   In this problem, the whole boolean formula is in product of sums form, meaning each clause is $(A \vee B \vee C \ldots)$ and every clause is boolean AND together. Within each clause, variables can be negated.

   Every sat problem can be converted to 3 SAT, where each clause has 3 literals.

   This problem was the first problem proven to be **NP** complete, first by showing that it is in fact in **NP**, and that every problem in **NP** can be reduced to 3 SAT using a poly time computable function. It worked by encoding the turing machine as boolean variables and clauses.

2. 3-Dimensional matching

   Given three disjoint finite sets X, Y, Z each of cardinality n, and a set $S \subseteq X \times Y \times Z$. Does there exist a subset $S' \subseteq S$ such that $|S'| = n$ and no two elements of S' agree in any coordinate?

3. Vertex cover

   Given a graph $G = (V, E)$ and a positive interger $K \leq |V|$, is there a subset $V' \subseteq V$ where every vertex in G' lies on an edge in G and $|V'| \leq K$?

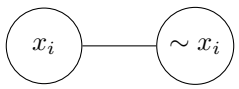
   Proof. To show that vertex cover is NP complete, we first show that it is in NP. Given $V, k,$ and V', we just need to check that

   (a) $|V'| \leq k$
   (b) $\forall (u, w) \in E, u \in V' \vee w \in V'$

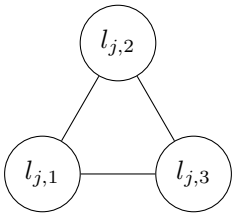
   This is obviously polynomial time computable. Next we need to show that it is NP hard, to do so we show that 3SAT $\leq_m^p$ Vertex Cover.

   Given a collection of variables $\{x_1, x_2, \ldots, x_n\}$ and a collection of clauses $\{c_1, c_2, \ldots c_m\}$, each clause is composed of 3 literals $\{l_{j,1}, l_{j,2}, l_{j,3}\}$ where $1 \leq j \leq m$.

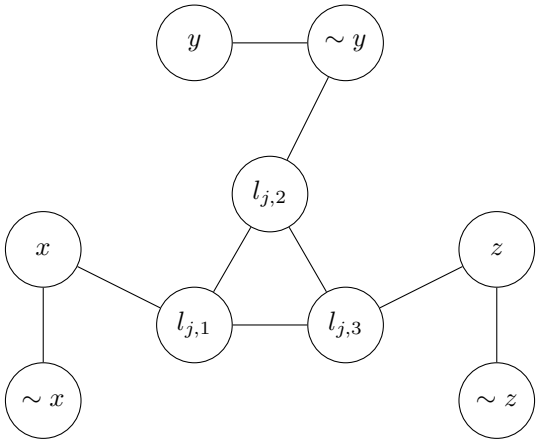
   For each variable x_i, we construct 2 vertices that represent x_i and $\sim x_i$, and an edge $(x_i, \sim x_i)$. The idea is that we force the vertex cover to pick 1 of the vertices, representing the boolean value of x_i

Then for each clause $\{l_{j,1}, l_{j,2}, l_{j,3}\}$, we construct a triangle, where exactly 2 vertices must be in the vertex cover.



For each clause, we connect each literal to the variable it represents, for example, given the clause $x \vee \sim y \vee z$, it would look like



Now, selecting the right k allows us to choose exactly 2 literals in each clause and exactly 1 boolean assignment for each variables to be in the vertex cover, we can use $k = n + 2m$ for this.

The literal not choosen represents the literal we select to be true, since in each clause, only 1 of the literals need to be true for the whole clause to be true. Since this literal was not choosen, then the variable that the literal is connected to must be choosen to be in the vertex cover.

Hence if there exists a vertex cover for this graph, there must be a satisfying truth assignment to each variable for the satisfiability problem. And given a satisfiability problem and a correct assignment of truth values to each variable, we can simply select either x_i or $\sim x_i$ to be in the cover, and for each clause the variable is connected to, select the literals not connected to x_i or $\sim x_i$.

Hence 3SAT $\leq_m^p$ Vertex Cover, and vertex cover is NP-complete.

