18.404 Notes

Jason Chen
Lecturer: Michael Sipser

Last Updated: February 5, 2021

Contents

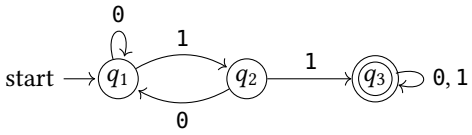
1 Automata and Languages

The class website is here.

1.1 Finite Automata

We're going to start with a discussion of computability, in which we need models of computation.

Let's start with an object M_1 that is represented by the following diagram:



We have the states q_1, q_2, and q_3, transitions represented by the arrows and their labels, the start label labeled with an incoming arrow, and the accepting state with a double outline.

The inputs are strings, and to determine whether a string is accepted or rejected, we begin at the start input, and follow arrows corresponding to each character until we reach the end of the string. If the ending state is an accepting state, we say that the input is accepted, and otherwise the input is rejected.

1 **Example**

In M_1, `0101` is rejected and `01101` is accepted.

For a finite automata M, we can think about all of the strings that M accepts:

$$\{\text{strings } w \mid M \text{ accepts } w\}.$$

We usually call this the language of M, and we also sometimes say that M recognizes A.

1.1.1 More Rigorously

2 **Definition** (Finite Automaton)

A **finite automaton** M is a tuple $(Q, \Sigma, \delta, q_0, F)$ where

- Q is a finite set of states,
- Σ is a set of symbols, called the **alphabet**,
- $\delta\colon Q \times \Sigma \to Q$ is the transition function, i.e. the arrows in the diagram,
- $q_0 \in Q$ is the start state, and
- $F \subseteq Q$ are the accepting states.

3 **Remark**

Since δ has a codomain of Q, this is a deterministic finite automata, i.e. there is only one possible "next" output given a state and a symbol.

4 **Definition**

We say that a finite automaton $M = (Q, \Sigma, \delta, q_0, F)$ **accepts** a string (a finite sequence of symbols in Σ) $w = w_1 w_2 \dots w_n$, where $w_i \in \Sigma$ if there exists a sequence of states $r_0, r_1, \dots, r_n$ where

- $r_0 = q_0$,
- $r_n \in F$, and
- $\delta(r_{i-1}, w_i) = r_i$ for $1 \leq i \leq n$.

5 **Definition**

The **language** of M is $L(M) = \{w \mid M \text{ accepts } w\}$. Then, we say that M **recognizes** $L(M)$.
If $A = L(M)$ for some finite automaton M, then A is a **regular** language.

6 **Example** (Regular Languages)

- $A = \{w \mid \texttt{11} \text{ is a substring of } w\}$ is regular.
- $B = \{w \mid w \text{ starts and ends with the same symbol}\}$ is regular.
- $C = \{w \mid w \text{ has an equal number of } \texttt{0}\text{s and } \texttt{1}\text{s}\}$ is regular.

1.1.2 Operations

We can also define operations on languages. Let A and B be languages. Then, the **union** of A and B is

$$A \cup B = \{w \mid w \in A \text{ or } w \in B\}.$$

The **concatenation** of A and B is

$$A \circ B = \{xy \mid x \in A \text{ and } y \in B\}.$$

We will often omit the circle for convenience. The **star** operator gives

$$A^* = \{x_1 x_2 \dots x_k \mid x_i \in A, k \in \mathbb{N}\}.$$

Note that the empty string ε is in A^* from the case $k = 0$.

7 **Proposition**

If A and B are regular, then so is $A \cup B$.

Proof. First we'll show that $A \cup B$ is regular. Suppose $M_A = (Q_A, \Sigma, \delta_A, q_A, F_A)$ and $M_B = (Q_B, \Sigma, \delta_B, q_B, F_B)$ recognize A and B respectively. We assume that the alphabets are the same, but the proof would be similar albeit messier if they were not the same. The idea is that we can "run both of the machines at the same time". In particular, we will keep track of the states of each machine in ordered pairs.

Our desired finite automaton that accepts $A \cup B$ is $(Q, \Sigma, \delta, q, F)$, where

- $Q = Q_A \times Q_B$,
- $\delta\colon ((a, b), \sigma) \mapsto (\delta_A(a, \sigma), \delta_B(b, \sigma))$,
- $q = (q_A, q_B)$, and
- $F = \{(a, b) \in Q_A \times Q_B \mid a \in F_A \text{ or } b \in F_B\} = (F_A \times Q_B) \cup (Q_A \times F_B)$. □

8 **Remark**

Note that we can prove that $A \cap B \coloneqq \{w \mid w \in A \text{ and } w \in B\}$ is regular with the exact same machine as $A \cup B$ except we replace the final states with $F_A \times F_B$.

1.2 More on Closure of Regular Operations

9 **Proposition**

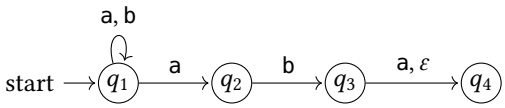
If A and B are regular, then so is $A \circ B$.

We can attempt to prove using the same method as proposition 7. Suppose that $M_A = (Q_A, \Sigma, \delta_A, q_A, F_A)$ accepts A and $M_B = (Q_B, \Sigma, \delta_B, q_B, F_B)$ recognizes B. Then, intuitively, we want to run M_A, and then run M_B on the input. However, it's not clear how the new machine can decide when to split the input.

In particular, for a given DFA M, there is only one possible path that M will take for a specific input. So M is not able to test all potential split points.

1.3 Nondeterministic Finite Automata

To fix this problem, let's allow the machine to make decisions.



The idea is that at a particular state (e.g. q_1), there may be multiple edges that the machine could take (e.g. back to q_1 or to q_2 when the input is a). To think about a path that the machine takes while going through an input, we keep track of a subset of the states that the machine could be at. Moreover, if there exists an arrow exiting with an ε, then we include both the current state as well as the state that the ε arrow goes to, as the machine can be at both.

10 **Example**

In the machine described by the diagram above, ab and aba are accepted, while ba and abaa are rejected.

11 **Definition**

A **nondeterministic finite automata** (NFA) is tuple $(Q, \Sigma, \delta, q_0, F)$ where

- Q is a finite set of states,
- Σ is a finite alphabet,
- $\delta\colon Q \times \Sigma_\varepsilon \to \mathcal{P}(Q)$ is the transition function,
- $q_0 \in Q$ is the start state, and
- F is the set of accepting states.

where $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$. Then, we say that the NFA **accepts** a string $s = s_1 s_2 \dots s_n \in \Sigma^n$ if there exists a sequence of states $r_0, \dots, r_n$ such that

- $r_0 = q_0$,
- $r_n \in F$, and
- $r_i \in \delta(r_{i-1}, w_i)$ for $1 \le i \le n$.

12 **Proposition**

If N is a NFA, then $L(N)$ is regular.

Proof. Let $N = (Q, \Sigma, \delta, q_0, F)$. We will show that there exists a DFA M that is equivalent to N. The idea (again) is to keep track of the subset of the states that N can be at. Therefore, we let

- $Q' = \mathcal{P}(Q)$,
- $\delta' \colon (R, \sigma) \mapsto \bigcup_{r \in R} \delta(r, \sigma)$,
- $q_0' = \{q_0\}$, and
- $F' = \{q \in Q' \mid q \cap F \neq \varnothing\}$.

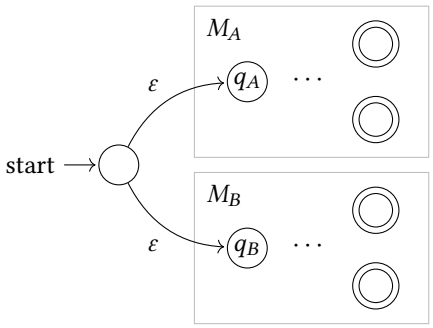
The desired DFA is then $(Q', \Sigma, \delta', q_0', F')$. □

13 **Remark**

We ignored ε in this proof, but the idea is to consider $E(R)$ for $R \in Q'$ to be the set of states that can be reached by following ε arrows, and then let $\delta' \colon (R, \sigma) \mapsto E(\bigcup_{r \in R} \delta(r, \sigma))$.

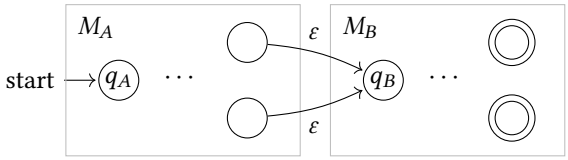
We can then write a new and better proof for one of our old propositions!

Proof (Closure of Regular Languages under $\cup$, Proposition 7). Suppose $M_A = (Q_A, \Sigma, \delta_A, q_A, F_A)$ recognizes A and $M_B = (Q_B, \Sigma, \delta_B, q_B, F_B)$ recognizes B. Then consider the following machine:



Note that this accepts factly the languages in $A \cup B$. □

Proof (Proposition 9). Suppose $M_A = (Q_A, \Sigma, \delta_A, q_A, F_A)$ recognizes A and $M_B = (Q_B, \Sigma, \delta_B, q_B, F_B)$ recognizes B. Then consider the following machine:

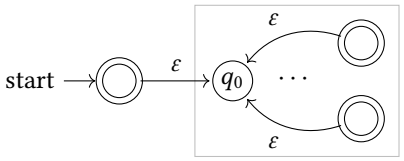


This recognizes AB. □

14 **Proposition**

Let A be a regular language. Then A^* is regular.

Proof. Let $M = (Q, \Sigma, \delta, q_0, F)$ recognize A. Then consider the following machine:



This accepts A^*. Note that we did not make q_0 accepting, but rather added a new start state. This is because within M, there may be a sequence in A that ends on q_0. □

1.3.1 Regular Expressions

Suppose we have an alphabet Σ. We want to find an easier way to describe languages. In particular, we can use smaller languages to build more complicated languages. We can use regular operations to do this.

A regular expression is an expression that uses the regular operations that build on languages. For example, we want something like $(\texttt{01} \cup \texttt{0})^*$ to be a regular expression.

15 **Definition**

An expression R is a **regular expression** if either:

- $R \in \Sigma \cup \{\varepsilon, \varnothing\}$,
- $R = R_1 \cup R_2$ where R_1 and R_2 are languages,
- $R = R_1 \circ R_2$ where R_1 and R_2 are languages, or
- $R = R_0^*$ where R_0 is a language.

Note that each of the atomic regular expressions represent languages:

- a represents $\{a\}$ for all $a \in \Sigma$
- ε represents $\{\varepsilon\}$
- $\varnothing$ represents $\varnothing$.

16 **Remark**

The expressions ε and $\varnothing$ do not represent the same language! ε is the language that contains the empty string, and $\varnothing$ is the empty language.

17 **Example**

Consider the regular expression $R = (\texttt{01} \cup \texttt{0})^*$. We have $\texttt{01} \cup \texttt{0} \longrightarrow \{\texttt{01}, \texttt{0}\}$, so R represents the language

$$\{\varepsilon, \texttt{0}, \texttt{01}, \texttt{00}, \texttt{001}, \texttt{0100}, \ldots\}.$$

This is the language of all strings such that whenever a $\texttt{1}$ appears, there is a $\texttt{0}$ before it.

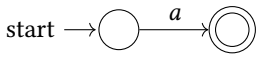
1.4 More on Regular Expressions

18 **Proposition**

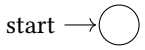
The set of languages that regular expressions determine is exactly all regular languages.

Proof (Regular expressions $\implies$ regular). It is fairly clear that all regular expressions determine regular languages. In particular, for the atomic regular expressions, we can find the NFAs:

- a for $a \in \Sigma_\varepsilon$



- $\varnothing$

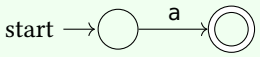


For the composite regular expressions, we just can use the regular operations for NFAs. □

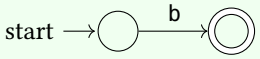
19 **Example**

If we want to build an NFA for $(\mathtt{a} \cup \mathtt{ab})^*$, we can just build it up from smaller parts:

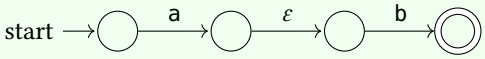
- $\mathtt{a}$



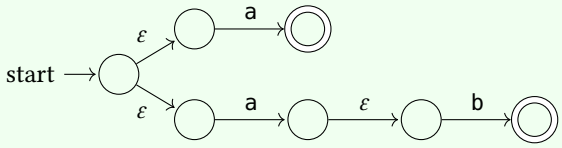
- $\mathtt{b}$



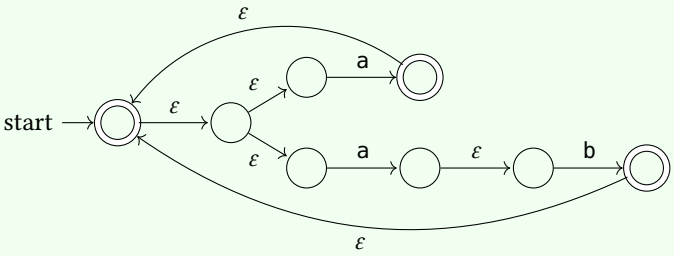
- $\mathtt{ab}$



- $\mathtt{a} \cup \mathtt{ab}$



- $(\mathtt{a} \cup \mathtt{ab})^*$



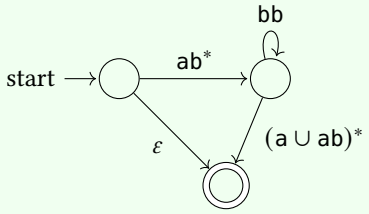
To prove that all NFAs can be represented by a regular expression, we will define another version of them that will make things more convenient.

20 **Definition**

A **generalized nondeterministic finite automaton (GNFA)** is an NFA that can have regular expressions as transition labels.

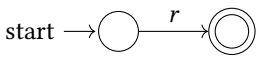
21 **Example**

This is a GNFA, and it accepts abbbabab:



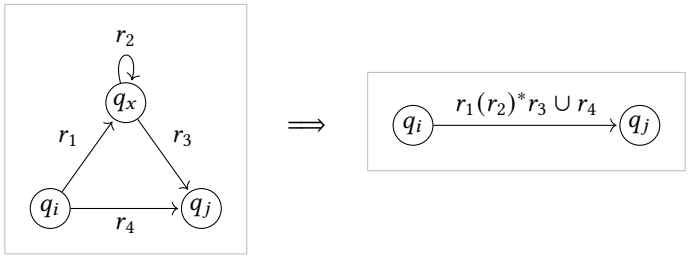
Proof (GNFA $\implies$ regex). Suppose we have a GNFA. For convenience, assume that the starting state is not accepting. (We can just add another starting state with an ε arrow pointing to it otherwise.) Also, assume that there are only arrows out of the initial state, and there are only arrows going into a single accepting state. We can ensure this by adding ε arrows.

We proceed by induction. The smallest machine that satisfies these conditions is one that has $k = 2$ states, and it looks like this



where r is some regular expression. This is the regular expression for this GNFA.

Now suppose that all GNFAs with at most $(k-1)$ states has an equivalent regular expression. Consider some GNFA G with k states. Our goal is to make a new GNFA with $k-1$ states. Let's remove some state g_x. For all states $q_i, q_j \neq g_x$ in G, we an make the following change to fix the machine:



This gives a machine with $k-1$ states, and it accepts the same language, as desired. □

1.5 Non-regularity

Let $B = \{w \mid w \text{ has an equal number of as and bs}\}$ and $C = \{w \mid w \text{ has an equal number of abs and bas substrings}\}$. It turns out that B is not regular but C is regular. It's not obviously clear how to determine this, so it would be nice to have a method to determine whether a language is regular or not.

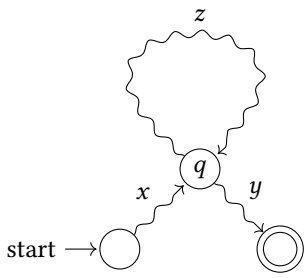
22 **Proposition** (Pumping Lemma)

If A is a regular language, there exists some $p \in \mathbb{N}$ called the **pumping length** such that if $s \in A$ with $|s| \geq p$, we can write $s = xyz$ where

- $xy^iz \in A$ for $i \in \{0, 1, 2, \ldots\}$,
- $y \neq \varepsilon$, and
- $|xy| \leq p$.

Proof. Let M be a DFA for a regular language A, and let p be the number of states of M. Let $s \in A$ have length $|s| \geq p$.

Consider the states that the machine will visit while going through the string. Since $|s| \geq p$, the machine will visit at least $p+1$ states. By the pigeonhole principle, the machine will visit some state q at least twice.



Then, that means there is a nonempty string y that transitions from state q to q, so we can repeat is as many times as possible. □

We can now use the pumping lemma to prove things!

23 **Claim**
Let $D = \{\mathtt{a}^k\mathtt{b}^k \mid k \in \mathbb{N}\}$. Then D is not regular.

Proof. Assume for the sake of contradiction that D is regular. Then let the pumping length be p, and consider $s = \mathtt{a}^p\mathtt{b}^p \in D$. By the pumping lemma, we can write $s = xyz$ where $y = \mathtt{a}^n$ for some $n > 0$. Then, this gives $\mathtt{a}^{p+kn}\mathtt{b}^p \in D$ for all $k \in \{0, 1, \ldots\}$, which is a contradiction. □

1.6 Context-Free Grammars

We will introduce a stronger model of computation.

24 **Example**
Consider a set of substitution rules like the following:

$$S \to \mathtt{0}S\mathtt{1}$$
$$S \to R$$
$$R \to \varepsilon$$

We have capital letters designating variables, and numbers or lowercase letters designating terminals. Start with the starting variable (usually the first one in the first line), and then make substitutions according to the rules.
For instance, we can have

$$S \to \mathtt{0}S\mathtt{1} \to \mathtt{00}S\mathtt{11} \to \mathtt{00}\varepsilon\mathtt{11} = \mathtt{0011}$$

Then, the language that this grammar G_1 describes is $L(G_1) = \{\mathtt{0}^k\mathtt{1}^k \mid k \geq 0\}$.

