CS 172 Lecture Notes

Alec Li

Fall 2022 — Professor Avishay Tal

Contents

Definitions

8/25/2022

Lecture 1

Introduction

1.1 Overview

What is computation? "Computation is the evolution process of some environment, by a sequence of simple, local steps."

Some examples of such environments include:

- bits in a computer
- computers in a network
- atoms in matter
- neurons in the brain
- prices in a market
- cells in a tissue

In the course, we'll be talking about three major topics:

- **Automata Theory**: With automata theory, we're talking about mathematical models of computation; we're abstracting computation to some mathematical model; in particular, we're talking about two models
  - Finite state automata: computation with constant memory
  - Context-free grammars: computation with a stack; this is particularly important in the context of compilers
- **Computability Theory**: With computability theory, we are concerned with what problems computers can and cannot solve.

  In particular, you may be familiar with the halting problem: determining whether any program will halt on a given input; this problem is undecidable.

  It's perhaps more interesting to think about how quickly a computer can solve things; ex. some computations may be instant, and some computations may take thousands of years. This leads into the third topic.
- **Complexity Theory**: With complexity theory, we are concerned with what problems computers can solve efficiently. In contrast to the definitive nature of decidability, there is still an open problem of P vs. NP.

1.1.1 Models of Computation

Mathematical modeling of computation gives us systematic ways to think and argue about computers.

Here, we're picking models that are simple; we want these models to be accurate in some aspects, but not accurate in others—this tradeoff allows us to study certain aspects of computation more easily.

1.1.2 First Lectures

We'll first talk about DFA: *deterministic finite automata*. This is a very simple model of computation, and we'll be able to say a lot about this model, demonstrating a lot of concepts that would be more complicated in stronger models.

The set of all languages that correspond to DFA is called the *regular languages*.

We will then talk about closure properties of regular languages; unions, concatenations, intersections, etc. of different languages.

This brings us to NFA: *nondeterministic finite automata*. It turns out that we can establish an equivalence between NFA and DFA; that is, any NFA can correspond to a DFA with a constant number of states. This allows us to establish closure properties under all languages.

Finally, we will show that regular languages correspond with *regular expressions*; the power of regular expressions is exactly the same as the power of regular languages, and we can convert DFA into regular expressions, and we can convert regular expressions into DFA.

This means that we have three views on regular languages, each giving us a different view on the same objects.

1.2 Deterministic Finite Automata

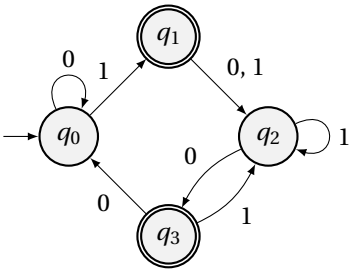


Figure 1.1: An example of a DFA.

Here, double circles denote which states are *accepting*; q_1 and q_3 are accepting, and q_0 and q_2 are not accepting. Here, q_0 is a starting state. The edges are transitions between states.

Suppose we give an input of `0110`. We start with a `0`, so we stay in q_0, the `1` makes us move to q_1, the next `1` makes us move to q_2, and the last `0` makes us move to q_3.

We call this path the computation path. The computation *accepts* if and only if this computation path ends in an *accepting state*.

1.2.1 Why DFA?

We'll be talking about context-free grammars, Turing machines, and circuits later on; why do we talk about DFA?

It's a simple model of computation; it serves more of a warm-up to more complicated models. For example, Turing machines are DFAs with some "extra features", and gives us a taste a lot of other concepts, ex. "non-determinism", and allows us to give a taste of limitations. With DFA, we'll be able to be very precise and accurate about its limitations, ex. some tasks are possible with x states, but not with $x-1$ states.

DFAs are also similar to streaming algorithms, where we have a high frequency stream of data, and only a small amount of memory. The output of the streaming algorithm gives us statistics about the stream.

1.3 Notation

Here is some vocabulary and notation regarding languages.

Definition 1.1: Alphabet

The *alphabet* Σ is a finite set, ex. $\Sigma = \{0,1\}$ or $\Sigma = \{a,b,c,d,\ldots,z\}$

Definition 1.2: String

A *string over* Σ is a finite length sequence of symbols from Σ.

We denote $|x|$ as the *length* of the string, Σ^* is the set of all strings over Σ, and ε is the empty string, i.e. $|\varepsilon| = 0$.

Definition 1.3: Language

A *language over* Σ is a set of strings over Σ.

Definition 1.4: Deterministic Finite Automata

A DFA (Deterministic Finite Automata) is a 5-tuple

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

- Q is the set of states (a finite set)
- Σ is the alphabet (a finite set)
- $\delta : Q \times \Sigma \to Q$ is the transition function
- $q_0 \in Q$ is the start state
- $F \subseteq Q$ is the set of accepting/final states

Example 1.5

Taking Fig. 1.1 as an example, we have

- $Q = \{q_0, q_1, q_2, q_3\}$
- $\Sigma = \{0, 1\}$
- δ can be thought of as a table:

	0	1
q_0	q_0	q_1
q_1	q_2	q_2
q_2	q_3	q_2
q_3	q_0	q_2

  That is, the rows correspond to the current state, and we look at the column corresponding to the current character, giving us the next state.

- q_0 is the starting state
- $F = \{q_1, q_3\}$

Here, it should be noted that we always know exactly where we go, since this is a deterministic automata; δ is always defined for every possible state and character input.

Definition 1.6: Computation Path

Let $w = w_1 w_2 w_3 \cdots w_n$ be a string in Σ^* of length n.

The *computation path* of M on w is the sequence $r_0, r_1, \ldots, r_n \in Q$ defined by:

$$\begin{cases} r_0 = q_0 \\ r_i = \delta(r_{i-1}, w_i) \quad \text{for } i = 1, 2, \ldots, n \end{cases}$$

Definition 1.7: Accepting

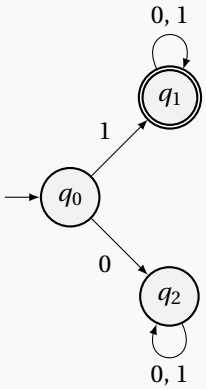
For a DFA M and computation path $r_0, r_1, \ldots, r_n \in Q$ of M on w, M accepts w if and only if $r_n \in F$.

Definition 1.8: Language

If M is a DFA, then $L(M)$ is the set of all strings that M accepts, i.e. the language recognized/computed by M.

Example 1.9

Consider the following DFA:



Here, this DFA does not accept `011`, but it does accept `101`.

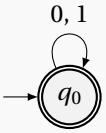
In general, we have

$$L(M) = \{w \in \{0,1\}^* \mid w \text{ starts with } 1\}.$$

Note that this is an infinite language, but it can be described with a DFA.

Example 1.10

Consider the following DFA:



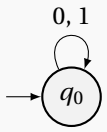
In this case, we have

$$L(M) = \Sigma^* = \{0,1\}^*.$$

That is, the language consists of all possible strings over $\Sigma = \{0,1\}$.

Example 1.11

Consider the following DFA:



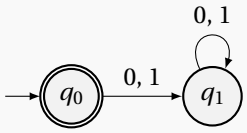
In this case, we have

$$L(M) = \varnothing.$$

Further, note that ε is not accepted ($\varnothing$ is not the same as $\{\varepsilon\}$).

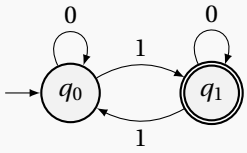
Example 1.12

How would we make an automata that accepts only the empty string?



Example 1.13

Consider the following DFA:



We claim that $L(M) = \{w \in \{0,1\}^* \mid w \text{ has an odd number of 1's}\}$. While we can argue this by following the execution procedure, we will prove this formally through induction.

For ease, suppose $A = \{w \in \{0,1\}^* \mid w \text{ has an odd number of 1's}\}$.

Proof. We claim that $\forall n \in \mathbb{N}$, the set of strings of length n accepted by M is

$$A_n = \{w \in \{0,1\}^* \mid w \text{ has an odd number of 1's, and } |w| = n\}.$$

Base case: For $n = 0$, $\varepsilon \notin L(M)$, and $\varepsilon \notin A_0$.

Inductive Hypothesis: Suppose the claim holds for all strings of length n.

Inductive Step: We will show the claim holds for strings of length $n+1$.

Suppose we have a string $w = w_1 \cdots w_n w_{n+1}$ of length $n+1$. We know from the IH that after reading the first n steps, we reach an accepting state if and only if the number of 1's is odd. At this point, we have four cases, depending on the last character (w_{n+1}) and whether $w_1 \cdots w_n$ is accepted.

If $w_1 \cdots w_n \in L(M)$ and $w_{n+1} = 0$, then we transition from q_1 to q_1; we still have an odd number of 1's, and we still accept w.

If $w_1 \cdots w_n \in L(M)$ and $w_{n+1} = 1$, then we transition from q_1 to q_0; we now have an even number of 1's, and we no longer accept w.

If $w_1 \cdots w_n \notin L(M)$ and $w_{n+1} = 0$, then we transition from q_0 to q_0; we still have an even number of 1's, and we still do not accept w.

If $w_1 \cdots w_n \notin L(M)$ and $w_{n+1} = 1$, then we transition from q_0 to q_1; we now have an odd number of 1's, and we now accept w. □

Proof. Another method is to show that $A = \{w \mid w \text{ has an odd number of 1's}\}$ is accepted, and it is the only set of strings that are accepted. That is, we want to show that $A = L(M)$; to do so, we need to show that $A \subseteq L(M)$ and $L(M) \subseteq A$.

To show that $A \subseteq L(M)$, we need to show that all strings with an odd number of 1's will be accepted. Looking at the execution path, we only transition to a different state when we read a 1. Since we start at q_0, we will always end up at q_1 if we have an odd number of 1's.

To show that $L(M) \subseteq A$, we need to show that all strings that are accepted must have an odd number of 1's. It is perhaps easiest to look at the contrapositive: if $x \notin A$, then we end at q_0. As such, suppose we have an even number of 1's in the string. This means that we make an even number of transitions between the two states, so we will always end up at q_0, as desired.

Together, since $A \subseteq L(M)$ and $L(M) \subseteq A$, we've shown that $A = L(M)$. □

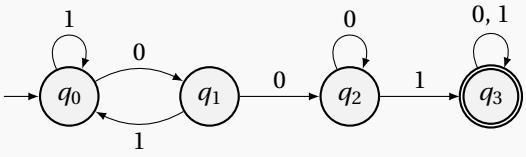
8/30/2022

Lecture 2

Constructing DFAs; NFAs

Example 2.1

Design a DFA that accepts all strings that contain "001" as a substring.

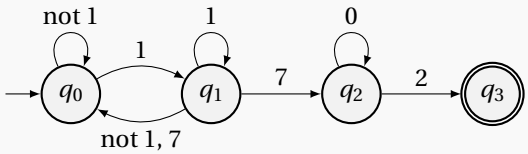


Here, we have the following states:

- q_0: we have seen nothing
- q_1: we have seen a 0
- q_2: we have seen a 01
- q_3: we have seen a 001; accept

Example 2.2

Design a DFA that accepts all strings ending in 172.



2.1 Regular Languages

Definition 2.3: Regular Language

A language L' is regular if L' is recognized by some DFA. That is, there exists a DFA M such that $L' = L(M)$.

Example 2.4

We've just seen that

$$\{w \mid w \text{ ends in } 001\}$$

is regular (from Example 2.1), and we've seen that

$$\{w \mid w \text{ has an odd number of 1's}\}$$

is also regular (from Example 1.13).

2.1.1 Closure Properties of Regular Languages

Let A, and B are two languages over an alphabet Σ. We have the following operations:

- Union: $A \cup B = \{w \mid w \in A \vee w \in B\}$
- Intersection: $A \cap B = \{w \mid w \in A \wedge w \in B\}$
- Complement: $\overline{A} = \{w \in \Sigma^* \mid w \notin A\}$
- Reverse: $A^R = \{w^R \mid w \in A\}$, i.e. if $w = w_1 w_2 \cdots w_n$, then $w^R = w_n \cdots w_2 w_1$
- Concatenation: $A \cdot B = \{vw \mid v \in A, w \in B\}$, i.e. the set of all possible concatenations between A and B
- Star: $A^* = \{s_1 s_2 \cdots s_k \mid k \geq 0 \text{ and each } s_i \in A\}$, i.e. the set of all possible (finite) concatenations of strings in A.

Theorem 2.5: Closure Properties of Regular Languages

If A and B are regular languages, then so are $A \cup B$, $A \cap B$, $\overline{\vec{A}}$, A^R, $A \cdot B$, and A^*.

Theorem 2.6: Union of Regular Languages

The union of any two regular languages is also a regular language. In other words, the class of languages is closed under the union operator.

Proof. Let L_1 and L_2 be two regular languages over Σ.

Let $M_1 = (Q_1, \Sigma, \delta_1, q_0^{(1)}, F_1)$ be a DFA recognizing L_1, and let $M_2 = (Q_2, \Sigma, \delta_2, q_0^{(2)}, F_2)$ be a DFA recognizing L_2.

We want to construct a DFA $M = (Q, \Sigma, \delta, q_0, F)$ that recognizes $L = L_1 \cup L_2$.

The idea here is to run both DFAs "in parallel".

Here, we construct a M as follows:

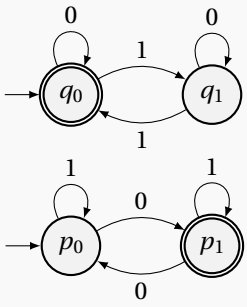
- $Q = \{(q_1, q_2) \mid q_1 \in Q_1, q_2 \in Q_2\} = Q_1 \times Q_2$
- Σ does not change
- $q_0 = (q_0^{(1)}, q_0^{(2)})$

- $F = \{(q_1, q_2) \mid q_1 \in F_1 \text{ or } q_2 \in F_2\}$
- $\delta : Q \times \Sigma \to Q$ is defined by
$$\delta((q_1, q_2), \sigma) = (\delta_1(q_1, \sigma), \delta_2(q_2, \sigma)).$$
That is, we make the transition in both DFAs at the same time.

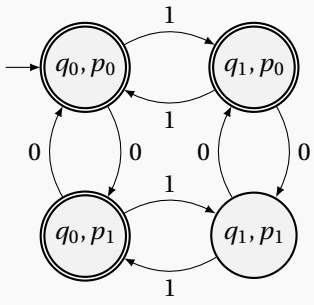
□

Example 2.7

Suppose we have the following two DFAs:



The DFA for the union would be:



Theorem 2.8: Intersection of Regular Langauges

If L_1 and L_2 are regular languages, then so is $L_1 \cap L_2$.

Proof. The proof follows exactly as with the union; the only change is in the accepting states:

$$F = \{(q_1, q_2) \in Q \mid q_1 \in F_1 \text{ and } q_2 \in F_2\}.$$

□

Theorem 2.9: Complement of Regular Languages

If L is a regular language, then so is $\overline{L} = \{w \in \Sigma^* \mid w \notin L\}$

Proof. If $M = (Q, \Sigma, \delta, q_0, F)$ recognizes L, then $M' = (Q, \Sigma, \delta, q_0, Q \setminus F)$ recognizes $\overline{L}$. □

2.2 Non-deterministic Finite Automata

Theorem 2.10: Reverse of Regular Languages

If L is a regular language, then so is $L^R = \{w_1 \cdots w_k \mid w_k \cdots w_1 \in L \text{ and each } w_i \in \Sigma\}$

An initial proof attempt for the reverse of a language is as follows:

Proof (attempt). Suppose L is a regular language. Let M be a DFA that recognizes L; we want to build a DFA M^R that recognizes L^R.

The idea here is that if M accepts w, then w describes a computation path in M from the start state to an accepting state:

$$q_0 = r_0 \xrightarrow{w_1} r_1 \xrightarrow{w_2} r_2 \cdots \xrightarrow{w_k} r_k \in F.$$

To accept the reverse string, it may seem natural to just reverse everything, and make the initial state the accepting state, and make the accepting state the initial state.

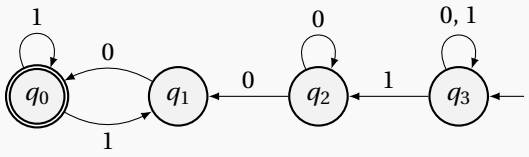
There are a few issues here:

- If there are multiple accepting states, we cannot have multiple initial states in the reverse DFA
- In the original DFA, we know that there is always exactly one outgoing edge for every combination of state and symbol, but it is not the case that every state has an incoming edge from each symbol; reversing the edges means that the resulting DFA does not have exactly one outgoing edge for every combination of state and symbol.

□

Example 2.11

Let us look at what this attempted transformation does to our 001 case from Example 2.1.



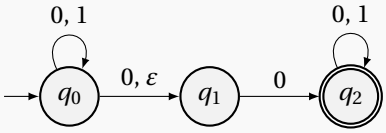
We want this DFA to accept 100, but although there *is* a path to an accepting state, there are many other paths for 100, and some paths (ex. for 111) are undefined.

To resolve this issue, we want to say that M^R accepts a string if there is *some* computation path (consistent with the string) from *some* start state to *some* accepting state.

Notice that this isn't a DFA anymore; there is no longer any deterministic transition to take from a given state and symbol. This is a *nondeterministic finite automata* (NFA). In other words, we need to guess which transition to take.

Example 2.12

Here is another example of an NFA, now with more capabilities. Specifically, we can make ε-moves; we have the ability to make a step without reading anything.



What is the set of strings accepted by this NFA?

We can see that the NFA does not accept ε, as the furthest we can go is 1 by taking the ε-move.

We can also see that the NFA accepts the string "0", as we can take the ε-move and then transition to q_2 and accept.

We can also see that the NFA does not accept the string "1"; we can either stay put and do not accept, or we can take the ε-move and then get stuck.

In general, this NFA accepts all strings that contain a 0. If we contain a 0, then we stay put for all the 1's prior, and take the ε-move and transition with the 0 to q_2. Further, if we accept the string, the only way to get to the accepting state is through the transition with the 0 symbol, so we must have a 0 in the string in order to accept.

In this course, we will allow multiple start states in an NFA. However, we can easily convert an NFA with many start states into an equivalent NFA with just a single start state. To do so, we have an artificial initial start state, and add ε-moves to the real start states (Fig. 2.1).

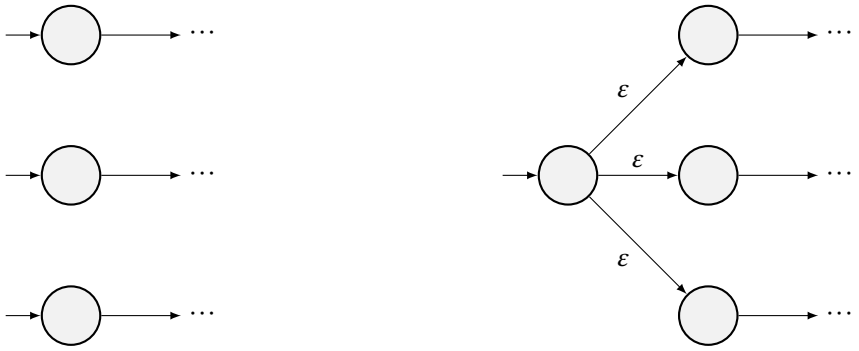
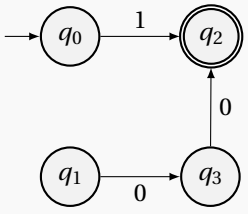


Figure 2.1: Conversion between an NFA with multiple initial states to an NFA with one initial state

Example 2.13



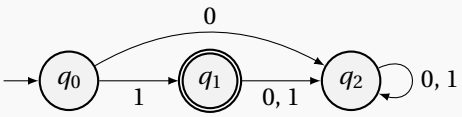
Here, notice that we can never accept any strings with length more than 2; we simply do not have any more paths to take.

In particular, we have $L(M) = \{1, 00\}$.

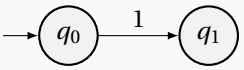
For the same language, NFAs can be simpler and/or smaller than DFAs.

Example 2.14

Suppose we want a DFA for the language $\{1\}$:



Suppose we want an NFA for the language $\{1\}$:



It should be clear that the NFA is a lot simpler and smaller than the DFA.

Definition 2.15: Non-deterministic Finite Automata

An NFA is a 5-tuple $N = (Q, \Sigma, \delta, Q_0, F)$ where

- Q is the set of states (finite)
- Σ is the alphabet
- $\delta : Q \times \Sigma_\varepsilon \to 2^Q$ is the transition function

  Here, $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$, and 2^Q is the set of all possible subsets of Q; we give the NFA a set of possible next states to choose from.
- $Q_0 \subseteq Q$ is the set of start states
- $F \subseteq Q$ is the set of accepting states

Definition 2.16: Accepting in an NFA

Let $w \in \Sigma^*$. We say that N accepts w if there exists a sequence of states $r_0, r_1, \ldots, r_n$ and w can be written as $w_1, \ldots, w_n$ with each $w_i \in \Sigma \cup \{\varepsilon\}$ such that

- $r_0 \in Q_0$
- $r_i \in \delta(r_{i-1}, w_i)$ for $1 \le i < n$
- $r_n \in F$

Notice here that for an NFA, there are many possible computation paths, each possibly accepting or rejecting, compared to the unique computation path in a DFA.

Are DFAs and NFAs the same? It turns out that surprisingly, yes! DFAs are equivalent to NFAs.

In this sense, we can see that here, verifying (i.e. with an NFA) is not easier than computing (i.e. with a DFA). However, later when we talk about P vs. NP, we'll see that it's been conjectured that verifying is indeed easier than computing.

9/1/2022

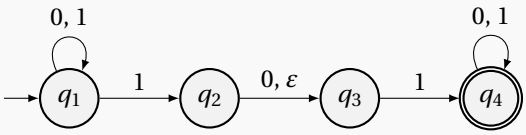
Lecture 3

Non-deterministic Finite Automata

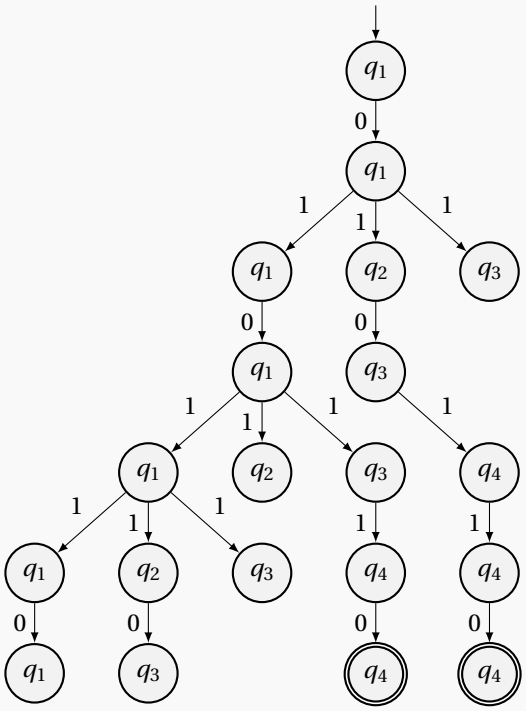
Recall that we tried to show that regular languages are closed under reversals, but we failed using DFAs, while also motivating the idea of NFAs.

Example 3.1

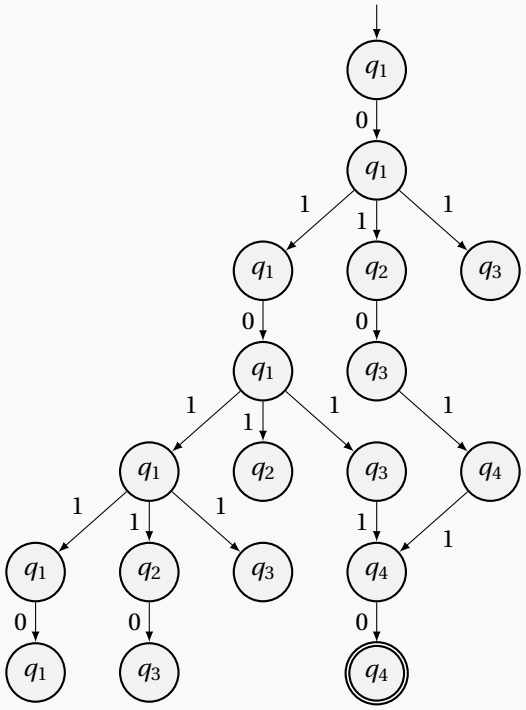
Consider this NFA:



The computation of the input "010110" on this NFA is as follows:



Here, when we look at this computation tree, we can see that there are several repeated computations; if we reach the same state at the same time on different branches, we don't need to look at the same state multiple times. In this sense, we don't really care about the entire tree of paths; we only care about the DAG of paths, merging branches when the subtrees are identical (i.e. we reach the same state at the same time on different branches).



