CS365: Models of Computation (Advanced)

Felix Zhou [1]

April 7, 2021

[1]From Professor Eric Blais's Lectures at the University of Waterloo in Winter 2021

Contents

Chapter 1

Languages

1.1 Alphabets, String, and Languages

Just like how graphs are fundamental to graph theory, alphabets, strings, and languages are the main mathematical objects of study for computer science.

Definition 1.1.1 (Alphabet)
A nonempty finite set.

The elements of an alphabet are *symbols*.

Let $k \in \mathbb{Z}_+$ be fixed. While the definition of an alphabet allows for arbitrary symbols, we may without loss of generality, by relabelling if necessary, take "the" alphabet with k symbols to be

$$\Sigma_k := \{0, 1, \ldots, k-1\}.$$

Example 1.1.1
Two important alphabets are the unary alphabet $\Sigma_1 = \{0\}$ and the binary alphabet $\Sigma_2 = \{0, 1\}$.

Example 1.1.2
An alphabet with 3 symbols $\{\alpha, \beta, \delta\}$ and "the" alphabet with 3 symbols $\Sigma_3 = \{0, 1, 2\}$.

Definition 1.1.2 (String)
A string over the alphabet Σ is a finite sequence of symbols from Σ.

Example 1.1.3
Any integer represented in base 10 is a string over Σ_9.

We emphasize the importance that a string is finite.

Due to this definition, we may actually define a *metric* on the strings over an alphabet as follows: Let x, y be strings over the alphabet Σ. Write x_i to be the i-th symbol of x. Define $d(x, y)$ to be 2^{-j} where j is the largest integer such that $x_i = y_i$ for all $i < j$ and $d(x, y) = 0$ if $x = y$. It is crucial that x, y are finite sequences for this to be a metric.

The *length* of the string x, denoted $|x|$, is the number of symbols in x.

The *empty string* is the unique string of length 0, denoted ϵ. For $\epsilon \neq x$ of length $|x| = n$, we write

$$x = (x_1, x_2, \ldots, x_n)$$

so that x_i is the i-th symbol in x.

We can encode $\mathbb{Z}$ as strings over Σ_2 as follows: Fix $z \in \mathbb{Z}$. Take the binary representation of $|z|$ and append a 0 at the beginning of the string if $z < 0$. With the exception of 0, all non-negative integers begin with 1 in its most significant digit, hence we may easily distinguish negative numbers, 0, and positive numbers.

We can also encode $\mathbb{Q}$ as strings over Σ_2 with some wastefulness: Encode 0 as 0. Fix $0 \neq \frac{p}{q} \in \mathbb{Q}$ where $\gcd(p, q) = 1$. Take the binary representations of $|p|, |q|$. If one of the strings have smaller length, 0 pad its beginning to obtains strings x, y so that $|x| = |y|$. Encode

$$\frac{p}{q} \mapsto \begin{cases} 0xy, & \frac{p}{q} < 0 \\ xy, & \text{else} \end{cases}.$$

We may distinguish positive numbers from non-positive numbers using the parity of the length of encoding.

For text files, there is a alphabet Σ and the text file can be viewed as a string over Σ. Let $k := |\Sigma|$. We can uniquely represent each symbol of Σ with a 0-padded string over Σ_2 with length $\lceil \log_2 k \rceil$. Namely, the 0-padded binary representation of i for the i-th symbol. The text file can then be encoded by encoding each individual symbol and then concatenating the encodings. This is essentially the ASCII encoding.

Since we can encode text files, we can also encode graphs as text files of the adjacency list.

Specifically, take $\Sigma := \{0, 1, \ldots 9\} \cup \{:, \text{“,”}, \text{“\n”}\}$ to be the alphabet of the text file. Without loss of generality, the vertices are numbers $0, 1, \ldots, n-1$. Then an adjacency list can be encoded in a text file as follows

$$\begin{gathered} 0 : 1, 2, 3 \backslash \text{n} \\ 1 : 2, 4, 7 \backslash \text{n} \\ \ldots \\ n-1 : n-2, n-1 \backslash \text{n} \end{gathered}$$

We can then encode the text file using the method described above.

Each finite set of strings can be uniquely sorted lexicographically. We can thus store the sorted strings of a set in a comma separted text file and apply the prior encoding. Namely, if Σ is the alphabet from which our finite sets of strings are sourced, take $\Sigma' := \Sigma \cup \{\text{“,”}\}$ to be the alphabet of the text file, where “,” is ANY new separator symbol that does not belong to Σ. Let $w^{(1)}, \ldots, w^{(n)}$ be the lexicographically sorted set of strings. Then we may encode the set in a text file

$$w^{(1)}, \ldots, w^{(n)}$$

and then apply the prior encoding.

We cannot in general encode real numbers over strings over Σ_2. Indeed, there are uncountably many real numbers but only countably many strings over Σ_2.

Given any alphabet Σ, we can first encode its strings over Σ_2 by encoding each symbol as equally lengthed (0-padded) strings over Σ_2 by mapping the i-th symbol of Σ to the the binary representation of i. Thus the problem reduces to encoding strings over Σ_2 with strings over the unary alphabet. This is achieved by some enumeration of such strings (there are countably many) and mapping the i-th string to the unary string of length i.

Definition 1.1.3 (Language)
A language over the alphabet Σ is a set of strings over Σ.

Observe that we do note require a language to be *finite*, ie contain a finite number of strings. The *cardinality* of a finite language L is the number of strings in L, and is denoted $|L|$.

The *empty language* $\varnothing$ is the unique language with no strings.

The language containing all strings over Σ is denoted Σ^*.

The language containing all strings of length n over Σ is denoted Σ_n.

1.2 String Operations

Definition 1.2.1 (Concatenation)
The concatenation of two strings $x, y \in \Sigma^*$ is the string

$$xy := (x_1, \ldots, x_k, y_1, \ldots, y_\ell).$$

Remark that concatentation is associative in general but not in general commutative. Indeed, simply consider $x = 0, y = 1$.

Definition 1.2.2 (Power)
The n-th power of the string $x \in \Sigma^*$ is the string x^n obtained by concatenating n copies of x.

By convention, the 0-th power of x is $x^0 = \epsilon$.

For $n = 2$, we say x^2 is the *square* of x. Observe that $|x^2| = 2|x|$ so that $x^2 = x$ if and only if $x = \epsilon$.

Definition 1.2.3 (Primitive)
A string $x \in \Sigma^*$ is primitive if $x \neq y^n$ for any $y \in \Sigma^*$ and $n \geq 2$.

Definition 1.2.4 (Substring)
For any $x, y \in \Sigma^*$, we say x is a substring of y if there are $w, z \in \Sigma^*$ such that

$$y = wxz.$$

If there is $z \in \Sigma^*$ such that $y = xz$, then x is a *prefix* of y. Similarly, if there is $z \in \Sigma^*$ such that $y = zx$, then x is a *suffix* of y.

Observe that every string is a prefix, suffix, and substring of itself. Moreover, x being a substring of y implies $|x| \leq |y|$. So it x, y are substrings of each other, then $|x| = |y| \implies x = y$.

Definition 1.2.5 (Subsequence)
The string x is a subsequence of the string y if there exists indices

$$j_1 < j_2 < \ldots, j_{|x|}$$

such that $x_i = y_{j_i}$ for each $i = 1, 2, \ldots, |x|$.

The length of the shortest string over Σ_2 containing all 16 strings of length 4 as subsequences is 8. Indeed, it must be at least 8 since it must contain 0000, 1111 as subsequences, Moreover, the string 01010101 attains the desired lower bound.

Definition 1.2.6 (Reversal)
The reversal of a string x is the string

$$x^R = (x_n, x_{n-1}, \ldots, x_n).$$

We say the string x is a palindrome if $x = x^R$.

Let $x \in \Sigma_2^*$. Observe that x contains a palindromic substring of length at least 4 if and only if it contains a palindromic substring of length 4 or 5. Notice that if x does not contain a palindromic substring of length at least 4, then any palindromic substring of xz necessarily contains z. Thus we may iteratively find a lengthwise maximal string containing no palindromic substring of length at least 4. The two strings achieving the maximum are 00010111 and 11101000.

1.3 Language Operations

The usual set operations of union, intersection, set difference, symmetric difference, and complement all apply to languages.

Definition 1.3.1 (Concatenation)
The concatenation of two languages A, B over Σ^* is the language

$$AB := \{xy : x \in A, y \in B\}.$$

Observe that $A\{\epsilon\} \neq A\varnothing$ since the latter is actually the empty language by definition.

Similary to strings, we can use concatenation to define the powers of a language.

Definition 1.3.2 (Power)
The n-th power of a language L over Σ is the language

$$L^n := \{x^{(1)} \dots x^n : x^{(i)} \in L\}.$$

By convention, $L^0 = \{\epsilon\}$.

We can also use concatenation to define another operation.

Definition 1.3.3 (Star)
For a language L, define

$$L^* := \bigcup_{n \geq 0} L^n.$$

The star operation is also called the *Kleene star*. Notice that $A^* \neq \varnothing$ as $A^0 := \{\epsilon\} \subseteq A^*$ and A^* is finite if and only if $A = \varnothing, \{\epsilon\}$.

1.4 Countability & Uncountability

Assuming the axiom of choice, we can define a total order on the collection of all sets by

$$|S| \leq |T|$$

if and only if there exists and injective map $\phi : S \to T$.

Equivalently, $|S| \leq |T|$ if and only if there exists a surjective map $\psi : T \to S$.

Two sets S, T have the same cardinality, denoted

$$|S| = |T|$$

if and only if $|S| \leq |T|, |T| \leq |S|$.

From the Cantor-Shröder-Bernstein theroem, an equivalent definition is the existance of a bijective map between S, T.

It is easy to check that this defines a partial order on the collection of all sets. However, to prove that every pair of sets is comparable requires the axiom of choice.

Definition 1.4.1 (Finite)
A set is finite if

$$|S| = |[n]|$$

for some $n \in \mathbb{N}$.

The cardinality of natural numbers is denoted $|\mathbb{N}| = \aleph_0$.

Definition 1.4.2 (Countable)
The set S is countable if

$$|S| \leq |N| = \aleph_0.$$

Notice that countable sets can be either finite or infinite. The set of all strings over any alphabet is countable.

Proposition 1.4.1
For any alphabet Σ, the set Σ^* of strings over Σ is countable.

Proof
We will define an injection $\phi : \Sigma^* \to \mathbb{N}$.

Fix $k \in \mathbb{N} \cup \{0\}$. Notice that there are only a finite number of strings of length k. In fact, there are $|\Sigma|^k$ such strings.

For each k, enumerate the strings of length k

$$w_{k,1}, w_{k,2}, \ldots, w_{k,|\Sigma|^k}.$$

We then "flatten" these indices along the natural numbers to obtain the following injection

$$\phi(w_{k,j}) := j + \sum_{i=0}^{k-1} |\Sigma|^k.$$

We say a set is *uncountable* if it is not countable.

Definition 1.4.3 (Power Set)
For any set S, the power set $\mathcal{P}(S)$ is the set of all subsets of S.

Theorem 1.4.2 (Cantor's Theorem)
For any set S,

$$|\mathcal{P}(S)| \not\leq |S|.$$

Proof
Suppose towards a contradiction that there is some surjection $\psi : S \to \mathcal{P}(S)$.

We claim that there is some $T \in \mathcal{P}(S) \setminus \psi(S)$, which would contradiction the surjectivity of ψ. Let $T \in \mathcal{P}(S)$ be defined as

$$T := \{s \in S : s \notin \psi(s)\}.$$

Clearly, $T \in \mathcal{P}(S)$. Fix $s \in S$. We have $T \neq \psi(s)$ since $s \in T \Delta \psi(s)$. This concludes the proof.

Corollary 1.4.2.1
If S is countably infinite, $\mathcal{P}(S)$ is uncountably infinite.

Proof
If $\mathcal{P}(S)$ is not uncountably infinite, then $|\sqrt{}(S)| \leq |S|$, which contradicts Cantor's theorem.

Corollary 1.4.2.2
For any alphabet Σ, the set of languages over Σ is uncountable.

Proof
The set of all strings Σ^* is countably infinite by our prior work. By definition, the set of languages over Σ is exactly $\mathcal{P}(\Sigma^*)$.

The result follows by Cantor's theorem.

Chapter 2

Finite Automata

2.1 Deterministic Finite Automata

A DFA is an abstract machine whose behavior can be described by directed graph.

Nodes indicate the state of the machine and arcs indicate the state transitions when the machine reads the next symbol in a string.

Definition 2.1.1 (Deterministic Finite Automaton)
An abstract machine described by

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

Q is a finite set of states.
Σ is the input alphabet.
$\delta : Q \times \Sigma \to Q$ is the transition function.
$q_0 \in Q$ is the start state.
$F \subseteq Q$ is the set of accepting states.

A machine *accepts* a string if and only if it ends in one of the accepting states.

Definition 2.1.2 (Accepts)
The DFA $M = (Q, \Sigma, \delta, s, F)$ accepts the string $w \in \Sigma^n$ if and only if there is a sequence of states $r_0, r_1, \ldots, r_n \in Q$ where $r_0 = q_0$, $r_i = \delta(r_{i-1}, w_i)$ for each $i = 1, 2, \ldots, n$ and $r_n \in F$.

The *size* of the DFA M is the number of states in Q.

The DFA at the top of the section accepts the string 0010110. The certificate is the sequence of states

$$q_0, q_0, q_1, q_2, q_0, q_1, q_2.$$

A string is *rejected* by a DFA if it is not accepted.

Definition 2.1.3 (Language Recognized)
The language recognized by a DFA M is

$$L(M) := \{x \in \Sigma^* : M \text{ accepts } x\}.$$

2.1.1 Regular Languages

Definition 2.1.4 (Regular Language)
The language is regular if there is a DFA M such that $A = L(M)$.

By swapping the accepting states with non-accepting states of the DFA, we see that $\bar{A}$ is regular if and only if A is regular.

Proposition 2.1.1
Every finite language is regular.

Proof
Let A be a finite language over Σ. Let M be the DFA consisting of a trie. The accepting states are at nodes representing the last symbol in each string of A.

Proposition 2.1.2
There are languages that are not regular.

Proof
We have shown the set of all languages is uncountably infinite. It suffices then to show that the set of regular languages is countable.

Fix $n \in \mathbb{Z}_+$. There are only a finite number of DFAs of size n as there are only a finite number of directed graphs of size n, each of which gives rise to at most 2^n (finite) DFAs depending accepting states.

But since $\mathbb{Z}_+$ is countable, so is the number of DFAs.

2.1.2 Separating Words Problem

Proposition 2.1.3
For every two $x \neq y \in \Sigma^n$, there is a DFA with at most $n+2$ states that accept x but not y.

Proof
Consider the DFA M which which is a dipath with states

$$\epsilon = x_0, x_1, x_2, \ldots, x_n$$

with x_n the accepting state. Add another "failure" state f which transitions to itself always. We transition $x_{i-1} \mapsto x_i$ only if we see x_i. Otherwise, we transition to f.

Clearly M accepts x. Since $y \neq x$, there is some minimal index i such that $y_i \neq x_i$. Then $x_{i-1} \mapsto f$ upon reading y and M will not accept y.

Lemma 2.1.4
If $m < n \in \mathbb{Z}_+$, there is some prime $p \in O(\log n)$ for which

$$m \not\equiv n \mod p.$$

Proof
Let $p_1 = 2, p_2 = 3, \ldots, p_\ell$ be first ℓ primes.

Suppose $m \equiv n \mod p_j$ for all $1 \leq j \leq \ell$. Then by the chinese remainder theorem,

$$m \equiv n \mod \prod_{j=1}^{\ell} p_j.$$

It can be proven using the first Chebychev function ϑ that the product of primes at most x for x sufficiently large is strictly greater than 2^x. See `https://math.stackexchange.com/a/2645232/734472`.

Thus by choosing $x \in O(\log n)$, some prime at most x exists.

Proposition 2.1.5
For every $x \in \Sigma^n, y \in \Sigma^m$ where $m < n$, there is a DFA with $O(\log n)$ states accepting x but not y.

Proof
Consider $b(x), b(y)$, the binary string representation of $|x|, |y|$. The idea is to track length of the string we have reaAd so far modulo p for some $p \in \mathbb{Z}_+$.

If $p \in O(\log n)$, then it is clear that the DFA has $O(\log n)$ states. By the previous lemma, it suffices to take $p \in O(\log n)$ are required.

It can also be shown that we can separate two length n words with $o(n)$ states. In fact, the best bound we know, by Zachary Chase, is $\tilde{O}(n^{\frac{1}{3}})$. On the other hand, the only lower bound is $\Omega(\log n)$. Can we improve either?

2.2 Nondeterministic Finite Automata

Definition 2.2.1 (Nondeterministic Finite Automaton)
An abstract machine described by

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

Q is a finite set of states.
Σ is the input alphabet.
$\delta : Q \times \Sigma \cup \{\epsilon\} \to \mathcal{P}(Q)$ is the transition function.
$q_0 \in Q$ is the start state.
$F \subseteq Q$ is the set of accepting states.

An ϵ-*transition* in the NFA M is a transition of the form $\delta(q, \epsilon)$ for some $q \in Q$. The NFA can choose to do an ϵ-transition any time, without processing any symbol from its input.

Definition 2.2.2 (ϵ-Reachable)
A state $r \in Q$ is ϵ-reachable from $q \in Q$ in an NFA M if there are states $s_i, 0 \leq i \leq \ell$ that satisfy

$$s_0 = q, s_\ell = r, s_i \in \delta(s_{i-1}, \epsilon), i \in [\ell].$$

We say the state $r \in Q$ is ϵ-*reachable* from a set $S \subseteq Q$ of states if it is ϵ-reachable from some state $q \in S$.

Definition 2.2.3 (Accept)
The NFA M accepts $w \in \Sigma^n$ if and only if there are states $r_i, 0 \leq i \leq n$ such that r_0 is ϵ-reachable from q_0, r_i is ϵ-reachable from $\delta(r_{i-1}, x_i)$ in M for each $i \in [n]$ and $r_n \in F$.

Definition 2.2.4 (Language Recognized)
The language recognized by the NFA N is

$$L(N) = \{x \in \Sigma^* : N \text{ accepts } x\}.$$

Let M be an NFA. Let N be the NFA obtained from M by adding a new state f and an epsilon transition from each accepting state of $F(M)$ of f. The only accepting state of N is f. Hence without loss of generality, we can always assume that an NFA has exactly 1 accepting state.

Consider the complete digraph on two states q_1, q_2 with arcs all being epsilon transitions. Supppose $F = \{q_1\}$. Then we can ϵ-transition from either state to the other and the NFA accepts all strings regardless of which state is the accepting state.

Proposition 2.2.1
Every regular language can be recognized by an NFA.

Proof
Let N be the DFA corresponding to the language L. let M be the NFA obtained from N by setting

$$\delta(M)(q, s) := \{\delta(N)(q, s)\}$$

and

$$\delta(M)(q, \epsilon) := \varnothing$$

for all $q \in Q$.

Then M accepts a string w if and only if N accepts w.