25 **Example**
Consider the grammar

$$E \to E + T \mid T$$
$$T \to T \times F \mid T$$
$$F \to (e) \mid \mathtt{a},$$

where the variables are $\{E, T, F\}$ and the terminals are $\{(,), +, \times, \mathtt{a}\}$.

One string that this grammar can generate is

$$E \to E + T \to T + T \to T + T \times F \to F + F \times \mathtt{a} \to \mathtt{a} + \mathtt{a} \times \mathtt{a}.$$

26 **Definition**

A **context-free grammar (CFG)** is $G = (V, \Sigma, R, S)$ where

- V is a finite set of variables,
- Σ is a finite set of terminals
- R is a finite set of rules, and
- $S \in V$ is a start variable.

If u, v, w are strings of variables and terminals, and $A \to w$ is a rule, we write $uAv \Rightarrow uwv$ (read uAv **yields** uwv). In particular, $u \Rightarrow v$ if we can get from u to v with one substitution.

We also write $u \overset{*}{\Rightarrow} v$ (read u **derives** v) if there exists a chain

$$u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \cdots \Rightarrow v.$$

Then, the **language** of a grammar G is

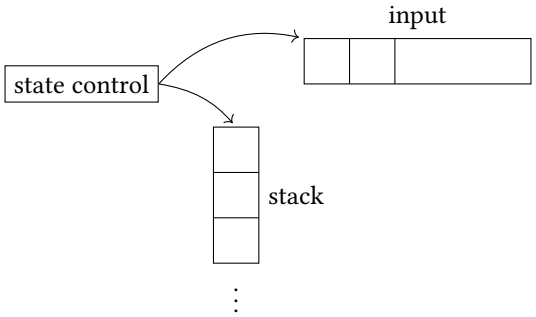
$$L(G) = \{w \mid S \overset{*}{\Rightarrow} w\}.$$

1.7 Pushdown Automata

Context-free grammars for context-free languages are similar to regular expressions for regular languages in the sense that they both generate strings in the language. DFAs and NFAs work in the opposite way, i.e. they recognize strings in the language.

We'll introduce a machine that can recognize context-free languages in the same way that DFAs and NFAs recognize regular languages.

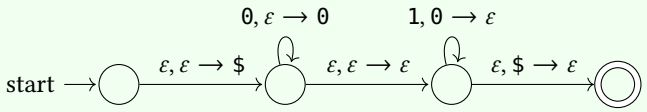
The idea is that there is a state control, which contains the states and transitions that we are familiar with. In addition to reading the input, the states can also control a stack, which is an object that has an infinite amount of memory to store things. The catch is that the machine can only read and write to the top of the stack.



27 **Example**

There exists a PDA that recognizes $\{\mathtt{0}^k\mathtt{1}^k\}$.

Consider the PDA that pushes a `0` onto the stack when it reads a `0` at the beginning, and pops a `0` from the stack when it reads a `1` after the string of `0`s. This accepts $\{\mathtt{0}^k\mathtt{1}^k\}$.



28 **Definition**

A **pushdown automata (PDA)** is $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ where

- Q is finite set of states,
- Σ is a finite input alphabet,
- Γ is a finite stack alphabet,
- $\delta \colon Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to \mathcal{P}(Q \times \Gamma_\varepsilon)$ is the transition function,
- $q_0 \in Q$ is the start state, and
- $F \subseteq Q$ is the set of accepting states.

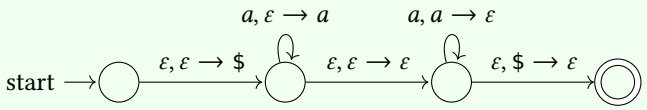
Then, we say that the pushdown automata $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ accepts some input $w = w_1 w_2 \dots w_n \in \Sigma^n$ if there exists a sequence of states $r_0, \dots, r_n$ and a sequence of strings $s_0, \dots, s_n \in \Gamma^*$ such that

- $r_0 = q_0$,
- $s_0 = \varepsilon$,
- $r_n \in F$,
- $(q_i, b) \in \delta(q_{i-1}, w_i, a)$, $s_{j-1} = at$, and $s_j = bt$ for some $a, b \in \Gamma_\varepsilon$ and $t \in \Gamma_\varepsilon^*$ for $1 \le i \le n$, and
- $r_n \in F$.

29 **Example**

There exists a PDA that recognizes $B = \{ww^R \mid w \in \{0,1\}^*\}$, where w^R is the reverse of w.

Let $\Gamma = \Sigma$, and let the PDA push the characters it reads onto the stack. Then at an arbitrary time, the PDA starts popping the read characters from the stack. By nondeterminism, this should work.



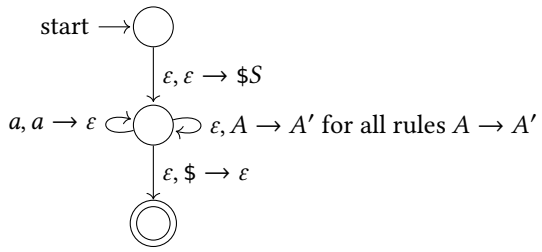
The reason why we push a $ onto the stack at the beginning is so that the machine has a way to tell if it is at the bottom of the stack. This is a common idea that will appear often.

1.8 CFGs and PDAs

30 **Proposition**

A language A is a CFL if and only if $A = L(P)$ for some pushdown automata P.

Proof (CFG → PDA). The idea is that we can simulate a CFG using the stack to remember the intermediate strings. Initially, we insert the starting variable in the stack, and then when we see a variable at the top of a stack, we can replace it with any of the variable's rules. When we see a terminal at the top of the stack, we compare it with the input. If the CFG can generate the string, then eventually, we will remove everything in the stack. Overall, we have the following PDA:



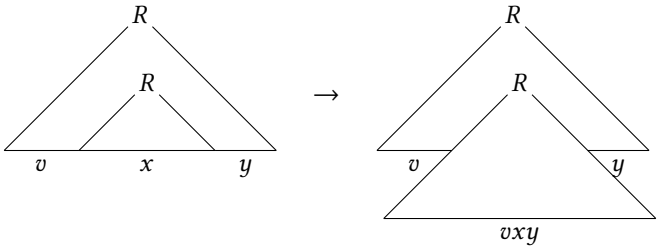
□

31 **Proposition** (Pumping Lemma for CFLs)

Given a CFL A, there exists a **pumping length** p such that for all $s \in A$ where $|s| > p$, we can write $s = uvxyz$ such that

- $uv^ixy^iz \in A$ for all $i \in \{0, 1, \dots\}$,
- $|vy| > 0$, and
- $|vxy| \le p$.

Proof. The idea is that given a CFG G, there is some maximum rule length, so given some final string length, we can find a lower bound for the parse tree height. Therefore, for a sufficiently large final string length, we have a sufficiently tall parse tree.



If $n = |V|$ is the number of variables, then consider a path from the initial variable to some terminal that has more than n variables. By pigeonhole, there exists some variable R that occurs more than once. Let the final result of the later R be x, and the earlier R be vxy. Then we can replace the later R with vxy, and using a similar strategy, we can replace the earlier R with any v^ixy^i.

More formally, if b is the longest length of the right sides of all rules, then for a tree of height h, the string length s satisfies $s \le b^h$. This means if we let $p = b^{|V|+1}$, we must have $h \ge |V| + 1$, and there must be some repeated variable.

Another thing we might need to worry about is if we have some rules of the form

$$R \to S \to \cdots \to R.$$

To fix this, we can just say that given some string $s \in A$, we will take the shortest parse tree for s, i.e. we don't waste any time with rules that don't do anything. □

32 **Corollary**

The language $B = \{\mathsf{a}^k\mathsf{b}^k\mathsf{c}^k\}$ is not a CFL.

Proof. For the sake of contradiction suppose B is regular. Then consider $s = \mathsf{a}^p\mathsf{b}^p\mathsf{c}^p \in B$, where p is the pumping length. By the pumping lemma, we can write $\mathsf{a}^p\mathsf{b}^p\mathsf{c}^p = uvxyz$ such that $|vxy| \le p$. In particular, this means that v and y can only contain two distinct letters. Therefore the pumping lemma states that $uxz \in B$. However, we know that at least one of the number of as, bs, and cs must have changed, but not all of them, which is a contradiction. □

33 **Corollary**

The language $C = \{ww \mid w \in \{\mathtt{a}, \mathtt{b}\}^*\}$ is not context free.

Proof. For the sake of contradiction, suppose that C is regular. Then consider $s = \mathtt{a}^p\mathtt{b}^p\mathtt{a}^p\mathtt{b}^p \in C$, where p is the pumping length. By the pumping lemma, we can write $s = uvwxy$ where $|vwy| \leq p$. Therefore, vwy cannot contain $\mathtt{a}$s from both strings, and also cannot contain $\mathtt{b}$s from both strings. Therefore, when we pump, we will be changing the number of $\mathtt{a}$s (or $\mathtt{b}$s) in one of the ws but not the other. □

2 Computability Theory

2.1 Turing Machines

We would like a model of model of computation that is stronger than this. In particular, it will be similar to a pushdown automata, except instead of a stack, the machine reads and writes to an unbounded tape that initially contains the input, that it can move left and right on, and it accepts/rejects by entering special halting states.

34 **Example**

Recall the language $B = \{\mathtt{a}^k\mathtt{b}^k\mathtt{c}^k\}$. We claim that there is a Turing machine for B.

The idea is that on the first pass (staring from the leftmost position), the machine tests to make sure that the input is of the form $\mathtt{a}^*\mathtt{b}^*\mathtt{c}^*$. Then, after that we can make multiple passes, each one replacing one $\mathtt{a}$, one $\mathtt{b}$, and one $\mathtt{c}$ with a blank space each time.

35 **Definition**

A **Turing machine (TM)** is a tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{reg}})$ such that

- Q is the set of states,
- Σ is the input alphabet not containing $\textvisiblespace$,
- Γ is the tape alphabet with $\textvisiblespace \in \Gamma$ and $\Sigma \subseteq \Gamma$,
- $\delta\colon Q \times \Gamma \to Q \times \Gamma \times \{\mathrm{L}, \mathrm{R}\}$ is the transition function,
- $q_0 \in Q$ is the start state,
- $q_{\text{accept}} \in Q$ is the accept state, and
- $q_{\text{reject}} \in Q$ is the reject state, where $q_{\text{accept}} \neq q_{\text{reject}}$.

The Turing machine begins its computation with the input on the leftmost part of the tape, and the rest of the tape is filled with $\textvisiblespace$. The read/write head starts on the leftmost space of the tape.

As the Turing machine computes, the state, tape contents, and head location changes. A combination of these three items is a **configuration** of the Turing machine. We represent them in a specific way. If the current state is q, the tape contains $uv \in \Gamma^*$, and the tape head is on the first character of v, then we say that the configuration is uqv.

We will say that a configuration C_1 yields C_2 if the Turing machine can go from C_1 to C_2 in one step. More formally, for all $u, v \in \Gamma^*$, if for some $a, b, c \in \Gamma$ we have $\delta(q_i, b) = (q_j, c, \mathrm{L})$, then

$$uaq_ibv \quad \text{yields} \quad uq_iacv,$$

and if $\delta(q_i, b) = (q_j, c, \mathrm{R})$, then

$$uaq_ibv \quad \text{yields} \quad uacq_jv.$$

If the head is at the left end, then receiving an L will not move the head.

The **starting configuration** of M on input w is the configuration q_0w. An **accepting configuration** is one where the state is q_{accept}, and a **rejecting configuration** is one where the state is q_{reject}. We call accepting and rejecting configurations **halting**

configurations, and do not yield any configurations. If the machine never halts, then we say that it **loops**.

Then, we say that M accepts a string w if there exists configurations $C_1, \dots, C_k$ such that $C_1 = q_0 w$, C_i yields C_{i+1}, and C_k is an accepting configuration. Then the **language of M** is

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

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

36 **Definition**

A language L is **Turing-recognizable** if there exists some Turing machine M such that $L = L(M)$.

A machine M is a **decider** if it always halts on all inputs. We then say that M **decides** $L(M)$.

Moreover, a language L is **Turing-decidable** if there exists some Turing machine that decides L.

2.2 Variants of Turing Machines

37 **Definition**

A **multitape Turing machine** is the same as a Turing machine, but there are some finite number k of tapes, and the transition function is

$$\delta \colon Q \times \Gamma^k \to Q \times \Gamma^k \times \{\mathrm{L}, \mathrm{R}\}^k.$$

The input is initially on the first string, while the others are blank.

38 **Proposition**

A language A is $L(M)$ for some multitape TM M if and only if A is Turing-recognizable.

Proof. The proof of the reverse direction is immediate.

To prove the forward direction, we want to take a given multitape TM M and convert it to a single tape TM. The idea is to keep track of all of the tapes on one tape, delimited by #s. Then, for every step that M takes, we find what multitape state the single tape TM corresponds to and update each tape accordingly.

More formally, let the alphabet be $\Gamma' = \Gamma \cup \{\dot{\gamma} \mid \gamma \in \Gamma\}$,

1. If the input is $w = w_1 w_2 \dots w_n$, initialize the tape with
$$\dot{w}_1 w_2 \dots w_n \# \dot{\sqcup} \# \dot{\sqcup} \# \dots \# \dot{\sqcup} \# \dot{\sqcup} \# \dots.$$
2. For each step that M takes:
   a) Scan tape to get the symbols that M is currently on, marked by dots.
   b) Update tape according to the transition function of M.
3. Repeat until accepted or rejected. Or just let it loop forever. □

39 **Definition**

A **nondeterministic Turing machine (NTM)** is the same as a standard Turing machine, but the transition function is

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

40 **Proposition**

A language A is $L(N)$ for some NTM if and only if A is Turing recognizable.

Proof (Idea). The proof of the reverse direction is immediate.

To build a Turing machine from a NTM, we just perform breadth first search on the configurations. Note that we should not use depth first search because if a branch loops forever, then we will never explore all of the branches of nondeterminism. □

41 **Definition**

An **enumerator** is a Turing machine that generates strings. One way to think about them is that they have two tapes, T_{compute} and T_{print}, and whenever the machine reaches a special state q_{print}, the machine will print everything on the tape T_{print}. The language of the machine is then the set of everything that the machine prints.

42 **Proposition**

A language A is a language for some enumerator if and only if A is Turing recognizable.

Proof. We will first prove that if we have an enumerator E with $A = L(E)$, then there exists a TM that recognizes it. In particular, the machine M works as follows:

0. On input w:
1. Run E. If E ever prints w, then accept.

This will accept all of the strings in A.

Now suppose that we're given a language A that some TM M recognizes. We want to construct an enumerator for A. To do so, note that Σ^* is countable, so let $\Sigma^* = \{s_1, s_2, \dots\}$. Then, we have the following enumerator:

1. For $i = 1, 2, \dots$:
   a) Start M_i, a copy of M, on s_i
   b) Let all of $M_1, \dots, M_i$ compute one more step.
   c) If any of the M_is accept, print s_i.

If M accepts s_k, then the enumerator will eventually print it. Moreover, this does not loop forever on one input. □

We've been describing our Turing machines with words instead of their states and transition functions. It turns out that this is fine, because of the Church-Turing Thesis.

43 **Claim** (Church-Turing Thesis)

Our intuitive idea of what an algorithm can compute is precisely what Turing machines can compute.

44 **Example**

If we have the language

$$D = \{p \in \mathbb{Z}[x_1, \dots, x_k] \mid p(a_1, \dots, a_k) = 0 \text{ for some } a_1, \dots, a_k \in \mathbb{Z}\},$$

then D is not decidable because we don't have a good (decidable) algorithm for it.

In particular, we can find a TM that recognizes D, but not one that decides D. An easy way to see this is to just test all elements in $\mathbb{Z}^k$ in order.

45 **Example**

Let $A = \{\langle G \rangle \mid G \text{ is a finite connected graph}\}$. Then A is decidable.

An easy way to see this is to just run a connected component algorithm on G, and then test whether the first connected component is the same as the whole graph. The implementation is as follows:

0. On input $\langle G \rangle$:
1. Select some node of G and mark it.
2. Repeat the following:
   a) For each unmarked node of G, mark it if it is connected to a marked node.
   b) If no new nodes were marked, stop this loop.
3. If all nodes are marked ACCEPT, otherwise REJECT.

2.3 Closure Properties

We now have many models of computation that give us many classes of languages. Let's go over their closure properties.

class	A^*	$A \circ B$	$A \cup B$	$A \cap B$	$\overline{A}$	$A \setminus B$
regular	yes	yes	yes	yes	yes	yes
context-free	yes	yes	yes	no[1]	no	no
decidable	yes	yes	yes	yes	yes	yes
recognizable	yes	yes	yes	yes	no	no

2.4 Decision Problems

Sometimes, we want to work with objects that are not strings. If A is an object, then let's say that $\langle A \rangle$ is an **encoding** of A into a string over Σ. If we have multiple objects $A_1, \ldots, A_k$, then we can encode all of them into one string using $\langle A_1, \ldots, A_k \rangle$. Note that $\langle A_1, \ldots, A_k \rangle$ is not necessarily just $\langle A_1 \rangle, \ldots, \langle A_k \rangle$ concatenated, because it might not be clear where each one ends.

2.4.1 DFAs

Acceptance Problem

Given a DFA, we would like to determine whether it will accept some string w. In particular, let

$$A_{\text{DFA}} = \{\langle B, w \rangle \mid B \text{ is a DFA that accepts } w\}.$$

Therefore, if testing if $a \in A$ is equivalent to determining if some DFA accepts some word.

