CS 4510 Automata and Complexity January 9th 2023

Lecture 1: Introduction

Lecturer: Abrahim Ladha *Scribe(s): Akshay Kulkarni*

1 Introduction

CS 4510: Automata & Complexity Theory. This course is primarily the study of two questions:

1. What are the limits of computation? (Computability Theory: Approx. 70% of course)
2. What makes some problems easy and others hard? (Complexity Theory: Approx. 30% of course)

Fundamentally about theoretical computer science—what is a computer and how can we effectively describe its limits and capacities? Automata are tools that we can use to reason about more powerful versions.

1.1 Formal Language Theory

Let Σ be a finite set of characters called the *input alphabet*.

Examples:

1. $\Sigma = \{a, b\}, \{1\}, \{0, 1\}, \{a, \ldots, z, A \ldots, Z\}$
2. $\Sigma^2 = \{aa, ab, ba, bb\}$ (strings of length 2)
3. $\Sigma^0 = \{\varepsilon\}$ (strings of length 0)
4. $\Sigma^* = \cup_{i=1}^{\infty} \Sigma^i = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \ldots$ (all strings)

A *language* is any subset of strings $L \subseteq \Sigma^*$

1.2 Automata

A finite automaton is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$:

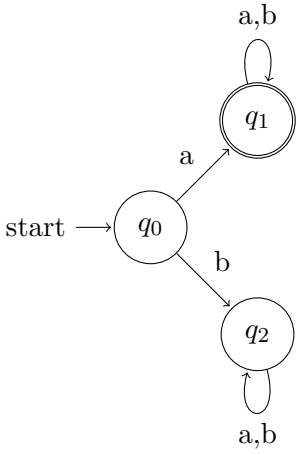
1. $Q = \{q_0, \ldots, q_k\}$ is a finite set called the *states*,
2. Σ is the *alphabet*,
3. $\delta : Q \times \Sigma \to Q$ is the *transition function*,
4. $q_0 \in Q$ is the *start state*
5. $F \subseteq Q$ is the set of *acceptance* states.

An automata M decides a language L if:

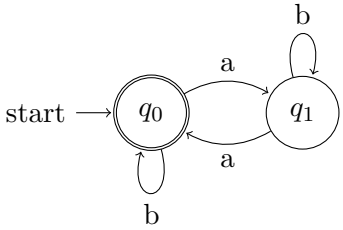
$$M(w) \textit{ accepts } L \iff w \in L \tag{1}$$
$$M(w) \textit{ rejects } L \iff w \notin L \tag{2}$$

Examples:

1. $L = \{w \in \Sigma^* \mid w \text{ begins with } a\}$



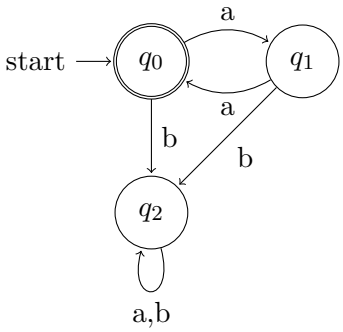
2. $L = \{w \in \Sigma^* \mid \#a(w) \text{ is even}\}$



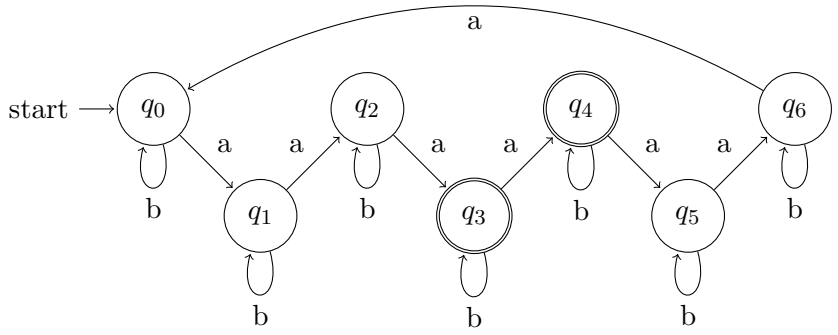
A decision problem is phrased as any yes/no question.

Examples:

1. $L = \{a^n \mid n \text{ is even}\}$



2. $L = \{w \in \Sigma^* \mid \#a(w) \equiv 3, 4 \pmod 7\}$

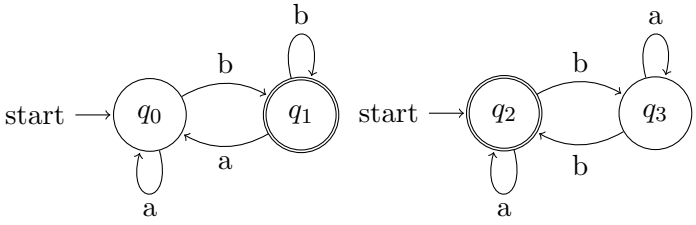


3. $L = \{w \in \Sigma^* \mid \text{int}(w) \text{ is prime}\}$
   We are not concerned with the alphabet really, not for the problems we want to study. Analogously, the primality of an integer has nothing to do with the base it is represented in. We are using languages to talk about decision problems. We have turned computational questions into ones about set membership. If an automata can determine correctly if some $w \in L$, then it certainly has the power of primality testing.

DFA's can be used to simulate two DFA's. For example, consider the languages

$$L_1 = \{w \in \Sigma^* \mid w \text{ ends with a } b\} \tag{3}$$
$$L_2 = \{w \in \Sigma^* \mid \#b(w) \text{ is even}\} \tag{4}$$

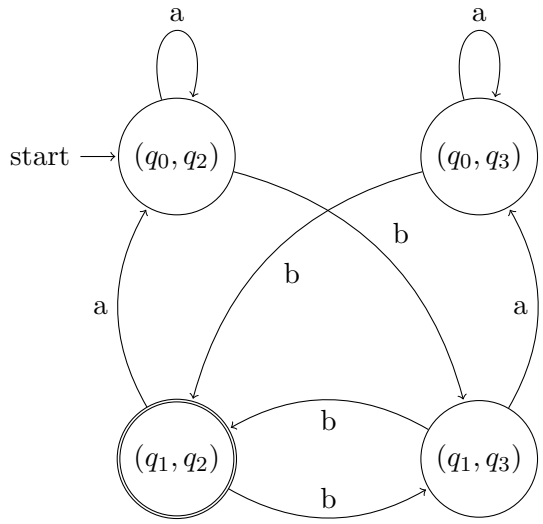
Lets make two DFAs for these languages.



We construct a DFA that simulates two DFAs, to compute $L_3 = L_1 \cap L_2$ Note that L_1 is decidable by the DFA $(\Sigma, Q_1, q_{01}, \delta_1, F_1)$ and L_2 is decidable by $(\Sigma, Q_2, q_{02}, \delta_2, F_2)$. We can use the properties of set intersection to define a DFA that accepts $L_1 \cap L_2 = (\Sigma, Q_3, q_{03}, \delta_3, F_3)$:

1. $Q_3 = Q_1 \times Q_2$
2. Σ is the same
3. $\delta_3((q_i, q_j), a) = (\delta_1(q_i, a), \delta_2(q_j, a))$ for $q_i \in Q_1$ and $q_j \in Q_2$
4. $q_{03} = (q_{01}, q_{02})$
5. $F_3 = F_1 \times F_2$

Out cartesian product DFA then looks like the following.



Note that if we made $F_3 = (Q_1 \times F_2) \cup (F_1 \times Q_2)$ we would have accepting states for the union of our two languages.

1.3 Regularity

A language is regular if and only if it is recognized by a DFA. We write the set of languages decidable by a DFA as $\mathscr{L}(DFA)$

CS 4510 Automata and Complexity January 11th 2023

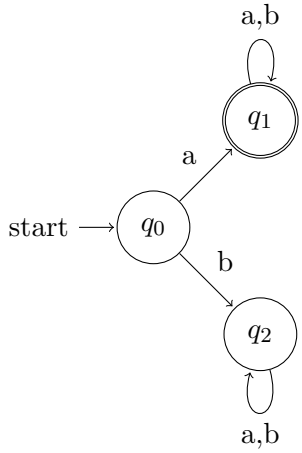
Lecture 2: Nondeterminism

Lecturer: Abrahim Ladha *Scribe(s): Samina Shiraj Mulani*

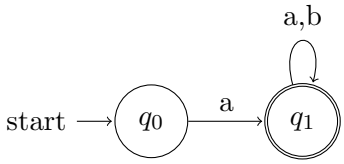
1 Introduction

We noted that DFAs are weak. Let's try to extend or generalize them. A DFA can be represented as $(Q, \Sigma, \delta, q_0, F)$. When thinking about extending DFAs, the only useful thing to extend is the way in which states interact with each other, i.e., δ.
Let's extend δ in 3 ways:

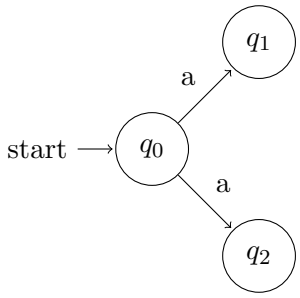
1. If a transition is undefined, we implicitly reject. As an example, Consider the following DFA which represents the language $\{w \in \Sigma^* \mid w \text{ begins with } a\}$



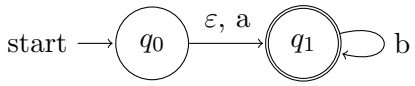
   We can now represent this as



2. Allow transitions of more than one of the same type. This means that you can have multiple outgoing transitions with the same input. For example



3. Allow "ε-transitions", which can be taken for free. For example



   a, ab, abb, b, bb, ε are some of the strings accepted by this NFA.

2 Coping with nondeterminism

Its important to understand nondeterminism and not just have deterministic coping strategies. We say a nondeterministic computation accepts if at least one computation path reaches an accept state. The following analogies help in visualizing this aspect -

1. **Depth First Search** DFS on the DFA until you reach an accept state.
2. **Lucky Coin** Imagine you flip a lucky coin that tells you exactly which fork/path to take.
3. **Alternate timelines** For each nondeterministic action, create multiple timelines, similar to creating multiple copies.

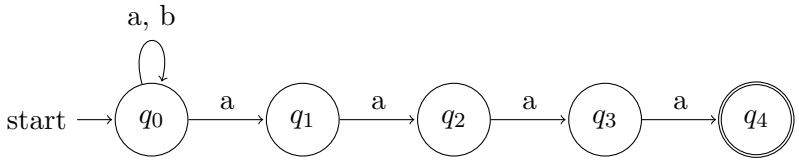
3 Definition and a few examples

A Nondeterminisitic Finte Automata (NFA) can be represented by a 5-tuple $(\Sigma, Q, q_0, \delta, F)$ where:

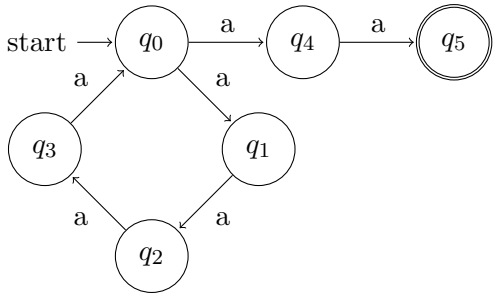
1. Σ - finite alphabet
2. Q - finite set of states
3. q_0 - denoted start state
4. δ - $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to \mathcal{P}(Q)$
   $\mathcal{P}(Q)$ represents the power set of Q, which is the set of all subsets of Q. The power set of a set Q has 2^n elements where n is the number of elements in Q.
5. F - the set of final states $F \subseteq Q$

Lets show a few examples

1. $L_1 = \{w \in \Sigma^* \mid w \text{ ends with } aaaa\}$



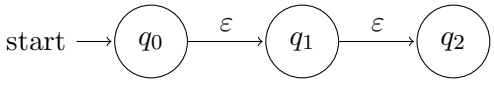
2. $L_{x,y} = \{a^{xn+y} \mid n \in \mathbb{N}\}$
   Lengths of the strings in this language form an Arithmetic Progression. We can show that there exists an NFA for every x, y. Note that the loop is of length x while the tail (q_4 to q_5 in the representation) is of length y.



Every DFA is an NFA, i.e., $\mathscr{L}(DFA) \subseteq \mathscr{L}(NFA)$. For every NFA, we will show you can construct an equivalent DFA. This means that $\mathscr{L}(NFA) \subseteq \mathscr{L}(DFA)$. Combining the aforementioned point, we get $\mathscr{L}(DFA) = \mathscr{L}(NFA)$ Note that when we perform a complement of the accept states in a DFA that represents a language L, we get the complement of the language. The same doesn't hold for an NFA due to the presence of the implicit reject state.

4 $\mathscr{L}(NFA) \subseteq \mathscr{L}(DFA)$

We simulate an NFA on a DFA. Each state of the DFA corresponds to any number of states of the NFA. Sure, a NFA can be in many states, but only finitely many. To each possible set of states, we will simulate that on our DFA as a single state. We first define the concept of reach.
$reach(q_i) = \{q_i \text{ and any state reachable from } q_i \text{ by } \varepsilon \text{ transitions}\}$. For example

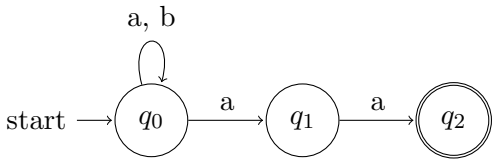


Then $reach(q_0) = \{q_0, q_1, q_2\}$. Let the NFA $N = (\Sigma, Q, q_0, \delta, F)$ Make $D = (\Sigma', Q', q_0', \delta', F')$

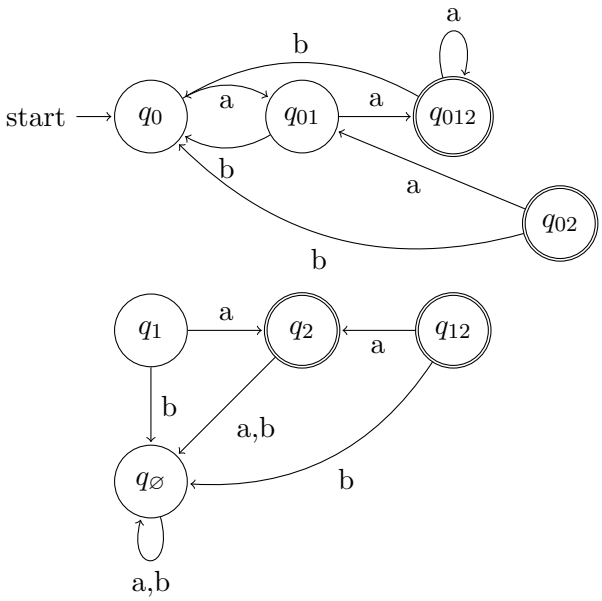
- $\Sigma' = \Sigma$
- $Q' = \mathcal{P}(Q)$

- $q_0{}' = reach(q_0)$
- $\delta'(\{q_1, ..., q_k\}, a) = \bigcup_{i=1}^{k} reach(\delta(q_i, a))$
- $F' = \{f \subseteq Q \mid f \cap F \neq \varnothing\}$

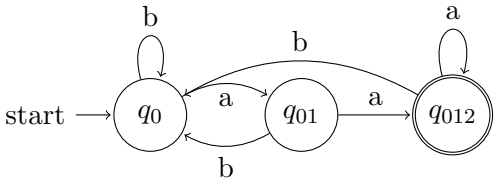
Example -
$L_2 = \{w \in \Sigma^* \mid w \text{ ends with } aa\}$



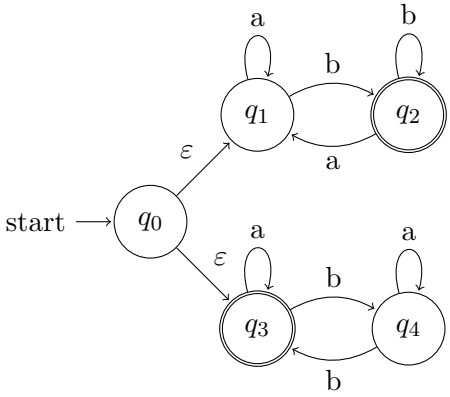
By following the above algorithm, we get the corresponding DFA



We observe that there are unreachable states (example - q_{02}). So the algorithm may or may not give a minimal DFA. On cleaning up these unreachable states, we get the following DFA



Each state represents a superposition of the states in the NFA. One utility of NFAs is that we can use them to create a more convenient representation of the union of two languages. Consider $L_3 = \{w \in \Sigma^* \mid w \text{ ends with } b\}$ and $L_4 = \{w \in \Sigma^* \mid \#b(w) \text{ is even}\}$ We can construct the following NFA that represents $L_3 \cup L_4$



We can also use this idea to prove that the union of 2 regular languages is always regular. An alternate is to follow last lecture's approach using Cartesian Product, which can be comparatively more cumbersome.

5 The Road Not Taken

By Robert Frost, emphasis mine

Two roads diverged in a yellow wood,
And sorry **I could not travel both**
And **be one traveler,** long I stood
And looked down one as far as I could
To where it bent in the undergrowth;
Then took the other, as just as fair,
And having perhaps the better claim,
Because it was grassy and wanted wear;
Though as for that the passing there
Had worn them really about the same,
And both that morning equally lay
In leaves no step had trodden black.
Oh, I kept the first for another day!
Yet knowing how way leads on to way,
I doubted if I should ever come back.
I shall be telling this with a sigh
Somewhere ages and ages hence:
Two roads diverged in a wood, and I—
I took the one less traveled by,
And that has made all the difference.

The moral of this poem in the context of our lecture is that Robert Frost is a deterministic actor, one who sees two roads and is forced to choose. If he was nondeterministic, he wouldn't have to choose. He could come to a fork in the road and just take it.

CS 4510 Automata and Complexity January 18th 2023

Lecture 3: Regular Expressions

Lecturer: Abrahim Ladha *Scribe(s): Yitong Li*

1 Regular Expressions

In the terminal, we usually enter "`ls *.pdf`" to search for every single file ending with ".pdf". Such language is called regular expressions.

Definition 1.1. We say that R is a **regular expression**, or regex, if R is one of the following:

(a) $\emptyset$ - empty set

(b) ε - empty string

(c) $a \quad \forall a \in \Sigma$

(d) R_i^*, R_iR_j or $R_i \cup R_j$ where R_i, R_j are regular expressions.

$\diamondsuit$

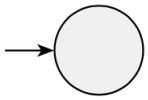
Our first three, our base cases. These are actually shorthand for the sets $\emptyset, \{\varepsilon\}, \{a\}$. Here are some examples of regular expressions:

1. $a^* = \{a^i \mid i \in \mathbb{N}\} = \{\varepsilon, a, aa, aaa, \dots\}$
2. $a^*ba^* = \{a^iba^j \mid i, j \in \mathbb{N}\}$ = all strings with a single b.
3. $\Sigma^*aab\Sigma^* = \{$all strings with aab as a substring$\}$
4. $(a \cup b)^*aab(a \cup b)^*$
5. $(\Sigma\Sigma)^* = ((a \cup b)(a \cup b))^* = ((aa \cup ab \cup ba \cup bb))^*$ = all strings of even length
6. $L_1L_2 = \{xy \in \Sigma^* \mid x \in L_1, y \in L_2\}$
7. $a^*\emptyset = \emptyset$
8. $\emptyset^* = \{\varepsilon\}$

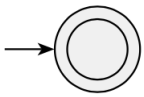
2 $\mathscr{L}(REX) \subseteq \mathscr{L}(NFA)$

To prove that $\mathscr{L}(REX) \subseteq \mathscr{L}(NFA)$, we want to show that for each regular expression, there exists an equivalent NFA. Given that regular expressions are recursively defined, it is natural to choose to proceed by induction. Let R be a regular expression. We start with the following base cases:

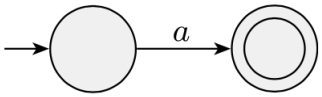
1. $R = \emptyset$.



2. $R = \varepsilon$.

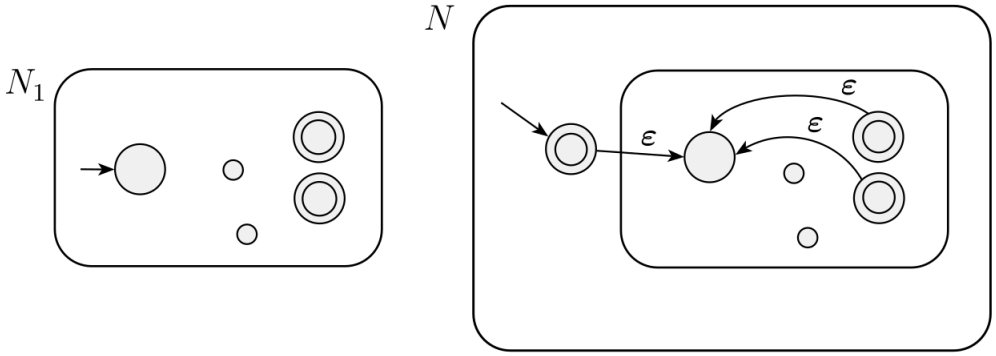


3. $R = a \in \Sigma$.



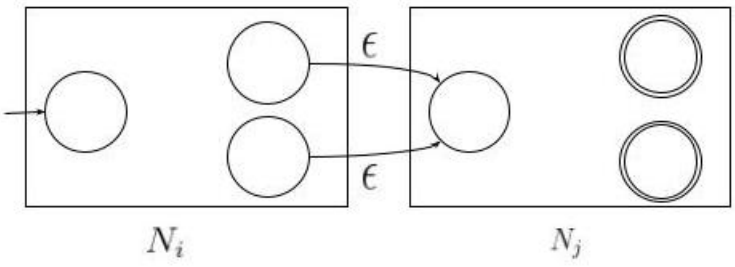
Next, we continue with the inductive steps. Let R_i, R_j be regular expressions that decide regular languages, by strong induction. We will show $R_i^*, R_iR_j, R_i \cup R_j$ are also regular. Since R_i, R_j are regular, there exist NFAs N_i, N_j to decide $L(R_i), L(R_j)$.

1. $R = R_i^*$.



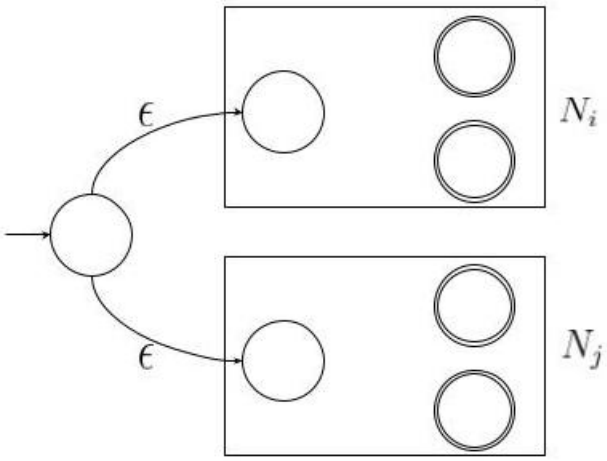
add new state q', ε-transition from q' and all states of F to the old start state q, mark q' as accepting.

2. $R = R_iR_j$.



remove final states F_i and $\forall f \in F_i$, add $\delta(f, \varepsilon) = q_j$ where q_j is the initial state of N_j.

3. $R_i \cup R_j$.



add new start state q and $\delta(q, \varepsilon) = \{q_i, q_j\}$.

3 Example from REX to NFA

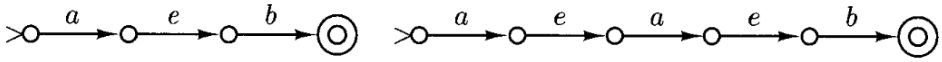
- $(ab \cup aab)^*$

stage 1

a; b

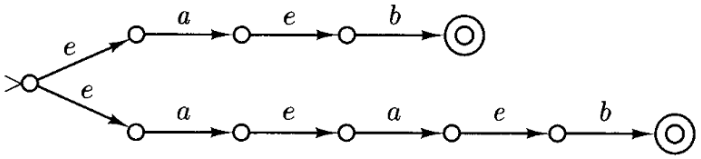
stage 2

ab; aab



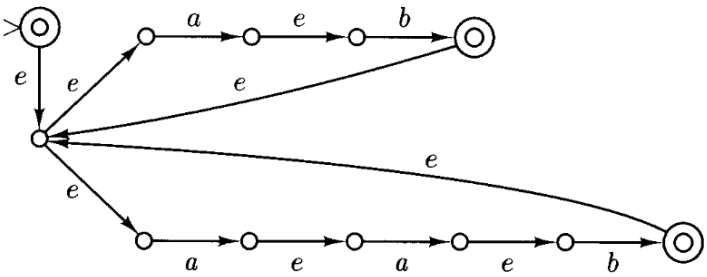
stage 3

$ab \cup aab$



stage 4

$(ab \cup aab)^*$



4 $\mathscr{L}(NFA) \subseteq \mathscr{L}(REX)$

Definition 4.1. The **GNFA** is defined as an NFA with the following properties:

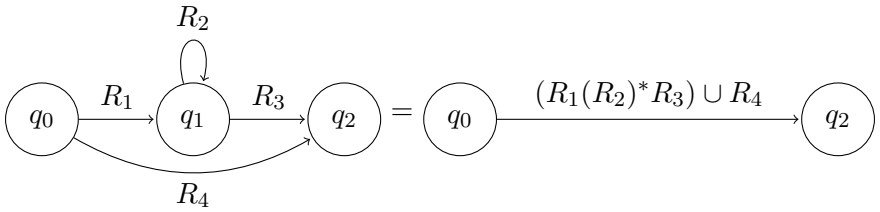
(a) The transitions have a regular expression on them.

(b) The start state has no incoming transitions

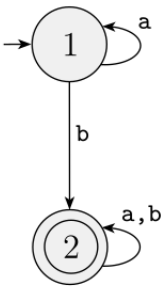
(c) The final state has no outgoing transitions

(d) Every pair of states has a transition.

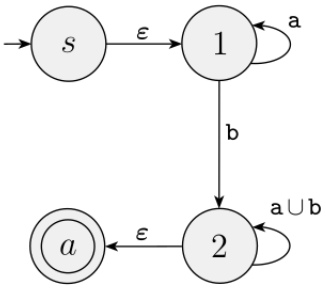
$\diamondsuit$ Taking a transition in a DFA is reading some single symbol of the front of the input. Taking a transition of a GNFA is nondeterministically choosing some prefix of the input which satisfies the regex on the transition. To convert an NFA to a regular expression, we first add a new start and final state. Then, rip out one state at a time using the following rules until only two states and one transition is left.



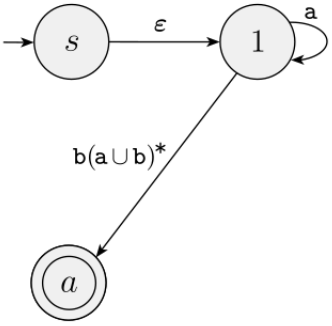
For most very connected NFAs, conversion to a regex will result in one with exponential length. Here is a relatively simple example.



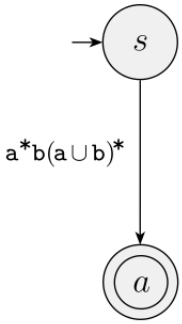
(a)



(b)



(c)



(d)

5 Set Closure

We say a set S is **closed** under an operation Δ if $\forall a, b \in S$, $a \Delta b \in S$.

1. $\mathbb{N}$ is closed under $+, \times$ and not closed under $-, \div$.
2. $\mathbb{Z}/\{0\}$ is closed under $+, -, \times$ and not closed under $\div$.
3. $\mathbb{Q}$ is closed under $+, -, \times$ and not closed under $\div$.
4. Regular languages are closed under $*, \circ, \cup$ and complement.

Although regular languages are closed under complement. Complement is not a valid operation of a regular expression. Using the operations regular languages are known to be closed under, we can prove closure under even more operations without having to contruct messy DFAs or NFAs. Recall we did a cartesian product of DFAs to prove that regular languages were closed under intersection. We give a shorter proof in the syntax of set theory.

Suppose that L_i, L_j are two regular languages. Then surely $\overline{L_i}, \overline{L_j}$ are regular, then surely $\overline{L_i} \cup \overline{L_j}$ is regular. Then so must be $\overline{\overline{L_i} \cup \overline{L_j}}$. From Demorgans law, we know that $\overline{\overline{L_i} \cup \overline{L_j}} = L_i \cap L_j$.

Similarly, we can show regular languages are closed under symmetric difference, or xor. We have a formula under composition of operators that we maintain closure under. $L_i \oplus L_j = (\overline{L_i} \cap \text{Ł}_j) \cup (L_i \cap \overline{L_j})$

CS 4510 Automata and Complexity Janurary 23rd 2023

Lecture 4: The Pumping Lemma

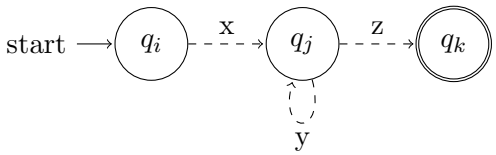
Lecturer: Abrahim Ladha *Scribe(s): Michael Wechsler*

1 Background

We previously mentioned that we have some intuition on the limitations of DFAs.

- ex) $\{a^n b^n \mid n \in \mathbb{N}\}$ should not be possible with finite memory since it needs a counter

Suppose we have a DFA, D, made up of p states. Consider a word, w, such that $|w| = p+1$. When we simulate $D(w)$, consider the sequence of states visited when deciding if $w \in L(D)$



By the Pigeonhole Principle, some state (q_j above) appears twice in this sequence, and our DFA must contain a loop. The details of the loop are not important, we just want to know that it exists.

⋆ Pigeonhole Principle: if you have $p+1$ pigeons and p pigeonholes, there must be at least 1 pigeonhole with greater than 1 pigeons in it

Claim: If $xyz \in L, \forall i, xy^i z \in L$ due to the DFA pictured above

Important Fact: If a language is regular, then it can be pumped

- Contrapositive (Equivalent): If a language cannot be pumped, then it is not regular
- Converse (**NOT** Equivalent): If a language can be pumped, then it is regular
  - This is not true and will be seen at the end of note section 3.2

2 Formula

The pumping lemma has many moving pieces and can be tricky to apply. I suggest you use this formula exactly. L is the language we want to prove is not regular.

1. Assume to the contrary, L is regular with pumping length p (pumping length is the number of states in L's DFA)
2. Choose some $s \in L$ such that $|s| \geq p$

3. For all cases $s = xyz$ such that $|xy| \leq p$ and $|y| > 0$
4. Choose $i \neq 1$ such that $xy^iz \notin L$
5. Conclude that L cannot be pumped, which means L is not regular

We require that $|s| \geq p$ so we have the pigeonhole condition, and we can pump the string. We require that $|y| > 0$ so we pump something non-trivial. For every language, you can pump $y = \varepsilon$ since its true that $\forall i\ \varepsilon^i = \varepsilon$. We require that $|xy| \leq p$ so that our pigeonhole condition occurs somewhere in what we have denoted as xy.

3 Examples

3.1 $L_1 = \{0^n1^n \mid n \in \mathbb{N}\}$

1. Assume to the contrary, L_1 is regular with pumping length p
2. Let $s = 0^p1^p$ and notice that $s \in L_1$ and $|s| = 2p \geq p$
3. There is only 1 case since the first p characters in the string are all 0s
   $x = 0^a,\ y = 0^b,\ z = 0^{p-a-b}1^p$ subject to $|xy| = a + b \leq p$ and $|y| = b > 0$
4. Choose $i = 2$
   $xy^iz = xy^2z = xyyz = 0^a0^b0^b0^{p-a-b}1^p = 0^{p+b}1^p$
5. We know that $b > 0$, so the number of 0s does not equal the number of 1s since $p + b > p$. Thus, L_1 cannot be pumped, and as a result, is not regular.

3.2 $L_2 = \{ww^R \mid w \in \Sigma^*\}$ (Even-length palindromes)

$\star$ Note: a string "raised" to R is the reverse of the string

1. Assume to the contrary, L_2 is regular with pumping length p
2. Let $s = 0^{p-1}110^{p-1}$ (We are choosing a poor s on purpose)
   Confirm that $s \in L_2$ and $|s| = 2p \geq p$
3. The first p characters in the string are different, meaning there are several cases (2)
   (a) $x = 0^a,\ y = 0^b,\ z = 0^{p-1-a-b}110^{p-1}$
   Subject to $|xy| = a + b \leq p$ and $|y| = b > 0$
   (b) $x = 0^a,\ y = 0^{p-1-a}1,\ z = 10^{p-1}$
   Subject to $|xy| = a + p - 1 - a + 1 = p \geq p$ and $|y| = p - 1 - a + 1 > 0$
4. Choose i for each case
   (a) Choose $i = 2$
   $xy^2z = xyyz = 0^a0^b0^b0^{p-1-a-b}110^{p-1} = 0^{p-1+b}110^{p-1}$
   Since $b > 0$, we know that $p - 1 + b \neq p - 1$. Therefore, the two sections of 0s are unequal and the string is not a palindrome

(b) Choose $i = 0$
$xy^0z = xz = 0^a10^{p-1}$
Since there is only one 1, this is not an even-length palindrome

5. For both cases, the language could not be pumped. Therefore, L_2 is not regular.

⋆ Note: Letting $s = 0^p0^p$ would result in a successful pump, but this does not mean the language is regular. Try to choose s carefully to avoid this situation.

3.3 $L_3 = \{\ ww \mid w \in \Sigma^*\}$, assume $\Sigma = \{0, 1\}$

1. Assume to the contrary, L_3 is regular with pumping length p
2. Let $s = 0^p10^p1$ and notice that $s \in L_3$ and $|s| = 2p + 2 \geq p$
3. There is only 1 case since the first p characters in the string are all 0s
$x = 0^a,\ y = 0^b,\ z = 0^{p-a-b}10^p1$
Subject to $|xy| = a + b \leq p$ and $|y| = b > 0$
4. Choose $i = 2$
$xy^2z = xyyz = 0^a0^b0^b0^{p-a-b}10^p1 = 0^{p+b}10^p1$
Take the right-most $p + 2$ characters in xy^2z. This string, which we'll call $w_2 = 10^p1$.
Now, there are two cases for the leftmost string, $w_1 = 0^{p+b}$.
   (a) If $b = 1$, xy^2z is not even length, and therefore not in L_3
   (b) If $b > 1$, the midpoint of $xy^2z = w_1w_2$ is in the first block of 0s. We can tell that $w_1 \neq w_2$, and therefore, xy^2z is not in L_3
5. Both cases end with the pumped string not being in L_3. Thus, L_3 cannot be pumped and is not regular.

3.4 $L_4 = \{0^n1^n2^n \mid n \in \mathbb{N}\}$ - A note on choosing a BAD s

- Assume to the contrary, L_4 is regular and with pumping length p
- Let $s = 0^{\lfloor p/3 \rfloor + 1}1^{\lfloor p/3 \rfloor + 1}2^{\lfloor p/3 \rfloor + 1}$
$s \in L_4$ and $|s| > p + 1 \geq p$
- There are 6 cases for this string

0.....01.....12.....2
x	y	z
x	y	z
x	y	z
x	y	z
x	y	z
x	y	z

There are too many cases. Choose a better s instead of continuing this proof. For each of the six cases, you would be required to construct x, y, z choose i, and show $xy^iz \notin L$. It can be easy to miss a case. A better choice is $s = 0^p1^p2^p$ which has a similar proof as shown previously.

Lets do a unary example.

3.5 $L_5 = \{1^{n^2} \mid n \in \mathbb{N}\}$

1. Assume to the contrary, L_5 is regular with pumping length p
2. Let $s = 1^{p^2}$ and observe that $s \in L_5$ and $|s| = p^2 \geq p$
3. There is only 1 case since the first p characters in the string are all 1s
   $x = 1^a,\ y = 1^b,\ z = 1^{p^2-a-b}$
   Subject to $|xy| = a + b \leq p$ and $|y| = b > 0$
4. Look at $i = 2$
   $xy^2z = xyyz = 1^a1^b1^b1^{p^2-a-b} = 1^{p^2+b}$

   Since $b > 0$, $p^2 + b > p^2$
   Since $a + b \leq p$, $b \leq p$
   Thus $p^2 + b \leq p^2 + p < p^2 + p + (p + 1) = p^2 + 2p + 1 = (p + 1)^2$

   By the first and third lines, we know $|1^{p^2}| < |1^{p^2+b}| < |1^{(p+1)^2}|$
5. By the last line, we can see that xy^2z falls between two adjacent strings in L_5. Therefore, its length is not some perfect square and is not in L_5. Thus, L_5 cannot be pumped and is not regular.

CS 4510 Automata and Complexity Janurary 25th 2023

Lecture 5: Context-Free Grammars

Lecturer: Abrahim Ladha *Scribe(s): Rishabh Singhal*

1 Background

Automata So far we have only looked at automata. These are usually tasked with **Decision or Recognition**. It's a fairly mechanical model, a decision procedure. You look at the input scanning left to right and do something.

- Given $w \in \Sigma$, is $w \in L(D)$? This is not that hard, you just run the automata on the input.
- Characterizing all of $L(D)$? This is much harder for an automata. If I give you a DFA or NFA and ask you to describe exactly the strings it accepts, this is not as easy.

Grammars In contrast, a grammar is tasked with **Production or Generation**. A grammar will non-deterministically produce only the correct strings, like a flower blooming. It doesn't start with an input to look at, it starts with nothing. Defined with the rules we give it, it will produce a string according to those rules.

- Given $w \in \Sigma$, is $w \in L(G)$? This is surprisingly non-trivial
- Characterizing all of $L(G)$? This is surprisingly easier.

2 Formal Definition of Context-Free Grammar

We represent a context-free grammar (CFG) as a four tuple (V, Σ, R, S) such that:

- V Non-Terminals or Variables. These are always capitalized like $\{S, A, B, ...\}$
- Σ Terminals or our alphabet. These are always lower-case like $\{a, b, c, ...\}$
- R Productions or Rules. Each are of them will be of form $V \to (V \cup \Sigma)^*$. The left-hand side of the production will be a single non-terminal and the right-hand side will be a string of terminals and non-terminals.
- $S \in V$ is our designated start non-terminal.

For $A \in V, w \in (V \cup \Sigma)^*$, with production of the form $A \to w$, we apply a production as a substring replacement of a "working string" like $xAz \implies xwz$, for $x, z \in (V \cup \Sigma)^*$. When we write $w_i \implies w_{i+1}$ we mean that w_i "yields" w_{i+1} after application of one production. If $S \implies w_1 \implies w_2 \implies w_3 ... \implies w_n$ with $w \in \Sigma^*$ we say that $w \in L(G)$ and may write $S \overset{*}{\implies} w$.

For a context-free grammar G, we characterize the set of strings in $L(G)$ as those and only those produced non-deterministically starting from S. Observe that a production halts when there are no more non-terminals in the working string. We say that a language L is context-free if ther exists a context-free grammar G such that $L = L(G)$.

2.1 Examples

Like a state diagram, you can give all parts of the CFG by just giving the set of productions. It implicitly gives the terminals and non-terminals, and we always denote S as the start non-terminal.

2.1.1 $\{a^n b^n \mid n \in \mathbb{N}\}$

We write $\{S \to aSb, S \to \varepsilon\}$ or just $\{S \to aSb \mid \varepsilon\}$. If we have two or more productions with the same beginning non-terminal, we may use "|" as a shorthand to "or" those productions together. Let us say we want to produce a^3b^3 the process we follow is

$$S \implies aSb \implies a(aSb)b \implies aaSbb \implies aaaSbbb \implies aaabbb \implies a^3b^3$$

We repeatedly apply the first production, and terminate when we have no more non-terminals in our working string. This occurs when we apply the second rule, $S \to \varepsilon$. Notice that it has to produce exactly the strings of the form $a^n b^n$. This was our canonical example of a non-regular language, the first one we used for pumping. This should convince you atleast, that the languages produced by context-free grammars, $\mathscr{L}(CFG)$, is not equal to the regular languages. Later we will show it is a strict super set.

2.1.2 $\{ww^R \mid w \in \Sigma^*\}$

Our productions are similar. $\{S \to aSa \mid bSb \mid \varepsilon\}$. This generates even length palindromes. As we apply productions, the left and right of our primary recursive production effectively act like two stacks, mirrors of each other. This generates the string which is a palindrome and these strings are also even in length. We can conserve the same idea, to generate palindrome of odd length.

2.1.3 $\{w\Sigma w^R \mid w \in \Sigma^*\}$

We write this as $\{S \to aSa \mid bSb \mid a \mid b\}$. We may combine ideas from the previous two examples to show the set of all palindromes is a context-free language, with the grammar $\{S \to aSa \mid bSb \mid a \mid b \mid \varepsilon\}$. We pumped a third language, $\{ww \mid w \in \Sigma^*\}$. As some foreshadowing, this language is not regular, but it is also not context free.

2.1.4 Σ^*

There exist many equivalent grammars for this language. These may include

- $S \to aSa \mid bSb \mid aSb \mid bSa \mid a \mid b \mid \varepsilon$
- $S \to aaS \mid abS \mid baS \mid bbS \mid a \mid b \mid \varepsilon$
- $S \to aS \mid bS \mid \varepsilon$

2.1.5 $\emptyset$

If a grammar produces no strings, not even ε, it is either trivial, or some how does not have a halting condition. There are a few you could come up with, but a non-trivial grammar for this would be $\{S \to A, A \to S\}$. No production of this terminates.

2.1.6 Dyck Language

Consider the grammar $\{S \to (S) \mid SS \mid \varepsilon\}$. This language is the set of balanced, or matching paranthesis. It has a special name, called the Dyck language.

We can prove it is not regular by closure. Assume to the contrary $L(G)$ was regular. Then by closure, so must be $L(G) \cap (^*)^*$. The left side enforces that the number of opens equals the number of closes, and the right hand side enforces that all the opens come before all the closes. The intersection is the logical and of these, so we see this intersection must be equal to $\{(^n)^n \mid n \in \mathbb{N}\}$, our canonical non-regular language, a contradiction. Therefore, the Dyck language is not regular.