Let N_1, N_2 be two NFAs. Without loss of generality, both have exactly 1 accepting state. Consider N obtained from N_1, N_2 as follows: Let s, a be 2 new states. f is the failure state, from which we can only transition to itself. s is the new start state, with two epsilon transitions to the state states of N_1, N_2, and transitions to the failure state otherwise. The accepting states of N is the union $F(N_1) \cup F(N_2)$.

We can also do the same "union" operation for two DFAs M_1, M_2. Since we already showed that regular languages are closed under complements, it suffices to show that the intersection

of two regular languages is regular, then apply DeMorgan's Law. We can create the DFA with states $Q_1 \times Q_2$, for which $delta(q_1, q_2) = (\delta_1(q_1), \delta_2(q_2))$. The set of accepting states is $F_1 \times F_2$.

2.3 Rabin-Scott Theorem

Lemma 2.3.1
For every NFA N, there is a DFA M that recognizes $L(N)$.

For a state q, let $c(q)$ be the set of states which is ϵ-reachable from q. For $R \subseteq Q$,

$$c(R) := \bigcup_{q \in R} c(q).$$

Proof
We construct a new DFA M.

Let the new states be $Q' := \mathcal{P}(Q)$, the powerset of states of N.

Define the new transition function $\delta' : \mathcal{P}(Q) \times \Sigma \to \mathcal{P}(Q)$ given by

$$(R, s) \mapsto \bigcup_{q \in R} c(\delta(q, s)).$$

The new state start is the set $q_0' := \{q_0\} \cup c(q_0)$. Finally, $F' \subseteq Q'$ is the set of accepting states where

$$F' := \{R \in Q' : Q \cap F \neq \varnothing\}.$$

Suppose N accepts string w. Let the certificate $q_0, q_1, \ldots, q_\ell$ be some sequence of states.

Let $r_0 := \{q_0\}$. If $\ell = 0$, then $q_\ell \in r_0 \cap F$, and M accepts w. Otherwise,

$$q_1 \in c(\delta(q_0, w_1)) \subseteq \bigcup_{q \in r_0} c(\delta(q, w_1)) =: r_1.$$

Inductively, suppose that $q_i \in r_i$. If $i = \ell$, then $q_\ell \in r_i \cap F$ and M accepts w. Otherwise,

$$q_{i+1} \in c(\delta(q_i, w_{i+1})) \subseteq \bigcup_{q \in r_i} c(\delta(q, w_{i+1})).$$

It follows by induction that w is accepted by M. Conversely, suppose w is accepted by N. Let $r_0, r_1, \ldots, r_\ell$ be the certificate of acceptance.

Put $s_0 := q_0$. The transition

$$(r_0, w_1) \mapsto \bigcup_{q \in r_0} c(\delta(q, w_1)) = r_1$$

means that r_1 is the set of states reachable from q_0 by one transition consuming w_1 and any number of ϵ-transitions. In general,

$$(r_{i-1}, w_i) \mapsto \bigcup_{q \in r_{i-1}} c(\delta(q, w_i)) = r_i$$

can be translated as starting from some state $q \in r_{i-1}$, do exactly one transition consuming w_i, then performing some finite number of ϵ-transitions.

Thus each transition in N is translated to a sequence of transitions in M: a non-ϵ-transition, followed by several ϵ-transitions. Since $r_\ell \cap F \neq \varnothing$,

Thus w is accepted by M as well.

Theorem 2.3.2
The set of languages that can be recognized by DFAs is the same as the set of languages that can be recognized by NFAs.

Proof
We have proven earlier that every regular language is recognized by some NFA.

The previous lemma shows the converse.

2.4 Regular Expressions

Regular expressions are used to describe strings succintly.

Definition 2.4.1 (Regular Expression)
A regular expression over the alphabet Σ is a nonempty string r over

$$\Sigma \cup \{|, *, (,), \epsilon, \varnothing\}.$$

such that falls into one of the following conditions:

- $r = \varnothing$
- $r = \epsilon$
- $r \in \Sigma$
- $r = (s|t)$ for regex s, r
- $r = (st)$ for some regex s, t
- $r = (s^*)$ for some regex s.

In the definition, we assume that the special symbols are not included in Σ. For clarity, we often omit the parentheses. We would then apply the operations in order of precedence: star, concatenation, unions.

2.4.1 Language of Regular Expressions

Definition 2.4.2 (Language Described)
The language described by a regular expression r over Σ is the language $L(r) \subseteq \Sigma^*$ where

1. $L(r) = \varnothing$ if $R = \varnothing$
2. $L(r) = \{\epsilon\}$ if $r = \epsilon$
3. $L(r) = \{r\}$ if $r \in \Sigma$
4. $L(r) = L(s) \cup L(t)$ if $r = (s|t)$ for some regex s, t
5. $L(r) = L(s)L(t)$ if $r = (st)$ for som regex s, t
6. $L(r) = L(s)^*$ if $r = (s^*)$ for some regex s.

The regex 0^*110^* describes the language

$$\{0\}^*11\{0\}^*.$$

2.4.2 Short Regular Expressions

Definition 2.4.3 (Length)
The length of a regular expression r over Σ is the number of symbols from Σ in the string r.

With the exception of the star operator repeated, the length of the string of the regular expression is linearly proportional with the length of the regex.

Problem 1
Let A_k denote the language over Σ_k which contains all nonempty strings in which no two consecutive symbols are identical.
What is the length of the shortest regular expression that can represent A_k.

The best bound we are aware is $O(k^{2.17})$. It is conjectured that the tightest bound is $O(k^2)$.

Let us consider a similar problem stated within linear algebraic terms.

Consider matrices $A, B \in M_n(\mathbb{F})$.

For $k \geq 1$, a *word* in A and B of length k is a product of the form $C_1 \cdots C_k$ for $C_i \in \{A, B\}$. By convention, the only word in A and B of length 0 is I_n.

For our purposes, an *algebra* is a vector space over a field K with a multiplication operation.

Thus the vector space $M_n(\mathbb{F})$ is an algebra with the matrix multiplication operation.

The *algebra generated by A and B* is the subspace

$$\mathcal{A} := \text{span}\{\text{words in } A \text{ and } B \text{ of length } k \geq 0\}.$$

The *spanning length* of A and B is the minimal positive integer ℓ such that

$$\mathcal{A} := \text{span}\{\text{words in } A \text{ and } B \text{ of length } 0 \leq k \leq \ell\}.$$

It is not hard to see that $\mathcal{A}$ is an algebra since words consisting of A, B concatenated yield more words over A, B.

Lemma 2.4.1
Any word of length n^2 can be expressed as a linear combination of words at most $n^2 - 1$

Proof
To begin, we note that since $\mathcal{A} \subseteq M_n(\mathbb{F})$, $\dim \mathcal{A} \leq \dim M_n(\mathbb{F}) = n^2$.

Let A_0 be an arbitrary word of length n^2.

It suffices to show that A_0 is in the span of words of lengths at most $n^2 - 1$

Let $w_i, i \in \mathbb{N}$ be any word of length i.

Consider the following algorithm:

step 1: take $\{w_1\} \cup \{A_0\}$, if they are dependent we are done else

step 2: take $\{w_1, w_2\} \cup \{A_0\}$, so that w_2 is not in the span of our previous set. If the union is dependent we are done else

step 3: take $\{w_1, w_2, w_3\} \cup \{A_0\}$, so that w_3 is not in the span of our previous set. If this union is dependent we are done else

step 4: repeat

The algorithm terminates at most with w_{n^2-1} since the size of an independent set of words is at most n^2.

Thus, A_0 can be expressed as a linear combinations of words at most $n^2 - 1$ as desired and we conclude the proof.

Proposition 2.4.2
The spanning length of A and B is at most n^2.

Proof
We argue by induction.

Let P_m be the statement that we can express all words of length m in terms of words at most n^2.

(1) Base Case

Let A_{n^2+1} be any word of length $n^2 + 1$.

then there is a word C of length n^2 such that either $A_{n^2+1} = CA$ or $A_{n^2+1} = CB$.

But by our lemma, C can be expressed as linear combinations of words at most $n^2 - 1$.

Say

$$C = \alpha_0 w_0 + \dots \alpha_{n^2-1} w_{n^2-1}$$

But then CA or CB can be expressed as words of length at most n^2 and we are done.

(2) Inductive Case

Suppose now, inductively that P_{m-1} holds where $m > n^2 + 1$.

Let A_m be any word of length m.

There is a word C of length $m-1$ such that either $A_m = CA$ or $A_m = CB$.

By supposition, we can express C with words at most n^2 and by the lemma, further expand this into words at most $n^2 - 1$.

But then CA or CB can be expressed as words of length at most n^2 and we are done.

By induction, we conclude that the spanning length of $\mathcal{A}$ is at most n^2.

2.5 Kleene's Theorem

Our goal is to prove that the class of regular languages is precisely the class of languages described by regular expressions!

Theorem 2.5.1 (Kleene)
The language A is regular if and only if it can be described by a regular expression.

For the forward direction, it suffices to show that every language can be described with a regular expression.

This direction is very intuitive. A language described by a regular expression is recursively constructed from other languages described by smaller regular expressions. Thus we intuitively think about combining NFAs.

Lemma 2.5.2
For every regular expression r, the language $L(r)$ described by r can be recognized by an NFA.

Proof
We argue by structural induction on r.

In the three base cases if $r = \varnothing, \epsilon, \sigma \in \Sigma$, it is straightforward to see that some NFA recognizes the languages described by r.

Suppose $r = (s|t)$ for some regular expressions s, t. Let M_s, M_t be NFAs which recognize $L(s), L(t)$, respectively. Then the NFA which has new starting state and epsilon transitions to the starting states of M_r, M_t. This NFA recognizes $L(r)$.

Now consider $r = (st)$ for some regular expressions s, t. Again, let M_s, M_t be NFAs which recognize $L(s), L(t)$, respectively. connect $M_s \to M_t$ by adding ϵ-transition from the accepting states of M_s to the starting state of M_t and setting the accepting states of M_s to $\varnothing$.

Finally, suppose $r = (s)^*$ for some regular expression r. Let M_s be a NFA which recognizes $L(r)$. Then, if none exist, add an ϵ-transition from each accepting state to the starting state.

By structural induction, $L(r)$ can be recognized by some NFA.

The converse is alot less constructive and we rely heavily on induction.

Lemma 2.5.3
Every language L that can be recognized by a DFA can be described with a regular expression.

Proof
Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA. Without loss of generality, relabel

$$Q = [m].$$

Pick any $p, q \in Q$ and define $L_{q \to r}(k)$ as the language of strings which induce paths from $q \to r$ in M while using only the states $[k]$ as intermediate states. We by induction on k that for all $p, q \in Q$,

$$L_{q \to r}(k)$$

can be described by some regex.

Indeed, if $k = 0$, then the result is clear as we cannot use any intermediate states so either $p = q$, in which case our language contains ϵ and possibly the (finite) loop symbols at p, or $p \neq q$, and includes the single arc symbols between p, q. Since finite languages are easily described by regex, $L_{p \to q}(0)$ can be described by a regex.

Now suppose $k \geq 0$. Observe that

$$L_{p \to q}(k+1) = L_{p \to q}(k) \cup L_{p \to k+1}(k) L_{k+1 \to k+1}(k)^* L_{k+1 \to q}(k).$$

The latter is the language which induced by $p(k+1)$-dipaths, then $k+1$-loops, and $(k+1)q$-dipaths.

But each of the constituant languages have been proven to be describable by a regular expression. Unions and concatenations are also easily expressed through regex. Hence by induction we are done.

Observe that $L(M) = \bigcup_{q \in F} L_{q_0 \to q}(m)$. Hence it is indeed describable by a regex.

This concludes the proof.

2.6 Pumping Lemma

A language is *non-regular* if it is not a regular language.

We saw many ways to prove that a language is regular. How about the converse?

Proposition 2.6.1
The language $L = \{0^n 1^n : n \geq 0\}$ is not regular.

Proof
Suppose towards a contradiction that there is some DFA $M = (Q, \Sigma_2, \delta, q_0, F)$ for which $L(M) = L$. Put $m := |Q|$. Choose $n >> m$ and consider the path induced by the string $0^n 1^n$. Let P_0 be the arcs taken by 0^n and P_1 the arcs taken by 1^n.

Since $n >> m$, P_0 necessarily contains a dicycle. Write

$$P_0 = WCW'$$

for some dicycle C. But then the string corresponding to the path

$$WC^k W' P_1$$

is also accepted by M for all $k \geq 0$.

In particular, there is some $N > n$ for which

$$1^N 0^n \in L(M) \setminus L.$$

By contradiction, L is not regular.

Lemma 2.6.2 (Pumping Lemma for Regular Languages)
For every regular language L, there is a number p such that for any $s \in L$ of length at least p, we can write

$$s = xyz$$

where $|y| > 0, |xy| \leq p$, and for each $i \geq 0$, $xy^i z \in L$.

A language L satisfying the condition of the lemma is said to satisfy the *pumping property*.

A value p for which L satisfies the pumping property is known as a *pumping length* of L.

Proof
First observe that if L is finite, then the lemma trivially holds. Indeed, we can simply pick p to be longer than the longest string in L and the pumping condition vacuously holds.

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA which recognizes L. Set $m := |Q|$. Since L is infinite, we may choose some string $w \in L$ such that $|w| > m$.

There is a unique walk W in M starting from q_0 and ending at some accepting state in F, whose arcs correspond to the symbols of w in the same order.

Since $m > q$, there is necessarily a dicycle C contained in W. Without loss of generality, let us assume that C is the first dicycle. That is, the last state in C is the first repeated state in W. Observe then that C ends at most at the $m+1$-st state in W.

Thus we can decompose W as

$$W = PCP'$$

where P, P' are the states visited before C and after P', respectively.

Since M accepts w, W ends in an accepting state. But then so does PC^iP' for all $i \geq 0$. Let x be the string corresponding to P, y the string corresponding to C, and z the string corresponding to P'.

We have $|y| > 0$ since the shortest dicycle contains at least 1 arc. Moreover, $|xy| \leq p$ by the choice of C. Finally, by construction,

$$\forall i \geq 0, xy^i z \in L.$$

The idea for the pumping lemma is that DFAs have no sense of "memory", thus the only way to produce longer strings is through "hardcoded" dicycles. The dicycles are "regularities" within the language which we know to exist through the pumping lemma, but may not be able to pinpoint. For example, for a machine to recognize the language $\{0^n 1^n : n \geq 0\}$,

it must somehow keep track of how many 0's it has seen so far and to "expect" the same amount of 1's. This is not possible when our dicycles are "hardcoded".

The contrapositive of the pumping lemma is useful for showing that some languages are non-regular.

Lemma 2.6.3 (Pumping Lemma; Contrapositive)
If for every integer p, there is a string $s \in L$ of length at least p such that for every decomposition $s = xyz$ with $|y| > 0$ anad $|xy| \leq p$, there exists a value $i \geq 0$ for which $xy^iz \notin L$, then L is not a regular language.

2.7 Non-Regular Languages

Proposition 2.7.1
The language $L := \{0^n1^n : n \geq 0\}$ is not regular.

Proof
Pick $p \geq 1$ and choose $s = 0^p1^p$. Then $s \in L$ and $|s| \geq p$.

Any decomposition $s = xyz$ with $|xy| \leq p$ and $|y| > 0$ must then have $y = 1^k$ for some $k \leq p$. But

$$xy^2z$$

then has an uneven number of 0's and 1's and cannot be in L.

By the pumping lemma, L is not regular.

Proposition 2.7.2
The language

$$L_< := \{0^m1^n : n > m \geq 0\}$$

is not regular.

Proof
Pick $p \geq 1$ and choose $s = 0^p1^{p+1}$. Then $s \in L$ and $|s| \geq p$.

Any decomposition $s = xyz$ with $|xy| \leq p$ and $|y| > 0$ must then have $y = 1^k$ for some $k \leq p$. But

$$xy^2z$$

has at least as many 0's as 1's and hence does not belong to L.

It follows by the pumping lemma that L is not regular.

Proposition 2.7.3
The language

$$L_> := \{0^m 1^n : m > n \geq 0\}$$

is not regular.

Proof
Pick $p \geq 1$ and choose $s = 0^p 1^{p-1}$. Then $s \in L$ and $|s| \geq p$.

Any decomposition $s = xyz$ with $|xy| \leq p$ and $|y| > 0$ must then have $y = 1^k$ for some $k \leq p$. But

$$xy^0 z = xz$$

has at most as many 0's as 1's and hence does not belong to L.

It follows by the pumping lemma that L is not regular.

There also languages over even the unary alphabet that are not regular.

Proposition 2.7.4
The language

$$L := \left\{0^{2^n} : n \geq 0\right\}$$

is not regular.

Proof
Pick $p \geq 1$ and choose $s = 0^{2^p}$. Then $s \in L$ and $|s| \geq p$.

Any decomposition $s = xyz$ with $|xy| \leq p$ and $|y| > 0$ must then have $y = 1^k$ for some $k \leq p$. But

$$\begin{aligned} 2^p &= |s| \\ &< |xy^2 z| \\ &\leq |s| + p \\ &= 2^p + p \\ &< 2^{p+1}. \end{aligned}$$

It follows by the pumping lemma that L is not regular.

2.7.1 Limitations of the Pumping Lemma

Proposition 2.7.5
The language

$$L_{\text{pal}^+} = \{ww^R x \in \Sigma_2^* : |w| \geq 1\}$$

satisfies the pumping property but is not regular.

Thus the pumping condition is a necessary but NOT sufficient condition to garantee regularity. The Myhill-Nerode thereom provides a characterization of regular languages.

Given a language L and a pair of strings x, y, define a *distinguishing extension* to be a string z such that exactly one of the two strings xz, yz belongs to L.

Define a relation R_L on strings by the rule that xR_Ly if there is no distinguishing extension for x, y. This is an equivalence relation on strings and thus partitions L into equivalence classes.

Theorem 2.7.6 (Myhill-Nerode)
L is regular if and only if R_L has a finite number of equivalence classes, and moreover the number of states in the smallest DFA recognizing L is equal to the number of equivalence classes in R_L.

2.8 Properties of Regular Languages

In the same sense of how continuous functions are rare but have extremely nice structure, most languages are not regular but those that are enjoy many structural properties.

2.8.1 Closure under Language Operations

We have previously shown in these notes that if A, B are regular languages, then $\bar{A}, A \cap B, A \cup B, AB, A^*$ are regular as well.

Since we can express

$$A \setminus B = A \cap \bar{B}$$
$$A \Delta B = (A \setminus B) \cup (B \setminus A),$$

set difference and symmetric difference also preserve regularity.

Our goal is to introduce more operations which preserve the regularity of languages.

Definition 2.8.1 (Language Expansion)
Then expansion of the language A over Σ is the language

$$A^{\uparrow} := \{x \in \Sigma^* : \exists y \in A, |y| = |x|, H(x, y) \leq 1\}.$$

Here $H(\cdot, \cdot)$ indicates the Hamming distance.

Proposition 2.8.1
For every regular language A, the language $A^{\uparrow}$ is also regular.