46 **Proposition**

A_{DFA} is decidable.

Proof. Let M be the following Turing machine with input $s = \langle B, w \rangle$:

- Test if $s = \langle B, w \rangle$ rejects or not.
- Simulate B on W.
- If B accepts, ACCEPT, else REJECT. □

We can define a similar language for NFAs.

47 **Corollary**

$A_{\text{NFA}} = \{\langle B, w\rangle \mid B \text{ is a NFA that accepts } w\}$ is decidable.

To prove this, we can just do the same thing as above, and simulate the NFA. However, we can do something clever and just reduce it to the previous problem.

Proof (Sketch). Convert B to a DFA B' and then use the fact that A_{DFA} is decidable. □

Emptiness Problem

48 **Proposition**

Let $E_{\text{DFA}} = \{\langle B\rangle \mid B \text{ is a DFA where } L(B) = \varnothing\}$. Then E_{DFA} is decidable.

Proof. The idea is to just perform a graph search on the states. Consider the TM M with input $\langle B\rangle$, described by:

- Mark the start state
- Repeat until nothing new is marked:
  - Mark state q if some previously marked state points to q.
- Accept if no accept node is marked, otherwise reject. □

Equivalence Problem

49 **Proposition**

Let $EQ_{\text{DFA}} = \{\langle B, C\rangle \mid B, C \text{ are DFAs and } L(B) = L(C)\}$. Then EQ_{DFA} is decidable.

Proof. The key idea is that if $L(B) = L(C)$, then we know that the symmetric difference is empty, i.e.

$$L(B) \,\Delta\, L(C) = \overline{L(B)} \cap L(C) \cup L(B) \cap \overline{L(C)} = \varnothing.$$

Therefore, we consider the following TM with input $\langle B, C\rangle$:

- Let D be the DFA such that
$$L(D) = \overline{L(B)} \cap L(C) \cup L(B) \cap \overline{L(C)}.$$
- Test if $\langle D\rangle \in E_{\text{DFA}}$, i.e. if $L(D)$ is empty. If yes then accept; reject if no. □

2.4.2 CFGs

50 **Proposition**

Let $A_{\text{CFG}} = \{\langle G, w\rangle \mid G \text{ is a CFG such that } w \in L(G)\}$. Then A_{CFG} is decidable.

Proof. Consider the following TM with input $\langle G, w\rangle$:

- Convert G to Chomsky's Normal form. We know that to derive a string of length n, it will take $2n - 1$ steps ($n - 1$ to get the correct number of symbols, and n to convert variables).
- Try all the derivations of length $2n - 1$. If any of them yield w, then accept, else reject. □

51 **Proposition**
Every CFL is decidable.

Proof. Let A be a CFL generated by the CFG G. Consider the TM M_G with input w:

- Test if $\langle G, w\rangle \in A_{\text{CFG}}$.
- If yes then ACCEPT; REJECT if no. □

52 **Proposition**
Let $E_{\text{CFG}} = \{\langle G\rangle \mid G \text{ is a CFG and } L(G) = \varnothing\}$. Then E_{CFG} is decidable.

Proof. Suppose we are given a list of rules. We can mark all the terminals, and then if everything on the right side of a rule is marked, then this means we can get the left side. Repeat until nothing new is marked. Then, if the initial variable is marked then ACCEPT; else REJECT. □

53 **Proposition**
Let $EQ_{\text{CFG}} = \{\langle G, H\rangle \mid G, H \text{ are CFGs and } L(G) = L(H)\}$. Then EQ_{CFG} is undecidable.

We can't use the same trick of using the symmetric difference we used last time for DFAs, because CFGs are not closed under complements or intersection! We'll prove this proposition later.

54 **Proposition**
Let $A_{\text{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM accepting } w\}$. Then A_{TM} is Turing recognizable.

Proof. The idea is to just make the machine do it. In particular, the following machine recognizes A_{TM}:

0. On input $\langle M, w\rangle$:
1. Simulate M on w.
2. If M accepts, then ACCEPT, if M rejects, then REJECT. □

2.5 Decidability of A_{TM} and Friends

55 **Theorem**
$A_{\text{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM accepting } w\}$ is not decidable.

Proof. Suppose for the sake of contradiction that there exists a TM H that decides A_{TM}. Let's construct another TM D that does the following:

0. On input $\langle M\rangle$, where M is a TM:
1. Run H on $\langle M, \langle M\rangle\rangle$.
2. If H accepts then REJECT, else ACCEPT.

Intuitively, D accepts $\langle M\rangle$ if and only if M on $\langle M\rangle$ does not accept.
Now consider running D on $\langle D\rangle$. Then D accepts if and only if D on $\langle D\rangle$ does not accept. This is a contradiction. □

The motivation comes from Cantor's diagonalization argument. In particular, consider the table:

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\cdots$	$\langle D \rangle$
M_1	ACC	REJ	ACC	$\cdots$	
M_2	ACC	ACC	ACC	$\cdots$	
M_3	REJ	REJ	REJ	$\cdots$	
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
D	REJ	REJ	ACC	$\cdots$	?

The TM D gives the opposite of the entries on the main diagonal, which yields a problem when we get to the row for D.

56 **Note**

Sometimes we *do* input programs into themselves! For example, compilers.

57 **Theorem**

If A and $\overline{A}$ are both Turing recognizable, then A is decidable.

Proof. Let R be a recognizer for A and S be a recognizer for $\overline{A}$. Then, consider the following TM:

0. On input string w:
1. Start running R and S on w.
2. Let R and S alternate taking steps until one accepts.
3. If R accepts, then ACCEPT, and if S accepts, then REJECT.

Note that this is a decider because one of R and S must accept, halting the TM. □

58 **Corollary**

$\overline{A_{\mathsf{TM}}}$ is not Turing recognizable.

2.6 Reducibility

2.6.1 Motivation

59 **Proposition**

The language $HALT_{\mathsf{TM}} := \{\langle M, w \rangle \mid \mathsf{TM}\ M \text{ halts on } w\}$ is undecidable.

We can use the idea of reducing the problem A_{TM} to $HALT_{\mathsf{TM}}$, so that if we prove that $HALT_{\mathsf{TM}}$ is decidable, then A_{TM} is decidable, which is a contradiction.

Proof. Assume for the sake of contradiction that a TM R decides $HALT_{\mathsf{TM}}$. We will construct a TM S that decides A_{TM}. In particular, let S be the Turing machine:

0. On input $\langle M, w \rangle$.
1. Run R on $\langle M, w \rangle$.
2. If R rejects, then M loops on w, so REJECT.
3. If R accepts, then M halts on w, so run M.
4. If M accepts then ACCEPT, otherwise M rejects so REJECT.

Therefore, we have a TM that decides A_{TM}, which contradicts the fact that A_{TM} is undecidable. □

If we can reduce a problem A to another problem B, then this means if we can solve B, then we can solve A. Equivalently, if we can't solve A, then we can't solve B. In particular, the "can't solve" here can be replaced with "undecidable". Then, we can use this to show that a problem X is undecidable by showing that A_{TM} reduces to X.

60 **Proposition**

The problem $E_{\mathsf{TM}} = \{\langle M \rangle \mid M \text{ is a TM and } L(M) = \varnothing\}$ is undecidable.

Proof. The idea is to reduce A_{TM} to E_{TM}.

For the sake of contradiction, suppose that some TM R decides E_{TM}. Then consider the following TM S:

0. On input $\langle M, w \rangle$:
1. Construct a new TM M_w that does the following:
   (0) On input x:
   (1) If $x \neq w$ then REJECT.
   (2) Run M on x.
   (3) If M accepted then ACCEPT, if M rejects then REJECT.
2. Run R on $\langle M_w \rangle$.
3. If R accepts, then $M_w = \varnothing \implies w \notin L(M)$ so REJECT.
4. If R rejects, then $M_w \neq \varnothing \implies M_w = \{w\} \implies w \in L(M)$, so ACCEPT.

This decides A_{TM}, which is a contradiction. □

2.6.2 Formal Definition

61 **Definition**

A function f is **computable** if there exists some TM F such that F on input w halts with $f(w)$ on the tape.

62 **Definition**

Given languages A and B (over the same alphabet Σ), A is **mapping reducible** to B (denoted $A \leq_m B$) if there exists a computable function $f : \Sigma^* \to \Sigma^*$ such that for all $w \in \Sigma^*$,

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

Then, we call f the **reduction** from A to B.

The intuitive idea of what mapping reducible ($A \leq_m B$) means is that A is an easier problem than B. In particular, if we can solve B, then we can solve A, because to determine if $w \in A$, we can just determine if $f(w) \in B$.

We can show that this definition of reducibility has the feature we want. In particular, suppose that $A \leq_m B$ and B is decidable. Then let R be a TM that decides B. Then we can construct a machine that decides A:

0. On input w:
1. Compute $f(w)$.
2. Give the same answer as R on $f(w)$.

Note that if $f(w) \in B \iff w \in A$, then R will accept, and similarly for $f(w) \notin B$. Therefore, A is also decidable.

Moreover, if $w \in A \iff f(w) \in B$, we also have $w \in \overline{A} \iff f(w) \in \overline{B}$, so $A \leq_m B \iff \overline{A} \leq_m \overline{B}$.

63 **Example**

We have $A_{\mathsf{TM}} \leq_m HALT_{\mathsf{TM}}$.

It suffices to give a reduction f. Let

$$f(\langle M, w\rangle) = \langle M', w\rangle,$$

where M' is the TM that is the following:

0. On w:
1. Run M on w.
2. If M accepts, then ACCEPT.
3. If M rejects, then loop.

Note that if $\langle M, w\rangle \in A_{\mathsf{TM}}$, then M' will accept w (i.e. halt), and $\langle M', w\rangle \in HALT_{\mathsf{TM}}$. If $\langle M, w\rangle \notin A_{\mathsf{TM}}$, then M' will loop, so $\langle M', w\rangle \notin HALT_{\mathsf{TM}}$.

64 **Example**

Let $EQ_{\mathsf{TM}} = \{\langle M, N\rangle \mid M \text{ and } N \text{ are TMs with } L(M) = L(N)\}$. Then EQ_{TM} and $\overline{EQ_{\mathsf{TM}}}$ are not Turing recognizable.

First we will show $\overline{A_{\mathsf{TM}}} \leq_m EQ_{\mathsf{TM}}$. It suffices to find a reduction, and here it is:

$$f(\langle M, w\rangle) = \langle M_1, M_2\rangle,$$

where M_1 is the machine that runs M on w and M_2 is the machine that always rejects.
Similarly, we can show that $\overline{A_{\mathsf{TM}}} \leq_m \overline{EQ_{\mathsf{TM}}}$ by using the same map except M_2 is now the machine that always accepts.

Recall that to show that B is undecidable, we can reduce A_{TM} to B.

Also, recall the set $D = \{\langle p\rangle \mid p \text{ is a multivariable polynomial that has integer roots}\}$. The question whether D is decidable or not is Hilbert's 10th problem.

65 **Proposition** (1971)

D is not decidable.

Proof (Sketch). Reduce A_{TM} to D. □

This reduction unfortunately is very complicated, so we will not go over it here. We can, however find another problem that is undecidable.

2.7 Post Correspondence Problem

Suppose we have dominoes, i.e. pairs of strings:

$$P = \left\{ \begin{bmatrix} u_1 \\ v_1 \end{bmatrix}, \begin{bmatrix} u_2 \\ v_2 \end{bmatrix}, \begin{bmatrix} u_3 \\ v_3 \end{bmatrix}, \ldots, \begin{bmatrix} u_k \\ v_k \end{bmatrix} \right\}.$$

Can we determine if there exists a nonempty tuple $(p_1, \ldots, p_k)$ of dominoes with $p_i \in P$ such that the top strings concatenated are the same as the bottom strings concatenated. Such tuples are called **matches**.

66 **Example**

Suppose that

$$P = \left\{ \begin{bmatrix} \texttt{aa} \\ \texttt{aba} \end{bmatrix}, \begin{bmatrix} \texttt{ab} \\ \texttt{aba} \end{bmatrix}, \begin{bmatrix} \texttt{ba} \\ \texttt{aa} \end{bmatrix}, \begin{bmatrix} \texttt{abab} \\ \texttt{b} \end{bmatrix} \right\}.$$

Then, the tuple

$$\left(\begin{bmatrix} \texttt{ab} \\ \texttt{aba} \end{bmatrix}, \begin{bmatrix} \texttt{aa} \\ \texttt{aba} \end{bmatrix}, \begin{bmatrix} \texttt{ba} \\ \texttt{aa} \end{bmatrix}, \begin{bmatrix} \texttt{aa} \\ \texttt{aba} \end{bmatrix}, \begin{bmatrix} \texttt{abab} \\ \texttt{b} \end{bmatrix} \right)$$

is a match because both the top and bottom are $\texttt{abaabaaaabab}$.

67 **Proposition**

The language $PCP = \{\langle P \rangle \mid P \text{ has a match}\}$ is undecidable.

We will prove this later today. First, we need a lemma:

68 **Lemma**

Given a set of dominoes P and some element $p \in P$, finding a match with first element p is equivalent to PCP.

Proof. If $u = u_1 u_2 \dots u_\ell$ is a string where u_i are characters, let

$$\star u = *u_1 * u_2 * \cdots * u_\ell$$

$$u\star = u_1 * u_2 * \cdots * u_\ell *$$

$$\star u\star = *u_1 * u_2 * \cdots * u_\ell * .$$

Then if

$$P = \left\{ \begin{bmatrix} u_1 \\ v_1 \end{bmatrix}, \begin{bmatrix} u_2 \\ v_2 \end{bmatrix}, \begin{bmatrix} u_3 \\ v_3 \end{bmatrix}, \dots, \begin{bmatrix} u_k \\ v_k \end{bmatrix} \right\},$$

where $p = \begin{bmatrix} u_i \\ v_i \end{bmatrix}$, then consider

$$P' = \left\{ \begin{bmatrix} \star u_1 \\ v_1 \star \end{bmatrix}, \begin{bmatrix} \star u_2 \\ v_2 \star \end{bmatrix}, \begin{bmatrix} \star u_3 \\ v_3 \star \end{bmatrix}, \dots, \begin{bmatrix} \star u_i \\ v_i \star \end{bmatrix}, \begin{bmatrix} \star u_i \\ \star v_i \star \end{bmatrix}, \begin{bmatrix} \star u_i \\ v_i \star \end{bmatrix}, \dots \begin{bmatrix} \star u_k \\ v_k \star \end{bmatrix}, \begin{bmatrix} \star \diamond \\ \diamond \end{bmatrix} \right\}.$$

This forces the element corresponding to p to go first, and then every thing else fits together. The last element is added so that we can actually finish the top string. A match of P' corresponds to a match in P with first element p. □

2.7.1 Linearly Bounded Automata

To prove PCP, we will consider a new kind of automaton to explore the idea of a computation history.

69 **Definition**

A **linearly bounded automaton (LBA)** is a TM whose tape is the size of the input.

70 **Proposition**

The language $A_{\text{LBA}} = \{\langle B, w \rangle \mid \text{LBA } B \text{ accepts } w\}$ is decidable.

Proof (Idea). Run B on w. Since the length of the tape is fixed, there are a finite number of configurations of the TM. Therefore, at some point the machine with either halt or repeat a configuration. If B repeats a configuration, then we know that it will loop.

In particular, if the input is of length n, then there are $N = n|Q||\Gamma|^n$ possible configurations of the LBA, so we can run B on v for $N + 1$ steps. If B is still running after this, then we know it will definitely loop. □

71 **Proposition**

The language $E_{\mathsf{LBA}} = \{\langle B\rangle \mid L(B) = \varnothing\}$ is undecidable.

Proof. We will show that A_{TM} reduces to E_{LBA}. The idea is to consider the possible computation history for a machine. In particular, a computation history is a sequence of configurations

$$C_1 \# C_2 \# C_3 \# \dots \# C_\ell,$$

delimited by #s, where C_i are of the form $t_1 q t_2$, where $t_1 t_2 \in \Gamma^n$ is the content of the tape, and q is the state that the machine is on, inserted into $t = t_1 t_2$ at the position of the head. We will create a machine that determines whether a certain computation history is valid and accepts. Then, we can use E_{LBA} to determine whether there exists a valid computation history that accepts.

Let the TM R decide E_{LBA}. Then consider the following TM S that decides A_{TM}:

0. On input $\langle M, w\rangle$:
1. Consider the LBA $B_{M,w}$ defined by
   a) On input x:
   b) Determine whether x is an accepting computation history for M on w.
   c) If yes then ACCEPT, otherwise REJECT.
2. Run R on $B_{M,w}$.
3. If R rejects then ACCEPT, otherwise, REJECT. □

2.7.2 Undecidability of *PCP*

Now we will prove proposition 67, i.e. *PCP* is undecidable.

Proof (PCP). The idea is to encode the computation history rules into the dominoes P that we can use to prove $A_{\mathsf{TM}} \leq_m PCP$.

Suppose the TM R decides *PCP*. Consider the following machine S that decides A_{TM}:

0. On input $\langle M, w\rangle$, where $w = w_1 w_2 \dots w_n$:
1. Construct the set
$$P_{M,w} = \left\{ \begin{bmatrix} \# \\ \# q_0 w_1 w_2 \dots w_n \# \end{bmatrix}, \begin{bmatrix} \# \\ \# \end{bmatrix}, \begin{bmatrix} q_{\text{accept}} \# \\ \# \end{bmatrix} \right\} \cup \left\{ \begin{bmatrix} a \\ a \end{bmatrix} \;\middle|\; a \in \Gamma \right\}$$
$$\cup \left\{ \begin{bmatrix} qa \\ br \end{bmatrix} \;\middle|\; \delta(q,a) = (r,b,R) \right\} \cup \left\{ \begin{bmatrix} cqa \\ rcb \end{bmatrix} \;\middle|\; \delta(q,a) = (r,b,L) \right\} \cup \left\{ \begin{bmatrix} aq_{\text{accept}} b \# \\ \# \end{bmatrix} \;\middle|\; a, b \in \Gamma^* \right\}$$
2. Run R on $P_{M,w}$.
3. If R accepts then ACCEPT; if R rejects then REJECT.