2.1.7 Arithmetic Expressions

Consider the following grammar:

$$\begin{aligned} S &\to S + T \mid T \\ T &\to T \times F \mid F \\ F &\to (S) \quad \mid a \end{aligned}$$

with $V = \{S, T, F\}, \Sigma = \{(,), \times, +, a\}$. Lets do an example of a long production to show this grammar generates $(a + a) \times a$

$$\begin{aligned} S \implies T \implies T \times F \implies F \times F \implies (S) \times F \implies \\ (S) \times a \implies (S + T) \times a \implies (T + T) \times a \implies (F + T) \times a \implies \\ (F + F) \times a \implies (F + a) \times a \implies (a + a) \times a \end{aligned}$$

2.1.8 One last example

On the homework, you were asked to pump the language $\{a^n b a^m b^{n+m} \mid n, m \in \mathbb{N}\}$. First notice that for some n, m that $a^n b a^m b^{n+m} = a^n b a^m b^n b^m$. We have matching blocks of the same size, but we can't pair them up as written. We notice that letters of the same kind obviously commute, so we see $a^n b a^m b^n b^m = a^n b a^m b^m b^n = a^n (b a^m b^m) b^n$. This gives us the intuition on how we would build our grammar as $\{S \to aSb \mid bR, R \to aRb \mid \varepsilon\}$. Just to work out some productions, they may look like

$$S \overset{*}{\implies} a^n S b^n \overset{*}{\implies} a^n b R b^n \overset{*}{\implies} a^n b a^m R b^m b^n \overset{*}{\implies} a^n b a^m b^m b^n = a^n b a^m b^{m+n}$$

3 Relationship with Regular Languages

We say a grammar is right-regular if it only has productions of the form $A \to aB$ or $A \to a$ or $A \to \varepsilon$, where A, B are any non-terminals, and a is any terminal. Certainly every right-regular grammar is also context-free, we claim that the right-regular grammars decide exactly the regular languages. The proof of this characterization is not complicated, but tedious[1]. Instead we will highlight just the part of given a DFA, how one might construct a right-regular grammar. This should convince you that we are working with a strictly more powerful computational model, $\mathscr{L}(DFA) \subsetneq \mathscr{L}(CFG)$. For a DFA of the form $(Q, \Sigma, q_0, \delta, F)$ we construct a grammar (V, Σ, R, S).

- For $Q = \{q_0, ..., q_k\}$ we have non-terminals $V = \{Q_0, ..., Q_k\}$
- The set of terminals for our grammar is identical to the alphabet for our DFA: $\Sigma = \Sigma$
- For q_0 the start state of our DFA, we designate our start non-terminal as Q_0
- For every transition of the form $\delta(q_i, a) = q_j$, we add production $Q_i \to aQ_j$
- For every $q_f \in Q$, we add production $Q_f \to \varepsilon$

Convince yourself of its correctness.

4 Closure of Context-Free Language

We will prove that CFLs are closed under union, concatenation, and star. Let G_1, G_2 be two CFGs to produce $L(G_1)$ and $L(G_2)$ with start non-terminals S_1, S_2 respectively.

$L(G_1) \cup L(G_2)$ Copy all productions, add new start state S, and a new production $S \to S_1 \mid S_2$

$L(G_1)L(G_2)$ Similarly, with new production $S \to S_1 S_2$

$L(G_1)^*$ Add new productions $S \to S_1 S \mid \varepsilon$

Later we will show CFLs are not closed under intersection or complement. This may be intuitive, if you observe the behavior of a CFG. It only knows how to grow correct strings.

[1] I have a more detailed proof here https://ladha.me/files/sectionX/regulargrammars.pdf

CS 4510 Automata and Complexity January 30th 2023

Lecture 6: Syntactic Structures

Lecturer: Abrahim Ladha *Scribe(s): Akshay Kulkarni*

1 Syntactic Structures

1.1 Language

Linguistics is the study of language. This includes its structure, etymology, history, everything. Its systematic. Fundamentally, language is communication. Suppose we have a listener and a speaker, and the speaker says "horse". The way we represent that idea syntactically, as a word or utterance, can have nothing to do with the idea itself. That's language. Language is an agreement we all have about what things mean. "Horse" is nothing. "Horse" is an utterance I make by moving vocal cords and exhaling air at a frequency you pick up with your ears.

The concept, the semantic value of "horse" is vastly more complicated than the sound or word itself. I say or utter "horse" and in our mind, you create an image of a herbivore quadruped, maybe its brown. From an information theory perspective, this is a lossy channel, as the idea contains more information than the word.

While animals have certainly demonstrated ability to understand and mimic language to a reasonable extent, humans are the only beings capable of implementing complex language. You can talk to animals, but they won't talk back. Some argue that language is central to humanity's evolutionary identity. As a species, or ability to communicate ideas is exactly what makes us human. It wasn't the upright posture, larger prefrontal cortex, opposable thumbs, toolmaking, no. It was the ability for us to come to consensus and work together on complex tasks. If I, as some paleolithic ape generate the idea "I go hunt mammoth". This idea is totally worthless, I go hunt mammoth the outcome is I get trampled. Instead I communicate this idea and now its "we go hunt mammoth", suddenly its more serious. First we took down a mammoth, and second, we built a computer. Theres some steps in between those two.

Since all ideas must be expressed as language, it was an old-world view that the study of language itself was the only way to study ideas. The study of language was the study

of everything. One of Chomskys accomplishments was to help separate these two. Syntax, the structure of language, and semantics, the meaning of language, are not interchangable.

1.2 Syntactic Structures

The importance of Chomsky's short monograph was not that he solved language in general, but rather he came into someone elses house with more math than them. He came into a very empirical field, and brought in an as theoretical as possible perspective. Using relatively simple intuitive arguments, he was able to make true generalizations about what is an incredibly complex system.

1.3 Chomskyan view of language

Consider a baby. It it not born speaking any language. Googoo gaga and so on. Totally unintelligible. Although a baby is not born knowing any language, it somehow knows how to learn a language. Airdrop that baby into a group of people speaking a language and as it mentally develops, it will learn how to speak among them. This would be independent of anything about the structure of the language. It would not necessarily learn in school what nouns and verbs are in order to speak. Syntax is an innate aspect of language determined by a "Universal Grammar". There are biological conditions which shape the structure of language. The way our brains have the wires and pipes cause limitations in what possible structures language must take.

To study language, it is okay for us to limit ourselves to english. Languages share many universal features. For example, delimination with a space. Have you ever thought about why sentences come in lists and not trees or some other structure in which may not have a kind of topological sort? All grammatical operations appear to be binary as well. The set of grammatical sentences appears to be infinite L, but appears to be contructed recursively from some sort of finite atomic pieces, like a basis. In english, that would be our alphabet, Σ. Secondly, for any english specific artifacts we may encounter, there are almost certainly analoguous issues in other languages.

As we develop a theoretical model for grammar, it is sufficient for us just to short the ability to distinguish the grammatical from the ungrammatical. Such a device or structure which can help us separate these two, can also help us generate new grammatical sentences.

1.4 The Independence of Grammar

Consider the following two sentences:

1. Colorless green ideas sleep furiously.
2. Furiously sleep ideas green colorless.

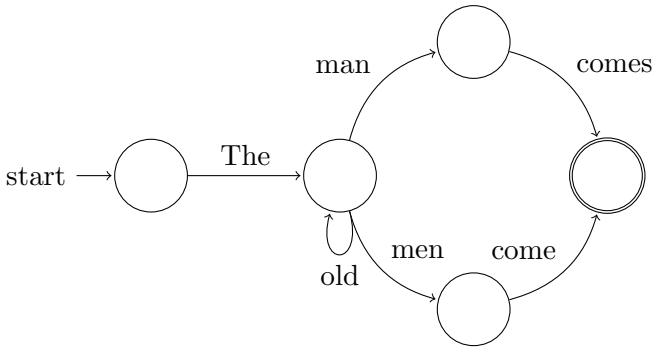
Let us study the first sentence. Certainly it is grammatical. Somehow in our brain exists a distinguisher, and we can read this sentence and come to the consensus that it is grammatically correct. Next of note is that the sentence is totally devoid of meaning. What would the subject the sentence be? Ideas? and they somehow can sleep? and do so furiously? Its colorless, yet green? It has no semantical value, and does not communicate

any idea (besides perhaps confusion). This is a kind of counter example, and forces us to separate syntax, the structure of language, from semantics, the meaning of language.

Second, note that the first sentence is grammatical, but the second one is not. The first, simply by intonation and word pattern seems comfortable. It would be easier to remember and recite. The second is ungrammatical, and troubling. Yet, these two sentences, the frequency of sequential word choices is equivalent because one is a word reversal of the other. These two sentences have been uttered equally likely in all of english, that is, a negligible amount[1]. This is our second observation. The ability of a syntactic structure to distinguish the grammatical from the ungrammatical must be independent of the sentences proximity to english. The first being grammatical, and the second not. Any model based on probability may be unable to distinguish these two based on this kind of frequency alone, but we argue, must be able to. It is also quite likely that sentences you come across have never been said before. Even the sentences you are reading now. Something like 15% of Google's daily searches have never been searched before.

1.5 English Contains some Regular Substructure

Natural languages are not formal formal languages, but we can apply similar arguments. Here we show a substructure of english has some similar structure to a regular language. For example, the following DFA[2] can be used to model the formation of a substructure of english:



We are not concerned with the study of finite languages, but of infinite ones decidable by finite structures. Here, this decides an infinite language because it has a point of recursion. It may be inappropriate to describe someone as "old old old old...", but it is not ungrammatical, it is a hyperbole.

1.6 English is Not Regular

First, recall our three non-regular languages

- $\{a^n b^n \mid n \in \mathbb{N}\}$
- $\{ww^R \mid w \in \Sigma^*\}$

[1] It is ironic that Chomsky chose this sentence because it would be statistically infrequent, but by choosing it as an example, he has made it very famous. See its own wikipedia page

[2] Chomsky calls this a Finite-State Markov Process (FSMP)

- $\{ww \mid w \in \Sigma^*\}$

We show by an analogy, that there does not exist a DFA for a substructure of english.

Proof. Let $S_1, S_2, ...$ be declarative sentences. Let S be the sentence "if S_i, then S_j". There is no reason we may not substitute S into S_i Observe that upon repetition of this n times, we get

$$\text{“(IF)(IF)(IF)(IF) } \ldots S_i \text{ (THEN)(THEN)(THEN)(THEN) } \ldots S_j\text{”}$$

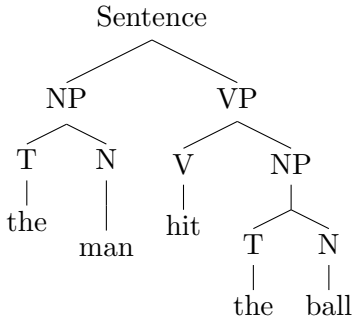
can we written as $(\text{IF})^n S_i (\text{THEN})^n S_j$, this is quite similar to our first known canonical non-regular language. We proved that language was non-regular by pumping, and similarly here, there would not exist a DFA for this substructure of english. Another good example is the Dyck language, the set of balanced parenthesis. One would recognize if a sentence had unbalanced paranthesis and distinguish it as ungrammatical. An an example, consider "((hello there.)(((". □

1.7 Phrase Structure

It had been known for centuries of the recursive nature of language. How sentences can be built from fragments, fragments from words, words from letters, and so on. Sentences have a hierarchical structure, and this structure is governed by the rules from grammar. Chomsky formalized these observations to justify what his next model of study was, and why it was ideal. He defines something called a phrase structure, which is a generalization of what we now call a CFG. For now, lets suppose phrase structures are just CFGs. We can remark that this device is incredibly useful as generative model for language. Consider the following model:

$$\begin{aligned}
\text{Sentence} &\to \text{Noun Phrase} + \text{Verp phrase} \\
\text{Noun Phrase} &\to \text{Article} + \text{Noun} \\
\text{Verb Phrase} &\to \text{Verb} + \text{Noun phrase} \\
\text{Articles} &\to \{\text{a, the, etc.}\} \\
\text{Noun} &\to \{\text{man, men, ball, etc.}\} \\
\text{Verb} &\to \{\text{hit, took, etc.}\}
\end{aligned}$$

Here, we have a phrase structure for a declarative substructure of english. A production of a sentence from our phrase structure can be expressed as a parse tree.



Its notable here that a parse tree gives less information than a list of productions, as just from the tree, you don't know what order the rules were applied in.

1.8 Limitations of our Phrase Structure

Although we note that this generates grammatical sentences, it can also generate ungrammatical ones. This example is with respect to singular and plural words.

1. "The man hit the ball."
2. "A men hit the ball."

The second sentence is clearly not grammatically correct.

Chomsky: "We must be able to limit the application of a rule to a certain context"

A context-free grammar is quite literally, free of context. If you have a production of $N \to \{$ nouns$\}$, then you can substitute in any noun. Like mad libs, it may not be grammatical. We want to consider applications of rules which are sensitive to context. A production can only be applied if conditions are met on the part of the working string before and after the substring we would insert. Comparison of language models:

Model	Example rule
Regular grammars	$A \to bE$
Context-free grammars	$A \to bCdEf \dots$
Context-sentitive grammars	$xAz \to xyz$

Here, $x, z \in (V \cup \Sigma)^*$. You can only make the substitution $A \to y$ when in the current working string, A is preceded by x and followed by z. These types of rules are called context-sensitive, because they are quite literally, sensitive to context. They are strictly stronger than context-free grammars, and we will not spend any more time on them. For our small piece of english we are studying, we can modify the phrase structure with context sensitive rules to solve our issue with singular and plural words as follows.

$$\begin{aligned} NP_1 &\to T_p + N_p \\ NP_2 &\to T_s + N_s \\ T_s &\to \text{a} \\ T_p &\to \text{the} \\ NP_s &\to \text{man} \\ NP_p &\to \text{men} \end{aligned}$$

Here, N_p, a non-terminal for plural nouns, cannot be preceded by T_s, singular articles. This makes the ungrammatical production of "a men" impossible.

I highly suggest you read Syntactic Structures in full. This is a high level overview of some of the simpler and early theorems made, and how they were argued.

1.9 Further Reading

- Syntactic Structures by Noam Chomsky
- Poverty of the Stimulus
- How To Know What Words Mean - Troublehacking with Drew Cleary

2 Chomsky Normal Form

Given a word w and a grammar G, is $w \in L(G)$? This is surprisingly non-trivial. We say a CFG is in Chomsky Normal Form (CNF) if it has productions only of the form:

$$A \to BC$$
$$A \to a$$

where the capital letters are any non-terminals, and the lower-case letters are any terminals. Additionally B, C cannot be the start state. and $S \to \varepsilon \iff \varepsilon \in L(G)$. Note that obviously $\mathscr{L}(CNF) \subseteq \mathscr{L}(CFG)$. We have a process to convert any CFG into CNF form, meaning that $\mathscr{L}(CFG) = \mathscr{L}(CNF)$.

1. Add a new start State $S_0 \to S$. Now every rule will not have the start state anywhere on the RHS.
2. Delete and patch all $A \to \varepsilon$ rules. For example if you have rules $R \to uAv, A \to \varepsilon$, you now have rules $R \to uAv \mid uv$.
3. Remove all unit rules $A \to B$ (i.e. $(A \to B, B \to C) = A \to C$
4. Convert rules of length greater than two into a chain of rules as follows. $(A \to u_1 \dots u_k) \to (A_1 \to u_1 A_1, A_2 \to u_2 A_2, \dots, A_{k-1} \to u_{k-1} u_k)$
5. $\forall a \in \Sigma$, replace a with A using $A \to a$.

Steps three and four may need to be repeated many times because applying one patch may introduce a need for another.

2.1 Advantages of CNF

Lets prove that if a word of length n is produced by a grammar in CNF, it takes exactly $2n - 1$ productions. Lets work backwards.

$$w_1 ... w_n \overset{*}{\Longleftarrow}_1 W_1 ... W_n \overset{*}{\Longleftarrow}_2 S$$

- The last productions (1) goes from n terminals to n non-terminals. At each production, exactly one non-terminal is replaced by exactly one terminal, so this takes n productions.

- For (2), to go from n non-terminals to one terminal, our start terminal, requires $n-1$ productions. Every rule of a grammar in CNF takes one non-terminal, and adds two. So for each production, if non-terminals are added, a production adds exacly one.

Combined, we see that a grammar in CNF form will take exactly $2n-1$ productions to produce a word of length n. This solves our acceptance problem. Convert your grammar to CNF, compute all words produced after $2n-1$ productions, and your candidate word is in this list $\iff w \in L(G)$. Consider the following conversion of a general CFG to one in CNF.

$$
\begin{aligned}
&S \to aSb \mid \varepsilon \\
&S_0 \to S,\ S \to aSb \mid \varepsilon \\
&S_0 \to S \mid \varepsilon,\ S \to aSb \mid ab \\
&S_0 \to aSb \mid ab \mid \varepsilon,\ S \to aSb \mid ab \\
&S_0 \to aX \mid ab \mid \varepsilon,\ S \to aX \mid ab,\ X \to Sb \\
&S_0 \to AX \mid AB \mid \varepsilon,\ S \to AX \mid AB,\ X \to SB,\ A \to a,\ B \to b
\end{aligned}
$$

Lets verify that $\{a^n b^n \mid n \in \mathbb{N}\}$ takes $2n-1$ productions. That $aabb$ takes seven productions.

$$S \implies AX \implies ASB \implies AABB \implies aABB \implies aaBB \implies aabB \implies aabb$$

CS 4510 Automata and Complexity February 8th 2023

Lecture 7: Pushdown Automata

Lecturer: Abrahim Ladha *Scribe(s): Rishabh Singhal*

1 Introduction

We mentioned previously how if we had a stack data structure, we could parse arithmetical expressions, like a classic data structures assignment. Lets do that. We are literally going to give an NFA a stack. We say a Pushdown Automata (PDA) is a tuple $(Q, \Sigma, \Gamma, q_0, \delta, F)$

- $Q = \{q_0, \ldots, q_k\}$ our set of statestates
- Σ = input alphabet
- Γ = Stack alphabet. By convention, we will usually set $\Gamma = \Sigma \cup \{\$\}$, where \$ is a special symbol we denote as the stack canary.
- $q_0 \in Q$ is the start state
- $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times (\Gamma \cup \{\varepsilon\}) = \mathcal{P}(Q \times (\Gamma \cup \{\varepsilon\}))$ our transition function
- $F \subseteq Q$ is the set of final or accepting states.

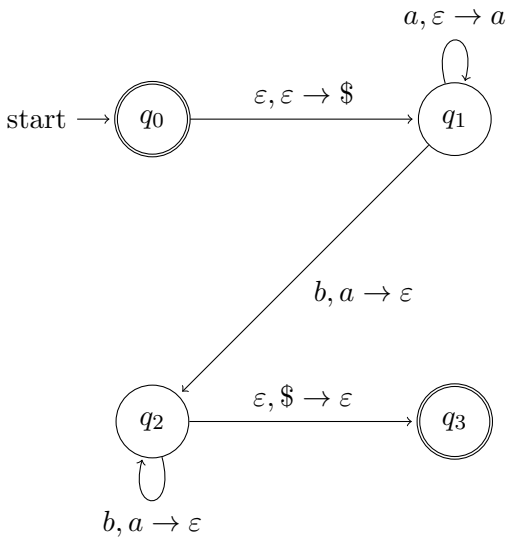
Notice that we have basically done as we said we would do. We just augmented an NFA to give it a stack. The transition function works as follows. You go from some state, optionally read a symbol off the input, and optionally pop the top of the stack. Then you transition to a new state and optionally push something onto the stack. When you see a transition of the form $a, b \to c$, you read a from input, pop b and push c onto the stack. Note that our defined PDA is explicitly nondeterministic. Deterministic PDAs exist, but unlike DFAs and NFAs, they are explicitly weaker. We will not cover them. Lets give some programming analogies to the types of transitions possible.

- $\varepsilon, \varepsilon \to \varepsilon$ We read nothing from the input, pop nothing, and push nothing. This means we really only change states, so this acts like an ε-transition in an NFA.
- $a, \varepsilon \to a; b, \varepsilon \to b$ We read a, b off the input, popped nothing, and pushed a, b respectively. We would only push exactly what we read, so this would push the input to the stack. If it was in a self-loop, it would push the entire input to the stack.
- $\varepsilon, a \to \varepsilon; \varepsilon, b \to \varepsilon$ Read nothing from the input, pop something, and push nothing. This would pop the top of the stack, whatever it may be. In a self-loop, it would dump the stack.
- $a, b \to b$ Read a off the input, pop b off the stack, then push b right back. This allows us to essentially transition off of "peeking" the top of the stack. We cannot read the top of the stack without popping it, but we can simulate this as popping it and then pushing it right back.

2 Examples

2.1 $\{a^n b^n \mid n \in \mathbb{N}\}$

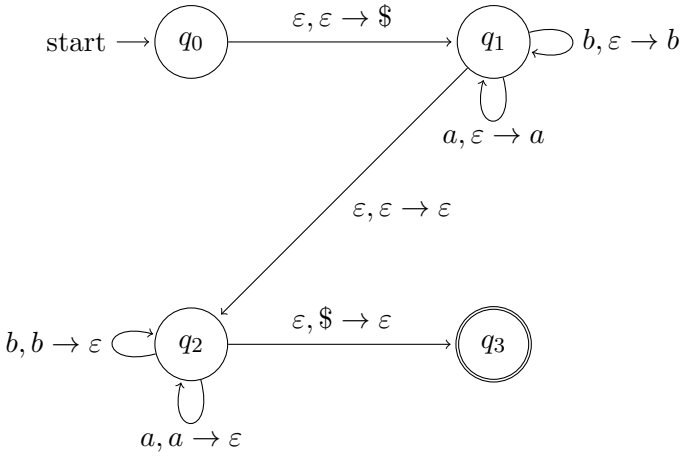
While reading a's off the input, we are going to push them. If we read a b, we start reading b's off the input, matching them to a's popping off the stack, and we accept only if we read as many b's as we previously pushed a's. We don't have an inbuilt mechanism to determine if the stack is empty. Thats why we begin almost every PDA by pushing \$. Then if we ever see our canary again, we know the stack is empty.



Note that we must also accept ε so the start state is marked as accepting.

2.2 $\{ww^R \mid w \in \Sigma^*\}$

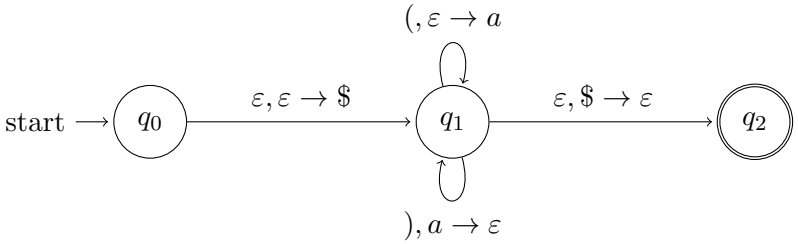
As you might suspect, our PDA will be similar, however before we could start matching the second half of our string by seeing a b. Here, we don't have that privelige. We can still solve it by nondeterministically guessing the midpoint of the input!



Note that all computation paths which incorrectly guess the midpoint of the string would implicitly reject. We also need not mark the start state as accepting as there exists a non-trivial computation path to accept ε. We would push the canary, epsilon transition, pop the canary, and accept.

2.3 Dyck Language

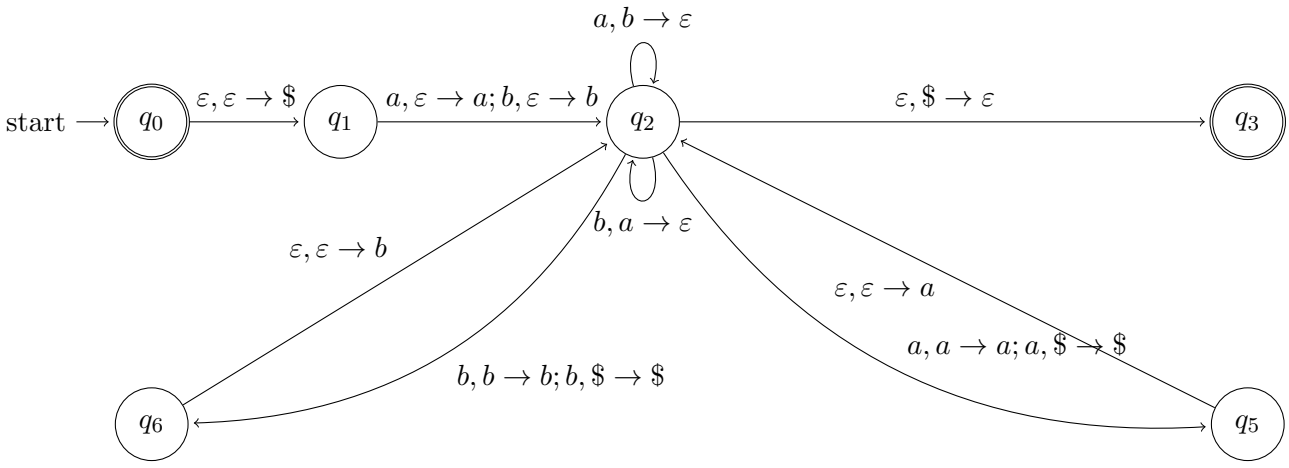
Recall that the Dyck language is the set of balanced paranthesis. You decide this language trivially with a counter. You increment your counter for every open you see, and decrement it for every close. If your counter ever goes negative, you reject. If your counter ends with anything other than zero, you reject. We will use the stack more simply here, as just a unary counter.



Here $\Sigma = \{(,)\}$ and $\Gamma = \{a, \$\}$. Just for clarity, I chose Γ to be something else, as it doesn't particularly matter what we count with.

2.4 $\{w \in \Sigma^* \mid \#a(w) = \#b(w)\}$

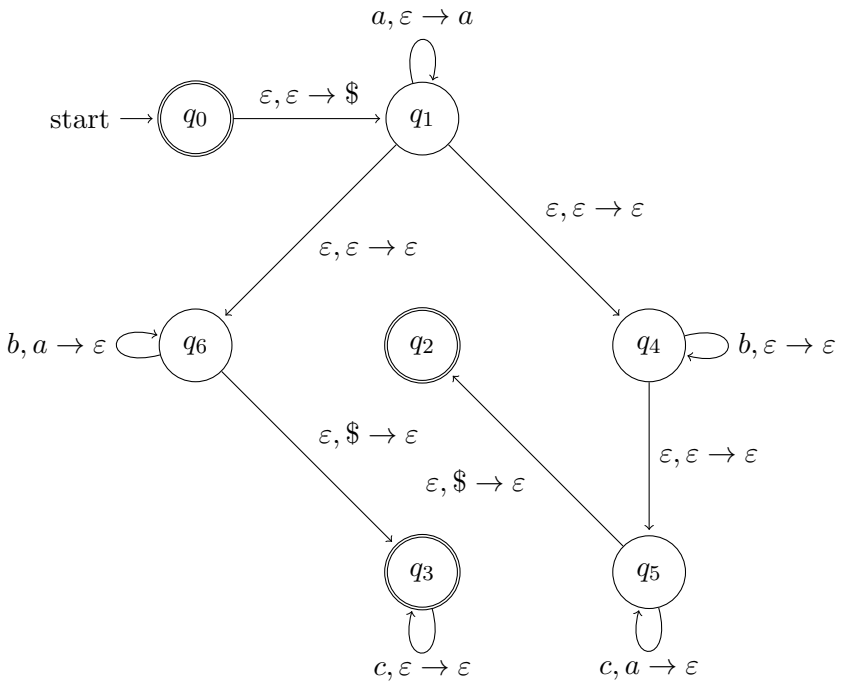
This will be similar to two PDAs we have seen, the Dyck language, and $a^n b^n$. However, instead of a positive counter, we would need a counter which could occasionally be negative. Different symbols on the stack will be used to represent the surplus in one direction or the other.



You should note we could do this PDA far simpler if we were allowed to push more than one symbol onto the stack per transition. We will show how to do this next lecture.

2.5 $\{a^i b^j c^k \mid i, j, k \in \mathbb{N}, i = j \textbf{ or } j = k\}$

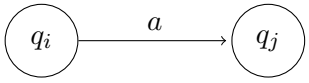
Since the definition of the language contains an or, why don't we make two PDAs for each case, and then use nondeterminism to join them together.

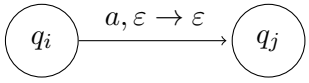


Although we won't show it, this PDA requires nondeterminism. There is no DPDA to decide this language.

3 $\mathscr{L}(NFA) \subsetneq \mathscr{L}(PDA)$

We can convert any NFA into a PDA by simply preserving the topology of the states, and ignoring the stack. Some for some $q_j \in \delta(q_i, a)$ in our NFA, we would have $(q_j, \varepsilon) \in \delta(q_i, a, \varepsilon)$ in our PDA.





Our containment was strict because we gave PDAs for many languages we know not to be regular. In fact, all the examples we gave were languages we know to be context free. Next time we will show that $\mathscr{L}(PDA) = \mathscr{L}(CFG)$.

CS 4510 Automata and Complexity February 13th 2023

Lecture 8: $\mathscr{L}(CFG) = \mathscr{L}(PDA)$

Lecturer: Abrahim Ladha *Scribe(s): Samina Shiraj Mulani*

1 $\mathscr{L}(CFG) \subseteq \mathscr{L}(PDA)$

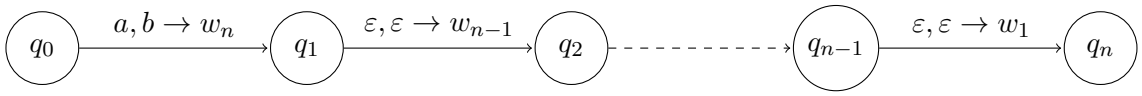
Recall the grammar with the following production rules $S \to aSb \mid \varepsilon$. which produces the language $\{a^n b^n \mid n \in \mathbb{N}\}$. As we produce strings, we get intermediate strings which are called "working strings" (example w.r.t aforementioned language - $aSb, aaaSbbb$). Once the working string does not have any non-terminals, thats the string produced by our choice of productions. We are trying to construct a PDA given a CFG. Consider some working string in the grammar. We will simulate part of it on the stack and part on the input. If we have a working string like $aaaSbbb$, anything that comes before the first non-terminal must be the prefix of the produced word. We don't need to keep this on the stack, but can just match it to the input.

Input → $aaa\dot{:}abbbb$ Stack → $Sbbb\$$

Working String → $aaa\dot{:}Sbbb$

- If top of the stack is a terminal, we pop it and advance input if it matches.
- If the top of the stack is a non-terminal, we pop it and non-deterministically choose its production and push it.

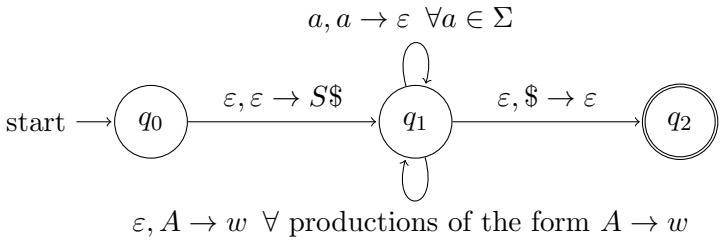
Observe that pushing a nonterminal's production on to the stack may involve pushing more than 1 symbol. We generalize and say that we can push $w_1 w_2 ... w_n$ on to the stack by simulating the insertion in the PDA by adding more states as shown. $a, b \to w_1 ... w_n$ would be



So, if we have $a, b \to abc$, top of the stack is now a. You should remember this.

1.1 CFG to PDA conversion

Our shortcut here allows us to represent the PDA for any CFG only using three states. If we didn't have our shortcut, each transition would need states for the length of the RHS of the production.

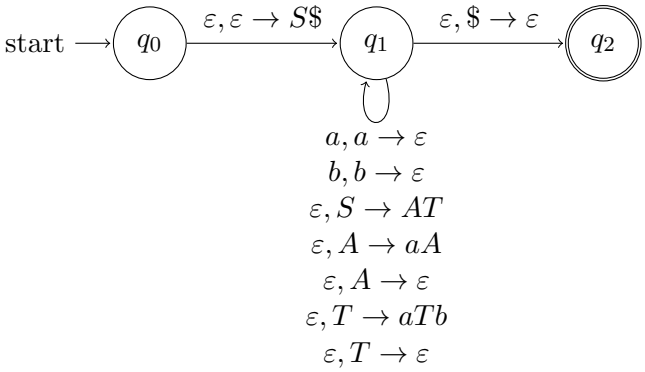


If I asked you to write a program to simulate a CFG, this might be non-trivial. Its interesting to note that actually the PDA is easier. Both PDAs and CFGs are nondeterministic, and we can have the nondeterminism of one simulate the nondeterminism of the other. The nondeterministic choice of productions becomes a nondeterministic choice of transitions.

1.2 Example

$\{a^m b^n \mid m \geq n\}$
One correct CFG may be
$S \to AT$
$A \to aA \mid \varepsilon$
$T \to aTb \mid \varepsilon$

Here, A produces like a^*, and T produces like $a^n b^n$ matching. So $S \to AT$ will give us $a^m b^n$ with $m \geq n$.



Lets do a computation of aab to show the PDA accepts this grammar. Here, the underline of the input represents the symbol we are looking at, and the top of the stack is the leftmost.

Input →	aab	$\underline{a}ab$	$\underline{a}ab$	$a\underline{a}b$	$a\underline{a}b$	$a\underline{a}b$	$aa\underline{b}$	$aa\underline{b}$	aab:	aab
Stack →	$S\$$	$AT\$$	$aAT\$$	$AT\$$	$T\$$	$aTb\$$	$Tb\$$	$b\$$	$\$$	empty stack

2 $\mathscr{L}(PDA) \subseteq \mathscr{L}(CFG)$

This proof is far harder than the previous one. Its easy to program a PDA to simulate a CFG. We will see its harder to "program" a CFG given a PDA. Instead of just doing an example, lets do a rigorous proof of correctness.

2.1 Make it nice

First, we have to make P nice, by modifying it in the following three ways.

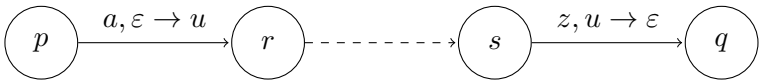
1. P only has one accept state.
   This trick also works for NFAs. Make a new final state, make all old accepting states as non-accepting, and then epsilon transition from all old accepting states to our single new one. This lets us assume there is only one accept state.

2. P accepts only with an empty stack.
   Simply make sure we begin and end with pushing and popping \$. The old accept state should dump the stack before popping \$ to accept. This lets assume the computation ends how it begins, with an empty stack.

3. Each transition pushes or pops but not both.
   Suppose we had a transition like $a, b \to c$. We can turn this into two transitions like $a, b \to \varepsilon$ and $\varepsilon, \varepsilon \to c$. This lets assume each move of the PDA either pushes or pops but not both.

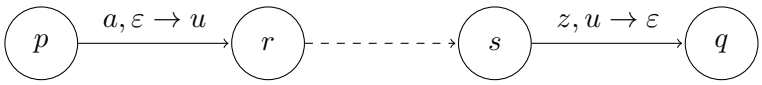
2.2 High level idea

For each $p, q \in Q$, make A_{pq} to represent strings which could take our PDA starting at state p and an empty stack, and ending on state q in an empty stack. The start variable should be A_{0f} where q_0 is the start state and q_f is the only final state. Since we are going from an empty stack to an empty stack, the first move must be a push and last move must be a pop. If the stack was never fully emptied from $p...q$ computation, then the first symbol pushed must be the last symbol popped, call it u. Let r, s be the next states making the computation path look like $pr...sq$. Here, r is the next state after p, and s is the state preceeding q. Say a is whats first read of the input, and z is whats last read, resulting in $A_{pq} \to aA_{rs}z$.



If the stack was ever emptied from $p...q$ computation, we handle this as $\forall p, q, r \quad A_{pq} \to A_{pr}A_{rq}$, where r would have been the state where the stack was empty. If it was empty at no other point, we can then inductively delegate A_{pr}, A_{rq} back to the first case. For PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, F = \{q_f\})$, we make the following CFG:

- $\forall p, q \in Q$ we add $A_{pq} \in V$ (V = set of nonterminals).
- $S = A_{0f}$
- $\forall p \in Q, A_{pp} \to \varepsilon$
- $\forall p, q, r \in Q, A_{pq} \to A_{pr}A_{rq}$
- $\forall p, q, r, s \in Q, u \in \Gamma, a, z \in \Sigma \cup \{\varepsilon\}$, add rule $A_{pq} \to aA_{rs}z$ if there exist transitions in δ as defined previously like



Given a PDA P, we have constructured a grammar G. To prove the correctness of this construction, we need to prove:

$A_{pq} \stackrel{*}{\Rightarrow} x \qquad \iff \qquad x$ brings P from p (empty stack) to q (empty stack).

2.3 ($\Longrightarrow$) direction

Given that $A_{pq} \stackrel{*}{\Rightarrow} x$, we want to show that x brings P from p (empty stack) to q (empty stack). We proceed by proof by induction on the length of the derivation.

Base case: Derivations of length one

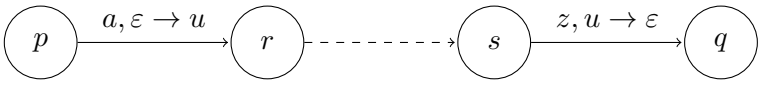
Out of all the rules, only one has no nonterminal on the right. There is only time for 1 production and that production must produce a string. $A_{pp} \to \varepsilon$. It is true that ε trivially brings P from p (empty stack) to p (empty stack).

Induction Hypothesis: Assume if $A_{pq} \stackrel{*}{\Rightarrow} x$ in $\leq k$ derivations, then x brings P from p (empty stack) to q (empty stack).

We have to show: True for derivations of length $k+1$

Assume $A_{pq} \stackrel{*}{\Rightarrow} x$ in $k+1$ steps. We have two cases.

1. First step in our derivation is of the form $A_{pq} \to aA_{rs}z$. By induction hypothesis, for $x = ayz$, $A_{rs} \stackrel{*}{\Rightarrow} y$ in $\leq k$ steps, so y must bring P from r (empty stack) to s (empty stack). By construction we only have the rule $A_{pq} \to aA_{rs}z$ if:



   So ayz brings P from p (empty stack) to r to s to q (empty stack). If y brings P from r (empty stack) to s (empty stack), then certainly it can bring P from r (just u in stack) to s (just u in stack). This means that $A_{pq} \stackrel{*}{\Rightarrow} x$ and $ayz = x$ brings P from p (empty stack) to q (empty stack).

2. The first step in our derivation is of the form $A_{pq} \to A_{pr}A_{rq}$. By induction hypothesis, $x = vw$, $A_{pr} \stackrel{*}{\Rightarrow} v$, $A_{rq} \stackrel{*}{\Rightarrow} w$ in at most k steps. So v brings P from p (empty stack) to r (empty stack) ans w brings P from r (empty stack) to q (empty stack). So, clearly $vw = x$ brings P from p (empty stack) to r (empty stack) to q (empty stack).

2.4 ($\Longleftarrow$) direction

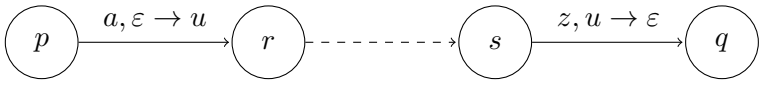
We want to show that if x brings P from p (empty stack) to q (empty stack), then $A_{pq} \stackrel{*}{\Rightarrow} x$. We proceed by proof by induction on length of computation.

Base case: Computation of zero steps.

A computation of zero steps means we don't have time to switch states, so consider any such x with $A_{pp} \stackrel{*}{\Rightarrow} x$. Zero steps also means that there is no time to read any input. So $x = \varepsilon$. But, we have the rule $A_{pp} \to \varepsilon$ as desired.

Induction Hypothesis: Assume its true for computation of length $\leq k$, that if x brings P from p (empty stack) to q (empty stack) in $\leq k$ computation steps then $A_{pq} \stackrel{*}{\Rightarrow} x$
We have to show: True for computations of length $k+1$.
Let x bring P from p (empty stack) to q (empty stack) in $k+1$ steps. We have two cases.

1. Suppose the stack is only empty at the beginning and end. So the first symbol pushed must be the last symbol popped. By our construction for $x = ayz$



   By induction hypothesis since y brings P from r (empty stack) to s (empty stack), then $A_{rs} \stackrel{*}{\Rightarrow} y$ and we have the rule $A_{pq} \rightarrow aA_{rs}z$. So, $A_{pq} \stackrel{*}{\Rightarrow} x$.

2. During the computation of length $k+1$, the stack is empty at some middle point, lets say, at state r. The computations from p to r and r to q take at most k steps. By induction hypothesis, for $x = vw$, $A_{pr} \stackrel{*}{\Rightarrow} v$ and $A_{rq} \stackrel{*}{\Rightarrow} w$. Since we have the rule $A_{pq} \rightarrow A_{pr}A_{rq}$, $A_{pq} \stackrel{*}{\Rightarrow} vw = x$.

3 Remarks

Recall that we say a language is context-free if there exists a CFG to produce it. We may now also say that a language is context-free if there exists a PDA to decide it. The PDAs decide exactly the same languages that CFGs produce. We did not give a formal proof that regular grammars produce only regular languages, but regular grammars are a strict superset of those which are context-free. This proof is an alternate way to see that the regular languages are also produced by context-free grammars. Every regular language can also be decided by a PDA, which is equivalent to some CFG.

This proof had many parts. We have to indidividually prove $\mathscr{L}(PDA) \subseteq \mathscr{L}(CFG)$ and $\mathscr{L}(CFG) \subseteq \mathscr{L}(PDA)$. We really only did half. For $\mathscr{L}(PDA) \subseteq \mathscr{L}(CFG)$, we gave a construction of a grammar G from a PDA P, and then we had to show $L(G) \subseteq L(P)$ and $L(P) \subseteq L(G)$. Then each of those had their own cases.