Proof
Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA recognizaing A. The idea is to first turn M into an equivalent NFA with no ϵ-transitions. Then we somehow allow this NFA a "one-time wildcard" arc which allows it to transition from state $q \in Q$ to any neighbour of q.

Indeed, first fix some state $q \in Q$ and $s \in \Sigma$. Suppose $\delta(q, s) = q'$. Create $|\Sigma| - 1$ copies of M, say $\{M_{q,t} : t \in \Sigma, t \neq s\}$ and ignore their starting states. Add an arc from M to $M_{q,t}$ through the transition

$$(q, t) \mapsto q'.$$

Then, repeat this operation for every $q \in Q, s \in \Sigma$ and let M' be the DFA obtained this way.

Clearly, if $x \in A$, then $x \in L(M')$. Moreover, if $x \in A^{\uparrow} \setminus A$, we can find some $y \in A, H(x, y) = 1$. We can take the path induced by y in M up until the different symbol, and transition to a copy of M where the remaining path induced by y leads to a final accepting state.

If an accepting path does not leave M, then it is an accepting path of M. Suppose now that our path leaves M, say it transitions from q in M to some state q' of $M_{q,t}$. Then there was some $s \in \Sigma$ for which

$$\delta(q, s) = q'.$$

The rest of the accepting path (less this one arc) is in $M_{q,t}$. Moreover, tracing the same path in M yields an accepting path in M. This the two paths "differ by a one-tiem wildcard arc".

See Figure 2.1.

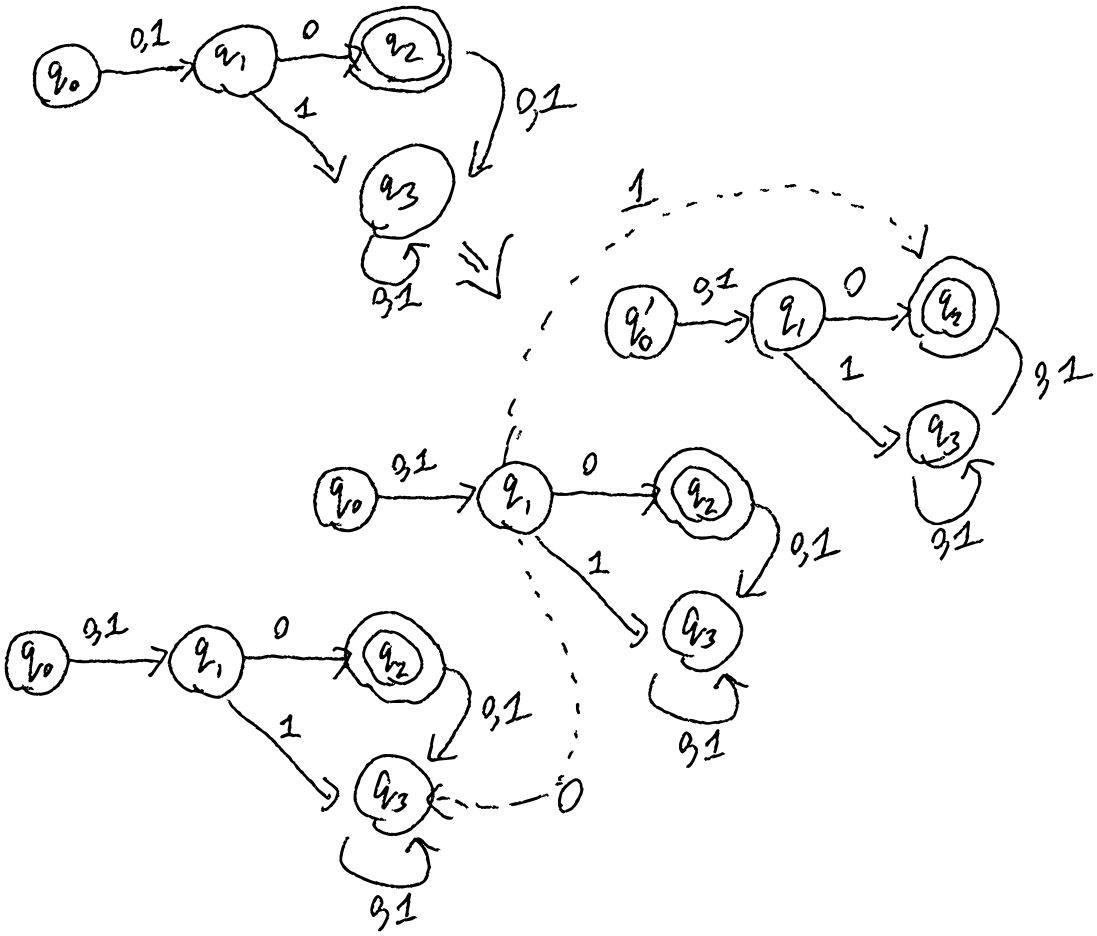

Figure 2.1: An illustration of the DFA corresponding to expansion.

2.8.2 Topological Separation

Theorem 2.8.2
There exists a language L over Σ_2 such that every infinite language $L' \subseteq L$ is non-regular.

Proof
Simply take $L := \{0^n 1^n : n \geq 0\}$. Any infinite subset $L' \subseteq L$ thus consists of some infinite subset of $Z \subseteq \mathbb{Z}_+$ such that

$$L' = \{0^n 1^n : n \in Z\}.$$

Pick $p \geq 1$ and choose the smallest $k \in Z, k \geq p$. Then $s = 0^k 1^k \in L$ and $|s| \geq p$.

Any decomposition $s = xyz$ with $|xy| \leq p$ and $|y| > 0$ must then have $y = 1^\ell$ for some $\ell \leq p$. But

$$xy^2z$$

then has an uneven number of 0's and 1's and cannot be in L'.

By the pumping lemma, L' is not regular.

Proposition 2.8.3
For every infinite regular language L, there is an infinite language $L' \subsetneq L$ that is regular.

This is sort of a cheat proof.

Proof
Take any nonempty finite subset $K \subseteq L$. Then K, L are both regular and hence

$$L' := L \setminus K$$

is an infinite regular language.

Theorem 2.8.4
For every infinite regular language L, there is a regular language $L' \subseteq L$ such that L' and $L \setminus L'$ are both infinite regular languages.

Proof
Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA which recognizes L. Since L is finite, M cannot be acyclic or else are only a finite number of paths to accepting states.

Let q be any state which is in some dicycle. For $i = 0, 1$, put

$$L_i = \{w \in L : \text{the number of times the path induced by } w \text{ visits } q \text{ has parity } i\}.$$

Then $L = L_0 \dot{\cup} L_1$.

Moreover, L_i is regular! The idea is to take the DFA

$$M_i = (Q \times \{0, 1\}, \Sigma, \delta', (q_0, 0), F \times \{i\}).$$

Thus the extra coordinate tracks the parity of times we visited q.

2.8.3 Testing Emptiness & Equivalence

Proposition 2.8.5
Given a DFA $M = (Q, \Sigma, \delta, q_0, F)$, we can determine whether $L(M) = \varnothing$ or not in $O(|Q| \cdot |\Sigma|)$ time.

Proof
If $F = \varnothing$, the task is trivial.

Now, observe that if $L := L(M) \neq \varnothing$, there is some string $s \in L$ of minimal length. The path induced by s must be simple, or else we can can find a strictly shorter string accepted by M. Conversely, if there is a simple path from q_0 to an accepting state, L is clearly not the empty language.

It follows that we can use any graph exploration algorithm (ie DFS, BFS, etc) which runs in linear time in order to decide if $L = \varnothing$. There are $|Q|$ states and $|Q| \times |\Sigma|$ arcs, hence the run-time guarantee holds.

Proposition 2.8.6
Givens DFA $M_1 = (Q_1, \Sigma, \delta_1, q_{1,0}, F_1)$ and $M_2 = (Q_2, \Sigma, \delta_2, q_{2,0}, F_2)$, we can determine whether $L(M_1) = L(M_2)$ or not in $O(|Q_1| \cdot |Q_2| \cdot |\Sigma|)$ time.

Proof
It suffices to reduce this problem to testing emptiness in a DFA with $|Q_1| \cdot |Q_2|$ states.

Observe that

$$\begin{aligned} L(M_1) = L(M_2) &\iff L(M_1) \setminus L(M_2) = \varnothing, L(M_2) \setminus L(M_1) = \varnothing \\ &\iff L(M_1) \cap \overline{L(M_2)} = \varnothing, L(M_2) \cap \overline{L(M_1)} = \varnothing. \end{aligned}$$

Hence we have reduced the problem to testing emptiness.

Now, we have shown earlier in these notes that a DFA recognizing $\overline{L(M_i)}$ is

$$(Q_i, \Sigma, \delta_i, q_{i,0}, Q_i \setminus F_i).$$

Moreover, we have shown that a DFA recognizing the $L(M_i) \cap \overline{L(M_{1-i})}$ is

$$(Q_i \times Q_{1-i}, \Sigma, \delta_i \times \delta_{1-i}, (q_{i,0}, q_{1-i,0}), F_i \times (Q_{1-i} \setminus F_{1-i})).$$

Thus with two iterations of our algorithm to test emptiness, we are done. This terminates in

$$O(|Q_1| \cdot |Q_2| \cdot |\Sigma|)$$

time.

Chapter 3

Context-Free Languages

3.1 Pushdown Automata

The main limitation of finite automata is the lack of memory. The pushdown automaton model gives us to access a stack, which allows us to recognize a richer class of languages.

Informally, imagine a NFA where the transitions become

$$a; b \to c.$$

An arc of this form can be followed when we read the symbol a as input, and the symbol popped from the stack is b. Following this transition causes c to be pushed onto the stack.

Any of a, b, c can be ϵ. When $a = \epsilon$, we do not read or advance the input string. When $b = \epsilon$, we do not read or pop a symbol from the stack. When $c = \epsilon$, no symbol is pushed onto the stack.

An *ϵ-transition* in which $a = b = c$ is labelled with ϵ.

We now introduce the notation

$$\Sigma_\epsilon := \Sigma \cup \{\epsilon\}$$

where we add a new symbol ϵ to Σ.

Definition 3.1.1 ((Nondeterministic) Pushdown Automaton)
A nondeterministic pushdown automaton (PDA) is an abstract machine defined by

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

- Q is a finite set of states
- Σ is the input alphabet
- Γ is the stack alphabet
- $\delta : Q \times \Sigma_\epsilon \times \Gamma_\epsilon \to \mathcal{P}(Q \times \Gamma_\epsilon)$ is the transition relation
- $q_0 \in Q$ is the start state
- $F \subseteq Q$ is the set of accepting states

We can obtain a deterministic PDA by modifying the definition of a DFA. But since we only discuss nondeterministic PDAs, we refer to them as simply PDAs.

Definition 3.1.2 (Accept)
A PDA M accepts the string $w \in \Sigma^*$ if and only if there exist a parameter $m \geq |w|$, symbols $y_1, \ldots, y_m \in \Sigma_\epsilon$, states $r_0, \ldots, r_m \in Q$, and strings $s_0, \ldots, s_m \in \Gamma^*$ such that

1. $w = y_1 \ldots y_m$
2. $r_0 = q_0$ an $s_0 = \epsilon$
3. $r_m \in F$
4. For each $i = 1, \ldots, m$, there is some $a, b \in \Gamma_\epsilon$ and $t \in \Gamma^*$ such that

$$\begin{aligned} (r_i, b) &\in \delta(r_{i-1}, w_i, a) \\ s_{i-1} &= at \\ s_i &= bt. \end{aligned}$$

Note that if we wanted to design a queue automata which is a PDA except with access to a queue instead of a stack, we can change the last few lines so that

$$s_{i-1} = ta, s_i = bt.$$

How about other data structures? What if we replace stack by a random access arrray? How about a priority queue which pops respects some "order" to the alphabet Σ and always pops the "biggest" symbol?

Definition 3.1.3 (Language of PDA)
The language of a PDA M is

$$L(M) := \{w \in \Sigma^* : M \text{ accepts } w\}.$$

We say that M *recognizes* $L(M)$.

Proposition 3.1.1
For every PDA M, there is a PDA M' with

$$L(M') = L(M)$$

and the property that M' only accepts when its stack is empty.

Proof
We can ensure that the first state which is pushed onto the stack is a new symbol α. For each final state $f \in F$, create a copy f' and the transition

$$\delta(f, \epsilon, \alpha) = (f', \epsilon).$$

Then set the new final states to $F' := \{f' : f \in F\}$.

There is a path from $q_0 \to f$ such that the stack is empty if and only if there is path from $q_0 \to f$ in the new PDA ending with α in the stack.

3.1.1 Regular Languages & PDAs

Proposition 3.1.2
Every regular language can be recognized by a pushdown automaton.

The idea here is that if we simply ignore the stack, then a PDA is essentially an NFA.

Proof
Let $M = (Q, \Sigma, \delta, q_0, F)$ be an NFA. Consider the PDA

$$M' := (Q, \Sigma, \varnothing, \delta', q_0, F).$$

The only change is the transition relation. For each $q \in Q$ and $s \in \Sigma_\epsilon$, define

$$\delta'(q, s, \epsilon) := \delta(q, s) \times \{\epsilon\}.$$

All other transitions relations are defined to be $\varnothing$.

Thus we simply ignore the stack and

$$L(M') = L(M).$$

Proposition 3.1.3
There is a pushdown automaton recognizing the language

$$L = \{0^n 1^n : n \geq 0\}.$$

Proof
See Figure 3.1 for a PDA recognizing L.

Note that the ϵ-transition is so that our PDA also recognizes ϵ.

This proves that PDAs are strictly more powerful than DFAS. Is there a class of languages which is a strict superset of context-free languages but not the set of all languages? Can we recognize every single language with (finite) automaton?

The language $\{0^n 1^n 2^n : n \geq 0\}$ is not context-free. However, it can be recognized by a automaton with 2 stacks. Does adding stacks always improve the ability of our automata's ability to recognize languages?

The language $\{ww : w \in \Sigma^*\}$ is not context-free. However, it can be recognized by a queue automaton. Can we classify the languages which are not context-free but recognizable by a queue automaton? How about adding multiple queues?

Proposition 3.1.4
There is a PDA which recognizes

$$L_{\text{pal}} = \{w \in \Sigma^* : w = w^R\}.$$

Proof
See Figure 3.2 for a PDA recognizing L.

Note that the ϵ-transition is so that our PDA also recognizes ϵ.

The arcs involving the symbol s runs over all $s \in \Sigma$. Thus from q_1, q_2, there are $|\Sigma|$ self-loops. Also, there are $|\Sigma|$ arcs from $q_1 \to q_2$.

3.1.2 Primitive Strings

Below is an open problem:

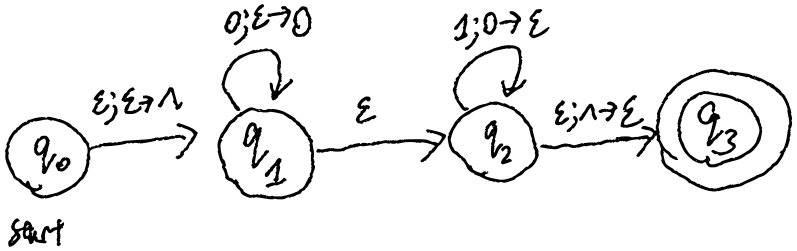

Figure 3.1: A PDA recognizing $\{0^n 1^n : n \geq 0\}$.

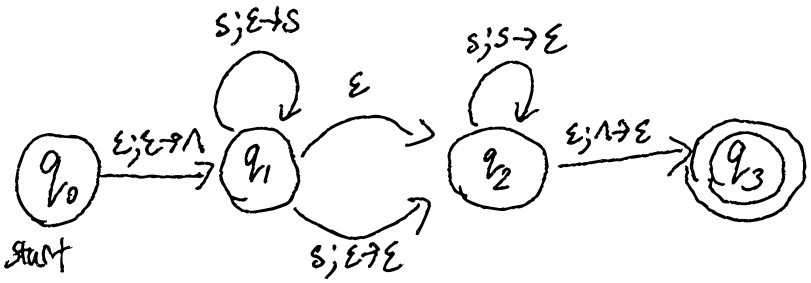

Figure 3.2: A PDA recognizing palindromes.

Problem 2 (Primitive Strings)
Does there exist a PDA that recognizes the language

$$L = \{w \in \Sigma_2^* : w \text{ is primitive}\}.$$

Intuitively, the PDA would need to be able to "loop" for each $n \geq 2$ that w is not y^n for some string y. We conjecture this is not possible without being able to read the string multiple times. Perhaps this would be possible with a queue automaton which somehow implements the greedy $O(n^2)$ algorithm which verifies primitivity.

3.2 Context-Free Grammars

Informally, a context free grammar is a set of rules that can be applied to individual variables to generate strings of variables and symbols. To generate a string from a grammer, we start with a special start variable and apply rules to one of the variables in the current string until no variables remain.

Definition 3.2.1 (Context-Free Grammar)
A context-free grammar (CFG) is a 4-tuple

$$G = (V, \Sigma, R, S)$$

where

- V is a set of variables (non-terminal symbols)
- Σ is a set of (terminal) symbols (terminals)
- R is a finite set of rules mapping $V \to (V \cup \Sigma)^*$
- S is the start variable in V

Definition 3.2.2 (Yield)
The string uAv yields the string uwv in the CFG $G = (V, \Sigma, R, S)$, denote

$$uAv \implies uwv,$$

when $A \in V$ is a variable, $u, v, w \in (\Sigma \cup V)^*$ are strings, and the rule $A \to w$ is in R.

Definition 3.2.3 (Derive)
The string u derives the string v in the CFG G, denoted

$$u \overset{*}{\implies} v,$$

when $u = v, u \implies v$, or there is a sequence of $k \geq 1$ strings $u_1, \ldots, u_k \in (V \cup \Sigma^*)$ such that $u \implies u_1 \implies \cdots \implies u_k \implies v$.

Note that if the rule $A \to \epsilon$ exists in our grammar, the string $AAA \overset{*}{\implies} \epsilon$.

3.2.1 Context-Free Languages

CFGs give us another way to define languages.

Definition 3.2.4 (Languaged Generated)
The language generated by the CFG $G = (V, \Sigma, R, S)$ is

$$L(G) = \{w \in \Sigma^* : S \overset{*}{\implies} w\}.$$

These languages are already very powerful! See `http://www.quut.com/c/ANSI-C-grammar-y.html`. Note that the set of semantically correct files for any language is most likely context-free. However, the set of files which will compile is most likely not context-free.

Example 3.2.1
A CFG generating $\{0^n 1^n : n \geq 0\}$ is

$$S \to \epsilon$$
$$S \to 0S1.$$

Example 3.2.2
A CFG generating the language of valid parentheses is

$$S \to \epsilon$$
$$S \to (S)$$
$$S \to SS.$$

Example 3.2.3
A CFG generating the language of binary palindromes is

$$S \to 0 \mid 1 \mid \epsilon$$
$$S \to 0S0$$
$$S \to 1S1.$$

Suppose G_1, G_2 are two CFGs. We can let S be the new start string, with rules

$$S \to S_1, S \to S_2.$$

Thus context-free languages are closed under unions.