The idea is to make R create a valid computation history if one exists. At each #, the computation history advances one step. For example, if $\begin{bmatrix} q_0 w_1 \\ aq \end{bmatrix} \in P_{M,w}$, then our sequence of dominoes might start out as

$$\begin{bmatrix} \# \\ \# q_0 w_1 w_2 \dots w_n \# \end{bmatrix}, \begin{bmatrix} q_0 w_1 \\ aq \end{bmatrix}, \dots, \begin{bmatrix} \# \\ \# \end{bmatrix}, \dots$$

which gives the combined domino:

$$\begin{bmatrix} \# q_0 w_1 w_2 \dots w_n \# \\ \# q_0 w_1 w_2 \dots w_n \# a q w_2 \dots w_n \# \end{bmatrix}.$$

The existence of a match in P that starts with $\begin{bmatrix} \# \\ \# q_0 w_1 w_2 \dots w_n \# \end{bmatrix}$ means that we have a valid sequence of configurations that

accept. In particular, if R accepts P then we know that M accepts w, and similarly if R rejects P. □

2.8 More Undecidability

72 **Proposition**
The language $ALL_{\text{CFG}} = \{\langle G\rangle \mid \text{CFG } G \text{ such that } L(G) = \Sigma^*\}$ is undecidable.

Proof. We will reduce A_{TM} to ALL_{CGF} using computation history. The idea is to construct a PDA whose language is all strings except the accepting computation history for M on w.

Suppose that the TM R decides ALL_{CFG}. Consider the following TM S that decides A_{TM}:

0. On input $\langle M, w\rangle$:
1. Construct the PDA $P_{M,w}$ that does the following:
   (0) On input $C_1\#C_2^{\mathcal{R}}\#C_3\#C_4^{\mathcal{R}}\#\ldots\#C_k$:
   (1) Nondeterministically accept if any of the following hold:
   - C_1 is not the starting configuration of M on w,
   - C_k is not an accepting state,
   - for some i, C_i does not yield C_{i+1}.
2. Convert $P_{M,w}$ to $G_{M,w}$.
3. Run R on $G_{M,w}$.
4. If R accepts then REJECT, if R rejects, then ACCEPT.

The way that $P_{M,w}$ works is that on each branch of nondeterminism, $P_{M,w}$ checks each condition for the input to be a valid computation history. To check that C_i yields C_{i+1} for each i, we require the input to have every other C_i be reversed, so that we can compare adjacent C_is. □

2.9 Recursion Theorem

Sometimes we think about machines that can construct replicas of themselves. We see this in biology! In fact, in computing, there exists a TM *SELF* that prints out its own description $\langle SELF\rangle$. To show that we can do this, we need a lemma.

73 **Lemma**
There exists a computable function $q\colon \Sigma^* \to \Sigma^*$ mapping $w \mapsto \langle P_w\rangle$, where P_w prints w.

Proof. This is straightforward. □

Now to construct *SELF*, suppose that it is composed of two TMs A and B that compute in that order. Let A print $P_{\langle B\rangle}$. Then, let B compute the function q as described above on the contents of the tape (which is $P_{\langle B\rangle}$), giving $\langle A\rangle$. Then, prepend $\langle A\rangle$ to the tape. This means that we *SELF* prints $\langle A\rangle\langle B\rangle$, which is just what we want.

We can even implement this in English!

Algorithm
Print out two copies of the following, the second copy in quotes.
"Print out two copies of the following, the second copy in quotes."

This gives us the following theorem:

74 **Theorem**
A TM can obtain its own description.

75 **Corollary**
The language A_{TM} is undecidable.

Proof. Suppose for the sake of contradiction that H decides A_{TM}. Then construct the TM R as follows:

0. On input w:
1. Get its own description $\langle R \rangle$.
2. Run H on $\langle R, w \rangle$.
3. Output the opposite of what H outputs.

This is a contradiction, because if we consider some input w, R cannot accept nor reject, as a contradiction would arise in both cases. □

76 **Proposition**
The language $MIN_{\mathsf{TM}} = \{\langle M \rangle \mid \langle M \rangle \text{ is the shortest description among all equivalent TMs}\}$ is unrecognizable.

Proof. Suppose for the sake of contradiction that MIN_{TM} is recognizable. Then some Turing enumerator E enumerates MIN_{TM}. Consider the following TM C:

0. On input w:
1. Get $\langle C \rangle$.
2. Run E until we get some description $\langle D \rangle$ longer than $\langle C \rangle$.
3. Output what D outputs when run on w.

This gives a contradiction, because D cannot be minimal, as C does the same thing as D. □

3 Complexity Theory

3.1 Introduction to Complexity Theory

From the 1930s to the 1950s, people studied computability theory, which is about determining what languages are recognizable and decidable. Then, from the 1960s to now, people study complexity theory, which studies which problems are decidable easily (in time, space, etc.).

Everything we will be looking at from now on will be decidable. What we will be thinking about is how much resources we need to decide it.

First, some motivating examples:

77 **Example**

Let $A = \{\mathtt{a}^k\mathtt{b}^k\}$. This is decidable because it is context free. We want to figure out how much time (number of steps) it takes to determine whether some string is in A. By convention, we take all the inputs of length n, and consider the input that takes the most time. We bound this function on n, and call this the **worst case time complexity**.

78 **Claim**

$\{\mathtt{a}^k\mathtt{b}^k\}$ can be decided by a 1-tape TM that uses $O(n^2)$ steps for inputs of length n.

Proof. Consider the Turing machine described by the following:

0. On input w:
1. Scan the input to make sure it is of the form $\mathtt{a}^*\mathtt{b}^*$. If not, then REJECT.
2. Repeat until everything is crossed off:
   a) Scan tape and cross off one a and one b.
   b) If the as finish before the bs or vice versa, then REJECT.
3. If the loop finishes, ACCEPT.

This TM indeed takes $O(n^2)$ time. □

Can we do better? The loop in step 2 seems a bit inefficient. Instead of crossing off one a and b at a time, we can try to cross off every other letter. If the number of as we cross off has the same parity as the number of bs we cross off each time, then there will be the same number of a and b. This takes $O(n \log n)$ time. Note that we took out the base in the log because $\log_2 n = \log n / \log 2$, and $\log 2$ is just a constant.

Can we do even better? In particular, can we find a TM that uses $o(n \log n)$?

79 **Note**

We can think of O as $\leq$, o as $<$, and Θ as $=$.

It turns out $O(n \log n)$ is optimal. Proving a certain algorithm is optimal is often hard, so we will not prove it here.

Another idea that we might have is to keep a counter. We cannot store this in the state control because the state control must be finite. Therefore, we must keep it on the tape. Deciding where to put the counter is crucial. If the position is fixed, then we will spend $O(n)$ time travelling to the counter, which is bad. Therefore, we can keep the counter with the head. The counter head will be of length $O(\log n)$, and we will have to move the counter on each step, which takes $O(n \log n)$ time. This gives us the same answer!

80 **Claim**

A 2-tape TM can decide A in $O(n)$ time.

This is clear because while scanning over the as, we can copy them to the second tape, and then compare them to the bs on the first tape.

This shows us that the model of computation we have will affect the complexity! This is bad, because this means our results depend on the model of computation. However, we can fix this by showing that the models only differ by a polynomial, and then saying that we don't care about polynomial differences.

81 **Proposition**

Every multitape TM that runs in $O(f(n))$ time has an equivalent 1-tape TM that runs in $O(f(n)^2)$.

Proof. Remember that if we have a multitape TM M, we constructed a 1-tape TM S by storing the multitape contents on a single tape one after another.

When M makes one step, S makes a complete pass through its whole tape. The length of the tape is $O(f(n))$, because at each step, M can only write one symbol to each tape. Therefore, on each step of M, S takes $O(f(n))$ steps. Since M takes $f(n)$ steps, S will run in $O(f(n)^2)$. □

Note that this only squared the time. It turns out that if we decide to ignore polynomial factors, i.e. two running times f and g are equivalent if $f = O(P(g))$ for some polynomial P, then the models of computation we care about are equivalent.

82 **Definition**

We say that a TM **runs in time $t(n)$** for a function $t : \mathbb{N} \to \mathbb{N}$ if the TM halts on all inputs of length n in $t(n)$ steps. Then, let

$$\mathrm{TIME}(t(n)) = \{B \mid \text{some TM decides } B \text{ in } O(t(n)) \text{ time}\}.$$

There is the idea that we have nested sets of languages based on the time in which they can be decided:

$$\mathrm{TIME}(n) \subseteq \mathrm{TIME}(n \log \log n) \subseteq \mathrm{TIME}(n \log n) \subseteq \mathrm{TIME}(n^2) \subseteq \cdots.$$

83 **Definition**

Let the **polynomial time languages** be $\mathrm{P} = \bigcup_k \mathrm{TIME}(n^k)$. These are the languages which have machines that decide them in polynomial time of the input size.

Note that this has nice properties we care about:

- model independent (for reasonable deterministic models[1])
- roughly corresponds to practical solvability.

[1] i.e. can't do an exponential (non-polynomial) amount of work on each step

84 **Example**

$$PATH = \{\langle G, s, t\rangle \mid G \text{ is a directed graph with a path from } s \text{ to } t\} \in \mathsf{P}.$$

Run depth first search. This takes $O(|V| + |E|)$ time, which is a polynomial in the input size.

3.2 Nondeterministic Time Complexity

Recall that from last time that $PATH \in \mathsf{P}$. Consider the similar problem

$$HAMPATH = \{\langle G, s, t\rangle \mid \text{directed graph } G \text{ has a path from } s \text{ to } t \text{ visiting each node exactly once}\}.$$

It turns out that whether this problem is in P or not is open. However, it seems that it if we are given a solution to this problem, we can check this solution in polynomial time. Let's rigorize this idea.

Recall that a nondeterministic Turing machine is defined just as we expect. Everything is the same except that the transition function is

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

85 **Definition**

Given an NTM M, we say that M **runs in time $t(n)$** for a function $t \colon \mathbb{N} \to \mathbb{N}$ if for all strings w of length n, all branches of M on w must halt within $t(n)$ steps. Then let

$$\mathrm{NTIME}(t(n)) = \{B \mid \text{some 1-tape NTM decides } B \text{ in } O(t(n)) \text{ time}\}.$$

86 **Definition**

Let the **nondeterministic polynomial time languages** be $\mathsf{NP} = \bigcup_k \mathrm{NTIME}(n^k)$. In particular, this is the set of languages for which there exists a NTM that decide them in polynomial time of the input size.

Note that we have $\mathsf{P} \subseteq \mathsf{NP}$, because every TM is a NTM. However, the problem of $\mathsf{NP} \subseteq \mathsf{P}$ is open.

87 **Example**

$HAMPATH \in \mathsf{NP}$.

Consider the following NTM:

0. On input $\langle G, s, t\rangle$:
1. Let V be the nodes and E be the edges of G. Also let $m \leftarrow |V|$.
2. Nondeterministically select a list of nodes $v_1, \ldots, v_m$ from V.
3. Check $v_1 = s$, $v_m = t$, $(v_i, v_{i+1}) \in E$ for each i, and there are no repetitions in the list of v_i.
4. If all checks passed then ACCEPT, otherwise REJECT.

Selecting a list of node $v_1, \ldots, v_m$ takes m^2 time, and the checking step takes $O(m)$ time. Therefore, this runs in polynomial time.

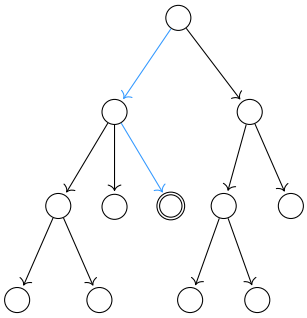
88 **Example**

COMPOSITES = $\{x \mid x$ is a binary string representing a composite number$\} \in$ NP. Here, we are taking n to be the length of x. In fact, *COMPOSITES* $\in$ P, but this is difficult to prove[2]. Note that this gives the surprising fact that *PRIMES* is also in P!

Consider the following NTM:

0. On input x:
1. Select some $0 < q < x$ nondeterministically.
2. If $q \mid x$ then ACCEPT, otherwise REJECT.

We also would like to rigorize what we mean by "checking" whether a solution works or not. We also have to be careful about how we measure an NTM's runtime. We will consider the maximum time it takes to decide if a word is in the language. Fortunately, we can fix both of these problems at the same time. In particular, we can consider the computation tree that the NTM can take:



If the NTM accepts, then we will say that the runtime is the depth of the node. Otherwise, the NTM rejects, and we will say that the runtime is the full height of the tree. Moreover, if the NTM accepts, there exists some path from the starting node to an accepting state. Note that a deterministic TM can simulate this NTM if it were told this path, i.e. which decisions to make whenever the tree branches. This motivates the following definition:

89 **Definition**

A **verifier** for a language A is a Turing machine M such that

$$A = \{w \mid \text{there exists some string } c \text{ such that } M \text{ accepts } \langle w, c\rangle\}.$$

The string c is called the **certificate**. Then, we say that A is **polynomially verifiable** if there exists some verifier M that runs in polynomial time with respect to the length of w.

The string c can be thought of extra information that helps M compute A. It can even be the answer to the problem! For example, a verifier for *HAMPATH* can take the input $\langle G, s, t, (v_1, \ldots, v_m)\rangle$, and just check whether $(v_1, \ldots, v_m)$ is a valid path connecting s to t.

In general, $A \in$ P if and only if testing whether $x \in A$ is decidable in polynomial time. Similarly, $A \in$ NP if and only if A is polynomially verifiable.

90 **Problem** (P vs. NP)

Does there exist a problem $p \in$ NP but $p \notin$ P? In other words, does P = NP? This problem is open.

[2] see the AKS primality test

3.2.1 CFLs

Recall that all CFLs are decidable, because we can find a context-free grammar in Chomsky normal form and test all of the possible grammars. We can show that CFLs are in NP by nondeterministically selecting a derivation of length $2n - 1$. However, we can say something stronger.

91 **Proposition**

If A is a CFL, then $A \in \mathsf{P}$.

Proof. We use the idea of dynamic programming. Since A is context-free, there exists some context-free grammar G in Chomsky normal form that generates it. Consider the following TM:

0. On input w:
1. Let $n \leftarrow |w|$.
2. For all lengths $k = 1, 2, \ldots, n$:
    a) Check whether each substring of length k can be each variable of the CFG by using the rules in the CFG and the subproblems.
    b) Remember the results and use old results if possible.

This is the bottom-up approach corresponding to selecting a point to split w and then determining whether each part is in A with dynamic programming. □

The key idea of dynamic programming is that we save answers to smaller subproblems to reuse later.

92 **Example**

We can show that $L = (\mathtt{aa} \cup \mathtt{aba} \cup \mathtt{aaab})^* \in \mathsf{P}$.

Suppose we wish to check if $w = \mathtt{aaaba} \in L$. We can use a bottom-up approach. We consider all possible prefixes of w, and determine what it produces by adding one instance of $(\mathtt{aa} \cup \mathtt{aba} \cup \mathtt{aaab})$.

	∘aa	∘aba	∘aaab
ε	aa	aba	aaab
a			
aa	aaaa	aaaba	aaaaab
aaa			
aaab	aaabaa	aaababa	aaabaaab
aaaba			

For the rows starting with a and aaa, we cannot find it in the table, so we skip the row. For aaaba, we find it in the table, so we ACCEPT.

3.3 Polynomial Time Reducibility

3.3.1 Satisfiability Problem

93 **Problem** (SAT)

Suppose we have a formula made from variables, $\wedge$s, $\vee$s, and $\neg$s. We want to determine whether there exists some assignment of variables over $\{\text{TRUE}, \text{FALSE}\}$ or $\{1, 0\}$ that makes the formula true. We call this problem *SAT*. More formally,

$$SAT = \{\langle \phi \rangle \mid \text{some assignment satisfies the formula } \phi\}.$$

94 **Example**

$(a \vee b) \wedge (a \vee \neg b)$ is satisfiable with $(a, b) = (1, 0)$ or $(a, b) = (1, 1)$.

$a \wedge \neg a$ is never satisfiable.

It is clear that $SAT \in$ NP because we can create an NTM that guesses all possible combinations.

Cool Special Cases

We call *k-SAT* the problem of SAT on a formula in **conjunctive normal form**, i.e. where each clause is the OR of k literals, and the clauses are ANDed together. For example,

$$(a \vee b \vee c) \wedge (\neg a \vee b \vee \neg b) \wedge (d \vee b \vee c) \wedge \cdots \wedge (\neg b \vee \neg c \vee \neg d)$$

is an instance of *3-SAT*.

It turns out that every instance of *SAT* can be reduced to *3-SAT*. Moreover, it turns out that *2-SAT* $\in$ P because we can construct a directed graph with nodes

$$V = \bigcup_{\text{variables } v} \{v, \neg v\},$$

and edges corresponding to the implications, and then run a connected components algorithm on the graph. In contrast, *3-SAT* is probably not in P.

3.3.2 Clique Problem

Suppose we have an undirected graph G. Then a ***k*-clique** is a set of k nodes where there exists an edge between each pair of nodes. Then, we have the natural problem

$$CLIQUE = \{\langle G, k \rangle \mid \text{undirected graph } G \text{ contains a } k\text{-clique}\}.$$

3.3.3 Reducing

95 **Definition**

We say A is **polynomial-time reducible** to B, denoted $A \leq_p B$ if and only if $A \leq_m B$ and the reduction is computable in polynomial time.