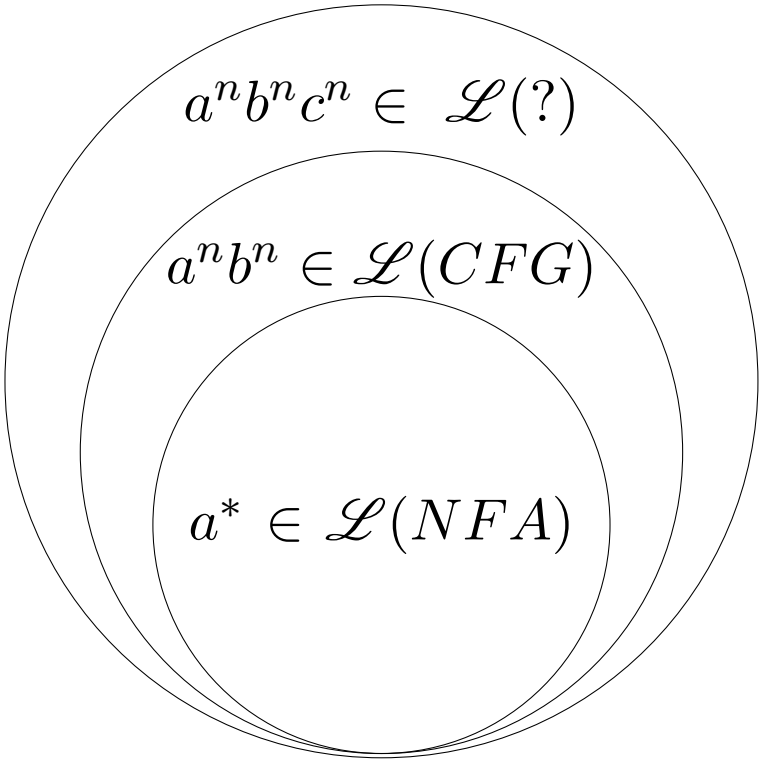
CS 4510 Automata and Complexity February 15th 2023

Lecture 9: The Pumping Lemma for Context Free Languages

Lecturer: Abrahim Ladha *Scribe(s): Rishabh Singhal*

1 Introduction

Lets give a high level picture of the space of languages as we know it so far. In the beginning, there was only the regular languages which had things like a^*. Next we found a class of languages called the CFLs which included things which were specifically not regular, like $\{a^n b^n \mid n \in \mathbb{N}\}$. Today, we will show the existence of languages outside of $\mathscr{L}(CFG)$, with our first example being $\{a^n b^n c^n \mid n \in \mathbb{N}\}$.



We will prove that $L = \{a^n b^n c^n | n \in N\} \notin \mathscr{L}(CFL)$, but for now, lets just assume so. Consider the following two languages. $L_1 = \{a^m b^n c^n \mid m, n \in \mathbb{N}\}$ and $L_2 = \{a^n b^n c^m \mid m, n \in \mathbb{N}\}$. Certainly they are context-free, since we can produce grammars for each. Here is the grammar for L_1:

- $S \to AT$
- $A \to aA \mid \varepsilon$
- $T \to bTc \mid \varepsilon$

And similarly, for L_2:

- $S \to RC$
- $C \to cC \mid \varepsilon$
- $R \to aRb \mid \varepsilon$

However, notice that $L_1 \cap L_2 = \{a^n b^n c^n \mid n \in \mathbb{N}\}$. The number of a's equals the number of b's, but then the number of b's must equal the number of c's. So transitively, we get our triple threat language.

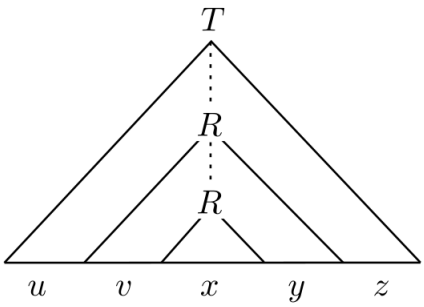
Assume to the contrary CFLs are closed under intersection. Since L_1, L_2 are CFLs, so should be $L_1 \cap L_2$. But $L_1 \cap L_2 = \{a^n b^n c^n \mid n \in \mathbb{N}\}$ a language we are going to pump and prove to not be context-free, a contradiction. Assume to the contrary CFLs are closed under complement, then $\overline{L_1}$, $\overline{L_2}$ would be context-free, and so would $\overline{\overline{L_1} \cup \overline{L_2}} = L_1 \cap L_2$. However, we will prove it is not, a contradiction. CFLs are closed under union, Kleene star, and concatenation, but not under intersection or complementation.

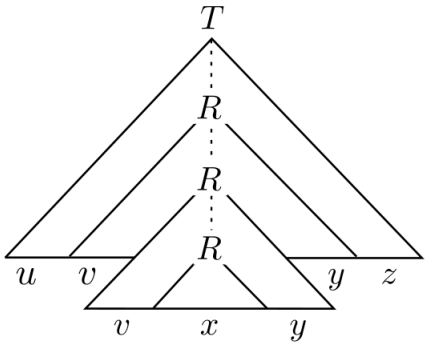
2 Parse Trees

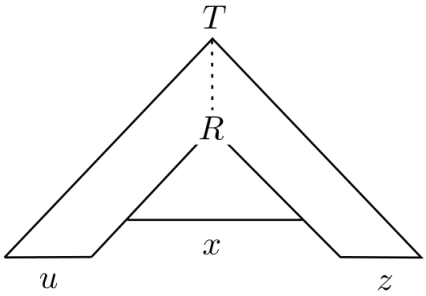
Recall that parse trees exist. This is what we will do our combinatorial "pumping" on. Any grammar is finite but can produce arbitrarily long words. Consider the production of a super long word with a huge parse tree. If the string is long enough such that a non-terminal is repeated twice on some path from the start non-terminal, I claim we can perform the following surgery on the parse tree. We may cut and copy the different subtrees with our repeated non-terminal. If some $uvxyz \in L$, then $uv^i xy^i z \in L$ as well.

Now lets derive the exact conditions for this to be true. Let b be the maximum length of the RHS of the largest rule of G. For any parse tree of G, a single step from the start has at most b children. Two steps has at most b^2, and so on. It may help to recall the derivation of the Master Theorem from Algorithms. If the tree height is h, the maximum length of the string it can derive is $|w| \leq b^h$. Conversely, if a generated string has length $b^h + 1$, its parse tree has height $h + 1$. For convention set $p = b^{|V|+1}$, so our parse tree has height greater than or equal to $|V| + 1$. Its true that we could satisfy this with just $b^{|V|} + 1$, but since $b^{|V|+1} > b^{|V|} + 1$, our tree height is still $\geq |V| + 1$ and we will need this later. Since G only has $|V|$ non-terminals, some non-terminal is repeated twice (or more) by the pigeonhole principle along a path from the start non-terminal.

- We want to produce a string such that its parse tree has height sufficient to be pumped, so we may choose any $|s| \geq p$.
- For $s = uvxyz$ as shown in the figure, we want v, y to both not be empty, so we may pump something nontrivial. We condition this as requiring $|vy| > 0$.
- There may be many more repeated non-terminals along the height of our tree, but we want to focus on to the last repeated one. In the picture we have $R \stackrel{*}{\Rightarrow} vxy$. We want this tree to be at most $|V| + 1$ high, so we require that $|vxy| \leq b^{|V|+1} = p$.







3 Recipe of Pumping Lemma

Recall that like the pumping lemma for regular languages, it is easy to miss one case here or there, or misapply it. Following this recipe exactly is the best way to ensure your proof is correct.

1. Assume the contrary L is context-free with pumping length p.
2. Choose some $s \in L$, $|s| \geq p$
3. List cases $\forall u, v, x, y, z$ with $s = uvxyz$ subject to $|vxy| \leq p$ and $|vy| > 0$.
4. For each case choose an i such that $uv^ixy^iz \notin L$
5. Conclude that the language must not be context-free.

Also unlike the pumping lemma for regular languages, we are not going to be able to list out every case mathematically, using superscripts as lengths. Instead, you have to think logically. Construct several "meta-cases" in which each sub-case is easy to argue. The meta-cases should be defined so that its obvious any case must fall into one meta-case or the other.

4 Examples

Lets do an example.

4.1 $\{a^n b^n c^n \mid n \in \mathbb{N}\}$

- Assume to contrary L is context-free with pumping length p.
- Choose $s = a^p b^p c^p$. Certainly $s \in L$ and $|s| \geq p$.
- We have two general cases:
  - v **and** y **each contain only one type of symbol.** There is three of $\{a, b, c\}$, only two of the $\{v, y\}$. So $uv^2xy^2z \notin L$ cannot contain an equal number of a's, b's, and c's. Some letter will not increase.
  - **If either** v **or** y **contains more than one symbol**. then uv^2xy^2z contains symbols out of order, and thus, not be in our language.
- It follows that L is not a CFL.

Lets give some remarks about the proof. First, for choosing s, before for regular languages, a bad choice of s just meant we had many cases but it could still be correct. For pumping context-free languages, a bad choice of s likely will not allow you to even complete the proof. Many obvious or first choices end up being pumpable, when we want to show they are not. Try to choose a string which is barely in the language. One where even the smallest perturbation brings it out. Secondly, Notice that we didn't really have to apply the fact that $|vxy| \leq p$. This is sometimes too fine-grained. This would eliminate the case where v may contain a's at the same time y may contain c's. Our meta cases are so general, these are absorbed. Finally, notice how we defined our meta cases as logical complements of each other. Either both v and y contain one symbol, or maybe one of them is a mix of symbols. Any possible cases must fall into one of those two huge general categories. If we tried to do it any other way, we may get dozens of cases, many of which are identical.

4.2 $\{a^i b^j c^k \mid 0 \leq i \leq j \leq k \in \mathbb{N}\}$

- Assume to the contrary that L is regular with pumping length p. Choose $s = a^p b^p c^p$. Clearly $s \in L$ and $|s| \geq p$. We have a few cases
- **both** v **and** y **are of only one type of letter.** We need to pump up or down depending upon what the letters actually are, so we divide into further sub cases.
  - If a's do not appear we pump down. Choose $i = 0$, so uv^0xy^0z has more a's than b's.
  - If b's do not appear:
    - But a's appear choose $i = 2$, so uv^2xy^2z has more a's than b's.
    - But c's appear choose $i = 0$, so $u\ v^0x\ y^0z$ has more b's than c's.
  - If c's do not appear choose $i = 2$, so uv^2xy^2z has more a's or b's than c's.
- **if** v **or** y **contain more than one type of symbol** we choose $i = 2$, as uv^2xy^2z will be out of order.
- We conclude that L is not CFL

Convince yourself that the cases listed are total. By out of order, we mean for example that $v = a^k b^j$. Then $uvv... = a^{p-k}a^k b^j a^k b^j ... = a^p b^j a^k b^j ...$ so somehow this string contains a's after the b's, and wouldn't be in the language.

4.3 $\{ww \mid w \in \Sigma^*\}$

There are many bad choices of s for this language. Choosing a few may lead you to a good one. I have the premonition to know we are going with a good choice of s here.

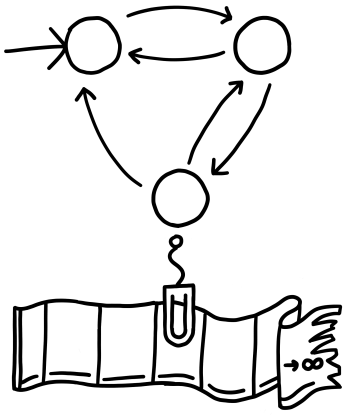
- Assume to contrary L is context-free with pumping length p. choose s = $0^p 1^p 0^p 1^p$ and confirm $s \in L$ and $|s| \geq p$.
- **If vxy is all in the first or last half**, Choose $i = 2$. let $uv^2xy^2z = w_1w_2$ with $|w_1| = |w_2|$. If vxy was in the first half, w_1 begins with 0 and w_2 begins with 1. If vxy was in the second half, w_1 ends with 0 but w_2 ends with 1. For either of those $w_1 \neq w_2$.
- vxy **straddles the midpoint.** Pump down so $uv^0xy^0z = 0^p 1^k 0^j 1^p$ where k, j cannot both be p. So now $0^p 1^k 0^j 1^p \neq ww$ unless $k = j = p$ and this is impossible since $|vy| > 0$.
- We conclude that L is not CFL

CS 4510 Automata and Complexity February 20th 2023

Lecture 10: Turing Machines

Lecturer: Abrahim Ladha *Scribe(s): Michael Wechsler*

Turing machines are so interesting, every lecture for the remainder of the course will be about Turing machines. I want this or that interesting sub-topic to be its own lecture, so that leaves us with nothing for today except definitions and programming. Recall the limitations of a PDA. You pop something out the stack, its gone into the ether, forever. What if we could iterate over our memory structure in a non destructive way. Our motivation is then to just give a DFA a tape.



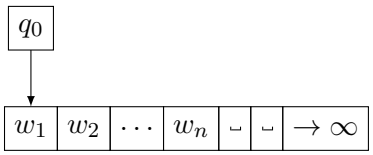
1 Definitions

A Turing machine is a tuple: $(Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r)$ where:

- Q : finite set of states
- Σ : finite input alphabet
- Γ : finite tape alphabet
  - $\textvisiblespace \in \Gamma$, this is the symbol for a blank space on the tape[1]
  - $\textvisiblespace \notin \Sigma$
  - $\Sigma \subsetneq \Gamma$
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{\text{L,R}\}$ is our (deterministic) transition function.
- $q_0 \in Q$: denoted start state
- $q_a, q_r \in Q$: denoted accept and reject states

[1]For latex, you may use either `textvisiblespace` or `sqcup`

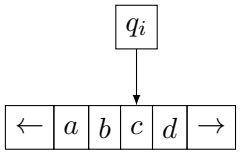
A Turing Machine is initialized with $w_1 w_2 ... w_n$ on the leftmost cells of the tape, and the tapehead on w_1. All other cells initialize to $\sqcup$.



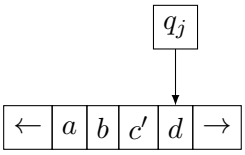
1.1 Configurations

A configuration of a Turing Machine is a string encoding of an instantaneous description, or snapshot, of a Turing Machine. We say a configuration C yields C' if C' follows C after one step of the transition function.. We write this as $C \vdash C'$. The entire description of the Turing machine at some moment is just the contents of the tape, the current state, and the position of the tape head. We encode these together as one string as follows.

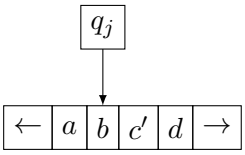
Consider the tape below



If $\delta(q_i, c) = (q_j, c', \mathrm{R})$, then $abq_icd \vdash abc'q_jd$



If $\delta(q_i, c) = (q_j, c', \mathrm{L})$, then $abq_icd \vdash aq_jbc'd$



$\star$ NOTE: $\vdash$ means yields

Notice that for sequential configurations, only a small local portion of the configuration is changed. The initial configuration of any Turing machine is always $q_0 w_1 \ldots w_n$. We may define the accepting and rejecting configurations appropriately, if they contain the accept or reject state. We say a Turing machine accepts $w_1 \ldots w_n$ if there exists a sequence of configurations

$C_0 = q_0 w_1 \ldots w_n$
$C_i \vdash C_{i+1}$
c_k is accepting

1.2 Create a Turing Machine, M, for $\{w\#w \mid w \in \Sigma^*\}$

Idea

mark checked as x	$abaa...\#abaa...$
skips xs to keep checking	$xxaa...\#xxaa...$

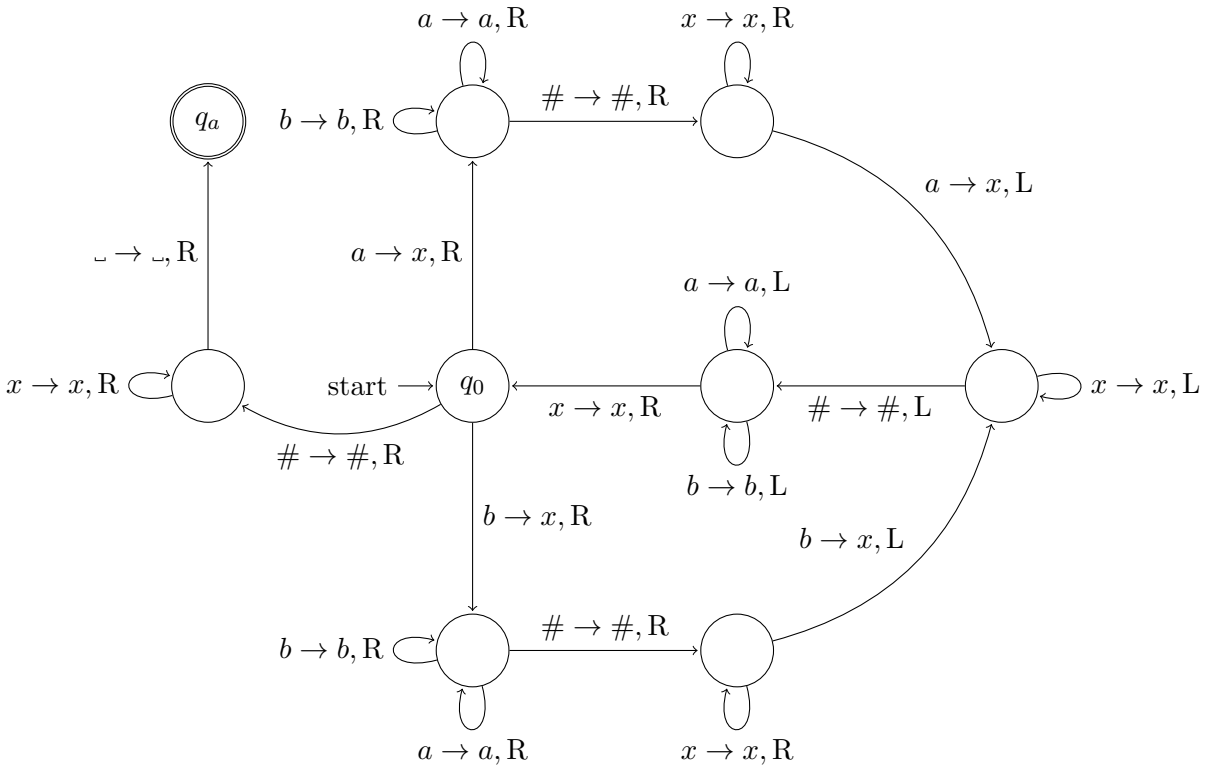
1.2.1 Pseudocode

M on input w:

1. mark symbol, keep track in states
2. loop right until #
3. loop past any x
4. if last marked = head:
    (a) mark
5. loop left until #
6. loop past any unmarked
7. reset to first unmarked (repeat from step 1)
8. if no symbols besides x before hash remain:
    (a) if no symbols besides x after hash remain:
        i. accept
9. reject

1.2.2 State Diagram

We can attempt to give a state diagram for this language. We will omit q_r, as it gets too messy. Undefined transitions in this diagram, you should understand to mean implicit rejection. Note how we have two branches, if we mark a or b. We use the states of the machine to keep track of what we have seen.

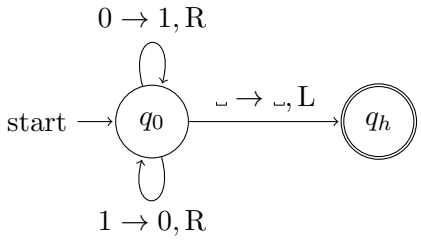


2 Computation

Unlike previous models, Turing machines do more than just decide languages. They can also compute! A function $f : \Sigma^* \to \Sigma^*$ is **Turing-Computable** (or just computable) if there exists a Turing Machine for all inputs w, which when initialized with w on the tape, halts with just $f(w)$ on its tape.

2.1 f : Bit Flips

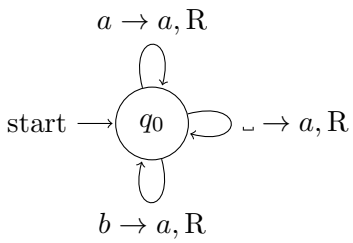
$\star$ NOTE: q_h is the halt state (accepting and rejecting do not matter here)



Example: Configurations for input 101

- q_0101
- $0q_001$
- $01q_01$
- $010q_0\sqcup$
- $01q_h0\sqcup$ HALTS in 4 steps

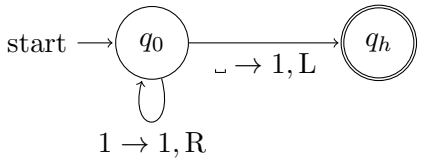
2.2 Turing Machines do NOT have to halt



The Turing Machine that writes a in every cell forever. Regardless of what it sees on its tape, its forever will march right, never stopping.

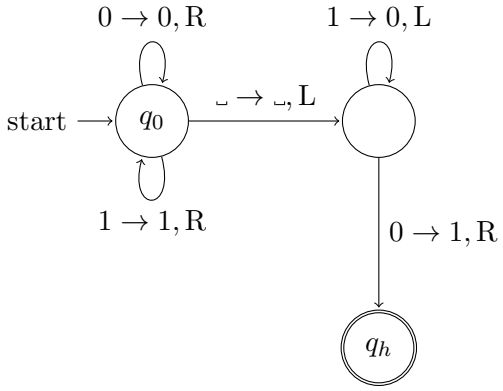
2.3 Successor Function: $S(x) = x + 1$

2.3.1 Unary



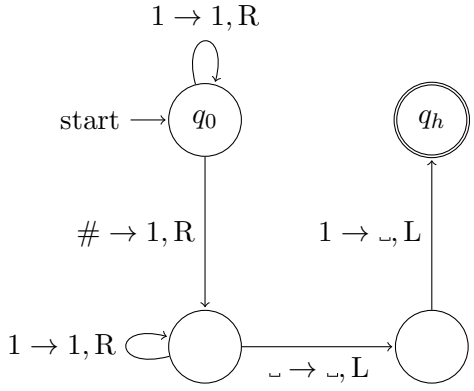
We begin with 1^x on the tape Halt with 1^{x+1}. The idea is we just get to loop to the end and toss a stick onto the pile.

2.3.2 Binary



For simplicity suppose the input is always given with a leading zero. We begin on the left on the input, so first we move all the way to the right. When adding 1 in binary, we loop from the right zeroing out all 1s until we find the first 0 and make it a 1.

2.4 Addition of 2 Numbers



$\mathrm{add}(x, y) = x + y$. Lets begin with $1^x \# 1^y$ on the tape. Halt with 1^{x+y} on the tape. The simplest idea is to replace the # with a 1 and remove the last 1 at the end. It would be challenging, but convince yourself you could give a Turing machine for addition in binary.

3 Decidability vs Recognizability

Recall, a function $f : \Sigma^* \to \Sigma^*$ is **Turing-Computable** (or computable) if there exists a Turing Machine for all inputs w, which when initialized with w on the tape, halts with $f(w)$ on its tape.

Additionally, a language, L, is **Turing-Decidable** (or just decidable or recursive) if there exists a Turing Machine, M, such that

- $w \in L \iff M$ accepts w
- $w \notin L \iff M$ rejects w

Notice that for every input, a decide always halts. A language, L, is **Turing-Recognizable** (or just recognizable or recursively-enumerable) if there exists a Turing Machine, M, such that

- $w \in L \iff M$ accepts w
- $w \notin L \iff M$ rejects w or gets stuck in a loop

We allow a recognizer to loop on some inputs. If the answer is supposed to be yes, it must always halt and accept. If the answer is suppose to be no, then it may halt and reject, or loop. It is clear to see that every decidable language is also recognizable, but is every recognizable language also decidable? We shall see.

CS 4510 Automata and Complexity February 23rd 2023

Lecture 10: Church-Turing Thesis

Lecturer: Abrahim Ladha *Scribe(s): Rishabh Singhal*

1 Introduction

The Church-Turing Thesis cannot be proved. Most agree that it is some kind of definition or worse, a "working hypothesis". Here, we will give the closest thing to a proof possible. Recall the narrative of our class. First we did regular languages, that was level one. We pumped out of those to get some context-free languages, that was level two. Now we are on level three, Turing machines. The Church-Turing Thesis essentially says that there is no level four. Our argument has two parts:

(I.) There is no computing device strictly more powerful than the human mind.

(II.) Turing Machines are equivalent in power to the human mind.

Together, these imply that a Turing Machine, although incredibly simple, is an excellent choice for us to reason about computation. A philosophical argument is unlike a proof. We want to make a convincing argument of some statement. The way we will do so is make atomic jumps, each convincing, then argue that the composition must be convincing.

2 Part I: Our Own Limits

Suppose our space of the languages looks like the following:

$$\mathscr{L}(NFA)\Big)\ \ \mathscr{L}(CFG)\Big)\ \ ...\ \ \mathscr{L}(Human)\Big)\ \ \mathscr{L}(?)$$

where $\mathscr{L}(Human)$ is the set of languages recognizable by a human. It's a pretty big class. If I give you a description of a language and a word, and you can tell if that word is in the language, then that language is in $\mathscr{L}(Human)$. Its not even yet clear if there are languages outside of this class. Could something exist with $\mathscr{L}(Human) \subsetneq \mathscr{L}(?)$ Our first observation is that if it existed, then we could not build it. Otherwise, we could understand it. Anything humans have managed to build, they do by understanding it. Even if we do not understand why some things happen, we understand what happens. So suppose then it is somehow an Alien device.

Assume to the contrary this alien device exists in a useful way and $\mathscr{L}(Human) \subsetneq \mathscr{L}(Alien)$[1]. Then there does not exist a simulation of $\mathscr{L}(Alien) \subseteq \mathscr{L}(Human)$. I claim that it it is then unfathomable. It is beyond our comprehension and therefore useless. A contradiction.

[1] If this computer wasn't strictly stronger, but its power somehow orthogonal, we could augment the alien computer with a human counterpart creating a strictly more powerful computer.

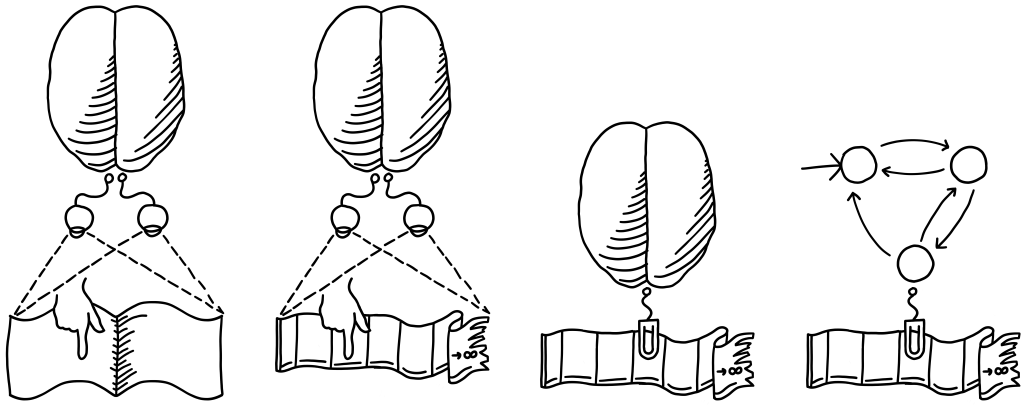


Figure 1: Successive simplications of computation as made in the Direct Appeal to Intuition

Note how we really couldn't argue that such a computer couldn't exist. Only that we couldn't build it, or it would have no effect on us in any measurable, meaningful, or interactive way. If something exists in no way which is fathomable, it doesn't make sense to even discuss it, like an agnostic God. The core of our argument here is that we have been able to simulate every computer so far in our head. Given a DFA and a word, you can run the DFA on the word in your mind. Given psuedocode, you can follow along line by line. If there was a device which we couldn't do this with, it may as well not exist relative to us.

3 Part II: The Direct Appeal to Intuition

These notes have been typeset separately: https://ladha.me/files/fancy-turing-notes.pdf

4 Church Turing Thesis

We now give our statement of the Church-Turing Thesis. For any kind of computation model, mechanical process, decision procedure C,

$$\forall C \ \mathscr{L}(M) \subseteq \mathscr{L}(TM)$$

Its useful to rephrase this as

$$\nexists C \ (TM) \subsetneq \mathscr{L}(C)$$

In human words, the Turing machine is the ultimate computer. The entire complexity and confusion of the mind, at least in our framework of decision problems, can be simplified to the humble Turing machine. This pathetic three button typewriter is equivalent in power to any realizable computer, and there is no greater. It is the supreme.

5 Evidence

We will now show a more traditional argument in favor of the Church-Turing Thesis. We will attempt to generalize the Turing machine. We will show that each generalization is

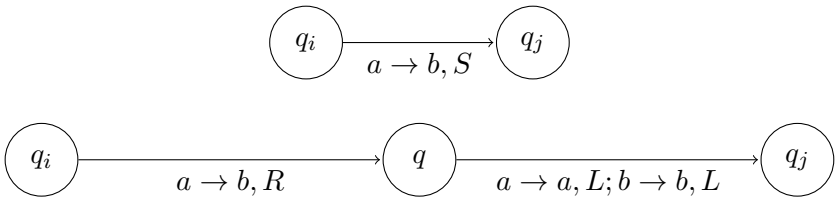
infact, just equivalent.

5.1 Turing machine with Stay

Consider Turing machines with a stay instruction. Instead of moving either left or right, we allow the tape head to remain on the same cell. Its transition function would be defined like:

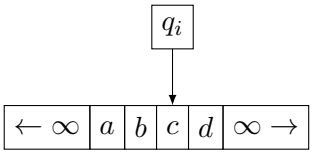
$$\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R, S\}$$

It is obvious that $\mathscr{L}(TM) \subseteq \mathscr{L}(stayTM)$. Let's prove $\mathscr{L}(stayTM) \subseteq \mathscr{L}(TM)$. If a Turing machine with stay has a normal L, R move, we leave it alone. If a Turing machine with stay has a S move, we can simulate it as a sequence of L, R moves. Specifically we will choose to move right, then back left[2].



5.2 Turing machine with Two Way Tape

Our Turing machine's tape is only infinite in one direction, so lets generalize it to be infinite in two. This can be called a bidirectional, doubly infinite, or two way tape.



It is true that $\mathscr{L}(TM) \subseteq \mathscr{L}(2wayTM)$, but the simulation needs an extra sentence. We put our one way tape on the two way tape and add a special marker leftmost of our tape. If we ever read it, we force ourselves right. Now lets show $\mathscr{L}(2wayTM) \subseteq \mathscr{L}(TM)$. There were many excellent simulation suggestions in class, but the elegant one is to just fold the tape in half! Here we extend our tape alphabet to cover pairs as so.

$$\Gamma^2 = \left\{ \frac{a}{b} \mid a, b, \in \Gamma \right\}$$

If you are in the right half of the tape, the transition function will essentially ignore the bottom half of the symbol. If we attempt to move into the left half, we start ignoring the tops and looking at the bottoms. We also flip every L, R move.

[2]The only reason we don't move left then right is the case we are on the leftmost square.

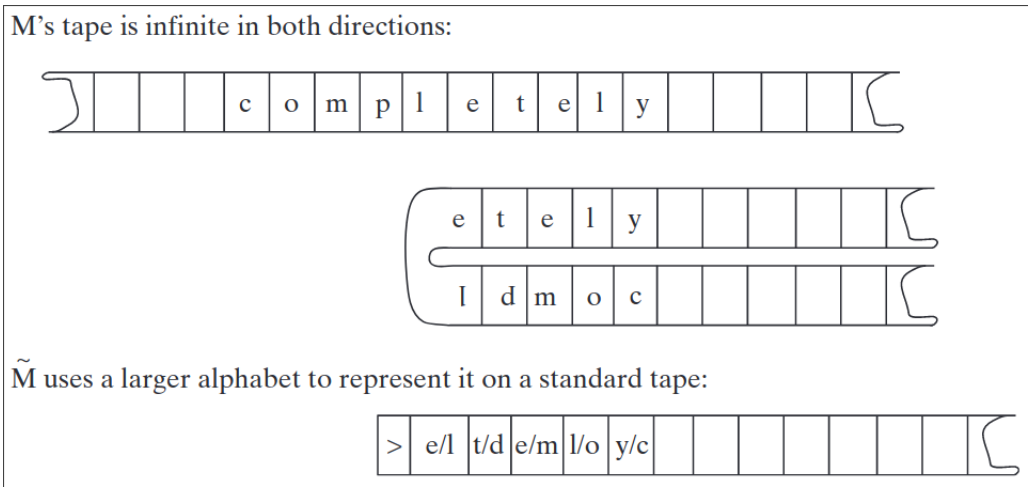


Figure 2: Folded tape in half, from the Arora-Barak book

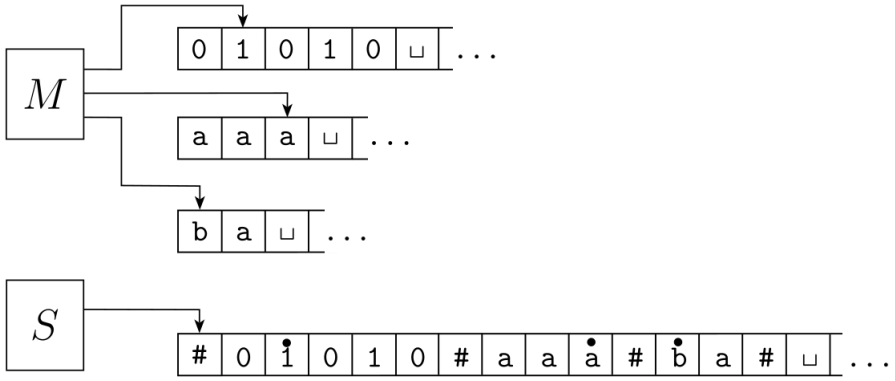


Figure 3: A Three tape Turing machine being simulated on a single tape, from the Sipser book

5.3 Multi-Tape Turing Machine

If you think about it, the negative half of a bidirectional tape is like a second tape. Lets define a Turing machine with multiple tapes. A multi-tape Turing machine is just that. It has k tapes it may read, right and move on all independently. Its transition function would look like

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

At some state, it will read the position of its k tape heads, makes k writes, and moves each head either left or right independently. Certainly $\mathscr{L}(TM) \subseteq \mathscr{L}(kTM)$ by simply ignoring all tapes except the first one.

We want to show $\mathscr{L}(kTM) \subseteq \mathscr{L}(TM)$. In order to do so, we are going to simulate the k tapes on a single tape. We initialize our single tape as

$$\#\dot{w}_1 ... w_n \# \dot{\sqcup} \# \dot{\sqcup} \# \dot{\sqcup} \# ... \# \dot{\sqcup} \#$$

The dot represents the position of that tape head over its tape. As we simulate a read, write, move of each tape head, we will scan over our tape making the appropriate adjustments. If we need more space than allocated, we pause or simulation, enter a shifting subroutine, insert blanks appropriately, and then continue. What was essential for this simulation was that at any time stamp, only a finite amount of space has been used. A Turing machine cannot use the infinite nature of the tape in any useful way.

5.4 Nondeterministic Turing machine

A nondeterministic Turing machine is defined exactly as you might think. At some state reading some symbol, there are more than one outgoing transitions which could be taken. There exists more than one computation path. Its transition function would be defined as

$$\delta : Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R\})$$

Its certainly true that $\mathscr{L}(TM) \subseteq \mathscr{L}(NTM)$ as a nondeterministic Turing machine is simply a generalization of a Turing machine. We now show the reverse way, that $\mathscr{L}(NTM) \subseteq \mathscr{L}(TM)$.

We can view a nondeterministic computation like a rooted tree. Each node in this computation tree can be represented by a configuration, with the initial configuration representing the root, $q_0 w_1 ... w_n$. Our deterministic simulator is going to attempt to search this tree for an accepting computation, if one exists. If our nondeterministic machine has some accepting configuration in this tree, then there exists an accepting computation path and this computation path halts. Our deterministic simulator will find this accepting configuration, and also halt. If our nondeterministic machine has no accepting configuration in this tree, then on all computation paths it either rejects or loops. So our tree may go on forever. Similarly, our deterministic simulator will halt or search the tree forever.

Recall the many tree traversal algorithms. An initial idea is to use DFS: Depth first search. This is not a good idea. If the first computation path we find is some infinite loop, we may miss an easy accepting configuration. The next idea is then to use BFS: Breadth first search. Our deterministic simulator is going to implicitly perform BFS on the computation tree.

Recall how BFS works on a rooted tree. It uses a queue. We pop an element off the queue and push its children. Just as a refresher, consider the following example. Here the traversal of the elements goes layer by layer from our initial node.

We are going to use the tape kind of like a queue. We will pop (read, then erase) a configuration off the front of the tape, compute the possible next configurations, and push (write) them to the end of the tape. For example if our nondeterministic machine had the following structure:

Order of visitation	Queue contents after processing node
	$[S]$
S	$[A\ C\ D\ E]$
A	$[C\ D\ E\ B]$
C	$[D\ E\ B]$
D	$[E\ B]$
E	$[B]$
B	$[\,]$

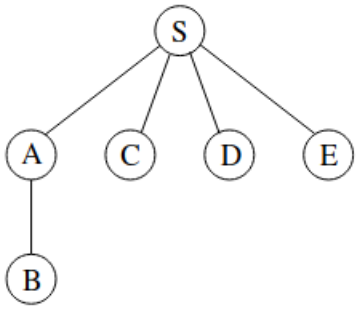
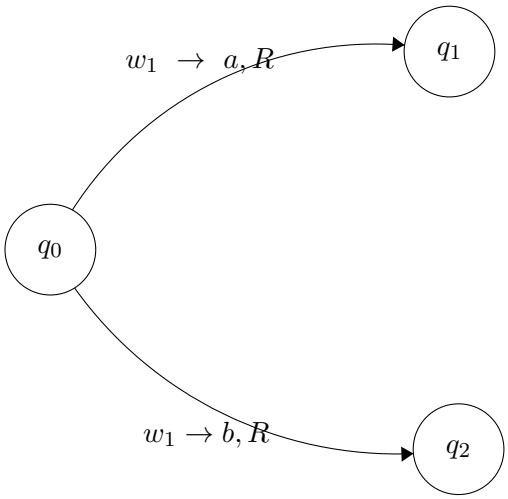
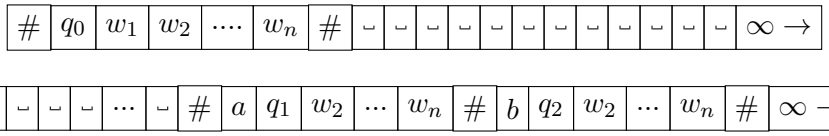


Figure 4: BFS, from the Dasgupta-Papadimitriou-Vazirani book



Then our tape would transform like:



The configuration $q_0 w_1 ... w_n$ yields two configurations nondeterministically, those being $aq_1 w_2 ... w_n, bq_2 w_2 ... w_n$. We would read the initial configuration on the tape, compute its two next configurations, and append those on our tape. We would then erase the initial configuration. We would then next look at the first configuration on the tape and repeat. While we are doing this, we check if a configuration contains an accept state, and if it does, we accept. If it contains a reject state, we toss the configuration and continue searching the others. Its worth noting the Sipser book does a slightly different more detailed simulation, still using BFS but with three tapes. I recommend you read it. We have shown that nondeterministic Turing machines are no more powerful than Turing machines. Keep this simulation in mind when we discuss the resource bounded variant of the question: $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$.

6 Turing Completeness

For any kind of computation model, mechanical process, decision procedure C, we define it to be Turing-complete if

$$\mathscr{L}(TM) \subseteq \mathscr{L}(C)$$

Recall that by the Church-Turing Thesis, we get the reverse implication for free, that $\mathscr{L}(C) \subseteq \mathscr{L}(TM)$. All we need in order to show that a computer is equivalent in power to a Turing machine is to be able to simulate a Turing machine on it. As a brief example, Python is Turing-complete since I believe you could write a Turing machine simulator in it. We could have applied this to the previous four generalizations to immediately get that they must be Turing-complete, but I wanted to actually work through some of the simulations. You may hear the phrase "Turing-complete" in pop culture[3] In practice, this can mean wildly different things which I won't expand on here. I only ask you not get confused. Now we show two surprising models of computation which are Turing-complete.

6.1 Unrestricted Grammar

An unrestricted Grammar is just that. Recall that for a CFG we had productions of the form

$$V \to (V \cup \Sigma)^*$$

A single nonterminal gets substring replaced by an arbitrary string of terminals and nonterminals. In comparison, an unrestricted grammar places no restrictions on the left hand side. It has productions of the form

$$(V \cup \Sigma)^* \to (V \cup \Sigma)^*$$

Any string of terminals and nonterminals can be replaced by any string of terminals and nonterminals. Here we lose the distinction the terminals and nonterminals as well. A string of terminals is not necessarily "terminal", and there may be more productions to follow.

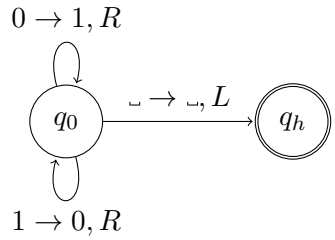
We show that unrestricted grammars are Turing complete. By the Church-Turing thesis, you could write a simulator for an unrestricted grammar, so $\mathscr{L}(UG) \subseteq \mathscr{L}(TM)$. We want to show its Turing complete, so given a Turing machine, we will construct an unrestricted grammar to simulate it. This will prove $\mathscr{L}(TM) \subseteq \mathscr{L}(UG)$.

For any computation of a Turing machine, there exists a sequence of configurations. Our simulation idea is that the sequence of working strings of our grammar will be analogous to this sequence of configurations. The next production we can apply with our unrestricted grammar will be analogous to the next configuration.

If our states were $\{q_0, ..., q_k\}$ then our nonterminals will be $\{S, A, Q_0, .., Q_k\}$ If in our Turing machine $abq_icd \vdash abc'q_jd$, we add productions $Q_ic \to c'Q_j$. Similarly if $abq_icd \vdash aq_jbc'd$ then we add the set of productions $aQ_ic \to Q_jac', bQ_ic \to Q_jbc'$. We add our production to set things up as $S \to Q_0AB$ and then A produces Σ^*, perhaps like $A \to aA \mid bA \mid \varepsilon$. If we have a halting state q_h we add production $Q_h \to \varepsilon$. We also want to read blanks so we have $B \to \sqcup B \mid \varepsilon$

Lets do an example. Recall the Turing machine which simply computes the bitflip of its input and halts.