Definition 3.2.5 (Context-Free Language)
A language is context-free if it can be generated by a context-free grammar.

3.3 Equivalence of CFGs and PDAs

We now show that the set of languages that can be recognized by pushdown automata is precisely the set of context-free languages.

Theorem 3.3.1
For every context-free grammar $G = (V, \Sigma, R, S)$, the language $L(G)$ can be recognized by a pushdown automaton.

First suppose we have a string $aAbBc$ where A, B are non-terminals. Then applying a rule $A \to S, B \to T$ in either order makes no difference.

$$aAbBc \implies aSbBc \implies aSbTc$$
$$aAbBc \implies aAbTc \implies aSbTc$$

Thus we can always use a *rightmost derivation*, where we always expand the leftmost non-terminal symbol.

Proof
Consider the PDA depicted in Figure 3.3.

$$M = \left(\{q_0, q_1\} \cup \bigcup_{(A \to x) \in R} \{x^{(1)}, x^{(2)}, \ldots, x^{(|x|-1)}\}, \Sigma, V \cup \Sigma, \delta, q_0, \{q_1\} \right)$$

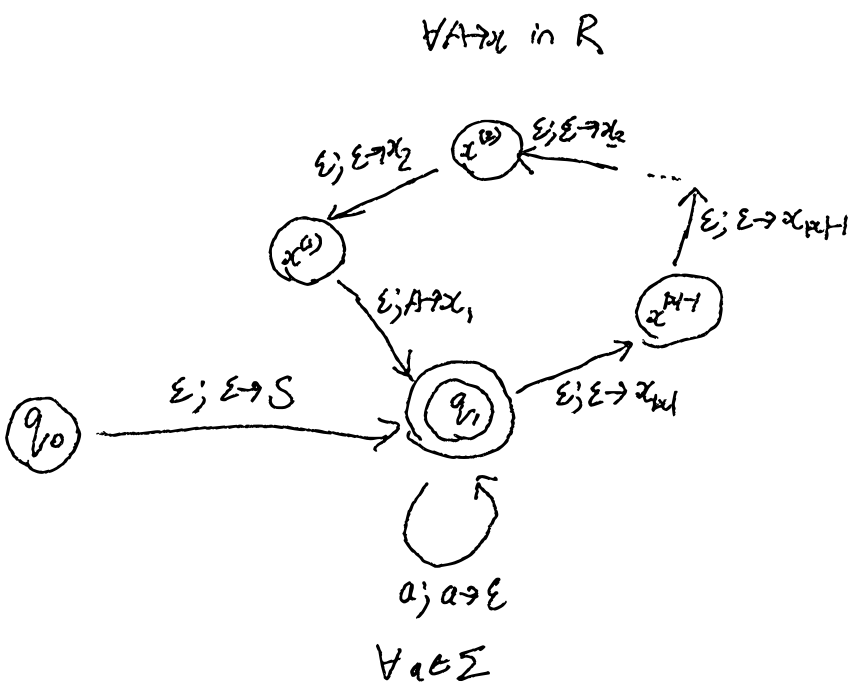

Figure 3.3: A PDA recognizing the language generated by a CFG.

where δ is given by

$$\begin{aligned}
\delta(q_0, \epsilon, \epsilon) &= \{(q_1, S)\} \\
\delta(q_1, \epsilon, A) &= \bigcup_{(A \to x) \in R} \{(x^{(|x|-1)}, x_{|x|})\} && \forall (A \to x) \in R \\
\delta(x^{(i)}, \epsilon, \epsilon) &= \{(x^{(i+1)}, x_i)\} && \forall (A \to x) \in R, \forall i \in [|x| - 2] \\
\delta(x^{(1)}, \epsilon, \epsilon) &= \{(q_1, x_1)\} && \forall (A \to x) \in R \\
\delta(q_1, a, a) &= \{q_1, \epsilon\} && \forall a \in \Sigma
\end{aligned}$$

and all other transition relations are $\varnothing$.

Essentially, we greedily parse the string from the leftmost side to "build" a leftmost derivation of the string, while using the stack to store intemediate derivations. We begin by pushing S onto the stack.

At each step, if the top stack symbol is a terminal, we "match" it with the next input symbol. Otherwise, the top stack symbol is non-termianl, and we "expand" it by pushing one of its expansion rule results onto the stack.

The only twist is that we are only allowed to push 1 symbol onto the stack at a time. Thus we need to implement pushing whole strings through a dicycle where each arc pushes the symbols of a string in reverse.

The other direction is most difficult, just as how showing that there is always a regex generating any regular language is more difficult than the converse. We simply this by first restricting our attention to a nicer class of PDAs.

Lemma 3.3.2
For every pushdown automaton M, there is a pushdown automaton

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

which satisfies

1. F consists of a single accept state.
2. M' always empties the stack before accepting.
3. Each transition $a; b \to c$ in δ has exactly one of b, c equal to ϵ, thus each transition either pushes or pops a symbol from the stack.

and $L(M) = L(M')$.

Proof
If F has multiple accept state, delete them, make a new accept state f with ϵ-transitions

from the previous accept states, and no transitions from f. Let M_1 be the PDA obtained this way.

To make M_1 accept only when the stack is empty, add a new stack symbol, say $\wedge$. By creating a new start state S', whose only transition leads to S (old start state) and pushes $\wedge$ onto the stack, all intermediate stack states now have $\wedge$ at its bottom. Then, for the sole accept state f, create loops which pops stack symbols other than $\wedge$. Finally, create the new sole accept state f' with an arc from f which pops the symbol $\wedge$. Let M_2 be the PDA obtained this way.

Now, to make M_2 always push or pop a symbol from the stack, we replace some arcs with dipaths of length 2. Indeed, if some transition does not push or pop, then replace this by a dipath which pushes then pops some garbage symbol. If some transition both pops and pushes, then replace this by a dipath which pops, then pushes. Let M' be the PDA obtained this way.

By construction $L(M) = L(M')$ and satisfies each of the 3 properties.

Theorem 3.3.3
For every PDA $M = (Q, \Sigma, V, \delta, q_0, F)$, the language $L(M)$ can be described by a context-free grammar.

Proof
We may assume without loss of generality that M satisfies the properties from the previous lemma. Let $f \in Q$ be the unique accepting state.

On a high-level, a leftmost derivation in our constructed grammar yields a path in the PDA.

Suppose $A_{pq} \overset{*}{\Longrightarrow} w \in \Sigma^*$. We interpret this as a path of computation starting from p, ending in q, consuming input w, and finishing by leaving the stack in the state of having popped A. Thus A_{pq} is a "promise" to pop.

In the case that we wish to transition from state q by reading s and popping A from the stack, we expand $A_{pq} \to s$. Intuitively, the promise to pop A was "fulfilled".

Remark that any inital transitions from q_0 necessarily push a symbol onto the stack, since we cannot pop and all transitions do exactly one of the two.

Consider the CFG $G = (\{A_{pq} : A \in \Gamma, p, q \in Q\}, \Sigma, R, S)$ where

$$\begin{aligned} S &\to \epsilon_{q_0 f} \\ A_{pq} &\to s B_{pq'} A_{q'q} && (q', B) \in \delta(p, s, \epsilon), \text{“push”} \\ A_{pq} &\to s && (q, \epsilon) \in \delta(p, s, A), \text{“pop”} \end{aligned}$$

By induction, $S \overset{*}{\Longrightarrow} w$ if and only if there is a path of computation in M starting from q_0, ending at f, which leaves the stack empty.

This proves the following theorem

Theorem 3.3.4
A language can be generated by a context-free grammar if and only if it can be recognized by a pushdown automaton.

3.4 Chomsky Normal Form

In general, CFGs have too much variety in their rules and it can be difficult to reason about them. It is thus useful to restrict ourselves to a more limited set of rules while still generating the same set of languages.

Definition 3.4.1 (Chomsky Normal Form)
The CFG $G = (V, \Sigma, R, S)$ is in Chomsky normal form (CNF) if every production rule is of the form

$$\begin{aligned} A &\to BC \\ A &\to a \\ S &\to \epsilon \end{aligned}$$

for some $A \in V$, some terminal symbol $a \in \Sigma$, and other variables $B, C \in V \setminus \{S\}$.

Example 3.4.1
A CFG generating the language $\{0^{2n} : n \geq 1\}$ is

$$\begin{aligned} S &\to AE \mid ZZ \\ A &\to AE \mid ZZ \\ E &\to ZZ \\ Z &\to 0. \end{aligned}$$

Example 3.4.2
A CFG generating the language $(01)^*$ is

$$\begin{aligned} S &\to AE \mid ZO \mid \epsilon \\ A &\to AE \mid ZO \\ E &\to ZO \\ Z &\to 0 \\ O &\to 1. \end{aligned}$$

Example 3.4.3
A CFG generating the language $L = \{0^m 1^n : m \geq n \geq 0\}$ is

$$\begin{aligned} S &\to A \mid \epsilon \\ A &\to BO \\ B &\to AZ \\ Z &\to ZZ \mid 0 \\ O &\to 1. \end{aligned}$$

Theorem 3.4.4
For every context-free grammar G, there is a context-free grammar G' in CNF such that $L(G) = L(G')$.

Proof
First, by adding the rule $S \to A$ for a new variable A and substituting all previous instances of S's in the rules with A's, we can ensure the start symbol does not appear on the right-hand side of any rule.

If the rule $A \to \epsilon$ exists for some non-start variable A, we can simply remove this rule. If there is a rule $B \to aAb$ for some strings a, b, then add the rule $B \to ab$. Note it is possible that $a, b = \epsilon$ so we create a new rule $B \to \epsilon$. But recursive repetition of this procedure eventually removes all but perhaps the ϵ rule $S \to \epsilon$. This ensures that there are no ϵ rules for non-start variables.

If there is a unit rule $A \to B$, we can also remove this rule. Then, for each rule $C \to aAb$, we can add the rule $C \to aBb$ to accomodate the deletion. This means there are no unit rules.

Next, by creating a new variable V_s for each $s \in \Sigma$ and adding the rule $V_s \to s$, we can substitute the appearances of s in the right hand sides of rules with V_s.

It remains to replace rules that generate variable strings of length greater than 2. Consider a rule $C \to ABa$ for some variables A, B, C and variable string a. Create a new variable V_{AB} with the rule $V_{AB} \to AB$. Then, replace $C \to ABa$ with $C \to V_{AB}a$.

This completes the proof.

3.4.1 Other Classes of Grammars

Definition 3.4.2 (Right-Regular Grammar)
A right-regular grammar is a CFG where every rule is of the form

$$A \to aB$$
$$A \to a$$
$$A \to \epsilon$$

There is a direct correspondance between NFA computations and right-regular grammar derivations. Thus the languages generated by a right-regular grammar are precisely the regular languages.

Definition 3.4.3 (Left-Regular Grammar)
A left-regular grammar is a CFG where every rule is of the form

$$A \to Ba$$
$$A \to a$$
$$A \to \epsilon$$

Intuitively, these are the reverses of right-regular generated languages and hence the regular languages once again.

Definition 3.4.4 (Linear Grammar)
A linear grammar is a CFG where every rule is of the form

$$A \to aB$$
$$A \to Ba$$
$$A \to a$$
$$A \to \epsilon$$

Since regular languages can be generated by right-regular grammars, they can certainly be

generated by linear grammars. However, the linear grammar

$$S \to A$$
$$A \to 0B \mid \epsilon$$
$$B \to A1$$

generates the language $\{0^n 1^n : n \geq 0\}$, hence linear grammars are strictly more powerful than DFAs/NFAs/Regex.

3.5 Pumping Lemma

While the pumping lemma was established by considering DFAs, it is easier to consider CFGs in CNF.

The first thing we notice is that if a CFL in CNF describes a finite language, the strings in those finite languages is bounded by the number of variables in the CFG.

Proposition 3.5.1
If $G = (V, \Sigma, R, S)$ is a context-free grammar in CNF with $|V| = m$ variables and G generates a string x of length $|x| \geq 2^m$, then $L(G)$ is an infinite language.

Proof
We argue by the contrapositive. Suppose G is finite. We wish to bound the number of leaves in the parse tree, which is precisely the length of x.

If $m = 1$, then the language consists of only ϵ. Hence we proceed assuming $m \geq 2$.

Consider a path P from the root of the parse tree S to a leaf of the tree. Observe that any internal node leads to at least 2 leaves. Therefore, if any variable A is repeated on P, then we can repeat the sequence of rules used between the first and second occurance of A as many times as we want to make x arbitrarily long.

Since $L(G)$ is finite, this cannot happen. Thus no variables can repeat on P and it contains at most m nodes. But the number of leaves is at most 1 more than the number of internal nodes.

The depth of any internal node is bounded above by $m-1$. Hence there are at most 2^{m-1} internal nodes and

$$2^{m-1} + 1 < 2^m$$

leaves.

Lemma 3.5.2 (Pumping Lemma, CFLs)
For every context-free language $L \subseteq \Sigma^*$, there is a number p such that for every $s \in L$ of length $|s| \geq p$, we can decompose

$$s = uvxyz$$

where

- $|v| + |y| > 0$
- $|vxy| \leq p$
- For all $i \geq 0$, $uv^ixy^iz \in L$

Proof
Let G be a CFG in CNF with m variables that generates L. Set $p := 2^m + 1$.

Let $x \in L$ be such that $|x| \geq p$. By the proof of the previous proposition, there must be a path P in the parse from the root node S to a leaf node with at least $m + 1$ internal nodes.

Consider the last $m + 1$ internal nodes. By the pigeonhold principle, at least one variable A is repeated.

Let x be the string formed by leaves of the subtree at the second instance of A. Then set vxy be the string formed by the leaves of the subtree at the first instance of A. Thus we can write

$$s = uvxyz$$

where $uvxyz$ is the string formed by leaves of the entire parse tree.

Now, the first instance of A has a subtree which does not contain the second instance of A. Hence at least one of v, y is not ϵ.

Moreover, $|vxy|$ is at most 1 more than the leaves of a depth m binary tree. This is at most $2^m + 1 =: p$ as required.

Finally, consider the subpath P' between the first and second instance of A which includes the first instance of A but not the second. For any $i \geq 0$, we may repeat P' times in the parse tree with the same derivation rules. This results in

$$uv^ixy^iz \in L$$

for all $i \geq 0$.

Observation 3.5.3
Both the pumping lemmas for regular languages and CFLs are achieved by analyzing the structure of their underlying characterizations. For regular languages, we observed that the only way to produce arbitrarily long strings in a DFA is through directed cycles. On the other hand, we utilize the "parse tree" of CFLs in CNF and identity strings in the CFL with leaves of the parse tree. Then by elementary graph theory, the only way to produce generate arbitrarily long strings is to repeat derivation rules in some path from the root to a leaf node.

Proposition 3.5.4
The language $L = \{0^n 1^n 2^n : n \geq 0\}$ is not context-free.

Proof
Suppose towards a contradiction that L is context-free and fix some pumping length $p \geq 1$ of L. Then $w = 0^p 1^p 2^p \in L$ and satisfies $|w| \geq p$. By the pumping lemma, we can decompose

$$w = uvxyz.$$

Case I: Suppose vxy contains only 0's, only 1's, or only 2's. Then clearly uv^2xy^2z is not in L.

Case II: Suppose vxy contains 0's and 1's or 1's and 2's. It cannot contain all 3 symbols since $|vxy| \leq p$.

But then uv^2xy^2z contains an imbalanced number of 0's, 1's, and 2's.

In either case, we contradict the pumping lemma. Thus by the arbitrary choice of p, we conclude that L is not context-free.

Conjecture 3.5.5
Let $f : \mathbb{Z}_+^3 \to \{0, 1\}$ be a boolean function. Then

$$L := \{0^m 1^n 2^\ell : f(m, n, \ell) = 1\}$$

is a CFL if and only if the value of f is independent of at least one of its arguments.

Proposition 3.5.6
The language

$$L := \{ww : w \in \Sigma_2^*\}$$

is not context-free.

Proof
Suppose towards a contradiction that L is context-free and that $p \geq 1$ is a pumping length of L. Consider $w := 0^p1^p0^p1^p$. Then $|w| \geq p$ and $w \in L$.

Suppose we can decompose

$$w = uvxyz$$

as in the pumping lemma.

Case I: Suppose vxy contains only 0's or only 1's. Then uv^2xy^2z is not in L.

Case II: Suppose vxy is a substring of 0^p1^p so that $v = 0^k, y = 1^\ell$ for some $k + \ell > 0$. Again, uv^2xy^2z is not in L.

Case III: Finally, suppose vxy is a substring of 0^p1^p. Thus $v = 1^k, y = 0^\ell$ for some $k + \ell > 0$. But then uv^2xy^2z is not in L again.

We have exhausted all cases since $|vxy| \leq p$. Thus we conclude that L cannot be context-free.

Conjecture 3.5.7
Is there a grammar which corresponds to automatons with multiple stacks and queue automatons? It seems that $\{0^n1^n2^n : n \geq 0\}$ can be easily recognized by an automaton with 2 stacks and $\{ww : w \in \Sigma_2^*\}$ is easily recognized by a queue automaton. What would those grammars look like?

3.6 Properties of Context-Free Languages

The class of context-free languages is a strictly larger set of languages than the regular languages. They are characterized by a strictly more powerful class of automaton. CFLs retain but also lose some properties of regular languages.

Proposition 3.6.1
CFLs are closed under

(i) unions

(ii) concatenation

(iii) star operation

(iv) reversal

Proof
Let G_1, G_2 be CFGs with start state S_1, S_2, respectively.

The CFG G consisting of the union of rules from G_1, G_2 as well as

$$S \to S_1 \mid S_2$$

generates the $L(G_1) \cup L(G_2)$.

The CFG G consisting of the union of rules from G_1, G_2 as well as

$$S \to S_1 S_2$$

generates the $L(G_1)L(G_2)$.

Let M be a PDA which recognizes L and has exactly one accepting state f which accepts if and only if the stack is empty. Then add an ϵ-transition from f to q_0 if none exist. Then, add an ϵ-transition from q_0 to f is none exist. The PDA M' recognizes the language L^*.

Let G be a CFG in CNF form. Thus it only contains rules of the form

$$S \to \epsilon$$
$$A \to BC$$
$$A \to a.$$

Let G' be the CFG obtained from G by changing rules of the form $A \to BC$ to

$$A \to CB.$$

Then G' generates L^R.

Conjecture 3.6.2
CFLs are closed under language expansion.

Proposition 3.6.3
CFG's are not in general closed under intersection and complementation.

Proof
Observe that the non context-free language

$$\{0^n 1^n 2^n : n \geq 0\} = (0^*\{1^n 2^n : n \geq 0\}) \cap (\{0^n 1^n : n \geq 0\}2^*)$$

is an intersection of CFLs.

Now, suppose towards a contradiction that CFLs are closed under complementation. But then any CFLs L_1, L_2,

$$L_1 \cap L_2 = \bar{L}_1 \cup \bar{L}_2$$

would be a CFL, which we have just disproven.