96 **Proposition**

3-SAT $\leq_p$ *CLIQUE*.

Proof. Suppose we have some instance of *3-SAT* ϕ with k clauses. Then we will construct the graph G as follows. There will be $3k$ nodes of G, one corresponding to each literal in each clause. The edges will be everything except

- edges between nodes in the same clause, and
- edges between nodes that are opposite literals (e.g. x and $\neg x$).

We will show that $\langle \phi \rangle \longrightarrow \langle G, k \rangle$ is the reduction that shows *3-SAT* $\leq_p$ *CLIQUE*.

If $\langle \phi \rangle \in$ *3-SAT*, then ϕ has a satisfying assignment. Then, at least one literal in each clause is true, which gives us the k nodes in the clique. Therefore, $\langle G, k \rangle \in$ *CLIQUE*.

If $\langle G, k \rangle \in$ *CLIQUE*, then there exists some k-clique in G. Since there are no edges from a clause to itself, the clique has one node per clause. Moreover, since there are no edges connecting opposite literals in G, the clique corresponds to an assignment. Therefore, if we assign all of the nodes in our clique to be true, we get an assignment satisfying ϕ. □

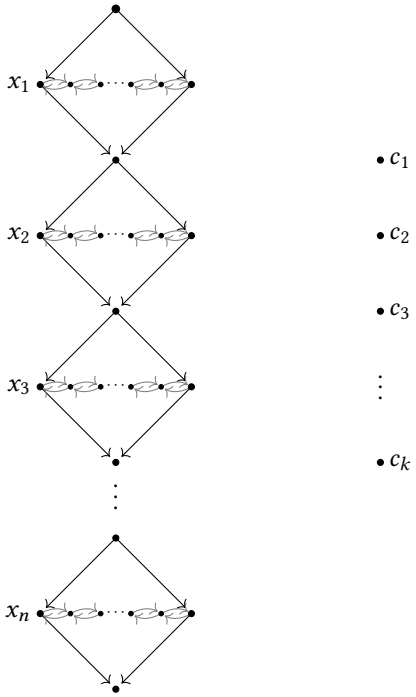
97 **Definition**

A problem B is **NP-hard** if for every $A \in \mathsf{NP}$, we have $A \leq_p B$. Then, B is **NP-complete** if we also have $B \in \mathsf{NP}$.

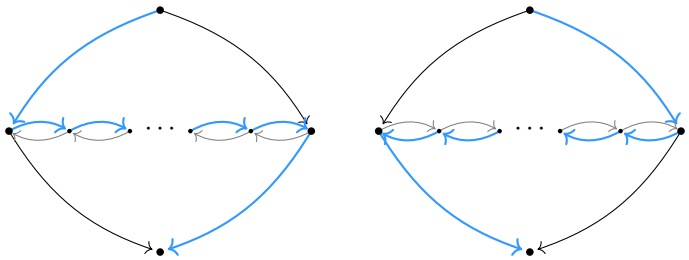
98 **Proposition**

3-SAT $\leq_p$ *HAMPATH*.

Proof. Suppose we have some formula ϕ with k clauses in n variables $x_1, \ldots, x_n$ in conjunctive normal form. Consider the graph with the following base structure:



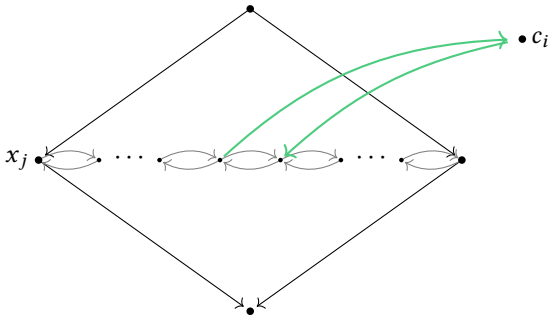
We have diamond structures corresponding to each variable, and a node corresponding to each clause. We want to encode the fact that ϕ is either satisfiable or not satisfiable with a Hamiltonian path. Overall, we will start at x_1, and work our way down to x_n. Let's look at one of the diamond structures more closely.



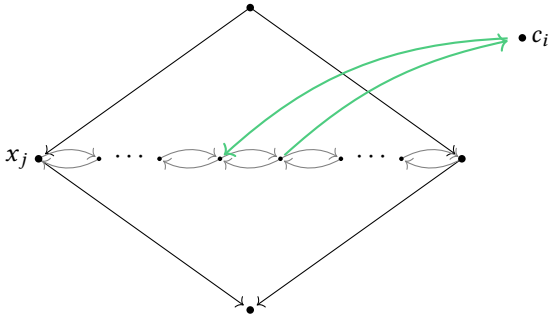
There are two ways in which we can go through a diamond structure. In particular, we can go left initially, go right through the internal nodes, and then left to get to the bottom node. Alternatively, we can go right initially and then everything gets flipped. Intuitively, this can correspond to whether the variable is set to true or false.

Now we want to actually encode the information of the formula and clauses being satisfied. In particular, we will visit the node corresponding to the clause if it is satisfied. Then it makes sense that if all the nodes are visited, all of the clauses are satisfied, and furthermore the formula is satisfied.

In particular, if the clause c_i contains the literal x_j, then we will connect the structure of c_i to x_j as follows:



Similarly, if the clause c_i contains the literal $\neg x_j$, then we connect the structures like so:



We have to be a bit careful about which internal nodes we connect the c_i node to. In particular, if we don't want them to interfere, we can have $3k + 3$ nodes, where the ith clause connects to the $3i$th node and $(3i + 1)$th node. This way, there is a buffer node between everything.

It is clear that if there is a satisfying assignment, then there will exist a Hamiltonian path through this graph, because we will choose which way we go through each diamond structure, and since each clause is satisfied, we we can visit each clause at some point while zig-zagging through the diamond structures.

Similarly, if there exists a Hamiltonian path, then we can find the assignment of variables by looking at which direction the path takes through the diamond structures. □

3.3.4 SAT is NP-Complete

99 **Theorem** (Cook-Levin)
SAT is NP-complete.

Proof. We already know that $SAT \in \mathsf{NP}$.

Let $A \in \mathsf{NP}$. We want to show that $A \leq_p SAT$. In particular, we want to find a reduction $f : w \mapsto \phi_w$ such that $w \in A \iff \phi_w \in SAT$.

Let A be decided by an NTM M in $O(n^k)$ time. The key idea is that we will try to make ϕ_w be the statement "M accepts w".

Definition
A **tableau** for M on w is an $N \times N$ table where row i is the configuration on the ith step of an accepting branch of M on w, where the whole branch is N steps.

q_0	w_1	w_2	$\dots$	w_n	$\sqcup$	$\sqcup$	$\sqcup$

We can then think of the statement "M accepts w" as the statement "a tableau for M on w exists".

In particular, we can build ϕ_w with the indicator variables

$$\left\{ x_{ij\sigma} = \begin{cases} 1 & \text{cell } (i,j) \text{ contains } \sigma \\ 0 & \text{else,} \end{cases} \middle| \; i, j \in \{1, \dots, N\}^2, \sigma \in \tilde{\Gamma} = \Gamma \cup Q \right\}.$$

Our formula will be of the form

$$\phi_w = \phi_{\text{cell}} \wedge \phi_{\text{start}} \wedge \phi_{\text{move}} \wedge \phi_{\text{accept}},$$

where

- ϕ_{cell} checks that each cell has exactly one symbol,
- ϕ_{start} checks that we have the right starting configuration,
- ϕ_{move} checks that each move is valid, and
- ϕ_{accept} checks that the final state is accepting.

We can write

$$\phi_{\text{cell}} = \bigwedge_{1 \leq i,j \leq N} \left[\bigvee_{\sigma \in \tilde{\Gamma}} x_{ij\sigma} \wedge \bigwedge_{\substack{\sigma,\tau \in \tilde{\Gamma} \\ \sigma \neq \tau}} (\overline{x_{ij\sigma}} \vee \overline{x_{ij\tau}}) \right].$$

The outer $\bigwedge$ goes through each cell. In particular, the first part inside the bracket says that there must be at least one symbol in each cell, and the second part says that for every pair of symbols (σ, τ), they cannot both be in the cell.

We have

$$\phi_{\text{start}} = x_{1,k,w} \wedge \bigwedge_{k=1}^{n} x_{1,k+1,w_k} \wedge \bigwedge_{k=n+1}^{N-1} x_{1,k,\sqcup}.$$

This just requires the first row to be $q_0 w_1 w_2 \dots w_n \sqcup \dots$. Similarly,

$$\phi_{\text{accept}} = \bigvee_{1 \le i,j \le N} x_{i,j,q_{\text{ACCEPT}}}.$$

The hardest subformula to come up with is ϕ_{move}. The idea is that we check that every 2×3 subgrid and make sure that the subgrid is valid. In particular, we have

$$\begin{aligned}\phi_{\text{move}} &= \bigwedge_{1 \le i,j \le N} [\text{subgrid at } (i,j) \text{ is valid}] \\ &= \bigwedge_{1 \le i,j \le N} \left[\bigvee_{\text{valid } \left[\begin{smallmatrix} abc \\ def \end{smallmatrix}\right]} x_{i,j,a} \wedge x_{i+1,j,b} \wedge x_{i,j+1,c} \wedge x_{i+1,j+1,d} \wedge x_{i,j+2,e} \wedge x_{i+1,j+2,f} \right].\end{aligned}$$

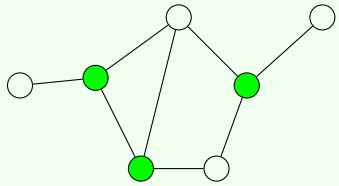
If ϕ_{move} is true, Therefore, ϕ_w is an instance of *SAT*, and a satisfying assignment exists □

3.3.5 Vertex Cover

Suppose we have an undirected graph $G = (V, E)$. A **vertex cover** of G is a set $X \subseteq V$ such that every edge $e \in E$ has at least one end point in X. A ***k*-vertex cover** is a vertex cover of size k.

100 **Example** (Vertex cover)

The green nodes of the following graph form a 3-vertex cover:



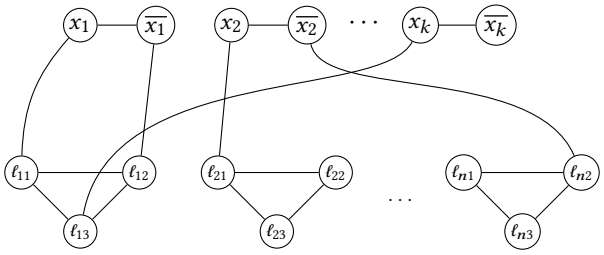
It is natural to then ask the question: given a graph G, does it have a k-vertex cover? In fact, we define

$$\textit{VERTEX-COVER} = \{\langle G, k\rangle \mid \text{graph } G \text{ has a } k\text{-vertex cover}\}.$$

101 **Proposition**

VERTEX-COVER is NP-complete.

Proof. We will show that *3-SAT* $\le_p$ *VERTEX-COVER*. Suppose we have an instance ϕ of *3-SAT* with variables $x_1, \dots, x_k$, and clauses $c_1, \dots, c_n$, where the clause c_i contains the literals $c_i = \ell_{i1} \vee \ell_{i2} \vee \ell_{i3}$. Then consider the graph G with the following nodes:



We have two connected nodes for each variable x_i. One for x_i and one for $\overline{x_i}$. We also have three connected nodes for each clause, one for each literal. Then, we also connect the variable nodes with the corresponding literals if they are the same. We then consider the reduction

$$f\colon \langle \phi \rangle \mapsto \langle G, k + 2n \rangle.$$

Note that to cover the edges between the variables x_i and $\overline{x_i}$, we need to use at least k vertices on the top. Similarly, to get all of the edges for the clauses, we need to use at least $2n$ vertices on the bottom. Therefore, it must be this distribution.

If ϕ has a satisfying assignment, then there exists some assignment for each x_i. This corresponds to the k vertices we select on top. Then, since all of the clauses are satisfied, for each clause c_i there exist some literal ℓ_{ij} in the clause that is satisfied. In particular, the edge from the variable to the literal is covered by the variable. Therefore, the 2 nodes we select for the clause can be the other $\ell_{ij'}$, and all edges relevant to that clause will be covered.

Similarly, if G has a $(k + 2n)$-vertex cover, then there is a set with k vertices on the top and $2n$ on the bottom. The k vertices correspond to the assignment of variables. Then for each clause c_i, we can only select 2 of the 3 vertices $\{\ell_{i1}, \ell_{i2}, \ell_{i3}\}$. This means that the edge connecting the last literal edge must be covered with the variable vertex, implying that the clause is satisfied. □

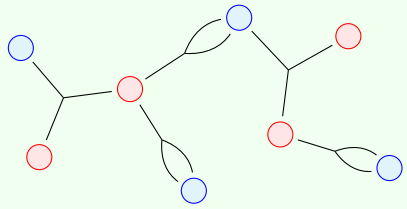
3.3.6 ALLCUT Problem

Consider a set of vertices V. Let T be a set of subsets of V with size 3. We can think of these as like 'edges', but they have three nodes. Suppose we color each node either blue or red. Call an triple $t \in T$ **happy** if it has at least one blue node and one red node. Then we can define the problem

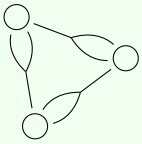
$$ALLCUT = \{\langle V, T\rangle \mid V \text{ is a set of nodes with triples } T \text{ that can be colored such that all triples are happy}\}.$$

102 **Example**

All of the triples of the following graph with its coloring are happy:



In contrast, the following vertices with triples shown do not have a coloring that makes all triples happy:

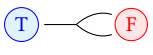


103 **Claim**

ALLCUT is NP-complete.

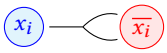
Proof. First, we know that $ALLCUT \in$ NP because we can nondeterministically guess the color for each vertex. We will show that *ALLCUT* is NP-hard by proving $3\text{-}SAT \leq_p ALLCUT$.

The key idea is that since we want to embed an asymmetrical problem in a symmetrical setting, we introduce a way to get rid of the redundancy. In this problem, we're trying to map true formulas to a colored graph, we will have two connected template nodes for the values of TRUE and FALSE. We can force them to have opposite colors by connecting them both with an edge, where one of them is connected twice.

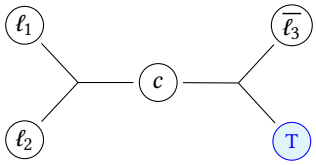


Then, we can connect other things to these nodes in order to enforce which color means true and which means false.

In our graph, we will also have a node for each possible literal, i.e. two nodes for each variable. We can force all of these to have opposite colorings using the same method we did for TRUE and FALSE.



Consider some clause $c = \ell_1 \vee \ell_2 \vee \ell_3$. We want to encode the fact that at least one of the literals is TRUE. To do this, we can form a new node for the clause, and then connecting the following nodes is sufficient:



To see why this works, note that if both ℓ_1 and ℓ_2 are colored with the same color as FALSE, then the node labeled c must be colored TRUE. Then, this forces $\overline{\ell_3}$ to be colored FALSE, which means ℓ_3 is true.

Now, if ϕ is an instance of *3-SAT* that is satisfiable, then we color each node corresponding to each literal with its truth value. Then, it is easy to check that the c node corresponding to a clause can be colored.

Conversely, if the graph is colorable, then we can take the colors that we have assigned to the literal clauses and get the assignment of values. □

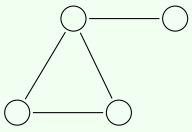
3.3.7 MAXCUT is NP-Complete

Consider an undirected graph $G = (V, E)$ and some $k \in \mathbb{N}$. Call an edge **happy** if we can color each node such that each edge has both colors. Let

$$MAXCUT = \{\langle G = (V, E), k\rangle \mid G \text{ is a graph such that at least } k \text{ edges are happy}\}.$$

104 **Example**

For the following graph G, we have $\langle G, 3\rangle \in MAXCUT$ but $\langle G, 4\rangle \notin MAXCUT$.



105 **Claim**

MAXCUT is NP-complete.

Proof. It is clear that $MAXCUT \in$ NP because we can just nondeterministically guess the coloring.

To prove that *MAXCUT* is NP-hard, we will show that $ALLCUT \leq_p MAXCUT$. Suppose we have an instance $\langle V, T\rangle$ of *ALLCUT*. Then, for each triple $\{u, v, w\} \in T$, we make three edges $\{u, v\}$, $\{v, w\}$, and $\{w, u\}$. If the original triple was happy, then exactly two of the new edges are happy. Therefore, we replace all triples in the original graph with these triangles, and require that $K = 2|T|$. In particular, $f(V, T) = \langle V, 2|T|\rangle$. □

3.4 Space Complexity

Similar to time complexity, we can define space complexity.

106 **Definition**

Given a function $f : \mathbb{N} \to \mathbb{N}$ (assuming $f(n) \geq n$ for convenience), we say that a TM M **runs in space $f(n)$** if M is a decider and uses $\leq f(n)$ tape cells on all inputs of length n.

Similarly, a NTM M **runs in space $f(n)$** if M is a decider and uses $\leq f(n)$ tapes cells on each branch for all inputs of length n.