[3] https://xkcd.com/2556/



We would have a sequence of configurations on input 101 as

$$q_0101 \vdash 0q_001 \vdash 01q_01 \vdash 010q_0\sqcup \vdash 01q_h0$$

Then our grammar would have a sequence of productions like

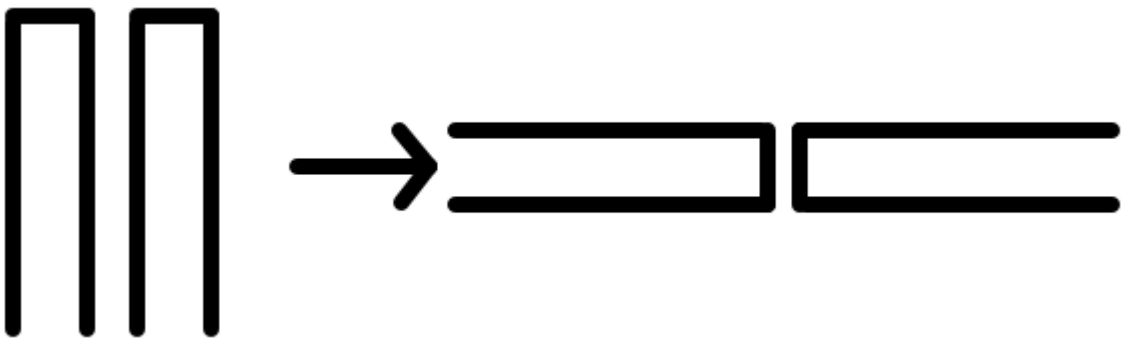
$$S \implies Q_0AB \stackrel{*}{\implies} Q_0101\sqcup \implies 0Q_001\sqcup \implies 01Q_01\sqcup \implies 010Q_0\sqcup \implies 01Q_h0\sqcup \implies 010\sqcup$$

Notice that for any two sequential configurations, only a small local part of the string is changed. This will be essential for many proofs in the future. Its clear that the grammar simulates the machine but we have been vague on the details. If our Turing machine computes a function f, then our grammar will produce $f(\Sigma^*) = \{f(w) \mid w \in \Sigma^*\}$. You can perhaps believe we could fill in the details to get the simulation to accept or reject appropriately, rather than compute everything. Or perhaps compute only the string we wanted. The important part is the simulation.

6.2 $\mathscr{L}(TM) \subseteq \mathscr{L}(2PDA)$

A 2PDA is defined as you might think, a PDA with two stacks. The transition function could be defined to read from them one at a time or to push and pop both simultaneously. Either way, we claim that a PDA with two stacks is Turing complete. First by the Church-Turing Thesis notice that $\mathscr{L}(2PDA) \subseteq \mathscr{L}(TM)$. We will show $\mathscr{L}(TM) \subseteq \mathscr{L}(2PDA)$ by simulation of a Turing machine on a PDA with two stacks.

Recall the intuitive limitation of a PDA with one stack. If you need to read deep into the stack, you have to pop things out losing them into the ether. With two stacks, instead of popping them out and losing them, just push them into the second stack. We join our two stacks together to form a bidirectional tape! The proof becomes obvious by the following change in perspective:



Denote one stack as the "left" stack and the other as the "right" stack. For some Turing machine right move of the form $a \to b, R$, we pop a from the right stack and push b to the left stack. For some Turing machine left move of the form $c \to d, L$, we pop c from the right stack, push d to the right stack. Then we pop whatever is on top of the left stack and push it to the right stack.

Here, we get a interesting observation. A PDA with one stack (PDA) is strictly stronger than a PDA with no stacks (NFA). A PDA with two stacks (Turing-complete) is strictly stronger than a PDA with one stack (context-free). But a PDA with three stacks is not strictly stronger than a PDA with two stacks, its equivalent. In the language of set theory:

$$\mathscr{L}(0PDA) \subsetneq \mathscr{L}(1PDA) \subsetneq \mathscr{L}(2PDA) = \mathscr{L}(3PDA) = \mathscr{L}(4PDA) = ...$$

Two stacks is the limit!. Its just enough to get all computation. The "degrees of freedom" or amount of power a machine needs to be Turing-complete is relatively small.

7 Physical Realizability

A common theme in any Turing-complete model of computation is that we appeal to physics and intuition. They all share some common traits.

- Program descriptions are of finite length
- A constant amount of work done in unit time. If more work needs to be done, successive operations must be performed

This is the heart of the Church-Turing Thesis. Our understanding of algorithms is invariant in the way we choose to represent them. As long as some basic requirements are met, many models of computation are Turing-complete. There do exist theoretical "Super-Turing" models of computation, but these are explicitly unrealistic on purpose. For example, they allow $\Gamma = \mathbb{Q}$. This would allow you to encode any string into a single cell, and compare arbitrarily long strings in constant time. Something totally infeasible and unintuitive.

CS 4510 Automata and Complexity February 27th 2023

Lecture 12: Countability

Lecturer: Abrahim Ladha *Scribe(s): Rishabh Singhal*

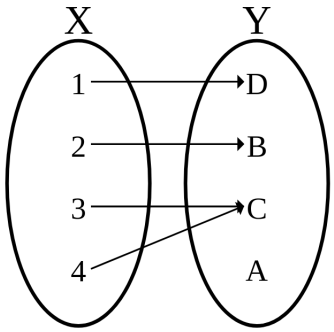
1 Introduction

Infinity used to be just a figure of speech, or perhaps a useful abstraction, not a real thing. In the late 19th and early 20th centuries, Georg Cantor undertook a serious attempt to formalize and understand the infinite, generalizing ideas from finite sets to infinite ones.

We denote the cardinality of the set S as $|S|$. If S is finite then $|S|$ is just the size. But what is the cardinality of the natural numbers $|\mathbb{N}|$? Certainly for all finite sets F, it is true that $|F| < |\mathbb{N}|$. When we talk about the cardinality of infinite sets, we want to preserve our intuition as much as possible. If A is a subset of B then $A \subseteq B \implies |A| \leq |B|$.

Definition: We say a set S is "countable" if $|S| \leq |\mathbb{N}|$. All finite sets are countable. We say a set is "countably infinite" if $|S| = |\mathbb{N}|$. How can we show that a set has the same cardinality as natural numbers?

- Recall $f : A \to B$ is one to one (injective) if $f(a) = f(b) \implies a = b$.
- Recall $f : A \to B$ is onto (surjective) if $\forall y \; \exists x$ such that $f(x) = y$. There do not exist any unmapped elements in the co-domain.



See how both 3,4 map to the same element? That makes this function **not** injective. See how A is unmapped? That makes this function also **not** surjective. We say a function is bijective if it is injective and surjective. Bijection gives us a natural "same size-ness" because if there is a bijection between two sets, the elements seem to pair up nicely, meaning they should have the same size.

Definition: We say a set S is countably infinite if $\exists \; f : \mathbb{N} \to S$ which is a bijection. Recall the inverse of a bijection is also a bijection so equivalently if $\exists \; g : S \to \mathbb{N}$ which is bijective.

2 Examples of Countably Infinite Sets

2.1 Those "other naturals"

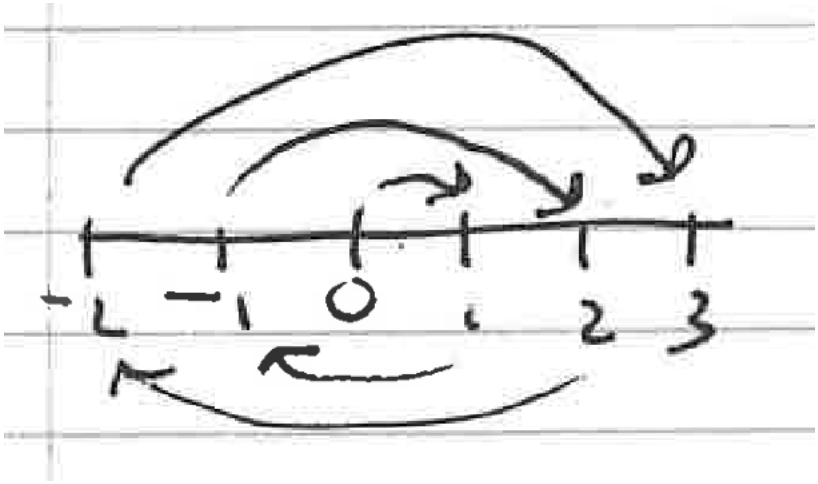
Outside of this class you may not consider zero to be a natural number. Lets prove it doesn't really matter, $|\mathbb{N}| = |\mathbb{N}_{\geq 1}|$. Recall that $\mathbb{N} = \{0, 1, 2, 3, ...\}$ and $\mathbb{N}_{\geq 1} = \{1, 2, 3, ...\}$. To prove these sets have the same cardinality, we give an obvious bijection. Namely $f : \mathbb{N} \to \mathbb{N}_{\geq 1}$ by $f(n) = n + 1$. The elements pair up obviously like $0 \to 1, 1 \to 2$ and so on, so our function is certainly bijective. This shows that if you add or remove a constant amount of elements from a countably infinite set, its still countably infinite.

2.2 The Evens

What is the cardinality of the even numbers? Define $2\mathbb{N} = \{0, 2, 4, 8, ...\}$. Our bijection is again obviously $f(n) = 2n$. This shows that an infinite subset of a countably infinite set is still countably infinite.

2.3 The Integers

Recall $\mathbb{Z} = \{-2, -1, 0, 1, 2, ...\}$. When you are asked to give a bijection, it is equivalent to showing that you can order the elements of a set in some way. Intuitively, if you can "count" them. A bad idea is to first order the elements like 0,1,2,... because then we will never reach the negative numbers. A better idea is to dovetail the negative and positive integers in the following way.



If you were to actually work out what this bijection would be like functionally, you would get

$$f(n) = \begin{cases} \frac{-n}{2} & n \text{ is even} \\ \frac{n+1}{2} & n \text{ is odd} \end{cases}$$

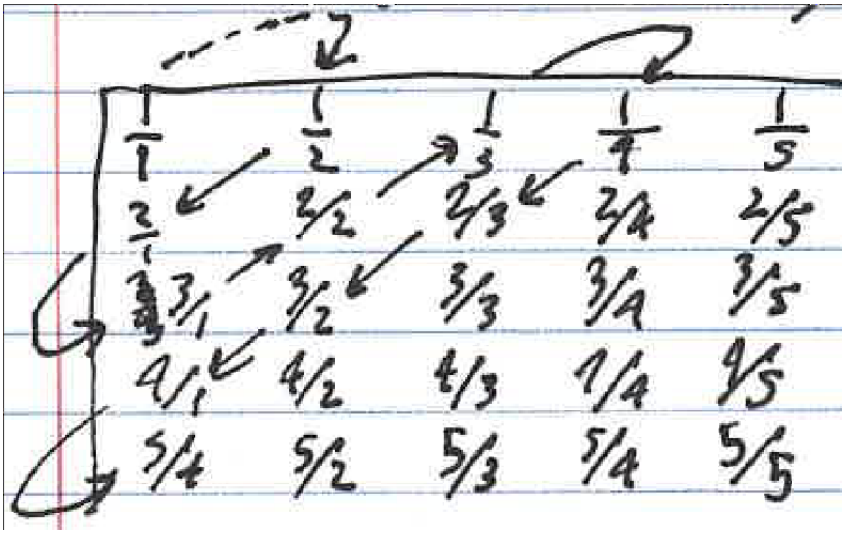
The integers feel like "twice as many" of the naturals so this can show that two countably infinite sets is countably infinite. A countability infinite also need not be well-ordered.

2.4 The Rationals

We define the rational numbers as $\mathbb{Q} = \{ \frac{a}{b} \mid a, b \in \mathbb{N}; a, b \neq 0\}$. They do not contain repetitions, so $\frac{1}{1}, \frac{2}{2}$ are not distinct. Rational numbers have some very different properties than the previous examples. For example for the natural numbers, there is only a finite number of naturals between any two naturals, but this isn't true for the rationals.

- $\mathbb{N}$: $\exists x$ where $0 < x < 1$? no
- $\mathbb{Q}$: $\exists x$ where $\frac{a}{b} < x < \frac{c}{d}$? yes

The naturals appear in discrete steps, but between any two rationals, there exists an infinite amount of rationals. Why? The average of any two rationals is a rational, so the midpoint between any two, you will find a rational[1]. Recursively applying this idea will give you an infinite amount between any two! The mathematically correct term for this is "dense". Could there be more rationals than naturals? It feels like there is a lot more of them. It turns out even despite this, the rationals are still only countably infinite, that $|\mathbb{N}| = |\mathbb{Q}|$. This bijection is a little less obvious. Put all the rationals into a table with columns and rows ordered by numerators and denominators. A bad idea would be to try to go left to right row by row. You would never reach the second row. The idea is then to compose the anti-diagonals ignoring duplicates!



This certainly is a bijection. Its surjective since every element is hit somewhere in this criss-crossing, since every element is on some anti-diagonal. Its injective as every element only can appear once in this ordering.

Here's another solution. Consider the function $f(a/b) = 2^a 3^b$. This function is bijective to some set $S = \{2^i 3^j \mid i, j \in \mathbb{N}_{\geq 1}\}$. Notice that $|\mathbb{Q}| = |S|$. Also notice that since $S \subseteq \mathbb{N}$ then $|S| \leq |\mathbb{N}|$. So by transitivity $|\mathbb{Q}| = |S| \leq |\mathbb{N}| \implies |\mathbb{Q}| \leq |\mathbb{N}|$. We also know that $|\mathbb{N}| \leq |\mathbb{Q}|$ by the injection $f(a) = \frac{a}{1}$ so combined we see that $|\mathbb{Q}| = |\mathbb{N}|$. We could have also

[1] If you wanted to work it out, the rational between $\frac{a}{b}, \frac{c}{d}$ is $\frac{a}{b} + (\frac{c}{d} - \frac{a}{b})/2$. You could simplify that with arithmetic to get some rational.

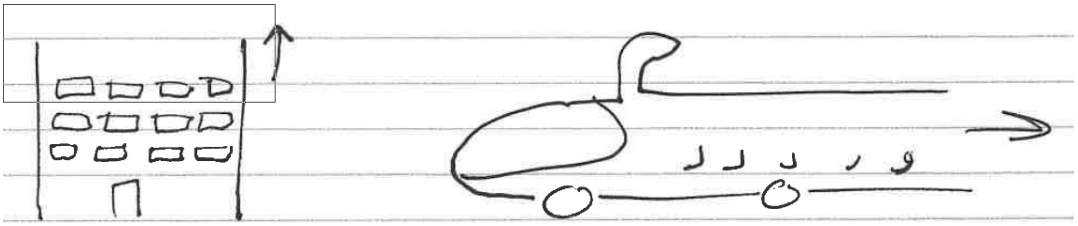
just observed that since $|\mathbb{Q}| \leq |\mathbb{N}|$, we know $\mathbb{Q}$ is countable. Observing that $\mathbb{Q}$ is infinite is enough to show it must be countably infinite.

2.5 Cartesian Products

The rationals are really just like, pairs of numbers. If we are tasked with finding a bijection for $\mathbb{N} \times \mathbb{N}$, we can immediately apply the same argument with the table and anti-diagonals. This is enough to prove that the cartesian product of two countable sets is countable. We can also immediately induct this argument to get that finitely many cartesian products of countable sets is countable. Notice that $\mathbb{N} \times \mathbb{N} \times \mathbb{N} = (\mathbb{N} \times \mathbb{N}) \times \mathbb{N}$. We know that $\mathbb{N} \times \mathbb{N}$ is countable. It remains countable if we perform one more cartesian product.

3 Hillbert's Hotel

Suppose we have an infinitely tall hotel of countably infinite rooms. Each room already has a guest, so the hotel is full.



- A single new guest arrives. Although every room already has a guest, the hotel staff aren't worried. They make each old guest move from room n into the next room, room $n+1$. Now room zero is empty for the new guest.
- Suppose an infinitely long bus arrives with countably infinite new guests. Even though the hotel seemingly has no space, the new guests can still be accomodated. Tell each old guest to move from room n to $2n$, then each of the new guests to move into the now empty odd-numbered rooms.
- What if a countably infinite number of infinitely long busses arrive, each with countably infinite more guests? I claim they can still be accomodated, and I leave it to you as an exercise to figure out how.

4 Cantor's Theorem

It would seem that you can play with infinity in most ways and remain countably infinite. If we were to say that $|\mathbb{N}| = \infty$, then the vibes are that $\infty + 3, 3 \cdot \infty, \infty^3$ all equal to ∞. These are all polynomially related. Could it be the case that $2^\infty = \infty$? It turns out, no. Lets denote $|\mathbb{N}| = \aleph_0$ and $2^{\aleph_0} = \aleph_1$. We will show these are two very different infinities.

Cantor's theorem tells us that there is no bijection between any set, and its power set[2].

$$|A| < |\mathcal{P}(A)|$$

Note that its obviously true for finite sets. if $A = \{x, y\}$ then $\mathcal{P}(A) = \{\emptyset, \{x\}, \{y\}, \{x, y\}\}$ and $|\mathcal{P}(A)| = 2^{|A|}$.

4.1 Diagonalization

We will prove Cantor's theorem for the countably infinite case. To do so, we present a new technique, called diagonalization. First we define the characterisitic sequence of a subset. If $S \subseteq \mathbb{N}$, to S we associate the infinitely long binary sequence $\chi \in \Sigma^\infty$ such that

$$X[i] = \begin{cases} 1 & i \in S \\ 0 & i \notin S \end{cases}$$

For example

- if $S = \{0, 3, 4\}$ then $\chi = 10011000000...$
- if $S = 2\mathbb{N}$ then $\chi = 10101010...$
- if $S = \mathbb{N}$ then $\chi = 11111...$
- if $S = \emptyset$ then $\chi = 00000...$

Notice immediately that to each subset, corresponds a unique characteristic sequence. There is a bijection between the set of infinitely long binary sequences, and the subsets of a countably infinite set. The infinite sequence of digits exactly characterizes which elements are and aren't in a subset. What is a subset if not just a selection of the elements? It is also important to remember that these sequences are infinitely long.

Let us proceed with the proof. Assume to the contrary that there exists some bijection $f : A \to \mathcal{P}(A)$ with A countably infinite. The elements of $\mathcal{P}(A)$ are exactly the subsets of A. So then there exists an ordering of the elements of $\mathcal{P}(A)$ like $S_0, S_1, S_2, ...$, where every element is in this ordering. Let $\chi_0, \chi_1, \chi_2, ...$ be the characteristic sequences of $S_0, S_1, S_2, ...$ defined in the same ordering. We define "the diagonal" D to be the infinite binary sequence with digits defined as

$$D[i] = 1 - \chi_i[i] = \overline{\chi_i[i]}$$

We take our ordering of characteristic sequences, find the ith one, find its ith digit, and then set the ith digit of D to be the exact opposite of that. D certainly is an infinite binary sequence, so it must be the characteristic sequence of some subset. Since f is bijective, D exists somewhere in our ordering. Suppose the subset corresponding to D is S_j in our order $S_0, S_1, S_2, ...$ Then D is the jth sequence in $\chi_0, \chi_1, \chi_2, ...$ so $D = \chi_j$.

What is $D[j]$? Well, since $D = \chi_j$ then $D[j] = \chi_j[j]$. But recall how we originally defined D, where $D[j] = 1 - \chi_j[j]$. Together, these imply that

$$\chi_j[j] = \overline{\chi_j[j]}$$

[2]Recall that a power set is the set of all subsets of a set

A digit cannot be zero and one simultaneously! Therefore, we see that we have reached a contradiction, and $|\mathcal{P}(A)|$ is not countable. Why is it called diagonalization? Well suppose you listed $\chi_0, \chi_1, ...$ into a table with each χ_i as a row:

χ_0	**0**	1	1	0	1	1	0	...
χ_1	0	**1**	0	0	0	0	0	...
χ_2	0	0	**0**	0	1	1	1	...
χ_3	0	0	1	**1**	0	0	1	...
χ_4	0	1	0	1	**1**	0	1	...
χ_5	0	0	1	1	1	**0**	0	...
...	...							

Then $D = 101001...$ is the opposite of the diagonal of the table. Since D is different than any row of the table, it exists no where in the table. For each row, it is defined to be different in atleast one place, namely the diagonal (i, i) but maybe more. Could it be χ_3? No because $\chi_3[3] = 1$ and $D[3] = 0$. Could it be χ_4? no, and so on. We assumed to the contrary that these sequences were countable and that we can order them, but no matter how we order them, we can always construct an element not in the ordering. So there can never exist a bijection $f : A \to \mathcal{P}(A)$. It is important to convince yourself that this argument is not circular, but self-referential.

5 Uncountability

We have now shown that $\mathcal{P}(A)$ is not countably infinite when A is countably infinite, it is something greater. We call these sets uncountable. Intuitively, a countably infinite set is one in which you can "count". It feels infinite in a discrete sense. At some element, you can choose a next one. Conversely, an uncountable set is literally "uncountable". Imagine a stream of water. What are the units? What is the "next" water?[3]. It feels infinite in a continuous sense.

By a similar diagonalization argument, you can prove the real interval $(0, 1]$ is uncountable, by diagonalizing over the decimal expansions beginning with zero[4]. Given that $(0, 1]$ is uncountable, you can prove that $\mathbb{R}_{\geq 0}$ is uncountable by the bijection $f(r) = 1/r - 1$. Essentially you can stretch the unit sized interval over the entire real positive line. We could have also performed the diagonalization proof directly on infinitely long binary strings Σ^∞ to show they are uncountable.

6 How to Prove Countability

6.1 Union of Two Countable Sets

Let A, B be countably infinite. Then there exists bijections $f : A \to \mathbb{N}$, $g : B \to \mathbb{N}$. We give a bijection for $A \cup B$ as

[3] If you recall that water is atoms then technically water is discrete and countable but the intuition is there even if the science isn't

[4] recall that $0.\overline{9} = 1$. There are a few proofs of this. One is that $1 - 0.999... = 0.0000...$ and another is to notice that $0.999... = 0.333... + 0.333... + 0.333... = 3(0.333...) = 3\frac{1}{3} = 1$

$$h(x) = \begin{cases} 2f(x) & \text{if } x \in A \\ 2g(x)+1 & \text{if } x \in B \end{cases}$$

We leave it to you as an exercise to show its bijective, reducing to the bijectivity of f, g.

6.2 Countable Union of Countable Sets

A countable union of countable sets is countable. Most unions you have ever seen have been countable. They index over $\mathbb{N}$ with $i = 1, 2, 3, ...$ but the index set of a union need not be countable in general. Consider

$$\bigcup_{x \in \mathbb{R}} \{x\} = \mathbb{R}$$

Here we index over $\mathbb{R}$, an uncountable set. Each element is a singleton containing just x, it is finite and therefore countable. But our union is over $\mathbb{R}$, uncountable. We have an uncountable union of countable sets, yet, it is uncountable.

Lets prove that a countable union of countable sets is countable. Let A be countable and each S_i be countable.

$$|\bigcup_{i \in A} S_i| \leq |A \times \mathbb{N}| \leq |\mathbb{N} \times \mathbb{N}| = |\mathbb{N}|$$

The first inequality holds by reordering the elements, and maximally assuming each S_i is not finite. The second inequality holds by assuming that A is not finite. The third inequality holds by what we previously proved. So a countable union of countable sets is countable. This proof is actually very rough and requires more rigor, but you get the idea.

6.3 Three solutions

Let's do a problem. Let $\mathbb{N}^*_{\geq 1}$ be the set of finite sequences of natural numbers greater than one. It may contain things like $[1, 11, 1]$ or $[23, 100, 18]$ and so on. We give three solutions to showing this set is countably infinite.

- Let A_i = sequences which sum to i, for example A_3 would contain $[1, 1, 1], [3], [2, 1]$ and so on. Since each sequence sums to something, The A_i's partition $\mathbb{N}^*_{\geq 1}$

  $$\mathbb{N}^*_{\geq 1} = \bigcup_{i=1}^{\infty} A_i$$

  Notice that each A_i is finite, so countable. Then $\mathbb{N}^*_{\geq 1}$ is a countable union of countable sets, so its countable.

- consider the map: $F([x_1, x_2, x_3, ...]) = 2^{x_1} 3^{x_2} 5^{x_3} ...$ or more generally

  $$F([x_1, ..., x_k]) = \prod_{i=1}^{k} p_i^{x_i}$$

where p_i is the i'th prime. By the fundamental theorem of arithmetic, every number has a unique prime factorization, and this immediately gives us that our map is injective. Suppose two sequences exist a, b with $F(a) = F(b)$. Then they are divisible by exactly the same power of two, so then they share the same first element, x_1. Repeating this argument we see that $a = b$. Therefore, we have an injection $F : \mathbb{N}^*_{\geq 1} \to \mathbb{N}$ which implies that it is countable.

- There is a injection hiding in us all along. What is the difference between the two sequences $[1, 1]$ and $[2, 3, 4]$? Is it the length? Is it the number of elements? I put these on the board, you immediately know that the sequences are different. You didn't check the lengths or the elements, so how you did you know? The answer is that the two sequences are different because they look different! Define our injection $f(a) =$ "a". That is, it is the string casting function. Now its certainly true that "$[1, 1]$" $\neq$ "$[2, 3, 4]$". We observe that $f(\mathbb{N}^*_{\geq 1}) \subseteq \Sigma^*$ and subsets of countable sets are countable. Why is Σ^* countable? It is the countable union of countable sets. Recall $\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup ...$

This last point leads us to a powerful theorem called the **Typewriter principle**: If some set S has elements $a \in S$ where every element can be *uniquely* described by a string. Then S is countable. Lets prove it. If every element of S can uniquely be described by a string, then $f : S \to \Sigma^*$ is injective. The co-domain $f(S)$ is a set of strings, so $f(S) \subseteq \Sigma^*$ and f is certainly bijective to $f(S)$ so we see that S is bijective to a subset of a countable set, and is therefore, countable. This is not sufficient to show uncountability. Showing some elements of a set have some infinite encoding isn't enough, since you must also show that there does not exist a unique finite encoding. This turns out to be as hard as finding a bijection. Please only use it to show countability.

We now have an entire toolbox to show a set is countable. Let C be any countable set, and we want to show S is countable. We can do any of the following

- Give a bijection $f : C \to S$
- Give a bijection $f : S \to C$
- Give an injection $f : S \to C$
- Give a surjection $f : C \to C$
- Give an ordering of every element where no element appears twice
- Show that S is a subset of some countable set, since $S \subseteq C \implies |S| \leq |C| \implies |S| \leq \aleph_0$
- Show that S is representable as a countable union of countable sets
- Arrange the elements of S into a grid and compose the anti-diagonals, or some other pattern to implicitly give a bijection
- Show it is the closed under operations we know do not change the cardinality of the set, for example $S = (\{C \times C\} \cup \{0, 1\})^k$.

- Show that its elements can be uniquely represented as finite strings and apply the typewriter principle.

Common known countable sets include $\mathbb{N}, \mathbb{Z}, \Sigma^*, \mathbb{N}^*$, and so on. Every language is also countable.

7 How to prove Uncountability

Let U be some known uncountable set. We give several ways to show a set S is uncountable.

- Diagonalization
- Find a bijection $f : S \to U$
- Show that S contains some uncountable subset. Since if $U \subseteq S$ is uncountable then $|U| \leq |S| \implies \aleph_1 \leq |U|$
- Find an injection $f : S \to U$
- Apply Cantor's theorem, show that it is the powerset $\mathcal{P}(A)$ of some countably infinite A

We do have far fewer ways to show a set is uncountable than to show a set is countable. Which tool you use depends on ease of use.

8 Rejection

What are numbers? They were not there when we started all this. We logically construct the naturals by defining zero to exist, and the sucessor function $S(x) = x + 1$. By repeated application we can produce the numbers. It is well understood that they are the product of some infinite process. 0,1,2,... Ongoing. Forever. There are those who reject this idea. They do not object to the naturals, but the manipulation of an infinite process. They distinguish between the infinite process 0,1,2,... and calling this infinite process $\mathbb{N} = \{0, 1, 2, ...\}$, and then messing around with $\mathbb{N}$. These are the finists. But without what they object to, we are unable to construct the countable and uncountable. There is an even stronger group, known as the ultrafinists. I will leave you with a quote.
I have seen some ultrafinitists go so far as to challenge the existence of 2^{100} *as a natural number, in the sense of there being a series of "points" of that length. There is the obvious "draw the line" objection, asking where in* $2^1, 2^2, 2^3, ..., 2^{100}$ *do we stop having "Platonistic reality"? Here this . . . is totally innocent, in that it can be easily be replaced by 100 items (names) separated by commas. I raised just this objection with the (extreme) ultrafinitist Yessenin-Volpin during a lecture of his. He asked me to be more specific. I then proceeded to start with* 2^1 *and asked him whether this is "real" or something to that effect. He virtually immediately said yes. Then I asked about* 2^2 *, and he again said yes, but with a perceptible delay. Then* 2^3 *, and yes, but with more delay. This continued for a couple of more times, till it was obvious how he was handling this objection. Sure, he was prepared to always answer yes, but he was going to take* 2^{100} *times as long to answer yes to* 2^{100} *then he would to answering* 2^1*. There is no way that I could get very far with this.*

CS 4510 Automata and Complexity March 8th 2023

Lecture 13: Foundation of Mathematics

Lecturer: Abrahim Ladha *Scribe(s): Yitong Li*

1 A Motivation from Geometry

Recall last time we had a lecture on pure mathematics, countability, and set theory. The lecture before that was kind of pseudo-philosophical on the Church-Turing thesis. The lecture before that one was on engineering, programming and understanding the Turing machine. To keep the trend of not having one, today's lecture will be on history.

We will go from 300 BC to 1936. We begin of course, with the Greeks. Around 300 BC, Euclid wrote "The Elements", several treatises in geometry. It is one of the most influential texts of all time. It established mathematics as a deductive rather than empirical science. It has been in print for millennia and comes second only to the bible. Just because calculus wasn't invented yet didn't mean you didn't have math class. You used to have to take a three course series on classical geometry.

Recall that a theorem is some statement proved. From what? Other theorems? Not quite. There is some flow of implications in this giant tree of knowledge. Follow back to eventually reach some root: The axioms. An axiom is a statement that needs no proof. It may be assumed to be true. It is usually so trivial to be anything but true. For example, associativity of addition: $(a + b) + c = a + (b + c)$.

Euclid defined[1] his first five axioms, or postulates as follows:

1. any two points may be connected by a line segment.
2. any line segment may be extended infinitely in both directions.
3. for any point and radius, there exists a circle.
4. all right angles equal each other
5. given a line l and a point p, not on that line, there exists exactly one line through p parallel to l.

A proof is an application of axioms with the "rules of deduction" which are themselves axioms. All of the axioms for Euclid's elements are a model for what we now call Euclidean geometry. From the axioms, you can prove things like: every square has four equal right angles, the sum of the interior angles of a triangle is 180°, if a triangle has two equal angles it has two equal sides, and so on.

Euclid's elements have nothing to do with geometry. It is about rigorous and systematic thinking. It is nothing more than a by-product of the school of thought that Euclid and other

[1] Of course, he did it in ancient Greek. Here they have been modernly rephrased. Look up Playfair's axiom if you are interested in this rephrasing.

Greeks came from. Plato's Theory of Forms asserts that ideas are the supreme achievement of human beings. They are a refined, pure reflection of the capability of the human mind. There exists the Real: the material, empirical, measurable, and approximate world. There also exists the Ideal: one of concepts and thought. The Real and Ideal certainly have a duality[2]. This school of thought asserts that the world, the materiality, is shaped by some things prior to it, the immaterial. When I draw a triangle on the board, understand this exists no where except in your mind. No Real triangle you can form from sticks, or by drawing in the sand can ever reach the precision of the Ideal triangle. However, by studying the Ideal, it may reveal to you something about the Real. To understand the material, you only need to understand the non-material. Abraham Lincoln famously used the Elements to train as a lawyer.

> At last I said, 'Lincoln, you never can make a lawyer if you do not understand what demonstrate means'; and I left my situation in Springfield, went home to my father's house, and stayed there till I could give any proposition in the six books of Euclid at sight. I then found out what demonstrate means, and went back to my law studies.

In 1854 in an unpublished note, he used this rigorous thinking to assert abolition.

> If A. can prove, however conclusively, that he may, of right, enslave B. — why may not B. snatch the same argument, and prove equally, that he may enslave A? – You say A. is white, and B. is black. It is color, then; the lighter, having the right to enslave the darker? Take care. By this rule, you are to be slave to the first man you meet, with a fairer skin than your own. You do not mean color exactly? – You mean the whites are intellectually the superiors of the blacks, and, therefore have the right to enslave them? Take care again. By this rule, you are to be slave to the first man you meet, with an intellect superior to your own. But, say you, it is a question of interest; and, if you can make it your interest, you have the right to enslave another. Very well. And if he can make it his interest, he has the right to enslave you.

Millennia was spent trying to refine Euclid's Elements, to show they were only as good and simple as necessary. A set of axioms is independent if no axiom can be proved from the others, like independence with respect to a basis of a vector space. If an axiom could be proved from the others, then it need not be an axiom. Remove it, and simply take it as a theorem. The fifth axiom took a lot attention as if it was the first really unobvious one. The first four are just definitions. Let A be the parallel postulate and EE the axioms.

First, people tried to see if you could derive the parallel postulate from the others. Notationally, we would present this as $(EE - A) \vdash A$? Here $\vdash$[3] means provable from a set of axioms. It is similar to an implication. We now know the fifth postulate is independent, so this is impossible.

Secondly, we wanted to see if it was even a necessary axiom. So there were attempts to prove that $(EE - A + \neg A) \vdash 0 = 1$. That is, if you removed the axiom and assumed its

[2]Maybe are better known by other names to you, theory and practice?

[3]latex is `vdash`

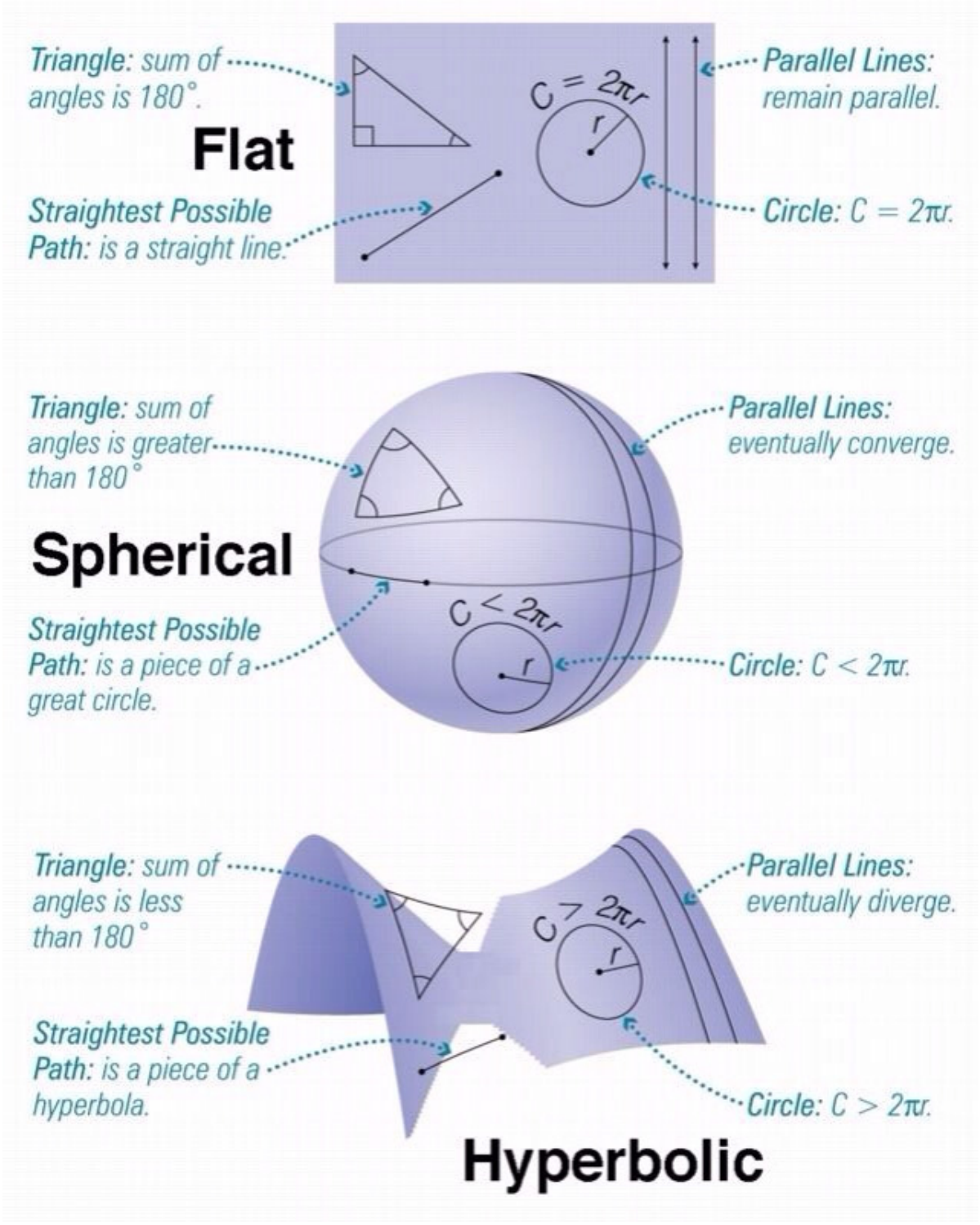


Figure 1: Three models of geometry, and their different theorems

negation, you would produce a contradiction. This should work for any usable and necessary axiom, but they discovered something insane. Taking the negation doesn't produce an inconsistency, instead it produces two different consistent models!

Recall the parallel postulate says "Given a line and a point not on that line, there exists **exactly one** line through the point parallel to the line." Our negations will be changing "exactly one" to "greater than one" or "less than one". We are changing the number of possible parallel lines from $= 1$ to > 1 or < 1.

If the number of parallel lines through the point equals one, we live in a flat world, the Euclidean plane. Any other parallel lines would be equivalent since they intersect at all points. If the number of parallel lines were greater than one, then we exist not on the plane, but on a saddle, or a pringle. There exists many "parallel" lines through the point which do not intersect our line. This is also called hyperbolic, or Lobachevskian geometry. If the number of parallel lines is less than one, as in there do not exist any parallel lines anywhere ever, we are in the model of spherical or ellipsoid geometry. We are embedded onto an egg, or globe. The lines on a sphere are only the great circles, smaller bands of latitudes are curves.

Note that these are consistent models. They cannot prove $0 = 1$ but all the theorems which these models derive are slightly different. For example, in spherical geometry, a triangle has the sum of its interior angles $> 180°$. You can construct a triangle with three right angles, something impossible to do in the Euclidean plane. Begin at the equator, go to the north pole, turn 90 degrees, go back to the equator, and turn 90 degrees again to go back to where you began. On a sphere, a triangle has the sum of its interior angles equal to $180°$ if and only if has area zero[4]. Euclidean geometry is supposed to be "intuitive," but we discovered it was on all shaky foundations. We formulated Euclidean geometry following our empirical experiences in the Real, measuring angles and generalizing our observations. Maybe the Real could follow these models instead? Who can say? How can we know that every time weve measured the angles of a triangle, it hasn't technically been $180 + \varepsilon$ this whole time? We do live on a sphere after all. This sparked more serious concerns about the foundation of all of mathematics.

2 A Foundation from a Theory of Sets

A modern first attempt of formalizing mathematics was done by Gottlob Frege's "Begriffsschrift". It builds off of previous work by Aristotle's Organon and Leibniz. It identifies as "a formal language modeled on that of arithmetic, for pure thought". He also created many of our rules of deduction, for example

- $A \implies \neg\neg A$
- $c = d \implies f(c) = f(d)$
- $(A \implies B) \wedge A \implies B$[5]

[4] In fact, the fifth postulate is logically equivalent to "there exists a triangle whose interior angles sum to $180°$"

[5] This is called modus ponens

Theses are just a few of the rules of deduction noted by Frege, that we now use for propositional logic. You may use these without even realizing they are axioms.

Set theory was a natural foundation for mathematics, as they can derive numbers but also greater and more interesting structures. The motivation was to create a unifying foundation for all of mathematics. Bertrand Russell noticed the following issue, applied to many axiomatic systems. Suppose we have the following axioms for some simple theory of sets

$$\forall x \forall y [\forall z (z \in x \Leftrightarrow z \in y) \Rightarrow x = y] \tag{1}$$

The axiom of extensionality simply defines the equality relation of sets. Two sets are equal if they contain the same elements.

$$\exists y \forall x [x \in y \Leftrightarrow \varphi(x)] \tag{2}$$

For any predicate φ, the axiom of unrestricted comprehension basically freely allows you to define any set you want. It allows you to make statements like "let y be the set of primes, or horses or whatever". If you can define it with a logical predicate, then there exists a set of those elements. As an example of a predicate, the following is one for the prime numbers

$$Prime(x) = \forall x \neg \exists z [(x > 1) \wedge (z \leq x) \wedge \neg (z = 1) \wedge \neg (z = x) \wedge \neg (z|x)] \tag{3}$$

We however, are in a theory of sets. We can construct the numbers from the sets, and we will describe how to later. But for now consider predicates over sets. This generality of unrestricted comprehension is also our fragility. Consider the set of all sets which do not contain themselves. Let $\varphi(x) = x \notin x$. A perfectly valid predicate. By the axiom of unrestricted comprehension, we see that

$$\exists y \forall x [x \in y \Leftrightarrow x \notin x]$$

Since it's true $\forall x$, we may specify, and consider the case for one selection of x. What happens for $x = y$?

$$y \in y \iff y \notin y$$

A contradiction![6] We are not in a proof by contradiction, yet we have derived a contradiction. We have shown that Frege's axiomatic system was capable of producing an inconsistency. Although this attack was devastating to Frege's words, it did not deter much the resolve of the formalists. The formalists are a school of thought revolving around building Hilbert's program. They seek a secure, rigorous, and logical foundation in which to secure all of mathematics. Roughly with these goals:

1. All math written in a precise formal language manipulated according to well-defined rules

2. Completeness: all that is true is provable

[6] see Logicomix 162-171

3. Consistency: you should provably be unable to obtain a contradiction

4. Decidability: There should exist an algorithm to decide the truth value of any statement.

The most significant effort in this regard was by Russell and Whitehead, They spent decades and thousands of pages to build up a "Theory of Types." To give a quick summary of twenty years of work, they hoped to avoid self-reference by using these types. Anything of some type i is unable to construct sentences which reference other things of type i (including itself). They thought removing self-reference would create a strong and bulletproof foundation. The axiom of unrestricted comprehension was modified to the axiom of restricted comprehension. Now you can only construct subsets of other sets. By restricting comprehension, we appear to avoid Russell's paradox.