Here, to keep track of all of the paths and states, we only need to keep track of $|\Sigma|$ states at any given symbol; it doesn't actually exponentially grow too much if we're trying to determine whether any path reaches an accepting state.

Theorem 3.2: NFAs are equivalent to DFAs

We claim that NFAs are equivalent to DFAs; i.e. we can simulate a NFA with a DFA, still with finitely many states.

Proof. Suppose we have an NFA $N = (Q, \Sigma, \delta, Q_0, F)$. We want to construct a DFA $M = (Q', \Sigma, \delta', q_0', F')$ that simulates the NFA.

To learn if an NFA accepts, we could do the computation "in parallel", maintaining the set of *all possible states* that can be reached at any given time.

That is, at each level of the DAG (starting from the initial states), we want to remember the set of possible states the NFA could be in.

For now, let us assume there are no ε-moves; we'll add them in later. Here, we have

- $Q' = 2^Q = \{R \mid R \subseteq Q\}$
- $\delta' : Q' \times \Sigma \to Q'$ is defined as
$$\delta(R, \sigma) = \bigcup_{r \in R} \delta(r, \sigma).$$
That is, we take the union of all possible next states starting from any of the current states.
- $q_0' = Q_0$
- $F' = \{R \mid R \subseteq Q, (\exists q \in F)(q \in R)\}$; that is, the set of accepting states are the set of subsets that contain any accepting state in the NFA.

To handle ε-moves, we define the ε-closure of a set $R \subseteq Q$ to be the set of all states reachable from R by taking 0 or more ε-moves, denoted as $\varepsilon(R)$.

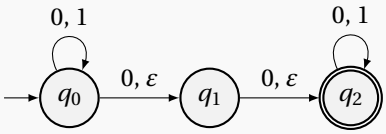
We can then modify the DFA to include ε-moves as follows:

- $Q' = 2^Q$
- $\delta' : Q' \times \Sigma \to Q'$ is defined as
$$\delta(R, \sigma) = \varepsilon\left(\bigcup_{r \in R} \delta(r, \sigma)\right).$$
- $q_0' = \varepsilon(Q_0)$
- $F' = \{R \in Q' \mid R \text{ contains an accepting state of } N\}$

□

Example 3.3

Here's an example of an ε-closure of a set. Suppose we have an NFA

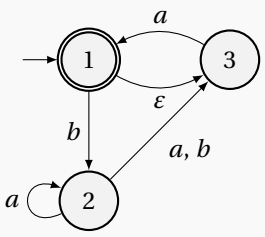


We have

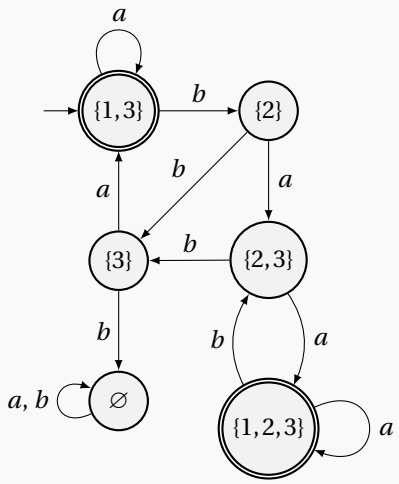
$$\begin{aligned}
\varepsilon(\{q_0\}) &= \{q_0, q_1, q_2\} \\
\varepsilon(\{q_1\}) &= \{q_1, q_2\} \\
\varepsilon(\{q_2\}) &= \{q_2\} \\
\varepsilon(\{q_1, q_2\}) &= \{q_1, q_2\} \\
\varepsilon(\varnothing) &= \varnothing \\
\varepsilon(Q) &= Q
\end{aligned}$$

Example 3.4

Suppose we have the following NFA:



The converted DFA is



Notice that every state here has an outgoing edge for every possible symbol. Further, notice that we don't have the states $\{1\}$ and $\{1,2\}$, as these are not reachable by the NFA. To be fully rigorous and complete, we could draw these states, but they would be disconnected from the DFA; these states are also not necessary to include, as they do not help us accept anything.

Intuitively, we can't ever be at $\{1\}$, since $\varepsilon(\{1\}) = \{1,3\}$ and we'll never only be at state 1. Similar reasoning applies to why we can't ever be at $\{1,2\}$

As such, it can help when constructing such a DFA from an NFA to explore all possible reachable states, to then save memory.

We've just shown that NFAs can be simulated by DFAs; the two are equivalent. However, does this mean that non-determinism can be eliminated for free? Not quite; we need more memory to store all of the extra states we need for the equivalent DFA.

At this point, with this equivalence, we've shown that regular languages are closed under reversals:

Theorem 3.5

The reverse of any regular language is also regular.

Proof. If a language L can be recognized by a DFA M, then its reverse L^R can be recognized by an NFA N by flipping the arrows and flipping the accepting and starting states.

Then, since NFAs are equivalent to DFAs, there exists a DFA M' such that $L(M') = L(N) = L^R$. □

Notice that with the equivalence between DFAs and NFAs gives us more flexibility to prove that a language is regular; we only need to construct an NFA, not a DFA.

Revisiting the union theorem, we can construct an NFA to recognize the union of two languages L_1 and L_2.

A simple way to construct this NFA is to just combine the DFAs without any modifications; we have multiple start states, and we're still simulating both automata at the same time.

Theorem 3.6

Regular languages are closed under concatenation.

Proof. Suppose we have DFAs M_1 and M_2 for languages A and B. We can construct an NFA for $A \cdot B = \{uv \mid$

$u \in A, v \in B\}$ by simply connecting all accepting states of M_1 to the starting state of B with ε-moves, and remove accepting states from M_1.

This way, after reading an accepting string for A, we'd end up at an accepting state for M_1, which we can then go to M_2 to check whether the second half is accepting. □

Theorem 3.7

Regular languages are closed under the star operator.

Proof. Let L be a regular language that can be recognized by a DFA M. We can construct an NFA N for L^* by adding ε-moves from all accepting states to the initial state. However, this does not accept the empty string; to take care of this, we add an additional state, becoming the new start state, with an ε-move to the old start state.

Formally, here is the construction for the NFA $N = (Q', \Sigma, \delta', q_0, F')$ from the DFA $M = (Q, \Sigma, \delta, q_1, F)$:

- $Q' = Q \cup q_0$
- $F' = F \cup \{q_0\}$
- $\delta' : Q' \times \Sigma \to Q'$ is defined as

$$\delta'(q,a) = \begin{cases} \{\delta(q,a)\} & \text{if } q \in Q,\ a \neq \varepsilon \\ \{q_1\} & \text{if } q \in F,\ a = \varepsilon \\ \{q_1\} & \text{if } q = q_0,\ a = \varepsilon \\ \varnothing & \text{if } q = q_0,\ a \neq \varepsilon \\ \varnothing & \text{otherwise} \end{cases}.$$

We want to show that $L(N) = L^*$, so we need to show two things: (1) $L(N) \supseteq L^*$, and (2) $L(N) \subseteq L^*$.

Suppose $w = s_1 s_2 \cdots s_k \in L^*$ where each $s_i \in L$. We show that N accepts w by induction on k.

In the base case, for $k = 0$, we have $w = \varepsilon$, and we know that we accept the empty string, as the initial state is accepting.

For $k = 1$, we know that $w \in L$, so w corresponds to an accepting path in the DFA, so there exists some path from q_1 to an accepting state. Adding the first ε-move from q_0 to q_1 gives us the accepting path in the NFA.

In the inductive step, suppose N accepts all strings $v = s_1 \cdots s_k$ for some k with all $s_i \in L$. We will show that N accepts $u = s_1 \cdots s_k s_{k+1}$ for $s_i \in L$.

By the inductive hypothesis, we know that N accepts $s_1 \cdots s_k$, so we have some computation path from q_0 to an accepting state. We can then take the ε-move to reach q_1; at this point, since $s_{k+1} \in L$, there must exist some computation path for s_{k+1} from q_1 to an accepting state. Taking this computation path will lead us again to some accepting state in the NFA, and the NFA accepts u, as desired.

Proving the other direction, we want to show that if w is accepted by N, then $w \in L^*$. To do this, we can perform induction on the number of ε-transitions in the path.

Suppose the claim holds for some k, i.e. every u with an accepting path in N that makes k ε-transitions is in L^*. We will show the claim holds for $k+1$.

Suppose w is a string accepted by a path P in N with $k+1$ ε-transitions. Consider the last ε-transition in P. Suppose we break P into two parts: everything before the last ε-transition, and everything after the ε-transition. Everything before the last ε-transition is accepting by the IH, and the suffix is a string with a computation path from q_1 to an accepting state with no ε-transitions. This suffix must then be a string in the original language, as we have an accepting path in the original DFA.

This means that $w \in L^*$, as desired. □

9/6/2022

Lecture 4

Regular Expressions

We can think of regular expressions as a different way to describe regular languages. It's computation as a simple, logical description.

Definition 4.1: Regular Languages (inductive definition)

Let Σ be an alphabet. We define the set of regular expressions over Σ inductively:

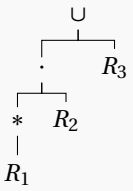
- For any $\sigma \in \Sigma$, σ is a regexp.
- ε is a regexp
- $\varnothing$ is a regexp
- If R_1 and R_2 are regexps, then $(R_1 \cdot R_2)$, $(R_1 \cup R_2)$, $(R_1)^*$ are regexps.

We also have a precedence order if we leave out parentheses: $*$, then $\cdot$, then $\cup$.

Example 4.2

For example, $R_1^* R_2 \cup R_3$ is equivalent to $(((R_1^*) \cdot R_2) \cup R_3)$.

We can also think of this as a tree:



Example 4.3

Regular expressions also represent languages; for example,

- For $\sigma \in \Sigma$, the regexp σ represents $\{\sigma\}$
- The regexp ε represents $\{\varepsilon\}$
- The regexp $\varnothing$ represents $\{\varnothing\}$
- If R_1 and R_2 are regexps representing L_1 and L_2, then $(R_1 \cdot R_2)$ represents $L_1 \cdot L_2$
- Similarly, $(R_1 \cup R_2)$ represents $L_1 \cup L_2$ (this is sometimes denoted $(R_1 + R_2)$)
- Similarly, $(R_1)^*$ represents L_1^*

For every regexp R, we denote by $L(R)$ the language that R defines. In particular, we have

- $L(0) = \{0\}$
- $L(\varepsilon) = \{\varepsilon\}$

- $L(\varnothing) = \varnothing$
- $L(R_1 \cdot R_2) = L(R_1) \cdot L(R_2)$
- $L(R_1 \cup R_2) = L(R_1) \cup L(R_2)$
- $L(010) = \{010\}$

  Note that we will usually omit the $\cdot$.

Definition 4.4: Accepted by a regexp

A string $w \in \Sigma^*$ is *accepted* or *matched* by R if $w \in L(R)$.

Example 4.5

The strings 0, 010, 0101010, matches $(01)^*0$.

The string 01110110010 matches $(0 \cup 1)^*0$ (i.e. all strings ending with a 0).

Example 4.6

Suppose $\Sigma = \{0, 1\}$.

- Write a regexp for the language $\{w \mid w \text{ has exactly a single } 1\}$

$$R = 0^*10^*.$$

- Write a regexp for the language $\{w \mid w \text{ ends in } 001\}$

$$R = (0 \cup 1)^*001.$$

- Write a regexp for the language $\{w \mid w \text{ contains 001 as a substring}\}$

$$R = (0 \cup 1)^*001(0 \cup 1)^*.$$

- Write a regexp for the language $\{w \mid w \text{ starts and ends with the same symbol, } w \neq \varepsilon\}$

$$R = (0\Sigma^*0) \cup (1\Sigma^*1) \cup 0 \cup 1.$$

- Write a regexp for the language $\{w \mid w \text{ is of even length}\}$

$$R = (\Sigma\Sigma)^*.$$

- Write a regexp for the language $\{w \mid \text{every odd position of } w \text{ is } 1\}$

$$R = (1\Sigma)^* \cup (1\Sigma)^*1 = (1\Sigma)^*(\varepsilon \cup 1).$$

Example 4.7

Here are some edge cases regarding $\varnothing$ and ε:

- $L((101)^*\varnothing) = L((101)^*) \cdot L(\varnothing) = \varnothing$
- $L(\varnothing^*) = \{\varepsilon\}$
- $L(R \cdot \varepsilon) = L(R)$
- $L(R \cdot \varnothing) = \varnothing$
- $L(R \cup \varnothing) = L(R)$

Theorem 4.8

A language L can be represented by some regular expression if and only if L is regular.

In particular, this means that DFAs, NFAs, and regular expressions are all equivalent.

Lemma 4.9

If a language is described by a regular expression R, then it is regular.

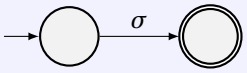
Proof. We prove by induction on the length of the regexp.

Briefly, here is how we define the length of a regexp;

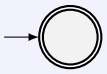
- $\text{len}(0) = 1$
- $\text{len}(\varepsilon) = 1$
- $\text{len}(\varnothing) = 1$
- $\text{len}(R_1 \cdot R_2) = \text{len}(R_1) + \text{len}(R_2) + 1 = \text{len}(R_1 \cup R_2)$
- $\text{len}(R^*) = \text{len}(R) + 1$

In the base case, we have a regexp of length 1. There are three cases:

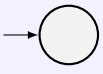
- For $R = \sigma$, $L(R) = \{\sigma\}$ is a regular language, as we have the NFA



- For $R = \varepsilon$, $L(R) = \{\varepsilon\}$ is a regular language, as we have the NFA



- For $R = \varnothing$, $L(R) = \varnothing$ is a regular language, as we have the NFA



Suppose the statement holds for all regexps of length $< k$, for $k \geq 2$.

There are three possibilities:

- $R = (R_1 \cdot R_2)$

  We know that $L(R_1)$ and $L(R_2)$ are regular by the IH, and by closure properties $L(R) = L(R_1) \cdot L(R_2)$ is also regular.

- $R = (R_1 \cup R_2)$

  Similarly, we know that $L(R_1)$ and $L(R_2)$ are regular by the IH, and by closure properties $L(R) = L(R_1) \cup L(R_2)$ is also regular.

- $R = (R_1)^*$

  Similarly, we know that $L(R_1)$ is regular by the IH, and by closure properties $L(R) = L(R_1)^*$ is also regular.

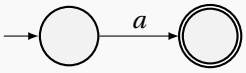
□

Notice that this process will give us a direct construction of an NFA for the language represented by a regexp.

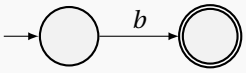
Example 4.10

Give an NFA that accepts the language represented by $(ab \cup a)^*$.

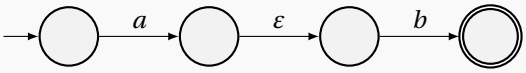
The NFA recognizing only "a" is



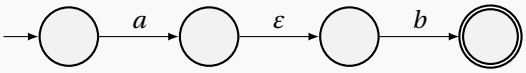
The NFA recognizing only "b" is

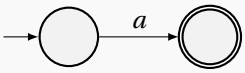


With this, the NFA recognizing "ab" is

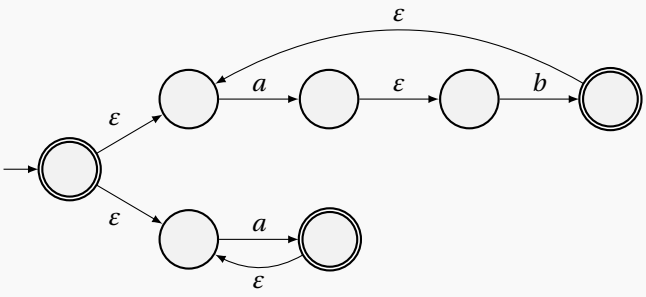


Further, the NFA recognizing "$ab \cup a$" is





Lastly, the NFA recognizing "$(ab \cup a)^*$" is



Lemma 4.11

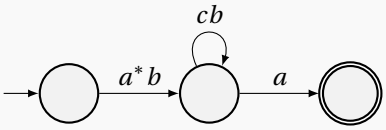
If L is a regular language, then L can be represented by a regular expression.

Proof. The idea is to transform an NFA for L into a regular expression by removing one state at a time, and relabeling edges with regular expressions.

The model we end up with is a "generalized NFA". □

Example 4.12

Here is an example for a generalized NFA (GNFA):



- Is "$aaabcbcbcba$" accepted?

  Yes; "$aaab$" corresponds to a^*b, "$cbcbcb$" corresponds to cb multiple times in the self loop, and "a" is the last transition.

- Is "bba" accepted?

  No; b corresponds to a^*b, but now we have a b, that doesn't match anything.

- Is "$bcba$" accepted?

  Yes; b corresponds to a^*b, cb goes in the self loop, and a brings us to the accepting state.

This GNFA accepts the language $L(a^*b(cb)^*a)$.

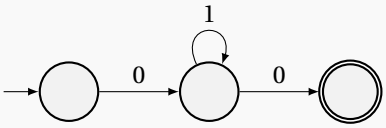
The proof sketch is as follows:

Suppose we start with an NFA N for L. To make things cleaner, let us add a unique start and accept states, connecting the original starting and accepting states by ε-moves.

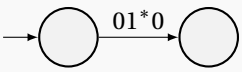
While the GNFA has more than 2 states, we will pick an internal state, then "rip it out" and relabel the edges to maintain the acceptance of paths.

Example 4.13

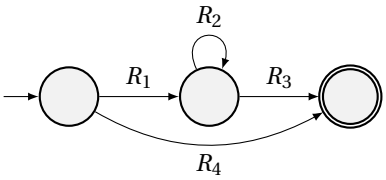
Suppose we have the following NFA:



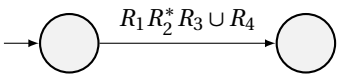
Taking out the middle node, we have



More generally, if we have the following GNFA:



Removing the middle node converts this GNFA into

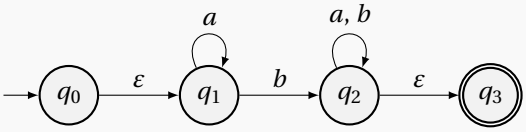


Notice that in a GNFA, we only need one edge between any pair of states; if there are more, we can simply take the union of the regular expressions and merge it into one edge.

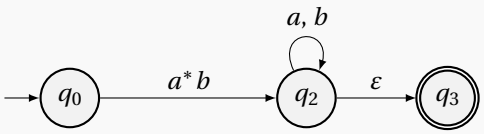
The formal proof takes some time, but this proof sketch should encompass the main ideas.

Example 4.14

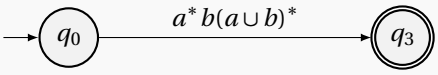
Suppose we have the following NFA:



Eliminating q_1, we have



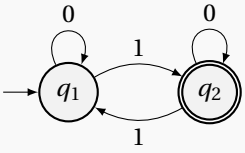
Eliminating q_2, we have



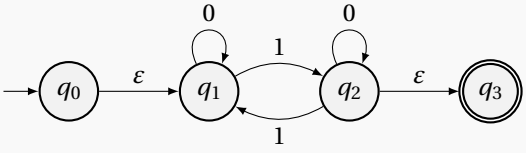
Example 4.15

Give a regular expression for the language $L = \{w \mid w \text{ has an odd number of 1's}\}$.

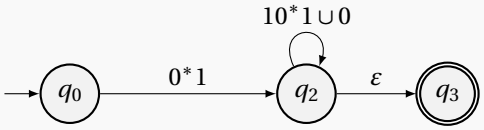
The DFA for L is



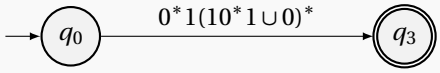
Converting this into a GNFA, we have



Eliminating q_1, we have



Eliminating q_2, we have



Looking back, we started with DFAs, and defined regular languages. To show closure properties, we needed to introduce the idea of NFAs to show the closure of reversals. We then showed that NFAs are actually equivalent to DFAs, which allowed us to finish showing the closure properties.

Now, we've introduced regular expressions by looking at regular languages in a different perspective, building it up from basic building blocks and $\cup$, $\cdot$, $*$. It turns out that we can get all possible regular languages in this way.

The interesting thing about what we've done so far is that in order to prove certain properties, we needed to introduce several different models, which all could be converted into realistic machines (DFAs).

All of this allows us to conclude that regular expressions, DFAs, NFAs, and regular expressions are all equivalent; all of these models describe the same kind of objects.

9/8/2022

Lecture 5

Non-regular Languages

We've talked about regular languages so far, but there are lots of languages that are non-regular (in fact, most of them).

For example, $L = \{w \mid w \text{ has the same number of 0's and 1's}\}$ is non-regular; intuitively, this is because we can't count with a DFA; with only finitely many states, we can only count to a finite amount, meaning we'll eventually run out of space.

Here are some other examples of non-regular languages that we'll prove today:

- $L = \{a^n b^n \mid n \geq 0\}$
- $L = \{s \cdot s \mid s \in \{a,b\}^n\}$
- $L = \{a^r \mid r \text{ is prime}\}$

Example 5.1

We will show that $L = \{a^n b^n \mid n \geq 0\}$ is not regular.

Proof. Suppose for contradiction that L is regular, i.e. it is accepted by some DFA $M = (Q, \Sigma, \delta, q_0, F)$. Further, the most important thing about this DFA is that it has a finite number of states; here, let $p = |Q|$.

Consider the input $w = a^p b^p \in L$, and consider the computation path of M on w. Notice that this string is longer than then number of states in the DFA.

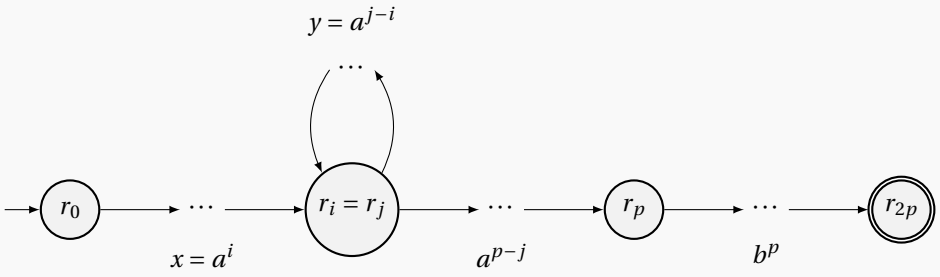
Note that if we have 10 states, we can't even count to 10 (since we need a $0, 1, \ldots, 10$); in particular, in this case, we don't know how many a's we saw, since its length is $p = |Q|$.

The computation path is as follows:

$$q_0 = r_0 \xrightarrow{a} r_1 \xrightarrow{a} r_2 \xrightarrow{a} \cdots \xrightarrow{a} r_p \xrightarrow{b} \cdots \xrightarrow{b} r_{2p} \in F.$$

What can we say about this sequence of states? Since this path up to r_p is of length $p+1$, there must be some repeated state (by the pigeonhole principle). Since there is a collision, there must be some cycle in the path before r_p. That is, there must be some $r_i = r_j$ for $0 \le i < j \le p$.

Therefore, let $x = a^i$, i.e. the string before the collision, let $y = a^{j-i}$, i.e. the string in the cycle, and $z = a^{p-j}b^p$, i.e. the string after the cycle. This will end up at r_{2p}, an accepting state.



Another string that will be accepted would be xz, or xyz, or $xyyz$, etc. However, a lot of these strings are not supposed to be accepted.