Observation 3.6.4
Let $f, g : \Sigma^* \to \{0, 1\}$ be booleans functions which are true if the input satisfies some condition. Consider two CFLs of the form

$$A = \{x \in \Sigma^* : f(x) = T\}$$
$$B = \{y \in \Sigma^* : g(y) = T\}.$$

Then the intersection

$$A \cap B = \{z \in \Sigma^* : f(z) \wedge g(z)\}$$

expresses then logical and of the f, g.
DFAs, we only have access to finite memory, thus the logical and can still be computed with finite memory. However, CFLs have access to a form of infinite memory, hence the logical and is in some sense too powerful to be preserved.

It is a bonus challenge to provide a formal definition of deterministic pushdown automata (DPDA) and deterministic context-free languages (DCFLs). The class of DCFLs is a strict subset of CFLs.

They also share more properties with regular languages than CFLs. DCFLs ARE closed under complementation and therefore intersections as well.

Theorem 3.6.5
The classes of deterministic and standard CFLs satisfy strict inclusion.

Proof
It is clear that any DCFL can be simulated by a CFL, whose transition relation is reduced to a singleton.

Conversely, let

$$L := \{0^n 1^n 2^n : n \geq 0\}$$

and consider $\bar{L}$. We can write it as

$$\bar{L} = \overline{0^*1^*2^*} \cup \{0^m 1^n 2^\ell : m \neq n\} \cup \{0^m 1^n 2^\ell : n \neq \ell\} \cup \{0^m 1^n 2^\ell : m \neq \ell\}$$

This is a union of regular language and 3 CFLs. Thus $\bar{L}$ is a CFL.

We claim that $\bar{L}$ is not a DCFG. Indeed, if it is, L would be a DCFG and therefore a CFG, which is a contradiction.

3.6.1 Topological Separation

One of the previous properties of regular languages was that for any infinite regular language L, there is an infinite regular language $L' \subseteq L$ such that $\bar{L'}$ was also infinite.

Whether this holds for CFLs is an open problem!

Problem 3
Prove or disprove. Every CFL L whose complement $\bar{L}$ is infinite, there exists a CFL $L' \supseteq L$ such that $L', \bar{L'}$ are both infinite.

Chapter 4

Computability

4.1 Turing Machines

Definition 4.1.1 (Deterministic Turing Machine)
A deterministic Turing machine is an abstract machine

$$M := (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$$

where

- Q is a finite set of states
- Σ is the input alphabet
- Γ is the tape alphabet
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the transition function
- $q_0 \in Q$ is the initial state
- $q_{\text{acc}} \in Q$ is the accept state
- $q_{\text{rej}} \in Q \setminus \{q_{\text{acc}}\}$ is the reject state

A transition from state A to B along the arc $a \to b; R$ is understood as the transition that is followed when the current state is A, and the symbol at the current head is a. We then overwrite a with b and move the tape head one to the right. Note that either a, b can be $\square$, which is used to denote blank cells with no content.

To be a valid Turing machine specification, the tape alphabet must include all of Σ as well as $\square$.

The intended meaning of $q_{\text{acc}}, q_{\text{rej}}$ is that the Turing machine halts whenever it arrives at the accepting or rejecting state. Thus even though we must define transitions from $q_{\text{acc}}, q_{\text{rej}}$, we will never specify such transitions.

Definition 4.1.2 (Configuration of Turing Machine)
A configuration of a TM T is a string wqy with $wy \in \Gamma^*$ and $q \in Q$ where

- q is the current state of the automaton
- wy is the current string on the tape
- the position of the tape head is on the first symbol in y

The *inital configuration* on input x is $q_0 x$.

We say that the configuration wqy is an *accepting configuration* if $q = q_{\text{acc}}$, an *rejecting configuration* if $q = q_{\text{rej}}$, and a *halting configuration* if it is an accepting or rejecting configuration.

Definition 4.1.3 (Yield)
For any strings $w, y \in \Gamma^*$, symbols $a, b, c \in \Gamma$, and states $q, q' \in Q$, the configuration $waqby$ in M yields $wq'acy$, denoted by

$$waqby \vdash wq'acy$$

when $\delta(q, b) = (q', c, L)$. Moreover,

$$qby \vdash q'\square cy$$

when $\delta(q, b) = (q', c, L)$.
Similarly,

$$waqby \vdash wacq'y$$

when $\delta(q, b) = (q', c, R)$. As well as

$$waqb \vdash wacq'\square$$

when $\delta(a, b) = (q, c, R)$.

Definition 4.1.4 (Derive)
We say the configuration wqy derives the configuration $w'q'y'$ in M, denote by

$$wqy \vdash^* w'q'y'$$

if there is a finite sequence of configurations $w_iq_iy_i, i \in [k]$ such that

$$wqy \vdash w_1q_1y_1 \vdash \cdots \vdash w_kq_ky_K \vdash w'q'y'.$$

Definition 4.1.5
The TM M with initial state q_0

- accepts x is (q_0, x) derives an accepting configuration
- rejects x if (q_0, x) derives a rejecting configuration
- halts on x if it either accepts or rejects x

4.1.1 Decidable Languages

Definition 4.1.6 (Decides)
A TM M decides $L \subseteq \Sigma^*$ if it accepts every $x \in L$ and rejects every $x \notin L$.

A language is *decidable* if there is a TM that decides it.

Observation 4.1.1
It is possible that a TM M never halts on some input x! In this case, M does not decide any language. This is in contratary to PDAs and NFAs which ALWAYS corresponds to some language.

The set of decidable languages is also known as the set of *recursive* languages.

4.1.2 Recognizable Languages

Definition 4.1.7 (Recognize)
The TM M recognizes the langauge L if it accepts every $x \in L$ and either rejects or does not halt for every $x \notin L$.

A language is *recognizable* when there is some TM that recognizes it.

Conjecture 4.1.2
The set of recognizable languages is also known as the set of *recursively enumerable* languages. Is there a collection of enumerable languages? If so, what is the relation to the recursively enumerable languages?

Note that although not every TM decides a language, every TM recognizes a language. Moreover, the set of decidable languages is a subset of the set of recognizable languages.

4.2 Church-Turing Thesis

Although a TM seems like a very restricted model of computation, we believe that it is just as powerful as ANY other reasonable model of computation.

Hypothesis 4.2.1 (Church-Turing Thesis)
Any decision problem that can be solved by an algorithm on any computer that we can construct in this universe corresponds to a language that can be decided by a Turing machine.

We cannot prove the Church-Turing Thesis as it requires a formal mathematical definition of the terms involved in the statement.

Even if we cannot prove it, we can heuristically verify it on various models of computation.

Question 4.2.2
It feels as if the Church-Turing Thesis is basing the concept of computation on classical physics. As we know now, there are problems for which there exist efficient quantum algorithms but no known classical counterparts.
Is there a generalization of the Church-Turing Thesis which takes into account quantum physics?

4.2.1 Turing Machines with Registers

Definition 4.2.1 (Register Turing Machine)
A register TM has a finite number of registers which can each store one symbol from the tape alphabet. The transitions of these register TM are determined by the current state, the content of the tape at the current head position, and the content of all the registers.
On each transition, the register TM can overwrite the content of the current tape cell as well as all of the registers.

Proposition 4.2.3
Every language that can be decided by a register TM M can also be decided by a TM M'.

Proof
Let $\mathcal{C}$ be the set of possible register configurations. Let M' be a TM with state set $Q(M) \times \mathcal{C}$.

A transition from state A to B through the arc $a \to b; L$ with register operation $S \to S'$ in M corresponds to the transition from state (A, S) to (B, S') through the arc $a \to b; L$ in M'.

The case for $a \to b; R$ is symmetric.

4.2.2 Subroutine Turing Machines

Definition 4.2.2 (Subroutine Turing Machine)
A subroutine TM is a TM that can call TMs to run as black-box subroutines on the input. A call to a subroutine can be represented as a special type of state in the subroutine TM. All actions on the input tape (including movement of the current head position) persist after the subroutine halts, and two transitions are followed out of its special state; one for when the subroutine accepts and the other for when it rejects.

Proposition 4.2.4
Every language that can be decided by a subroutine Turing machine can also be decided by a TM.

Proof
It suffices to show that we can simulate subroutine TMs with register TMs with one register.

The states of our TM is obtained by taking the union of all non-subroutine states in the subroutines TM as well as any of the ones it could call and new "return states" J_M for each possible subroutine M.

The tape operations are unchanged upon entering a subroutine. Upon exiting, whether we halted by accepting the string or rejecting the string is indicated by transitioning to the state J_M with a 1 or 0 in the register.

Then we transition from J_M depending on the value in the return register to the appropriate state in "parent" machine.

Observation 4.2.5
Our previous proof can be tweaked to work with arbitrary recursion. However, we need a "stack" to store the state of the "parent" machine to which we should transition after recursing.
To do this, we introduce a separator symbol "|" which divides the tape into 2 segments: the "stack" and the "heap". Moreover we add a pointer indicator symbol # used on the heap which serves to track where the pointer should return to following an operation on the stack.
Each element on the stack indicates the previous level of recusion. Specifically, it stores the state upon which we should transition following a subroutine.
The idea is identical to implementing recursion in assembly.

4.2.3 Multitape Turing Machines

Definition 4.2.3 (Multitape Turing Machine)
A multitape TM is a TM with $k \geq 1$ tapes instead of one. It also has k tape heads, one per tape, which can move independently of each other, and the current symbol on each tape can be read and overwritten on each transition.
In otherwords, the transition function of a multitape TM is a function of the form

$$\delta : Q \times \Gamma^k \to Q \times (\Gamma \times \{L, R, N\})^k,$$

where N indicates no movement.
Initially, the input is written in tape 1 and the other tapes are empty.

Theorem 4.2.6
Every language that can be decided by a multitape TM can also be decided by a TM.

Proof
It suffices to show that we can simulate a multitape TM with a register TM.

We "compress" the k tapes into one. First observe that in finite time, each of the k tapes only contain a finite number of non-$\square$ symbols. Let "|" be a new separator symbol and # be a new symbol indicating that the next symbol should be where the pointer is placed on the current tape.

Thus given an input string x, the initial configuration is

$$| \, q_0 \# x \, | \, \# \, | \, \# \, | \, \# \, |$$

for $k = 4$.

For each transition, we move the pointer $k - 1$ times, stopping each time we see a # (we keep the pointer at the first #). Then, we apply the desired operation on the i-th tape.

When we wish to shift the ith-tape to the left, we will copy all strings to left of the pointer one cell to the left (adding a new empty character if necessary). Then we write $\square$ to the newly freed up cell. This might be necessary when moving the # indicator as that requires us to shift the tape to the right when moving the indicator to the left and shifting the tape to the left when moving it to the right. The other case where this is necessary is when we extend the i-th tape to the left.

The cases for the right are analogous.

Observation 4.2.7
It can be shown that the lambda calculus model of computation can be simulated by TMs.

4.3 Universal Turing Machines

Definition 4.3.1 (Universal Turing Machine)
A universal TM is a TM U such that any input M, x which encodes a TM M and a string x accepts if M accepts x and rejects if M rejects x.

Theorem 4.3.1
There exists a Universal TM.

Proof
It suffices to show that there is a universal TM with 1 register and 2 tapes. The first tape only stores the input TM M and x. The second tape simulates the tape of M. Finally, the register stores the current state of the simulation.

There are only 3 states $q_0, q_{\text{acc}}, q_{\text{rej}}$. We transition from $q_0 \to q_{\text{acc}}$ if the register indicates the we are in the accepting state of simulation and similary $q_0 \to q_{\text{rej}}$ if the register indicates the simulation rejects x. All other transitions are loops at q_0.

To simulate a transition from states $A \to B$ through the arc $a \to b; L$, the head at the second tape must be over a cell containing the symbol a and the register holds the state A. We overwrite the register to B, the seconday tape to b and move the head of the secondary tape to the left.

4.3.1 Turing Completeness

A Turing-complete model of computation is one that is "as poweful" as TMs. The Church-Turing thesis states that such models are the "most powerful" models of computation.

Definition 4.3.2 (Turing Complete)
A model of computation is Turing-complete if for every decidable language L, there is a machine/algorithm in this model that decides L.

Proposition 4.3.2
If a universal Turing machine can be implemented in the model of computation $\mathcal{M}$, then $\mathcal{M}$ is Turing-complete.

Proof
Fix a decidable language and let M be a TM which decides L. By implementing a universal TM U in $\mathcal{M}$, we can decide L simply by feeding it the input M, x.

By definition, $\mathcal{M}$ is Turing-complete.

Observation 4.3.3
Piano arithmetic is Turing-complete.

4.4 Non-Deterministic Turing Machines

Definition 4.4.1 (Nondeterministic Turing Machine)
A Nondeterministic Turing machine (NTM) is an abstract machine defined by

$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, a_{\text{rej}})$$

where

- Q is a finite set of states
- Σ is the input alphabet
- Γ is the tape alphabet
- $\delta : Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R\})$ is the transition relation
- $q_0 \in Q$ is the initial state
- $q_{\text{acc}} \in Q$ is the accept state
- $q_{\text{rej}} \in Q \setminus \{q_{\text{acc}}\}$ is the reject state

Definition 4.4.2 (Decide)
The NTM M decides $L \subseteq \Sigma^*$ if and only if

(i) for every $x \in L$, there is a computational path that accepts x

(ii) ALL computational paths for x halt

(iii) for $x \notin L$, all computational paths reject x

Definition 4.4.3 (Recognize)
The NTM M recognizes $L \subseteq \Sigma^*$ if and only if

(i) for every $x \in L$, there is a computational path that accepts x

(ii) for $x \notin L$, no computational paths accepts x

Theorem 4.4.1
The set of languages that can be decided by TMs is exactly the same as the set of languages that can be decided by NTMs.

Proof
Let M be a TM. Let M' be the NTM obtained from M by changing the transition function to a transition relation where the image of any input is a singleton given by the transition function of M. Then M' decides L if and only if M decides L.

Conversely, consider a NTL M' and an input string x. We can simulate M' with a TM M which performs "backtracking".

Indeed, let M be a TM with 2 tapes. We introduce a new separator symbol | and a head indicator symbol #. The primary tape is a stack holding copies of the tape of M' separated by | with # indicating the head location of the "current" tape. The secondary tape is stack of states of M, with | indicating that the next state on the stack is a "backtrack" state.

Suppose we are simulating M' with configuration aqb. Enumerate

$$\delta(q, b_1) = \{(q_i, t_i, T_i) : i \in [m]\}.$$

Let $S_i(ab)$ be the tape after overwriting b_1 with t_i and translating according to T_i. Initially, add $| \, S_1(ab) \, | \, S_2(ab) \, | \cdots | \, S_m(ab)$ to the primary tape. and $| \, q_1 q_2 \dots q_m$ to the secondary tape. We then continue the simulation as if we are in state q_m with tape $S_m(ab)$.

If the current symbol in the secondary tape is not |, then we arrived at the current state through a forward arc. Thus we would add all possible next states to the tapes. If the current symbol in the secondary tape is |, then we have exhausted all computational paths from some configuration and we "pop" the primary and secondary tape to "backtrack" to a parent configuration.

If we ever land in an accept state, the simulation terminates and we accept x. If the primary and secondary tapes become empty, then we reject x.

Observation 4.4.2
Although we can simulate NTMs with TMs, it is not "efficient" in the sense that we are simply brute force attempting every possible computation path.

4.4.1 Equivalence of Recognizability

The equivalence between deterministic and nondeterministic Turing machines hold for the case of recognizing langauges as well.

Theorem 4.4.3
The set of languages that can be recognized by TMs is exactly the same as the set of languages that can be recognized by NTMs.

Question 4.4.4
Why are NFAs equivalent to DFAs, NTMs equivalent to TMs, but PDAs are not equivalent to deterministic PDAs?

4.5 Decidable Languages

In order to understand what languages we can decide with any computer, it suffices by the Church-Turing thesis to consider Turing machines.

4.5.1 Context-Free Languages

Theorem 4.5.1
Every context-free language is decidable.

Proof
Let L be context-free. We must show that there is a NTM T for which $w \in L$ if and only if there is a computational path in T which accepts w. Moreover, all computation paths of T terminate.

Let G be a CFG of L in CNF. We simulate the derivation of w by using the tape to store intermediate derivations. To ensure that all computation paths terminate, we observe that each rule of G of the form $A \to BC$ increases the length of the final string by 1, hence we permit at most $|w| - 1$ applications of these types of rules.

To see that there is a computation path which accepts w if and only $w \in L$, observe that this is the definition of G.

Observation 4.5.2
We could have simply simulated PDAs with Turing machines. However, it is harder to show that all computational paths terminate.

Proposition 4.5.3
The language $L := \{0^n 1^n 2^n : n \geq 0\}$ is decidable.

Proof
A 4-tape TM T for which the input is read in the first tape, and the 2nd, 3rd, 4th tapes are used to track the number of 0s, 1s, 2s, respectively easily decides L.

4.5.2 Clique

Proposition 4.5.4
The language L_{CLIQUE} consisting of the encoding of every graph G on $n \geq 1$ vertices that contains a clique of size at least $\frac{n}{2}$ is decidable.

Proof
Remark that if there is a clique of size at least $\frac{n}{2}$, if and only if there exists a clique of size $S = \lceil \frac{n}{2} \rceil$.

Suppose the graph G is encoded into an adjacency matrix. For every subset $V' \subseteq V$ of the vertices such that $|V|' = S$, we check if every pair of vertices $v, w \in V'$ is adjacent. If there is some such V', then we accept G. Otherwise, we reject G.

4.5.3 Accepting DFAs

Proposition 4.5.5
Let A_{DFA} be the language of all encodings of (M, x) where M is a DFA and x is a string accepted by M. Then A_{DFA} is decidable.

Proof
Our algorithm simply simulates M on the input x.

4.5.4 Closure Properties

Theorem 4.5.6
The class of decidable languages are closed under

- union
- intersection
- complementation
- set difference
- star operation
- reversal

Proof
Union: Let T_1, T_2 be two TMs which decide L_1, L_2. Then we can run T_1 on an input x and then T_2 on an input x to decide if $x \in L_1 \cup L_2$.

Complementation: Simply swap the accepting and rejecting states for a TM T which decides L.

Intersection We know that $L_1 \cap L_2 = \bar{L}_1 \cup \bar{L}_2$. Hence the result follows from our work above.

Set Difference: We know that $L_1 \setminus L_2 = L_2 \cap \bar{L}_2$, hence we have already proven the claim.

Star Operation: First we claim that decidable languages are closed under concatentation. Let L_1, L_2 be decidable and x an input, For each $i \in \{0, 1, \ldots, |x|\}$, guess that $x[1, \ldots, i], x[i+1, \ldots, |x|]$ are in L_1, L_2 and verify if it is the case. Then $x \in L_1 L_2$ if and only this is true for some i.

Recall that $L^* = \bigcup_{k \geq 0} L^k$. We then guess some $0 \leq k \leq |x|$ such that $x \in L^k$. Then, we split x in the finitely many possible configurations of k substrings and test that each substring belongs to L. Then $x \in L^k$ if and only if this is true for some configuration.

Reversal: Observe that

$$L^R = \{w^R : w \in L\} = \{w : w^R \in L\}.$$

Hence on an input x, we can simply make a reverse copy of x and check if a TM deciding L accepts x^R.