- unrestricted comprehension: $\{x \mid \varphi(x)\}$
- restricted comprehension: $\{x \subseteq z \mid \varphi(x)\}$

Is this system useful? Let's roughly prove $1 = 1$. First let $\emptyset$ exist by the axiom of restricted comprehension with some useless $\varphi(x) = \{x \in y \mid (x \in x) \wedge \neg(x \in x)\}$. Nothing satisfies it so we create $\emptyset$. Sometimes the empty set exists axiomatically, but here it follows from restricted comprehension. Now let $0 := \emptyset$, for shorthand. Let $S(w) := w \cup \{w\}$ also for shorthand. Notice $S(\emptyset) = \emptyset \cup \{\emptyset\} = \{\emptyset\}$. Lets denote that as one. Namely $1 := S(0) = S(\emptyset) = \emptyset \cup \{\emptyset\} = \{\emptyset\}$. We may now apply the axiom of extensionality. Is it true $\forall z,\ z \in \{\emptyset\} \iff z \in \{\emptyset\}$? Yes, then this implies that $1 = 1$. Supposedly to prove if $1 + 1 = 2$ took Principia Mathematica three hundred and seventy two pages. This $1 = 1$ example is also from the axioms of ZFC[7] technically. Could PM serve as a suitable foundation? Free of issue? or somewhat incapable?

3 Gödel Incompleteness

Godel showed the futility of Russell and Whitehead's effort. We say an axiomatic system is

1. Complete: if $\forall p$, there exists a proof of p if it's true, or a proof of $\neg p$ if it's false. This asserts provable $\iff$ true.

2. Consistent: $\forall p$ there exists a proof of $p \wedge \neg p$. Every statement is exactly true or exactly false. No statement can be false and true simultaneously.

A system being complete means in some sense, it is "total". From the axioms, all statements are provable. There is no theorem which requires some missing secret axiom. It also asserts if something is true, there must exist a proof of it, and a way to deduce such a proof.

A system being consistent is the bare minimum requirement for it being useful. This asserts you cannot prove $0 = 1$. If you could, then everything follows trivially.

[7] ZFC is a modern conservative axiomatic set theory, which stands for Zermelo-Fraenkel plus the axiom of Choice. We didn't use the axiom of choice here

For years and years, Russell and Whitehead searched for a proof that PM was consistent. Since PM was intended to serve as a foundation of all of mathematics, they were trying to show $PM \vdash Con(PM)$.

3.1 Gödel's First Incompleteness Theorem

Godel show for any axiomatic system, including PM. It cannot be both consistent and complete. Rephrasing: There does not exist a complete and consistent axiomatic system with sufficient arithmetic. Let's proceed with the proof.

Let $Dem(p, r) = 1 \iff p$ is a proof of r[8]. Here, Dem stands for demonstrates. Let

$$g = \neg\exists p \ [\ Dem(p, g) \ ]$$

In human words, g says "I am not provable" or "There does not exist a proof of me." Notice the self-reference. Since by our assumption, our axiomatic system is complete and consistent, g is one of provably true or provably false. We have two cases.

1. g is provably true. Then g asserts there is no proof of g. Then g is true and not provable. Now there exists a true but unprovable statement, so we are incomplete.

2. g is provably false. Then $\neg g$ is provably true.

   $$\neg g = \neg\neg\exists p \ [\ Dem(p, r) \ ] = \exists p \ [\ Dem(p, r) \ ]$$

   So $\neg g$ implies there exists a proof of g, so g is provably true. Then $(\neg g \wedge g)$ is provably true and we are inconsistent.

3.2 Gödel's Second Incompleteness Theorem

Not only does Gödel says that achieving a complete and consistent axiomatic system that is strong enough impossible but any system is capable of proving its own consistency. Logically, for any axiomatic system AS:

$$AS \nvdash Con(AS)$$

The consistency of AS cannot be proven from within AS. Assume to the contrary $AS \vdash Con(AS)$. That there exists a proof of the consistency of AS from within AS. Let this proof be denoted as C. Since the proof of Gödel's first incompleteness theorem assumes the consistency of AS, we may replace this assumption with the proof C. Then we proceed and observe $C \implies g$, our diagonal sentence. Since we can construct g, then AS was not simultaneously consistent and complete, a contradiction. No system is capable of proving its own consistency.

It turns out that some toy systems can be complete and consistent, but they cannot prove their own consistency. You need to use techniques from outside the system to prove

[8]Technically, Gödel encoded proofs, statements, everything as numbers using a Gödel numbering, and then defined these functions based off of those numberings. Pedantically, it should be said like "p is a number which represents a proof of a statement which is represented by the number r". To prove this statement was constructible from simple arithmetic, he had to build it up using some forty formula.

them. If PM was attempting to be a model for all of mathematics, then there there is no "outside" and such a proof of the consistency of PM is unprovable. Not only were the formalists, Russell, Whitehead, and Hilbert losers, they were double losers. The proof they spent decades searching for could never exist.[9]

4 Turing's Undecidable

Alan Turing takes a class foundations, where he learns Gödel Incompleteness. It also contains a description of a large, unsolved problem we will call "Hilbert's decision problem" or the "Entscheidungsproblem". It is phrased as "Give a procedure that takes as input a statement, and returns yes/no if it's always true or always false." Hilbert genuinely believed there were no unsolvable problems. Turing was twenty two when he gave a negative answer. First, he had to formalize the notion of computation, and to do so, he invented what we now call the Turing machine. Next, he described the Church-Turing thesis to convince us that this definition was in fact universal.

Recall that a language $L \subseteq \Sigma^*$ is decidable if there exists a Turing machine M such that

$$x \in L \iff M \text{ accepts } w$$
$$x \notin L \iff M \text{ rejects } w.$$

We can rephrase Hilbert's decision problem as "give a process to decide every language". Following the Church-Turing thesis, the decidable languages give a characterization of the concept of an "algorithm". A purely mechanical process in which a decision on yes or no is reached. If every language is decidable, then there exists an algorithm to solve every problem. There do not exist any unsolvable problems. Could every language be decidable? Every problem be solvable? Turing said no, there exist undecidable languages. He did so in two ways.

First, notice that the languages decidable by a Turing machine are countable. Each language is decidable by many deciders, but one decider decides each language. Let D be the set of all deciders and $\mathscr{L}_D(TM)$ be the decidable languages. We may map each decidable language to exactly one of its deciders to see $|\mathscr{L}_D(TM)| \leq |D|$. By the typewriter principle, we see that $|D|$ is countable. Therefore, so must be the decidable languages.[10] Next observe that the number of languages is uncountable. If $L \subseteq \Sigma^*$ then $L \in \mathcal{P}(\Sigma^*)$. Since $|\Sigma^*|$ is countable, $|\mathcal{P}(\Sigma^*)|$ is uncountable, by Cantor's Theorem. There exists no surjection $\mathscr{L}_D(TM) \to \mathcal{P}(\Sigma^*)$ so there must exist undecidable languages, infinitely many in fact. Most languages are undecidable using this simple nonconstructive, counting proof.

Next, Turing showed constructively that there exist real concrete unsolvable problems. Define $HALT$ as

$$HALT = \{ \langle M, w\rangle \mid M \text{ halts on } w\}.$$

$HALT$ is a language of pairs of encodings of Turing machines and possible inputs, where $\langle M, w\rangle \in \text{HALT} \iff M$ halts on input w. We show that $HALT$ is not decidable. This

[9] See Logicomix 283-286

[10] I previously left this as an exercise for you to show $|\mathscr{L}(NFA)|$ was countable.

means there is no general algorithm to decide if a Turing machine will halt on an input! A provably unsolvable problem.

Assume to the contrary that $HALT$ is decidable. Then there exists a Turing machine $H(\langle M \rangle, w)$ on input $\langle M \rangle$ and w always correctly says yes/no if M halts on w. Notice that since H is a decider, it always returns and never loops. We give a visual proof, representing H like an API or some IDE plugin or something. The middle circuit is the decider for H. We build D around using calls to H, like H is its subroutine. D takes in one argument and passes it to both arguments of H. Then if H returns true, D infinitely loops. If H returns false, then D simply returns and halt.

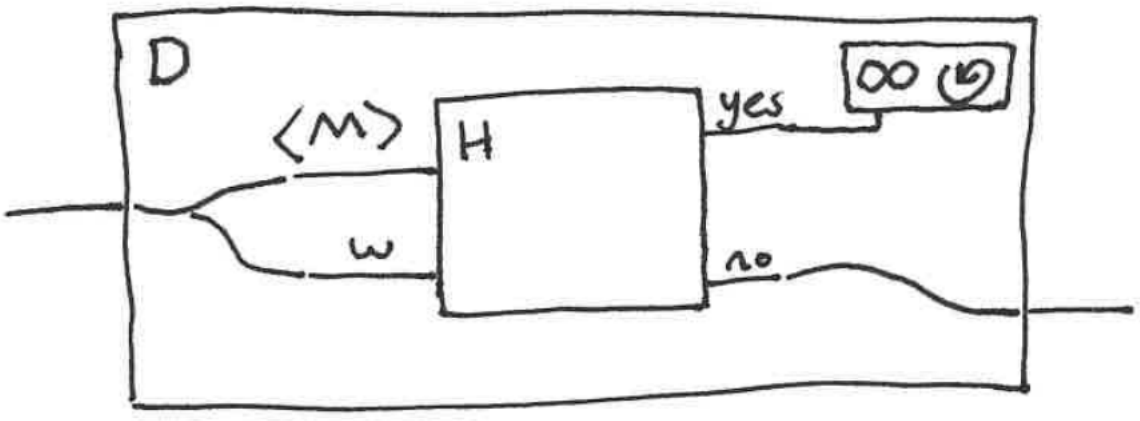


Figure 2: The Halting Problem

In pseudocode, D is doing the following with H:

```
def D(M):
    if H(M,M):
        while True:
            continue
    else:
        return
```

What is D on input $\langle D \rangle$? $D(\langle D \rangle)$? There is no problem with asking this question. We may run the code of a machine on the machine itself with no problem. Compilers can compile themselves. We have two cases.

- $D(\langle D \rangle)$ halts $\iff$ $H(\langle D \rangle, \langle D \rangle)$ returns true $\iff$ $D(\langle D \rangle)$ loops
- $D(\langle D \rangle)$ loops $\iff$ $H(\langle D \rangle, \langle D \rangle)$ returns false $\iff$ $D(\langle D \rangle)$ halts.

A contradiction. No decider for H can exist and we see it is undecidable.

5 Conclusion

We have given three proofs in three different settings. If you have a keen eye, you may note these are really all the same proof. They all have the same structure. They are

all diagonalization over different settings. Sets, logical formulas, and decidable languages. These three proof all have the telltale common features of a proof by diagonalization. There is some negation, and some self-reference, or diagonal. A set not containing itself, a formula saying something about its own unprovability, or a machine contradicting being run on its own code.

6 Moral of History

What is the moral of the history here? We should be incredibly thankful that Hilbert's program failed. Had it succeeded, mathematics would have been drained of all its creativity. There would exist perfect automatic theorem provers. All of mathematics, all of the complex and beautiful technical arguments could be reduced to symbolic manipulation. In Formulario Mathematico, Peano develops a symbolic language for mathematics. He says

> Each professor will be able to adopt this Formulario as a textbook, for it ought to contain all theorems and all methods. His teaching will be reduced to showing how to read the formulas, and to indicating to the students the theorems that he wishes to explain in his course.

Mathematics is an ancient, and didactic, and even dramatic tradition. You sit in front of a board and a lecturer like humanity has for millennia. Reduction of this art to something as mechanical as a combine harvester, reaping theorems, is controversial, putting it politely.

Leibniz, centuries before the foundational crisis in mathematics made a similar remark on this mechanization. He noted that ideas were compounded from some "alphabet of human thought". He also remarked that complex ideas proceed from these by a process analogous to arithmetical multiplication.

> It is obvious that if we could find characters or signs suited for expressing all our thoughts as clearly and as exactly as arithmetic expresses numbers or geometry expresses lines, we could do in all matters insofar as they are subject to reasoning all that we can do in arithmetic and geometry. For all investigations which depend on reasoning would be carried out by transposing these characters and by a species of calculus.

This is Leibniz's motivation to build some of the first mechanical calculators. I am personally thankful that the mechanization of mathematics failed. Otherwise, I would not have this job. Some of Hilbert's program has been salvaged. The consensus is that ZFC forms a safe and conservative foundation for much of the usable parts of mathematics. This is independent of Gödel's theorems, which say that ZFC could never be not both complete or consistent.

A second moral is to not bet against the youth. Russell was 29 when he showed his Paradox. Frege was 53. Russell chose to go down the same path, attempting to build a system that Frege could not. Gödel was 24 when he showed his incompleteness theorems. By then, Russell had aged to 59. Turing was 24 when he proved the existence of unsolvable

problems. Hilbert was 74.[11] It can become easy to become entrenched in your own ideas for decades. All it may take someone younger to come in with a different perspective.

7 Further Reading

- The referenced graphic novel is called Logicomix. A copy from the internet archive can be found here
  https://archive.org/details/Logicomix-Comic-EarlyLifeOfBertrandRussell
- One of the best in depth proofs of Gödel Incompleteness is from https://evoniuk.github.io/Godels-Incompleteness-Theorems/index.html
- Gödel's original proof is not too beyond your ability. A translated copy (the original was in German) may be found in THE UNDECIDABLE, a collection of basic papers edited by Martin Davis
- The construction of the naturals from axiomatic set theory was done following the Von Neumann ordinals. Following our sucessor function $S(w) = w \cup \{w\}$, The first few are
  - $0 : \emptyset$
  - $1 : \{\emptyset\}$
  - $2 : \{\emptyset, \{\emptyset\}\}$
  - $3 : \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\}$
  - $4 : \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}, \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\}\}$

  There are other constructions of the naturals as well, from set theory.
- Lobachevsky is equally remembered for his model of geometry, as he was for accusations and rumors of plagarism. Tom Lehrer wrote a song about it.
  https://www.youtube.com/watch?v=gXlfXirQF3A
- In this worksheet, I go into a slightly more technical proof of Gödel's theorems https://ladha.me/files/sectionX/godel.pdf.
- There is also a video on my youtube channel here
  https://www.youtube.com/watch?v=VpehBGEenWY
- There is a popular science book called Gödel, Escher, Bach which argues that self-reference something something consciousness.

[11] These may be off by a year or two since I don't want to count months.

CS 4510 Automata and Complexity March 13th 2023

Lecture 14: Undecidability by Reduction

Lecturer: Abrahim Ladha *Scribe(s): Abrahim Ladha*

Today we are going to solidify our understanding of what we can know about the unknown. Recall last time we discussed the work of Russell, Gödel, and Turing. We showed there exist unanswerable questions in two ways

- There exist unprovable statements
- There exist unsolvable problems

We proved these using a specialization of the diagonalization technique. Note that nothing is preventing us from stating these unsolvable problems or unprovable statements, only proving or solving them. Today we expand on Turing's work. Our first known undecidable language is

$$HALT = \{ \ \langle M, w \rangle \ | \ M \text{ halts on } w\}$$

1 Some Closure

Recall the definition of decidable and recognizable languages. We say a language L is decidable ($L \in \mathscr{L}_D(TM)$) if there exists a Turing machine M such that

- $w \in L \iff M$ accepts w
- $w \notin L \iff M$ rejects w

We say a L language is recognizable ($L \in \mathscr{L}_R(TM)$) if there exists a Turing machine M such that

- $w \in L \iff M$ accepts w
- $w \notin L \iff M$ rejects or loops on w

Notice that by definition, $\mathscr{L}_D(TM) \subsetneq \mathscr{L}_R(TM)$. We showed that $HALT \notin \mathscr{L}_D(TM)$ but it turns out that $HALT$ is recognizable! Lets give a recognizer. Notice this correctly

Algorithm 1 Recognizer for $HALT$

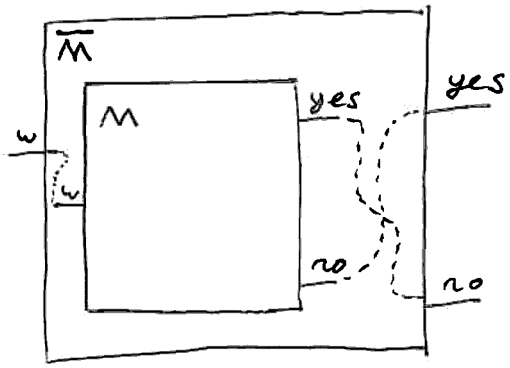
```
on input ⟨M, w⟩
simulate M on w
if M accepts or rejects w then
    accept
end if
```

recognizes $HALT$. If $\langle M, w \rangle \in HALT$ then we know M halts on w, so if we simulate it we halt and correctly accept. But if $\langle M, w \rangle \notin HALT$, then M loops on w and so do we. The

step of simulating M on w does not terminate and we do not reach the conditional. This is then a correct recognizer for $HALT$. Since $HALT$ is recognizable but not decidable, our containment is strict. So Turing proved that not every language is decidable. But is every language recognizable? By a similar counting argument, we know that there are uncountably many languages and only countably many recognizable languages, so most languages are unrecognizable. Is there a notable unrecognizable language, in the same sense that $HALT$ is a notable undecidable language? Lets prove two theorems about closure to show the answer is yes.

1.1 A First Theorem

First we show that the decidable languages are closed under complement. That $L \in \mathscr{L}_D(TM) \iff \overline{L} \in \mathscr{L}_D(TM)$, that the decidable languages are closed under complement. This one is easy. If a language is decidable, then there exists a decider M for it. Since its a decider, it halts on all inputs. Construct a new Turing machine $\overline{M}$ which is just M but we swapped its accept and reject state. Since M was a decider, so is $\overline{M}$, and we see that it decides $\overline{L}$, so it is decidable. We may represent this visually using the following diagram for $\overline{M}$.

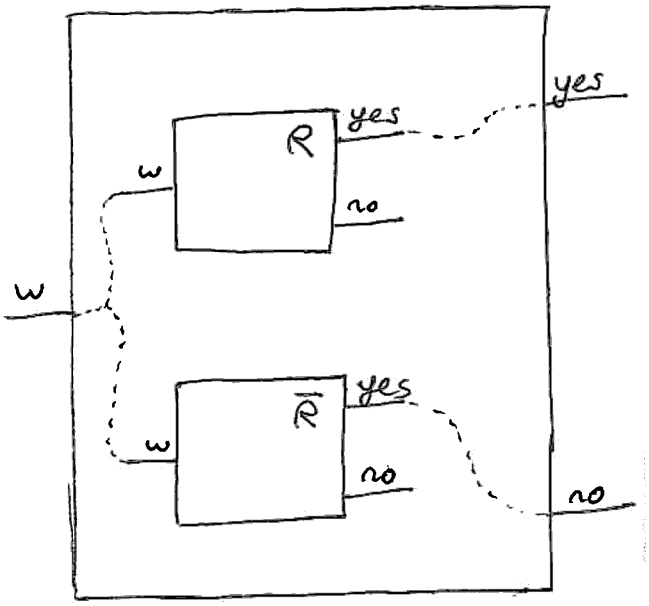


1.2 A Second Theorem

Now lets prove that if $L, \overline{L} \in \mathscr{L}_R(TM) \implies L \in \mathscr{L}_D(TM)$. If a language and its complement are both recognizable, then the language is decidable. Suppose that $L, \overline{L}$ are recognizable with recognizers $R, \overline{R}$. We give a decider for L as follows. A recognizer is not guaranteed to halt on all inputs, but it is guaranteed to halt on the good ones. Let us argue correctness, and why our construction is a decider. If $w \in L$, then by definition R halts and accepts on w, so our machine will halt and accept if $w \in L$. If $w \notin L$, then by definition $\overline{R}$ halts and accepts, so our machine will halt and reject if $w \notin L$. Then for all w, it correctly and exactly decides L so we see that L is decidable. We also now give the equivalent circuit diagram. Of independent interest, this proof shows you how to run two simulations "in parallel". They aren't really in parallel, but rather dovetailed together one at a time. This is necessary. For example, if $w \in L$, this may make $\overline{R}$ loop on w. But R will eventually halt on w. We could not run these simulations sequentially.

Algorithm 2 Decider for L

```
on input w
R = ...
R̄ = ...
while True do
    Simulate R on w for one step
    Simulate R̄ on w for one step
    if R accepts then
        accept
    end if
    if R̄ accepts then
        reject
    end if
end while
```



1.3 An Unrecognizable Language

We now immediately apply the theorems to find a useful unrecognizable language. Assume to the contrary that $\overline{HALT}$ is recognizable. Then since $HALT$ is recognizable, this would imply that $HALT$ is decidable. But we proved by diagonalization its not decidable, a contradiction. Therefore, $\overline{HALT}$ is not recognizable.

Here we have proved existence of an unrecognizable language. The takeaway is that we did not have to use diagonalization. Simply by the fact we can relate problems to one another, we were able to prove this language was also undecidable and unrecognizable. Let us generalize this idea to prove many more languages are undecidable.

2 Many-One Reductions

Recall that a function $f : \Sigma^* \to \Sigma^*$ is said to be computable (or Turing-computable) if there exists a Turing machine M such that for every $w \in \Sigma^*$, M begins with w on the tape and halts with $f(w)$ on the tape. By the Church-Turing Thesis, in some sense, the computable functions are the largest class of functions.

For $A, B \subseteq \Sigma^*$ languages, we say that A is many-one[1] reducible to B (written as $A \leq_m B$) if there exists a computable function f such that $w \in A \iff f(w) \in B$. Notice that this also implies that $w \in \overline{A} \iff f(w) \in \overline{B}$. It maps A to B and $\overline{A}$ to $\overline{B}$. The $\leq_m$ relation satisfies the following few properties. If $A \leq_m B$ then:

- If B is decidable, then A is decidable ($B \in \mathscr{L}_D(TM) \implies A \in \mathscr{L}_D(TM)$)
- If A is undecidable, then B is undecidable ($B \notin \mathscr{L}_D(TM) \implies A \notin \mathscr{L}_D(TM)$)
- If B is recognizable, then A is recognizable ($B \in \mathscr{L}_R(TM) \implies A \in \mathscr{L}_R(TM)$)
- If A is unrecognizable, then B is unrecognizable ($B \notin \mathscr{L}_R(TM) \implies A \notin \mathscr{L}_R(TM)$)

This about how going left is "simpler" and going right is "more unsolvable". The relationship between A, B if $A \leq_m B$ is that A lower bounds B, and B upper bounds A. The four statements we mentioned are intuitive but require proof. We only prove the first one. The other three are proved similarly.

Suppose that $A \leq_m B$ and B is decidable. Then there exists a computable function f as our many-one reduction. We give a decider for A as follows.

Algorithm 3 Decider for A given f as the reduction $A \leq_m B$ and decider for B

```
on input w
compute f(w)          ▷ This computation halts by definition of a computable function
if f(w) ∈ B then      ▷ Since B is decidable, we can run its decider on f(w)
    accept
else                  ▷ If the decider for B rejects, we reject
    reject
end if                ▷ Note this halts on all inputs, so it is a decider
```

[1] or mapping

3 Some Reductions

We now use the method of reduction to show more undecidable problems.

3.1 An Acceptance Problem

Let

$$A_{TM} = \{ \ \langle M, w \rangle \ | \ M \text{ accepts } w \}$$

This language looks close to $HALT$, so it shouldn't be surprising that it is also undecidable. We will prove it is undecidable not by diagonalization, but by reduction. We will show $HALT \leq_m A_{TM}$[2].

Assume to the contrary that A_{TM} is decidable. We give a decider for $HALT$.

Algorithm 4 Decider for $HALT$ given decider for A_{TM}

```
on input ⟨M, w⟩
if ⟨M, w⟩ ∈ A_TM then                ▷ If M accepts w it certainly halts on w
    accept
else                                 ▷ If M doesn't accept on w, it must reject or loop
    build M' from M swapping accept and reject states q_a, q_r
    if ⟨M', w⟩ ∈ A_TM then           ▷ M' accepts w ⟺ M rejects w
        accept                       ▷ If M rejects w, it certainly halts on w
    else                             ▷ If M doesn't accept or reject w, it must loop
        reject
    end if
end if                               ▷ Note this halts on all inputs, so it is a decider
```

Given that A_{TM} was decidable, we were able to construct a decider for $HALT$. But we know by diagonalization that $HALT$ is undecidable, a contradiction. Therefore, we see that A_{TM} is undecidable. Note that A_{TM} is recognizable and $\overline{A_{TM}}$ is unrecognizable for similar reasons to $HALT$ and $\overline{HALT}$.

3.2 An Emptiness Problem

Let

$$E_{TM} = \{ \ \langle M \rangle \ | \ L(M) = \emptyset \ \}$$

This language consists of encodings of Turing machines which accept nothing. We show it is undecidable. The reduction for this one is slightly more advanced. There exists no reduction $A_{TM} \leq_m E_{TM}$[3] but we can do the reduction $A_{TM} \leq_m \overline{E_{TM}}$. By showing the complement $\overline{E_{TM}}$ is undecidable, so must be E_{TM}.

Assume to the contrary E_{TM} is decidable. We give a decider for A_{TM}

Here, our decider constructs a new machine M' with hardcoded M, w. Now notice that M' automatically rejects all strings which aren't w. So $L(M') =$ either $\emptyset$ or $\{w\}$. Which one

[2]Its notable the Sipser book does the reverse, showing A_{TM} is undecidable by diagonalization, and then showing $HALT$ is undecidable by reduction, namely $A_{TM} \leq_m HALT$

[3]See the exercise in the Sipser book

Algorithm 5 Decider for A_{TM} given decider for E_{TM}

```
on input ⟨M, w⟩
construct M' with M, w hardcoded
if ⟨M'⟩ ∈ E_TM then
    reject
else
    accept
end if
```

Algorithm 6 M' hardcoded from M, w

```
on input x
M = ...
w = ...
if x ≠ w then
    reject
else
    Simulate M on input w
    if M accepts w then
        M' accepts x
    end if
end if
```

depends on what happens to M on input w. Our many-one reduction is $f(\langle M, w\rangle) = \langle M'\rangle$ such that

- if $\langle M'\rangle \in E_{TM}$, then $L(M') = \emptyset$. So $w \notin L(M')$ so M must have rejected or looped on w. Either way $\langle M, w\rangle \notin A_{TM}$
- if $\langle M'\rangle \notin E_{TM}$, then $L(M') = \{w\}$. This was only true if M accepted w so we see that $\langle M, w\rangle \in A_{TM}$.

We witness that $\langle M, w\rangle \in A_{TM} \iff \langle M'\rangle \notin E_{TM} \iff \langle M'\rangle \in \overline{E_{TM}}$ so $A_{TM} \leq_m \overline{E_{TM}}$. We conclude that E_{TM} is undecidable.

3.3 An Equivalence Problem

The more languages we prove are undecidable, the easier the next ones become. We have to make the educated decision on which undecidable language to reduce from, but atleast we have the choice. Let

$$EQ_{TM} = \{\ \langle M_1, M_2\rangle \mid L(M_1) = L(M_2)\}$$

This language is the set of encodings of pairs of Turing machines which recognize the same language. We now prove it is undecidable. This reduction is much simpler, and we choose to reduce from E_{TM} to show $E_{TM} \leq_m EQ_{TM}$.

Assume to the contrary EQ_{TM} is decidable. We give a decider for E_{TM}

Algorithm 7 Decider for E_{TM} given decider for EQ_{TM}

on input $\langle M \rangle$
Let $M_\emptyset$ be some hardcoded Turing machine to reject all strings
if $\langle M, M_\emptyset \rangle \in EQ_{TM}$ **then**
    accept
else
    reject
end if

This one is pretty simple. Its logically equivalent to `def iszero(x): return x==0`. Our reduction is $f(\langle M \rangle) = \langle M, M_\emptyset \rangle$. We observe that $\langle M \rangle \in E_{TM} \iff \langle M, M_\emptyset \rangle \in EQ_{TM}$ so $E_{TM} \leq_m EQ_{TM}$ and EQ_{TM} is undecidable.

Of the languages we have shown, this one is the most applicable. Suppose you were given some code and asked to rewrite it in a modern language. How do you know if your rewrite is equivalent? I mean like semantically equivalent. On all inputs, the programs behave identically and equivalently. Since this language is undecidable, there is no algorithm which could take in both pieces of code and definitively say yes or no if they are semantically equivalent. Thats why the best you can do is a thousand unit tests and hope there is no missing case. This can never guarantee they are equivalent, but it can guarantee they may be close enough you couldn't notice a difference if there was one. Specifically for EQ_{TM}, it is a "more unsolvable" problem than the ones we have shown. Lets prove it.

3.4 A "More Unsolvable" Problem

In general for a language $L \subseteq \Sigma^*$ and a class $C \subseteq \mathcal{P}(\Sigma^*)$, we say that $L \in$ co-C if $\overline{L} \in C$. It is not true in general that $\overline{C} =$co-C. We specifically say that a language is co-Turing recognizable (or co-recognizable) if $\overline{L} \in \mathscr{L}_R(TM)$. Note that since $HALT$ was recognizable and not decidable, $\overline{HALT}$ is co-recognizable and not decidable. The languages which are both recognizable and co-recognizable are exactly the decidable languages. This will elucidate our map later on. To show a language isn't recognizable, we may combine the following facts

- If $A \leq_m B$, and if A isn't recognizable, niether is B
- A_{TM} is recognizable and not decidable, $\overline{A_{TM}}$ is co-recognizable, unrecognizable, and not decidable.
- $A \leq_m B \iff \overline{A} \leq_m \overline{B}$

We combine these facts to prove some B is unrecognizable by showing either of the following

$$\overline{A_{TM}} \leq_m B \iff A_{TM} \leq_m \overline{B}$$

We can give a reduction from A_{TM} to the complement of a language, to show a language is unrecognizable. We have shown many undecidable languages, but could they perhaps be recognizable or co-recognizable. Are there any which are neither recognizable nor co-recognizable? Yes, lets prove it. We show EQ_{TM} is neither recognizable nor co-recognizable.

In some sense, this makes it "more unsolvable" than any of the languages we have shown so far. A recognizer for an undecidable language feels atleast half right[4].

First we show EQ_{TM} is not recognizable. Sipser has a more informative reduction to prove this. I found this shorter, cuter proof using only the calculus of reductions, but it is less informative. Recall we proved $A_{TM} \leq_m \overline{E_{TM}}$. This implies $\overline{A_{TM}} \leq_m E_{TM}$. Also recall we proved $E_{TM} \leq_m EQ_{TM}$. Many-one reducibility is a transitive relation (something you should have to prove) so we see that

$$\overline{A_{TM}} \leq_m E_{TM} \leq_m EQ_{TM} \implies \overline{A_{TM}} \leq_m EQ_{TM}$$

Since $\overline{A_{TM}}$ is unrecognizable, so is EQ_{TM}.

Now lets show EQ_{TM} isn't co-recognizable. We equivalently show $\overline{EQ_{TM}}$ isn't recognizable. We would show $\overline{A_{TM}} \leq_m \overline{EQ_{TM}}$ but this is equivalent to $A_{TM} \leq_m EQ_{TM}$. So it suffices to give a reduction from A_{TM} to E_{TM}. We want a computable function $f(\langle M, w\rangle) = \langle M_1, M_2\rangle$ such that

$$\langle M, w\rangle \in A_{TM} \iff \langle M_1, M_2\rangle \in EQ_{TM}$$

Our reduction is as follows.

Algorithm 8 Reduction from A_{TM} to EQ_{TM}

on input $\langle M, w\rangle$
build M_{Σ^*} to accept all strings
build M_2 such that on input x, it runs M on w and accepts x if M accepts w
return $\langle M^*_\Sigma, M_2\rangle$

Notice that $L(M_{\Sigma^*}) = \Sigma^*$ obviously. So

$$\langle M_{\Sigma^*}, M_2\rangle \in EQ_{TM} \iff L(M_2) = \Sigma^* \iff M \text{ accepts } w \iff \langle M, w\rangle \in A_{TM}$$

We conclude that EQ_{TM} is not recognizable or even co-recognizable.

3.5 Language Problems for our Other Computational Models

What is the decidability these language problems relative to our other, weaker automata? Consider the following table. Let D mean decidable and U mean undecidable.

	$A\_$	$E\_$	$EQ\_$	$ALL\_$
DFA, NFA, REGEX	D	D	D	D
CFG, PDA	D	D	U	U
TM, and more	U	U	U	U

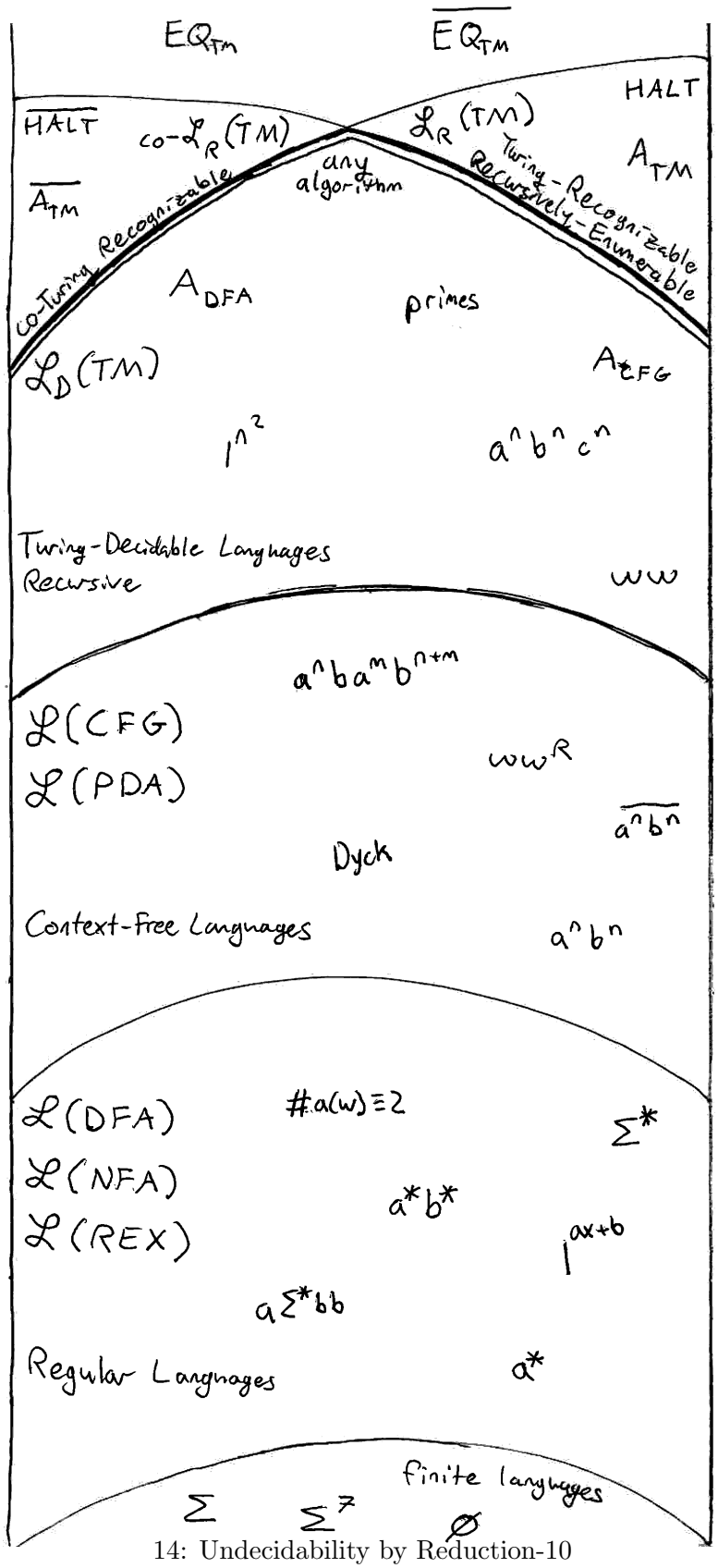
Lets try to give a brief summary of what is an entire chapter of Sipser. Note that if we have proved two kinds of computational models or automata to be equivalent, they should both be decidable or both be undecidable. Otherwise, you could transform from one to the other, and decide it.

[4]some other books even call recognizable languages semi-decidable

- A_{DFA} is decidable, simply run the word on the DFA. Similarly by the equivalence of DFAs, NFAs, and regular expressions, they also all have decidable acceptance problem.
- E_{DFA} is decidable. Treat the DFA like a graph and see if an accept state is reachable from the start state using DFS or BFS or any other graph traversal algorithm. Similarly, ALL_{DFA} is decidable by checking to see if a reject state is reachable from the start state.
- EQ_{DFA} is decidable. We didn't prove it, but there exists an algorithm to convert DFAs into a "normal form" where they are isomorphic (in a vertex and edge colored graph sense) if they decide the same language. A regular language cannot have two different looking DFAs for it and both of them be in this minimal normal form.
- A_{CFG} is decidable. This was the point of Chomsky normal form. There also exists the CYK dynamic programming algorithm to decide this.
- E_{CFG} is decidable. For each non-terminal, you mark it if it is capable of producing strings. You repeat this until you can test if the start non-terminal is capable of producing strings.
- EQ_{CFG}, ALL_{CFG} are both undecidable. This should surprise you. Its certainly seems like a hard problem. While you can tell if a CFG produces any word, or a specific word, how can you decide if a CFG doesn't non-deterministically skip over some word? You can't. Since CFLs aren't closed under complement, E_{CFG}, ALL_{CFG} do not have the same duality like they do for the regular languages. The proof of this undecidability is not beyond you, but it would simply take a lecture. It uses the method of computation histories, which we will go into next lecture. This also implies that for two semantically equivalent grammars, there doesn't exist a normal or minimal form like there is for DFAs. Chomsky normal form isn't then really a "normal form".
- For Turing machines, we proved that A_{TM}, E_{TM}, EQ_{TM} are undecidable. ALL_{TM} is undecidable for similar reasons. This also holds true for any Turing-complete computational model.

The takeaway here is that the more powerful a computer is, the less we can know about the languages they decide, just from looking at their descriptions. Notice that these are all language membership problems. All code problems are decidable. That would include things like "This Turing machine has seventeen states", easily checkable. We don't care about the computers at all. Our true love is the languages. We only use these computational models as a tool to study the classes of languages that they characterize.

It took fourteen lectures, but we finally have enough information to give a full and complete world map. Note that we are quite limited. By the Church-Turing Thesis, anything beyond the decidable languages is incomprehensible. Unfathomable. The recognizable and co-recognizable languages are atleast half fathomable. This may look like a complete map, but we know the world is much bigger than what we can understand. There is an edge to it. The part we can see is only countably many of the languages. A tiny pathetic window into the vast scale of the uncountably large universe of languages.


EQ_{TM}
$\overline{EQ_{TM}}$
HALT
$\overline{HALT}$
$co\text{-}\mathcal{L}_R(TM)$
$\mathcal{L}_R(TM)$
A_{TM}
$\overline{A_{TM}}$
any algorithm
co-Turing Recognizable
Turing-Recognizable
Recursively-Enumerable
A_{DFA}
primes
$\mathcal{L}_D(TM)$
A_{CFG}
1^{n^2}
$a^n b^n c^n$
Turing-Decidable Languages
Recursive
ww
$a^n b a^m b^{n+m}$
$\mathcal{L}(CFG)$
$\mathcal{L}(PDA)$
ww^R
$\overline{a^n b^n}$
Dyck
Context-Free Languages
$a^n b^n$
$\mathcal{L}(DFA)$
$\#a(w) \equiv 2$
Σ^*
$\mathcal{L}(NFA)$
$\mathcal{L}(REX)$
$a^* b^*$
1^{ax+b}
$a\Sigma^* bb$
Regular Languages
a^*
finite languages
Σ
Σ^7
$\emptyset$

CS 4510 Automata and Complexity March 15th 2023

Lecture 15: Post's Correspondence Problem

Lecturer: Abrahim Ladha *Scribe(s): Samina Shiraj Mulani*

1 Introduction

Last time we proved A_{TM}, E_{TM}, EQ_{TM} are undecidable. You may notice these are all problems which are just variations of language acceptance problems. You should be asking the following two questions:

- Are all language acceptance problems undecidable for Turing machines?
- Are the only useful unsolvable problems variations of language acceptance problems?

The answer to the first question is yes. Rice's theorem states that all non-trivial semantic properties of Turing machines are undecidable. This is not really a theorem about Turing machines, rather it is about the recognizable languages. But we can really only study these languages through the lens of Turing machines. A property is non-trivial if not every Turing machine has or hasn't the property. For example, the property "M is a Turing machine which is a Turing machine" is trivial. You can show a property to be non-trivial by giving one Turing machine with the property, and one without. A property is semantic if its about the language instead of the encoding itself. A syntactic property is about the encoding of the machine. For example, "M has 17 states". Easily decidable, count the states. A semantic property might be "M recognizes a language which has some (maybe different) Turing machine to recognize the same language with 17 states". Syntactic properties are about the encodings. Semantic properties are about the languages. Intuitively, a semantic property requires somehow knowing something about the execution of the machine without simulating it.

The answer to the second question is no. The point of today's lecture is only to show you that there exists an unsolvable puzzle. The problem statement has nothing to do with Turing machines. The existence of algorithmically unsolvable problems is not as conditional as it feels on the Church-Turing Thesis. There do exist unsolvable problems with nothing to do with language theory. Here, we give a puzzle with no algorithmic solution. It is provably unsolvable.

2 Problem Statement

Let a "domino" or "tile" be a pair of strings, consisting of an upper and lower portion. For example, a set of tiles could be

$$\left\{\left[\frac{b}{ca}\right], \left[\frac{a}{ab}\right], \left[\frac{ca}{a}\right], \left[\frac{abc}{c}\right]\right\}$$

We say a set has a "match", if given unlimited copies of each tile, there exists a sequence (possibly with repetition) where the concatenations of the top equal the concatentations of the bottom. For example, given the previous set of tiles, consider the sequence 2,1,3,2,4.

$$\left[\frac{a}{ab}\right]\left[\frac{b}{ca}\right]\left[\frac{ca}{a}\right]\left[\frac{a}{ab}\right]\left[\frac{abc}{c}\right]$$

- The top elements concatenated are $a \cdot b \cdot ca \cdot a \cdot abc = abcaaaabc$
- The bottom elements concatenated $= ab \cdot ca \cdot a \cdot a \cdot abc = abcaaaabc$

So this set has a match.