□

4. Max cut

   Given an undirected graph $G = (V, E)$ and a positive integer $K \leq |E|$, is there a cut of G with size $> K$? A cut partitions the vertices into two sets (X, Y), the size of the cut is the number of edges removed to form the partition.

5. Independent set

   Given a graph $G = (V, E)$ and a positive integer $K \leq |V|$, is there a subgraph of size $\geq K$ where no vertices are adjacent?

   Proof. Given a graph $G = (V, E)$, if $V' \subseteq V$ is a vertex cover for G, that means that $V - V'$ is an independent set. To see this, suppose $\{u, v\} \subseteq V - V'$, then there cannot exist $(u, v) \in E$ otherwise one of u or v will be in V'. Hence there exists no edges between any vertices in V'.

   In the other direction, if V' is an independent set for G, then $V - V'$ is a vertex cover. □

6. Clique

   Given a graph $G = (V, E)$ and a positive integer $K \leq |V|$, is there a complete subgraph of size $\geq K$? Basically finding a subgraph of G where every vertice is adjacent.

   Proof. We can reduce independent set to clique problem. Given a graph $G = (V, E)$, then there exists an independent set in G iff there exists a clique in $\overline{G}$.

   This is trivial, if V' is an independent set, then no vertices are adjacent, so in the complement of the graph, all the vertices in V' will be adjacent to each other. □

7. Hamiltonian circuit

   Given a graph $G = (V, E)$, does G contain a hamiltonian circuit? i.e is there a simple circuit that goes through all vertices exactly once?

8. Partition

   Given a finite set A and a size funtion where $\forall a \in A, s(a) > 0$, is there a subset A' of A such that

$$\sum_{a \in A'} s(a) = \sum_{a \notin A'} s(a).$$

   $s(a)$ is given in binary for every a, so input length is proportional to $|A| + \sum_{a \in A} \log s(a)$

9. Set cover

   Given a finite set A and a number k, a collection of subsets $S = \{S_1, S_2, \ldots, S_m\}$ whose union is A, is there a subset $S' \subseteq S$ where $|S'| \leq k$ such that the union is still A?

10. Traveling salesman problem

    Given a complete weighted graph $G = (V, E)$ and a bound B, does there exist a hamiltonian circuit in G of weight $\leq B$? Weights and B are provided in binary, so length of input is proportional to $|V| + |E| + \log B + \sum_{e \in E} \log w_e$

5.3 Constant improvements

Given that a language L can be accepted by some machine M that is $S(n)$ space bounded, then there exists a machine M' that can accept the same language with the same number of tapes in $\lceil cS(n) \rceil$ for some $c > 0$.
The idea is that each cell of M' represents m cells of M, so if the tape alphabet of M is Γ, then the tape alphabet for M' is Γ^m, so the space used by this machine is $\left\lceil \frac{S(n)}{m} \right\rceil$

The same idea can be applied to time complexity, given some conditions. Our machine M must have $k \geq 2$ tapes, and must be $T(n)$ time bounded such that $T(n) \notin O(n)$, i.e. $\lim_{n \to \infty} \frac{T(n)}{n} = \infty$. Then we can construct M' that is $\lceil cT(n) \rceil$ time bounded for $c > 0$
The idea is to first apply the space compression so each cell takes up m cells of the original machine. Then for every step of M', we look at the left and right neighbours of the current cell, this gives us $3m$ cells of information. Since there is a finite number of states, this can be simulated by a DFA, we can figure out whether the machine will accept, reject (including never halting), or move out of this $3m$ region. This can be precomputed, so in the worst case our machine needs to take 8 steps (3 steps to read the region, 2 steps to write to the region, 3 steps to leave the region).
In one 'basic' step, we have simulated at least m steps of M,

5.4 Some results

$DTIME(S(n)) \subseteq DSPACE(S(n))$, i.e. you can not take more time than the amount of space you time, because using up 1 space requires 1 unit of time.

If L is in $DSPACE(S(n))$, and $S(n) \geq \log n$, then there exists a constant c depending on L such that L is in $DTIME(c^{S(n)})$. Intuitively since L is $S(n)$ space bounded, there is a finite number of states the machine can be in, so you can take c to be $|\Gamma|$.

If L is in $NTIME(T(n))$, then there exists a constant c depending on L such that L is in $DTIME(c^{T(n)})$. Intuitively this is just a deterministic simulation of the non-deterministic TM.