4.6 Undecidable Languages

4.6.1 Existence

Proposition 4.6.1
There exist undecidable languages.

Proof
For any alphabet Σ, the class of all languages over Σ is uncountable. We argue there are only a countable number of Turing machines. To see this, fix integers $k \geq 2$ and $\ell \geq |\Sigma| + 1$.

Fix a tape alphabet Σ of size ℓ. Then there are only a finite number of TMs with input alphabet Σ, tape alphabet Γ, using k states.

Now, any valid tape alphabet is differentiated only by its length. Thus there are only countably many tape alphabets. Then, there are only countably many TMs since the number of possible states is countable.

4.6.2 First Undecidable Language

Definition 4.6.1 (Accepting Language)
The accepting language is given by

$$A_{\text{TM}} := \{\langle M, x\rangle : M \text{ is a TM that accepts } x\}.$$

Theorem 4.6.2
A_{TM} is undecidable.

Proof
Suppose towards a contradiction that there is a TM A which decides A_{TM}.

Let B be the TM obtained from A which rejects $\langle T\rangle$ if A accepts $\langle T, \langle T\rangle\rangle$. Otherwise, B accepts $\langle T\rangle$ if A rejects $\langle T, \langle T\rangle\rangle$.

Suppose B accepts $\langle B\rangle$. Then A accepts $\langle B, \langle B\rangle\rangle$. But by definition, B then rejects $\langle B\rangle$.

Suppose now that B rejects $\langle B\rangle$. Then A rejects $\langle B, \langle B\rangle\rangle$. But by definition, B then

accepts $\langle B \rangle$.

By contradiction, we conclude that B and therefore A cannot exist.

Observation 4.6.3
We can decide the subset of A_{TM} consisting of $\langle M, x \rangle$ where M is a DFA. Why does this not in general work for TMs?
It seems like the issue is the ability of TMs to simulate other TMs, an ability which is not possible for DFAs.

4.7 More Undecidable Languages

4.7.1 Halting

Definition 4.7.1 (Halting Language)
The halting language is

$$H_{\text{TM}} := \{\langle M, x \rangle : M \text{ halts on input } x\},$$

the set of encodings of TMs M and strings x such that M halts when it is run on input x.

Theorem 4.7.1
The language H_{TM} is undecidable.

Proof
Suppose towards a contradiction that there is a TM T that decides H_{TM}.

On an input $\langle M, x \rangle$, Let M' be the TM obtained from M by creating a new state l for which all transitions lead back to itself (loop). Then all transitions to q_{rej} are re-routed to l. Clearly M accepts x if and only if M' halts on x.

Thus we can decide A_{TM} using the TM

$$T\langle M', x \rangle.$$

Question 4.7.2
It seems like the Halting language/problem is introduced in lower level courses first. However, we introduced the accepting language first. Is there a reason for doing so? Maybe the concept of "Halting" is much more intuitive and requires less background to understand than the concept of accepting a string?

4.7.2 Emptiness

Definition 4.7.2
The language

$$\text{EMPTY}_{\text{TM}} := \{\langle M \rangle : L(M) = \varnothing\}.$$

Theorem 4.7.3
The language EMPTY_{TM} is undecidable.

Proof
Suppose there is a TM T which decides EMPTY_{TM}. On an input $\langle M, x \rangle$, let M' be the TM obtained from M which first checks by brute force that the input string is exactly x, then feeds it to M.

Then $L(M') \neq \varnothing$ if and only if M accepts x. Hence we can decide A_{TM} by checking

$$\text{EMPTY}_{\text{TM}}(\langle M', x \rangle).$$

4.7.3 Equality

Definition 4.7.3 (Equality)
The language

$$\text{EQ}_{\text{TM}} := \{\langle M_1, M_2 \rangle : L(M_2) = L(M_2)\}$$

is the set of encodings of TM pairs that recognize the same language.

Theorem 4.7.4
EQ_{TM} is undecidable.

Proof
Let T be a TM which decides EQ_{TM}. We can decide EMPTY_{TM} using the TM

$$\text{EQ}_{\text{TM}}(\langle M, \varnothing \rangle),$$

where $\varnothing$ indicates some TM which decides the empty language.

4.8 Rice's Theorem

Rice's theorem generalizes the idea that when L is represented using a TM that recognizes it, we cannot decide whether L includes a given string x, even if $L = \varnothing$.

4.8.1 Properties of Decidable Languages

Definition 4.8.1 (Property)
A property of recognizable languages is a subset of the set of all recognizable languages.

We say that a property of recognizable languages is *non-trivial* if it is not the empty set nor the set of all recognizable languages.

Definition 4.8.2
For every Turing machine language P, let

$$L_P := \{\langle M \rangle : L(M) \in P\}$$

denote the language of all encodings of Turing machines that recognize a language in P.

Theorem 4.8.1 (Rice)
For every non-trivial property P of recognizable languages, the language L_P is undecidable.

Proof
By taking $\bar{P}$ if necessary, we may assume without loss of generality that $\varnothing \notin P$.

Suppose there is a TM T_P which decides L_P. Our goal is to obtain a TM which decides

A_{TM}.

First, observe that since P is non-trivial, we can find some TM T such that $L(T) \in P$.

Given an input $\langle M, x\rangle$, let $\tilde{M}$ be the machine which acts on input y as follows:

1. Simulate M on the input x.
2. If M accepts x, proceed. If M rejects x, loop in place instead (thus M either accepts or does not halt on x).
3. Simulate T on y.

Thus if M accepts x, $L(\tilde{M}) = L(T) \in P$. Otherwise, $L(\tilde{M}) = \varnothing \notin P$.

Hence the machine H such that on input $\langle M, x\rangle$ feeds $\tilde{M}$ to T_P decides A_{TM}. But A_{TM} is undecidable and hence so is L_P.

Observation 4.8.2
Even though Rice's theorem seems very powerful, a key limit is that the language in question must be "related" to the action of Turing machines. Thus it does not say much about a general language whether it is decidable or not.
Is there some sort of "Pumping Lemma" for decidable languages? It might be difficult to get a necessary and sufficient condition for a language to be decidable, but what about some non-trivial necessary conditions?

4.9 Recognizability

So far, we have been focusing on decidable languages. Clearly, any decidable language is recognizable, since a TM which decides L also recognizes L.

Moreover, we know that there are only countably many TMs over an alphabet with a fixed tape alphabet. Hence there can only be countably many recognizable languages while the class of all languages is uncountable.

Question 4.9.1
Each progressively larger class of languages we have considered (regular, context-free, decidable, recognizable) have been countable.
Is there some interesting class of languages that is uncountable (excluding the class of all languages)?

4.9.1 A Recognizable but Undecidable Language

Theorem 4.9.2
A_{TM} is recognizable.

Proof
Since we know universal TMs exist, this is not too hard.

Indeed, on an input $\langle M, x \rangle$, simply simulate M on x. Then return "accept" if M accepts x.

4.9.2 Decidability & Recognizability

Theorem 4.9.3
A language is decidable if and only if both L and its complement $\bar{L}$ are recognizable.

Proof
The forward direction is easy. The decidable languages are closed under complements, hence if L is decidable, both $L, \bar{L}$ are decidable and therefore recognizable.

Suppose now that $L, \bar{L}$ are recognizable. Let M, M' be TMs that recognize $L, \bar{L}$. Using two universal TMs, we can simulate M, M' on an input x.

Then $x \in L$ if M accepts x and $x \notin L$ if M' accepts x. Hence we can decide if $x \in L$ using this machine and L is decidable.

Definition 4.9.1 (Corecognizable Language)
A language is corecognizable if its complement is recognizable.

4.9.3 First Unrecognizable Language

Corollary 4.9.3.1
The language $\overline{A_{\text{TM}}}$ is not recognizable.

Proof
If it were, then by the above theorem, A_{TM} would be decidable.

Observation 4.9.4
We can extend this result to any recognizable but undecidable language. The complement of any such language is necessarily not recognizable.

Chapter 5

Time Complexity

5.1 TIME Complexity Classes

We have previously focused on *computability theory*, the idealized scenarios where machines have unlimited resources. We now shift to *complexity* theory, where computers have bounded resources.

5.1.1 Time Cost & Complexity

Definition 5.1.1 (Time Cost)
The time cost of the Turing machine M on input x is the number of transitions it follows before it halts.

Note that we will restrict our attention to TMs that always halt.

Definition 5.1.2 (Worst-Case Time Cost)
The (worst-case) time cost of the TM M is the function $t : \mathbb{N} \to \mathbb{N}$ where $t(n)$ is the maximum time cost of M on any input x of length $|x| = n$.

Observation 5.1.1
We can alternatively define average, expected, and best-case time costs.

The tie cost is a measure of efficienty of individual TMs. We use this to measure the notion

of the complexity of decidable languages.

Definition 5.1.3 (Time Complexity Class)
For every function $t : \mathbb{N} \to \mathbb{N}$, the time complexity class TIME(t) is the set of all languages that can be decided by a multitape TM with worst-case time cost bounded above by $O(t)$.

Note that we defined TIME(t) is based on multitape TMs.

5.1.2 First Examples

The class TIME(n) is the set of languages that can be computed by linear-time TMs and is particularly interesting.

Proposition 5.1.2
The class TIME(n) is a strict superset of the class of all regular languages.

Proof
By simulating DFAs, there is a linear-time TM which decides any regular language

However, there is clearly a 2-tape TM which decides $\{0^n1^n : n \geq 0\}$ in linear time. Hence the inclusion is strict.

Proposition 5.1.3 (Bonus)
The language $L := \{0^n1^n2^n : n \geq 0\}$ is not context-free but satisfies $L \in$ TIME(n).

Proof
We have already shown it cannot be context-free.

Consider the 3-TM M for which the primary tape holds the input. Let x be an input string. First ensure $x \in 0^*1^*2^*$ in linear time.

Then, as we iterate through the 0's, use the secondary and tertiary tape to keep 2 copies of the count. As we iterate through the 1's, subtract a counter from the secondary tape per symbol and reject the string if we ever see "too many" 1's. As we iterate through the 2's, subtract a counter from the tertiary tape per symbol and reject the string if we ever see "too many" 1's. Then we accept if and only if the seconday and tertiary heads are on an empty cell.

We can thus decide L in linear-time.

Proposition 5.1.4 (Challenge)
TIME(1) consists of the languages where membership can be decided by examining a constant prefix of the string.

Proposition 5.1.5
For every $t \in o(n)$, $\text{TIME}(t) = \text{TIME}(1)$.

Proof
Suppose towards a contradiction that there is some $L \in \text{TIME}(t) \setminus \text{TIME}(1)$. We claim that for each $N \in \mathbb{N}$, there is some $x \in L, |x| =: n \geq N$ such that there is $y \notin L$ with

$$x_1 \dots x_{\frac{n}{2}} = y_1 \dots y_{\frac{n}{2}}.$$

If the claim holds, then by choosing a sufficiently large N such that $n \geq N$ implies

$$t(n) < \frac{1}{2}n,$$

we see that no TM can distinguish x, y in time $t(n)$ and arrive at the desired contradiction.

To see the claim, suppose that there is some $N_0 \in \mathbb{N}$ such that for all $x \in L, |x| =: n \geq N$ implies there are no $y \notin L$ that agrees with x on the first $\frac{1}{2}n$ symbols.

There is at least one string $\bar{x} \in L$ of length at least N_0 since L is necessarily infinite. Let $n_0 := |\bar{x}|$. There are no strings $y \notin L$ which agrees with $\bar{x}$ on the first $\frac{1}{2}n_0$ symbols.

Thus any $y \notin L$ either has length less than $\frac{1}{2}n_0$, or differs from $\bar{x}$ in the first $\frac{1}{2}n_0$ characters. In other words, any $|z| \geq \frac{1}{2}n_0$ belongs in L as long as it agrees with x_0 on the first $\frac{1}{2}n_0$ symbols.

But n_0 is a constant! Moreover, there are only a finite number of strings with length less than $\frac{1}{2}n_0$. Thus we actually have $L \in \text{TIME}(1)$, proving the claim by contradiction.

5.1.3 Linear Speedup Theorem

The reason why we defined $\text{TIME}(t)$ with multitape TMs is that the definition of (single-tape) TMs are essentially equivalent.

Theorem 5.1.6 (Linear Speedup)
For every $\epsilon > 0$, if there is a multitape TM M that decides L with time cost $t(n)$, then there is also a multitape TM M' that decides L and has time cost

$$\epsilon t(n) + n + 2.$$

Proof
We may as well assume that $t \in \Omega(n)$, since otherwise, there is only TIME(1).

The idea is to compress multiple tape cells into a single symbol from a larger alphabet.

Let m be a constant. The new tape alphabet is $\Gamma \cup \Gamma^m$. We may introduce new states but only finitely many. We also add an additional tape. The head position within each tape is stored as states.

First, compress the input onto an additional tape by iterating through the n input tape cells and compressing every m symbols into a single tuple in the new tape. Note that we need 2 extra steps. The first step lands past the end of the input. The second "ends" the copying step since m does not necessarily divide n and we might be "midway" through an m-tuple.

Note that the naive description outlined above requires $\frac{n}{m}$ extra steps to realign the tape head. We can eliminate this by only copying as required during the simulation.

Now, at any given configuration, our TM can read all m cells to the left and right of its head position in a tape by moving left, right, right, left. Then, in at most 4 more steps, it can updaate the cells within m of its current position. In other word, with only 8 steps, we can simulate m steps within the original TM.

The overall time cost is at most

$$2 + n + \frac{n}{m} + \frac{8}{m}t(n) \leq \frac{9}{m}t(n) + n + 2.$$

By choosing $m \geq \frac{9}{\epsilon}$, we are done.

Observation 5.1.7
The statement (as well as proof) of the linear speedup theorem is analogous to "hardware" improvements in the everyday computer. Given a slow piece of hardware, we can copy over an input to a faster piece of hardware (the transfer is limited by the slower hardware), then speed up computation by a constant factor.
What more, we even have the hardware trick of "lazy evaluation" in which we only transfer the input when necessary for the computation. Unlike most of the results we have seen in this class, this theorem seems very deeply inspired by hardware (or hardware is very inspired by this theorem)!

5.2 Time Hierarchy Theorem

Generally, giving TMs more time to run allows them to decide more languages.

5.2.1 Weak Time Hierarchy Theorem

Theorem 5.2.1 (Weak Time Hierarchy)
$\text{TIME}(n) \subsetneq \text{TIME}(n^3)$.

Proof
Let T be the TM such that on input $\langle M, x\rangle$, simulates M on x for at most $|x|^2$ steps. If M accepted, output reject. If M did not halt or rejects, output accept.

Now, T always halts and thus T decides $L := L(M)$. Taking into consideration the $O(\log n)$ time per step to decrement a counter, $L \in \text{TIME}(n^2 \log n) \subseteq \text{TIME}(n^3)$. We now argue that $L \notin \text{TIME}(n)$.

Suppose there is a TM H that decides L in $O(n)$ time. Consider the input $\langle H, 1^k\rangle$ where k is a sufficiently large so that T can simulate the entire execution of H on $\langle H, 1^k\rangle$. Thus T outputs the opposite of what H outputs on $\langle H, 1^k\rangle$. It follows that H does NOT decide the same language as T.

5.2.2 Time Hierarchy Theorem

Definition 5.2.1 (Time-Constructible)
The function $T : \mathbb{N} \to \mathbb{N}$ is time-constructible if for every n, there is a Turing machine that writes $t(n)$ ones on the tape in time $O(t(n))$.

Theorem 5.2.2
For every time-constructible function $t : \mathbb{N} \to \mathbb{N}$,

$$\text{TIME}(t(n)) \subsetneq \text{TIME}(t(n) \log t(n)).$$

Proof
Let T be the TM such that on input $\langle M, x \rangle$, simulates M on x for at most $t(|x|)$ steps. If M accepts, output "reject". If M does not halt or reject, output "accept".

Now, T always halts and thus T decides $L := L(M)$. By construction, $L \in \text{TIME}(t(n) \log t(n))$, since simulation incurs logarithmic overhead. We now argue that $L \notin \text{TIME}(t(n))$.

Indeed, suppose there is some machine H which decides L in time $t(n)$. Consider the input $\langle H, 1^k \rangle$ where k is sufficiently large so that $n := |\langle H, 1^k \rangle| \in \Theta(k)$ and T can simulate the entire execution of H on $\langle H, 1^k \rangle$. Thus T outputs the opposite of what H outputs on $\langle H, 1^k \rangle$. It follows that H does NOT decide the same language as T.

5.2.3 Necessity of Time Constructibility

Theorem 5.2.3
There exist non-time-constructible functions $t : \mathbb{N} \to \mathbb{N}$ with $t \in \omega(n)$ for which

$$\text{TIME}(t(n)) = \text{TIME}(t(n) \log t(n)).$$

Proof
Suppose towards a contradiction that all functions $n^x, x > 1$, satisfy

$$\text{TIME}(n^x) \subsetneq \text{TIME}(n^x \log n).$$

We can uniquely pick some $L_x \in \text{TIME}(n^x \log n) \setminus \text{TIME}(n^x)$. Observe that for any $y < x$,

$$L_x \notin \text{TIME}(n^y \log n) \subseteq \text{TIME}(n^x).$$

Thus have an injective function

$$x \mapsto L_x.$$

Since $\mathbb{R}_{>1}$ is uncountable, we have a contradiction.

There is some $\bar{x} > 1$ at which this statement necessarily fails. Note that then $n^{\bar{x}}$ is necessarily non-time-constructible.

5.3 P

Definition 5.3.1 (Polynomial-Time Turing Machine)
A polynomial-time TM is a TM with time cost $O(n^k)$ for some $k \geq 1$.

Definition 5.3.2 (P)
The class P is the set of all languages that can be decided by some polynomial-time TM. Thus

$$\mathrm{P} := \bigcup_{k \geq 0} \mathrm{TIME}(n^k).$$

Note that $\mathrm{P} = \bigcup_{k \geq C} \mathrm{TIME}(n^k)$ for any $C \geq 0$. But it is NOT equal when k is bounded above.

5.3.1 Motivation

Strength

One reason to study P is the strength of the conclusion that $L \notin \mathrm{P}$.

Observation 5.3.1
A quasi polynomial is a function of the form

$$2^{O(\log^c n)}$$

where $c \geq 1$ is a constant. Note that $c = 1$ means that it is linear.
For $c > 1$, any quasi polynomial is not bounded above by any polynomial. This is the smallest time complexity class I can identify which still could still contain a language $L \notin P$.

Closure

P is closed under subroutine calls. If a polytime subroutine TM uses black-box calls to another polytime TM, the total time complexity remains polynomial.

Robustness

Hypothesis 5.3.2 (Cobham-Edmonds Thesis)
Any decision problem that can be solved in polynomial time by an algorithm on any physically-realizable computer corresponds to a language that is in P.