We prove that PCP is algorithmically unsolvable. There is no algorithm given a set of tiles to determine if there is a match or not. Restated as decidability of a language

$$PCP = \{\ \langle P \rangle\ \mid P \text{ is a set of tiles with a match } \} \text{ is undecidable.}$$

The proof idea is simple but has lots of small details. First, lets explore its universality in some way.

3 Proof Idea

3.1 Forcing a start

First note we can set up a set of tiles such that we can force any decision making procedure to temper its behavior a certain way. For example, for the following set of tiles, the first (and last) choices are fixed. Any procedure is tempered into picking the first tile first.

$$\left\{\left[\frac{\#b}{\#}\right], \left[\frac{a}{b}\right], \left[\frac{\$}{a\$}\right]\right\}$$

It is the only tile where the top and bottom begin with the same symbol. Similarly the last tile for any match of this set (if it exists) is also forced.

3.2 Forcing a next tile with deficiency

For any decision making procedure, we can force it so that the *next* tile has to begin the way we want it. Note that a decision making procedure need not make selections of tiles sequentially. There is a lot of creative things algorithms can do. But if a certain tile set has a match, then the nth tile must have the desired property we will force. Consider the set

$$\left\{\left[\frac{\#a}{\#}\right], \left[\frac{a}{a}\right]\right\}$$

Suppose it was forced to choose the first tile.[1] Now the "working strings" of the top and the bottom are $\#a$ and a respectively. Since the top is longer than the bottom, the next

[1] Forget for a moment that the second tile by itself is a match for this set. We will show a way around this later.

tile is forced to have its bottom begin with an a. That means we can only choose tiles of the form $\frac{\cdots}{a\cdots}$.

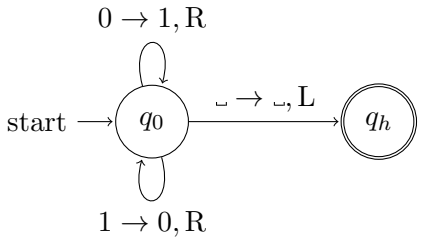
Also notice this deficiency is never satisfied. A decision making procedure will be forced to choose tiles ad infinitum. The working strings will always be $\#a^{k+1}$ and $\#a^k$. This idea, intuitively can encode a Turing machine which loops.

Using these ideas, we can encode the transition function of a Turing machine into a set of tiles. With the right setup, we can ensure that the tile instance only has a match if M accepts w. We will force the first tile, then force each of the next tiles to behave according to our Turing machine transition function. Then we will ensure there is a "cap piece" to match the deficiency only if M accepts w.

4 Proof of Unsolvability

4.1 Computation History

A computation history is a sequence of configurations in some string encoding made useful. Here, we will construct a set of tiles such that its only string match is this computation history. for example, the following is a computation history for the following machine.



$\#q_0 10\#0q_0 0\#01q_0\text{␣}\#0q_h 1\text{␣}\#$

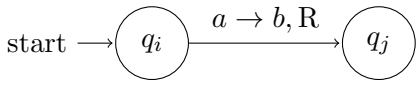
We may define an accepting computation history to be a computation history, where the last configuration is an accepting one. Notice that an accepting computation history is just a string which only exists if M accepts w. If M loops on w, such a computation history would be infinite in length, and then not a string. If M rejected w, such a computation history would end with a rejecting configuration instead of an accepting one. This is the heart of the method of accepting computation histories. We will use the fact that this string only exists if M accepts w, and we will create a set of tiles such that the only match is the accepting computation history. Then the set of tiles only has a match if there exists an accepting computation history, which only exists if M accepts w.

4.2 Construction

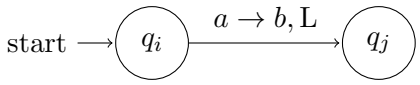
We begin our tile with this starting one.

$$\left[\frac{\#}{\#q_0 w_1 w_2 ... w_n \#}\right]$$

Notice that the next tile is forced to begin with q_0 at the top. We will only add two[2] such tiles, one with q_0a and the other with q_0b, so that only one gets picked to match to q_0w_1.



Given a right transition in our machine, our configurations would change like $q_ia \to bq_i$. So we emulate this in our tiles. We add one tile per right move transition. If we have transition $\delta(q_i, a) = (q_j, b, R)$, we add tile $\left[\frac{q_ia}{bq_j}\right]$



Of course, we must also simulate left moves, so if $\delta(q_i, a) = (q_j, b, L)$, our configurations would change looking like $cq_ia \to q_icb$. We add one domino per selection of c. Suppose $\Gamma = \{a, b, \sqcup\}$. We need one for each possible left move of our machine. Note that we added one tile per right move, but three tiles per left move. This imbalance is just an artifact of the way we encode a snapshot of the state of the machine as a string. $\left[\frac{aq_ia}{q_jab}\right], \left[\frac{bq_ia}{q_jbb}\right], \left[\frac{\sqcup q_ia}{q_j\sqcup b}\right]$

I hope you see the pattern here. We have created a set of tiles such that the decisions made to create a match are forced to simulate the Turing machine according to its transition function. The first tile creates a deficiency on the top. As the next sequence of tiles are forced fix this deficiency. As they do, they compute the next configuration and append it to the bottom!

$\left[\frac{a}{a}\right], \left[\frac{b}{b}\right], \left[\frac{\sqcup}{\sqcup}\right]$

We need some more tiles to make sure everything is set up. We add one singleton tile (shown on the left) $\forall a \in \Gamma$ to make copies of the rest of the tape for us. Recall in a sequence of configurations, only a small local part of each sequential configuration changes. Most of the tape remains unchanged. These tiles are for performing this copying for us.

[2]Technically $|\Gamma|$ for a well defined transition function

$\left[\frac{\#}{\#}\right], \left[\frac{\#}{\sqcup\#}\right]$

We also need a cap between configurations and a way to use more space. Recall a configuration can have more blanks ($\sqcup$) like leading zeroes because the tape is infinite. We only choose to write as many as necessary, one. If we want more, it will have to be done for the next configuration.

$\left[\frac{aq_a}{q_a}\right], \left[\frac{bq_a}{q_a}\right], \left[\frac{\sqcup q_a}{q_a}\right]$

The accept state being q_a, we add the following tiles. This basically has the q_a "eat" the rest of the tape. This is so our cap fits nicely. Recall that on accepting/rejecting/halting, the tape head may end where ever. This creates a slight amount of complexity for us, so we use this to simplify. We do not need to have the q_a eat right if we modify the machine to loop all the way to the right just before accepting.

$\left[\frac{q_a\#\#}{\#}\right]$

Once you reach the accept state in a Turing machine, you halt. However, our match will keep going to clean up the tape only so we can insert nice end cap. This completes the match. Note we have no end cap for rejection. This cap can only be placed if M accepts w.

You may have noticed we add tiles to not enforce the rule of a single start, like $\left[\frac{a}{a}\right]$ or $\left[\frac{\#}{\#}\right]$. We now modify all our tiles to enforce the start we want is the actual start. Given a set of dominoes we modify them in the following way. For $u = u_1...u_n$, let

$\bullet u = \bullet u_1 \bullet u_2 \bullet ... \bullet u_n$

$u\bullet = u_1 \bullet u_2 \bullet ... \bullet u_n\bullet$

$\bullet u\bullet = \bullet u_1 \bullet u_2 \bullet ... \bullet u_n\bullet$

Let $\left[\frac{t_s}{b_s}\right]$ be the start tile, $\left[\frac{t_e}{b_e}\right]$ be the end tile.

Given our set of tiles -

$$\left\{\left[\frac{t_s}{b_s}\right]\left[\frac{t_1}{b_1}\right]...\left[\frac{t_k}{b_k}\right]\left[\frac{t_e}{b_e}\right]\right\}$$

We modify them like -

$$\left\{ \left[\frac{\bullet t_s}{\bullet b_s \bullet}\right] \left[\frac{\bullet t_1}{b_1 \bullet}\right] ... \left[\frac{\bullet t_k}{b_k \bullet}\right] \left[\frac{\bullet t_e \bullet}{b_e \bullet}\right] \right\}$$

This can be generalized to make our reduction more like

$$A_{TM} \leq_m MPCP \leq_m PCP$$

but this correctly makes the start and end tiles for out match exactly the ones we want. It does come at the cost of using more symbols, and our match being twice as long. It is as if we skipped over every other cell of the tape. Our final set of tiles is then

$\left[\frac{\bullet \# }{\bullet \# \bullet q_0 \bullet w_1 \bullet w_2 \bullet ... \bullet w_n \bullet \# \bullet}\right]$	One start tile
$\left[\frac{\bullet q_i \bullet a}{b \bullet q_j \bullet}\right]$	For each right move transition like $\delta(q_i, a) = (q_j, b, L)$ we add one tile
$\left[\frac{\bullet a \bullet q_i \bullet a}{q_j \bullet a \bullet b \bullet}\right], \left[\frac{\bullet b \bullet q_i \bullet a}{q_j \bullet b \bullet b \bullet}\right], \left[\frac{\bullet \sqcup \bullet q_i \bullet a}{q_j \bullet \sqcup \bullet b \bullet}\right]$	For each left move transition like $\delta(q_i, a) = (q_j, b, L)$, we add $\lvert\Gamma\rvert$ tiles
$\left[\frac{\bullet a}{a \bullet}\right], \left[\frac{\bullet b}{b \bullet}\right], \left[\frac{\bullet \sqcup}{\sqcup \bullet}\right]$	$\lvert\Gamma\rvert$ tiles for copying
$\left[\frac{\bullet \# }{\# \bullet}\right], \left[\frac{\bullet \# }{\sqcup \bullet \# \bullet}\right]$	two extra tiles to help between configurations
$\left[\frac{\bullet a \bullet q_a}{q_a \bullet}\right], \left[\frac{\bullet b \bullet q_a}{q_a \bullet}\right], \left[\frac{\bullet \sqcup \bullet q_a}{q_a \bullet}\right]$	either $\lvert\Gamma\rvert$ or $2\lvert\Gamma\rvert$ tiles for eating
$\left[\frac{\bullet q_a \bullet \# \bullet \# \bullet}{\# \bullet}\right]$	One end tile

Lets stress why the computation is correct. We begin like:

$\left[\frac{\#}{\# C_0}\right]$	Then we are forced to add tiles in which the tops match C_0. By doing so, we have chosen the bottom to compute and place C_1
$\left[\frac{\# C_0 \#}{\# C_0 \# C_1}\right]$	Now, we must repeat, matching C_1 to force us to compute and place C_2.

$$\left[\frac{\#C_0\#C_1\#}{\#C_0\#C_1\#C_2}\right] \qquad \text{And so on.}$$

The only way we can match is if we fix the deficiency, and the only way to do that is to place the end tile. We can only place the end tile if M accepts w. The match for our set of tiles exists if and only if there is an accepting computation of M on w. We had no reject end tile. If the machine loops, this computation history would be infinite and so there would be no match. We see that our construction $f(\langle M, w\rangle) = \langle P\rangle$ is correct. Namely

$$\langle M, w\rangle \in A_{TM} \iff \langle P\rangle \in PCP$$

So we conclude PCP is undecidable.

For any kind of structure, we can note if there are enough degrees of freedom for us to simulate the transition function of a Turing machine, but perhaps not too many to make its problems too easy, any such structure will have unsolvable questions. This goes far beyond computational questions. There are unsolvable problems in combinatorics, geometry, topology, and more. Now that we have shown a simple combinatorial problem which is unsolvable, we can use this in further reductions.

5 Baba is You

Now we show Baba is You is undecidable. If we suppose that $BABA$ was solvable, that is, given any Baba is You level, there exists an algorithm to determine if it is winnable or not, we claim then you could solve PCP, a problem we just proved unsolvable. Our reduction would be added on like

$$A_{TM} \leq_m MPCP \leq_m PCP \leq_m BABA$$

The proof idea is given a set of tiles, to construct a Baba is You level which is winnable if and only if the tile set has a match. A reappearing theme is that the intuition is clear, even if the necessary gadgets are very complex.

- The paper: https://arxiv.org/abs/2205.00127
- The videos: https://www.youtube.com/playlist?list=PLE75TLHOnaOKrQsrhCUgOmuAX7l7dI66N
- The Baba is You level editor has online play where you can play other custom levels. https://hempuli.itch.io/baba-is-you-level-editor-beta

CS 4510 Automata and Complexity 3/27/2023

Lecture 16: Kolmogorov Complexity

Lecturer: Abrahim Ladha *Scribe(s): Michael Wechsler*

1 Introduction

Consider the following two strings:

1111111111 1101111101

The first string can be described simply. It has a relatively short description of just its length. The second string is less simple. Maybe we couldn't call it complex, necessarily, but it is certainly less simple. If you were to describe it, your description would also have to include information about the location of the two zeroes.

A measure is a function $\mu : \{\text{things}\} \to \mathbb{R}^+$ (or $\mathbb{N}$), where $\mu(x) = 0 \implies x$ has nothing, or none of "it".If $\mu(x) > \mu(y)$, then x has more of "it" than y. Examples of measures include length, area, volume, for their respective objects. Cardinality is a measure on sets.

2 Definition

We want to construct a measure on strings for their "algorithmic complexity." Given a string, how hard is it to describe? Is it simple or complex? How much information does the string communicate? Can we even measure this? We can try. What is a "description" anyway? Lets follow our intuition towards a formalization. A program is a description! This leads us to an intuitive definition. Define $K : \Sigma^* \to \mathbb{N}$ to be the Kolmogorov Complexity of a string.

$$K(x) = \text{the length of the shortest program to print } x \text{ and halt.}$$

Mathematically, this could be represented as

$$K(x) = \min_{p \in \Pi} \Big(|p| : U(p, \varepsilon) = x \Big) \tag{1}$$

x : the string
p : a program
Π : set of all programs
$|p|$: length of program (like a string)

U : universal simulator (runs p on ε)
p : program
ε : takes no input
x : prints x and halts; output x

2.1 Invariance of the Definition

Why did we say a program and not a Turing Machine? It is like asymptotic analysis in the theory of algorithms. Our complexity measure is independent of the language it is written in. Unlike the theory of algorithms, rather than rely on our intuition, we can prove this independence.

Consider the language specific definitions for Python and Rust, named $K_{\texttt{py}}$ and $K_{\texttt{rust}}$. By the Church-Turing Thesis, since Rust is Turing-Complete, we can certainly write a Python interpreter in Rust. Let this program written in Rust to interpret Python be called $\pi_{\texttt{pyinrust}}$. Now, given any Python program, combined with this interpreter in Rust, we just have a Rust program. It might look something roughly like

```
fn interpret(python_code: &str) {
...
}
fn main() {
    let pyprog: &str = "#!/usr/bin/python3\ndef f()\n\t...";
    interpret(pyprog)
}
```

Suppose there is a python program *p.py* with $|p.py| = K_{\texttt{py}}(x)$. This minimal Python program can be used to create a Rust program, as seen above. We observe:

$$K_{\texttt{rust}}(x) \leq K_{\texttt{py}}(x) + |\pi_{\texttt{pyinrust}}| \tag{2}$$

We can only say $\leq$ and not $=$ since we do not know if there exists a smaller Rust program. But the existence of this Rust program which prints x upper bounds $K_{\texttt{rust}}(x)$. Notice, by the Church-Turing Thesis, that Python is also Turing-Complete. Thus, we can also write a Rust interpreter in Python.[1] By a symmetrical argument, there exists a Rust interpreter in Python named $\pi_{\texttt{rustinpy}}$ and we observe:

$$K_{\texttt{py}}(x) \leq K_{\texttt{rust}}(x) + |\pi_{\texttt{rustinpy}}| \tag{3}$$

Notice that our interpreters are independent of anything about x, like its complexity or length. These interpreters are of constant size. You could have a python program of a billion gigabytes, and the code to interpret the python program would remain a few megabyte or whatever. Next, notice our two inequalities are symmetric. For any two $a, b \in \mathbb{N}$, if $a \leq b + O(1)$ and $b \leq a + O(1)$, then $|a - b| \leq O(1)$. We combine our inequalities this way to get that there exists a constant c such that

$$\forall x \ |K_{\texttt{py}}(x) - K_{\texttt{rust}}(x)| \leq c \tag{4}$$

So, the difference between our two algorithmic complexities only differs by some constant. This can obviously be generalized for all Turing-complete programming languages. We may drop the subscript and just consider $K(x)$ rightfully as a universal definition.

[1] A Rust interpreter in Python seems far less useful than a Python interpreter in Rust. Yet, you should imagine that someone could write such a program. For computability, we do not care about the difference between compiled and interpreted. A compiled language is really a translation into the language of machine instructions, which is then arguably just interpreted by the CPU. There do exist interpreters for traditionally compiled languages, like C. This distinction is unimportant. Arbitrary.

3 Examples

Here are some examples to understand Kolmogorov complexity better.

3.1 $K(x)$

Notice that for any $x \in \Sigma^*$, there exists a program to print it. Somewhat obviously, just have the program contain the string hardcoded. Such a program may look like.

```
def f():
    x = '.......................'
    print(x)
```

This program takes no input and prints x, for any $x \in \Sigma^*$. So we observe that

$$\forall x \; K(x) \leq |x| + c \tag{5}$$

where c is some constant independent of the input. For example, |"`print()`"| $= 7$, so $c \geq 7$. It is independent of the input, but dependent on the programming language, which we don't care about.

3.2 $K(xx)$

What about the Kolmogorov Complexity of some string concatenated with itself. What is $K(xx)$ in terms of x? A bad idea is to hardcode xx.

```
def badidea():
    xx = '.............................................................'
    print(xx)
```

Rather than an upper bound of $K(xx) \leq |xx| + c = 2|x| + c$, we can construct a program to only store x instead of xx and then compute xx from x.

```
def goodidea():
    x = '........................'
    print(x.append(x))
```

This gives us a better upper bound of $K(xx) \leq |x| + c'$. Note that $c' > c$ since our program needs the logic to compute xx from x. It's still a constant though. For future reference, all constants are not necessarily equal, but all are independent of the input.

3.3 $K(x^n)$

What about many concatenations, like $K(x^n)$ for some n?

```
def f():
    x = '........................'
    n = .....
    ans = ''
    for i in range(n): ans.append(x)
    print(ans)
```

The size of our program as a function of the input is $|x|$ and $|n| = \log n$. So

$$K(x^n) \leq |x| + \log n + c \tag{6}$$

Before, we didn't need $\log n$ as it was only constantly many concatenations. Now we must keep a counter. Also notice we need not hardcode x. What if there was a much shorter way to compute x? Let g be some minimal program which takes no input and returns[2] x. Maybe its size is much smaller than the size of x ($|g| << |x|$).

```
def f():
    x = g()
    n = .....
    ans = ''
    for i in range(n): ans.append(x)
    print(ans)
```

Thus, $K(x^n) \leq K(x) + \log n + c$. We replaced the hardcoded x with a computation of x.

3.4 $K(x^R)$

What about x^R, the reversal of string x? What is $K(x^R)$ in relation to $K(x)$? If some program p prints x, we can create a program q to print x^R. Note q is like p. It computes x, but instead of immediately printing it, it computes x, reverses it, then prints it. We observe this reversal operation is independent of the input, so $|q| = |p|+c$. Thus, $|K(x)-K(x^R)| \leq c$ for some constant c. This also underlines our intuition of K being a measure of natural descriptive complexity. If a string is complex or simple, its reversal should remain complex or simple. Our intuition on simple or complex strings is invariant to reversing.

4 Compression

Let's get back to some intuition about randomness. Some strings appear to have very short, simple descriptions, like 1^{2^n}. A program to print this string needs to only really contain information about n, which is much smaller than the length of the string. Others strings appear to have long descriptions, or at least no short descriptions. We may say a string is incompressible if $K(x) \geq |x| - c$ for some c. The shortest description of an incompressible string isn't much shorter than the string itself. We may say a string is compressible if it is not incompressible. How many strings of length n are compressible by 2 bits? Just two measly bits. Let's compute it as a ratio:

$$\frac{\text{strings of length } n \text{ compressible by two bits}}{\text{all strings of length } n} = \frac{|\{x \in \Sigma^n \mid K(x) \leq |x| - 2\}|}{|\Sigma^n|} \leq \tag{7}$$

$$\frac{|\{p \in \Pi \mid \text{p a program with } |p| \leq n-2\}|}{2^n} \leq \frac{\bigcup_{i=0}^{n-2} \Sigma^i}{2^n} \leq \frac{\sum_{i=0}^{n-2} 2^i}{2^n} = \frac{2^{n-1}}{2^n} = \frac{1}{2} \tag{8}$$

[2]the difference between returning and printing is an engineering issue, and we don't care about the difference enough. Obviously if there is a program to return a string, there is a similarly sized program to print the string.

HALF?! Only half of the strings of length n are compressible by 2 bits. This generalizes so only $\frac{1}{4}$ are compressible by 3 bits and $\frac{1}{2^{d-1}}$ are compressible by d bits. This is a very lazy upper bound[3], but it's still sufficient to show us that most strings are incompressible. Less than $\frac{1}{1000}$ strings are incompressible by 11 bits.

The stress is on "most" strings. We have found a deep connection between randomness and information content. A uniformly random string has overwhelming probability to be incompressible. The compressible strings are the *lucky* ones. If you have files many many gigabytes in size, only 1/1000 of them are compressible by a byte or more.

Why does file compression work in practice? Consider some fixed setting, like images of fixed dimension. Most images look like TV static. By most we mean in a uniformly random sense, the color of each pixel being drawn according to a coin, you will generate an image which looks like garbage. In contrast, most of the "useful" images generated by humans are full of patterns for our pattern matching brain. A picture of a parrot may have a large splotch of red. Lossy encodings like JPEG and lossless algorithms like Lempel-Ziv exploit these patterns to generate short descriptions.

Back in the world of strings, the compressible strings are the lucky ones. If you were to generate the bits of a string by a random coin flip, its going to have overwhelming probability of having near equal number of zeroes and ones. With negligible, insignificant probability would it have any exploitable pattern or structure. How likely is the string xx or 1^n as an output of this part of this random process? Most strings are incompressible because most strings do not have any pattern.

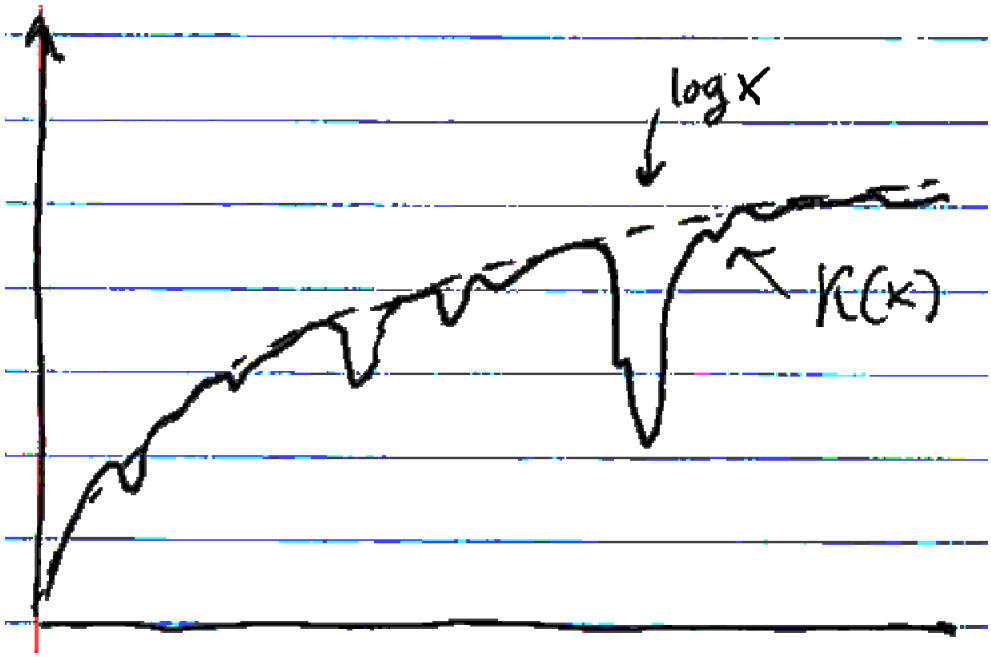
Heres a deep remark. Although since we believe $\mathsf{P} \neq \mathsf{NP}$, we are unable to computationally distinguish random strings from those produced by a pseudo-random generator. Yet if an arbitrarily long random string is incompressible, arbitrarily long pseudo-random strings all have short descriptions. Those descriptions being simply the algorithm of the pseudo-random generator, the seed, and the string's length.

5 Graph of $K(x)$

Lets try to plot $K(x)$, but instead of $K : \Sigma^* \to \mathbb{N}$, consider $K : \mathbb{N} \to \mathbb{N}$. We witness the following behavior of K.

- $K(x)$ grows unbounded. $\nexists c \ \forall x \ K(x) < c$. To prove this, consider n such that $K(1^n) > c$. Note that n can get really big, but c cannot. The function must be growing.
- $K(x)$ "hugs" $\log x$. We proved most strings are incrompressible, so the graph should hover near $\log x$ for most x.
- $K(x)$ dips infinitely often. A small program for one string implies an infinite family of small programs for an infinite family of strings. We showed that $K(x) \approx K(x^R)$. A description of a string being simple or complex does not depend on the direction we read it. This same intuition can be used to see that $K(x) \approx K(2x) \approx K(3x) \approx K(2^x) \approx K(2^{2^x}) \approx K(x + \sqrt{x})$ and so on.

[3] it's much larger than actual amount. We were extremely generous with our overestimation and still concluded a very small upper bound.



- $K(x)$ has continuous properties. Recall the definition of continuity of a real valued function. We say f is continuous if when $x, x+\varepsilon$ are close, so are $f(x), f(x+\varepsilon)$. This means $|x-x_0| < c_1 \implies |f(x)-f(x_0)| < c_2$. In terms of $K(x)$, $\forall x$, $|K(x)-K(x\pm 1)| < c$. Take the program that prints x. Modify it to add 1, now you have a program to print $x+1$. Note that $K(x)$ cannot actually be continuous, as it is discretely valued. But it may not have a sporadic behavior if you were to plot it like a line.

A plot of $K(x)$ with the following properties might look like the figure.

6 $K(x)$ is not computable

We have a imagined[4] graph of $K(x)$, but we never gave an algorithm. That's because there is none. $K(x)$ is not a computable function. We will prove this with diagonalization. Assume to the contrary $K(x)$ is computable, and there is a program which may compute it. Then we may construct the following algorithm.

Algorithm 1 M

```
on input w
for x ∈ Σ* lexographically do
    if K(x) > |w| then
        print x
        halt
    end if
end for
```

[4] I traced this from the Li Vitanyi book

On input w, it searches for the smallest lexographic string with Kolmogorov Complexity greater than the length of w. The for loop iterates like $x \in \{\varepsilon, 0, 1, 00, 01, \ldots\}$. What is M on input $\langle M \rangle$ or $M(\langle M \rangle)$? As the algorithm proceeds, $M(\langle M \rangle)$ will search for some smallest string x such that $K(x) > |\langle M \rangle|$ and print it. But since M itself prints this x, we see that $K(x) \leq |\langle M \rangle|$. A contradiction. It is impossible for both $a > b$ and $a \leq b$ to both be true. Thus, $K(x)$ is not computable.

Note this proof is a little rough, but perhaps you get the picture. Kolmogorov complexity is defined for machines which take no input, but here we allow M to take input. The way around this would be to encode M somehow with its own size. This is possible but requires a little math. You couldn't just compute the size, then hardcode it in, as this would then change the size. There also exists something called Kleene's recursion theorem, which allows a program to obtain a copy of its own description, and compute with it. I didn't want to get into these details.

7 The Method of Incompressibility

This is a useful proof technique derived from Kolmogorov Complexity. Like how the pigeonhole principle shows existence of an object with some desired property, the method of incompressibility shows most objects have some desired property. Here, we mean "most", truly in a Kolmogorov-random sense. It is one of the strongest techniques we have for average case and worst case lower bounds. The proofs usually follow some similar structure. You assume something to the contrary, and then show that this implies some succinct description of an incompressible object.

7.1 Infinitude of the Primes

There are infinitely many primes. There is a classic proof due to Euclid you may know. There are many other proofs of this result as well. Here, we will prove it using the method of incompressibility.

Suppose there are only finitely many primes, $p_1, \ldots, p_m$. Then $\forall n \in \mathbb{N},\ \exists e_1, \ldots, e_m$ such that $n = p_1^{e_1} \cdots p_m^{e_m}$. Thus, $\langle e_1, \ldots, e_m \rangle$ is a description of n The program to print n would have hardcoded $e_1, ..., e_k$. It would bruteforce recompute all the primes, and then compute $n = {p_1}^{e_1} \cdot ... \cdot p_m^{e_m}$ and print it. Note that each e_i can be described in $\log e_i$ bits, so $K(n) \leq \log e_1 + \cdots + \log e_m$. A worse case is that n is a prime power, so if $n = p_i^{e_i}$ for some i, then $e_i = \log_{p_i} n$. It follows then that each $e_i \leq \log n$.

$$K(n) \leq \log e_1 + \cdots + \log e_m \leq \log\log n + \cdots \log\log n \leq m \log\log n \tag{9}$$

where m is independent of the input. So, $\forall n,\ K(n) = O(\log\log n)$. For any incompressible n, we have a contradiction.

We showed that if there were finitely many primes, then every number could be described succinctly by its prime powers. But we know most numbers are incompressible. We concluded that $\forall n,\ K(n) = O(\log\log n)$, but we know for most n that $K(n) = \Omega(\log n)$.

7.2 Proving non-regularity of languages

Let me give you a tool I am calling this the "extremely weak KC-regularity lemma". The book by Li and Vitanyi has two generalizations of this. I have tried to simplify it enough just to demonstrate it in part of a lecture. Tto show you some application of Kolmogorov complexity to something you already know.

Assume L is regular. Let $xy \in L$ where $|xy|$ is some function of n, and y is the minimal string such that $xy \in L$ ($xy' \in L$ with larger y' may exist). Then $K(y) = O(1)$.

7.2.1 Proof

If L is regular, there exists a DFA, D, for it. Run x on D to get to some state q_i. Since y is the minimal suffix of xy ($\nexists y'$ for which $xy' \in L$ and $|y'| < |y|$), y is the minimal string to bring D from state q_i to an accept state. Then, D, q_i, and this discussion are a unique description of string y. Thus, $K(y) \leq |D| + |q_i| + c$. Recall the F in DFA stands for finite, so $K(y) \leq |D| + |q_i| + c = O(1)$.

7.2.2 How to Use the Lemma

We will use the extremely weak lemma to prove languages are not regular. Follow this recipe to apply the lemma

1. Assume to the contrary L is regular
2. Choose some $xy \in L$ with $|xy|$ is a function of some n. You just can't choose xy to be constant, and technically you are choosing an infinite family of strings instead of just one. It may make more sense in the following examples.
3. Choose x. Compute y from xy and x, ensure that the desired y is minimal and also a function of n. Do not choose xy and x such that $|y| = O(1)$.
4. Apply the lemma to get $K(y)$ is $O(1)$
5. But note that as the complexity of y should grow as something greater than a constant.
6. Reach a contradiction, and conclude.

7.2.3 $\{a^n b^n \mid n \in \mathbb{N}\}$

We will write this proof out in the explicit steps.

1. Assume to the contrary $L = \{a^n b^n \mid n \in \mathbb{N}\}$ is regular
2. Choose $xy = a^n b^n$
3. Choose $x = a^n$. Thus, $y = b^n$ and is minimal
4. By the lemma, $K(b^n) = O(1)$
5. But note that this is false for large enough n. The complexity of the string b^n will grow as a function of n.
6. We have found a contradiction and L is therefore not regular.

7.2.4 $\{1^{n^2} \mid n \in \mathbb{N}\}$

The point of using the Kolmogorov complexity instead of pumping is so we get fast, terse, correct proofs of non-regularity. We will do the following proofs in this spirit.

Assume to the contrary $L = \{1^{n^2} \mid n \in \mathbb{N}\}$ is regular. Choose $xy = 1^{(n+1)^2}$ and $x = 1^{n^2}$. Thus, $y = 1^{(n+1)^2 - n^2} = 1^{2n+1}$ and is minimal. By the lemma, $K(1^{2n+1}) = O(1)$. But n may get arbitrarily large, a contradiction.

See how fast that proof was?

7.2.5 $\{ww^R \mid w \in \Sigma^*\}$

Assume to the contrary $L = \{ww^R \mid w \in \Sigma^*\}$ is regular. Choose $xy = (ab)^n(ba)^n$ and $x = (ab)^n$. Thus, $y = (ba)^n$ and is minimal. By the lemma, $K((ba)^n) = O(1)$, but $K((ba)^n)$ grows as a function of n, a contradiction.

These proofs looks easy, because they hide quite a bit of the mechanics, and some things can go wrong. For example, if you chose $xy = a^n a^n$ and $x = a^n$, then a minimal y would not be $y = a^n$, but it would be $y = a$ or aa. These are not a function of some n, and we would not reach a contradiction. Like pumping, choosing a good string (in this case, xy) is important. The stronger version of this lemma is more difficult but makes this issue clearer.

7.2.6 $\{1^p \mid p$ **is prime** $\}$

Assume to the contrary $L = \{1^p \mid p \text{ is prime}\}$ is regular. Choose $xy = 1^p$, where p is the $(k+1)^{th}$ prime. Choose $x = 1^{p'}$, where p' is the k^{th} prime. Thus, $y = 1^{p-p'}$ is minimal. By the lemma, $K(1^{p-p'}) = O(1)$, but the difference between primes grows unbounded, a contradiction.

8 A Hint Towards Computational Learning Theory

Suppose we loosened our definition of $K(x)$ so that the programs to print x need not be perfect, only approximate. Recall, our definition that $K(x) =$ "the length of the shortest program to print string x". Suppose the following synonym substitutions were made:

- length → size
- shortest → simplest
- program → description
- prints → approximates
- string → dataset

Now, we have $K(x) =$ "the size of the simplest description which approximates dataset x". That sounds a lot like Occam's Razor. Following this logic, you could formalize Occam's Razor under PAC[5] learning. In practice, since $K(x)$ is not computable, there is much more success with computable restrictions, such as

[5] Probably Approximate Correct

$$K^t(x) = \min_{p \in \Pi} \{|p| : U(p, \varepsilon) = x \text{ and halts in } t(|x|) \text{ steps}\} \tag{10}$$

9 Further Reading

You have to look at the Li Vitanyi book. Chapter two may guide you towards understanding more about the complexity itself. Chapter six will give you may applications of the method of incompressibility. These include Turing machine simulation lower bounds, average case complexity of heapsort, Hastad's switching lemma for circuit lower bounds, and much more. Chapter eight has some connections between Kolmogorov complexity and information theory. I also recommend you read 6.4 of the Sipser book and maybe this old worksheet of mine https://ladha.me/files/sectionX/kolmogorov.pdf

CS 4510 Automata and Complexity 3/29/2023

Lecture 17: Complexity Classes

Lecturer: Abrahim Ladha *Scribe(s): Akshay Kulkarni*

1 Introduction

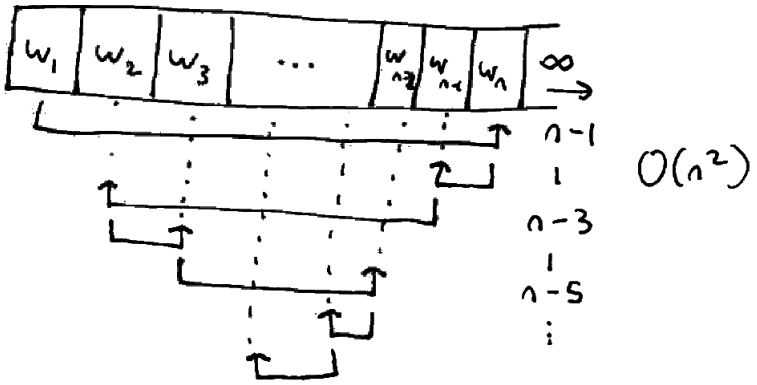
1.1 Motivation

We begin our final unit entirely on computational complexity. This lecture will simply consist of some early and motivating theorems, mostly before the development of NP-completeness.

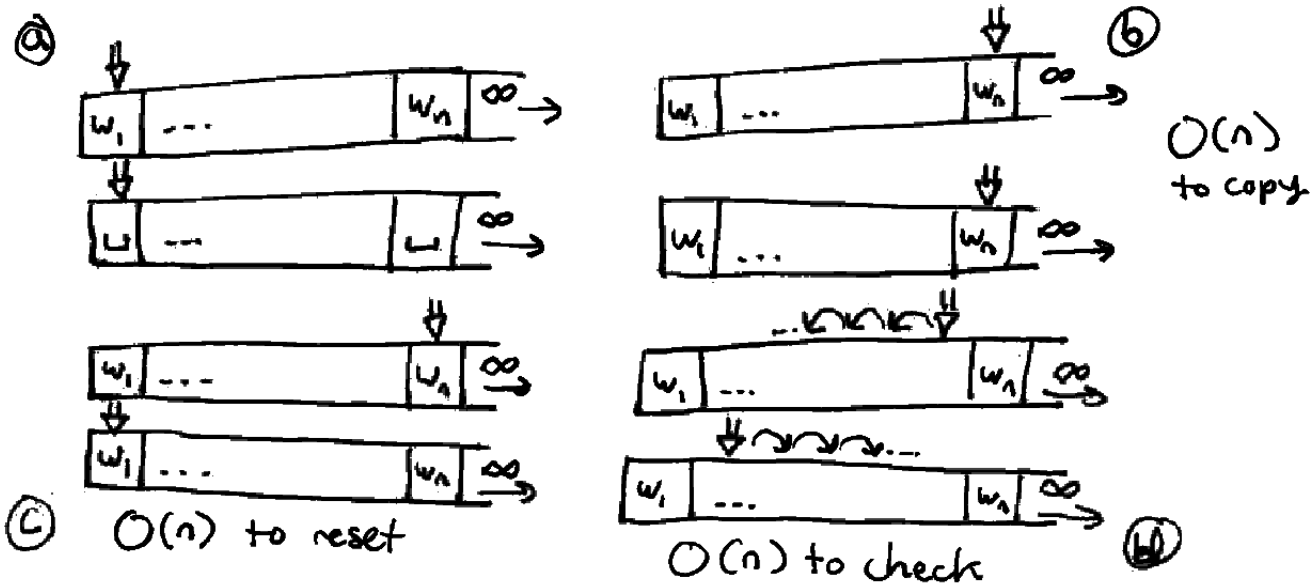
First we need a good computational model of a "hard" or "easy" computation. There is a reason I have been beating you with Turing machines. Turing machines make an excellent model for complexity. Recall that a Turing machine performs a constant amount of work in unit time. If more work is to be done, successive steps must be taken. This is exactly what makes it an excellent model, because this is exactly how algorithms work in reality There did exist historically some functional but Turing-complete models of computation, and they do not have this property. For example, there exists a lambda calculi for string copying. It can produce xx from x in a single step. It might look like $\lambda x[xx]$. This isn't as good of a model, as to copy an arbitrarily long string should take some number of steps as a function of the length of the string. Longer strings should take longer to copy. It is not clear here what the "step" is in a functional model, but it certainly clear what a step is for a Turing machine.

1.2 Turing Machine Variants

Does the variant and choice of Turing machine matter?. For now, we set aside the non-deterministic Turing machine (NTM) and only consider reasonable and realizable models. Consider the language of palindromes $PAL = \{ww^R \mid w \in \Sigma^*\}$. There is an algorithm to decide PAL on a single-tape deterministic Turing machine (DTM). Check the first symbol, then the nth symbol, then the second symbol, and so on. The limitation of this machine is that it is not random access. To read the last symbol starting from the first takes a linear number of steps because the tape head has to loop over the entire input. To decide PAL, this takes $n + (1) + (n-2) + 1 + \cdots = O(n^2)$ by Gauss's trick.



We can give a better algorithm on a two-tape DTM as follows. Copy the input to a second tape, reset one tape head. Loop both heads in opposite directions on the tape, comparing symbols. These three steps each take linear time giving a $O(n)$ time algorithm on our two tape DTM.



Obviously, any stronger model must also take at least linear time, as to decide if $ww^R \in PAL$ must look at all the symbols for correctness[1]. Can a single-tape DTM decide PAL in $O(n)$ or even $o(n^2)$ time? Surprisingly, no. any single tape deterministic Turing machine to decide PAL must take $\Omega(n^2)$ (and so $\Theta(n^2)$) steps. This is surprising! It means on a one tape DTM, there is no way to do better for this language than the obvious way. There are

[1]Most algorithms should take linear time. If an algorithm takes sublinear time, it doesn't even have time to look at the entire input, so it could only compute some toy language like a regular one. For example checking if the first symbol is a one takes constant time. Binary search is sublinear, but this is only on the random access model, not on a Turing machine. Still, binary search is efficient because it doesn't have to look at the entire input.

two proofs, a more classic combinatorial one, and one which uses Kolmogorov complexity. Essentially, if a machine could decide PAL in $o(n^2)$ time, you could use this machine to compress an incompressible string. I recommend this proof as your final project.

2 P

Let TIME($f(n)$) := be the class of languages decidable by a Turing machine in $f(n)$ steps. Similarly define NTIME($f(n)$), SPACE(($f(n)$), NSPACE($f(n)$). Let

$$\mathsf{P} = \bigcup_{k=0}^{\infty} \text{TIME}(n^k)$$

Why is P a good definition of the class of intuitively efficient algorithms? We give some arguments in favor

1. Most problems seem to have naive, trivial, brute-force solutions putting the problems atleast in EXP. If there exists a polynomial time algorithm, then either the problem is trivial, ridiculous, or we have some deeper intuition about what the problem actually is (i.e. mathematical theory). An exponential time algorithm may be simple, but a polynomial time algorithm usually requires a non-trivial understanding of the structure of the problem itself. For example consider graph traversal. A bad way would be to enumerate all paths and check them this way. A better way is to notice how any sub-path of a path is itself a path. The shortest path from s to t cannot be longer than the shortest path from s to t through some v. This recurrent structure is what leads to efficient algorithms like DFS/BFS/Dijkstra's and so on.