For example, we have

$$xyyz = a^i a^{2(j-i)} a^{p-j} b^p = a^{p+j-i} b^p \notin L.$$

□

What did we exploit here? We exploited the fact that in any DFA of finite size, any computation path on a long enough string has the following form:

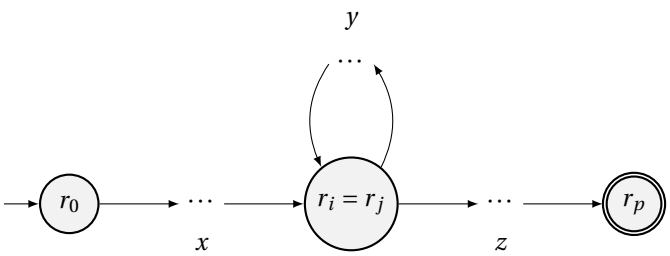


Figure 5.1: Computation paths on long strings in a DFA must have a cycle.

In particular, this means that $xz \in L(M)$, $xyz \in L(M)$, $xyyz \in L(M)$, etc. In general, this means that for all i, $xy^i z \in L(M)$. If the language we chose does not have this property, then we have our contradiction.

Lemma 5.2: The Pumping Lemma

Let L be a regular language. Then, there is a constant $p = p_L$ (defined to be the *pumping constant* for L) such that for every $w \in L$ with $|w| \ge p$ can be "pumped". That is, w can be written as $w = xyz$ with:

- $|xy| \le p$
- $|y| \ge 1$
- $(\forall k \in \mathbb{N})(xy^k z \in L)$.

Proof. Since L is regular, let M be a DFA that recognizes L. Let $p = |Q|$ be the number of states in M.

For any word $w \in L$, with $|w| = n \geq p$, consider the computation path of M on w:

$$q_0 = r_0 \xrightarrow{w_1} r_1 \xrightarrow{w_2} r_2 \to \cdots \xrightarrow{w_n} r_n.$$

Formally, we define this computation path with $r_0 = q_0$, and $r_{i+1} = \delta(r_i, w_{i+1})$.

Since this path has $n \geq p$ states, by the pigeonhole principle, there exists some $0 \leq i < j \leq p$ such that $r_i = r_j$. That is, there is some collision in these $p+1$ states.

With this collision, we can define x, y and z as in Fig. 5.1. Here, we know the following:

- $|xy| = j$, since we consumed the first j symbols of the input and met with the collision point again. In particular, we have $|xy| = j \leq p$.
- $|y| = j - i \geq 1$, since we will always have $i < j$.
- Since this is a DFA, we also have $(\forall k \in \mathbb{N})(xy^k z \in L)$.

  To see why, after x, we end up at r_i, and y^k will bring us back to $r_i = r_j$. The last substring z will leave us at r_n. Since $r_n \in F$, this state is accepting, and this string is accepted.

□

With this lemma, we no longer need to use the pigeonhole principle and re-prove the results. We can instead directly use the lemma to produce the proof.

Example 5.3

Show that $L = \{a^n b^n \mid n \geq 0\}$ is not regular.

Proof. Suppose for contradiction that L is regular. By the pumping lemma, there is a constant p such that for every $w \in L$ with $|w| \geq p$, w can be pumped.

Suppose we pick $w \in L$ such that $w = a^p b^p \in L$. We have $|w| \geq p$, and w can be written as xyz with $|xy| \leq p$, $|y| \geq 1$, and $(\forall k)(xy^k z \in L)$.

In particular, we have $x = a^i$ and $y = a^{j-i}$ for some $i < j \leq p$, since x and y cannot contain any b's (since $|xy| \leq p$. Further, we have $z = a^{p-j} b^p$.

When we pump w, we increase the number of a's, but we do not change the number of b's, giving us a contradiction, since for example $xy^2 z \notin L$. □

Example 5.4

Consider $L_2 = \{w \mid w \text{ has an equal number of } a\text{'s and } b\text{'s}\}$.

Proof. We can use the same exact proof with $a^p b^p$. However, here's an alternative proof utilizing the fact that $L_1 = \{a^n b^n \mid n \geq 0\}$ is not regular.

By contradiction, suppose L_2 is regular. In addition, $L_3 = L(a^* b^*)$ is regular, since $a^* b^*$ is a regexp. We further know that $L_2 \cap L_3$ is regular, since it is an intersection of regular languages.

However, $L_2 \cap L_3 = \{a^n b^n \mid n \geq 0\} = L_1$, which contradicts with the fact that L_1 is non-regular. □

Example 5.5

Consider $L_4 = \{s \cdot s \mid s \in \{a,b\}^*\}$

For example, $\varepsilon \in L_4$, $aa \in L_4$, $abab \in L_4$, $aaabbaaabb \in L_4$.

Proof. Suppose by contradiction that L_4 is regular. Let p be the pumping constant of L_4.

Let $w = a^p b a^p b \in L$; we have $|w| = 2p+2 \geq p$.

We can break up this string into three parts; since $|xy| \leq p$, x and y must not contain the b; both must be substrings of a^p.

In particular, we have $x = a^{|x|}$, $y = a^{|y|}$, and $z = a^{p-|x|-|y|} b a^p b$.

Pumping down, we have $xz = a^{p-|y|} b a^p b \notin L_4$. This is because there is no way to write this string as $s \cdot s$.

Formally, suppose for contradiction that $xz = s \cdot s$. It must be the case that s ends with a b, since xz ends with a b. We only have two b's, so s must look like $a^i b$. However, in this case with xz, the number of a's are not equal, so there cannot be such an s, and $xz \notin L$. □

The most important part of these kinds of proofs is in picking the string w to pump. There are situations where we can pick a string that gives us something that actually *is* in the language.

For example, with the previous example, suppose we picked $w = a^p \cdot a^p$. The pumping lemma says that we can write $w = xyz$. Here, we can pick $x = \varepsilon$, $y = a^2$, and $z = a^{2p-2}$, giving us for all $k \in \mathbb{N}$, $xy^k z = a^{2k+2p-2} = a^{k+p-1} a^{k+p-1} \in L$.

This doesn't give us our contradiction, and we've picked the wrong string—it does not mean that L is regular.

Some important things to note:

- You have *no control* on p; it is defined by the language, and we don't know what it is.
- You *have control* over the string $w \in L$, and in particular w may depend on p.
- You have *no control* on the partition of w to xyz, other than what is promised by the pumping lemma. In particular, the pumping lemma just says that *there exists* a partition that satisfies the constraints.

Example 5.6

Consider $L_5 = \{a^{n^2} \mid n \geq 0\}$.

Note that the following technique works if we have arbitrary long gaps in the lengths of strings in L, and we'll use a similar proof for primes.

Proof. Suppose for contradiction that L_5 is regular, and from the pumping lemma, let p be the pumping constant of L_5.

Consider $w = a^{p^2}$, with $|w| \geq p$.

The pumping lemma says we can write $w = xyz$ with $|xy| \leq p$ and $1 \leq |y| \leq p$. In particular, we have $x = a^{|x|}$, $y = a^{|y|}$, and $z = a^{p^2-|x|-|y|}$.

Looking at xy^2z, we have $a^{|x|} a^{2|y|} a^{p^2-|x|-|y|} = a^{p^2+|y|} \in L_5$.

However, $p^2 < p^2 + |y| \leq p^2 + p < (p+1)^2$, so $p^2 + |y|$ is never a square, giving us a contradiction, as $xy^2z \notin L_5$. □

Example 5.7

Consider $L_6 = \{a^r \mid r \text{ is prime}\}$.

Proof. Suppose for contradiction that L_6 is regular, and from the pumping lemma, let p be the pumping constant of L_6.

Notice that $(p+1)! + k$ for any $2 \le k \le p+1$ are all composite.

Consider $w = a^q$ for q as the smallest prime larger than $(p+1)! + (p+1)$, giving us $|w| = (p+1)! + p + 1 \ge p$.

The pumping lemma says that we can write $w = xyz$ with $|xy| \le p$ and $1 \le |y| \le p$. In particular, we have $x = a^{|x|}$, $y = a^{|y|}$, and $z = a^{q-|x|-|y|}$.

Looking at xz, we have $a^{|x|}a^{q-|x|-|y|} = a^{q-|y|}$.

Notice that $q - k$ for all $1 \le k \le p$ are all composite, since we chose q to be the first prime after $(p+1)! + p + 1$ (i.e. the next prime below q is below $(p+1)!$, which is more than p away). Since $1 \le |y| \le p$, no matter what y is, $q - |y|$ is not prime, and as such $a^{q-|y|} \notin L_6$. This is a contradiction, since L_6 would incorrectly accept this string. □

Proof. Here is another proof, relying on the fact that we can pump as many times as we like.

Here, we choose $w = a^r \in L$ for some prime $r \ge p$. By the pumping lemma, the following must all be accepted:

$$\begin{aligned} w^{(0)} &= xz = a^{r-|y|} \in L \\ w^{(1)} &= a^r \in L \\ w^{(2)} &= a^{r+|y|} \in L \\ &\vdots \\ w^{(r)} &= a^{r+r|y|} \in L \end{aligned}$$

However, $r + r|y|$ cannot be prime, and thus $w^{(r)}$ shouldn't be accepted. □

Example 5.8

Consider $L_7 = \{sts \mid s, t \in \{0,1\}^+\}$. In particular, L_7 is the set of all strings that start and end with the same substring, with some non-empty substring in the middle (i.e. $|s|, |t| \ge 1$).

Proof? Suppose we choose $w = 0^p 1 10^p 1$.

The pumping lemma says that we can write $w = xyz$ such that $|xy| \le p$ and $|y| \ge 1$.

However, if we choose $x = \varepsilon$, $y = 0^2$, and $z = 0^{p-2}110^p1$, we have

$$xz = 0^{p-2}110^p1 = \underbrace{0^{p-2}1}_{s}\underbrace{10^2}_{t}\underbrace{0^{p-2}1}_{s},$$

and actually $xz \in L_7$. □

This attempt shows that it can sometimes be hard to pick a w such that pumping it will give a string not in L; some care and thought needs to be put in to choose such a string.

9/13/2022

Lecture 6

Non-regular Languages (cont.)

Last time, we introduced the pumping lemma to prove that certain languages are not regular, i.e.

- $L_1 = \{a^n b^n \mid n \geq 0\}$
- $L_2 = \{s \cdot s \mid s \in \{0,1\}^*\}$
- $L_3 = \{a^{n^2} \mid n \geq 0\}$
- $L_4 = \{a^p \mid p \text{ prime}\}$

Today, we'll look at another technique to show that certain languages are non-regular. We'll later see that this technique is also useful for regular languages as it can be used to find the *smallest* DFA for L.

Before we talk about the technique, let us do a brief recap of equivalence relations.

Definition 6.1: Relation

For a set A, a *relation* R is a subset of $A \times A$. For $a, b \in A$, we denote by $a \sim_R b \iff (a,b) \in R$; we say "$a$ relates to b". Here, note that (a,b) is an ordered pair.

Definition 6.2: Equivalence Relation

A relation R is an *equivalence relation* if it satisfies the following properties:

- Reflexivity: $\forall a \in A$, $(a,a) \in R$
- Symmetry: $\forall a, b \in A$, if $(a,b) \in R$, then $(b,a) \in R$
- Transitivity: $\forall a, b, c \in A <$ if $(a,b) \in R$ and $(b,c) \in R$, then $(a,c) \in R$.

Example 6.3

Here are some examples of equivalence relations:

1. Let $A = \mathbb{N}$ and take $m \in \mathbb{N}$. The relation $R_m = \{(a,b) \mid a \equiv b \pmod{m}\}$ is an equivalence relation.
2. Let $G = (V,E)$ be an undirected graph, with $A = V$.

   The relation $R_G = \{(u,v) \mid u \text{ and } v \text{ are connected in } G\}$ is an equivalence relation.
3. Let $f : A \to X$. The relation $R_f = \{(a,b) \mid f(a) = f(b)\}$ is an equivalence relation.

Definition 6.4: Equivalence Class

Let R be an equivalence relation on a set A. For a set $a \in A$, we define the *equivalence class*

$$[a]_R = \{b \in A \mid (a,b) \in R\}.$$

Corollary 6.5

We claim the following properties hold:

- If $(x,y) \in R$, then $[x]_R = [y]_R$.

- If $(x, y) \notin R_<$ then $[x]_R$ is disjoint from $[y]_R$, i.e. $[x]_R \cap [y]_R = \varnothing$.

The first equivalence relation we will define regarding languages and strings is as follows.

Let L be a regular language, and let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA for L. We define the following relation over the set of strings Σ^*:

$$R_M = \{(x, y) \in \Sigma^* \times \Sigma^* \mid M \text{ ends up in the same state on inputs } x \text{ and } y\}.$$

We claim that R_M is an equivalence relation:

- R_M is reflexive: a string s will always end up at the same state as itself.
- R_M is symmetric by definition (order doesn't matter).
- R_M is transitive: if x and y reach the same state, and also y and z reach the same state, then all three strings reach the same state, and in particular x and z reach the same state.

How many equivalence classes are there for R_M? An upper bound is the number of states in M, and the exact number is the number of reachable states in M.

Notice that even though we may have infinitely many strings, once we have a DFA we now only have finitely many equivalence classes (i.e. states).

The next equivalence relation we'll define is with respect to a language. Before we do this, we must first define *distinctness* with respect to strings.

Definition 6.6: Distinguishable

Let L be a regular language over an alphabet Σ.

Two strings $x, y \in \Sigma^*$ are *distinguishable* with respect to a language L if there is some string $z \in \Sigma^*$ such that $xz \in L$ and $yz \notin L$, or $xz \notin L$ and $yz \in L$.

Two strings x and y are *indistinguishable* with respect to L if the converse holds, i.e. $\forall z \in \Sigma^*$, either both xz and yz are in L, or neither xz nor yz are in L.

Now, let us define the following relation:

$$\begin{aligned} R_L &= \{(x, y) \in \Sigma^* \times \Sigma^* \mid x \text{ and } y \text{ are indistinguishable with respect to } L\} \\ &= \{(x, y) \in \Sigma^* \times \Sigma^* \mid \forall z (xz \in L \iff yz \in L)\} \end{aligned}$$

We claim that R_L is an equivalence relation:

- R_L is reflexive: a string x will always be indistinguishable from itself.
- R_M is symmetric by definition (order doesn't matter).
- R_M is transitive: Suppose x and y are indistinguishable, and y and z are indistinguishable. We want to show that x and w are indistinguishable.

  In particular, we know that for all $z \in \Sigma^*$, $xz \in L \iff yz \in L$; we also know that for all $z \in \Sigma^*$, $yz \in L \iff wz \in L$.

  Together, we have that $\forall z \in \Sigma^*$, $xz \in L \iff yz \in L \iff wz \in L$, meaning $xz \in L \iff wz \in L$, and as such x and w are indistinguishable.

Example 6.7

Consider the language $L = \{0^n 1^n \mid n \geq 0\}$. (Recall that we showed this is a non-regular language last time.)

- Are the strings $x = 0$ and $y = 00$ distinguishable?

Yes; if we append $z = 1$, then $xz = 01 \in L$ and $yz = 001 \notin L$. This means that x and y are in two distinct equivalence classes, and are distinguishable.

- Are the strings $x = 0^i$ and $y = 0^j$ for $j > i$ distinguishable?

  Yes; if we append $z = 1^i$, then $xz = 0^i 1^i \in L$ and $yz = 0^j 1^i \notin L$. This means that x and y are in two distinct equivalence classes, and are distinguishable.

The second item suggests that there are infinitely many strings that are distinguishable from each other. Further, this means that we have an infinite number of equivalence classes with respect to R_L

Example 6.8

Consider the language $L_2 = \{0,1\}^*$.

All strings are indistinguishable with respect to L_2, since all strings are accepted by L_2.

Example 6.9

Consider the language $L_3 = L((010)^*)$.

- Are $x = 01$ and $y = 0$ distinguishable with respect to L_3?

  Yes; we can append $z = 0$ to get $xz = 010 \in L$ and $yz = 00 \notin L$.

  We can also append $z = 10$ to get $xz = 0110 \in L$ and $yz = 010 \in L$.

- Are $x = \varepsilon$ and $y = 010$ distinguishable with respect to L_3?

  No; for an arbitrary $z \in L$, we know that $xz \in L$ iff $z = (010)^i$ by definition of the language. This in turn means that $yz = (010)^{i+1} \in L$; both xz and yz are in L or are both not in L.

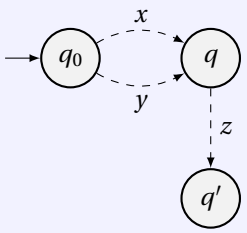
Lemma 6.10

If L is a regular language over Σ and $M = (Q, \Sigma, \delta, q_0, F)$ is a DFA for L, then for any pair of strings $x, y \in \Sigma^*$, $(x,y) \in R_M \implies (x,y) \in R_L$.

In other words, if two strings reach the same state in M, then they are indistinguishable with respect to L.

Proof. Let $(x,y) \in \Sigma^*$ with $(x,y) \in R_M$.

By the definition of R_M, this means that M reaches the same state on the computation path for x and y.



For any suffix $z \in \Sigma^*$ we append, M would reach the same state on xz and yz, since we start from the same state after reading x compared to reading y; at this point, we do not know whether we have read x or read y.

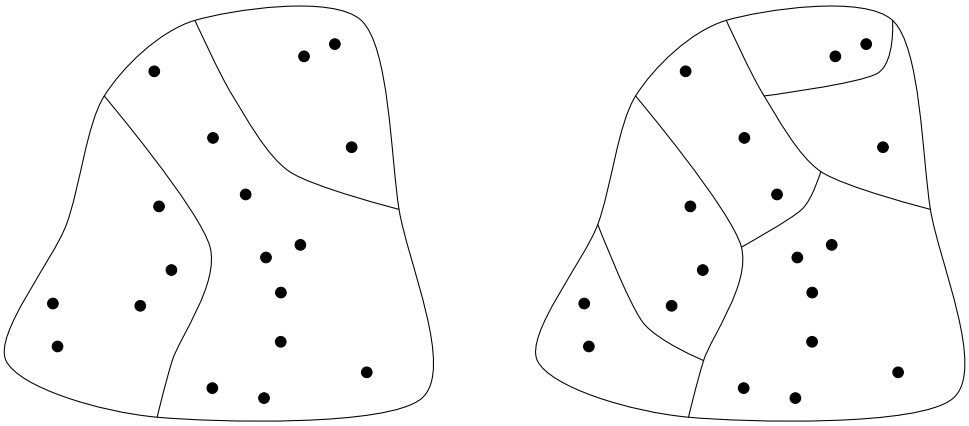
In particular, either both $xz, yz \in L$ or neither xz, yz are in L. □

We've just shown that R_M is a *refinement* of R_L.

Definition 6.11: Refinement

If R_1 and R_2 are two equivalence relations on a set A such that $(x, y) \in R_1 \implies (x, y) \in R_2$, then R_1 is said to be a refinement of R_2.

Pictorially, we have the following:



R_2 R_1

Here, each pair $(x, y) \in R$ corresponds to a dot. We can partition $A \times A$ into regions corresponding to equivalence classes. Saying that R_1 is a refinement of R_2 means that we're dividing up these equivalence classes more with R_1 compared to R_2.

One conclusion we can draw from this is as follows:

Corollary 6.12

If R_1 refines R_2, then the number of equivalence classes for R_1 is at least as much as the number of equivalence classes for R_2.

Corollary 6.13

If L is regular, then R_L has finitely many equivalence classes.

Proof. In particular, the number of equivalence classes of R_L is at most the number of equivalence classes of R_M, which in turn is at most the number of states in M, which is finite. □

With this corollary, we have a way to prove whether a language is non-regular. That is, the contrapositive says that if R_L has infinitely many equivalence classes, then L is not regular.

Example 6.14

Consider the language $L = \{0^n 1^n \mid n \geq 0\}$ from before.

We showed that for every $i < j$, $x = 0^i$ and $y = 0^j$ are distinguishable; appending $z = 0^i$ makes $xz \in L$ but $yz \notin L$.

This means that 0^i and 0^j will always fall in different equivalence classes for any $i \neq j$, and as such there are an infinite number of equivalence classes. This in turn means that L is not regular.

Example 6.15

Consider the language $L_2 = \{a^{n^2} \mid n \geq 0\}$.

To show that L_2 is is non-regular, we need to show that R_{L_2} has infinitely many equivalence classes.

In particular, we can show that $\forall i \neq j$, $a^{i^2} \not\sim a^{j^2}$, i.e. a^{i^2} and a^{j^2} are distinguishable; WLOG, suppose $i < j$.

For $x = a^{i^2}$ and $y = a^{j^2}$, if we append $z = a^{2i+1}$, then we have

$$xz = a^{i^2} a^{2i+1} = a^{i^2+2i+1} = a^{(i+1)^2}$$
$$yz = a^{j^2} a^{2i+1} = a^{j^2+2i+1}$$

Here, we make use of the fact that $2i+1 < 2j+1$ to see that $j^2 < j^2 + 2i + 1 < j^2 + 2j + 1 = (j+1)^2$.

This means that $j^2 + 2i + 1$ is not a perfect square, and thus $a^{j^2+2i+1} \notin L_2$, making x and y distinguishable.

To show that L is not regular, it suffices to come up with an infinite set of strings $S \subseteq \Sigma^*$ such that any two strings in S are distinguishable with respect to L.

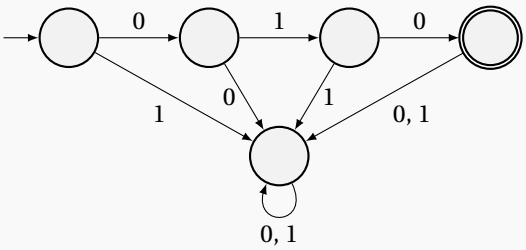
Corollary 6.16

If L is regular, then for any DFA M for L, the number of states in M is at least the number of equivalence classes in R_L.

Example 6.17

Consider $L = \{010\}$. Show that the smallest DFA for L has size 5.

For an upper bound, we just need to show that there exists some DFA with size 5 that recognizes the language:



For the lower bound, we claim that ε, 0, 01, 010, and 0100 are all pairwise distinct with respect to L.

We need to check all pairs, which is tedious, but not hard. For example 0 and 01 are distinguishable using $z = 0$, and 010 and 0100 are distinguishable using $z = \varepsilon$.

As a remark, recall that we can construct an NFA for L with only 4 states.

So far, we showed that if L is regular, then R_L has finitely many equivalence classes. It turns out that the converse is also true; this means that we can actually define regular languages solely on the number of equivalence classes in R_L. This is another equivalent characterization of regular languages, on top of DFAs, NFAs, and regular expressions.

Theorem 6.18: Myhill–Nerode Theorem

A language $L \subseteq \Sigma^*$ is regular if and only if R_L has finitely many equivalence classes.

Proof. We already proved the forward direction (if L is regular, then R_L has finitely many equivalence classes).

To show the reverse, suppose L is a language with finitely many equivalence classes in R_L. Then, it suffices to show that there is a DFA M that recognizes L using a number of states equal to the number of equivalence classes of R_L.

We define the DFA as follows:

- $Q = \{[x]_{R_L} \mid x \in \{0,1\}^N\}$

  That is, Q is the set of representative elements for each equivalence class; we have one state for each equivalence class in R_L.

- $q_0 = [\varepsilon]_{R_L}$
- $\delta : Q \times \Sigma \to Q$ is defined as

  $$\delta([x]_{R_L}, a) = [xa]_{R_L}.$$

  It's not clear that $[xa]_{R_L}$ is always well-defined. If we choose a different representative, does this still work? To show that this operation is well defined, we need to show that if we pick a different representative y, we have the same result.

  In particular, suppose $x \sim y$; for all such y, we have

  $$\delta([x]_{R_L}, a) = [xa]_{R_L} = [ya]_{R_L} = \delta([y]_{R_L}, a).$$

  The middle equality makes use of the fact that xa and ya are related; in particular, we have for all z,

  $$(xa)z \in L \iff x(az) \in L \iff y(az) \in L \iff (ya)z \in L.$$

- $F = \{[x]_{R_L} \mid x \in L\}$

  Since we have $x \sim y \implies x \in L \iff y \in L$ (i.e. taking $z = \varepsilon$), this set is also well defined; it doesn't depend on the representative element.