Observation 5.3.3
In some sense, the Cobham-Edmonds thesis stipulates that P completely captures essence of what it means to be "easy, fast, tractable". However, it ignores factors such as constants, the size of the exponent of the polynomial, and the typical size of the input.
The linear speedup theorem partially addressses the first factor, since it shows that hardware improvements make constants irrelevant. However, (classical) hardware improvements are limited by the physical laws of the world, hence it is unreasonable to assume hardware improvements will continue unimpeded.
The exponent of the polynomial matters significantly when the typical input size is very large. Indeed, the simplex method is very fast in practice. Its smoothed analysis has a running time of $\tilde{O}(n^3)$ where the $\tilde{O}$ notation hides a polylogarithmic factor. But on even moderate sized linear programs, the textbook simplex algorithm struggles (without using other speedup techniques).

There is much less consensus on the validity of the Cobham-Edmonds Thesis than on the Church-Turing thesis. Quantum computers might disprove the Cobham-Edmonds thesis if there exists a polytime quantum algorithm that can decide any language $L \notin P$.

5.3.2 First Observations

Proposition 5.3.4
Every regular language is in P.

Proof
Every regular language is in TIME(n) $\subseteq$ P.

Proposition 5.3.5
The languages $L_{0^n1^n}, L_{0^n1^n2^n}$ are in P.

Proof
There is a 3-tape and 4-tape TM which decides both languages in linear time, respectively.

Definition 5.3.3 (E)
The class

$$\mathrm{E} := \bigcup_{k \geq 0} \mathrm{TIME}(2^{kn}).$$

Definition 5.3.4 (Exp)
The class

$$\textsc{Exp} := \bigcup_{k \geq 0} \mathrm{TIME}\left(2^{n^k}\right).$$

Theorem 5.3.6
$P \subsetneq E \subsetneq \textsc{Exp}$.

Proof
We argue by the time hierarchy theorem.

For any $k \geq 0$,

$$n^k \log n^k \in o(2^n).$$

Hence $\mathrm{P} \subsetneq \textsc{Exp}$.

Now,

$$2^{0 \cdot n} \log 2^{0 \cdot n} = 1 \in o(2^{n^1}).$$

Hence $\mathrm{E} \subsetneq \textsc{Exp}$.

5.4 NP

The class P is especially interesting to study in relation to the class NP.

Definition 5.4.1 (Time Cost)
The time cost of the NTM M on input x is the maximum number of transitions it follows before it halts, over all computation paths.

Question 5.4.1
Why is the time cost defined as the maximum on the number of transitions over all computation paths and not just the minimum/average/median/etc? Does it change anything if we modify the definition to one of the following above?

Definition 5.4.2 (Worst-Case Time Cost)
The (worst-case) time cost of a NTM is the function $t : \mathbb{N} \to \mathbb{N}$ where $t(n)$ is the maximum time cost of M on any input x of length $|x| = n$.

Definition 5.4.3 (Nondeterministic Time Complexity Class)
For every function $t : \mathbb{N} \to \mathbb{N}$, the nondeterministic time complexity class $\text{NTIME}(t)$ is the set of all languages that can be decided by a nondeterministic multitape TM with worst-case time cost bounded by $O(t)$.

Definition 5.4.4 (NP)
$\text{NP} := \bigcup_{k \geq 0} \text{NTIME}(n^k)$.

5.4.1 Polynomial-Time Verifiers

Definition 5.4.5 (Verifier)
A verifier is a TM with an additional certificate tape. The input to a verifier is a pair (w, c) with the input string w written on the main tape and the certificate string c written on the certificate tape.

Definition 5.4.6 (Languaged Recognized)
The language recognized by the verifier V is

$$L(V) := \{w \in \Sigma^* : \exists c, V \text{ accepts } (w, c)\}.$$

The verifier *decides* $L(V)$ if it always halts. A *polynomial-time verifier* is a verifier with time

cost $O(n^k)$ for some $k \geq 0$.

Theorem 5.4.2
A language L is in NP if and only if it can be decided by a polynomial-time verifier.

Proof
$(\implies)$ Let T be an NTM which decides L and has time cost $O(n^k), k \geq 0$. Now, the verifier V be obtained from T such that V interprets the input (x, c) as an encoding of some $O(n^k)$ length computation path along which T can take on x, and simulates T on x through that path.

Then $x \in L(T)$ if and only if there is some path encoded as c such that T accepts c if and only if $x \in L(V)$.

$(\impliedby)$ Let V be a poly-time verifier which decides L and has time cost $O(n^k), k \geq 0$. Let T be the NTM obtained from V which on an input $x, |x| = n$, chooses some string c with length $O(n^k)$ and simulates V on (x, c).

Then $x \in L(V)$ if and only if there is some c such that V accepts (x, c) if and only if $x \in L(T)$.

Question 5.4.3
It has been mentioned in other courses that if a lanugage is in both NP and co-NP, it is very likely to be in P (ie matchings, b-matchings, and many combinatorial optimization problems).
Why is this the case?

5.4.2 P vs NP

Clearly P $\subseteq$ NP. Whether we have equality is one of the biggest open problems in computer science today.

Part of the reason for the popularity of this problem is the philosophical question whether it is just as easy to find solutions to a problem than to just verify a solution for teh same problem. Intuitively, the answer is no but a proof with respect to the P vs NP problem eludes us!

Proposition 5.4.4
NP $\subseteq$ Exp.

Proof
Any NTM with time cost $O(n^k)$ can be simulated by a TM with time cost $O(2^{n^k})$ which essentially implements backtracking on all possible computation paths.

5.5 NP-Completeness

We seek to capture the notion of being the "hardest" languages in NP to compute.

5.5.1 Polynomial-Time Reductions

Definition 5.5.1 (Polynomial-Time Reducible)
Given two languages $A, B \subseteq \Sigma^*$, the language A is polynomial-time reducible to B, denoted

$$A \leq_P B$$

if there is a function $f : \Sigma^* \to \Sigma^*$ such that

1. For every $x \in \Sigma^*$, we have $x \in A \iff f(x) \in B$.
2. There is a polynomial-time TM M that on input $x \in \Sigma^*$, erases it and repaces it with $f(x)$ on the tap and then halts.

Reductions that satisfy the conditions in the definition are also known as *Karp reductions*.

Lemma 5.5.1
For every pair of languages A, B such that $A \leq_P B$,

(i) $B \in \mathrm{P} \implies A \in \mathrm{P}$

(ii) $B \in \mathrm{NP} \implies A \in \mathrm{NP}$

Proof
Suppose $B \in \mathrm{P}$ and let T be a TM that decides it. We can decide A in polytime using the polytime TM M which computes f and then simulates T.

Suppose $B \in \mathrm{NP}$ and let V be a polytime verifier for it. We can attain a polytime verifier M for A which on input (x, c), simulates V on $(f(x), c)$.

5.5.2 NP-Hardness & Completeness

Definition 5.5.2 (NP-hard)
The language L is NP-hard if every language $A \in \text{NP}$ satisfies

$$A \leq_P L.$$

Definition 5.5.3 (NP-Complete)
The language L is NP-complete if it is NP-hard and satisfies $L \in \text{NP}$.

Proposition 5.5.2
If any NP-hard language L is in P, then $P = \text{NP}$.

Proof
We already know that $P \subseteq \text{NP}$. Let $A \in \text{NP}$. Since $A \leq_P L$, then $A \in \text{P}$ as well. It follows that $\text{NP} \subseteq \text{P}$ and we conclude the proof.

5.5.3 Existence of an NP-Complete Language

Theorem 5.5.3
There exists an NP-complete language.

Proof
Define

$$\text{TM}_{\text{SAT}} := \{\langle V, x, 1^m, 1^t\rangle : \exists c \in \Sigma^*, |c| \leq m, V \text{ accepts } (x, c) \text{ in at most } t \text{ steps}\}.$$

Note that $\text{TM}_{\text{SAT}} \in \text{NP}$. Indeed, an NTM which decides TM_{SAT} would try all strings $c \in \Sigma^*, |c| \leq m$ and simulate V on (x, c). This is polytime since the length of the encoding is proportional to $|x| + m + t$.

Now, let L be any language in NP and V a polytime verifier for L with time cost $O(n^k)$. For each $x \in L$, there is some certificate c_x such that V accepts (x, c_x) in t_x steps. Write $n_x := |x|$. Define $f : \Sigma^* \to \Sigma^*$ by

$$f(x) := \langle V, x, 1^{O(n_x^k)}, 1^{O(n_x^k)}\rangle$$

Clearly, $x \in L \iff f(x) \in \text{TM}_{\text{SAT}}$.

To see that f is polytime computable, Now, it takes time $|V| + n_x + O(n_x^k)$ to erase x and replace it with $f(x)$. This is certainly a polynomial of the input size.

5.6 Satisfiability

We showed the existence of an NP-complete language. We strive towards a simple, natural language that is also NP-complete.

Observation 5.6.1
The NP-complete language we explored in the previous section is helpful in that its NP-completeness follows also immediately from the definition of NP-completeness. However, it feels very artificial and difficult to work with. One of the powers of NP-complete languages is that any language in NP reduces to it. But working with our previous language does not give any insight towards more natural problems in which we are interested.
On the other hand, there are very simple reductions between 3-SAT and other classical NP-complete problems such as independent set, clique, vertex cover, etc.

5.6.1 Boolean Formulas & SAT

A *Boolean variable* is one that takes on values either True or False. A *literal* is a variable x or its negation $\bar{x}$. A *clause* is the OR (disjunction) of one or more literals.

Definition 5.6.1 (Boolean Formula in Conjunctive Normal Form)
A Boolean formula in CNF is an AND (conjunction) of one or more clauses.

Definition 5.6.2 (Satisfiable)
A boolean formula is satisfiable if there exists some assignment of True or False values to its underlying variables that causes the formula to evaluate to True.

Definition 5.6.3 (Satisfiability Language)
The satisfiability language is

$$\text{SAT} := \{\langle \phi \rangle : \phi \text{ is satisfiable}\}.$$

Lemma 5.6.2
$SAT \in \text{NP}$.

Proof
Consider the verifier V such that on input $\langle \phi, c \rangle$ verifies that c is an assignment of variables such that ϕ evaluates to True. This requires verifying each clause has at least one literal which evaluates to true and is linear in the input size of ϕ.

5.6.2 Tableaux

The NP-completeness of SAT is obtained by considering tableaux.

Definition 5.6.4 (Tableau)
A tableau is a sequence of configurations of a TM.

Definition 5.6.5 (Valid Tableau)
A tableau is valid if the sequence corresponds exactly to the configurations of a TM starting from its initial configuration all the way to the configuration where the machine halts.

Definition 5.6.6 (Accepting Tableau)
A valid tableau is accepting when the TM is in an accepting state in its last configuration.

The name tableau comes from the fact that we can represent one using a table, with each row representing a configuration.

The *height* of a tableau is the number of rows of the table that represents it, which is the number of configurations of the TM from the start of its execution to the moment that it halts. This also represents the time cost of a TM on a given input.

The *width* of a valid tableau is the length of the strings representing the configuration. For convenience, we add blank symbols to represent empty tape cells so that the configurations all have the same length and line up.

5.6.3 Tableau & SAT

In order to prove that SAT is NP-complete, we need to show that every language $L \in \text{NP}$ and every string w, we can construct in polytime a boolean formula ϕ_w that is satisfiable if and only if $w \in L$.

5.7 Cook-Levin Theorem

We now complete the proof of the Cook-Levin theorem, which stipulates that SAT is NP-complete.

For $L \in \text{NP}$ and M a NTM that decides L in polytime. We want to show that $w \in L$ if and only if there is an accepting tableau for M on input w.

For a tableau T of size $h \times w$, we define

$$x_{i,j,\sigma} := \begin{cases} T, & T_{i,j} = \sigma \\ F, & T_{i,j} \neq \sigma \end{cases}$$

for each $i \leq j, j \leq w$ and $\sigma \in Q \cup \Gamma$. We wish to construct ϕ_w so that it is satisfied if and only if the Boolean variables $x_{i,j,\sigma}$ represent an accepting tableau for M on input w. We can do this by introducing constraints to enforce the different conditions that the Boolean variables must satisfy to represent such a tableau.

5.7.1 Cell Constraints

The first step is to ensure that the Boolean variables $x_{i,j,\sigma}$ really encode some tableau. For this to occur, each cell (i, j) of the tableau must contain EXACTLY one symbol.

Lemma 5.7.1
For each cell (i, j), there is a Boolean formula $\phi_{i,j}$ that is satisfied if and only if there is exactly one symbol σ for which $x_{i,j,\sigma}$ is True.
Furthermore, there is a Boolean formula ϕ_{cell} that is satisfied if and only if the condition above is true for all cells (i, j) in the tableau.

Proof
Take

$$\phi_{i,j} := (\vee_{\sigma \in Q \cup \Gamma} \phi_{i,j,\sigma}) \wedge (\wedge_{\sigma \neq \pi \in Q \cup \Gamma} \neg x_{i,j,\sigma} \vee \neg x_{i,j,\pi})$$

and

$$\phi_{\text{cell}} := \wedge_{i \leq h, j \leq w} \phi_{i,j}.$$

5.7.2 Initial & Final Constraints

Our next step is to verify that this tableau corresponds to the configurations of M when it runs on w.

Lemma 5.7.2
There is a Boolean formula $\phi_{\text{start},w}$ that is satisfied if and only if the first row of the tableau encoded in the Boolean variables corresponds to the initial configuration of M on input w.

We assume every row of the tableau is indexed by $-w \leq j \leq w$ and that the first row of the tableau must start with q_0 in the 0-th position.

Proof
Consider the formula

$$\phi_{\text{start},w} := (\wedge_{j<0} x_{1,j,\square}) \wedge x_{1,0,q_0} \wedge (\wedge_{j=1}^{n} x_{1,j,w_j}) \wedge (\wedge_{j=n+1}^{w} x_{1,j,\square})$$

Lemma 5.7.3
There is a Boolean formula ϕ_{acc} that is satisfied by some Boolean variables that satisfy ϕ_{code} if and only if the last row of teh tableau encoded in the Boolean variables corresponds to an accepting configuration.

Proof
Take

$$\phi_{\text{acc}} := \vee_{j \leq w} x_{h,j,q_{\text{acc}}}.$$

5.7.3 Valid Tableau Constraints

We need to check that every row is obtained by a transition in M from the previous row.

Lemma 5.7.4
For each cell (i, j), there is a Boolean formula ϕ_{valid} that is satisfied by some Boolean variables that satisfy ϕ_{code} if and only if they encode a valid tableau for M.

Proof
Consider the rows $j, j+1$ and a 2×3 window in these two rows.

$$\begin{matrix} a,b,c \\ d,e,f \end{matrix}$$

For row i, row $i+1$ can differ from it by at most 3 consecutive positions.

$$\begin{matrix} a,q,c \\ a,c',q \end{matrix}$$

$$\begin{matrix} a,q,c \\ q,a,c' \end{matrix}$$

$$\begin{matrix} a,b,q \\ a,b,c' \end{matrix}$$

$$\begin{matrix} a,b,q \\ a,q,b \end{matrix}$$

$$\begin{matrix} q,b,c \\ b',q,c \end{matrix}$$

$$\begin{matrix} q,b,c \\ a,b',c \\ a,b,c \\ q,b,c \end{matrix}$$

$$\begin{matrix} a,b,c \\ a,b,q \end{matrix}$$

If row $i+1$ does not follows from a transition from row i, some window must not be valid. Hence we simply check that all windows are valid using a conjunction over all subformulas which stipulate each window is valid.

Theorem 5.7.5 (Cook-Levin)
The language SAT is NP-complete.

Proof
We have already showed that SAT is a member of NP. Let $L \in$ NP and M an NTM which decides L.

For each $x \in L$, let

$$\phi(x) := \phi_{\text{cell}} \wedge \phi_{\text{start},w} \wedge \phi_{\text{acc}} \wedge \phi_{\text{valid}}$$

denote the boolean formula which is satisfiable if and only if there is a tableau describing an accepting computation path in M if and only if M accepts x. This is certainly polytime computable since each formula is polytime computable.

By definition,

$$L \leq_P \text{SAT}$$

and SAT is NP-complete.

Observation 5.7.6
The guided proof of the Cook-Levin theorem relies heavily on tableaux. The main advantage of this is the 2×3 window which allows us to reduce the problem of checking whether each row follows from a transition in the previous row to checking whether the two rows satisfy a linear number of finite subproblems.
Alternatively, we can build our boolean formulas directly from the configurations of an accepting computation path. This is more direct and intuitive from my perspective since the tableau seems to abtract out too much from TMs.

5.8 Beyond P & NP

There many fascinating problems related to the classes P, NP that go beyond the P vs NP question itself.

5.8.1 coNP

The class coNP is defined as

Definition 5.8.1 (coNP)
coNP is the language

$$\text{coNP} := \{L \subseteq \Sigma^* : \bar{L} \in \text{NP}\}.$$

Proposition 5.8.1
If NP $\neq$ coNP, then $P \neq$ NP.

Proof
We argue by the contrapositive. Suppose P = NP. Then

$$\text{coNP} = \{L \subseteq \Sigma^* : \bar{L} \in \text{P}\}.$$

But if $\bar{L} \in$ P, then we can easily decide if $L \in$ P by outputing the opposite of a TM which decides $\bar{L}$. Hence

$$\text{coNP} = \text{P} = \text{NP}$$

as desired.

5.8.2 NP Intersect coNP

Proposition 5.8.2
P $\subseteq$ NP $\cap$ coNP.

Proof
If $L \in$ P $\subseteq$ NP, then $\bar{L} \in$ P as well. Hence $\bar{L} \in$ coNP by definition. It follows that $P \subseteq$ NP $\cap$ coNP.

Proposition 5.8.3
The language

$$\text{FACTOR} := \{\langle M, N\rangle : M < N, d \mid N \text{ for some } 1 < d < M\}$$

is in NP $\cap$ coNP.

Proof
We know that FACTOR is in NP since an NTM which tries if any $1 < d < M$ divides N decides FACTOR in polytime.

Moreover, FACTOR is in coNP. Indeed, observe that

$$\overline{\text{FACTOR}} = \{\langle M, N\rangle : M \geq N \vee \forall 1 < d < M, d \nmid N\}.$$

This language is certainly in NP since an NTM which first checks if $M \geq N$ then verifies if N has a divisor between $(1, M)$ in parallel decides it in polytime.

5.8.3 Ladner's Theorem

If P $\neq$ NP, there are still many questions about the structure of the class NP. For instance, it is not true that every language in P is either in P or is NP-complete.

Theorem 5.8.4 (Ladner)
If P $\neq$ NP, there exist some languages in NP $\setminus$ P that are not NP-complete.

Proof
Consider the padded SAT language

$$L := \{\langle \phi, 1^{|\phi|^{H(\phi)}} \rangle\}$$

where H is defined as follows:

Enumerate all TMs $M_1, M_2, \ldots$ such that for every TM M and some $N \in \mathbb{N}$, there is some $k \geq N$ for which $M = M_k$.

Then $H(n)$ is the minimum number $i < \log\log n$ such that M_i decides if $x \in L$ or not in at most $1 + i \cdot |x|^i$ steps, for each x of length $|x| < \log_2 n$. If no such number exists, then $H(n) = \log\log n$.