2. Polynomials are closed under operations which our intuition of "efficient" is also closed under. If you have two algorithms A, B. If one or both is inefficient, then the composition, running both of them sequentially, should intuitively be inefficient. If both are efficient, then running both sequentially should be efficient. If $f(x), g(x)$ are polynomials, then $f(x) + g(x), f(x)g(x), f(g(x))$ are also polynomials. A combination of efficient algorithms should be efficient, and a combination of efficient and inefficient algorithms should be inefficient. Polynomials preserve this.

3. Although there exist languages with $O(n^{100})$ algorithms which require $\Omega(n^{99})$ steps, we don't have any practical examples of this. The highest polynomial run time you would see in an algorithms course might be cubic time $O(n^3)$[2]. The highest polynomial run time I know is for the LLL algorithm. In $O(n^8)$ time, it finds a short orthonormal basis of a lattice. Its poly-time, but practically infeasible. It would appear to stall on reasonably small inputs. However, the achievement of the authors was that they were able to bring the problem from far beyond P into P. Once you get a polytime algorithm at all, it seems likely in practice that it can be improved. There were several papers on "pruning" which make the algorithm more efficient in a way thats

[2]Like chain matrix multiplication. Maybe Bellman-Ford on a dense graph. Most things appeared to be linear or quadratic time.

hard to asymptotically measure. The engineers will get a hold of the problem and make it practicality efficient, even optimizing the constants. Although the vanilla algorithm may not appear to complete in reasonable time on most inputs, the version of the algorithm which is included in most libraries, it appears to halt instantly on all inputs you could test. This is one example, but the case is evident for many other algorithms. If there is enough intuition to give a polynomial time algorithm at all, its likely this intuition can be extended and the problem can be made easier and easier. The languages which are solvable in $\Omega(n^{99})$ are nonconstructive, useless. They aren't real practical problems, and are designed via diagonalization to have this property, and do nothing else.

4. All Turing machine variants appear to simulate each other with at most polynomial overhead. What a word-RAM machine does is T steps takes a one-tape DTM T^4 steps. The word-RAM model is our best computer, and the one-tape DTM might be our worst, yet the overhead is still only a polynomial. Although within P, they may take different time for different languages, A definition of P is equivalent for all these models. The extended Church-Turing Thesis says that not only are all these variants as powerful as each other. The run-time of any one model to simulate another will not have super polynomial overhead.

3 NP

Let

$$\mathsf{NP} = \bigcup_{k=0}^{\infty} \text{NTIME}(n^k)$$

The definition given in your algorithms course is that NP is the class of languages verifiable in polynomial time. We now prove these definitions are equivalent. Let NP be the class of languages decidable by a non-deterministic Turing machine which halts in a polynomial number of steps. We say a language $A \in \mathsf{NP}_v$ if there exists a deterministic polynomial time verifier V for A. The verifier will take as input a word w and a witness or certificate c and $V(w, c)$ will accept or reject accordingly if $w \in A$.

Let $A \in \mathsf{NP}_v$, then there exists a polynomial time verifier V, which runs in $O(n^k)$ time for some k. We will build a NTM to decide A as follows.

Algorithm 1 N on input w

Nondeterministically guess certificate c of max length n^k
run $V(w, c)$
accept $\iff$ V accepts

Clearly, N runs in polynomial time. Since this is for all languages verifiable in polynomial time, we see $\mathsf{NP}_v \subseteq \mathsf{NP}$.

Let $A \in \mathsf{NP}$, then there exists a polynomial time NTM to decide A. We show A is a polynomial-time verifiable. Our witness c is just our nondeterministic choices. So, V is a deterministic polynonial-time verifier correctly for A, so $\mathsf{NP} \subseteq \mathsf{NP}_v$. Since we proved the

Algorithm 2 V on input $\langle w, c \rangle$

```
Simulate N deterministically on w
if faced with a nondeterministic choice then
    Get the next bit of c
end if
if current branch of N's computation accepts then
    accept
end if
```

containment both ways, we see that $\mathsf{NP} = \mathsf{NP}_v$, and may drop the subscript. From now on when we talk about NP, we can use either definition based on conveineince.

4 $\mathsf{P} \subseteq \mathsf{NP}$

We prove $\mathsf{P} \subseteq \mathsf{NP}$ in both ways, using both definitions of NP.

1. First, note that by the generalization of nondeterminism, every deterministic polynomial-time Turing machine is also a nondeterministic Turing machine, so $\mathsf{P} \subseteq \mathsf{NP}$
2. If $A \in \mathsf{P}$, then there exists a polynomial time algorithm to decide A. Ee prove A is also verifiable in polynomial time. The verifier will simply ignore the witness and simulate the polytime decider for A. This implies $\mathsf{P} \subseteq \mathsf{NP}$.

5 More Classes

$$\mathsf{PSPACE} = \bigcup_{k=0}^{\infty} \mathrm{SPACE}(n^k)$$

$$\mathsf{L} = \mathrm{SPACE}(\log(n))$$

We give some rough ideas about why the following chain holds.

$$\mathsf{L} \subseteq \mathsf{NL} \subseteq \mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PSPACE} \subseteq \mathsf{NPSPACE} \subseteq \mathsf{EXP} \subseteq \mathsf{NEXP} \subseteq \mathsf{EXPSPACE} \subseteq \mathsf{NEXPSPACE}$$

- $\mathsf{L} \subseteq \mathsf{NL}$ follows from the generalization of non-determinism.
- $\mathsf{NL} \subseteq \mathsf{P}$ follows from the fact there exists a polytime algorithm for an NL-complete problem.
- $\mathsf{P} \subseteq \mathsf{NP}$ as proved previously.
- $\mathsf{NP} \subseteq \mathsf{PSPACE}$, since $\textsc{sat} \in \mathrm{SPACE}(n)$. Recall that $\forall L \in \mathsf{NP}$, $L \leq_p \mathrm{SAT}$.
- Later we will prove that $\mathsf{PSPACE} = \mathsf{NPSPACE}$, but this containment as shown follows obviously

Note that PSPACE contains languages we think are decidable only in exponential time, so we won't discuss too much on the right half of this chain.

6 More Questions than Answers

We have absolutely no idea how to solve $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$. We do understand to some level how hard the problem[3] actually is. The problem itself is connected via a massive web of implications to many more problems.

Consider the following two subchains.

1. $$\mathsf{L} \subseteq \mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PSPACE}$$

   We can prove $\mathsf{L} \subsetneq \mathsf{PSPACE}$. Since these are the left and right sides, one of the containments in this chain must be strict. Note that if you could prove $\mathsf{P} = \mathsf{PSPACE}$, this would imply that $\mathsf{P} = \mathsf{NP}$, so if $\mathsf{P} = \mathsf{PSPACE}$ is an open problem. If you could prove that $\mathsf{L} = \mathsf{P}$ and $\mathsf{NP} = \mathsf{PSPACE}$, then it must be the case that $\mathsf{P} \neq \mathsf{NP}$. These are open problems as well.

2. $$\mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{EXP}$$

   Note that $\mathsf{NP} \subseteq \mathsf{EXP}$ since you can give a deterministic exponential time algorithm to brute force all polynomial sized certificates. We can prove $\mathsf{P} \subsetneq \mathsf{EXP}$, so again there exists a strict containment in this chain. Proving $\mathsf{NP} = \mathsf{EXP}$ would imply $\mathsf{P} \neq \mathsf{NP}$, so $\mathsf{NP} = \mathsf{EXP}$ is also an open problem.

The history of complexity theory is a history of failure. Any problem which could be reasonably asked may accidentally imply something about $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$ and thus becomes as hard as the problem itself. The failure to solve this one problem has shifted major directions in research over the past fifty years. Every new research direction, every new theorem, every new foundation has been built with the motivation and direction to solve this one problem. A massive effort has been undertaken in order to try and solve the problem with zero success. We have built the shoulders of giants. Ironically, as we have understood the problem better, we are farther away from solving the problem than we were when we began.

Besides these structural connections, there are many more ways which might resolve the question. These include

- Proving existence or nonexistence of a one-way function
- Showing a super polynomial lower bound or a polynomial time algorithm for any individual NP-complete problem.
- Giving a polytime algorithm to convert 3SAT instances into 2SAT ones.
- Proving that random generators are indistinguishable from psuedorandom ones.
- A proof that every property expressible by a second order existential statement is also expressible in first order logic with a least fixed point operator
- There is no polynomially bounded propositional proof system.

And much much much more.

[3] Whenever I may refer in passing to "the problem", I could only refer to the one problem, this problem. $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$.

CS 4510 Automata and Complexity 4/3/2023

Lecture 18: In and around NP

Lecturer: Abrahim Ladha *Scribe(s): Michael Wechsler*

This lecture could also be titled "The Cook-Levin Theorem and Ladner's Theorem".

1 Reductions

What's the point of intractability and NP-completeness?[1] Suppose you are given a task, and asked to produce an algorithm for some problem, but you can't. So, you then try to prove the problem is intractable or unsolvable. This is pretty hard to do in practice, but it can happen. More likely, you can prove the problem was NP-complete. That elevates it to a special club: a class of problems all as hard as each other. A fast algorithm for one would imply a fast algorithm for all, and that $\mathsf{P} = \mathsf{NP}$. There are thousands of such problems across many domains.

We say for two languages $A, B \subseteq \Sigma^*$ that A is polytime reducible to B ($A \leq_p B$) if $\exists$ a function, f, which is computable (i.e. halts on all input) in polytime with

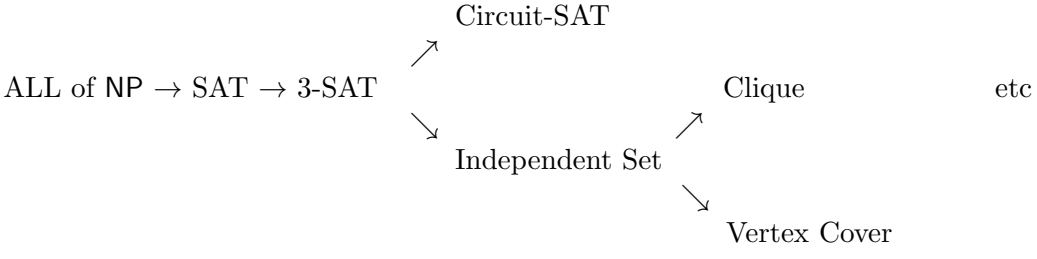
$$w \in A \iff f(w) \in B$$

If $A \leq_p B$, A lower bounds B and B upper bounds A. It is analogous to many-one reductions ($\leq_m$), which are computable, but these are computable in polytime. $\leq_m$ can be used to prove problems are as solvable/unsolvable as each other, while $\leq_p$ can be used to prove problems are as easy or hard as each other.

2 NP-Completeness

We say language B is NP-complete if $B \in \mathsf{NP}$ and $\forall L \in \mathsf{NP}$ that $L \leq_p B$. You may also prove a language to be NP-complete much easier by using the transitivity of the $\leq_p$ relation. Choose some known NP-complete language, A, and prove $B \in \mathsf{NP}$ and $A \leq_p B$. Cook and Levin proved for us independently that SAT is NP-complete, which means $\forall L \in \mathsf{NP}$, $L \leq_p$ SAT.

By the web of reductions, we have thousands of other NP-complete problems.



[1] see the attached drawings of Garey and Johnson

This chain can go on and on, and contains cycles.[2] These reductions only work if there is known NP-complete problem, which we are going to prove. A reduction is just a tranformation from one language to one language which preserves correctness. The Cook-Levin Theorem proves for every language in NP, there is a reduction to SAT. We will do it generically, for any language in NP.

3 SAT

Recall the definition of SAT:

- Variable: one of $x_1,\ x_2,\ \ldots,\ x_l$
- Literal: one of $x_1,\ x_2,\ \ldots,\ x_l,\ \neg x_1,\ \neg x_2,\ \ldots,\ \neg x_l$. A variable or its negation.
- Clause: an OR of literals, such as $(x_1 \vee \neg x_2 \vee x_3)$
- CNF Formula: an AND of clauses, such as $(x_1 \vee x_2) \wedge (\neg x_3 \vee x_1)$
- Assignment: a selection of $x_1,\ x_2,\ \ldots,\ x_l \in \{0,1\}$. We say an assignment is satisfying if when you plug in $x_1,\ x_2,\ \ldots,\ x_l$ into a CNF formula that $\phi = 1$.

$$\text{SAT} = \{\langle\phi\rangle \mid \phi \text{ is a satisfiable CNF}\}$$

CNFs are surprisingly expressive. Usually in real-world constraint problems, you have a set of constraints and must satisfy all of them, but there may be more than one way to satisfy each. For example,

$$(x_1 \vee \neg y_1) \wedge (\neg x_1 \vee y_1) \wedge \ldots \wedge (x_n \vee \neg y_n) \wedge (\neg x_n \vee y_n)$$

is true if and only if $x_1 = y_1,\ \ldots,\ x_n = y_n$. $x = x_1 \ldots x_n$ and $y = y_1 \ldots y_n \Rightarrow x = y$. This formula is satisfiable if and only if $x = y$. This is a CNF for string equality.

4 Cook-Levin Theorem

We want to prove that $\forall L \in \text{NP},\ L \leq_p \text{SAT}$.

Obviously SAT $\in$ NP since there is a verifier, $V(\phi, c)$, that checks if c is a satisfying assignment to ϕ in polytime.

Let $L \in \text{NP}$. Then there exists a nondeterministic polytime machine, N, such that N accepts $w \iff w \in L$. Consider the computation history for N accepting w. We use this to construct a CNF formula, Φ, such that $w \in L \iff \Phi \in \text{SAT}$. Φ will be satisfiable if and only if N accepted w. The idea is conceptually simple but has many of little details. Since $L \in \text{NP}$, N is at most polytime, say n^t, and is then also polyspace, say n^s. Take the sequence of configurations of an accepting computation history $C_0,\ C_1, \ldots$ and line them up in a table\tableau like so

[2] I hope you recall some of these reductions from the unit in your algorithms course. You should have done many reductions, but all were conditional on the assumption that SAT was NP-complete. Here, we finally prove it.

#	q_0	1	1	1	␣	#
#	0	q_0	1	1	␣	#
#	0	0	q_0	1	␣	#
#	0	0	0	q_0	␣	#
#	0	0	0	␣	q_a	#

Note: There are n^t rows and $n^s + 2$ish columns

The dimension of the tableau is time by space $= n^t \times n^s$, making it polynomial sized. We will create a CNF formula, Φ, to loop over the table and check its correctness.

$$\Phi = \Phi_{cell} \wedge \Phi_{start} \wedge \Phi_{move} \wedge \Phi_{accept}$$

- $\Phi_{cell} = 1 \iff$ exactly one symbol is in each cell of the table
- $\Phi_{start} = 1 \iff$ first row is the initial configuration
- $\Phi_{move} = 1 \iff$ the $i + 1^{th}$ row is the $C_{i+1^{th}}$ configuration, following the i^{th} row
- $\Phi_{accept} = 1 \iff$ there is an accepting configuration in the table

Let $x_{i,j,s}$ be the variables with $1 \leq i \leq n^t,\ 1 \leq j \leq n^s,\ s \in Q \cup \Gamma \cup \{\#\}$ where $x_{i,j,s} = 1 \iff$ cell $[i, j] = s$. $x_{i,j,s}$ means symbol s is in cell $[i, j]$. Let $C = Q \cup \Gamma \cup \{\#\}$.

- $\Phi_{cell} = 1 \iff$ exactly one symbol is in each cell of the table. For example, we want a relationship like $x_{i,j,a} = 1 \implies x_{i,j,b} = 0$. We can do this by ANDing clauses which make sure atleast one symbol is on for each cell, and no more than one is on for each cell. This ensures each element of the table has exactly one symbol.

$$\Phi_{cell} = \bigwedge_{\substack{1 \leq i \leq n^t \\ 1 \leq j \leq n^s}} \Big[\Big(\bigvee_{s \in C} x_{i,j,s} \Big) \bigwedge \Big(\bigwedge_{\substack{s,t \in C \\ s \neq t}} (\overline{x_{i,j,s}} \vee \overline{x_{i,j,t}}) \Big) \Big]$$

$\bigwedge_{\substack{1 \leq i \leq n^t \\ 1 \leq j \leq n^s}}$: double for loop over the entire two dimensional table

$\bigvee_{s \in C}$: guarantees at least one symbol is in each cell

$\bigwedge_{\substack{s,t \in C \\ s \neq t}}$: guarantees no more than one symbol is in each cell

We are essentially using the syntax of SAT to write a program to check correctness of our table. Its not essential to understand how this "programming language" works. Its more important to understand that this was even possible.

- $\Phi_{start} = 1 \iff$ first row is the initial configuration of N on w with appropriate space.

  $$\Phi_{start} = x_{1,1,\#} \wedge x_{1,2,q_0} \wedge x_{1,3,w_1} \wedge \ldots \wedge x_{1,n+2,w_n} \wedge x_{1,n+3,\sqcup} \wedge \ldots \wedge x_{1,n^s-1,\sqcup} \wedge x_{1,n^s,\#}$$

  To satisfy Φ_{start}, the corresponding table must have the first row of our desired configuration

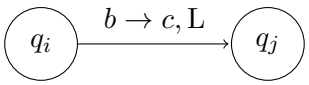
- $\Phi_{accept} = 1 \iff$ there is an accepting configuration in the table

  $$\Phi_{accept} = \bigvee_{\substack{1 \le i \le n^t \\ 1 \le j \le n^s}} x_{i,j,q_a}$$

  Loop over the entire table to make sure q_a symbol exists somewhere

- $\Phi_{move} = 1 \iff$ the $i+1^{th}$ row is the $C_{i+1^{th}}$ configuration, following the i^{th} row

  Φ_{move} is the hardest one. We want it to enforce that each row follows the preceding one by only legal moves according to the transition function δ of N. With the first initial configuration enforced, we want Φ_{move} to enforce row two is the second configuration and so on. The way Φ_{move} will work is check every 2×3 window of the table and determine if it's a legal 2×3 window. For example, for transitions like

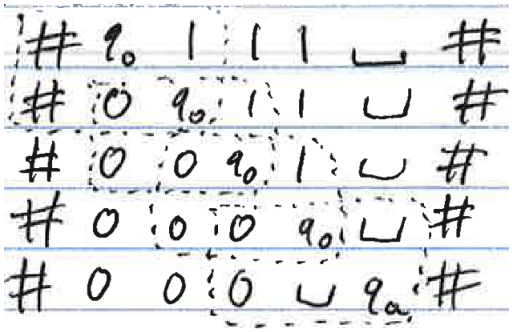
  

a	q_i	b
q_j	a	c

  this is a legal window. There are other legal 2×3 windows like

a	b	c
a	b	c

a	a	q_i
a	a	b

  For this table, I have dotted a few windows near the head. If the whole table is legal, the windows near the head are the only ones which aren't copy windows. Convince yourself that the $i+1^{th}$ row follows from the i^{th} row if and only if every 2×3 window is legal. These are equivalent conditions.

$$\Phi_{move} = \bigwedge_{\substack{1 \le i \le n^t \\ 1 \le j \le n^s}} [\text{the } (i,j) \text{ window is legal}]$$

By legal, we mean according to δ of N. This whole proof is similar to the PCP proof.

$$\Phi_{move} = \bigwedge_{\substack{1 \le i \le n^t \\ 1 \le j \le n^s}} [\bigvee_{\substack{a_1,\ldots,a_6 \\ \text{is legal}}} (x_{i,j-1,a_1} \wedge \ldots \wedge x_{i+1,j+1,a_6})]$$

$\bigwedge_{\substack{1 \le i \le n^t \\ 1 \le j \le n^s}}$: double for-loop over two dimensional table, checking all 2×3 windows

$\bigvee_{\substack{a_1,\ldots,a_6 \\ \text{is legal}}} (x_{i,j-1,a_1} \wedge \ldots \wedge x_{i+1,j+1,a_6})$: checks if window i, j is legal

We finish by construction of

$$\Phi = \Phi_{cell} \wedge \Phi_{start} \wedge \Phi_{move} \wedge \Phi_{accept}$$

Note: Φ is satisfiable only if:

1. Each cell of the table contains exactly one symbol
2. The first row is a start configuration
3. The $(i+1)^{th}$ row is the $C_{i+1^{th}}$ configuration following the i^{th} row as the C_i^{th} configuration
4. One of the configurations is accepting

So Φ is satisfiable only if an accepting computations history of N on w exists, which can only happen if there is a computation of N on w so $w \in L \iff \Phi \in \text{SAT}$

Now we argue that this reduction takes polytime to compute. Note that for a polynomial sized table, each of the subformulas also took polynomial time to construct so the computation to build Φ takes polytime. We construct Φ by just a few for-loops. We observe $L \le_p$ SAT. Since SAT $\in$ NP and $\forall L \in$ NP, $L \le_p$ SAT, we conclude SAT is NP-complete.

4.1 Importance of this Finding

Now that we have proven SAT is NP-complete, we may prove many other languages are NP-complete, not by repeating the proof, but by a simple reduction. For example, if you prove 3SAT $\in$ NP and SAT $\le_p$ 3SAT, then since we proved $\forall L \in$ NP that $L \le_p$ SAT, we can use transitivity. $L \le_p$ SAT $\le_p$ 3SAT $\implies L \le_p$ 3SAT. The reduction reuses and transforms the proof, rather than redoing it. SAT is not the only language that could be proved as a genesis NP-complete problem. Sipser and CRLS both include a proof by a similar construction that CircuitSAT is NP-complete. Levin originally proved a kind of tiling problem. Cook proved not SAT necessarily, but tautologies are NP-complete.

SAT $\in$ P $\implies$ P = NP. To prove, recall $\forall L \in$ NP that $L \leq_p$ SAT. So if SAT $\in$ P, then there is a polytime algorithm for SAT. Since every $L \in$ NP is polytime reducible to SAT, combining this reduction plus the polytime algorithm for SAT is a polytime algorithm for L. So $L \in$ P, but since L is any language in NP, we see NP $\subseteq$ P. Since we know P $\subseteq$ NP, we conclude P = NP. We do not believe SAT has a polytime algorithm. We don't even believe SAT has a quasi-polytime algorithm.

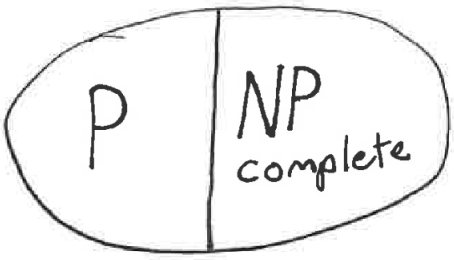
5 Ladner's Theorem

Not all languages in NP $\setminus$ P are NP-complete if P $\neq$ NP. We will prove it shortly. Factoring is a candidate for an NP-intermediate problem. It is (believed) not to be in P, but has sub-exponential time algorithms. The general number field sieve has a runtime for factoring $\in TIME(o((1+\varepsilon)^n))$ $\forall \varepsilon > 0$. We believe that factoring cannot be in P, otherwise, cryptography doesn't exist. We say a function is quasi-polynomial if it is super polynomial, yet subexponential. Such functions do exists, and algorithms exists with quasi-polynomial run-time.

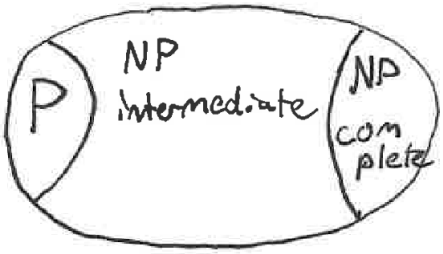
5.1 Exponential Time Hypothesis

The hardness of SAT can be formalized as an assumption, ETH = Exponential Time Hypothesis. Essentially, ETH states SAT cannot be solved in subexponential time. It is a stronger assumption than P $\neq$ NP since it implies P $\neq$ NP but also other things. If you assume ETH, you are assuming SAT has no $2^{o(n)}$ time algorithm. In certain proofs, if we assume ETH instead of P $\neq$ NP, it will make some proofs easier.

This proof disproves this



This proof proves this



5.2 The proof

We prove Ladner's Theorem: If P $\neq$ NP then there exists languages which:

- are not in P
- are in NP
- are not NP-complete

We are proving that if $\mathsf{P} \neq \mathsf{NP}$ then the NP-intermediate languages exist. If you could prove these exists unconditionally, then of course you found a language in $\mathsf{NP} \setminus \mathsf{P}$, and would prove $\mathsf{P} \neq \mathsf{NP}$. We will proceed by a kind of diagonalization, proving a weaker result. We will assume ETH instead of $\mathsf{P} \neq \mathsf{NP}$ to get an easier proof with the same ideas.

Assume ETH. There is no sub-exponential time algorithm for SAT, SAT $\notin TIME(2^{o(n)})$. Recall that we measure the run-time of an algorithm as a function of the input size. If we take a reasonable problem, and then just pad on a bunch of stuff, we can say an algorithm is sub-exponential in the padded size, even if it wasn't sub-exponential in the true size. We pad SAT by a quasi polynomial amount.

$$L = \{\langle \phi, 1^{2^{\sqrt{|\phi|}}} \rangle \mid \phi \in \text{SAT}\}$$

- First we show that $L \in \mathsf{NP}$. Our witness is the assignment, same as SAT. Our verifier V on input $\langle w, c \rangle$ checks if w is of the form $\langle \phi, 1^{2^{\sqrt{|\phi|}}} \rangle$, doing some math to count the padding. Then it checks if c is a satisfying assignment for ϕ. If it is then it accepts. In terms of the size of the input, the verifier V takes polynomial time, so we see that $L \in \mathsf{NP}$

- Next we show $L \notin \mathsf{P}$. Suppose it was. Then there exists an algorithm, A, to decide L in time polynomial in the size of the input. A runs in time $O((n + 2^{\sqrt{n}})^k)$ for some k, where n is the size of just the formula and not the padding. We give a sub-exponential time algorithm, A', for SAT. A' decides SAT in time in $O(2^{\sqrt{n}}) + O((n + 2^{\sqrt{n}})^k) = 2^{O(\sqrt{n})} = 2^{o(n)}$. This violates our assumption of ETH, so $L \notin \mathsf{P}$

Algorithm 1 A'

on input ϕ
build $\langle \phi, 1^{2^{\sqrt{|\phi|}}} \rangle = y$
run $A(y)$

- Now we show L is not NP-complete. The proof idea is that if it was, there is a reduction for it, such a reduction could be used to solve SAT too fast, violating ETH. Assume to the contrary that L is NP-complete. Then there exists a polytime computable reduction such that SAT $\leq_p L$. This reduction function, f, works such that $f : \psi \to \langle \phi, 1^{2^{\sqrt{|\phi|}}} \rangle$ and $\psi \in SAT \iff \langle \phi, 1^{2^{\sqrt{|\phi|}}} \rangle \in L$, where ψ, ϕ may be different. Since our f is polytime, there exists some k such that $|f(\psi)| = n^k$. Any polynomial time algorithm (here, a reduction) can only produce a polynomial-sized output. It take time to write that output down. Here $|\psi| = n$, the size of the input. Since the output is $\langle \phi, 1^{2^{\sqrt{|\phi|}}} \rangle$, ϕ must be small enough such that $2^{\sqrt{|\phi|}} \leq |f(\phi')| = n^k$, so

  $$2^{\sqrt{|\phi|}} \leq n^k \Rightarrow \sqrt{|\phi|} \leq k \log n \implies |\phi| \leq (k \log n)^2$$

  or that $|\phi| << n$, $o(n)$.

  Since $|\phi|$ is much smaller than n, than $|\psi|$, its faster for us to brute force check assignments of ϕ than of ψ. To see if $\psi \in$ SAT, perform the polytime reduction and try all assignments of ϕ. This will take time $2^{(k \log n)^2} = 2^{o(n)}$, violating ETH.

Therefore, we conclude that assuming ETH implies there exists a language L such that $L \notin \mathsf{P}$, $L \in \mathsf{NP}$, but L is not NP-complete. So the class NP-intermediate exists.

Note that the difficulty of the proof changes if we have to use the assumption $\mathsf{P} \neq \mathsf{NP}$ instead of ETH. We reached a contradiction twice using the fact that SAT had a $2^{o(n)}$ time algorithm. If we had to do the full proof, we would have to show that SAT had a polytime algorithm.

CS 4510 Automata and Complexity 4/13/2023

Lecture 19: Savitch's Theorem

Lecturer: Abrahim Ladha *Scribe(s): Rahul*

So far, we proved a few theorems in and around NP. We proved the Cook-Levin theorem, that SAT was NP-complete. We also proved Ladner's theorem, that if $\mathsf{P} \neq \mathsf{NP}$ then there exists languages $\notin \mathsf{P}$, $\in \mathsf{NP}$ and not NP-complete. Today's lecture will be on space, that "other" resource. Space is a very different resource than time. After an algorithm finishes running, you get the space back. You can never get the time back. This makes space both a less interesting and more interesting resource to study since it uses techniques and tricks which would not work for time. They are less applicable, but interesting in their own right. For example, performing super-exponential search to use one less unit of space.

1 Space as a Resource

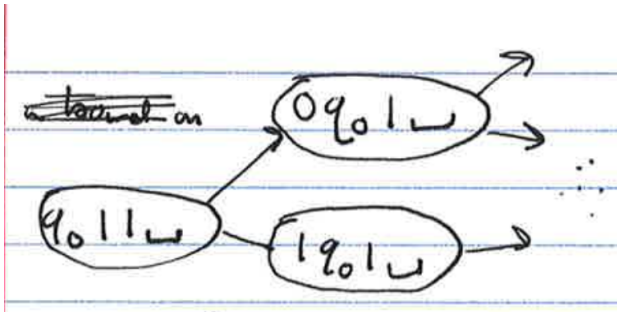
Recall $\mathsf{TIME}(f(n)), \mathsf{SPACE}(f(n))$ are the classes of languages decidable in $f(n)$ time or space, respectively. We prove the following containment chain:

$$\mathsf{TIME}(f(n)) \subseteq \mathsf{SPACE}(f(n)) \subseteq \mathsf{TIME}(2^{O(f(n))})$$

Consider a language decidable in $f(n)$ time. There exists a Turing machine which takes $f(n)$ steps to decide this language on inputs of length n. At each step, it may use at most one new cell of the tape. So a machine which uses $f(n)$ time can use no more than $f(n)$ space. The first containment then follows. We show a stronger result to prove the second containment.

$$\mathsf{SPACE}(f(n)) \subseteq \mathsf{NSPACE}(f(n)) \subseteq \mathsf{TIME}(2^{O(f(n))})$$

The first containment follows from the generalization of non-determinism. We can now show the second containment in a creative way. Given a language decidable by a non-deterministic Turing Machine in $f(n)$ space, we want to show this language is decidable deterministically in $2^{O(f(n))}$ time. We will do so by graph search! For some specific N, w, let the configuration graph G be a directed graph such that each node corresponds to a configuration of N on w. Note that if N runs in $f(n)$ space, then this graph is not infinite. There exists a bound of the possible number of vertices. Also notice that since as defined, N must halt on all inputs, this graph does not contain a cycle.



Note that we do not count the input as a part of the space used. In some models, the input is on a separate read-only tape. Since our machine N is non-deterministic, our graph may have a arity greater than one. We may assume it has arity no more than two. In order to show $\mathsf{NSPACE}(f(n)) \subseteq \mathsf{TIME}(2^{O(f(n))})$ we give an algorithm which runs in $2^{O(f(n))}$ deterministic time. First using N, w build the configuration graph. Then we perform BFS from the start configuration C_o to an accepting one C_a. BFS is linear time in the size of input. This graph has worst case $2^{O(f(n))}$ nodes, as that is the number of possible configurations of an $f(n)$ space machine. It also takes that long to build the graph, so we see this is a $2^{O(f(n))}$ deterministic algorithm so $\mathsf{NSPACE}(f(n)) \subseteq \mathsf{TIME}(2^{O(f(n))})$.

2 Savitch's Theorem

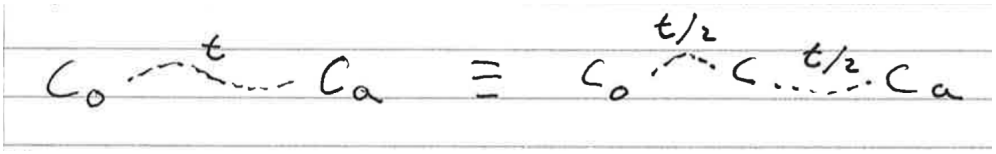
Our main result today:

$$\mathsf{NSPACE}(f(n)) \subseteq \mathsf{SPACE}(f^2(n))$$

with some conditions on f. Let us first interpret this result. We somehow are able to "de-nondeterminisfy" something with only polynomial overhead in the resource used. Could such a technique apply to P vs NP? Probably not, or someone would have found it by now. So although we only get polynomial space cost, we can infer we probably will get a super-polynomial, maybe exponential time cost. Our deterministic algorithm may only use $f^2(n)$ space, but it should probably use $2^{f(n)}$ time to perform this simulation.

A second immediate remark is that since polynomials are closed under composition, multiplication, we see $\mathsf{NPSPACE} = \mathsf{PSPACE}$. The study of space, already looks very different than the study of time. This should be an analogous problem to P vs NP. Unlike that problem, this result is unexpected, and we have been able to solve it.

Now onto the proof. Rather than some naive strategy, we are going to use divide and conquer. We want to simulate a nondeterministic Turing machine N which uses $f(n)$ space, deterministically using no more than $f^2(n)$ space. If C_o is the start configuration, C_a is the accept configuration, and C is some other configuration, notice that $C_o \overset{*}{\vdash} C_a$ in t steps if $C_o \overset{*}{\vdash} C$ in $t/2$ steps and $C \overset{*}{\vdash} C_a$ in $t/2$ steps for some t.



This will be our divide and conquer recurrence. We brute force search for some C and perform our recurrence in this way. Importantly, our recursive calls are run sequentially and reuse space.

It certainly is correct. M is a deterministic simulator of N, so it decides the same language. Now onto the analysis. If N uses $f(n)$ space, we hope to show M simulates N in no more than $f^2(n)$ space. For each recursive call, a stack frame containing C_i, C_j, t is stored. Each level of recursion uses $O(f(n))$ space, as a worst case. C_i, C_j are of size $O(f(n))$ since N uses $f(n)$ space. Each level divides $t = 2^{df(n)}$ in half. It may help if you recall anything about the Master theorem, or even geometric series. Here we won't measure

Algorithm 1 $M(N, w)$ Deterministic simulator of N on w

C_0 = start configuration of N on w
C_a = accepting configuration of N
d = chosen such that N has no more than $2^{df(n)}$ configurations
$YIELDS(C_0, C_a, 2^{df(n)})$

Algorithm 2 $YIELDS(C_i, C_j, t)$

```
if C_i = C_j then
    return true
end if
if t = 1 then
    if C_i ⊢ C_j in one step by δ of N then
        return true
    end if
else t > 1
    for configuration C of N of size f(n) do
        YIELDS(C_i, C, t/2)
        YIELDS(C, C_j, t/2)
        return true if both calls return true
    end for
end if
return false
```

time, but space. The depth of our recursion tree is $\log t = log(2^{df(n)}) = O(f(n))$. Since each level of our recursion tree takes $O(f(n))$ space and our recursion has $O(f(n))$ depth we observe the total space used is $O(f(n)) \cdot O(f(n)) = O(f^2(n))$

I mentioned that there were some restrictions on $f(n)$. First is that we may assume it is space-constructible, that M can compute $f(n)$ within $O(f(n))$ space. Most obvious functions have this property, but some crazy ones do not. Second is that f was super-linear, that $f(n) \geq n$. This can be improved to $f(n) \geq log(n)$ with some automata specification. A final remark, Hartmanis came up with a similar idea but to prove a theorem about context-free languages[1].

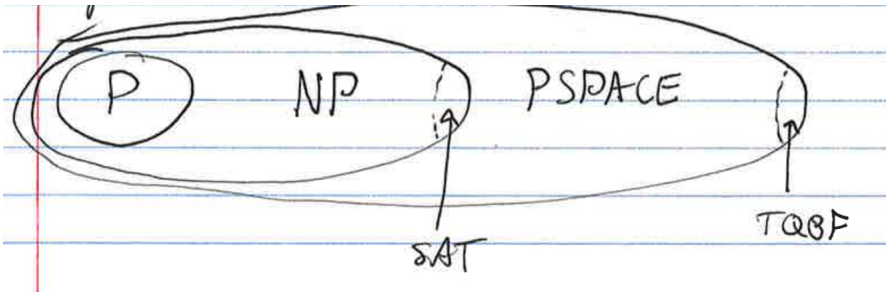
3 PSPACE-completeness

Recall that SAT is NP-complete, a boolean formula might look like $(x_1 \vee x_2 \vee x_3)$. This is not a boolean formula so much as it is a logical formula! We just hide the quantifiers. We say a boolean formula is satisfiable if there *exists* a satisfying assignment. We could simply quantify over the assignment, like $\exists x_1 \exists x_2 \exists x_3 (x_1 \vee x_2 \vee x_3)$.

What if we allow for universal quantifiers? Like $\forall x_1 \forall x_2 \exists x_3 (x_1 \vee x_2 \vee x_3)$? This is called TQBF: True Quantifies Boolean Formula. TQBF = $\{ \phi \mid \phi$ is a true quantified boolean

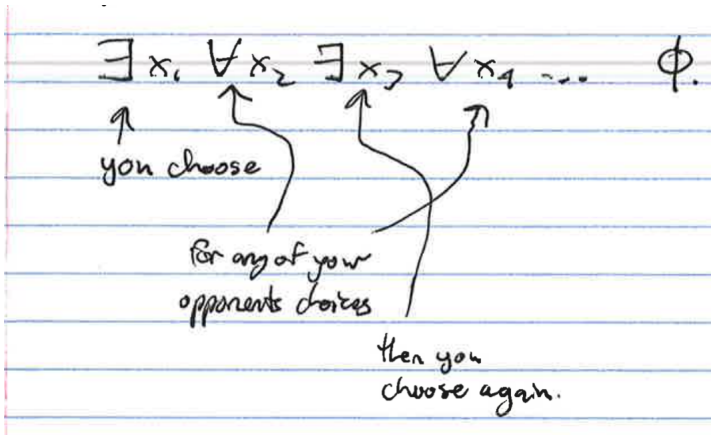
[1]See this post by Lipton for some fascinating history of the theorem `https://rjlipton.wpcomstaging.com/2009/04/05/savitchs-theorem`

formula }. Turns out that as SAT is NP-Complete, TQBF PSPACE-complete. The intuition is that since TQBF is a generalization of SAT, it should be harder than SAT.



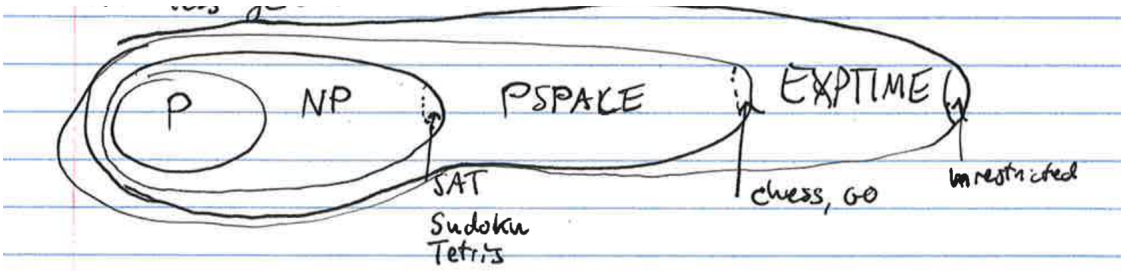
Notice SAT has structure like most puzzles. A puzzle is a single-player device in which you make a sequence of decisions to reach some goal. Intuitively, $\exists x_1, \exists x_2, ...$ is your sequence of decisions. Many puzzles are NP-complete since they can encode this structure.

Notice TQBF has structure like two player games of perfect information. Consider a TQBF with quantifiers like $\exists\exists\forall\forall\exists\exists$... you can reformulate this into a TQBF with quantifiers which only alternate, like $\exists\forall\exists\forall$... With a little abuse of types, you turn two of the same kind of quantifier into one as $\exists x_1 \exists x_2 \equiv \exists(x_1, x_2)$. Having a TQBF with alternating quantifiers looks like a game! It is a literal minimax. You make a choice, then for all possible moves the opponent could make, then you make a choice, then the opponent, and so on.



Most two player games, under appropriate restrictions and generalizations, are PSPACE-complete. Chess, checkers, Go and more. Some appropriate restrictions would be that the game require perfect information (no shadowed areas of the map), be generalized in some way[2] and a polynomial bound on the depth of number of moves. Without this bound many of these games are actually EXPTIME-complete although their proofs are less general.

[2]Recall that chess is played on a fixed game with a fixed number of pieces. There is no way to measure its complexity as a function of some n, as its technically a finite game. Generalized chess is proven to be PSPACE-complete if you generalize the board size as a function of n.



Because of how we can interpret TQBF vs SAT, we can also intuitively say that games are harder than puzzles. It should require a proof that TQBF is PSPACE-complete, like we did for SAT, but we don't have enough time.

CS 4510 Automata and Complexity 4/17/2023

Lecture 20: Relativization

Lecturer: Abrahim Ladha *Scribe(s): Samina Shiraj Mulani*

1 Diagonalization

First we begin with a review of diagonalization. Diagonalization is a proof technique. You order some elements, construct a "fixed point", or a "diagonal", and take its opposite, reaching some kind of contradiction. We will reprove the halting problem, but this time, using a functional notation.
Let $\varphi_1, \varphi_2, ...$ be an enumeration of the recursive functions (these correspond to Turing machines which are allowed to loop). We prove that there is no total computable function h -

$$h(x,y) = \begin{cases} 1 & \varphi_x(y) \text{ halts} \\ 0 & \varphi_x(y) \text{ loops} \end{cases}$$

Assume to the contrary that h is total and computable. Consider the function d:

$$d(x) = \begin{cases} 1 & \text{if } h(x,x) = 0 \\ \text{loops} & \text{if } h(x,x) = 1 \end{cases}$$

Certainly d exists since h does. d is a recursive function, so there must exist some i such that $d = \varphi_i$. Consider d on its own index i.

$$d(i) = 1 \iff h(i,i) = 0 \iff \varphi_i(i) \text{ loops} \iff d(i) \text{ loops} \tag{1}$$

$$d(i) \text{ loops} \iff h(i,i) = 1 \iff \varphi_i(i) \text{ halts} \iff d(i) \text{ halts} \tag{2}$$

We arrive at a contradiction. There cannot exist a total recursive function for h.