We claim that on input x, the computation path reaches state $[x]_{R_L}$. We can show this claim by induction on the length of x.

In the base case, we have $x = \varepsilon$; M reaches $q_0 = [\varepsilon]_{R_L}$ as desired, as it's the initial state.

Suppose $x = x_1 \cdots x_n$. By induction, after $x_1 \cdots x_{n-1}$, we reach $[x_1 \cdots x_{n-1}]_{R_L}$. In the nth step, we move from $[x_1 \cdots x_{n-1}]_{R_L}$ to $[x_1 \cdots x_n]_{R_L}$ by construction, as desired.

This means that $x in L(M) \iff [x]_{R_L} \in F$ by the above, and $[x]_{R_L} \in F \iff x \in L$ by definition of F.

As such, the set of strings recognized by the DFA is exactly the strings in L, and the number of states in the DFA is equal to the number of equivalence classes in L> □

Although this is a construction for the DFA, it is still not clear how to construct the DFA algorithmically; it isn't clear how exactly we can find the equivalence classes, and it's also not clear how we can check whether we can check two strings are equivalent.

However, this theorem gives us the following corollary:

Corollary 6.19

Let M be a DFA. Let $L = L(M)$.

Then, the number of states in M is at least the number of equivalence classes in R_L.

Furthermore, there exists a DFA M' with $L(M') = L = L(M)$ with the number of states in M' equal to the number of equivalence classes in R_L.

In other words, the "minimum DFA" for L has precisely a number of states equal to the number of equivalence classes in R_L.

Next time, we'll show how to efficiently find the minimal automaton that is equivalent to M. That is, we want an algorithm which given a DFA M will output another DFA M' such that

- $L(M) = L(M')$
- M' has the fewest states among all DFAs which recognizes L. That is, the number of states in M' is equal to the number of equivalence classes in R_L.

9/15/2022

Lecture 7

Minimal Automata

This week, we'll focus on how to efficiently find the minimal DFA M' that is equivalent to some DFA M.

To start with, given a DFA M for a language L, how can we find the equivalence classes of R_L? Once we have these equivalence classes, we can use the contraction from last time.

Recall that

$$R_M = \{(x, y) \mid x \text{ and } y \text{ lead to the same state in } M\}$$
$$R_L = \{(x, y) \mid \forall z, \text{ either both } xz, yz \in L \text{ or neither}\}$$

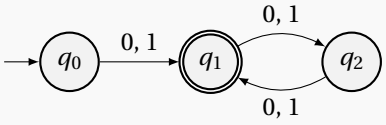
Further, recall that R_M refines R_L. This means that every equivalence class of R_L is a union of equivalence classes for R_M. As such, we just need to check which states in M are equivalent.

In particular, we need to see which states corresponding to equivalence classes in R_M are combined when we consider R_L.

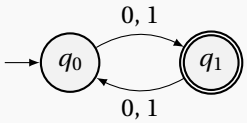
Example 7.1

Consider the language of all strings of odd length.

We have the following DFA:



Notice that q_0 and q_2 are essentially the same; we can merge them together as



Definition 7.2: Equivalent States

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA. We say that two states $p, q \in Q$ are *equivalent* if $\forall z \in \Sigma^*$ if either both paths reading z starting from p and q accept or neither accept.

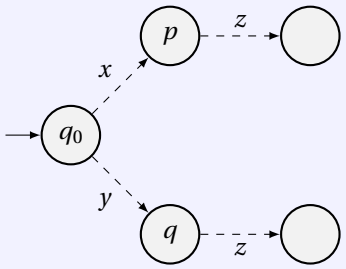
This gives us a relation

$$R_{Q,M} = \{(p, q) \in Q \times Q \mid p \text{ and } q \text{ are equivalent}\}.$$

Lemma 7.3

Let $x, y \in \Sigma^*$. Let p and q be two states reached by running M on x and y respectively. Then, $(x, y) \in R_L \iff (p, q) \in R_{Q,M}$.

Proof. Consider the following diagram.



Suppose $(x, y) \in R_L$. This means that for any $z \in \Sigma^*$, either both $xz, yz \in L$, or both $xz, yz \notin L$.

However, this means that if we start from p and start from q, no matter what z we append, we'll always both accept or both reject. As such, p and q must be equivalent and $(p, q) \in R_{Q,M}$.

On the other hand, suppose $(p, q) \in R_{Q,M}$. This means that for all $z \in \Sigma^*$, the computation paths starting from p and starting from q either both accept or both reject.

This means that for all z, either both $xz, yz \in L$, or $xz, yz \notin L$. As such, x and y are indistinguishable and $(x, y) \in R_L$. □

Observe that while $R_{Q,M}$ is well-defined, it is not clear how to compute it efficiently.

As a first try, notice that just separating the states as accepting or rejecting can distinguish some of them. Formally, we say that p and q are *0-distinguishable* if p is an accept state and q is a reject state, or vice versa. We call this "0-distinguishable", since we only need a string of length 0 to distinguish them.

We can go even further and say that two states p and q are *1-distinguishable* if either p and q are 0-distinguishable or if there is $a \in \Sigma$ such that $\delta(p, a)$ and $\delta(q, a)$ are 0-distinguishable.

In other words, we only need a string of at most length 1 to distinguish the states, i.e. this string ends up at two differently categorized states (i.e. one accepting, one rejecting).

In general, we say that two states p and q are *j-distinguishable* if either p and q are $(j-1)$-distinguishable, or if there is $a \in \Sigma$ such that $\delta(p, a)$ and $\delta(q, a)$ are $(j-1)$-distinguishable.

This leads us to define the following relation:

$$R_{Q,M,j} = \{(p, q) \in Q \times Q \mid p \text{ and } q \text{ are not } j\text{-distinguishable}\}.$$

Lemma 7.4

We claim that $(p,q) \in R_{Q,M,j}$ if and only if $\forall z \in \Sigma^*$ of length at most j, either both paths starting from p and q following z end in an accepting state, or neither paths end in an accepting state.

Proof. We can proceed by induction on j.

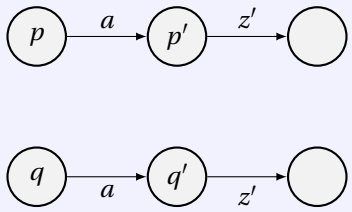
In the base case, for $j = 0$, two states $(p,q) \in R_{Q,M,0}$ if and only if both p, q accept or both reject. This is equivalent to saying that for all z of length 0 (i.e. just the empty string), either both paths following z accept, or both paths reject (the paths are basically going nowhere).

Let $j \geq 1$ and let p, q be two states that are j-indistinguishable (i.e. $(p,q) \in R_{Q,M,j}$). We want to show that for every z of length at most j, both paths following z from p and q either both accept or both reject.

For any $a \in \Sigma$, suppose we let $p' = \delta(p,a)$ and $q' = \delta(q,a)$, i.e. we take one step according to symbol a. We know that p' and q' are not $(j-1)$-distinguishable; otherwise if they *are* $j-1$-distinguishable, then p and q would be j-distinguishable. In particular, we know $(p', q') \in R_{Q,M,j-1}$.

By the inductive hypothesis, for any z' of length at most $j-1$, either both paths starting from p' and q' following z' accept or both reject.

However, any z of length at most j starting from p and q can be broken down into one symbol a, followed by a string z' of length at most $j-1$. This means that p and q must also be j-indistinguishable.



For the other direction, we can prove the contrapositive; if $(p,q) \notin R_{Q,M,j}$, then there exists a string z of length at most j that distinguishes them. This follows from a similar inductive proof, by breaking down z into a single symbol a and a string of length at most $j-1$. □

Here are some observations as a result of this:

- $R_{Q,M,0}$ is refined by $R_{Q,M,1}$, which is refined by $R_{Q,M,2}$, etc.

  Intuitively, we're splitting states into two categories with $j = 0$, and splitting each of these categories further with $j = 1$, etc.

- $R_{Q,M,j}$ can be computed efficiently from $R_{Q,M,j-1}$ with runtime $O(|Q|^2 \cdot |\Sigma|)$

  In particular, for every two states and every character, we are checking whether appending this character (and moving to the next states) makes the states $j-1$-distinguishable.

- If $R_{Q,M,j+1} = R_{Q,M,j}$, then $\forall i \geq j$, $R_{Q,M,i} = R_{Q,M,j}$.

  The previous point means that $R_{Q,M,j}$ depends only on $R_{Q,M,j-1}$. This means that once we reach a point where the refinement doesn't do anything, i.e. $R_{Q,M,j+1} = R_{Q,M,j}$, then the next refinement will also do nothing; the inputs will be the same, and thus will generate the same relation $R_{Q,M,i}$ for further $i \geq j$.

  Proof. Formally, it suffices to show that if $R_{Q,M,j} = R_{Q,M,j+1}$, then $R_{Q,M,j+1} = R_{Q,M,j+2}$ (because then we can apply the same logic starting from $R_{Q,M,j+1}$ and $R_{Q,M,j+2}$).

  Take any two states $(p,q) \in R_{Q,M,j+1}$; we want to show that $(p,q) \in R_{Q,M,j+2}$ as well (the opposite direction is just by definition of refinement). The contrapositive says that if $(p,q) \notin R_{Q,M,j+2}$, then $(p,q) \notin R_{Q,M,j+1}$.

If $(p,q) \notin R_{Q,M,j+2}$, then p and q are $(j+2)$-distinguishable. This means that there are two cases; either p and q are $(j+1)$-distinguishable, in which case we are done, or there exists some $a \in \Sigma$ such that $p' = \delta(p,a)$ and $q' = \delta(q,a)$ are $(j+1)$-distinguishable.

From our assumptions, since $R_{Q,M,j+1} = R_{Q,M,j}$, then p' and q' are also j-distinguishable. This means that p and q must be $(j+1)$-distinguishable; we took a string of at most length $j+1$ to distinguish the two states.

In either case, we have shown that p and q are $j+1$ distinguishable, so $(p,q) \notin R_{Q,M,j+1}$, proving the contrapositive and thus the original claim. □

Notice that we're always guaranteed to stop refining. To see why, we can look at the equivalence classes of $R_{Q,M,j}$. Each time we refine, we increase the number of equivalence classes, but we can't have any more equivalence classes than we do states. This means that we'll always stop at some point before $j = |Q|$.

More formally, we have the following corollary.

Corollary 7.5

Let $n = |Q|$. Then $\forall i \geq n$, $R_{Q,M,i} = R_{Q,M,n}$.

Proof. Let us denote by $\#(R)$ the number of equivalence classes according to R.

We know initially that $\#(R_{Q,M,0}) = 2$, since we have either accepting or rejecting states.

Suppose for contradiction that $R_{Q,M,n} \neq R_{Q,M,n+1}$, where $n = |Q|$. This means that $R_{Q,M,n+1}$ is a true refinement, and as such for every $j \leq n$, $R_{Q,M,j} \neq R_{Q,M,j+1}$. Otherwise, whenever the refinements stop, we'll continue to stop—this would be a contradiction.

Let us look at the number of equivalence classes for each one of these relations. Since these are all true refinements (i.e. they do something), we have

$$\begin{aligned}\#(R_{Q,M,n}) &\geq \#(R_{Q,M,n-1}) + 1 \\ &\geq \#(R_{Q,M,n-2}) + 2 \\ &\vdots \\ &\geq \#(R_{Q,M,0}) + n = n + 2\end{aligned}$$

This is larger than the number of states, which is impossible. This means that $R_{Q,M,n} = R_{Q,M,n+1}$, as claimed. □

As a result, when looking for equivalent states, we only need to look up to $j = n = |Q|$; we don't need to go any further.

This gives us the following algorithm.

As input, we have a DFA $M = (Q, \Sigma, \delta, q_0, F)$; as output, we want to produce a minimal DFA $M' = (Q', \Sigma, \delta', q', F')$.

0. Remove unreachable states in M.
1. Compute $R_{Q,M,0}$. That is, we split the states into either accepting or rejecting, constructing a table for whether (p,q) are related under $R_{Q,M,0}$.
2. For $j = 1$ to $|Q|$, compute $R_{Q,M,j}$ from $R_{Q,M,j-1}$.

   That is, in each iteration, we loop through all pairs of states and symbols and check whether following the symbol gives us a pair of $j-1$-indistinguishable states.
3. Construct $M' = (Q', \Sigma, \delta', q_0', F')$ as follows (here, $[q]$ is shorthand for $[q]_{R_{Q,M,|Q|}}$:
   - $Q' = \{[q] \mid q \in Q\}$

That is, we only take the distinguishable states, choosing representative if there are multiple equivalent states under the relation.

- $\delta'([q], a) = [\delta(q, a)]$

  That is, for the representative state, we follow the path according to the symbol a as normal.

- $q_0' = [q_0]$
- $F' = \{[q] \mid q \in F\}$

Lemma 7.6

$L(M) = L(M')$ and M' has a number of states equal to the number of equivalence classes in R_L for $L = L(M)$.

Proof. To show $L(M) = L(M')$, we need to show that $x \in L(M) \iff x \in L(M')$.

Suppose $x \in L(M)$. This means that the computation path of x in M ends in an accept state $q \in F$. Suppose we take the same string and simulate in M'; by definition, this path reaches $[q]$. This state is also an accepting state according to M', i.e. $[q] \in F'$. This means that $x \in L(M')$.

Suppose $x \notin L(M)$. The proof is exactly the same as before, except we work with rejecting states, i.e. states $q \notin F$ in M and $[q] \notin F'$ in M'.

Next, we want to show that the number of states in M' is equal to the number of equivalence classes of R_L.

The states in M' are $Q' = \{[q_1], [q_2], \ldots, [q_m]\}$, one for each class of equivalent states in M. We want to show that the number of equivalence classes in R_L is exactly m.

By definition, there exists an $x_1 \in \Sigma^*$ such that the computation path of M on x_1 reaches q_1, etc. for each q_i.

We want to show that these x_i's are distinguishable (and as such are in different equivalence classes of R_L).

Recall that we've shown that two strings x_i and x_j are distinguishable with respect to L if and only if q_i and q_j (i.e. the states they end up in) are not equivalent. Since the latter is true by construction, these two strings are indeed distinguishable.

This means that $\#(R_L) \geq m$, since we have at least m distinct strings that are pairwise distinguishable.

Further, since M' computes L, the number of states in M' is at least the number of equivalence classes in R_L. This means that $\left|Q'\right| = m \geq \#(R_L)$.

Together, we have that $\#(R_L) = m$, as claimed. □

9/20/2022

Lecture 8

Streaming Algorithms

8.1 Streaming Algorithms

A lot of data arrives in a sequence $x_1, x_2, x_3, x_4, \ldots$, where we don't have enough memory to store the entire stream. Here, we want to "summarize" this stream of data, manipulate it, or apply some function to it.

Here are some examples of such data streams where streaming algorithms are used:

- Video
- Sound
- Network Traffic

- Machine Learning Data (e.g. labeled examples)

A streaming algorithm has a "small" memory much less than the number of elements seen so far. Further, we go over the data stream once while updating the memory with each new data element.

Streaming algorithms with constant memory isn't too much different from finite state automata, with some exceptions with streaming algorithms.

In particular, we allow the memory to grow *slowly* with the input length m, and we will allow the algorithm to output more than just accepting or rejecting. We also allow the alphabet to be non-finite.

Here are some motivating examples:

- Given a stream of ids, we want to find the number of unique ids
- Given a stream of ids, we want to find the most frequent id

In general, we want to compute $f : \Sigma^* \to \gamma$, with some properties, ex. $f(x_1, \ldots, x_6)$ is the number of distinct elements in $\{x_1, \ldots, x_t\}$.

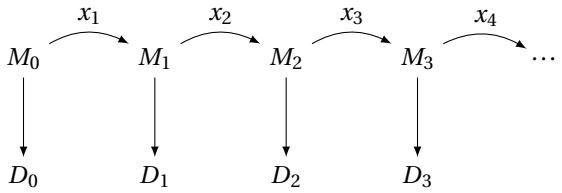
In the streaming model, we have a memory of m bits, with an update role:

$$\delta : \{0,1\}^m \times \Sigma \to \{0,1\}^m,$$

and an output rule

$$D : \{0,1\}^m \to \Gamma.$$

We'll refer to the deterministic model in this course, though the randomized model is very interesting and can do more.



Example 8.1

Suppose we have a stream over the alphabet $\{0,1\}$, and at any point in time n you want to output 1 if and only if the number of 0's is equal to the number of 1's.

We can't do this in constant memory; streaming algorithms with a constant memory are directly equivalent to regular languages and finite automata. Since the language of strings with an equal number of 0's and 1's is not a regular language, we cannot do it with constant memory.

However, we *can* do this with $O(\log n)$ memory. In particular, we store the difference in the number of 1's and 0's in the stream; i.e. add 1 if we see a 1, and subtract 1 if we see a 0. After reading $x_1, \ldots, x_n$, this is just a number between $[-n, n]$, so $1 + \lceil \log_2 n \rceil$ bits suffice to store this count.

At the end, we answer yes if and only if this count is equal to 0.

It turns out that we can't do better than this. To formally prove this, we can use a more finely grained version of Theorem 6.18, where we compute the number of equivalence classes for strings up to some length n.

Recall that two strings $x, y \in \Sigma^*$ are *distinguishable* with respect to a language L if $\exists z \in \Sigma^*$ such that $xz \in L$, $yz \notin L$, or vice versa.

Definition 8.2: Length n Distinguishability

Two strings $x, y \in \Sigma^*$ are *length-n distinguishable* with respect to L if $\exists z \in \Sigma^*$ such that

- $xz \in L$, $yz \notin L$ or vice versa
- xz, yz has length at most n

For notation, we denote $\Sigma^{\leq n} = \{w \in \Sigma^* \mid |w| \leq n\}$ and $L^{\leq n} = \{w \in L \mid |w| \leq n\}$.

Definition 8.3: State Complexity

The *state complexity* of L up to length n is defined as $SC_n(L)$, the number of states in the smallest DFA for $L^{\leq n}$.

Definition 8.4: Memory Complexity

The *memory complexity* of L up to length n is defined as $MC_n(L) = \log_2(SC_N(L))$.

Note that if $L^{\leq n}$ is finite, then $L^{\leq n}$ is regular, since we know we require only finite memory.

Proposition 8.5

If L has m inputs $x_1, \ldots, x_m \in \Sigma^*$ that are length-n distinguishable, then $SC_n(L) \geq m$ and $MC_n(L) \geq \log_2(m)$.

Proof. If $x_1, \ldots, x_n$ are length-n distinguishable with respect to L, then we know that for any $i \neq j$, there exists a z such that $x_i z \in L$, $x_j z \notin L$, or vice versa, and both $|x_i z|, \left|x_j z\right| \leq n$.

Notice that this also directly means that there exists a z such that $x_i z \in L^{\leq n}$, $x_j z \notin L^{\leq n}$, or vice versa.

This means that $x_1, \ldots, x_m$ are distinguishable with respect to $L^{\leq n}$. Myhill–Nerode (Theorem 6.18) states that the number of states in a DFA for $L^{\leq n}$ is at least m.

This means that $SC_n(L) \leq m$, which directly implies that $MC_n(L) \leq \log_2(m)$. □

Example 8.6

Suppose we have the example from earlier, where we want to output 1 if and only if the number of 0's and 1's are equal.

For a lower bound, we show that the state complexity of L up to length n is at least $\frac{n}{2}$.

If we take the strings $\{\varepsilon, 0, 00, 000, \ldots, 0^{\frac{n}{2}}\}$, we can see that all of these strings are length-n distinguishable. Formally, any two strings 0^i and 0^j for $i < j \leq \frac{n}{2}$ are n-distinguishable; appending $z = 1^i$ will make $0^i z \in L$ and $0^j z \notin L$.

Another set of strings could be $\{1^{\frac{n}{2}}, 1^{\frac{n}{2}-1}0, 1^{\frac{n}{2}-2}0^2, \ldots, 10^{\frac{n}{2}-1}, 0^{\frac{n}{2}}\}$. Formally, any two strings $1^{\frac{n}{2}-i}0^i$ and $1^{\frac{n}{2}-j}0^j$ are n-distinguishable; appending $z = 0^{\frac{n}{2}-i}1^i$ will distinguish the strings.

An upper bound is $2n$, since in the earlier solution we need only states for each of $[-n, n]$.

We can think of streaming algorithms as defined in terms of $m = m(n)$ bits, depending on n, an upper bound on the length of the stream, which we know in advance. An alternative definition is to use memory that can increase as we go along the stream. We'll mostly focus on the case where we know the length of the input in advance.

The fact that we know the length of the stream doesn't pose much of an issue; in fact, when showing the lower

bound, if we use the value of n, we're showing that an algorithm that has the *additional* information of the size of the memory still requires some amount of memory. This means that an algorithm *without* this information can only do worse; we know less information about the input.

Example 8.7

Suppose we have a stream of bits, and at any point we output 1 iff the string so far was a palindrome.

Can we do this with constant memory? No, since the language of palindromes is not regular.

Can we give a lower bound on the memory size required to solve this problem? A naive algorithm is to just store the entire stream in memory; we can show that this is pretty much the best we can do.

To show this lower bound, we need to come up with a set of strings that are pairwise length-n distinguishable. We can just take all of the strings of length $\frac{n}{2}$. There are $2^{\frac{n}{2}}$ such strings, and we claim that these are all pairwise length-n distinguishable. In particular, if we have $x, y \in \{0,1\}^{\frac{n}{2}}$ for $x \neq y$, we can just append $z = x^R$ to the end of the strings, and we have $xz = xx^R$ is a palindrome but $yz = yx^R$ is not a palindrome.

This means that $SC_n(L \text{ palindrome} \geq 2^{\frac{n}{2}}$ and $MC_n(L \text{ palindrome}) \geq \log_2(2^{\frac{n}{2}}) = \frac{n}{2}$.

Using this example of palindromes, we can show that counting the number of distinct elements is just as hard. That is, we can use a reduction.

The idea is to embed palindromes into the distinct element problem. That is, if distinct elements is easy, then palindromes is easy; if palindromes is hard, then distinct elements is hard.

In particular, we want to show that if we can solve distinct elements, then we can solve palindromes. To do this,given a stream of bits $x_1, x_2, \ldots, x_n$, we want to embed this information into another stream $y_1, \ldots, y_n$, such that if we know how to count the number of distinct elements among $y_1, \ldots, y_n$, then we know whether $x_1, \ldots, x_n$ is a palindrome.

To construct $y_1, \ldots, y_n$, we can convert x_i with $y_i = (i, x_i)$ from $1 \leq i \leq \frac{n}{2}$, and replace x_i with $y_i = (n-i, x_i)$ for $\frac{n}{2} < i \leq n$.

If x is a palindrome, then we know that $(1, x_1) = (1, x_n)$, $(2, x_2) = (2, x_{n-1})$, etc. In particular, we know that the number of distinct elements of $y_1, \ldots, y_n$ is exactly $\frac{n}{2}$. Otherwise, there are more than $\frac{n}{2}$ distinct elements. This means that x is a palindrome if and only if y has exactly $\frac{n}{2}$ distinct elements.

Here, notice that $y_1, \ldots, y_n$ are over the alphabet $[\frac{n}{2}] \times \{0,1\}$, which is allowed by our streaming model.

Since we can't solve palindromes with less than linear memory, then we also can't solve distinct elements with less than linear memory.

When constructing the reduction, we want the reduction itself to be a streaming manipulation; we want to respect the memory constraints of the problem, so that if we have a good algorithm for one problem, we will also have a good algorithm for the other.

8.2 Communication Protocols

For an alternative perspective on the lower bound, consider a generic stream of length n: $x_1, x_2, \ldots, x_{\frac{n}{2}}, y_1, y_2, \ldots, y_{\frac{n}{2}}$.

The goal is to compute $f(x, y) = 1$ if $x \cdot y$ is a palindrome, and 0 otherwise.