Clearly, $L \in$ NP, since $H(n)$ is polynomial time computable, and we can nondeterministically check if ϕ is satisfiable.

$\underline{L \notin \text{P:}}$ Suppose $L \in$ P. Then there is some TM M that decides L in $O(n^c)$ steps for some constant $c > 0$. We can pick some $i > c$ such that $M = M_i$. Hence $n > 2^{2^i}, H(n) \leq i$.

It follows that $H(n) \in O(1)$. But then the length of $\langle \phi, 1^{|\phi|^{O(1)}} \rangle$ is $O(|\langle \phi \rangle|)$ and we can decide SAT in polytime with a TM which decides L in polytime.

This is absurd given the assumption that $P \neq$ NP.

$\underline{L \text{ is not NP-complete:}}$ First we claim that for each i, $H(n) = i$ for only finitely many n. Otherwise, consider any $x \in \Sigma_2^*$ and choose $n > 2^{|x|}$ such that $H(n) = i$. By definition, this means M_i decides if $x \in L$ in $i \cdot |x|^i$ time. Specifically, $L \in$ P, which is a contradiction.

Suppose now there is a poloytime reduction f from SAT to L such that

$$\phi \in \text{SAT} \iff f(\phi) = \langle \psi, 1^{\psi^{H(|\psi|)}} \rangle \in L.$$

But then for sufficiently large $|\phi|$,

$$|\psi| < |\phi|.$$

This means for a sufficiently large Boolean formula, we can recursively apply this reduction until it reduces to a sufficiently small formula for which we can solve in constant time.

Thus SAT $\in$ P, contradicting the assumption that $P \neq$ NP.

We have shown that L cannot be in P nor be NP-complete, concluding the proof.

5.8.4 Polynomial Hierarchy

We can extend the notion of verifiers to obtain richer conditions for acceptance.

Definition 5.8.2 ($\forall\exists$-Verifier)
A $\forall\epsilon$-verifier is a TM V with two extra tapes that include certificates c_1, c_2 that recognizes the language

$$L(V) = \{w \in \Sigma^* : \forall c_1, \exists c_2, V \text{ accepts } (w, c_1, c_2)\}.$$

Definition 5.8.3 (Π_2^p)
Π_2^p is the class of all languages that can be decided by polytime $\forall\exists$-verifiers

Theorem 5.8.5
If P = NP, then P = Π_2^p as well.

Proof
Suppose P = NP. Pick any $L \in \Pi_2^p$ which is decided by the polytime verifier V. Now, consider the language

$$L' := \{\langle w, c_1\rangle : w, c_1 \in \Sigma^*, \exists c_2 \in \Sigma^*, V \text{ accepts } (w, c_1, c_2)\}.$$

Then it is clear that $L' \in$ NP = P since V is essentialy a polytime verifier for L' as well.

There is a polytime TM M' that decides L' by assumption. Observe that $x \notin L$ if and only if exists c_1, M' rejects $\langle x, c_1\rangle$. Thus $\bar{L} \in$ NP = P = coNP by our prior work.

Since P is closed under complementation, $L \in$ P as well. It follows by the arbitrary choice of L that

$$\text{P} = \Pi_2^P.$$

By extending the above generalization, we can obtain an infinite class of complexity classes whose union is known as the *polynomial hierarchy*.

Chapter 6

Space Complexity

6.1 SPACE Complexity Classes

6.1.1 Space Cost & Complexity

Definition 6.1.1 (Space Cost)
The space cost of the TM M on input x is the total number of distinct tape cells that are visited by M's tape head before M halts.

We can extend this notion to multitape TMs as the total number of visited cells in all the tapes to obtain the space cost.

Definition 6.1.2 (Worst-Case Space Cost)
The worst-case space cost of the TM M is the function $s : \mathbb{N} \to \mathbb{N}$ where $s(n)$ is the maximum space cost of M on any input x of length $|x| = n$.

It is clear that $\text{TIME}(s) \subseteq \text{SPACE}(s)$ since it takes $s(n)$ steps to visit $s(n)$ cells.

Definition 6.1.3 (Space Complexity Class)
For every function $s : \mathbb{N} \to \mathbb{N}$, the space complexity class $\text{SPACE}(s)$ is the set of all languages that can be decided by a multitape TM with worst-case space cost bounded above by $O(s)$.

Example 6.1.1
Any regular language can be decided with a TM in $O(1)$ space. Indeed, simply simulate a DFA.

We define

$$\mathrm{SPACE}_1(s)$$

to be the set of languages that can be computed by single-tape TMs with space cost s. Note that we always have $\mathrm{SPACE}_1(s) = \mathrm{SPACE}(s)$ since we can simulate a k-tape TM with a single tape TM where symbols are k-tuples.

6.1.2 Time & Space

Theorem 6.1.2
For every function $s : \mathbb{N} \to \mathbb{N}$,

$$\mathrm{NTIME}(s) \subseteq \mathrm{SPACE}(s) \subseteq \mathrm{TIME}(2^{O(s)}).$$

Proof
$\mathrm{NTIME}(s) \subseteq \mathrm{SPACE}(s)$: For any single computation path, on an input of length n, a TM deciding a language in $\mathrm{NTIME}(s)$ can only visit $O(s(n))$ cells.

$\mathrm{SPACE}(s) \subseteq \mathrm{TIME}(2^{O(s)})$: Let M be a k-tape TM which decides We claim that on an input w of length n, M has at most $2^{O(s(n))}$ possible configurations.

Indeed,

$$|Q| \cdot (O(s))^k \cdot |\Gamma|^{O(s(n))} \leq 2^{O(s(n))}.$$

Thus after $2^{O(s(n))}$ steps, M must be in an infinite loop, which is impossible since it decides a language.

6.1.3 PSPACE

Definition 6.1.4 (PSPACE)
$\textsc{PSpace} := \bigcup_{k \geq 1} \mathrm{SPACE}(n^k)$

Corollary 6.1.2.1
$\mathrm{P} \subseteq \mathrm{NP} \subseteq \textsc{PSpace} \subseteq \textsc{Exp}$

Proof
We have already shown the first two inclusions. To see that PSPACE $\subseteq$ EXP, simply apply the previous theorem with $s(n) = n^k$ for some constant k.

Since P $\subsetneq$ EXP, at least one of the inclusions in the chain is proper. It is believed that all are proper, but so far all of them are open problems. Establishing any of them would represent a significant breakthrough in complexity theory.

Problem 4
Show either P $\neq$ PSPACE and/or PSPACE $\neq$ EXP.

6.2 Savitch's Theorem

We can consider the space complexity of the P vs NP problem by considering the space cost of NTMs.

6.2.1 Nondeterministic Space Complexity

Definition 6.2.1 (Space Cost)
The space cost of the NTM M on input x is the maximum total number of distinct tape cells that are visited by M's tape head before M halts, with the maximum taken over all possible computational paths.

Definition 6.2.2 (Worst-Case Space Cost)
The (worst-case) space cost of the NTM M is the function $s : \mathbb{N} \to \mathbb{N}$ where $s(n)$ is the maximum space cost of M on any input x of length $|x| = n$.

Definition 6.2.3 (NSPACE)
For every function $s : \mathbb{N} \to \mathbb{N}$, the time complexity class NSPACE(s) is the set of all languages that can be decided by a multitape NTM with worst-case space cost bounded above by $O(s)$.

Definition 6.2.4 (PSPACE)
NPSPACE $:= \bigcup_{k \geq 1} \text{NSPACE}(n^k)$

6.2.2 The Derivation Language

Definition 6.2.5 (Derivation Language)
The language Derive $:= \{\langle N, x, c_1, c_2, t\rangle : \text{on } x, \text{ NTM } N \text{ transitions } c_1 \to c_2 \text{ in } \leq t \text{ steps}\}$.

Lemma 6.2.1
There is a TM that decides Derive and has space cost $O(s \log t)$ on input $\langle N, x, c_1, c_2, t\rangle$ when N has space cost $O(s)$ on input x.

Proof
The problem is equivalent to seeking a path of length at most t between two nodes u, w of a digraph on at most $|V| \leq 2^{O(s)}$ vertices. We can recursively solve this by asking if there is an intermediary node v such that there are uv, vw-dipaths of lengths at most $\frac{t}{2}$.

The recursion depth is $O(\log t)$. If we implement the recusion using a stack, then there will always only by $\log t$ subproblems on the stack. Since each configuration takes $O(s)$ space to store, the total space complexity is $O(s \log t)$.

6.2.3 Proof of Savitch's Theorem

Theorem 6.2.2 (Savitch)
For every $s : \mathbb{N} \to \mathbb{N}$,

$$\text{NSPACE}(s(n)) \subseteq \text{SPACE}(s(n)^2).$$

Proof
Let N be an NTM with time cost $s(n)$. To decide deterministically whether $x \in L(n)$, we can ask if

$$\langle N, x, q_0, q_{\text{acc}}, 2^{O(s)}\rangle \in \text{Derive}\,.$$

By the previous lemma, this can be decided in time

$$O(s \log 2^{O(s)}) \subseteq O(s^2)$$

as desired.

Corollary 6.2.2.1
PSPACE = NPSPACE

Proof
Any language L which can be decided by an NTM of space cost $O(n^k)$ can be decided by a TM of space cost $O(n^{2k})$.

Savitch's theorem does indeed leave one more fundamental open problem: is the quadratic blowup necessary to go from nondeterministic to deterministic TMs?

Problem 5
Does

$$\text{NSPACE}(s(n)) \subseteq \text{SPACE}(s(n)^c)$$

hold for any $c < 2$?

6.3 L & NL

Althought any TM which read the entire input has space cost at least linear, but it is possible that a TM has sublinear "extra" space cost.

6.3.1 Basic Definitions

Definition 6.3.1 (Input/Work Tape)
An input/work tape TM is a multitape TM whose input is on the first read-only tape and where the other tapes are standard read/write tapes.

Definition 6.3.2 (Space Cost)
The space cost of an input/work tape TM M is the minimal function $s : \mathbb{N} \to \mathbb{N}$ such that on every input of length n, the matchine M scans at most $s(n)$ distinct cells in the work tapes.

6.3.2 Space Complexity Classes

Definition 6.3.3
SPACE(s) is the set of all languages that can be decided by a deterministic input/work tape TM with worse-case space cost bounded above by $O(s)$.
NSPACE(s) is teh set of all languages that can be decided by a nondeterministic input/work tape TM with worst-case space cost bounded above by $O(s)$.

Note that for $s \in \Omega(n)$, the classes SPACE(s), NSPACE(s) are identical to the previous definition.

6.3.3 L & NL

Definition 6.3.4
We define $\mathrm{L} := \mathrm{SPACE}(\log n)$ and $\mathrm{NL} := \mathrm{NSPACE}(\log n)$.

Proposition 6.3.1
$\mathrm{L} \subseteq \mathrm{NL} \subseteq \mathrm{P} \subseteq \mathrm{NP}$.

Proof
It is clear that $\mathrm{L} \subseteq \mathrm{NL}$ and $\mathrm{P} \subseteq \mathrm{NP}$. Thus we argue that $\mathrm{NL} \subseteq \mathrm{P}$. Recall that a TM which halts with space cost $O(s)$ can have at most $2^{O(s)}$ distinct configurations. In particular, any NTM with space cost $O(\log n)$ has time cost at most $2^{O(\log n)}$ and thus the language it decides is in P.

It seems intuitive that some of these inclusions are strict, however, all of them are open problems!

Problem 6
Prove that $L \subsetneq \mathrm{NP}$.

6.4 NL-Completeness

Savitch's theorem shows that $\mathrm{NL} \subseteq \mathrm{SPACE}(\log^2 n)$, but this does not resolve whether $L = \mathrm{NL}$. We would like to study languages which are complete for NL.

6.4.1 Log-Space Reductions

Let us first define a new type of reduction.

Definition 6.4.1 (Log-Space Reducible)
Given two languages $A, B \subseteq \Sigma^*$, the language A is log-space reducible to B, denoted

$$A \leq_L B$$

if there is a function $f : \Sigma^* \to \Sigma^*$ such that

1. For every $x \in \Sigma^*$, we have $x \in A \iff f(x) \in B$
2. There is a logarithmic-space TM M that on input $x \in \Sigma^*$, erases it and replaces it with $f(x)$ on the tape and then halts.

Observation 6.4.1
Recall that a $O(\log n)$-space TM has at most $2^{O(\log n)}$ distinct states and thus must be computable in polynomial time.
The converse does not hold! Indeed, a polytime TM might take more than logarithmic space.

Lemma 6.4.2
For every two languages $A \leq_L B$,

1. $B \in \mathrm{L}$ implies $A \in \mathrm{L}$
2. $B \in \mathrm{NL}$ implies $A \in \mathrm{NL}$

Proof
Let T_B be a TM which decides B in logspace. To decide if a given string x in A, we can simply compute $f(x)$ in logspace and feed it to T_B.

The case for NL is identical.

6.4.2 NL-Completeness

Definition 6.4.2 (NL-Complete)
The language L is NL-complete if

1. Every language $A \in \text{NL}$ satisfies $A \leq_L L$
2. $L \in \text{NL}$

6.4.3 Connectivity

Problem 7 ((s,t)-connectivity)
Given a digraph $G = (V, E)$ and two vertices $s, t \in V$, does there exist a path from s to t in G?

The correponding language is

$$\text{CONN} := \{\langle G, s, t\rangle : \text{there is an } st\text{-dipath in } G\}.$$

Lemma 6.4.3
CONN $\in$ NL.

Proof
This is equivalent to asking if there is an st-dipath of length at most $n := |V|$, since any simple dipath has length at most n.

We can nondeterministically guess a next vertex, and store both a counter for the length of the current path computed, as well as the current vertex. This uses logarithmic space.

If the length of the path exceeds n, we terminate the computation. Since all computation paths terminate, we conclude that the described TM decides CONN in logspace.

Theorem 6.4.4
CONN is NL-complete.

Proof
Let G_x be the configuration graph for a logspace NTM M which decides some $A \in \text{NL}$ acting on the input x of length n. We may assume without loss of generality that M only accepts on an empty tape.

Since M uses $O(\log n)$ space, there are at most $2^{O(\log n)} = O(n)$ distinct configurations. Let s_x denote the initial configuration and t_x the unique accepting configuration.

Define $f(x) := \langle G_x, s_x, t_x \rangle$. Clearly, f is logspace computable. Hence to decide if $x \in A$, it suffices to decide if $f(x) \in$ CONN as desired.

Problem 8
Is CONN in L?

6.5 NL = co-NL

Definition 6.5.1 (coNL)
We define

$$\text{coNL} := \{L \subseteq \Sigma^* : \bar{L} \in \text{L}\}.$$

Proposition 6.5.1
If L = NL, then NL = coNL.

Proof
Consider any NL-complete language A. Observe that $\bar{A}$ is coNL-complete. Since NL = L, there is a logspace deterministic TM M which decides A. The TM which accepts a string if and only if M rejects the string then decides $\bar{A}$ in logspace.

Hence

$$\text{NL} = \text{L} = \text{coNL}$$

as desired.

Theorem 6.5.2 (Immerman-Szelepscenyi)
NL = coNL.

6.5.1 Non-Connectivity

Our goal is to show that there is some NL-complete language L in coNL. Equivalently, since $\bar{L}$ is coNP-complete, we want to show that $\bar{L} \in$ NL.

We do so by considering the complement of CONN.

6.5.2 Non-Connectivity with Advice

Lemma 6.5.3
Let

- G is a directed graph
- $s, t \in V(G)$
- $\ell \in \mathbb{N}$ is a length parameter
- k denotes the number of vertices reachable from s by paths of length at most ℓ in G

There exists a logspace NTM that accepts $\langle G, s, t, \ell, k \rangle$ if and only if there is a path from $s \to t$ of length at most ℓ within G.

Proof
For each vertex $v \in V - t$, we guess if v is reachable from s by a path of length at most ℓ and verify each guess with a single path of length at most ℓ (we might fail to verify a correct guess). Reject if we make any wrong guesses.

Let κ be the number of vertices guessed. If $\kappa = k$, reject. Else if $\kappa = k - 1$, guess a path of length at most ℓ to t and accept if our guess is correct. Otherwise, reject.

Since $\kappa \in O(\log|V|)$, our TM is certainly logspace.

If there is an st-dipath of length at most ℓ, then there is at least one computational path where we accept the string. Conversely, if no such path exists, there is no computational path which leads to acceptance.

6.5.3 The Proof

Theorem 6.5.4
$\overline{\text{CONN}} \in \text{NL}$.

Proof
It suffices to show how we may compute k from the previous lemma.

Note that initially, for $\ell = 0$, we have $k_0 = 1$. Thus suppose we know $k_{\ell-1}$.

Initialize $c := 0$. Then for each arc $uv \in E$, if the previous TM accepts

$$\langle G, s, u, \ell - 1, k_{\ell-1} \rangle,$$

then update $c := c + 1$. Return $k_\ell := c$.

Since we may reuse variables, this algorithm certainly only uses logarithmic space.

6.6 More on Space Complexity

6.6.1 PSPACE-Completeness

Recall that the question of whether P and PSPACE are distinct is currently open.

Definition 6.6.1 (PSpace-Complete)
The language L is PSPACE-complete if

- Every $A \in$ PSPACE satisfies $A \leq_P L$
- $L \in$ SPACE.

Proposition 6.6.1
If L is PSPACE-complete and $P \subsetneq$ PSPACE, then L $\notin$ P.

Proof
If $L \in$ P, then every language in PSPACE is in P as well.

We consider the language of every true totally quantified boolean formulas, denoted TQBF.

Theorem 6.6.2
TQBF is PSPACE-complete.

The P vs PSPACE question would be resolved if we can determine whether TQBF is in P or not.

Problem 9
Prove or disprove that TQBF $\in$ P.

6.6.2 Small Space Complexity Classes

We now turn to languages which use very small amounts of space.

Theorem 6.6.3
The class SPACE(1) is the set of regular languages.

Proof
A TM which uses only constant extra space can be turned into one which uses no extra space by increasing the number of states.

Hence we may simulate such a TM with an NFA.

Problem 10 (Challenge)
For every $s \in o(\log \log n)$, $\text{SPACE}(s) = \text{SPACE}(1)$.

6.6.3 Time & Space

Definition 6.6.2 (SC)
The class SC is the class of languages that can be decided by a TM with time cost $O(n^k)$ and space cost $O(\log^\ell n)$ for some constant $k, \ell \in \mathbb{N}$.

The name SC was chosen to stand for Steve's Class after Stephen Cook for his work on this class.

Definition 6.6.3 (PolyL)
The class

$$\textsc{PolyL} := \bigcup_{\ell \geq 1} \text{SPACE}(\log^\ell n).$$

Proposition 6.6.4
$\text{SC} \subseteq \text{P} \cap \textsc{PolyL}$.

Proof
Let $L \in \text{SC}$ be decided by a TM M in polytime and polylogspace.

Then $L \in \mathrm{P}$ and $L \in \textsc{PolyL}$ by definition. Hence the result holds.

Problem 11 (Open)
Is $\mathrm{SC} = \mathrm{P} \cap \textsc{PolyL}$?