Then, we can define

$$\text{SPACE}(f(n)) = \{A \mid \text{some TM decides } A \text{ in } O(f(n)) \text{ time}\}$$
$$\text{NSPACE}(f(n)) = \{A \mid \text{some NTM decides } A \text{ in } O(f(n)) \text{ time}\}.$$

Then, similar to time complexity we can define

$$\text{PSPACE} = \bigcup_k \text{SPACE}(n^k)$$
$$\text{NPSPACE} = \bigcup_k \text{NSPACE}(n^k).$$

For $f(n) \geq n$, it is immediate that $\text{TIME}(f(n)) \subseteq \text{SPACE}(f(n))$. This is because if a machine runs in time $O(f(n))$, then it uses at most $O(f(n))$ space as well, because the machine can only use one tape cell per time step. This implies that $\text{P} \subseteq \text{PSPACE}$, and for a similar reason, we have $\text{NP} \subseteq \text{NPSPACE}$.

Conversely, we have $\text{SPACE}(f(n)) \subseteq \text{TIME}(2^{O(f(n))})$, because if a machine uses $O(f(n))$ space and must halt, then it runs in time $c \cdot |\Gamma|^{O(f(n))}$ for some constant c.

107 **Theorem**

$\text{NP} \subseteq \text{PSPACE}$.

Proof. Note that $SAT \in \text{PSPACE}$ because we can keep track of the formula and an assignment in polynomial space, and test each possible assignment. Then, for $A \in \text{NP}$, we know $A \leq_p SAT$, so $A \in \text{PSPACE}$. □

108 **Example**

We have $\text{coNP} = \{\overline{A} \mid A \in \text{NP}\} \subseteq \text{PSPACE}$. This is because

Some examples of problems in coNP are the unsatisfiability problem, or tautology problem

$$UNSAT = \{\langle\phi\rangle \mid \text{formula } \phi \text{ is not satisfied for some assignment}\}$$
$$TAUT = \{\langle\phi\rangle \mid \text{formula } \phi \text{ is satisfied for all assignment}\}.$$

3.4.1 Quantified Boolean Formulas

Let a **quantified boolean formula** be a boolean formula with quantifiers $\forall$ and $\exists$.

109 **Example**

The two formulas

$$\forall x \exists y [(x \vee y) \wedge (\overline{x} \vee \overline{y})]$$
$$\forall y \exists x [(x \vee y) \wedge (\overline{x} \vee \overline{y})]$$

are quantified boolean formulas that are TRUE and FALSE respectively.

Consider the problem of deciding whether a quantified boolean formula is true or not, i.e.

$$TQBF = \{\langle \phi \rangle \mid \phi \text{ is a TRUE quantified boolean formula}\}.$$

The T in $TQBF$ stands for true.

110 **Proposition**
$TQBF \in$ PSPACE.

Proof. Consider the following algorithm:

0. On input $\langle \phi \rangle$:
1. If there are no quantifiers, then there are only constants, so we can evaluate ϕ directly and ACCEPT if ϕ is true.
2. If $\phi = \exists x \psi$ for some quantified boolean formula ψ, then recursively evaluate ψ with $x \in \{\text{TRUE}, \text{FALSE}\}$. If either case accepts, then ACCEPT.
3. If $\phi = \forall x \psi$ for some quantified boolean formula ψ, then recursively evaluate ψ with $x \in \{\text{TRUE}, \text{FALSE}\}$. If both cases accept, then ACCEPT.

Note that this algorithm can be run in polynomial space because there are at most n recursive calls, and each recursive call uses $O(1)$ space. □

3.4.2 Word Ladder

There are some games where you have two words and try to get from one to the other by changing one letter at a time, while remaining on valid words. E.g.

$$\text{WORK} \to \text{PORK} \to \text{PORT} \to \text{SORT} \to \text{SOOT} \to \text{SLOT} \to \text{PLOT} \to \text{PLOY} \to \text{PLAY}.$$

Let's define the problem

$$LADDER_{\text{DFA}} = \{\langle B, u, v \rangle \mid B \text{ is a DFA such that there exists a sequence } y_0, \dots, y_k \text{ in } L(B) \text{ such that } y_0 = u,\ y_1 = v, \text{ and } y_i \text{ differs from } y_{i+1} \text{ in one symbol}\}.$$

111 **Proposition**
$LADDER_{\text{DFA}} \in$ NPSPACE.

Proof. Consider the following nondeterministic algorithm:

0. On input $\langle B, u, v \rangle$:
1. Make a copy of u, called y.
2. Nondeterministically change one letter.
3. Repeat either until $y = v$ (in which case ACCEPT), or $|u|^{|\Sigma|}$ letters have been changed (in which case REJECT), where Σ is the alphabet of B.

Note that this algorithm runs in polynomial space for all branches, so $LADDER_{\text{DFA}} \in$ NPSPACE. □

112 **Proposition**
$LADDER_{\text{DFA}} \in$ PSPACE.

Proof. Let's write $u \to v$ if $\langle B, u, v \rangle \in LADDER_{\text{DFA}}$, and $u \xrightarrow{k} v$ if the ladder has $\le k$ steps. Consider the following algorithm that tests if $u \xrightarrow{k} v$:

0. On input $\langle B, u, v\rangle$:
1. If $k = 1$, then check directly.
2. For each w of length m:
   a) Test $u \xrightarrow{k/2} w$ and $w \xrightarrow{k/2}$ recursively.
   b) If both succeed then ACCEPT.
3. If no such w work, then REJECT.

The number of levels of recursion is $\log k$, and the space used per level is m, namely in storing w. Therefore, this algorithm uses $m \log k$ space.

To test if $u \to v$ in general, then it suffices to test $u \xrightarrow{|\Sigma|^m} v$. This takes space

$$m \log d^m = m^2 \log d = O(m^2) \le O(n^3).$$

□

113 **Theorem** (Savitch)

For any function $f : \mathbb{N} \to \mathbb{N}$ where $f(n) \ge n$,

$$\mathsf{NSPACE}(f(n)) \subseteq \mathsf{SPACE}(f(n)^2).$$

Proof. Let $A \in \mathsf{NSPACE}(f(n))$ decided by an NTM M that runs in space $O(f(n))$. We want to show that there is a deterministic algorithm for A that runs in $O(f(n)^2)$ space. Note that since M halts, the maximum number of steps it can take is $d^{f(n)}$, where d is some constant depending on M.

Let's (suggestively) write $C_i \xrightarrow{k} C_j$ where C_i and C_j are configurations of M, if C_i can yield C_j in k steps. Then M accepts w if $C_{\text{START}} \xrightarrow{d^{f(n)}} C_{\text{ACCEPT}}$.

Consider the following algorithm that tests if $C_i \xrightarrow{k} C_j$:

0. On input $\langle C_i, C_j, k\rangle$:
1. If $k = 1$, then check directly according to the rules of M.
2. For each possible configuration C_{MID}:
   a) Test $C_i \xrightarrow{k/2} C_{\text{MID}}$ and $C_{\text{MID}} \xrightarrow{k/2} C_j$.
   b) If both succeed, then ACCEPT.
3. If no such w work, then REJECT.

There are $\log k$ levels of recursion, and the space used at each level is the amount of space we use to write C_{MID}, which is $O(f(n))$. Therefore, the total space we use is $O(f(n) \log k)$.

The problem of interest is for $k = d^{f(n)}$, and the total runtime is $O(f(n) \cdot \log d^{f(n)}) = O(f(n)^2)$. □

Therefore, we have the chain

$$\mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PSPACE} = \mathsf{NPSPACE},$$

where the problems of whether these subsets are strict are open.

3.5 PSPACE-Completeness

114 **Definition**

A problem B is **PSPACE-hard** if for every $A \in \mathsf{PSPACE}$, we have $A \le_p B$. Then, B is **PSPACE-complete** if we also have $B \in \mathsf{PSPACE}$.

Note that here we are using a polynomial time reduction because if we used a PSPACE reduction, everything would be PSPACE-complete. In general, we usually want to use a reduction that is weaker than the space.

115 **Proposition**
TQBF is PSPACE-complete.

Proof. First, we know that $TQBF \in$ PSPACE from proposition 110.

Now suppose we have some $A \in$ PSPACE using a TM M running in $O(n^k)$ space. We want to show that $A \leq_p TQBF$. If $w \in A$, consider the tableau for M on w. Then, we want to determine whether there exists a sequence of configurations $C_{\text{START}}, \dots, C_{\text{ACCEPT}}$. More specifically, we want to write an instance of *TQBF* that says "M accepts w".

Let's try to solve a more general problem. Given configurations C_i and C_j and some $k \in \mathbb{N}$, we want to write a quantified binary formula $\phi_{C_i,C_j,k}$ that says $C_i \xrightarrow{k} C_j$ i.e. that there exists a computation history tableau that goes from C_i to C_j in k steps. Like in Cook-Levin We can use a similar strategy that we used to prove Savitch's theorem, i.e. recursion.

The quantified formula $\phi_{C_i,C_j,k}$ is equivalent to there existing some intermediate configuration C_m such that $C_i \xrightarrow{k/2} C_m$ and $C_m \xrightarrow{k/2} C_j$. In particular, we can write

$$\phi_{C_i,C_j,k} = \exists C_m \, (\phi_{C_i,C_m,k/2} \wedge \phi_{C_m,C_j,k/2}).$$

However if we recuse with this formula, this gives us a formula of size $O(k)$. This is bad because our initial k will be exponential in the input size. We can fix this by using the $\forall$ feature of quantified formulas. In particular,

$$\phi_{C_i,C_j,k} = \exists C_m \, \forall (C_i', C_j') \in \{(C_i, C_m), (C_m, C_j)\} \, (\phi_{C_i,C_m,k/2}).$$

This gives us a formula of length $O(\log k)$, which is good.

One thing to be careful about is that we said $\exists C_m$ and $\forall (C_i', C_j')$, when *TQBF* was defined for boolean variables. The way to fix this is to use the indicator variables, like we did in the proof of Cook-Levin. More specifically, we can write $\forall x \in \{s_1, s_2\} \, \varphi$ as $\forall x \, [(x = s_1 \vee x = s_2) \rightarrow \varphi]$.

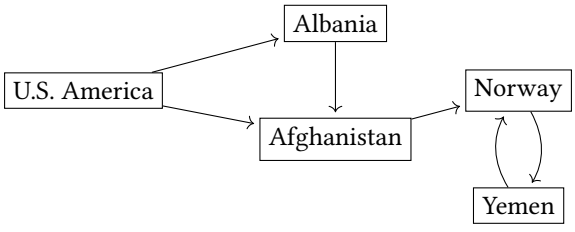
Therefore, we can write $\phi_{C_{\text{START}},C_{\text{ACCEPT}},2^{dn^k}}$ as a formula in $O(\log 2^{dn^k}) = O(dn^k)$ space, and therefore $w \mapsto \phi_{C_{\text{START}},C_{\text{ACCEPT}},2^{dn^k}}$ is the reduction that we are looking for. □

3.6 Games

A **game** is loosely defined to be a competition between two players that are trying to achieve opposing goals.

3.6.1 Geography Game

Consider the following game between two people: We start with a country. The next player has to select a country that starts with the last letter of the previous. The last player that can name a country that has not been previously said wins. We can draw a graph that that looks like the following representing the game.



It is natural to ask who wins this game. In this case there are a finite number of nodes, so it is easy to determine the winner. Let's generalize this game. Suppose we have a directed graph $G = (V, E)$ with a designated starting vertex $a \in V$, and the players select

nodes that have not been previously used along a path until they are not able to. The last person who successfully selects a node wins. Then we can define the problem

$$GG = \{\langle G, a\rangle \mid \text{Player 1 has a winning strategy in the generalized geography game on graph } G \text{ starting with } a\}.$$

116 **Proposition**

GG is PSPACE-complete.

Proof. It is clear that $GG \in$ PSPACE.

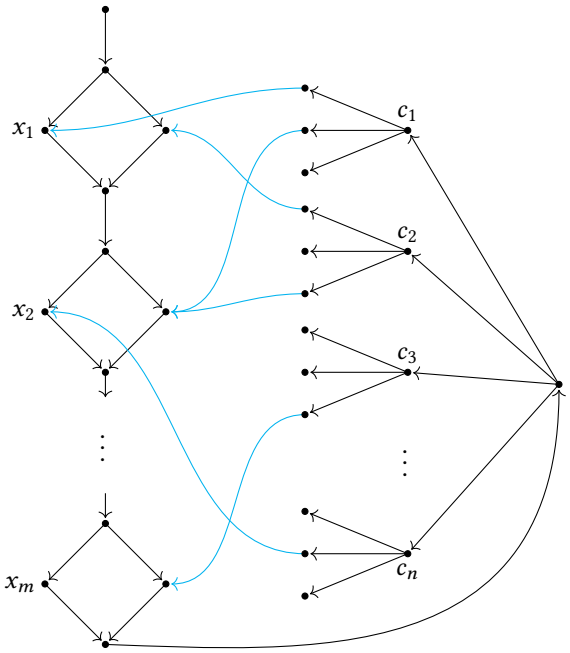
To prove that *GG* is PSPACE-hard, we will prove $TQBF \leq_p GG$. Suppose we have some instance $\langle\varphi\rangle$ of *TQBF*

$$\varphi = \exists x_1 \forall x_2 \exists x_3 \forall x_4 \ldots \exists x_m \psi,$$

where ψ is in conjunctive normal form. Let's create a game where we suggestively name our players E and A. The idea is that E will select the values of the variables $x_1, x_3, \ldots, x_m$, while A will select the values of $x_2, x_4, \ldots, x_{m-1}$. Note that we can always write φ in this form by converting to prenex normal form, and then adding dummy variables so that the $\exists$ and $\forall$ alternate.

Let's say that E wins if the selection of values causes ψ to be TRUE, and A wins if the selection of values causes ψ to be FALSE. Then, it turns out that E wins if and only if $\varphi \in TQBF$.

Consider the following graph G:



There is a diamond structure for each variable x_i. The players alternate picking the left or right path, and at the end we have another structure to check whether the formula is true or not. We want E to win if the resulting formula with the chosen variables is true. To check this in a game setting, the idea is to let A claim that a certain clause is false, and then within the clause, E chooses the literal that they claim is true. Then, we connect the literal to the corresponding node in the diamond structure. If the literal is actually true, then A has no move, so E wins, as desired. Conversely, if E is not able to select a true literal, then A will have another move, and then E loses, as desired. □

3.7 Logspace

We would like to consider problems that can be solved in less than linear space. However, all of our current Turing machines use at least $O(n)$ space because the tape must contain the input. To extend to problems that can be solved in a smaller space constraint, we will consider a 2-tape TM where the input is on a read-only tape, and the second tape is a standard read/write tape that starts empty. Then we only count the space that the read/write tape uses. Then, we say that a problem is **solvable in log space** if the machine that solves the problem only uses $O(\log n)$ space on its second tape.

Some motivation for this are CD-ROMs and the internet. The input (entire data on the internet) might be large, but we can download them to a local machine.

117 **Definition**

The class of languages decidable in log space on a deterministic Turing machine is

$$\mathsf{L} = \mathrm{SPACE}(\log n) = \{\mathscr{L} \mid \mathscr{L} \text{ is solvable in log space}\}.$$

Similarly, we define the class of languages decidable in logarithmic space on a nondeterministic Turing machine to be

$$\mathsf{NL} = \mathrm{NSPACE}(\log n) = \{\mathscr{L} \mid \mathscr{L} \text{ is solvable in log space nondeterministically}\}.$$

118 **Example**

$\{\mathsf{a}^k\mathsf{b}^k \mid k \in \mathbb{N}\} \in \mathsf{L}$.

We can keep a counter of the number of as on the second tape when we read them, and then decrement when we get to the bs.

Recall the problem

$$PATH = \{\langle G, s, t\rangle \mid G \text{ is a directed graph with a path from } s \text{ to } t\}.$$

We have that $PATH \in \mathsf{NL}$ because a Turing machine can keep track of just the current node and nondeterministically guess the next node while using a counter to make sure that we don't loop. However, it is not known whether $PATH \in \mathsf{L}$. In fact, the problem of whether $\mathsf{L} = \mathsf{NL}$ is open.

We can show that $\mathsf{NL} \subseteq \mathrm{SPACE}((\log n)^2)$ from Savitch's theorem. We can also show that $\mathsf{L} \subseteq \mathsf{P}$. In particular, we can consider the number of configurations that the machine that solves a problem in L, which is exponential in $O(\log n)$. Then, the maximum amount of time that the machine can run for is also exponential in $\log n$, which is bounded by a polynomial in n.

There is a more interesting fact though.

119 **Proposition**

$\mathsf{NL} \subseteq \mathsf{P}$.

Unfortunately if we try to use Savitch's theorem, we get $\mathsf{NL} \subseteq \mathrm{SPACE}((\log n)^2) \subseteq \mathrm{TIME}(n^{(\log n)^2}) \not\subseteq \mathsf{P}$,

Proof. Suppose we have some NTM N that runs in $O(\log n)$ space. We would like to find some TM M that runs in polynomial time.

Given some input w, consider the **computation graph** of N on w, where the nodes are configurations of N on w, and the edges connect $C_i \to C_j$ if C_i yields C_j according to the rules of N. The number of nodes is polynomial in n, so the graph is polynomial in size. Therefore, our machine M can create this graph.

Now, since $PATH \in \mathsf{P}$, we can test if there exists a path from C_{START} to C_{ACCEPT} in the graph. If yes then ACCEPT, else REJECT. □

This means that we have the chain of complexity classes

$$\mathsf{L} \subseteq \mathsf{NL} \subseteq \mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PSPACE} = \mathsf{NPSPACE}.$$

Similar to P and NP, the problem of whether $\mathsf{L} = \mathsf{NL}$ is also open.

3.7.1 NL-completeness

120 **Definition**

A **log-space transducer** is a TM that has a read-only input, a write-only output, and a work tape that is $O(\log n)$ space. We then say that A is **log-space reducible** to B, denoted $A \leq_L B$ if there is some function f computable by a log-space transducer such that $w \in A \iff f(w) \in B$.