Here are some observations:

1. $M_{\frac{n}{2}}$ is a function of only x (since we've seen only $x_1, \ldots x_{\frac{n}{2}}$.
2. $\forall x \in \{0,1\}^{\frac{n}{2}}$, there exists a unique y such that $f(x, y) = 1$.

This means that the mapping $x \to M_n(x)$ must be one-to-one. Otherwise, there are x and x' that are mapped to the same M, but then the output on $x \cdot y$ should be the same as its output on $x' \cdot y$, which is a contradiction.

This means that the memory size should be at least $\frac{n}{2}$.

In other words, we've shown that after reading the first $\frac{n}{2}$ of the stream, we must have some small amount of memory to keep track of what we've read, otherwise we won't be able to determine the correct output. This gives us the lower bound we want.

This can be seen as a more generic proof technique. We can imagine that the stream is partitioned into two parts, and we want to determine a function $f(x, y)$ on the concatenation $x \cdot y$.

This gives us a connection between streaming algorithms and communication protocols. In particular, suppose we have two players Alice and Bob, each with a part of the input; ex. Alice gets $x_1, \dots, x_{\frac{n}{2}}$ and Bob gets $y_1, \dots, y_{\frac{n}{2}}$. The goal is for Alice and Bob to compute whether $x \cdot y$ is a part of the language.

In particular, Alice can look at her data, summarize it, and send it to Bob, who can look at the message and say whether $f(x, y)$ says yes or no.

If we have a streaming algorithm, then Alice can run the streaming algorithm on the first $\frac{n}{2}$ symbols of the input, and send the internal memory state $M_{\frac{n}{2}}$ to Bob, who can then start from this memory state and continue running the streaming algorithm on the rest of the string, giving a final output.

This means that a fast streaming algorithm is easily converted into as solution to the communication problem. That is, a good streaming algorithm gives a good communication protocol. In the contrapositive, if we can show a lower bound on memory to solve the communication problem, then we also have the same lower bound on memory to solve the streaming algorithm.

A lot of lower bounds on streaming algorithms come from this connection.

Example 8.8

Any *randomized* algorithm that can estimate the number of distinct elements up to time n with additive error $\sqrt{n}$ must have memory $\geq \Omega(n)$.

That is, even with all of this leniency, we still can't solve the problem with less than n memory.

This is a very complicated problem, but we have two parts:

1. (Indyk, Woodruff) We can construct a reduction from a problem in communication complexity called the Grapped-hamming-Distance (GHD) problem: With Alice having $x_1, \dots, x_n$ and Bob having $y_1, \dots, y_n$, the goal is to estimate $\left|\{i \mid x_i \neq y_i\}\right|$ (i.e. the number of symbols where x_i and y_i are different), up to an error of $\pm\sqrt{n}$.
2. (Chakrabarti, Regev) With this reduction, we can construct a lower bound on GHD, which then gives a lower bound for the streaming algorithm for distinct elements.

Communication complexity is a very interesting area of research; the one-way communication model (where Alice sends Bob only one message) was introduced by Shannon in the 1940s (in the same paper that introduced error correcting codes); the two-way communication model was introduced by Yao in the 1980s.

Communication complexity is also useful in various areas of computer science:

- Distributed computing
- Data structures
- Circuit complexity
- Complexity of linear programs
- VLSI (circuit design)

For example, with circuit design, if we split up the circuit into two parts, we can use the reduction to communication protocols to show some lower bound on the amount of information we need to transfer between these two parts of the circuit, giving us a lower bound on the number of wires between these two parts.

9/22/2022

Lecture 9

Context Free Grammars

In the first few weeks, we introduced regular languages as the languages recognized by DFAs and regular expressions. We also showed that there are non-regular languages like $L = \{a^n b^n \mid n \geq 0\}$.

Today, we'll introduce a stronger model of computation called context-free grammars (CFG) that generate context-free languages (CFL).

Context free grammars are *stronger* than regular expressions;

- it has a recursive structure
- it can "count"
- it's used to model human languages
- it's also used a lot in compilers and parsers in different programming languages

Similar to the equivalences between DFAs, NFAs, and regexp, we have that context-free grammars (CFG) are equivalent to context-free languages (CFL), which are equivalent to pushdown automata (PDA).

Let's start with some examples of context-free grammars.

Example 9.1

Suppose we have an alphabet $\{0,1,2\}$. The context-free grammar is defined by some relation rules, with variables on the left, and some collections of variables/symbols on the right.

- $A \to 0A1$
- $A \to B$
- $B \to 2B$
- $B \to \varepsilon$

Grammars consist of:

- a set of variables , e.g. $\{A, B\}$
- a set of terminals/alphabet, e.g. $\{0,1,2\}$
- derivation rules or "production rules": $\{A \to 0A1, A \to B, B \to 2B, B \to \varepsilon\}$
- start variable, i.e. A

Each derivation rule is of the form "variable → sequence of variables and/or terminals".

Starting with A, at any given time we can replace it with a derivation rule. For example, we have

$$\begin{aligned}
A &\to 0A1 && (A \to 0A1)\\
&\to 00A11 && (A \to 0A1)\\
&\to 000A111 && (A \to 0A1)\\
&\to 0000A1111 && (A \to 0A1)\\
&\to 0000B1111 && (A \to B)\\
&\to 00002B1111 && (B \to 2B)\\
&\to 000022B1111 && (B \to 2B)\\
&\to 0000221111 && (B \to \varepsilon)
\end{aligned}$$

What strings can a grammar generate?

- We start with the start variable
- At any point, we maintain a string of variables and terminals
- At any point, we pick a variable x in your string (if it exists), and replace it with the RHS of some derivation rule $x \to \ldots$.
- If no variable exists in your string, then halt.

This sequence of substitutions is called a *derivation*.

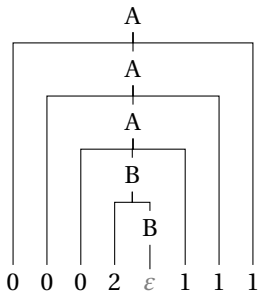
The language of a grammar $L(G)$ is defined to be the set of all strings over Σ that can be generated/derived by the grammar G.

Example 9.2

What kind of strings could be generated by the previous example?

All strings would be of the form $0^n 2^m 1^n$, for all $n, m \geq 0$; mathematically, $L(G) = \{0^n 2^m 1^n \mid n, m \geq 0\}$.

Another way to view the derivation is with parse trees:



Putting all the leaves at the same level, the string is at the very bottom.

Definition 9.3: Leftmost Derivation

A leftmost derivation is a derivation in which each step replaces the leftmost variable.

Example 9.4

Consider the grammar

$$E \to E + E$$
$$E \to E \times E$$
$$E \to 0 \mid 1 \mid 2$$

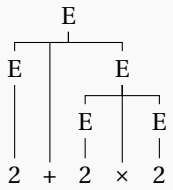
What is a leftmost derivation of $2 + 2 \times 2$?

We have

$$\begin{aligned} E &\to E + E && (E \to E + E) \\ &\to 2 + E && (E \to 2) \\ &\to 2 + E \times E && (E \to E \times E) \\ &\to 2 + 2 \times E && (E \to 2) \\ &\to 2 + 2 \times 2 && (E \to 2) \end{aligned}$$

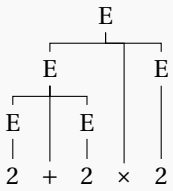
Notice that each time we used a derivation, we replaced the left-most variable in the string, and replaced it with its corresponding expression.

This corresponds to a parse tree:



Another leftmost derivation is as follows:

$$\begin{aligned} E &\to E \times E && (E \to E \times E) \\ &\to E + E \times E && (E \to E + E) \\ &\to 2 + E \times E && (E \to 2) \\ &\to 2 + 2 \times E && (E \to 2) \\ &\to 2 + 2 \times 2 && (E \to 2) \end{aligned}$$



The fact that there could be different parse trees and different derivations for the same string means that there could be issues or conflicts with assigning semantics to these strings (ex. in mathematics, or in programming languages, etc.)

Definition 9.5: Context-Free Grammar

A CFG is a 4-tuple $G = (V, \Sigma, R, S)$:

- V is a finite set of *variables*
- Σ is a finite set of *terminals* (i.e. the alphabet)
- R is a finite set of *derivation rules*, with each rule of the form

  $$\text{variable} \to \text{sequence of variables and terminals.}$$

- $S \in V$ is the start rule.

We want V and Σ to be disjoint, to avoid confusion.

Intuitively, the reason why this is a *context-free* grammar is because we don't care what we've derived before; we focus only on a single variable and replace it.

Definition 9.6: Yields

For $u, w \in (V \cup \Sigma)^*$, we say that $u \implies w$, or "u yields w" if we can replace a variable in u with the RHS of a derivation rule to get w.

Definition 9.7: Derives

For $u, w \in (V \cup \Sigma)^*$, we say that u derives w, or $u \overset{*}{\Longrightarrow} w$ if $u = w$ or if

$$u \Longrightarrow u_1 \Longrightarrow u_2 \Longrightarrow \cdots \Longrightarrow u_k \Longrightarrow w.$$

For shorthand notation, we sometimes just write the derivation rules; further, with the example from Example 9.1, we can also further make this shorthand

$$A \to 0A1 \mid B$$
$$B \to 2B \mid \varepsilon$$

Context-free grammars are especially good in capturing recursive structure, as we'll show in the next example.

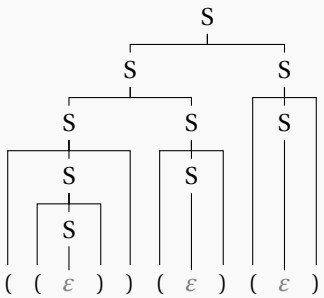
Example 9.8

With the alphabet $\{(,)\}$, to capture all strings with properly nested parentheses.

$$S \to (S)$$
$$S \to SS$$
$$S \to \varepsilon$$

Show that $S \overset{*}{\Longrightarrow} (())()()$.

With a parse tree, we have

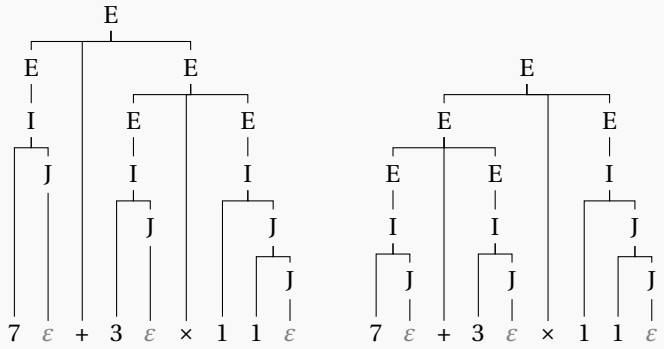


Example 9.9

We can write a grammar for all simple mathematical expressions with $+$, $\times$.

$$E \to E + E \mid E \times E \mid I \qquad \text{(expressions)}$$
$$I \to 1J \mid 2J \mid \cdots \mid 9J \mid 0 \qquad \text{(integers)}$$
$$J \to \varepsilon \mid 0J \mid 1J \mid \cdots \mid 9J$$

How do we generate $7 + 3 \times 11$? There are two possible parse trees:



Note that these two trees have different semantics.

Note that ambiguity is a property of the *grammar*, not of the language. That is, we can have two grammars that generate the same language, but one of them is ambiguous and the other is not.

9.1 Syntax Analysis

Given a string of tokens $w \in L(G)$, we can find a parse tree for w. Syntax analysis is the process of interpreting what these tokens mean.

Example 9.10

A non-ambiguous grammar for the previous example is as follows:

$$
\begin{aligned}
E &\to E + T \mid T && \text{(expression)} \\
T &\to T \times F \mid F && \text{(term)} \\
F &\to (E) \mid I && \text{(factor)} \\
I &\to 1J \mid 2J \mid \cdots \mid 9J \mid 0 && \text{(integer)} \\
J &\to \varepsilon \mid 0J \mid 1J \mid \cdots \mid 9J
\end{aligned}
$$

Here, when designing this grammar, we're enforcing the fact that multiplication precedes addition. It's perhaps a little bit backwards, since we're deriving addition before multiplication, but when we compute the expression, we start from the bottom of the tree, so we compute the multiplication before the addition with this grammar.

9.2 Context Free Languages and Regular Languages

We claim that context free languages are at least as strong as regular languages. We'll be giving two proofs of this claim; one through closure properties, and one as a reduction from DFA to CFG.

9.2.1 Closure Properties

Proposition 9.11

Context free languages are closed under the union.

In particular, if G_1 is a CFG for L_1, and G_2 is a CFG for L_2, then there exists a grammar G that is a CFG for $L = L_1 \cup L_2$.

Proof. We know that G_1 has some start state S_1, and G_2 has some start state S_2.

We can then define G by adding another state S as the new start state, with the new derivation rule $S \to S_1 \mid S_2$.

This means that we can either start in G_1 or start in G_2, giving us the ability to derive everything in the union, and everything in the union can be derived from G.

One caveat is that G_1 and G_2 need to be defined on disjoint sets of variables; just to make sure that this doesn't happen, we can just change the names of these variables so that they are disjoint. □

Proposition 9.12

Context free languages are closed under concatenation.

In particular, if G_1 is a CFG for L_1, and G_2 is a CFG for L_2, then there exists a grammar G that is a CFG for $L = L_1 \cdot L_2$.

Proof. Very similar to the proof for unions, we can again add a new start state S, with the new derivation rule $S \to S_1 S_2$. □

Proposition 9.13

Context free languages are closed under the Kleene star.

In particular, if G_1 is a CFG for L_1, then there exists a grammar G that is a CFG for $L = L_1^*$.

Proof. Very similar to the previous proofs, we can add a new start state S with the derivation rule $S \to \varepsilon \mid SS_1$. □

We could also verify that there is a CFG for the empty string, symbols from the alphabet, the empty set, etc. as base cases.

We then claim that if R is a regexp over Σ, then $L(R)$ is a CFL.

In particular, recall that a regexp, R is either ε or a symbol from Σ, or it is one of $R_1 \cup R_2$, $R_1 \cdot R_2$, or R_1^*. In any of these cases, R is still a CFL, so all regexp are CFLs.

Something interesting is that context free languages are not closed under intersections or complements.

Example 9.14

Consider $L_1 = \{0^n 1^n 2^m \mid n, m \geq 0\}$ (this is a CFL), and $L_2 = \{0^n 1^m 2^m \mid n, m \geq 0\}$ (similarly, this is also a CFL).

The intersection is $L_1 \cap L_2 \{0^n 1^n 2^n \mid n \geq 0\}$; we'll show next time that this is not a CFL. Intuitively, it's impossible to recurse in three different places.

9.2.2 Reduction from DFA

Starting from a DFA $M = (Q, \Sigma, \delta, q_0, F)$, we can derive a grammar $G = (Q, \Sigma, R, q_0)$, with derivation rules:

- For every $\delta(q, \sigma) = p$ in M, we add a rule $q \to \sigma p$ in G.
- For every $q \in F$, we add a rule $q \to \varepsilon$.

We claim that any computation path accepting w in M is a derivation of w in G, and vice versa.

In particular, the computation path is

$$q_0 = r_0 \xrightarrow{w_1} r_1 \xrightarrow{w_2} r_2 \xrightarrow{w_3} \cdots \xrightarrow{w_n} r_n \in F.$$

The corresponding derivation is

$$r_0 \implies w_1 r_1 \implies w_1 w_2 r_2 \implies w_1 w_2 w_3 r_3 \implies \cdots w_1 w_2 \cdots w_n r_n \implies w_1 \cdots w_n.$$

In one direction, each transition in the computation path corresponds directly to a rule being applied, until we eventually get to an accepting state, in which case we apply the $q \implies \varepsilon$ rule. In the other, notice that we'll always have intermediate containing some sequence of terminals and one variable, which gets converted into ε at the very end.

Notice that all rules fall in one of the following forms:

$$V \to TV$$
$$V \to \varepsilon$$

All regular languages can be generated by a rule set of this form.

Similarly, all context free grammars can be generated with rules of the following forms:

$$V \to VV$$
$$V \to T$$
$$S \to \varepsilon$$

This is called Chomsky's Normal Form, and the parse tree will look like a binary tree.

Example 9.15

Design a CFG for $L_1 = \{0^n 1^n \mid n \geq 0\}$.

The set of rules are:

$$S \to 0S1 \mid \varepsilon.$$

Example 9.16

Design a CFG for $L_2 = \{w \mid w \text{ is a palindrome}\}$.

The set of rules are:

$$S \to \varepsilon \mid 0 \mid 1$$
$$S \to 0S0 \mid 1S1$$

Example 9.17

Design a CFG for $L_3 = \{w \mid w \text{ contains at least three 1's}\}$.

The set of rules are:

$$S \to X1X1X1X$$
$$X \to 0X \mid 1X \mid \varepsilon$$

Here, notice that X can derive all possible strings.

9/27/2022

Lecture 10

Pushdown Automata, Pumping Lemma for CFL

10.1 Pushdown Automata

Pushdown automata are non-deterministic finite automata equipped with an (unbounded size) stack. The stack starts out empty, and we can push to the top, look at the top, or pop from the top of the stack.

Unlike DFAs/NFAs, PDA have more than a constant memory. The stack can hold unlimited amount of information/memory, but access to this information is restricted.

Pictorially, Fig. 10.1 shows a diagram for a finite state machine running a DFA. We have a control block with a pointer to an input symbol, and we move along the input symbol reading in this information, updating the internal state.

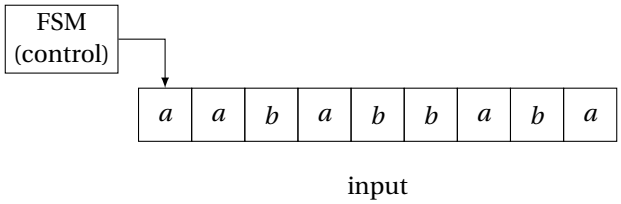


Figure 10.1: Diagram for a DFA

Figure 10.2 shows a diagram for a finite state machine running a PDA. Here, we have the same pointer to the input, except now we have an additional stack that we can push and pop to.

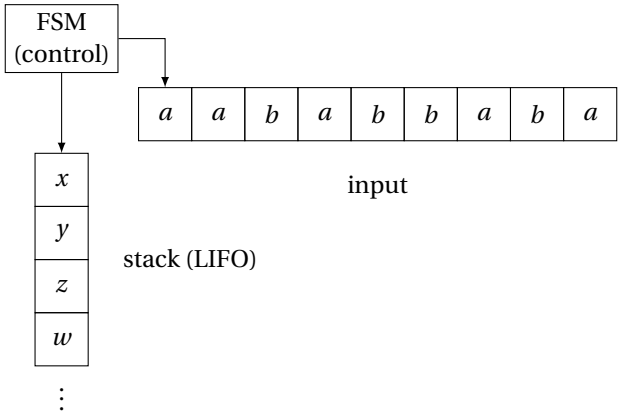


Figure 10.2: Diagram for a PDA

In particular, at any point in time, we update the internal state in the control, and then either push an element to the stack, or pop an element from the stack. This move can depend on what is at the top of the stack. After we perform a stack operation, we then move our input pointer to the right.

Example 10.1

Design a pushdown automata for $L = \{0^n 1^n \mid n \geq 0\}$.

Informally, we can use the stack to essentially count. In particular, we keep pushing symbols to the stack until we see a 1. After we see our first 1, we can start popping from the stack as long as we see 1's. If the number of 0's are equal to the number of 1's, then the stack will be empty.

That is, if the stack is empty when the input ends, then we accept. If anything else happens (i.e. if there is more input to consider but the stack is empty, or if there is still 0's leftover, or if we see another 0 after we see our first 1), then we reject.

Here, PDAs will use non-determinism. There is also a model of deterministic PDAs, but we focus on the non-deterministic model in this course. Unlike DFAs and NFAs, non-deterministic PDAs are *strictly more powerful* than deterministic PDAs, but this is out of scope for the class.

Definition 10.2: Pushdown Automata

A pushdown automaton (PDA) is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$, where

- Q is a finite set of states
- Σ is the input alphabet (a finite set)
- Γ is the stack alphabet (a finite set, potentially much larger than the input alphabet)
- $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to P(Q \times \Gamma_\varepsilon$ is the transition function

  The transition function gets three inputs; the current state, the next input symbol, and the top element of the stack (or ε).

  It then decides (non-deterministically) to push/pop/do nothing to the stack, and update the current state.
- $q_0 \in Q$ is the start state
- $F \subseteq Q$ are the accept states

Here, $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$ and $\Gamma_\varepsilon = \Gamma \cup \{\varepsilon\}$.

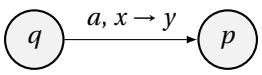
To understand δ more, let us give some specifics.

Let $q \in Q$, $a \in \Sigma_\varepsilon$, and $x \in \Gamma_\varepsilon$ be inputs to δ.

Let $p \in Q$ and $y \in \Gamma_\varepsilon$. We should interpret $(p, y) \in \delta(q, a, x)$ as:

- If $x = \varepsilon$ and $y \in \Gamma$, then y is pushed to the stack.
- If $x \in \Gamma$, and $y = \varepsilon$, then x is popped from the stack (only if the top of the stack is x).
- If $x = \varepsilon$ and $y = \varepsilon$, then do nothing to the stack.
- If $x \in \Gamma$ and $y \in \Gamma$, then we swap x and y (only if the top of the stack is x)

In the state diagram, we have



Here, a is the symbol we see, x is at the top of the stack, and y is what we put at the top of the stack after the transition.

Definition 10.3: Accept (PDA)

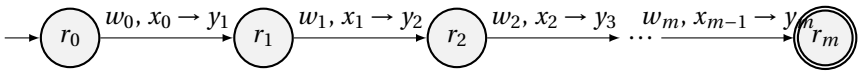
A PDA *accepts* a string w if w can be written as $w = w_1 \cdots w_n$ for $w_i \in \Sigma_\varepsilon$ and there exists a sequence $r_0, r_1, \ldots, r_m \in Q$ and a sequence $s_0, s_1, \ldots, s_m \in \Gamma^*$ such that

- *Initialization*: $s_0 = \varepsilon$, and $r_0 = q_0$
- *Step*: For $i = 0, 1, \ldots, m-1$, $(r_{i+1}, y_{i+1}) \in \delta(r_i, w_{i+1}, x_i)$, where $s_i = x_i \cdot t$, $s_{i+1} = y_{i+1} \cdot t$, for $x_i, y_{i+1} \in \Gamma_\varepsilon$

and $t \in \Gamma^*$.

- *Finally*: $r_m \in F$

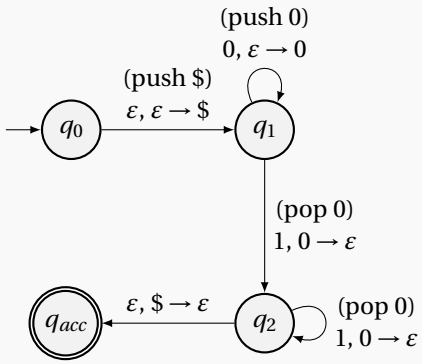
Pictorially, we have the computation path



Example 10.4

Design a pushdown automata for $\{0^n 1^n \mid n \geq 0\}$.

An initial question: how do we tell whether the stack is empty? We can push a unique symbol at the top of the stack initially; when we look at the top of the stack, if we see this unique symbol, we know that the stack is empty.



What would the PDA do on 000111?

symbol	state	action	stack
-	q_0	-	-
-	q_1	push \$	\$
0	q_1	push 0	\$ 0
0	q_1	push 0	\$ 0 0
0	q_1	push 0	\$ 0 0 0
1	q_2	pop 0	\$ 0 0
1	q_2	pop 0	\$ 0
1	q_2	pop 0	\$
-	q_{acc}	pop \$	-

At this point, there is nothing else we can do, while we have more input symbols left. This means that the string is rejected; there is no possible computation path to an accepting state.

Example 10.5

Design a pushdown automata for palindromes (of even length), i.e. $L = \{w w^R \mid w \in \{0,1\}^*\}$.