Theorem 5.4.1. *Suppose $S_2(n)$ and $S_1(n)$ are both $\geq \log n$ and $S_2(n)$ is fully space constructible (uses exactly $S(n)$ space on all inputs n) and $S_2(n)$ grows faster than $S_1(n)$, i.e. $\lim_{n \to \infty} \frac{S_1(n)}{S_2(n)} = 0$. Then there is a language in $DSPACE(S_2(n)) - DSPACE(S_1(n))$.*

Proof. Construct M that is $S_2(n)$ space bounded, has a fixed number of tapes (at least 3), and let L denote the language accepted by M. We show that L will not be in $DSPACE(S_1(n))$.

M rejects all inputs of the form 1^k.
M on input of the form $1^k 0x$:

Mark out space $S_2(|1^k 0x|)$ on the work tape. This is needed to figure out how much space $S_2(|1^k 0x|)$ takes, and allows us to reject when the simulation crosses this boundary.

Simulate M_x on the same input, if the simulation attempts to use more than $S_2(|1^k 0x|)$ space, then reject.

Else if M_x halts on the same input, then M accepts iff M_x did not accept the input.
In the above simulation, we assume that M_x uses only 2 tapes and uses the alphabet $\{0, 1, B\}$. Note that M is also $S_2(n)$ space bounded, hence L is in $DSPACE(S_2(n))$.

Suppose by way of contradiction that M' is a $S_1(n)$ space bounded machine that accepts L. Then we can construct a 2 tape machine M_x that accepts L and is also $S_1(n)$ space bounded. WLOG we can assume M_x halts on all inputs.
Now we try to simulate M_x on M, let k be large enough such that $cS_1(|1^k0x|) < S_2(|1^k0x|)$, then we know that this simulation must complete, and M either accepts or rejects. Then M accepts 1^k0x iff M_x did not, this is a contradiction as 1^k0x should be accepted by both machines.
Thus there can not be a machine M' that accepts L and is S_1 space bounded.

□

Theorem 5.4.2. *Suppose $T_2(n)$ is fully time constructible and $T_2(n), T_1(n) \geq (1+\epsilon)n$. Suppose that $T_2(n)$ grows faster than $T_1(n)$ by more than a log factor, i.e. $\lim_{n\to\infty} \frac{T_1(n)\log(T_1 n)}{T_2(n)} = 0$ (because turing machine simulation is $n \log n$).*
Then there exists a language in $DTIME(T_2(n))$ but not in $DTIME(T_1(n))$

The proof is very similar to space hierarchy, we just have the nlogn restriction from the universal turing machine.

Proof. Construct M that is $O(T_2(n))$ time bounded, has a fixed number of tapes (at least 5), and let L denote the language accepted by M. We show that L will not be in $DTIME(T_1(n))$.

M rejects all inputs of the form 1^k.
M on input of the form 1^k0x:
Mark out $T_2(|1^k0x|)$ on the work tape, so that we can count how many steps is taken. This allows us to reject when the simulation uses more time than $T_2(|1^k0x|)$
Simulate M_x on the same input, if the simulation attempts to use more than $T_2(|1^k0x|)$ time, then reject.
Else if M_x halts on the same input, then M accepts iff M_x did not accept the input.
In the above simulation, we assume that M_x uses only 2 tapes and uses the alphabet $\{0, 1, B\}$. Note that M is $O(T_2(n))$ space bounded, hence L is in $DTIME(T_2(n))$ by linear speedup theorem.

Suppose by way of contradiction that M' is a $T_1(n)$ time bounded machine that accepts L. Then we can construct a 2 tape machine M_x that simulates M and accepts L and is $cT_1(n)\log T_1(n)$ time bounded. WLOG we can assume M_x halts on all inputs.
Now we try to run M_x on M, let k be large enough such that $cT_1(|1^k0x|)\log T_1(|1^k0x|) < T_2(|1^k0x|)$, then we know that this simulation must complete, and M either accepts or rejects. Then M accepts 1^k0x iff M_x did not, this is a contradiction as 1^k0x should be accepted by both machines.
Thus there can not be a machine M' that accepts L and is T_1 time bounded.

□