121 **Definition**

A problem B is **NL-hard** if for all $A \in \mathsf{NL}$, we have $A \leq_L B$. Then, B is **NL-complete** if we also have $B \in \mathsf{NL}$.

Let's look at a proposition that we would expect to be true.

122 **Proposition**

If $A \leq_L B$ and $B \in \mathsf{L}$, then $A \in \mathsf{L}$.

To prove this, we might try to take a TM R that decides B in $O(\log n)$ space, and then construct the TM S that does the following:

0. On input w:
1. Run f on w to get $f(w)$.
2. Test using R if $f(w) \in B$. If yes then ACCEPT, else REJECT.

However, this does not work! R does not have enough memory to store the value of $f(w)$. Luckily, there is an easy fix to this. In particular, we just don't store the value of $f(w)$, and we compute $f(w)$ every time to get the part we need. In particular, let $f(w, i)$ be the ith bit of $f(w)$. Note that this is also computable in log space. We can just request $f(w, i)$ whenever we need the ith bit of $f(w)$.

Proof. Let $f(w, i)$ be the function computable in log space that gives the ith bit of $f(w)$, where f is the reduction for $A \leq_L B$. Then, consider the following TM that runs in log space:

0. On input w:
1. Simulate the decider for B on an abstract input x.
2. When the decider asks for the ith bit of x, give it $f(w, i)$.
3. Return whatever the decider returns. □

123 **Theorem**

PATH is NL-complete.

Proof. We showed earlier that $PATH \in \mathsf{NL}$. To show that *PATH* is NL-complete, suppose $A \in \mathsf{NL}$. Then, we want to find a log-space reduction $f : w \mapsto \langle G, s, t \rangle$ where $w \in A \iff \langle G, s, t \rangle \in PATH$. As in proposition 119, we can let G be the computation graph, $s = C_{\text{START}}$, and $t = C_{\text{ACCEPT}}$.

It suffices to show that we can compute f in log space. Each state in G can be represented in $O(\log n)$ space. For the edges, we must determine whether one state goes to another in log space. This works because we only have to analyze two states at a time, which can be stored in $O(\log n)$ space. □

We have another surprising result. In general, people believe that $\mathsf{P} \subsetneq \mathsf{NP} \cap \mathsf{coNP} \subsetneq \mathsf{NP}, \mathsf{coNP}$ and $\mathsf{NP} \neq \mathsf{coNP}$. However, the analogous statement for logarithmic space is not the same! In particular, we have the following theorem:

124 **Theorem**
NL = coNL.

Proof. We proceed by proving that $\overline{PATH} \in \mathsf{NL}$. Consider some instance $\langle G, s, t\rangle$ of $\overline{PATH}$, where $G = (V, E)$. Let

$$R = \{u \mid \text{vertex } u \text{ is reachable from } s\}$$

and $c = |R|$. The idea is that if we know c, then we can find c reachable nodes, and if t is not in the set, we accept. To do this, we can iterate through the nodes, and nondeterministically guess whether a node u is reachable or not. If we guess that it is reachable, then we will try to find a path using the same method we used to show that $PATH \in \mathsf{NL}$.

Now it suffices to find c. To do this, we will let R_i be the set of nodes reachable from u in i steps, and let $c_i = |R_i|$. If $|V| = m$, we have

$$R_1 \subseteq R_2 \subseteq R_3 \subseteq \cdots \subseteq R_m = R.$$

Consider the following nondeterministic inductive procedure to find $c_m = c$. We know that $c_0 = 1$. To find c_{i+1} from c_i, we will check both R_i and R_{i+1}. In particular, for each potential node $v \in V$ of R_{i+1}, we find all nodes $u \in R_i$ by following a path of length at most i, and keeping a counter to make sure we find all $u \in R_i$. If $u \to v$ is an edge for any u, then we know that $v \in R_{i+1}$, so increment c_{i+1}.

Note that this algorithm only needs to store m, c_i, c_{i+1}, u, v, and some bounded number of counters and pointers, so this runs in log space. □

Strongly Connected Problem

Suppose we have a graph G. Then we say that G is **strongly connected** if all points are reachable from every other point. This motivates the problem

$$SC = \{\text{directed graph } G = (V, E) \mid \text{all pairs } (s, t) \in V^2 \text{ has a path between them}\}.$$

We know that $SC \in \mathsf{NL}$ because we can loop through all pairs $(s, t) \in V^2$ and run *PATH*.

125 **Proposition**
SC is NL-complete.

Proof. We proceed by showing $PATH \leq_L SC$. Suppose we have an instance $\langle G, s, t\rangle$ of *PATH* where $G = (V, E)$. We want to find an instance of *SC* that has the same answer. Consider the graph $G' = (V, E')$ where the edges are

$$E' = E \cup \{(u, s) \mid u \in V\} \cup \{(t, v) \mid v \in V\}.$$

The idea is that if there exists a path from s to t, then we can get from any node u to v by going to s first, taking the path from s to t, and then going to v. The first and last steps are possible because of the edges we added.

Conversely, if there is no path from s to t in G, then there will still not be a path from s to t in G', as desired. □

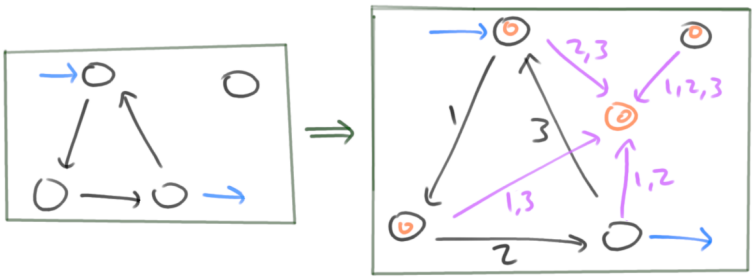
ALL_{DFA}

Recall the problem

$$ALL_{\mathsf{DFA}} = \{\mathsf{DFA}\ A \mid A \text{ accepts all strings}\}.$$

126 **Claim**
ALL_{DFA} is NL-complete.

Proof. It is clear that $ALL_{\text{DFA}} \in \text{coNL} = \text{NL}$ because we can just nondeterministically select a sequence of moves, and if we find a rejecting state we accept.



We will show that ALL_{DFA} by proving $\overline{PATH} \leq_L ALL_{\text{DFA}}$. Suppose we have some instance $\langle G, s, t \rangle$ of $PATH$ where $G = (V, E)$. Consider the DFA M where the states are $V \cup q_A$ where q_A is a new accepting state. Moreover, everything except t is an accepting state. Label each each transition with its own symbol. Then, in order to make it a DFA, all of the remaining edges will go to q_A.

If $\langle G, s, t \rangle \in \overline{PATH}$, then there is no way to get to the reject state, and $M \in ALL_{\text{DFA}}$. Conversely, if $\langle G, s, t \rangle \in PATH$, there exists some sequence of edges (and therefore indices) that reach the non-accepting state, and $M \notin ALL_{\text{DFA}}$. □

3.8 Hierarchy Theorems

So far, we have that

$$\text{L} \subseteq \text{NL} \subseteq \text{P} \subseteq \text{NP} \subseteq \text{PSPACE} = \text{NPSPACE}.$$

Today we will show that $\text{NL} \subsetneq \text{PSPACE}$. First, we will work with space as it is slightly easier, and then prove similar results for time.

3.8.1 Space

127 **Definition**
A function $f : \mathbb{N} \to \mathbb{N}$, where $f(n) = O(\log n)$ is **space constructible** if $1^n \mapsto f(n)$ is computable in $O(f(n))$ space.

128 **Example**
Most functions like $\log n$ or $\exp n$ are space constructible. However, $f(n) = \log\log\log n$ is not space constructible. In particular, if a TM runs in $O(\log\log\log n)$ space, then it can only recognize a regular language, which is basically having no tape. This implies that any TM that has a space complexity less than $O(\log\log\log n)$ can only recognize a regular language. This space is called a **gap**.

129 **Theorem** (Space hierarchy theorem)
If $f : \mathbb{N} \to \mathbb{N}$ is space constructible, there exists a language A decidable in $O(f(n))$ space but not $o(f(n))$ space.

Proof. The idea is to find a machine D that runs in space $O(f(n))$ and such that $L(D) \neq L(M)$ for all M running in $o(f(n))$ space. We can easily force D to run in $O(f(n))$ space by just using $f(n)$ space. To ensure that $L(D)$ is not decidable in $o(f(n))$ space, we will use the idea from diagonalization. In particular, If M is a machine potentially deciding $L(D)$ in $o(f(n))$ space, we will make M and D disagree on $\langle M \rangle$.

Consider the following machine:

0. On input w:

1. Let $n \leftarrow |w|$.
2. Since f is space constructible, compute $f(n)$ and mark off this much tape. If the later steps try to use more then REJECT.
3. If w is not of the form $\langle M \rangle \mathtt{10}^n$ for some TM M, then REJECT.
4. Simulate M on w for $2^{f(n)}$ steps. If it reaches this point, REJECT.
5. If M accepts, then REJECT. If M rejects, then ACCEPT.

The reason why we only run M for $2^{f(n)}$ is because we don't want it loop. We arbitrarily choose to reject if M loops (because M is not a decider anyway).

Now suppose that M runs in $o(f(n))$ space. Then consider running the machine on $\langle M \rangle$. The algorithm will not reject in steps 3 or 4, and in step 5, it will do the opposite of what M on $\langle M \rangle$ does. Therefore, M cannot decide the language of this machine. □

130 **Corollary**
NL ⊊ PSPACE.

Proof. By Savitch's theorem, NL ⊆ SPACE($(\log n)^2$). By the space hierarchy theorem, SPACE($(\log n)^2$) ⊊ SPACE(n) ⊆ PSPACE. Therefore, NL ⊊ PSPACE. □

3.8.2 Time

Our results for time will be very similar to space.

131 **Definition**
A function $t : \mathbb{N} \to \mathbb{N}$ is **time constructible** if $\mathtt{1}^n \mapsto t(n)$ is computable in $O(t(n))$ time.

132 **Theorem** (Time hierarchy theorem)
If $t : \mathbb{N} \to \mathbb{N}$ is time constructible, then there exists a language A decidable in $O(t(n))$ time but not $o(t(n)/\log n)$ time.

The reason why we have an extra factor of $\log n$ is because in order to use the same proof, we must keep a counter of the number of steps used near the head. It is possible for a stronger theorem to be true, but we cannot use the same proof as we did in space.

Proof. The proof is very similar to the proof for theorem 129. Let D be the machine:

0. On input w:
1. Let $n \leftarrow |w|$.
2. Since t is space constructible, compute $t(n)$ and make a binary counter with the value $\lceil t(n)/\log t(n) \rceil$.
3. If w is not of the form $\langle M \rangle \mathtt{10}^n$ for some TM M, then REJECT.
4. Simulate M on w for $\lceil t(n)/\log t(n) \rceil$ steps using the counter, and moving the counter with the head.
5. If M accepts, then REJECT. If M rejects, then ACCEPT.

The factor of $\log t(n)$ comes from the binary counter and moving it to the head at every step that we're simulating M. Otherwise, everything else is the same. □

3.9 Intractable Problems

Recall that we have

$$\mathsf{L} \subseteq \mathsf{NL} \subseteq \mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PSPACE},$$

where NL $\neq$ PSPACE. The problems of whether L $\stackrel{?}{=}$ P and P $\stackrel{?}{=}$ PSPACE are open, but NL $\neq$ PSPACE implies that at least one of them is false.

Recall that EQ_{DFA} is decidable, and using the same TM, we have that $EQ_{\text{DFA}} \in$ P. We can also show that $EQ_{\text{NFA}} \in$ PSPACE.

133 **Proposition**

$EQ_{\text{NFA}} \in$ PSPACE.

Proof. Since PSPACE is deterministic, $A \in$ PSPACE $\iff \overline{A} \in$ PSPACE. We proceed by showing that $\overline{EQ_{\text{NFA}}} \in$ NPSPACE = PSPACE. The idea is to nondeterministically guess the string that yields different results for the two NFAs. It is clear that this runs in polynomial space. □

Similarly, we have a similar theorem for regular expressions:

134 **Corollary**

$EQ_{\text{REGEX}} \in$ PSPACE.

Proof. This is immediately because the conversion from REGEX $\to$ NFA can be done in polynomial space. □

3.9.1 Exponential Space

Consider a modified version of regular expressions that has exponentiation, i.e.

$$R^i = \underbrace{RR\dots R}_{i \text{ times}}.$$

Let a regular expression with this feature be called an REGEX↑.

135 **Definition**

Let the languages decidable in exponential time and space be, respectively

$$\text{EXPTIME} = \bigcup_k \text{TIME}(2^{n^k})$$
$$\text{EXPSPACE} = \bigcup_k \text{SPACE}(2^{n^k}).$$

136 **Definition**

We say that a language B is **EXPSPACE-complete** if

- $B \in$ EXPSPACE and
- for all $A \in$ EXPSPACE, we have $A \leq_p B$, i.e. is **EXPSPACE-hard**.

137 **Theorem**

$EQ_{\text{REGEX}\uparrow} = \{\langle R_1, R_2\rangle \mid R_1 \text{ and } R_2 \text{ are REGEX}\uparrow \text{ such that } L(R_1) = L(R_2)\}$ is EXPSPACE-complete.

Proof. We will show that $EQ_{\text{REGEX}\uparrow}$ is EXPSPACE-complete. It is clear that $EQ_{\text{REGEX}\uparrow} \in$ EXPSPACE because $EQ_{\text{REGEX}} \in$ PSPACE. In particular, we can expand the REGEX↑ into a REGEX in exponential space, and then use the PSPACE algorithm for EQ_{REGEX} on the exponentially sized REGEXs.

Now consider some $A \in \text{EXPSPACE}$ decided by a TM M that runs in space $O(2^{n^k})$. Consider the reduction mapping $w \mapsto \langle R_1, R_2 \rangle$ where R_1 is a REGEX↑ that generates all strings, and R_2 is a REGEX↑ that generates all strings except for the rejecting computation history of M on w. Then M accepts w iff $L(R_1) = L(R_2)$, as desired.

To actually construct R_2, we can just check to make sure that the computation history makes a mistake. Let $\Delta = \Gamma \cup Q \cup \{\#\}$ be the alphabet for the configurations. Consider the configuration history

$$c_1 \# c_2 \# \dots \# c_{\text{reg}},$$

where we pad all configurations at the end with $\sqcup$ for convenience. Then let

$$R_2 = R_{\text{bad-start}} \cup R_{\text{bad-move}} \cup R_{\text{bad-reject}},$$

where the parts are defined as follows (where we adopt the notation $X - a \coloneqq X \setminus \{a\}$):

- Let $R_{\text{bad-start}} = S_0 \cup S_1 \cup S_2 \cup S_n \cup S_{\text{blanks}}$, where S_i means that the ith character is incorrect. In particular,
$$S_i = \Delta^i (\Delta - w_i) \Delta^* \qquad \text{and} \qquad S_{\text{blanks}} = \Delta^{n+1} (\Delta - \varepsilon)^{2^{n^k} - n - 2} (\Delta - \sqcup) \Delta^*.$$
Note that we have used the exponentiation function of the REGEX↑ so that $R_{\text{bad-start}}$ is not exponentially long.
- Similarly, let $R_{\text{bad-reject}} = (\Delta - q_{\text{reject}})^*$.
- For $R_{\text{bad-move}}$, we can use the same idea as we did in Cook-Levin. We look at three consecutive symbols in a configuration as well as the configuration after it, and we reject the sets of symbols that do not correspond to a legal move. In particular,
$$R_{\text{bad-move}} = \bigcup_{\text{bad } (abc, def)} \Delta^* abc \Delta^{2^{n^k} - 2} def \Delta^*.$$

Note that R_2 is polynomial in length, as desired. □

3.10 Oracles

138 **Definition**

Given a language A, an **A-oracle machine** (A-OM) is a TM that magically knows the answer to A.

One way to formalize this is to add two extra tapes to our regular TMs. One request tape and one answer tape. When we change the request tape, the answer tape changes to the answer of the instance on the request tape immediately, and we can then read from it.

Even though we can never have oracles in real life, we study oracles because if we can show that we can't solve a problem even with an oracle, then we know the problem is really hard.

139 **Definition**

Let P^A be the class of languages solvable in polynomial time by an A-OM. If $L \in \text{P}^A$, then we say that we can solve L in polynomial time relative to SAT. We can similarly define NP^A.

140 **Example**

A SAT-OM can solve SAT. In particular, we can copy the input into the request tape, and then accept or reject according to the response we get on the answer tape.

Note that this means that

$$\text{NP} \subseteq \text{P}^{SAT}.$$

Having an oracle feels like performing reductions, but it is more powerful. In particular, we can request multiple instances of a problem. Moreover, we know that $\mathsf{P}^{SAT} = \mathsf{co}(\mathsf{P}^{SAT}) = \mathsf{coP}^{SAT}$ because we are working with determinism, so we also have $\mathsf{NP} \subseteq \mathsf{coP}^{SAT} \implies \mathsf{coNP} \subseteq \mathsf{P}^{SAT}$.

141 **Example**

Let $MINFORM = \{\langle\phi\rangle \mid \phi \text{ is a minimal formula}\}$. We claim $MINFORM \in \mathsf{coNP}^{SAT}$.

The idea is to nondeterministically guess a shorter formula φ, and then use the oracle to check whether they are equivalent by writing the formula $(\phi \Rightarrow \varphi) \wedge (\varphi \Rightarrow \phi)$.

3.10.1 Limits on Diagonalization

It turns out that the problem of whether $\mathsf{P}^{SAT} \stackrel{?}{=} \mathsf{NP}^{SAT}$ is open. However, we have the following surprising fact:

142 **Proposition**

$\mathsf{P}^{TQBF} = \mathsf{NP}^{TQBF} \subseteq \mathsf{NPSPACE}$.