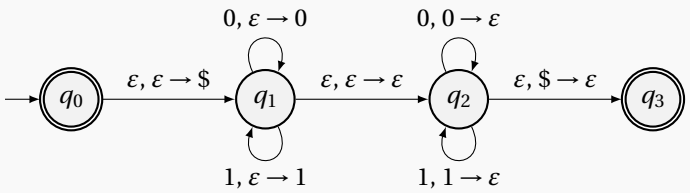
We'll be crucially using non-determinism here; if we know the middle point of the string, it's not hard to check whether the string is a palindrome; we'd push the first half to the stack, and pop these symbols when looking at the second half, comparing them for equality. We can just use non-determinism to guess the middle point here.

On 011110, this PDA would behave like so:

symbol	action	stack
-	push \$	\$
0	push 0	\$ 0
1	push 1	\$ 0 1
1	push 1	\$ 0 1 1
	guess middle point	
1	pop 1	\$ 0 1
1	pop 1	\$ 0
0	pop 0	\$
-	pop \$	-

We see that the stack is empty, so we accept the string.

Formally, the pushdown automata can be described with the following state diagram:



10.2 PDA and CFG

Recall that we said that a language is context-free if it is described by a context-free grammar. We'll show today that pushdown automata are equivalent to context-free grammars.

To show that a CFG is equivalent to a PDA, we want to simulate a CFG with a PDA, and vice versa.

To use a PDA to simulate a CFG, an idea is to perform a left-most derivation, and use a stack to maintain the intermediate strings.

Let G be a CFG; a string $w \in L(G)$ iff there is a derivation sequence of w from the start variable.

At any point, we want to replace the leftmost variable with a sequence of variables and terminals according to some derivation rule. The idea is to use the stack to store the current intermediate string of variables and terminals. Ideally, we'd start with a stack of only the start variable, and we end with a stack containing the symbols in w.

However, some complications arise.

Example 10.6

Consider the grammar

$$S \to 0S1 \mid X$$
$$X \to 2X$$

Consider the string $w = 0002111$; we want to somehow simulate the derivation

$$S \to 0S1 \to 00S11 \to 000S111 \to 000X111 \to 0002X111 \to 0002111$$

Going from a stack of $\{S\}$ to $\{0, S, 1\}$ (0 is the top of the stack), we can just pop S and push $\{0, S, 1\}$. However, the next step poses an issue; we can only see 0, but not an S, so we don't know what's in the middle of the stack.

However, notice that once we derive the 0 at the beginning of the string, nothing will ever erase the 0. This means that we can pop the 0 and compare it with the input string. If they're the same, we keep going, and if they aren't then we reject.

Generally, as long as there are terminals at the top of the stack, we can pop them and compare with the input string. Otherwise, if we see a variable, then we can non-deterministically replace it with some derivation rule.

Eventually, we'll go through the entire string, and we'll go through the entire input string and we want to check for an empty stack with the unique $ trick from earlier.

Theorem 10.7

A language is context-free if and only if it is recognized by some pushdown automata.

Proof. To show that a CFG is equivalent to a PDA, we want to simulate a CFG with a PDA, and vice versa.

To use a PDA to simulate a CFG, an idea is to perform a left-most derivation, and use a stack to maintain the intermediate strings.

Let G be a CFG; a string $w \in L(G)$ iff there is a derivation sequence of w from the start variable.

Pseudocode for a PDA for $L(G)$ is as follows:

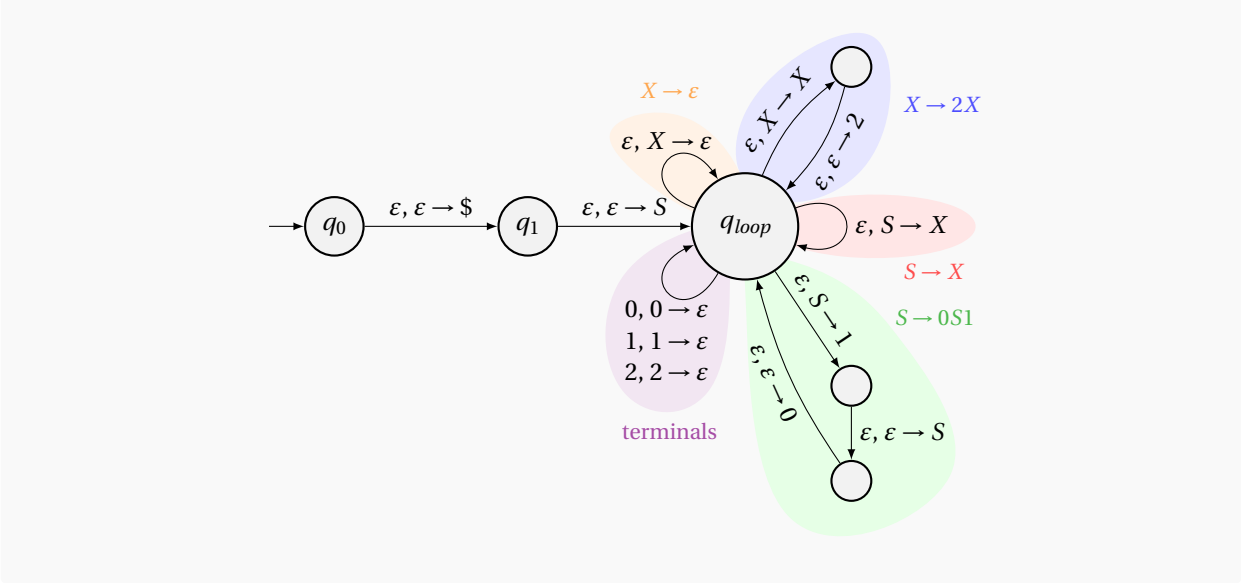
1. Push a $ to the stack, and push the start variable to the stack.
2. Repeat:
   (a) If the top of the stack is av variable A, non-deterministically pick a derivation rule $A \to \cdots$. Pop A and push the string of variables and terminals.
   (b) If the top of the stack is a terminal a, compare it to the next symbol in the input. If matched, pop a and move the input cursor to the right.
   (c) If the top is $, and we finished reading the string, then we accept.

For the other direction, see the textbook; lemma 2.27. □

Example 10.8

We can use the procedure to construct a DFA for the following grammar:

$$S \to 0S1 \mid X$$
$$X \to 2X \mid \varepsilon$$



10.3 Non-Context Free Languages

Definition 10.9: Chomsky's Normal Form

A CFG G is in *Chomsky's Normal Form* if every rule is of the form $A \to BC$ where A, B, C are variables, or of the form $A \to a$ where A is a variable and a is a terminal.

In addition, we permit $S \to \varepsilon$.

We claim that for any CFG G, there exists an equivalent CFG G' is Chomsky's normal form. Intuitively, we can always reduce rules of the form $A \to BCD$ into something like $A \to XD$ and $X \to BC$, etc.

We know that $L_1 = \{0^n 1^n \mid n \geq 0\}$ is non-regular but context-free. What about $L_2 = \{0^n 1^n 2^n \mid n \geq 0\}$?

Here, we can introduce the pumping lemma for CFL.

Informally, in a CFL, any long enough string can be "tandem" pumped, i.e. there are two substrings that can be "pumped together".

Lemma 10.10

Let L be a context-free language.

Then, there is a constant $p = p_L$ (the pumping constant for L) such that for every $w \in L$ with $|w| \geq p$ can be "tandem pumped".

That is, w can be written as $w = uvwxy$ such that

- $|vwx| \leq p$
- $|v| + |x| \geq 1$
- $\forall k \in \mathbb{N}$, $uv^k wx^k y \in L$

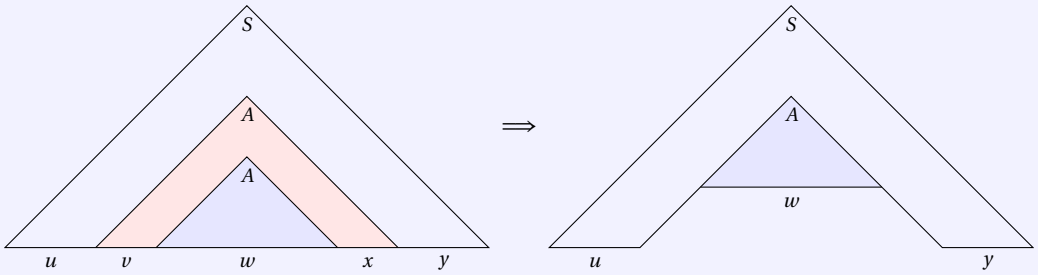
Proof. For a proof idea, we know that if L is a CFL, then there exists a CFG G for L in Chomsky's normal form.

Consider a very long string s in L, with $|s| \geq p$. Consider a parse tree for s; Since G is in CNF, then the parse tree is a binary tree (i.e. each variable has exactly two children). In particular, the number of leaves (equal to

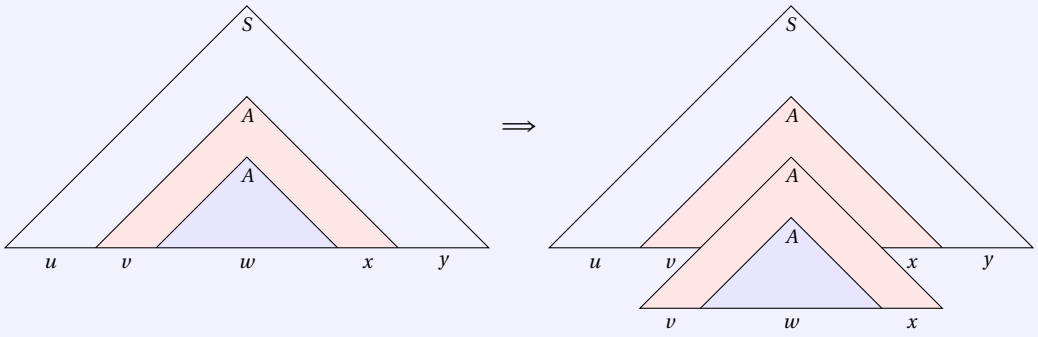
p, since this corresponds to the length of the string) is at most 2^d for depth d, so $d \geq \log_2 p$.

If $\log_2 p$ is larger than the number of variables, then there must exist a path that has a duplicate variable.

Looking at the parse tree, we can visually see how uwy can be derived, replacing the first occurrence of A with the second occurrence of A:



We can also see how uv^2wx^2y can be derived, replacing the second occurrence of A with the first occurrence of A to go down further; we can repeat the same process to get to uv^kwx^ky.



□

Example 10.11

Show that $L_2 = \{0^n1^n2^n \mid n \geq 0\}$ is not a CFL.

Proof. If L is a CFL, then there exists a p such that all strings $w \in L$ of length $\geq p$ can be "tandem pumped".

Consider $w = 0^p1^p2^p$. We want to show that w cannot be pumped. In particular, for any partition $w = uvwxy$ such that $|vwx| \leq p$ and $|v| + |x| \geq 1$.

We claim that v can't have more than one type of character, and x can't have more than one type of character (otherwise, we'd have something like $0^i1^j0^k$, which are rejected).

This means that among v and x, we can't possibly pump all three characters at the same time; there will always be some character that is not pumped, and there will be an unequal amount of each character in the pumped string. □

10/4/2022

Lecture 11

Turing Machines

In the middle third of the course, we'll be talking about computability. Here, we're interested in what can and cannot be computed, and we'll introduce a new model of computation called *Turing machines*.

From now on, we'll be talking about Turing machines, since it is the most powerful model so far; anything we've talked about so far can be simulated by a Turing machine.

So far, we've described and analyzed several models of computation.

We first talked about finite state automata, which corresponds to computation with constant memory. This model is very restricted, and can't even count—we showed this restriction through the pumping lemma and Myhill-Nerode.

We then talked about pushdown automata, which corresponds to computation with unlimited memory but restricted access (i.e. a LIFO stack). This model has more capabilities, and we can recognize languages like 1^n2^n, or palindromes; however, it still can't recognize languages like $0^n1^n2^n$.

The next model we will consider is much more powerful, and is able to recognize languages like $0^n1^n2^n$, or even $0^n1^n2^n3^n$, etc.

11.1 Turing Machines

A Turing machine is a mathematical model that captures what computers can do. It turns out that Turing machines are equivalent in power to any program in Python, C, etc. or any other language.

This model is very similar to a DFA (recall the diagram from Fig. 10.1).

In particular, we have a finite state control, with a pointer to the input, where each step the pointer moves right.

In a Turing machine, there are only a few differences:

- In each step, the pointer can move left or right
- In each step, we can both read and write
- The input tape is infinitely long (to the right), initialized to be blank.

As a remark, there is an edge case where we try to go left in the left-most cell; we'll say that we just stay put.

Formally, we have the following definition:

Definition 11.1: Turing Machine

A Turing Machine is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{acc}, q_{rej})$:

- Q is the finite set of states
- q_0 is the start state
- Σ is the input alphabet
- q_{accept} is the special accept and halt state, and q_{reject} is the special reject and halt state

  With q_{accept} and q_{reject}, whenever the Turing machine reaches this state, it accepts/rejects and stops.
- Γ is the finite tape alphabet, with $\Sigma \subseteq \Gamma$, and in addition the blank space $\_ \in \Gamma$, as this is the initial value of the infinite tape.
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$.

  In particular, we take in the current state and a symbol from the tape alphabet, and we output three

things: the next state, what we write to the tape, and whether we move left or right.

Definition 11.2: Language (Turing machine)

The *language* of a Turing machine M is the set of strings accepted by M. This is also called *the language recognized by M*.

Notice that on a given input $x \in \Sigma^*$, a TM M may either:

1. Halt and accept after finitely many steps
2. Halt and reject after finitely many steps
3. Does not halt (i.e. infinite loop)

The third option is new; it was impossible in our previous models. In particular, with a DFA we will only make one step for each symbol in the input, whereas in a Turing machine, we can modify the input and we don't have a known limit for the steps it takes.

Example 11.3

Consider the following program:

```
def foo():
    n = 5
    while n < 1000:
        n += 2

foo()

```

Since we'll always at some point exceed $n = 1000$, this program will always halt.

However, consider the following program:

```
def bar():
    n = 5
    while n != 1000:
        n += 2

bar()

```

Since n is always odd, it'll never be exactly equal to 1000, so it'll never halt.

Example 11.4

Consider the following example:

```python
def is_prime(n):
    for i in range(2, n):
        if n % i == 0:
            return False
    return True

def goldbach_conjecture():
    n = 6
    while True:
        flag = False
        for i in range(2, n):
            j = n - i
            if is_prime(i) and is_prime(j):
                flag = True

        if flag == False:
            print("halt!")
            return

        n += 1

goldbach_conjecture()

```

Nobody knows whether this program halts or not; it depends on whether the Goldbach conjecture is true.

This example tells us that determining whether a program halts is at least as hard as solving the Goldbach conjecture—this is a teaser for what we'll be talking about next.

Example 11.5

Give a Turing machine for the language $L = \{w\#w \mid w \in \{0,1\}^*\}$, with the input alphabet $\{0,1,\#\}$. (Note that this language is not context-free; we'll prove this in the homework.)

The tape for the Turing machine will initially look like this:

w_1	w_2	$\cdots$	w_n	$\#$	w_1	w_2	$\cdots$	w_n			$\cdots$

The tape alphabet will be $\Gamma = \{0,1,\#,\_,X\}$. In particular, we'll add another symbol X to the tape alphabet—it'll be used to erase symbols on the tape.

Intuitively, let us think about what we can do if we have a slightly different machine with two pointers. Here, we can move one of the pointers until we hit a "#" symbol, and start at the symbol immediately after. Now, we can move both pointers at the same time, comparing symbols until we reach the end of the input.

With only one pointer, we want to find a way to simulate the two pointers with just one pointer. In particular, we can move between the two copies of w, picking up where we last left off. To figure out where to go, we'll leave some breadcrumbs with our extra symbol X.

At a high level, we have the following steps:

1. Mark the first letter with X, and "remember" it in one of the finite states.

   Next, we go right until we see a #, and then move one more step, where we'll see a 0 or a 1.

2. Check that the letter we remember is the same as the current letter. If not, then reject immediately.

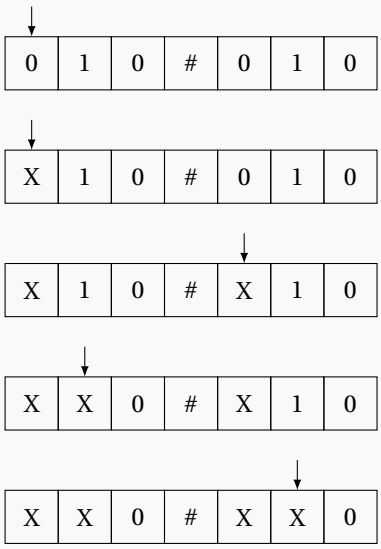
3. Otherwise, the letters are the same, so we will first delete the current letter (since we've already checked it), marking it with an X, and we go left, passing the #, until the next X in the tape.

   Next, we can take one step to the right. If the current letter is a #, then we check the entire tape is marked—if so, then we've check all the letters, and we accept. Otherwise, we reject, since there are more inputs that we haven't seen before.

   Otherwise (i.e. the current letter is not #), go back to step 1.

As an edge case, it could be the case that the input even match $(0 \cup 1)^*\#(0 \cup 1)^*$ in the first place—we can just check this at the beginning before the steps, i.e. check that the input matches the regular expression $(0 \cup 1)^*\#(0 \cup 1)^*$ (we can do this, since a DFA can do this).

The first few steps of execution for such a Turing machine is as follows:



11.2 Variants on the Turing Model

11.2.1 Staying Put

Recall the standard model of the Turing machine; in each step, we can either move left or move right, but we can't stay in the same place. Consider a variable where we can move left, move right, or stay in the same place—is this more powerful?

It turns out that it's still equivalent; we can show this formally by showing that one can simulate the other.

One direction is very easy; if we can implement something where we only move left or right, we can definitely implement something where we can also stay. In the other direction, we want to show that a machine that can stay put can also be simulated with a standard Turing machine. The idea is that to stay put, we can just move right and the immediately move left.

In particular, suppose we have a machine that can move left, right, or stay, with states $\{q_0, q_1, \ldots, q_n\}$. In an equivalent standard Turing machine, we'd have the states $\{q_0, q_1, \ldots, q_n, q_0', q_1', \ldots, q_n'\}$.

Here, we define

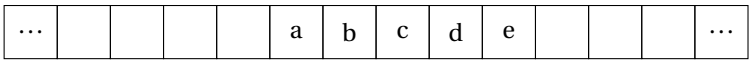
$$\begin{aligned}
\delta(q_i, a) = (q_j, b, L) &\implies \delta'(q_i, a) = (q_j, b, L) \\
\delta(q_i, a) = (q_j, b, R) &\implies \delta'(q_i, a) = (q_j, b, R) \\
\delta(q_i, a) = (q_j, b, S) &\implies \delta'(q_i, a) = (q_j', b, R) \\
&\text{and } \delta'(q_j', a) = (q_j, a, L)
\end{aligned}$$

11.2.2 Two-way Infinite Tape

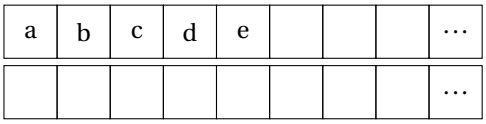
Recall that in the standard model of the Turing machine, the tape is infinite in only one direction. Consider the variant on a Turing machine where we have a two-way infinite tape. Is this variant more powerful?

It turns out that it is equivalent. To show this, it suffices to show that any two-way infinite TM can be simulated by a standard TM M'.

In particular, we have the following tape:



We can imagine that we flip the left half of the tape to form two one-sided infinite tapes stacked on top of each other:



Here, we can imagine that each column of two cells is one "symbol" in an extended alphabet. Redefining the transition function, we can simulate this two-sided infinite tape with a standard Turing machine.

11.2.3 Multitape Turing machines

Consider a variant where we have multiple (finite) different tapes, each with its own pointer, not necessarily synchronized. Is this variant more powerful than the standard model?

One solution is to interleave the tapes. We can keep track of where the individual pointers are by extending the alphabet, marking the symbols with an arrow if a pointer is currently at that location. To simulate the machine, we can then iterate through the tape until we hit a marked symbol, and simulate the machine working only on the subset of interleaved cells.

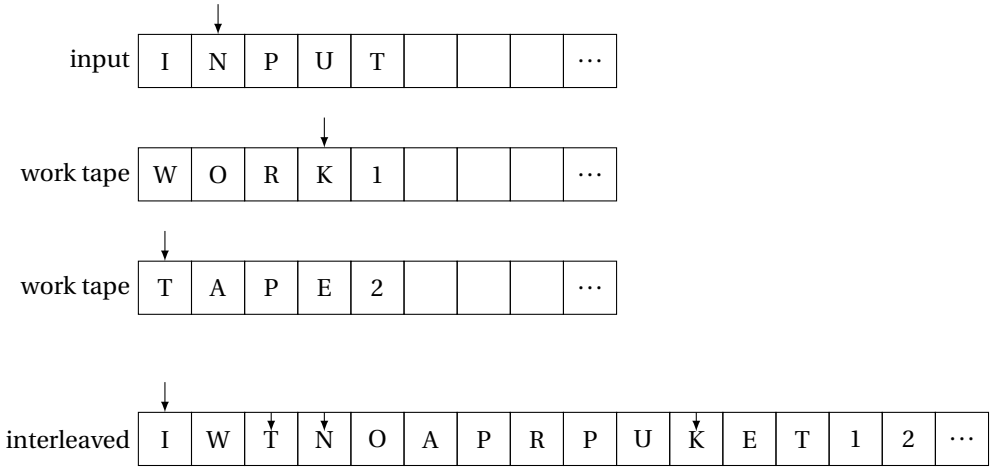


Figure 11.1: Equivalence between a multitape Turing machine and a standard Turing machine

Alternatively, we could also stack the tapes, letting each column be an individual symbol; we can then redefine the tape alphabet and transition function accordingly as well, much like with the two-sided infinite tape.

Note that this simulation *is* costly; we need a $O(n)$ scan in order to simulate one step of the multitape Turing machine.

This equivalence is very helpful; if we can create a Turing machine with multiple tapes, then there will exist some Turing machine that does the same thing with just one tape. This makes it a lot easier to simulate various languages with Turing machines.

10/6/2022

Lecture 12

Turing Machines (cont.)

12.1 Nondeterministic Turing Machines

A non-deterministic Turing machine is a lot like a standard Turing machine, but it accepts an input x if *there exists* a computation path on x that halts and accepts. (This is a lot like how NFAs generalize DFAs.)

In particular, for an input x, there could be many (and even infinite) computation paths, and some paths are infinitely long. At each point in time, there are only finitely many options for the next move (i.e. at most $|Q \times \Gamma \times \{L, R\}|$).

In the third part of this course, we'll talk a lot about this model, especially when talking about P vs. NP.

It turns out that we can simulate a non-deterministic TM with a deterministic TM; how do we do this?

We can't use the subset construction in the same way we simulated an NFA with a DFA, because the internal state of a Turing machine also includes the unbounded tape. This means that the number of internal states could be infinite, so we can't possibly keep track of all possible states for a non-deterministic Turing machine in a deterministic Turing machine.

If we look at the tree of possible branches, a possible idea is to search through this tree, looking for the accept state. An initial idea is to use DFS, but we can get stuck at a branch that never halts. BFS could perhaps work, but we can use a variant of DFA, called *iterative deepening DFS*.

That is for $i = 1, 2, \ldots$, we check all paths of length $\leq i$ in the tree. This way, we'll never get stuck in a branch that never halts. It's perhaps not the most efficient, but we don't require too much memory; all we need to keep track of is the current path and the counter for i.

To simulate a non-deterministic Turing machine, we'll use multiple tapes; see Fig. 12.1

I	N	P	U	T				⋯
1	1	1	1	⋯	1			
d_1	d_2	d_3	d_4	⋯	d_i			

Figure 12.1: Simulating a non-deterministic Turing machine

Here, the first tape consists of the input, and we have two extra tapes. The second tape includes i in unary, i.e. the content of the tape is 1^i. The third tape keeps track of the direction we go to for each level of the tree. That is, if $d_1 = 5$, then we take the 5th child of the first level, and if $d_2 = 1$, then we take the 1st child of the second level, etc. Each $d_k \in \{1, \ldots, b\}$, with b defined as the maximum number of children of any node in the computation tree.