2 Time Hierarchy Theorem

Now we prove a weaker form of the time-hierarchy theorem. A hierarchy is like a ladder, we are able to prove that more asymptotic time gives more power. The strongest form of the theorem says

$$\mathsf{TIME}(o(f(n)/\log f(n)) \subsetneq \mathsf{TIME}(f(n))$$

We prove a weaker form of the theorem to demonstrate the same technique. We show

$$\forall k \ \mathsf{TIME}(n^k) \subsetneq \mathsf{TIME}(n^{k+1})$$

The containment is obvious, so we only need to show existence of a language computable in time $O(n^{k+1})$ but not in time $O(n^k)$. We will diagonalize over all languages which run in time n^k and make sure our language runs in time n^{k+1}.
Let $M_1, M_2, ...$ be an enumeration of the Turing machines in $\mathsf{TIME}(n^k)$. Construct a Turing machine D as follows.

Algorithm 1

D on input w_i
    Compute $n = |w_i|$
    Simulate M_i on w_i for n^k steps
    if M_i accepts w_i **then**
        reject
    end if
    if M_i rejects w_i **then**
        accept
    end if

Notice D on input w_i returns $1 - M_i(w_i)$. So $\nexists j$ such that $L(D) = L(M_j)$, so $L(D) \notin \mathsf{TIME}(n^k)$. Since it differs from every n^k-time machine, there is no n^k time algorithm to decide $L(D)$. What is the cost of simulation? Turns out this is complicated but we can safely upper bound that simulation of M_i for a single step takes at most $O(n)$ steps for the simulator. Since $n^k n = n^{k+1}$, we conclude $L(D) \in \mathsf{TIME}(n^{k+1})$. Two quick comments, first, the n^k machines are not enumerable, so this is only a rough proof idea. There is a fix around that, but it can be quite messy. Second, notice that the tightness of our hierarchy depends on the complexity of simulation. This can vary actually from this or that Turing machine formalization. The fact there is a hierarchy at all is the take-away, rather than the specific hierarchy itself.

3 Remarks

Both of these proofs had the same structure. We interacted with other machines but only in a black-box way. We simulated them only to disagree with their output. We did nothing with M except run it. We did not deal with any of the internal mechanics of computation. We can separate the decidable from the undecidable and an infinite hierarchy of deterministic time classes. Note that such a proof could also separate P from EXP. Could we use such techniques to separate P from NP? Turns out, no. Thats the point of today's lecture.

4 Relativization

The oracle of Delphi was like a witch or a shaman. You would bring her gifts and she would answer your questions. There would be no provided explanation. Antiquated version of a magic eight ball. An oracle machine has some similar mysticism. It is a Turing Machine with

an additional tape. It writes down a query on this special tape and issues an instruction. Then the tape clears and all that is left is the answer, a 1 or 0. This occurs in unit time. We formalize an oracle as a language A, and the oracle machine M^A. This oracle machine M^A can test membership to language A in unit time. Many natural things we have discussed appear to be representable in an oracle way. Nondeterminism was originally formulated like an oracle. Both many-one and polytime reductions could be formalized in an oracle fashion.

For a class C and language A, we let C^A be the languages decidable by C-machines with oracle access to A. For example, consider the structure of $\mathsf{P}^{\mathsf{SAT}}$. Certainly, any oracle machine can ignore its oracle, so $\mathsf{P} \subseteq \mathsf{P}^{\mathsf{SAT}}$. Also notice that all of NP is deterministic polytime computable relative to SAT. Since SAT is NP-complete under polynomial time reduction, $\mathsf{NP} \subseteq \mathsf{P}^{\mathsf{SAT}}$. Certainly $\mathsf{P}^{\mathsf{SAT}}$ is very different than P. We won't explain why, but $\mathsf{NP}^{\mathsf{SAT}}$ is actually bigger than NP.

We don't really care about comparing two classes, one with and one without the oracle. We care about a "relativized world". One in which every machine has oracle access. For some fixed A, what does the world look like? What is the relationship between $L^A, \mathsf{P}^A, \mathsf{NP}^A, \mathsf{PSPACE}^A$ and so on. How does this relativized world differ from our own? For each fixed A, there exists an entire separate world with its own language and rules and relationships. We can relate these worlds to our own.
We say a proof "relativizes" if you can copy paste it from our world to the relativized world with only a minor modification.

Algorithm 2

D on input w_i
- Compute $n = |w_i|$
- Simulate M_i on w_i for n^k steps
- **if** M_i accepts w_i **then**
  - reject
- **end if**
- **if** M_i rejects w_i **then**
  - accept
- **end if**

Algorithm 3

D^A on input w_i
- Compute $n = |w_i|$
- Simulate* ${M_i}^A$ on w_i for n^k steps
- **if** ${M_i}^A$ accepts w_i **then**
  - reject
- **end if**
- **if** ${M_i}^A$ rejects w_i **then**
  - accept
- **end if**

In our world, D simulates M_i. In the relativized world, D^A simulates ${M_i}^A$. If ${M_i}^A$ makes an oracle call to A, D^A simulates this instruction of ${M_i}^A$ by calling its own oracle. Here the simulation only has this slight difference. Could there exist a proof of diagonalization for $\mathsf{P} \neq \mathsf{NP}$? It might look like:

Algorithm 4

Enumerate P machines $M_0, M_1, ...$
D on input w_i
- Simulate M_i on w_i
- Return opposite

Somehow show $L(D) \in \mathsf{NP}$

Algorithm 5

Enumerate P^A machines $M_0, M_1, ...$
D^A on input w_i
- Simulate* ${M_i}^A$ on w_i
- Return opposite

Somehow show $L(D^A) \in \mathsf{NP}^A$

If a relativizing proof exists in our world, we say it relativizes to all worlds. If $\mathsf{P} \neq \mathsf{NP}$ in our world, and it is provable in this way, then $\forall A\ , \mathsf{P}^A \neq \mathsf{NP}^A$. It holds in all relativized worlds. However, if there exists A such that $\mathsf{P}^A = \mathsf{NP}^A$, then there exists a world whose version of the problem is $\mathsf{P}^A = \mathsf{NP}^A$, so there cannot exist a relativizing proof that $\mathsf{P} \neq \mathsf{NP}$ from our own. Similarly if $\exists B$, a relativized world where $\mathsf{P}^B \neq \mathsf{NP}^B$, then there cannot exist a relativizing proof that $\mathsf{P} = \mathsf{NP}$.We show two oracles A, B such that

$$\mathsf{P}^A = \mathsf{NP}^A \qquad\qquad \mathsf{P}^B = \mathsf{NP}^B$$

Since there exists two worlds, one where $\mathsf{P} = \mathsf{NP}$ and one where $\mathsf{P} \neq \mathsf{NP}$, no proof of $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$ in our world can generalize to all worlds. Our demonstration of contradictory relativizations will prove that there is no relativizing proof of P vs NP in our world! You cannot use diagonalization to separate P from NP!!!!!

5 A World Where its True

We choose A to elevate $\mathsf{P}^A, \mathsf{NP}^A$ to the same class where non-determinism gives no power. Certainly $\forall A\ \mathsf{P}^A \subseteq \mathsf{NP}^A$ so we show for some A that $\mathsf{NP}^A \subseteq \mathsf{P}^A$. We will use space complexity. Let $A = \mathsf{TQBF}$. Then

$$\mathsf{NP}^A = \mathsf{NP}^{\mathsf{TQBF}} \subseteq \mathsf{NPSPACE} = \mathsf{PSPACE} \subseteq \mathsf{P}^{\mathsf{TQBF}} = \mathsf{P}^A$$

Relative to TQBF, every PSPACE language is decidable in nondeterministic polynomial time. The second containment holds by Savitch's theorem. The final one holds similarly to why $\mathsf{NP} \subseteq \mathsf{P}^{\mathsf{SAT}}$. We observe then that relative to $A = \mathsf{TQBF}$ that $\mathsf{NP}^A = \mathsf{P}^A$.

6 A World Where its False

Showing oracle B such that $\mathsf{P}^B \neq \mathsf{NP}^B$ will be much harder. Ironically, we will construct B by diagonalization. We want to show a language exists which could not be in P^B. How we will show it cannot be done in polynomial number of steps? For correctness, we will require an exponential number of oracle queries. Since each query takes unit time, an exponential number of queries implies that the machine to decide this language must take exponential time, and and thus could not have been in P^B.
Let L_B = set of strings such that there is some string of the same length in B. We don't say which string to require making 2^n queries.

$$L_B = \{w \mid \exists\, x \in B \text{ with } |x| = |w|\}$$

Let ${M_1}^B, {M_2}^B, ...$ be an ordering of the oracle machines of P^B. Lets even suppose they are weakly sorted to guarantee ${M_i}^B$ halts in time n^i on all inputs.
We construct B in a sequence of steps such that ${M_i}^B$ does not decide L_B is ensured in stage i. At each stage, only a finite amount of strings have been decided to be in B and decided

to not be in B.
Suppose we are stage i. Let w be the largest string in B. Choose n such that $2^n > n^i$ and $n > |w|$. We will increase the knowledge about B such that ${M_i}^B$ accepts $1^n \iff 1^n \notin L_B$. Run ${M_i}^B$ on 1^n. On its query to oracle B, if it has been queried by that string before, the oracle will respond consistently. If B has not seen the string before, it will prophesize no, Since ${M_i}^B$ runs in time n^i, it does not have time to query B on all 2^n strings of length n. If ${M_i}^B$ accepted 1^n, all other strings of length n are declared not to be in B. If ${M_i}^B$ rejected 1^n, declare one string of length n to be in B. Therefore $L({M_i}^B) \neq L_B \notin \mathsf{P}^B$.
Why is $L_B \in \mathsf{NP}^B$? Rather than testing all 2^n strings against the oracle, nondeterministically guess the right one to test the oracle against. This takes unit time on an NP^B machine but exponential time on a P^B machine. Since $L_B \in \mathsf{NP}^B \setminus \mathsf{P}^B$, we observe $\mathsf{P}^B \neq \mathsf{NP}^B$.

7 Frustration. Coping. Crying.

This result has set the stage for the next half-century of complexity research. Any proof which could resolve P vs NP would genuinely have to use new techniques, ones which do not relativize. We weren't even sure at the time if such techniques existed! The last fifty years has seen attempts trying to bend the rules. To list a few:

- Randomness: What if SAT is decidable in polytime by an algorithm which returns the assignment correctly only two thirds of the time? Maybe the deterministic requirement of the algorithm is too stringent.
- Approximation: What if there exists an algorithm for SAT to satisfy a majority of the clauses in polytime, but this last stretch to all clauses requires exponential? Maybe the correctness requirement of the algorithm is too stringent.
- Circuits: Proofs using circuits do not appear to relativize. Does there exist a super polynomial lower bound on circuit size for SAT? Maybe the uniformity requirement of the algorithm is too stringent.

All of these areas have been good at asking questions and bad at giving answers. We are further from answering P vs NP than when the question was conjectured. The relativization barrier was just the first hurdle, we would hit many many more. Truly, there is no harder problem. No problem has produced more corpses than P vs NP.

CS 4510 Automata and Complexity April 19, 2023

Lecture 19: P/poly

Lecturer: Abrahim Ladha *Scribe(s): Rishabh Singhal*

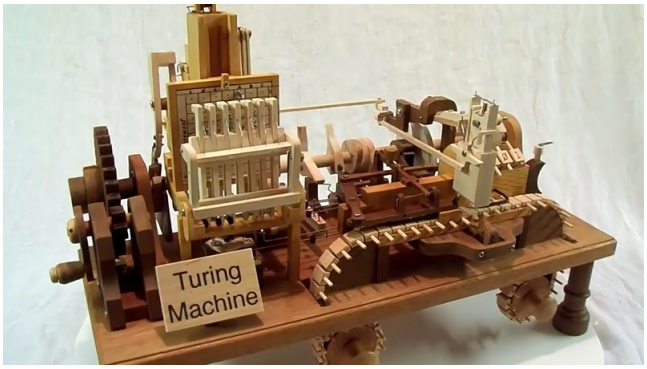
1 Review

Last time we proved that P vs NP has no relativizing proof. Today, we are going to explore one direction into non-relativizing techniques.

2 Motivation from Hardware

First, why are black box techniques so common? Why were all the early proofs black box anyway? Why were all the early proofs black box anyway? My hypothesis is that even though a Turing Machine is a simplification of computation to its base essentials, it is still too complicated. There is some state of a moving tape head that moves conditionally are reading the tape. There is some transition function that may requires a lookup, so a similar conditional read-move kind of device. It is not impossible to have a non-relativizing proof, using Turing Machines, but it appears to be unobvious.

Go to youtube and search for some homemade Turing-machines. If the Turing machine is truly a foundational computer, there should exist people building them. However, most of the builds implement a Turing machine on top of some other computer! These are mostly raspberry-pi style projects. There was only a single mechanical Turing machine I could find which wasn't implemented on top of some other kind of computer. Surprisingly, it was made of wood!



If you are looking for interesting hardware computers, there appears to be a million builds which use boolean circuits as a foundation instead of a Turing machine. There exists boolean circuit constructions from water and tubes, dominoes, marble machines, and so on.

Why are most of the computer builds with circuits and not with Turing machines? I thought the Turing machine was the foundational, most simplistic kind of computer? The answer is the same as why most proofs up to the relativization barrier were black box.

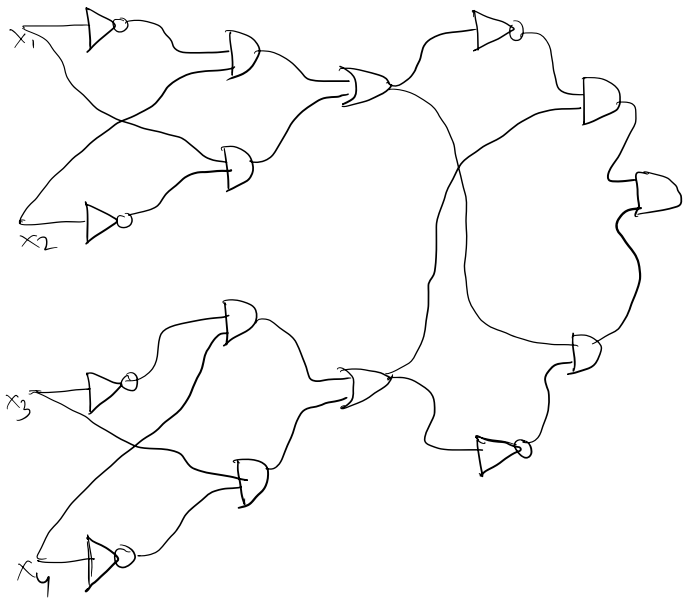


Figure 1: Circuit

The Turing machine has too many moving delicate parts. Even in minecraft, its easier to build a circuit using redstone then a conditionally moving tape head with pistons. What are the internals of a Turing machine? Its hard to say, its basically alive. What are the internals of a circuit? Just more circuits. Basically a glorified pachinko machine. You break a Turing machine into two, you have nothing. You break a circuit into two, you now have two circuits. The fact that the internals are simpler to inspect, this makes them an excellent candidate to sidestep the relativization barrier.

3 Circuits

A boolean circuit is a wiring of gates that is used to compute a boolean function $\{0,1\}^n \to \{0,1\}$. We say a model of computation is uniform if it's devices accept input of any arbitrary

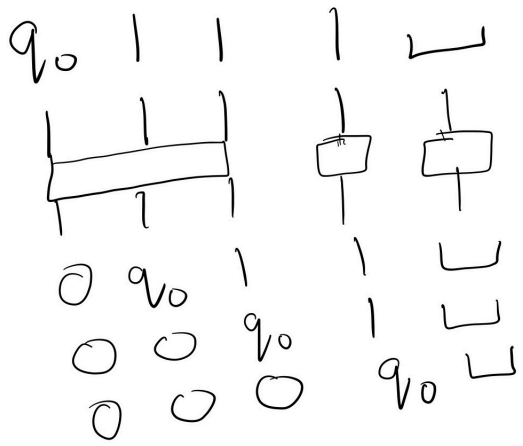


Figure 2:

size. These include TM's, even DFA's and PDA's. Circuits however have a fixed input size. A 32-bit adder will safely and correctly add inputs of less than 32 bits, but not more. A circuit family is a collection $\{C_0, C_1, C_2, ...\}$ where each C_i is a circuit with i input wires and one output wire. We say a circuit family $\{C_0, C_1, ...\}$ decides some language L if

$$w \in L \iff C_{|w|}(w) = 1 \tag{1}$$

The complexity of a circuit family is not something related to time but to the size of the circuit, the number of gates as a function of n. We let the class $\mathsf{SIZE}(f(n))$ denote languages decidable by circuit families of size $f(n)$. Surely this is not so different from time complexity. You might think that $\mathsf{SIZE}(f(n)) \subseteq TIME(f(n))$, but lets see what happens. Let $L \in \mathsf{SIZE}(f(n))$, then L has an $f(n)$ sized circuit family. To build a Turing machine to accept $w \in L$, we simply need to simulate the circuit $C_{|w|}$. Each gate takes constant time so this decides for L in time $f(n)$ so $L \in TIME(f(n))$. The problem is you cannot encode infinitely many boolean circuits into a constant-sized machine. What if you could compute the circuits? This is our second roadblock, we made no mention of the fact for any circuit family that $f(n) \to C_n$ need to be computable. In fact, the language HALT in unary HALT $= \{1^{\langle M,w \rangle} | M \text{ halts on } w\}$ has a polynomial-sized circuit family since as it turns out, all unary languages have polynomial-sized circuit families. We will elaborate on this later.

Instead, for a more constructive proof, we prove

$$TIME(f(n)) \subseteq \mathsf{SIZE}(f^2(n)) \tag{2}$$

4 Cook-Levin

We proceed by conversion of computation history of a machine that runs in time $f(n)$ to a circuit of size $f^2(n)$. The proof is basically identical to the Cook-Levin theorem without any polynomial restriction. First, convert $\Gamma \cup Q$ to binary, perhaps like $\{0, 1, q_0, w\} \to \{00, 11, 10, 01\}$. We add in a bunch of these gates to represent the transition from now to now. The tape, the sequential state updates of the machine during its execution, does not seem to appear anywhere. It has not vanished, it is encoded in the intermediary wires! This circuit exists to correctly simulate some fixed M on w. but w is provided on the input wires in a suitable encoding. Since M runs in $f(n)$ time, it can also use at most $f(n)$ space. The height of this circuit is the time, and the width is the space. So the number of gates is $c \cdot f(n) \cdot f(n)$, with the most gates being identity ones, but the others taking a constant more than one to be represented in a reasonable circuit basis. Either way, for $L \in TIME(f(n))$, we see $L \in \mathsf{SIZE}(f^2(n))$ completing the proof. Note that this could be improved to build a circuit of size $f(n) \log f(n)$. Either way, we see that not only are circuit families Turing-complete, they are also quite efficient!

5 Turing Machines which take Advice

For any class C and function f, we use C/f to denote the class of languages decidable by C-machines given access to f(n) bits of advice. There exists a second tape in which some string of answers is prewritten. The C machine may sequentially read from this tape to "take advice". Observe the following:

- $C/0 = C$
- If $f < g$ then $C/f < C/g$
- $\mathcal{P}(\Sigma^*) \subseteq \mathsf{P}/2^n$
- If f contains a one infinitely often, perhaps is the characteristic string of some undecidable language, then C/f may contain undecidable languages.

6 P/poly

We denote P/poly as the class of languages decidable by a Turing Machine which halts in polynomial time given access to a polynomial amount of advice. It turns out that P/poly is also exactly the class of languages that have polynomial-sized circuit families. Let's prove it.

Let L be decidable by a polynomial-sized circuit family then there exists a polynomial-sized circuit for each input size. Choose a description of this circuit to be the advice. The M on input w simply simulates w on C_n. This takes polynomial time so $L \in \mathsf{P}/\mathsf{poly}$.

Let $L \in \mathsf{P}/\mathsf{poly}$. We show there exists a polynomial-sized circuit family to be decidable L. Since $L \in \mathsf{P}/\mathsf{poly}$ there exists a polynomial time Turing Machine with access to polynomial advice. Convert the polynomial machine to a polynomial-sized circuit, and simply hard

code the advice, perhaps at the depth of the input. This resulting circuit is still polynomial sized so we observe that L has a polynomial-sized circuit family.

From here on, we may simply refer to $\mathsf{P/poly}$ as languages with polynomial-sized circuit families since we care about those more than machines which take advice.

It is true that $\mathsf{P} \subseteq \mathsf{P/poly}$ by the advice definition. Note that this containment is strict only since we allow non-computable circuit families. We could prove all unary languages have polynomial-sized circuits, including the undecidable ones previously mentioned. Since P can only contain decidable languages, the containment $\mathsf{P} \subsetneq \mathsf{P/poly}$ must be strict.

CS 4510 Automata and Complexity April 24, 2023

Lecture 22: The Polynomial Hierarchy

Lecturer: Abrahim Ladha *Scribe(s): Michael Wechsler*

1 Motivation

We will characterize the same thing three ways and hopefully prove just one theorem. Recall P is the class of languages we characterize as having a machine M which decides in polynomial time.

$$M(w) \text{ accepts} \iff w \in L$$

Recall NP is the class of languages verifiable in polynomial time, or decidable in nondeterministic polytime.

$$\exists x M(w, x) \text{ accepts} \iff w \in L$$

You may either think of x as the witness of a deterministic verifier or as a sequence of decisions or guesses that a nondeterministic machine makes. For M to run in polynomial time, we require that x be polysized. Recall that an NTM accepts if there exists ($\exists$) a computation branch. You may also recall that SAT is NP-complete, and we say that $\phi \in$ SAT if there exists ($\exists$) a satisfying assignment of the boolean variables of ϕ.

Recall coNP is the class of languages such that $\overline{L} \in \mathsf{NP} \iff L \in \mathsf{coNP}$. We can take the logical complement[1] of the definition of NP for a definition of coNP as

$$\forall x M(w, x) \text{ accepts} \iff w \in \overline{L}$$

Like SAT is NP-complete, coNP has its own complete problem called tautologies. TAUT $= \{ \phi \mid$ every assignment of ϕ is satisfying$\}$. We observe that NP and coNP have this interesting duality. A ying-yang structure emerges. NP is characterized by the existential quantifier $\exists$ ($\exists$ a witness, $\exists$ an assignment of SAT, or $\exists$ an accepting computation). coNP is characterized by the universal quantifier $\forall$ ($\forall$ witnesses, $\forall$ assignments of TAUT, or $\forall$ branches must be accepting). This duality motivates our discussion.

2 Generalizations of NP and coNP

For any class C, define the class $\exists C$ such that if M was a C-machine with a definition like

$$M(w) \text{ accepts} \iff w \in L \in C$$

then M' is a $\exists C$ machine such that

$$\exists x M'(w, x) \text{ accepts} \iff w \in L \in \exists C$$

[1] It should really be "$\forall x M(w, x)$ rejects" but we don't care abour rejection from L here, as we want to care about acceptance into $\overline{L}$. This distinction is arbitrary.

We naturally augment these deciders to take on witnesses[2]. We similarly define the class $\forall C$.

What is $\exists\mathsf{P}$? $\exists x M(w,x)$ accepts $\iff$ $w \in L$ and M runs in polytime. M is just a polytime verifier! So $\exists\mathsf{P}$ = classes of languages verifiable in polytime. Thus $\exists\mathsf{P} = \mathsf{NP}$. Similarly $\forall\mathsf{P} = \mathsf{coNP}$.

What is $\exists\exists\mathsf{P}$? $\exists x_1 \exists x_2 M(w, x_1, x_2)$ accepts and M is polytime? That's just two witnesses. It just complicates things. Each witness can be at most a polynomial number of bits anyway, so two witnesses is just a constant larger so $\exists\exists\mathsf{P} = \exists\mathsf{P} = \mathsf{NP}$. Similarly, $\forall\forall\mathsf{P} = \forall\mathsf{P} = \mathsf{coNP}$. Anytime we have a sequence of the same quantifier, we may compress them to one quantifier.

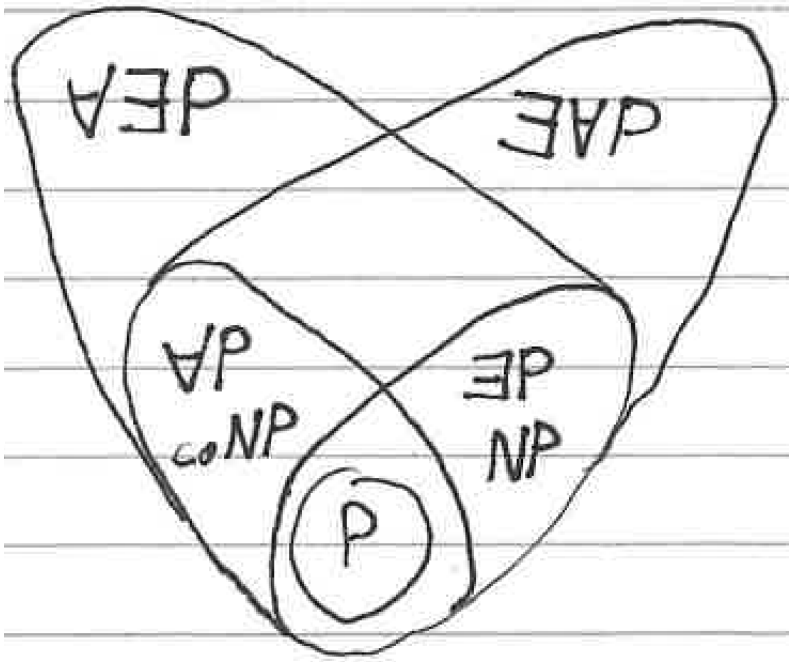
$$\exists\exists ... \exists\mathsf{P} = \exists\mathsf{P} = \mathsf{NP}$$

What about $\exists\forall\mathsf{P}$ and $\forall\exists\mathsf{P}$? Now things are getting interesting.

$$\exists x_1 \forall x_2 M(w, x_1, x_2) \text{ accepts } \iff w \in L \in \exists\forall\mathsf{P}$$

Note that we may add an $\exists$ quantifier to a $\forall\mathsf{P}$ machine which it ignores to show $\forall\mathsf{P} \subseteq \exists\forall\mathsf{P}$. We convert a $\forall\mathsf{P}$ machine to a $\exists\forall\mathsf{P}$ machine which ignores this witness. Adding an ignored parameter changes nothing of the program structure, so our machine still decides the same language. Since we can perform this surgery, $\forall\mathsf{P} \subseteq \exists\forall\mathsf{P}$. Similar logic can be used to show $\forall\mathsf{P} \subseteq \forall\exists\mathsf{P}$ and $\exists\mathsf{P} \subseteq \forall\exists\mathsf{P}$ and $\exists\mathsf{P} \subseteq \exists\forall\mathsf{P}$. Observe that $\forall\exists\mathsf{P}$ and $\exists\forall\mathsf{P}$ appear to be larger than $\exists\mathsf{P}$ and $\forall\mathsf{P}$.

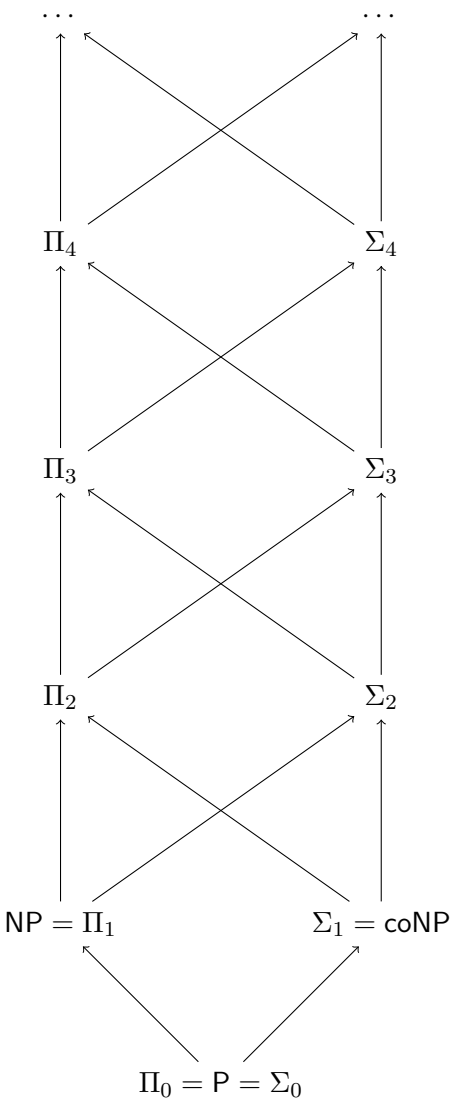
So does $\exists\forall\mathsf{P} = \forall\exists\mathsf{P}$? We don't know! $\exists\forall\mathsf{P}$ and $\forall\exists\mathsf{P}$ appear to have the same duality and dance that NP and coNP have.



[2]Although these auxilliary inputs maybe universally quantified and therefore not "witnessing" anything, we will still call them witnesses.

3 The Polynomial Hierarchy

We may repeat this argument on $\forall\exists\mathsf{P}$ and $\exists\forall\mathsf{P}$ as we did on $\exists\mathsf{P}$ and $\forall\mathsf{P}$. We see by an inductive or recursive argument this creates an infinite and alternating hierarchy. Countably many more generalizations of NP and coNP and their dualities. Here we use arrows to show containment to avoid too complex of a venn diagram.



Let $\Pi_0 = \Sigma_0 = \mathsf{P}$ and inductively define

$$\Sigma_i = \exists\Pi_{i-1} = \underbrace{\exists\forall\exists\forall\exists\ldots}_{\text{i}}\mathsf{P}$$

$$\Pi_i = \forall\Sigma_{i-1} = \underbrace{\forall\exists\forall\exists\forall\ldots}_{\text{i}}\mathsf{P}$$

It is important that the first quantifier of Σ_i is existential and the first quantifier of Π_i is universal. We define the polynomial hierarchy to be the class $\mathsf{PH} = \bigcup_{i=0}^{\infty}\Sigma_i = \bigcup_{i=0}^{\infty}\Pi_i$. We define a "level" of the polynomial hierarchy to be $\Pi_i \cup \Sigma_i$ for some i.

Note that by a generalizing the argument we made for NP and coNP, we see

$$\forall i\ \Pi_i \subseteq \Pi_{i+1}$$
$$\forall i\ \Sigma_i \subseteq \Sigma_{i+1}$$
$$\forall i\ \Sigma_i \subseteq \Pi_{i+1}$$
$$\forall i\ \Pi_i \subseteq \Sigma_{i+1}$$

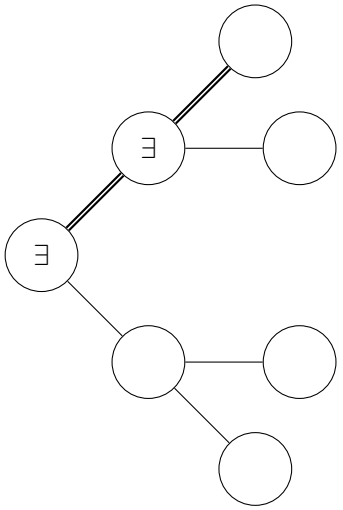
Whether or not these levels are strict is an open problem. Each level is defined only using finitely many quantifiers, so it would appear that $\mathsf{PH} \subseteq \mathsf{PSPACE}$ since $TQBF$ is a PSPACE-complete problem. If that containment is strict, it is also a an open problem. We would hope to show it is since $\mathsf{P} \subseteq \mathsf{PH} \subsetneq \mathsf{PSPACE} \Rightarrow \mathsf{P} \neq \mathsf{PSPACE}$. This is truly a beautiful class with beautiful structure of mostly theoretical interest.

We give two more quick equivalent characterizations of the polynomial hierarchy. First is using oracles:

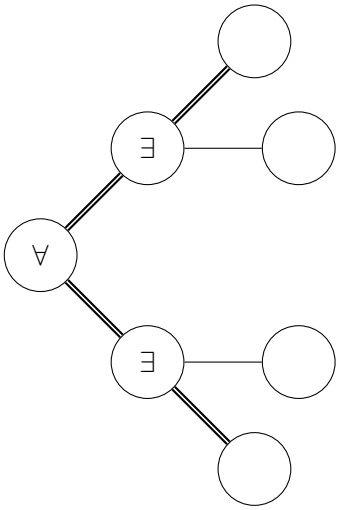
$$\Sigma_0 = \mathsf{P},\ \Sigma_1 = \mathsf{NP},\ \Sigma_2 = \mathsf{NP}^{\mathsf{NP}},\ \Sigma_3 = \mathsf{NP}^{\mathsf{NP}^{\mathsf{NP}}},\ \Sigma_i = \underbrace{\mathsf{NP}^{\mathsf{NP}^{\cdots^{\mathsf{NP}}}}}_{\text{i}},\ \ \Pi_i = \text{co-}\Sigma_i$$

We only mention this and leave it unjustified, but it is perhaps believable. The second characterization uses a generalized non-deterministic Turing Machine called an alternating

Turing machine. While a nondeterministic Turing machine accepts if just one branch accepts, an alternating Turing machine may pick from two transition functions at any step if it wants to require all branches to accept or just one:



NTM



ATM

For AP = alternating polynomial time, since $TQBF$ is PSPACE-complete, and an unbounded alternating polytime machine can simulate $TQBF$ problems alternating quantifiers. Like how SAT is NP-complete, $TQBF$ is AP-complete, and so AP = PSPACE.

We say a Σ_i-machine is one in which the first branch is at an existential one, and there are at most i existential or universal branching steps. We similarly define a Π_i-machine as an ATM which the first branching is a universal one, and there at most i universal or existential branching steps to depth i. Try to convince yourself that $\mathsf{NP} = \Sigma_i - \text{TIME(poly)}$, as you reformulate many guesses into just one. Rather than make a sequence of nondeterministic guesses, just make one bigger one. Why make two sequential coin flips when you can roll a four sided die.

$$\Sigma_i = \bigcup_{k=0}^{\infty} \Sigma_i\text{-TIME}(n^k)$$

$$\Pi_i = \bigcup_{k=0}^{\infty} \Pi_i\text{-TIME}(n^k)$$

Although the polynomial hierarchy may seem of a flamboyant, inapplicable interest, like other part of complexity, there are deep connections and ties. It may also be used to separate the complexity classes worth studying. Also most importantly, it looks cool.

4 Collapse

If for any i, $\Pi_i = \Sigma_i$ then $\mathsf{PH} \subseteq \Pi_i = \Sigma_i$. The polynomial hierarchy collapses to this i^{th} level. Each Π_i, Σ_i behave like a struct or pillar, supporting the levels above them. Were it the case that two distinct pillars were the same, our tower of Babel collapses. We prove a weaker idea, that if $\mathsf{P} = \mathsf{NP}$, then $\mathsf{PH} \subseteq \mathsf{P}$. That is, if $\Sigma_0 = \Sigma_1$, then the entire hierarchy collapses to Σ_0.

Assume $\mathsf{P} = \mathsf{NP}$. We proceed by induction. For $i = 1$, $\Pi_i = \mathsf{coNP}$ and $\Sigma_i = \mathsf{NP}$ are both $\subseteq \mathsf{P}$ by assumption. Now suppose that $\Pi_{i-1}, \Sigma_{i-1} \subseteq \mathsf{P}$. We prove $\Sigma_i \subseteq \mathsf{P}$. Since P is closed under complement and $\mathrm{co}\Sigma_i = \Pi_i$, this will also prove $\Pi_i \subseteq \mathsf{P}$ as desired. Let $L \in \Sigma_i$, then

$$w \in L \iff \underbrace{\exists x_1 \forall x_2 \ldots}_{i \text{ times}} M(w, x_1, x_2, \ldots) \text{ accepts}$$

Define

$$L' = \{\langle w, x_1 \rangle \mid \underbrace{\forall x_2 \exists x_3 \ldots}_{i-1 \text{ times}} M(w, x_1, x_2, \ldots) \text{ accepts } \}$$

Notice $L' \in \Pi_{i-1}$ since there are $i-1$ quantifiers and it begins with $\forall$. By our assumption that $\Pi_{i-1} \subseteq \mathsf{P}$ we see that $L' \in \mathsf{P}$. Thus, there exists a polytime algorithm for L' called M' such that

$$w \in L' \iff M'(w) \text{ accepts}$$

We may tranform this definition of L' into the original one for L to get

$$w \in L \iff \exists x_1 M'(w, x_1) \text{ accepts}$$

This is a polytime verifier, so $L \in \mathsf{NP}$. Since we assumed $\mathsf{P} = \mathsf{NP}$,we observe that $L \in \mathsf{P}$. Since L was any Σ_i language, then $\Sigma_i \subseteq \mathsf{P}$. So if $\mathsf{P} = \mathsf{NP}$, we may conclude that the polynomial hierarchy collapses to its 0^{th} level, to P, to ashes.

5 Karp-Lipton Theorem

This will be our final theorem of the class. Interpreting its statement is more difficult than the actual proof. It plainly states:

$$\mathsf{NP} \subseteq \mathsf{P/poly} \Rightarrow \mathsf{PH} \subseteq \Pi_2 \cup \Sigma_2$$

If SAT has a polynomial sized circuit family, then we do not have a polynomial hierarchy, it collapses to the second level. Originally Karp and Lipton proved it by a collapse to the third level, but Sipser improved it to the second level. The proof idea is to show $\Pi_2 \subseteq \Sigma_2$. Conversion of any $\forall\exists$-sentence to a $\exists\forall$-sentence means for any sentence higher in the hierarchy, we can repeatedly alternate and compress quantifiers until a sentence with many quantifiers is left with only two.

Let $\mathsf{NP} \subseteq \mathsf{P/poly}$, then SAT has a polynomial sized circuit family, $\{C_0, C_1, \ldots\}$. Each C_n takes as input an n-variable formula and outputs a single bit for yes/no if the input

formula was satisfiable. By a decision-to-search transformation, there exists a circuit family $\{C'_0, C'_1, \ldots\}$. Where each C'_n outputs n bits for not just if it was satisfiable or not, but the satisfying assignment itself. Since each C_n is polynomially-sized, so is each C'_n. This decision-to-search transformation should be believable, but to give you an example suppose we had a circuit to say if some formula ϕ was satisfiable or not. If x_1 is the first variable of ϕ, then $\phi \wedge x_1$ is a formula which is satisfiable if and only if ϕ was satisfiable, with $x_1 = 1$. We can play a hotter/colder game with the circuit families to infer not just if a formula was satisfiable, but what the actual satisfying assignment was. This decision-to-search transformation will incur only a polynomial overhead.

Let $L \in \Pi_2$. Then

$$w \in L \iff \forall x_1 \exists x_2 M(w, x_1, x_2) \text{ accepts}$$

We may convert M to a CNF φ_M using a Cook-Levin style construction. Note since M runs in a polynomial number of steps, φ_M is polynomially sized, and its construction takes polynomial time. Now, notice $\exists k$ such that $C'_k(\varphi_M, w, x_1) = x_2$. Since SAT has a polynomial sized circuit family, there exists a polysized circuit to search for this witness instead of quantifying over it. We can replace the existential quantification of x_2 with a computation of C'_k. Then rather than actually computing C'_k, we can just existentially quantify over it. So our definition of L has an equivalent statement:

$$\forall x_1 \exists x_2 M(w, x_1, x_2) \text{ accepts} \iff \exists C'_k \forall x_1 M(w, x_1, C'_K(\varphi_M, w, x_1)) \text{ accepts}$$

We may simply use existential quantification to guess the C'_k circuit. We converted a Π_2 sentence into a Σ_2 one! $L \in \Pi_2 \Rightarrow L \in \Sigma_2 \Rightarrow \Pi_2 \subseteq \Sigma_2$. We observe that if $\mathsf{NP} \subseteq \mathsf{P/poly}$, we collapse PH.

6 Further Study

I want to conclude with some advice on how to self study complexity theory. Your journey doesn't have to end here if you don't want it to. First, finish the Sipser book. It does not contain everything, but it does contain the best proofs of what it does cover. I wish it had a second volume. It's coverage of randomness, interaction, cryptography, and more may surprise you. Before you go further, you should definitely finish Sipser. After you finish Sipser, go through the first six chapters of the Arora-Barak book. All the other chapters (7+) cover an incredible breadth of material, and good pointers to other sources. These may include communication complexity, quantum complexity, the complexity of counting, and so on. Each of these chapters deserves (and has) their own books, but it's an introduction to these theories. You need to know what the things you don't know are called in order to google and learn them. Then go through Goldreich's and Papadimitriou's books. Goldreich has 400 pages of incredible notes. Finally, I recommend the books *The Nature of Computation* and Wigderson's *Mathematics and Computation*. Both of these are light on proofs, as a tradeoff for coming with incredible wisdom. If you want a proof, use the other books. If you want to what a proof means, use Wigderson's book. Of course, you may use me as a resource. If you have any questions, or come across anything in your own independent study, I would be happy to help and answer. Thank you for taking my class, I had a lot of fun.

- Introduction to the Theory of Computation, Michael Sipser 2012
- Computational Complexity: A Modern Approach, Sanjeev Arora and Boaz Barak, 2009
- Computational Complexity: A Conceptual Perspective, Oded Goldreich, 2008
- Computational Complexity, Christos Papadimitriou, 1994
- Mathematics and Computation, Avi Wigderson, 2019
- Goldreich's encyclopedic lecture notes. 375 pages across two semesters. An invaluable resource from 1999.
  `http://gen.lib.rus.ec/book/index.php?md5=fbb240574e6f5059fccdce95fab0ff38`
- Hatami's notes from 2022, also insanely useful.
  `https://www.cs.mcgill.ca/~hatami/comp531-F2022/files/Lectures.pdf`