Proof. We know that $\mathsf{P}^{TQBF} \subseteq \mathsf{NP}^{TQBF}$. Since *TQBF* is PSPACE-complete, we have $\mathsf{PSPACE} \subseteq \mathsf{P}^{TQBF}$. Therefore, we have the chain of inclusions

$$\mathsf{NP}^{TQBF} \subseteq \mathsf{NPSPACE} = \mathsf{PSPACE} \subseteq \mathsf{P}^{TQBF},$$

where the first inclusion is from the fact that $TQBF \in \mathsf{NPSPACE}$, so an NPSPACE machine can just solve *TQBF* itself instead of asking the oracle. Therefore, $\mathsf{P}^{TQBF} = \mathsf{NP}^{TQBF}$. □

Note that if we can use a diagonalization argument to show that $\mathsf{P} \neq \mathsf{NP}$, then we can just use the same argument to show that $\mathsf{P}^A \neq \mathsf{NP}^A$. However, this is a contradiction for $A = TQBF$, so we cannot use a pure diagonalization argument to prove $\mathsf{P} \neq \mathsf{NP}$.

143 **Question**

Recall that A_{TM} is undecidable from diagonalization. Why doesn't this rule out the existence of A_{TM}-OMs?

Diagonalization only states that A_{TM} is undecidable by regular TMs, and that $A_{L\text{-OM}}$ is undecidable by L-OMs.

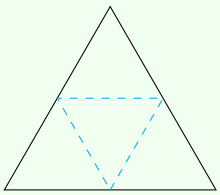
3.10.2 More Oracle Problems

Suppose we have two friends (but not that good of friends that they trust each other) Al and Bo (short for Alice and Bob) that want to split cake fairly.

The protocol goes as follows. There is a cake and several predetermined cuts. Al selects a subset of the cuts, and the cake is cut along those curves. If this process produces n pieces of cake, Bo then selects $\lceil n/2 \rceil$ pieces.

144 **Example**

If this was the original cake with predetermined cuts, Al can select all the cuts, and Bo must select two pieces, guaranteed to have equal area, so Al can guarantee fair cuts.



Let the problem be

$$CAKE = \{(\text{cakes, allowed cuts}) \mid \text{Al can guarantee fair cuts}\}$$

145 **Proposition**

$CAKE \in \mathsf{NP}^{UNSAT}$, i.e. using an NTM with an *UNSAT* oracle, we can solve *CAKE*.

Proof. Consider the following algorithm. Let the NTM nondeterministically select the subset of cuts that Al chooses. We have some pieces of cake with certain volumes. Consider the formula that says "the volumes of the partition are unequal", where the variables are indicator variables of whether a piece is in the partition. Then, use the oracle to determine whether this formula is unsatisfiable. If so, then any choice Bo will make must be fair. □

146 **Example**

Consider the problem

$$GETSAT = \{\langle \phi, i, b\rangle \mid \phi \text{ is satisfiable where in the minimal satisfying assignment, the } i\text{th variable is } b\}.$$

Then $GETSAT \in \mathsf{P}^{SAT}$ because we can substitute in b for the ith variable of ϕ, and then ask the oracle whether the new formula is satisfiable.

3.11 Probabilistic Turing Machines

147 **Definition**

Let a **probabilistic Turing machine** (PTM) be a decider NTM where at each nondeterministic step, there are two options, and there is a $\frac{1}{2}$ probability of choose each option.

For each branch, we can consider the probability that the machine has taken it, i.e. 2^{-k}, where k is the number of nondeterministic states in that branch. Then, we define

$$\mathbb{P}[M \text{ accepts } w] \coloneqq \sum_{\substack{\text{accepting} \\ \text{branches } b}} \mathbb{P}[M \text{ takes branch } b],$$

and $\mathbb{P}[M \text{ rejects } w] \coloneqq 1 - \mathbb{P}[M \text{ accepts } w]$.

148 **Definition**

For some $\varepsilon > 0$, we say that a PTM M decides A with error probability ε if:

- for all $w \in A$, $\mathbb{P}[M \text{ accepts } w] > 1 - \varepsilon$ and
- for all $w \notin A$, $\mathbb{P}[M \text{ rejects } w] > 1 - \varepsilon$.

Then we can define a new complexity class.

149 **Definition** (Bounded Probabilistic Polynomial Time)
Let BPP = $\{A \mid$ some polynomial time PTM decides A with error probability $1/3\}$.

The $\frac{1}{3}$ in the definition seems a bit strange. It turns out that the choice of the constant here does not matter!

150 **Proposition** (Amplification Lemma)
For $0 < \varepsilon_1, \varepsilon_2 < \frac{1}{2}$, any PTM M_1 with error probability ε_1 has an equivalent PTM M_2 with error probability ε_2.

Proof (Sketch). M_2 can just run M_1 multiple times, and return the answer that appears the most often. □

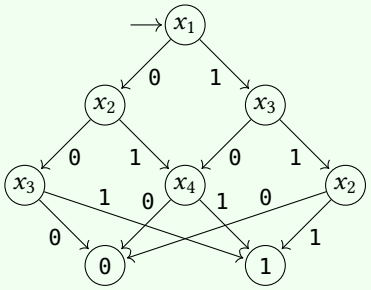
Therefore, we have the equivalent definition

$$\text{BPP} = \left\{ A \;\middle|\; \text{some polynomial time PTM decides } A \text{ with error probability } \varepsilon < \frac{1}{2} \right\}.$$

3.12 Branching Programs

151 **Definition**
A **branching program** (BP) is a directed acyclic graph where all nodes are labeled with a variable, and have two outgoing edges labeled `0` and `1`. There are two other output nodes labeled `0` and `1` that have no outgoing edges. Branching programs generally represent a boolean function, where we query each variable when we get to each node and follow the corresponding edge.

152 **Example** (Branching program)



We call a branching program **read-once** if every path from the start to one of the output nodes contains each variable at most once. We will abbreviate read-once branching programs as ROBPs. Then consider the problem EQ_{ROBP}.

153 **Claim**
$EQ_{\text{ROBP}} \in$ BPP.

One attempt at proving this is to try the following algorithm:

0. On input $\langle B_1, B_2 \rangle$:
1. Pick random boolean assignments to $x_1, \ldots, x_m$ and run B_1, B_2 on these assignments.
2. If they disagree then REJECT, else ACCEPT.

It turns out that this does not work because the two branching programs might only disagree on one input, so the algorithm will get that specific instance wrong with probability $1 - 2^m$.

Instead to prove this, we will need a lemma that follows from the fundamental theorem of algebra.

154 **Lemma** (Schwartz-Zippel Lemma)

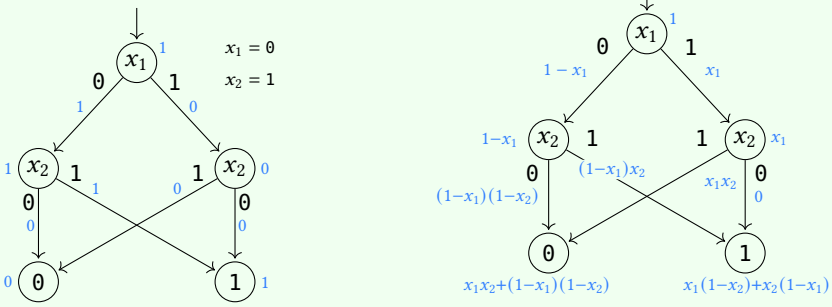
If $p \in \mathbb{F}_q[x_1, \ldots, x_m]$ is nonzero and has degree at most d in each variable, then over $r_1, \ldots, r_m \in \mathbb{F}_q$,

$$\mathbb{P}[p(x_1, \ldots, x_m) = 0] \leq \frac{md}{q}.$$

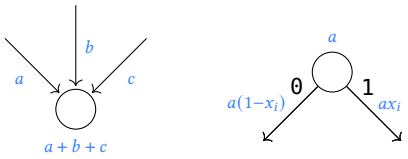
Proof (Sketch). Induct on m. Write $p(x_1, \ldots, x_m)$ as a polynomial in $F[x_1, \ldots, x_{m-1}][x_m]$ and use the fundamental theorem of algebra. □

Recall that we can label each object (nodes and edges) of a BP with `1`s and `0`s depending on which ones are used during the execution. We can even do this for general variables and also algebraically.

155 **Example**



More generally, we have the following labeling rules



Then, the formula the corresponds to the `1` output is the one the represents the BP.

156 **Claim**

For ROBPs B_1 and B_2 with polynomials P_1 and P_2, then over $r_1, \ldots, r_m \in \mathbb{F}_q$,

$$\mathbb{P}[P_1(r_1, \ldots, r_m) = P_2(r_1, \ldots, r_m)] = \begin{cases} 1 & B_1 \text{ and } B_2 \text{ are equivalent} \\ \leq \frac{m}{q} & \text{else.} \end{cases}$$

Proof (Sketch). Since B_1 and B_2 are read-once, the degree of the polynomials P_1 and P_2 in each variable is at most 1.

Now suppose that B_1 and B_2 are equivalent. Note that P_1 and P_2 can be written as sums of terms of the form $y_1 y_2 \cdots y_m$, where y_i is either x_i or $1 - x_i$. (If a certain variable x_i is not on a path, we can use the fact that $1 = x_i + (1 - x_i)$ and expand.) Moreover, in this form, the polynomials give us precisely the inputs of the ROBPs that output 1. Therefore, P_1 and P_2 are the same in this case.

Now suppose that B_1 and B_2 are not equivalent. Then note that $P_1 - P_2 \neq 0$ from looking at the representation of P_1 and P_2 with terms of the form $y_1 y_2 \cdots y_m$ as before.

The $\frac{md}{q}$ is from Schwartz-Zippel. Since these are read-once BPs, we have $d = 1$, so $\mathbb{P} = \frac{m}{q}$. We can choose $q \geq 3m$, so $\mathbb{P} \leq \frac{1}{3}$, as desired. □

157 **Corollary**

$EQ_{\text{ROBP}} \in \text{BPP}$.

3.13 Interactive Proofs

Consider the problem of graph isomorphism (denoted by $G_1 \cong G_2$):

$$ISO = \{\langle G_1, G_2\rangle \mid G_1 \cong G_2\}.$$

It is clear that $ISO \in \text{NP}$ because a certificate would just be a bijection of the vertices. However, ISO is not known to be NP-hard or in P.

Suppose that we have two graphs G_1 and G_2. Also suppose that we have two parties named P (prover) and V (verifier). If V runs in probabilistic polynomial time, and P has unlimited computation power, then how can P prove to V that $G_1 \cong G_2$ or $G_1 \ncong G_2$?

Consider the following protocol:

1. P claims either $G_1 \cong G_2$ or $G_1 \ncong G_2$.
2. If P claimed $G_1 \cong G_2$, then P provides the certificate, and V checks it. V either ACCEPTS or REJECTS at this case.
3. If P claims $G_1 \ncong G_2$, then V repeats the following procedure N times:
    a) V secretly takes one of G_1, G_2 randomly and scrambles it to another graph G isomorphic to the selection.
    b) V asks P which of G_1, G_2 that G is isomorphic to.
4. If P gets all of them right, then V ACCEPTS, else REJECT.

Note that if P lies in the first case, then V will know with the certificate. In the other case, V will catch the lie with probability $1 - 2^{-N}$.

158 **Definition**

Consider two parties P unlimited computation and V with probabilistic polynomial computation respectively. Both exchange messages until V accepts or rejects. Let

$$\mathbb{P}[(V \leftrightarrow P) \text{ accepts } w] \coloneqq \mathbb{P}[V \text{ ends up accepting}].$$

159 **Definition**

Let IP (for **interactive proofs**) be

$$\begin{aligned}\text{IP} \coloneqq \{A \mid &\text{there exists a protocol } (V \leftrightarrow P) \text{ where} \\ &w \in A \iff \mathbb{P}[(V \leftrightarrow P) \text{ accepts } w] \geq \tfrac{2}{3} \\ &w \notin A \iff \mathbb{P}[(V \leftrightarrow \tilde{P}) \text{ accepts } w] \leq \tfrac{1}{3} \text{ for all } \tilde{P}\}.\end{aligned}$$

We know that $\text{NP} \subseteq \text{IP}$ because the prover can just send the certificate.

We also know that $\text{BPP} \subseteq \text{IP}$ because the verifier can tell the prover to go away and just do the problem themself.

It turns out that $\text{IP} \subseteq \text{PSPACE}$, but we will not prove this. The idea of the proof is to explore all possible interactions in polynomial space. We will prove the weaker statement $\text{coNP} \subseteq \text{IP}$ by showing $\#SAT \in \text{IP}$. (Note that $\#SAT$ is coNP-hard because $\overline{SAT} \leq_p \#\text{SAT}$ by the reduction $\langle\phi\rangle \to \langle\phi, 0\rangle$.)

3.14 #SAT ∈ IP

Consider the problem

$$\#SAT = \{\langle \phi, k \rangle \mid \phi \text{ has exactly } k \text{ satisfying assignments}\}.$$

For a formula ϕ on variables $x_1, \ldots, x_m$ and $a_1, \ldots, a_i \in \{\mathtt{0}, \mathtt{1}\}$, let $\#\phi(a_1, \ldots, a_i)$ be the number of satisfying assignments with presets $x_1 = a_1, x_2 = a_2, \ldots, x_i = a_i$. Then,

- $\#\phi(a_1, \ldots, a_i) = \#\phi(a_1, \ldots, a_i, \mathtt{0}) + \#\phi(a_1, \ldots, a_i, \mathtt{1})$, and
- $\#\phi(a_1, \ldots, a_m) = \phi(a_1, \ldots, a_m)$.

There is a straightforward exponential protocol for #SAT. Suppose that P claims that $\langle \phi, k \rangle \in \#SAT$.

0. P sends $\#\phi()$, and V checks $k = \#\phi()$.
1. P sends $\#\phi(\mathtt{0})$, $\#\phi(\mathtt{1})$, and V checks $\#\phi() = \#\phi(\mathtt{0}) + \#\phi(\mathtt{1})$.
2. P sends $\#\phi(\mathtt{00})$, $\#\phi(\mathtt{01})$, $\#\phi(\mathtt{10})$, $\#\phi(\mathtt{11})$, and V checks $\#\phi(\mathtt{0}) = \#\phi(\mathtt{00}) + \#\phi(\mathtt{01})$ and $\#\phi(\mathtt{1}) = \#\phi(\mathtt{10}) + \#\phi(\mathtt{11})$.

⋮

m. P sends $\#\phi(\mathtt{0}^m), \ldots, \#\phi(\mathtt{1}^m)$, and V checks $\#\phi(\mathtt{0}^{m-1}) = \#\phi(\mathtt{0}^m) + \#\phi(\mathtt{0}^{m-1}\mathtt{1}), \ldots, \#\phi(\mathtt{1}^{m-1}) = \#\phi(\mathtt{1}^{m-1}\mathtt{0}) + \#\phi(\mathtt{1}^m)$.

$m+1$. V checks $\#\phi(\mathtt{0}^m) = \phi(\mathtt{0}^m), \ldots, \#\phi(\mathtt{1}^m)\phi(\mathtt{1}^m)$.

The problem is that at every step, we are doubling the number of things that the prover is sending, because we have to extend the preset variables by both $\mathtt{0}$ and $\mathtt{1}$. To fix this, the key idea is to extend by a non-boolean value instead. We will use the standard arithmetization:

- $x \wedge y \to xy$
- $\overline{x} \to 1 - x$
- $x \vee y \to x + y - xy$
- $\phi \to$ some polynomial P_ϕ with degree less than $|\phi|$

Moreover, we can extend $\#\phi$ to be

$$\#\phi(a_1, \ldots, a_i) = \sum_{a_{i+1}, \ldots, a_m \in \{\mathtt{0},\mathtt{1}\}} P_\phi(a_1, \ldots, a_i, a_{i+1}, \ldots, a_m)$$

over $a_1, \ldots, a_i \in \mathbb{F}_q$. Note that we still have

- $\#\phi(a_1, \ldots, a_i) = \#\phi(a_1, \ldots, a_i, \mathtt{0}) + \#\phi(a_1, \ldots, a_i, \mathtt{1})$, and
- $\#\phi(a_1, \ldots, a_m) = \phi(a_1, \ldots, a_m)$.

Now consider the new protocol:

0. P sends $\#\phi()$ and V checks $k = \#\phi()$.
1. a) P sends $\#\phi(z)$ as a polynomial and V checks $\#\phi() = \#\phi(\mathtt{0}) + \#\phi(\mathtt{1})$. V can get $\#\phi(\mathtt{0})$ and $\#\phi(\mathtt{1})$ by plugging into $\#\phi(z)$.
   b) V requests a random $r_1 \in \mathbb{F}_q$ and P sends $\#\phi(r_1)$.
2. a) P sends $\#\phi(r_1, z)$ and V checks $\#\phi(r_1) = \#\phi(r_1, \mathtt{0}) + \#\phi(r_1, \mathtt{1})$.
   b) V requests a random $r_2 \in \mathbb{F}_q$ and P sends $\#\phi(r_1, r_2)$.

⋮

m. a) P sends $\#\phi(r_1, \ldots, r_{m-1}, z)$ and V checks $\#\phi(r_1, \ldots, r_{m-1}) = \#\phi(r_1, \ldots, r_{m-1}, \mathtt{0}) + \#\phi(r_1, \ldots, r_{m-1}, \mathtt{1})$.
   b) V requests a random $r_m \in \mathbb{F}_q$ and P sends $\#\phi(r_1, \ldots, r_m)$.

$m+1$. V checks $\#\phi(r_1, \ldots, r_m) = P_{\phi(r_1, \ldots, r_m)}$ and ACCEPTS if true.