We can loop through all possible i (i.e. we keep incrementing without bound), and then we can loop through all possible strings in the third tape to go through all possible paths through the tree. If we find an accepting state, then we can halt and accept; otherwise, the TM will never halt, and we'd reject anyways.

So far, we have four variants of the Turing machine; one where we can stay put, one where we have a two-sided infinite tape, one where we have multiple tapes, and one where we have non-determinism.

This gives us a lot of leeway in determining whether a language can be accepted by a Turing machine. Further, it can help with constructing Turing machines for various combinations of languages (much like how we proved the union of regular languages with NFAs).

12.2 Turing Decidability

Recall that a Turing machine on a given input can either accept, reject, or not halt.

Definition 12.1: Recognized by a Turing machine

A language $L \subseteq \Sigma^*$ is *recognized* by a Turing machine M if $L = \{x \in \Sigma^* \mid M \text{ accepts on input } x\}$.

Definition 12.2: Decided by a Turing machine

A language $L \subseteq \Sigma^*$ is *decided* by a Turing machine M if it is recognized by M, and $\forall y \in \Sigma^*$, M either accepts or rejects y. That is, M halts on every input.

So far, we have the following hierarchy of languages:

$$\begin{gathered}\{\text{Regular languages}\} \\ \cap \\ \{\text{Context-free languages}\} \\ \cap \\ \{\text{Turing-decidable languages}\} \\ \cap \\ \{\text{Turing-recognizable languages}\} \\ \cap \\ \{\text{All languages}\}\end{gathered}$$

We can show that all of these containments are strict:

- The language $\{0^n 1^n \mid n \geq 0\}$ is context-free but not regular.
- The language $\{0^n 1^n 2^n \mid n \geq 0\}$ is Turing-decidable but not context-free.

  Another example that we already saw is the language $\{w\#w \mid w \in \Sigma^*\}$, which is Turing-decidable but not context-free.
- Turing-decidable and Turing-recognizable languages are not equal; we'll see why this is the case next time.
- Not all languages can be recognized by a Turing machine—the set of Turing-recognizable languages is countable, but the set of all languages is uncountable.

  In particular, each Turing-recognizable language can be described by a finite-length string—this enumerates all possible Turing-recognizable languages. However, the set of all possible languages is the powerset of Σ^*, i.e. $\mathcal{P}(\Sigma^*)$, which is uncountable (it has cardinality $2^{\aleph_0} > \aleph_0$).

  We'll use a similar concept of countability to show that certain problems cannot be decided by Turing machines.

For terminology, Turing-recognizable languages are also called *recursively enumerable languages*, and Turing-decidable languages are also called *recursive languages*.

Next time, we'll prove the following theorem:

Theorem 12.3: Turing's Theorem

Let A_{TM} be the set of tuples $(\langle M \rangle, x)$ such that $\langle M \rangle$ is the encoding of a TM, and $x \in \Sigma^*$ is a string such that $M(x)$ halts and accepts.

Then, A_{TM} is recognizable but undecidable.

In particular, with this theorem, we know that there does exist a language that is Turing-recognizable but not Turing-decidable.

12.3 Recognizability

Definition 12.4: Co-recognizability

A language L is *co-recognized* if there exists a Turing machine M such that:

- If $x \notin L$, then $M(x)$ accepts
- If $x \in L$, then either $M(x)$ rejects or M does not halt.

Intuitively, saying that L is co-recognized by M is exactly the same as saying that $\overline{L}$ is recognized by M.

Proposition 12.5

If L is decidable, then L is co-recognizable.

Proof. Since L is decidable, so there is a Turing machine M that always halts. To convert this into a Turing machine M' that co-recognizes L, we can just flip q_{acc} and q_{rej}.

With this transformation, whenever M accepts, M' will reject, and whenever M rejects, M' will accept. □

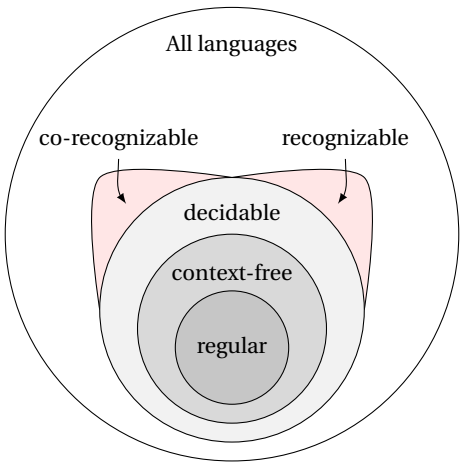
If a language is recognizable, is it necessarily also co-recognizable?

With the naive attempt, we can again swap q_{acc} and q_{rej}. Let M be the original Turing machine, and let $\overline{M}$ be the Turing machine with the swapped q_{acc} and q_{rej}.

If $x \in L$, then M accepts, and $\overline{M}$ rejects, which is perfectly fine. If $x \notin L$, we have two options; either M rejects or M never halts. In the first case, $\overline{M}$ will accept, which is perfectly fine. However, if M never halts, then $\overline{M}$ will also never halt, which is a problem.

We'll show later that the set of recognizable languages and co-recognizable languages are not the same; there exists some recognizable language that is not co-recognizable.

The Venn-diagram of languages now looks like this:



A next question is: does there exist any languages that are recognizable, co-recognizable, but not decidable? We'll show next that there does not exist any such languages; if a language is recognizable and co-recognizable, then it is decidable.

Theorem 12.6

L is decidable if and only if L is recognizable and co-recognizable.

Proof. We've already showed one direction; if L is decidable, then it is recognizable and it is also co-recognizable.

In the other direction, suppose L is recognizable by some TM M_1, and suppose L is co-recognizable by some TM M_2.

There are two cases:

- If $x \in L$, then $M_1(x)$ halts and accepts, and $M_2(x)$ either rejects or does not halt.
- If $x \notin L$, then $M_2(x)$ halts and accepts, and $M_1(x)$ either rejects or does not halt.

Notice that in both cases, one of the machines will halt and accept. How do we utilize this and combine M_1 and M_2?

A naive attempt is to first run $M_1(x)$, and then run $M_2(x)$. The issue here is that it is possible for $M_1(x)$ to run forever, never getting to run $M_2(x)$.

The solution is to run M_1 and M_2 in parallel. How do we do this? We can use a multitape machine. The first thing we do is copy the input to a second tape, and then move both pointers to the left-most position.

We now have the same input on both tapes—we can run M_1 on the first tape, and run M_2 on the second tape.

Here, we know that at least one of the machines will halt and accept. If M_1 halts and accepts, then the combined machine should halt and accept. If M_2 halts and accepts, then the combined machine should halt and reject. We can refine this a little further by taking rejections into consideration, but only considering acceptances is also fine (as one machine must accept).

Since this combined Turing machine always halts, L is decidable. □

Corollary 12.7

A_{TM} is not co-recognizable.

Proof. We know that by Theorem 12.3, A_{TM} is recognizable but not decidable.

If for contradiction we assume that A_{TM} is co-recognizable, then by Theorem 12.6 A_{TM} must be decidable, which we know is not true.

This means that A_{TM} must not be co-recognizable. □

12.4 Enumerability

Let us look more into why we also call Turing-recognizable languages *recursively enumerable*.

Here, we consider a slightly modified version of a Turing machine; in particular, we consider a Turing machine that also has a printer attached.

Definition 12.8: Enumerable

A language L is *enumerable* if there exists a Turing machine with a printer that prints $w_1\_w_2\_w_3\_\ldots$ with a blank input, such that:

- Each $w_i \in L$

- For each $x \in L$, some $w_i = x$.

Note that such Turing machines that print languages will usually print forever.

Theorem 12.9

A language L is Turing-recognizable if and only if L is enumerable.

Proof. Suppose L is Turing-recognizable; we want to show that L is also enumerable.

Here, we can enumerate over all possible strings, and run the machine for each string to determine whether any given string is accepted by the Turing machine.

However, we need to be careful on inputs that the machine never halts on. To resolve this issue, we can keep a counter i, and for each $i = 1, 2, \ldots$, we run M on all inputs of length $\leq i$ for $\leq i$ steps.

If the machine halts and accepts, then we output this string. Otherwise, we don't output it. We know that for any string, if M accepts the string, then for some i that is large enough, we will always be able to output this string, since each loop runs in finite time.

Further, notice that we're actually printing each string infinitely many times (i.e. if it's printed with at $i = k$, then it's also printed with $i = k + 1$, etc.)

In the other direction, suppose L is enumerable; we want to show that L is Turing-recognizable.

Here, we can simulate the Turing machine with a printer; whenever we print a string, we check whether it is equal to the input string. If it matches, then we accept and halt, and otherwise we keep trying and checking.

Interestingly, the resulting Turing machine will always halt on inputs that are accepted, and will never halt on inputs that are not accepted (i.e. we'll never halt on a string not in the language). □

10/11/2022

Lecture 13

Undecidable Problems, Diagonalization

Recall that we have the five sets of languages:

- $\mathsf{Reg} = \{L \subseteq \{0,1\}^* \mid L \text{ is regular}\}$
- $\mathsf{CF} = \{L \subseteq \{0,1\}^* \mid L \text{ is context-free}\}$
- $\mathsf{R} = \{L \subseteq \{0,1\}^* \mid L \text{ is decided by some Turing machine}\}$
- $\mathsf{RE} = \{L \subseteq \{0,1\}^* \mid L \text{ is recognized by some Turing machine}\}$
- $\mathsf{ALL} = \{L \subseteq \{0,1\}^*\}$

Today, the main theorem that we will prove is Turing's theorem (Theorem 12.3).

13.1 Cardinality

As a warm-up, let us review some concepts on cardinalities. In particular, suppose A and B are two sets.

We say that $|A| \leq |B|$ if and only if there exists a function $f : B \to A$ that is onto. Recall also that $|A| \leq |B|$ if and only if there exists a function $f : A \to B$ that is one-to-one.

Further, we say that $|A| = |B|$ if and only if $|A| \leq |B|$ and $|B| \leq |A|$. Equivalently, $|A| = |B|$ if and only if there exists a bijection $f : A \to B$.

Using these facts, we can prove that the integers, the rationals, and the natural numbers all have the same cardinality. However, how do we prove that something is *not* countable?

Theorem 13.1: Reals are uncountable

$|\mathbb{R}| \neq |\mathbb{N}|$

Proof. Suppose for contradiction that $|\mathbb{R}| = |\mathbb{N}|$.

Then, there exists a function $\mathbb{N} \to \mathbb{R}$ that is onto. Take $g : \mathbb{N} \to [0,1]$ defined by

$$g(n) = \begin{cases} f(n) & 0 \le f(n) \le 1 \\ 1 & 1 < f(n) \\ 0 & 0 > f(n) \end{cases}.$$

In particular, we also know that g is onto.

This means that we can write out a table where each row is a value of $g(n)$ for a value of $n \in \mathbb{N}$.

$$\begin{array}{lcl} g(0) & = & 0\ .\ \textcircled{5}2\ 1\ 4\ 9\ 3\ 5\ 6\ \cdots \\ g(1) & = & 0\ .\ 1\ \textcircled{4}1\ 6\ 2\ 9\ 8\ 5\ \cdots \\ g(2) & = & 0\ .\ 9\ 4\ \textcircled{7}8\ 2\ 7\ 1\ 2\ \cdots \\ g(3) & = & 0\ .\ 5\ 3\ 0\ \textcircled{9}8\ 1\ 7\ 5\ \cdots \\ \vdots & & \vdots \end{array}$$

However, if we take the diagonal digits d_i of this table, and change it to $d_i + 2 \pmod{10}$, we will get a new decimal number different from all of the other numbers in the table, namely

$$g(?) = 0.7691\ldots.$$

This number isn't in the table, so our table isn't complete; there will always be some other decimal number not listed in the table. This means that we can never fully list out all possible reals in $[0,1]$, which is a contradiction.

In particular, this shows that the reals in $[0,1]$ are uncountable, and thus the set of all reals is also uncountable. □

Theorem 13.2

Some languages are not Turing-recognizable.

Proof. Firstly, we claim that RE is countable. This is because each language in RE can be defined with a Turing machine that recognizes it. This description of such a Turing machine is a finite description (i.e. it's a finite length string over a finite alphabet).

More formally, we show that Σ^* is countable for any finite Σ; WLOG suppose $\Sigma = \{0, 1, 2, \ldots, |\Sigma| - 1\}$. We can explicitly define a one-to-one map from Σ^* to $\mathbb{N}$:

$$f(w_1 w_2 \cdots w_n) = w_1 + w_2 \cdot |\Sigma| + \cdots + w_n |\Sigma|^{n-1} + |\Sigma|^n.$$

Any Turing machine has a description $(Q, \Sigma, \Gamma, \delta, q_0, q_{acc}, q_{rej})$; this description can be encoded with {"0", "1", ",", "(", ")"}.

In particular, Q can be described with a list of binary strings representing each of the states. Σ and Γ can both be described also with a list of binary strings representing each symbol in the finite alphabet (it only

matters what the length of Σ is, as the symbols themselves are arbitrary). The δ function is just a table, so we can write it as a tuple of pairs, where each pair has the input and the output. q_0, q_{acc}, q_{rej} are just binary strings.

We can even take this further by taking this entire encoding and just turn it into binary, and we'd be able to encode it with just 0's and 1's.

The point here though is that we can always encode any finite description of a Turing machine, and this shows that the set of all Turing machines is countable, meaning RE is countable—each language can be mapped to a description of a Turing machine.

Further, since $|\mathsf{ALL}| = |\mathscr{P}(\Sigma^*)| = 2^{\aleph_0} > \aleph_0 = |\mathsf{RE}|$.

To be complete, what we're essentially doing is listing out a table where each column corresponds to an input $x_i \in \Sigma^*$, and each row contains the output of a Turing machine M_j for the given input.

	x_1	x_2	x_3	x_4	$\cdots$
M_1	<u>acc</u>	acc	acc	acc	acc
M_2	acc	<u>rej</u>	acc	rej	acc
M_3	acc	acc	<u>rej</u>	rej	acc
M_4	acc	rej	acc	<u>loop</u>	loop
M_5	rej	acc	loop	loop	<u>loop</u>
$\vdots$					

Looking at the diagonal, we can define a new language L such that $x_i \in L$ if and only if M_i rejects or loops (i.e. we're basically flipping the result on the diagonal). □

Now, we will prove Turing's theorem.

We first show that A_{TM} is undecidable.

Proof. Suppose for contradiction that A_{TM} is decidable. This means that there is a Turing machine H that decides A_{TM}.

In particular, if M accepts x, then $H((\langle M\rangle, x))$ halts and accepts; if M rejects or doesn't halt on x, then $H((\langle M\rangle, x))$ halts and rejects.

Let us define a new Turing machine D, which takes in an input $\langle M\rangle$, we run $H((\langle M\rangle, \langle M\rangle))$ and invert the output.

Looking at this further, if M accepts $\langle M\rangle$, then $H((\langle M\rangle, \langle M\rangle))$ should accept, which means $D(\langle M\rangle)$ should reject. If M doesn't accept $\langle M\rangle$, then $H((\langle M\rangle, \langle M\rangle))$ should reject, which means that $D(\langle M\rangle)$ should accept.

However, what does $D(\langle D\rangle)$ give us?

If D accepts $\langle D\rangle$, then $H((\langle D\rangle, \langle D\rangle))$ should accept, meaning $D(\langle D\rangle)$ should reject, which is a contradiction.

If D rejects $\langle D\rangle$, then $H((\langle D\rangle, \langle D\rangle))$ should reject, meaning $D(\langle D\rangle)$ should accept, which is also a contradiction.

This means that there is no possible output for $D(\langle D\rangle)$; D cannot exist, so H cannot exist, which means that A_{TM} is not decidable.

We can also see this as D being the Turing machine that computes the opposite of the diagonal of the table below; however, this language is not in the list of Turing machines, which gives a contradiction.

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\cdots$
M_1	<u>acc</u>	acc	acc	acc	acc
M_2	acc	<u>rej</u>	acc	rej	acc
M_3	acc	acc	<u>rej</u>	rej	acc
M_4	acc	rej	acc	<u>loop</u>	loop
M_5	rej	acc	loop	loop	<u>loop</u>
$\vdots$					

This language for the opposite of the diagonal is also called the *Russell language*

$$L_{Russell} = \{\langle M \rangle \mid M \text{ doesn't accept } \langle M \rangle\}.$$

□

13.2 Universal Turing Machines

Since any Turing machine can be encoded over some fixed alphabet, we can also have one universal Turing machine denoted U, where given input $(\langle M \rangle, x)$ simulates M on x.

Suppose we have multiple tapes; the first thing we do is copy x to a new tape, and copy q_0 to a third tape.

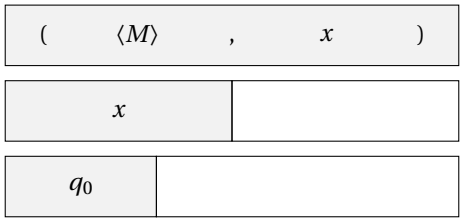


Figure 13.1: Setup for the universal Turing machine

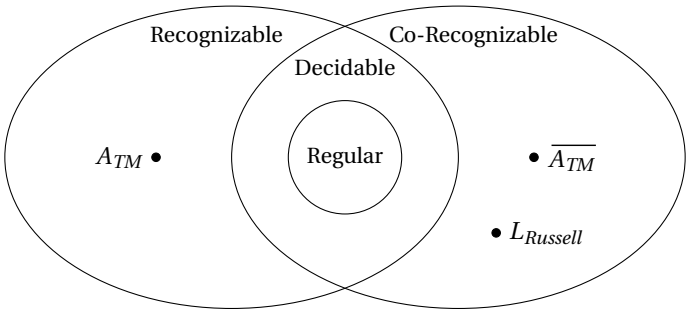
To simulate this Turing machine, we can repeat the following:

- Read the current symbol from x's tape; this is a.
- Find the relevant entry in M's δ-table that matches the current q and a
- Update the state in tape 3, write a symbol in tape 2, and move head accordingly.
- If $q = q_{acc}$ or q_{rej}, then halt and accept or reject accordingly.

In particular, U is useful, because U is a machine that recognizes A_{TM}. In particular, suppose we look at U on $(\langle M \rangle, x)$.

If M halts and accepts x, then U accepts $(\langle M \rangle, x)$. If M halts and rejects x, then U rejects $(\langle M \rangle, x)$. If M doesn't halt on x, then U doesn't halt, but that's okay, since we aren't accepting.

A corollary of Turing's theorem is that A_{TM} is not co-recognizable, and a second corollary is that $\overline{A_{TM}}$ is not recognizable.



What about a language that is neither recognizable, nor co-recognizable? (i.e. something outside of the Venn diagram?)

Consider the language $L = \{0 \cdot y \mid y \in A_{TM}\} \cup \{1 \cdot z \mid z \in \overline{A_{TM}}\}$.

This language is not recognizable, because we'd be able to solve $\overline{A_{Tm}}$ by prepending a 1. This language is also not co-recognizable, because we'd be able to solve A_{TM} by prepending a 0.

10/13/2022

Lecture 14

Reductions

We've shown prior that the language A_{TM} is undecidable but recognizable, and thus as a corollary $\overline{A_{TM}}$ is not recognizable. This leads us to another question: what other languages are undecidable or unrecognizable?

Since the set of recognizable languages (RE) is countable, and the set of all possible languages is uncountable, it turns out that *most* languages are unrecognizable.

Today, we'll show how we can prove that other languages are undecidable.

14.1 Reductions

Definition 14.1: Reduction

A reduces to B, denoted $A \leq B$, if given a decider for B, we can construct a decider for A.

In particular, we can use the decider for B as a black box in a subroutine for a decider for A; using these results, with some pre-processing and post-processing, we want to construct a decider for A.

As a result, if B is decidable, then A is decidable. In the contrapositive, if A is undecidable, then B is undecidable. (Note that this definition is informal; we'll look at a more formal definition next time.)

In a way, this means that reductions give us a relation between the "hardness" of two problems. If there is a reduction from A to B, then A is at least as hard as B.

Example 14.2: Halting Problem

Consider the halting problem; that is, the language $\text{HALT}_{TM} = \{(\langle M \rangle, w) \mid M \text{ is a TM that halts on input } w\}$.

We want to show that HALT_{TM} is undecidable.

Proof. Suppose for contradiction that HALT_{TM} is decidable. This means that there exists a decider R for HALT_{TM}. Our goal is to construct a decider S for A_{TM}.

For this decider, we have the following procedure for S on input $(\langle M \rangle, w)$:

- Run R on $(\langle M \rangle, w)$. There are two possibilities here: either R accepts, or R rejects.
- If R rejects, then we know that $M(w)$ does not halt, so we will reject as well.
- If R accepts, then we know that $M(w)$ halts in finite time. This means that we can just simulate $M(w)$, and return the same result. In other words, we're just running the universal machine $U(\langle M \rangle, w)$ and return the same result.

Formally, we can show that this construction decides A_{TM}.

For some input $(\langle M\rangle, w) \in A_{TM}$, then we know that M accepts w. As such, $R(\langle M\rangle, w)$ will accept, and $U(\langle M\rangle, w)$ will also accept, so overall, S will accept $(\langle M\rangle, w)$.

For some input $(\langle M\rangle, w) \notin A_{TM}$, then we have two cases: either M rejects w, or M loops forever. If M rejects, then $R(\langle M\rangle, w)$ will accept, but $U(\langle M\rangle, w)$ will reject, so overall, S will reject $(\langle M\rangle, w)$. If M loops forever, then $R(\langle M\rangle, w)$ will reject, so overall, S will also reject $(\langle M\rangle, w)$.

This shows that S is indeed a decider for HALT_{TM}; however, since we know that A_{TM} is undecidable, this is a contradiction; it must be the case that HALT_{TM} is undecidable as well. □

Now that we've shown that the halting problem is undecidable, we now have more examples to make reductions with; we can choose to make a reduction from A_{TM} or from HALT_{TM}.

Example 14.3

Consider the language $E_{TM} = \{\langle M\rangle \mid M \text{ is a TM and } L(M) = \varnothing\}$. That is, this is the language of Turing machines that reject all inputs.

We will show that E_{TM} is undecidable.

Proof. Suppose for contradiction that E_{TM} is decidable. This means that there exists a decider R for E_{TM}. We can construct a decider S for A_{TM}.

The idea here is that we can construct an intermediate Turing machine M', such that the language $L(M')$ depends only on whether or not $M(w) \in A_{TM}$, i.e. whether M accepts w. In particular, if M accepts w, we want $L(M')$ to be empty, and if M doesn't accept w, we want $L(M')$ to be nonempty.

As such, we have the following procedure for S on input $(\langle M\rangle, w)$:

- We can construct a description of a Turing machine M' as follows:
  - For M' on input x, we first check whether $x = w$.
  - If $x \neq w$, then we immediately reject.
  - If $x = w$, then we simulate M on w, and return that result.
- Return the opposite of the result of R on M'. That is, if $R(\langle M'\rangle)$ accepts, then we reject, and if $R(\langle M'\rangle)$ rejects, then we accept.

Now, we will prove why this construction decides A_{TM}.

For some input $(\langle M\rangle, w) \in A_{TM}$, then we know that M accepts w. Looking at M', for any run of $M'(x)$ for $x \neq w$, we know that M' will reject x. This means that we only care about $M'(w)$, which will end up simulating $M(w)$. Since $M(w)$ accepts, then $M'(w)$ will accept, and $L(M') = \{w\}$ is not empty. This means that $R(M')$ will reject, and overall, S will accept $(\langle M\rangle, w)$.

For some input $(\langle M\rangle, w) \notin A_{TM}$, then we know that M doesn't accept w. There are two cases; either M rejects w or M doesn't halt on w. In either case, M' will never accept w, so $L(M') = \varnothing$. This means that $R(\langle M'\rangle)$ will accept, and overall, S will reject $(\langle M\rangle, w)$.

This shows that S is indeed a decider for A_{TM}. However, since we know that A_{TM} is undecidable, this is a contradiction; it must be the case that E_{Tm} is undecidable as well.

Notice that we could also have done a reduction with M' on x that just ignores x and simulates $M(w)$ instead. In this case, if $M(w)$ accepts, then $L(M') = \Sigma^*$, and if $M(w)$ doesn't accept, then $L(M') = \varnothing$. The same logic holds from here. □

Example 14.4

Consider the language $\text{EQ}_{TM} = \{(\langle M_1 \rangle, \langle M_2 \rangle) \mid M_1 \text{ and } M_2 \text{ are TMs and } L(M_1) = L(M_2)\}$.

We will show that EQ_{TM} is undecidable.

Proof. Suppose for contradiction that EQ_{TM} is decidable. This means that there exists a decider R for EQ_{TM}. We can construct a decider S for E_{TM}.

We have the following procedure for S on input $(\langle M \rangle, w)$:

- We construct an intermediate M' that will always reject for any input.
- Run $R(\langle M \rangle, \langle M' \rangle)$, and return its output.

Analyzing this construction, we have two cases.

For some input $\langle M \rangle \in E_{TM}$, then $L(M) == L(M')$, which means that R will accept, so S accepts.

For some input $\langle M \rangle \notin E_{TM}$, then $L(M) \neq \varnothing$, so R will reject, and S rejects. □

Example 14.5

Consider the language $\text{REG}_{TM} = \{\langle M \rangle \mid M \text{ is a TM and } L(M) \text{ is regular}\}$.

We will show that REG_{TM} is undecidable.

Proof. Let's think about how we can reduce A_{TM} to REG_{TM}. We want a reduction such that:

- If M accepts w, then $L(M')$ is regular
- If M doesn't accept w, then $L(M')$ is non-regular.

Here, we need some regular language and some non-regular language; suppose we choose Σ^* to be the regular language, and $\{0^n 1^n \mid n \geq 0\}$ to be the non-regular language.

This leads us to our reduction.

Suppose for contradiction that REG_{TM} is decidable. This means that there exists a decider R for REG_{TM}. We want to construct a decider S for REG_{TM}.

We have the following procedure for S on input $(\langle M \rangle, w)$:

- We construct an intermediate M' on input x such that:
  - Check if $x = 0^n 1^n$. If so, we accept.
  - If not, then we run $M(w)$, and return its output.
- We then return the output of $R(\langle M' \rangle)$.

Here, if $(\langle M \rangle, w) \in A_{TM}$), then M accepts w, and M' accepts all inputs; this means $L(M') = \Sigma^*$ is regular, and R (and therefore S) accepts. If $(\langle M \rangle, w) \notin A_{TM}$, then M doesn't accept w, and M' only accepts $0^n 1^n$; this means that $L(M') = \{0^n 1^n \mid n \geq 0\}$ is non-regular, and R (and therefore S) rejects. □

Note that in all of these reductions, we don't run M' on some input x, we just write the *code* of M' and feed this code into a decider. We pass this code into some black box that will *guarantee* a correct output.

This leads us to the next big theorem, called Rice's theorem. Informally, it states that any non-trivial language of the following form is undecidable:

$$\{\langle M \rangle \mid M \text{ is a Turing machine and } L(M) \text{ satisfies } \ldots\}.$$

Formally, we must first define what a non-trivial language is, and what the "…" actually means.

Definition 14.6: Property of recognizable languages

P is a *property of recognizable languages* if $P \subseteq \{\langle M \rangle \mid M \text{ is a TM}\}$ and for any Turing machines M_1 and M_2, if $L(M_1) = L(M_2)$, then either both $\langle M_1 \rangle, \langle M_2 \rangle$ in P, or neither.

In particular, we must ensure that this property depends only on the *language* of the Turing machine, not on the Turing machine itself.

Definition 14.7: Non-trivial property

We say that a property P of recognizable languages is *non-trivial* if there exists an M_1 and M_2 such that $\langle M_1 \rangle \in P$ and $\langle M_2 \rangle \notin P$.

Now, the precise, rigorous statement of Rice's theorem as follows:

Theorem 14.8: Rice's Theorem

Let P be any non-trivial property of recognizable languages. Then P is undecidable.

Proof. Let P be a non-trivial property of recognizable languages. Suppose for contradiction that R decides P.

Let $M_\varnothing$ be a Turing machine that always rejects; in this case, $L(M_\varnothing) = \varnothing$. WLOG, suppose that $\langle M_\varnothing \rangle \notin P$. (If not, then we can just replace P with $\overline{P}$, which would also be decidable.)

Let M_{in} be a Turing machine such that $\langle M_{in} \rangle \in P$.

We'll show that if P is decidable, then so is A_{TM}. In particular, we want to construct an M' such that:

- If $(\langle M \rangle, w) \in A_{TM}$, then $L(M') = L(M_{in})$
- If $(\langle M \rangle, w) \notin A_{TM}$, then $L(M') = L(M)$

In particular, this intermediate M' on input x should do the following:

- Run $M(w)$; if it rejects, then reject.
- If $M(w)$ accepts, then run $M_{in}(x)$.

Here, we can see that this correctly decides A_{TM}.

If $(\langle M \rangle, w) \in A_{TM}$, then $M(w)$ accepts, and we run $M_{in}(x)$ for all inputs; this gives $L(M') = L(M_{in})$. This means that R accepts, which causes S to accept.

If $(\langle M \rangle, w) \notin A_{TM}$, then $M(w)$ doesn't accept, so M' never accepts, and $L(M') = L(M_\varnothing)$. This means that R rejects, which causes S to reject.

This shows that S decides A_{TM}; however, since we know that A_{TM} is undecidable, this is a contradiction. As such, it must be the case that P is undecidable. □

10/18/2022

Lecture 15

Mapping reductions

Today, we'll talk about mapping reductions, which are a more restricted type of reduction. Mapping reductions can prove that some languages are not only undecidable, but also not recognizable (or not co-recognizable).

For example, we've only shown that problems like E_{TM}, HALT_{TM}, EQ_{TM}, etc. are undecidable, but we don't know whether they are recognizable, co-recognizable, or neither. With mapping reductions, we are able to make this distinction; that is, if $A \leq_m B$ (i.e. we reduce A to B with a mapping reduction), then if B is recognizable, so is A (and similarly for co-recognizability).

Definition 15.1: Computable Mapping

A mapping $f : \Sigma^* \to \tilde{\Sigma}^*$ is computable if there exists a TM that always halts, and on input $x \in \Sigma^*$ the machine halts when the tape's content is $f(x)$.

Example 15.2

For example, the function $\texttt{reverse} : \{0,1\}^* \to \{0,1\}^*$ that reverses a string is computable, as we've shown in the homework.

Proposition 15.3

We claim that f is computable if and only if there exists a multi-tape Turing machine M such that:

1. M always halts
2. M has a designated output tape
3. M on input x eventually halts with the content of the output tape equaling $f(x)$.

Proof. We already know that we can simulate a multi-tape Turing machine with a standard Turing machine. After simulating the above multi-tape Turing machine M, we can just clear out all of the other symbols, replacing each symbol with the corresponding symbol in the output tape. □

Definition 15.4: Mapping Reducible

A language A over alphabet Σ is *mapping reducible* to a language B over alphabet $\tilde{\Sigma}$, written $A \leq_m B$, if there exists a computable function $f : \Sigma^* \to \tilde{\Sigma}^*$ such that for every $w \in \Sigma^*$,

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

f is called a (mapping) reduction from A to B.

A diagram for mapping reduction is shown in Fig. 15.1. Unlike the general framework we discussed last time, a mapping reduction only preprocesses the input, then passes $f(w)$ to the decider for B and returns its answer. (Note that we are not allowed any post-processing of B's answer.)

It should be clear that mapping reductions can still prove whether a language is decidable.

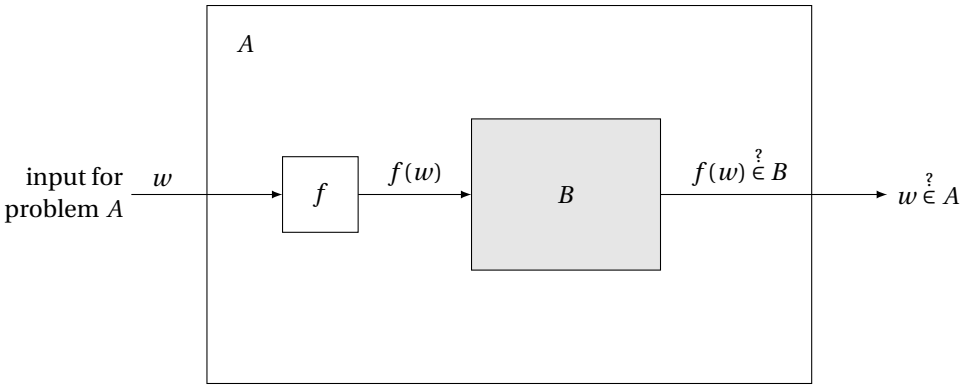


Figure 15.1: Pictorial representation of a mapping reduction

Theorem 15.5

If $A \leq_m B$ and B is decidable, then so is A.

In the contrapositive, if $A \leq_m B$, and A is undecidable, then so is B.

Let us now go through all the reductions we saw last time, and look at whether each are mapping reductions.

Example 15.6

Is the reduction in Example 14.2 for HALT_{TM} a mapping reduction?

No; the reduction is doing a lot more than just calling R; we do both pre-processing and post-processing.

Does a mapping reduction exist? Let us try to find a mapping reduction from A_{TM} to HALT_{TM}.

In particular, we want a function $f(\langle M \rangle, w) = (\langle M' \rangle, w')$ such that

$$(\langle M \rangle, w) \in A_{TM} \iff (\langle M' \rangle, w') \in \text{HALT}_{TM}.$$

Equivalently, we want to map a Turing machine M that accepts w to a Turing machine M' that halts on w'.

An idea is to change the machine M such that if M accepts, then M' halts, and if M rejects, then M' will not halt. Formally, we define M' on input x as:

- Run M on input x.
- If M accepts, then accept.
- If M rejects, then run forever.

Note that we don't need to consider the case where M runs forever on x; in this case, M wouldn't accept w, so we should be running forever anyways.

This transformation is our map $f(\langle M \rangle, w)$, outputting $(\langle M' \rangle, w)$.

We want to show that $(\langle M \rangle, w) \in A_{TM}$ if and only if $(\langle M' \rangle, w) \in \text{HALT}_{TM}$.

If $(\langle M \rangle, w) \in A_{TM}$, then we know that M accepts w, so M' will also accept (and thus halt) on w. This means $(\langle M' \rangle, w) \in \text{HALT}_{TM}$.

If $(\langle M \rangle, w) \notin A_{TM}$, then we know that M does not accept w. There are two cases; either M rejects on w, or M loops forever on w. In the first case, if M rejects w, then M' will run forever; in the second case, M' will still run forever, as we won't get past simulating M on w. This means that in both cases, M' will loop forever, so $(\langle M' \rangle, w) \notin \text{HALT}_{TM}$.

This means that we have constructed a mapping reduction from A_{TM} to HALT_{TM}.

Note that the reduction is just the function f, and not the entire process of taking the output and simulating it, etc.

Example 15.7

Is the reduction in Example 14.3 for E_{TM} a mapping reduction?

No; the reduction does some post-processing, which a mapping reduction cannot do.

It's hard to construct a mapping reduction from A_{TM} to E_{TM}, so we'll show instead a mapping reduction from A_{TM} to $\overline{E_{TM}}$.

In particular, we have the function f on input $(\langle M \rangle, w)$ that constructs and returns an M' such that:

- Check whether $x = w$.
- If not equal, reject.
- If equal, simulate M on x.

With this reduction, if $(\langle M \rangle, w) \in A_{TM}$, then $\langle M' \rangle \in \overline{E_{TM}}$.

A natural question follows: does there even exist a mapping reduction from A_{TM} to E_{TM}? It turns out that this is impossible. We'll show next why this is the case.

Proposition 15.8

If $A \leq_m B$, then $\overline{A} \leq_m \overline{B}$.

Proof. We can just use the same mapping reduction f in $A \leq_m B$ to show $\overline{A} \leq_m \overline{B}$. □

Theorem 15.9

If $A \leq_m B$, then the following holds:

- If B is recognizable, then A is recognizable.
- If B is co-recognizable, then A is co-recognizable.

Proof. First, we will show that if B is recognizable then A is also recognizable. Suppose B is recognized by some Turing machine M_B. We want to show that A is recognizable; that is, we want to show there exists an M_A that recognizes A.

In particular, we construct M_A that does the following:

- Compute $y = f(x)$ using the Turing machine for f (since f is computable, as we have a mapping reduction)
- Run M_B on y, and return the same

We claim that this construction of M_A correctly recognizes A.

Suppose $x \in A$. This means that $f(x) \in B$ since f is a mapping reduction from A to B. This means that M_B will accept $y = f(x)$, so M_A will also accept x.

Suppose $x \notin A$. This means that $f(x) \notin B$ since f is a mapping reduction from A to B. This means that M_B

will not accept $y = f(x)$, so M_A will also not accept x.

This means that M_A will always correctly accept or not accept an input x, so M_A does indeed recognizes A.

The second item follows easily from this analysis; that is, we'll now show that if B is co-recognizable, then A is also co-recognizable. In particular, if B is co-recognizable, then $\overline{B}$ is recognizable.

However, we know that if $A \leq_m B$, then $\overline{A} \leq_m \overline{B}$. We've shown previously that if $\overline{B}$ is recognizable, then $\overline{A}$ is also recognizable (i.e. with the same reduction), which then means that A is co-recognizable □

Corollary 15.10

We also have the contrapositive; if $A \leq_m B$, then the following hold:

- If A is *not* recognizable, then so is B.
- If A is *not* co-recognizable, then so is B.

Proposition 15.11

There exists no mapping reduction from A_{TM} to E_{TM}.

Proof. We first claim that E_{TM} is co-recognizable. That is, there exists a Turing machine R such that for any $\langle M \rangle \notin E_{TM}$, R accepts $\langle M \rangle$, and for any $\langle M \rangle \notin E_{TM}$, R doesn't accept $\langle M \rangle$.

Equivalently, this is the same as saying that if $L(M) \neq \varnothing$, then R accepts $\langle M \rangle$, and if $L(M) = \varnothing$, then R doesn't accept $\langle M \rangle$.

To construct R, we can just define it to run M on all inputs, dovetailing the simulations, until we find some input that is accepted. Once we find such an input, we can just accept; otherwise, we'll be running forever.

Since we now know that E_{TM} is co-recognizable, we now claim that there is no mapping reduction from A_{TM} to E_{TM}.

For contradiction, suppose there exists a mapping reduction from A_{TM} to E_{TM}. We know that E_{TM} is co-recognizable, and A_{TM} is recognizable. However, the previous theorem allows us to conclude that since E_{TM} is co-recognizable, and we have a mapping reduction from A_{Tm} to E_{TM}, then A_{TM} is also co-recognizable.

This means that A_{TM} is both recognizable *and* co-recognizable, so A_{TM} is decidable. This is a contradiction—we've shown that A_{Tm} is undecidable. □

More generally, if $A \in \mathsf{RE} \setminus \mathsf{R}$, and $B \in \mathsf{coRE} \setminus \mathsf{R}$, then there is no mapping reduction from A to B, and there is no mapping reduction from B to A.

Example 15.12

Is the reduction in Example 14.4 for EQ_{TM} a mapping reduction?

Yes; we're just doing some pre-processing, and returning the result.

Example 15.13

Is the reduction in Example 14.5 for REG_{TM} a mapping reduction?

Yes; we're just doing some pre-processing, and returning the result.

This means that we have the following results:

- $A_{TM} \leq_m \text{HALT}_{TM}$, which means $\text{HALT}_{TM} \notin \text{coRE}$
- $A_{TM} \leq_m \overline{E_{TM}}$, which means $E_{TM} \notin \text{RE}$
- $E_{TM} \leq_m \text{EQ}_{TM}$, which means $\text{EQ}_{TM} \notin \text{RE}$
- $A_{TM} \leq_m \text{REG}_{TM}$, which means $\text{REG}_{TM} \notin \text{coRE}$.

Theorem 15.14

EQ_{TM} is neither recognizable nor co-recognizable.

Proof. We've shown a mapping reduction from $\overline{A_{TM}} \leq_m E_{TM} \leq_m \text{EQ}_{TM}$, so EQ_{TM} is not recognizable (as $\overline{A_{TM}}$ is not recognizable).

To prove that EQ_{TM} is not co-recognizable, we want to show a mapping reduction from A_{TM} to EQ_{TM}.

In particular, we can construct a map f on input $(\langle M \rangle, w)$ that creates $(\langle M_1 \rangle, \langle M_2 \rangle)$ such that:

- M_1 on input x always accepts.
- M_2 on input x ignores x and simulates M on w and returns the same.

It follows that M accepts w if and only if $L(M_1) = L(M_2)$.

If $w \in M$, then M_2 will accept for all inputs, so $L(M_1) = L(M_2)$, and $(\langle M_1 \rangle, \langle M_2 \rangle) \in \text{EQ}_{TM}$, so we accept.

If $w \notin M$, then M_2 will also not accept for all inputs, so $L(M_1) \neq L(M_2)$, and $(\langle M_1 \rangle, \langle M_2 \rangle) \notin \text{EQ}_{TM}$, so we don't accept.

As such, we have a mapping reduction $A_{TM} \leq \text{EQ}_{TM}$, so EQ_{TM} is also not co-recognizable. □

With mapping reductions, we can now revisit Rice's theorem for recognizable languages:

Theorem 15.15: Rice's Theorem for recognizable languages

Let P be any non-trivial property of recognizable languages where $\langle M_\varnothing \rangle \notin P$. Then $A_{TM} \leq_m P$.

Proof. The same proof will give us a mapping reduction from $A_{TM} \leq_m P$. □

Corollary 15.16

If P is a non-trivial property of recognizable languages, then:

- if $\langle M_\varnothing \rangle \notin P$, then P is not co-recognizable.
- if $\langle M_\varnothing \rangle \in P$, then P is not recognizable.

Proof. The first item follows directly from the mapping reduction. For the second property, we can look at the complement of P, in which case $\langle M_\varnothing \rangle \notin \overline{P}$, so $\overline{P}$ is not co-recognizable, so P is not recognizable. □

10/20/2022

Lecture 16

Turing Reductions

Today, we'll be talking about Turing reductions, which are a more powerful kind of reduction, and is a good way to show that some language is undecidable. In particular, mapping reductions are quite restricted; we are only allowed

pre-processing, and we do not allow multiple calls to the inner function.

With Turing reductions, we have more relaxed constraints, but at the cost of not preserving recognizability.

16.1 Oracle Machines

Let B be any language. An oracle to B is an "external device" that is capable of reporting whether or not any string w is in B.

An *oracle Turing machine* (with oracle B) is a modified Turing machine that has the additional capability of querying the oracle B.

More concretely, we'll have a designated "oracle query tape" where we will write queries to the oracle. Once the query is ready, we transition to a unique state q_{query}. If y is the content of the oracle tape, we are guaranteed that:

- If $y \in B$, then the next state will be q_{yes}
- If $y \notin B$, then the next state will be q_{no}.

16.2 Turing Reductions

Definition 16.1: Turing Reducible

A language A is *Turing reducible* to a language B, written $A \leq_T B$ if there exists an oracle Turing machine M^B that decides A. That is, we can decide A given query access to B.

Example 16.2

In a trivial case, given oracle access to B, we can always decide B. We'd just query the input, and return the oracle's output.

Similarly, we can also decide $\overline{B}$; we'd just query the input, and return the opposite of the oracle's output. Notice that this is different from mapping reductions, in that we're allowed this post-processing.

Theorem 16.3

If $A \leq_T B$ and B is decidable, then A is decidable.

In the contrapositive, if $A \leq_T B$ and A is undecidable, then B is also undecidable.

Proof. Since $A \leq_T B$, there exists some oracle Turing machine M^B that decides A utilizing an oracle to B. Here, we're also assuming that B is decidable—this means that there exists a Turing machine that decides B.

To create a standard Turing machine that decides A, for every oracle call to B, replace it with a simulation of a decider for B. Intuitively, we now have an actual implementation of the "black box" oracle we used for M^B, so we can just substitute that in. □

Notice that if $A \leq_T B$ and B is recognizable, this does *not* mean that A is also recognizable.

In particular, we claim that $A_{TM} \leq_T \overline{A_{TM}}$ (more generally, for any language L, $L \leq_T \overline{L}$). Specifically, we can just call the oracle to $\overline{A_{TM}}$ and flip the output to decide A_{TM}.

Since $A_{TM} \leq_T \overline{A_{TM}}$, Turing reductions must not preserve recognizability, as here we'd have shown that $\overline{A_{TM}}$ is also recognizable, which we know is not true.

Example 16.4

We will show that $E_{TM} \leq_T A_{TM}$. (Notice that we've shown that $\overline{A_{TM}} \leq_m E_{TM}$, so there is no corresponding mapping reduction.)

In particular, we want to construct an oracle Turing machine $S^{A_{TM}}$ that decides E_{TM}; that is, $S^{A_{TM}}$ should decide whether a given input program $\langle M \rangle$ decides whether $L(M)$ is empty.

Here, we prepare a Turing machine N on input x such that:

- Ignore x and simulate M on all inputs in Σ^* in parallel

  Note that we've shown how to run inputs in parallel; we can simulate the first i inputs for the first i steps, and increment i as we continue. This way, we'll eventually get to the end of some program that accepts, and otherwise we'd loop forever.

- If M accepts one of them, then accept

Once we have N, we can query A_{TM} on $(\langle N \rangle, \varepsilon)$. If this accepts, then we reject,a nd if this rejects, then we accept.

If $\langle M \rangle \in E_{TM}$, then $L(M) = \varnothing$, so N will not accept ε (it'll never find an accepting input). This means that A_{TM} will reject $(\langle N \rangle, \varepsilon)$, so $S^{A_{TM}}$ accepts.

If $\langle M \rangle \notin E_{TM}$, then $L(M) \neq \varnothing$, so N will accept ε (it'll be able to find an accepting input). This means that A_{TM} will accept $(\langle N \rangle, \varepsilon)$, so $S^{A_{TM}}$ rejects.

This means that with an oracle to A_{TM}, we are able to decide E_{TM}, so E_{TM} is Turing reducible to A_{TM}; $E_{TM} \leq_T A_{TM}$.

As a side note, this is also a mapping reduction from E_{TM} to $\overline{A_{TM}}$; we'd only be doing pre-processing if given a machine for $\overline{A_{TM}}$.

Example 16.5

Show that $\overline{\text{EQ}_{TM}}$ is recognizable by an oracle TM with oracle access to A_{TM}.

Although this is not a Turing reduction (as we are only recognizing $\overline{\text{EQ}_{TM}}$), this is actually quite interesting, as EQ_{TM} and $\overline{\text{EQ}_{TM}}$ are neither recognizable nor co-recognizable.

Recall that

$$\text{EQ}_{TM} = \{(\langle M_1 \rangle, \langle M_2 \rangle) \mid M_1 \text{ and } M_2 \text{ are TMs, and } L(M_1) = L(M_2)\}.$$

This means that $\overline{\text{EQ}_{TM}}$ contains w such that either w is not of the right format, or $w = (\langle M_1 \rangle, \langle M_2 \rangle)$ for TMs M_1, M_2, with $L(M_1) \neq L(M_2)$.

That is, we want to construct $M^{A_{TM}}$ on input $(\langle M_1 \rangle, \langle M_2 \rangle)$ with the following procedure:

- Check that $\langle M_1 \rangle$ and $\langle M_2 \rangle$ are legal encodings; if not, accept
- For $i = 1, 2, \ldots$:
  - Let x be the ith string in lexicographical order
  - Query $A_{TM}(\langle M_1 \rangle, x)$ and $A_{TM}(\langle M_2 \rangle, x)$
  - If answers differ, then accept

Note that this is technically not a Turing reduction; we aren't able to decide $\overline{\text{EQ}_{TM}}$, but this still gives us something interesting.

Here, if $(\langle M_1 \rangle, \langle N_2 \rangle) \in \overline{\text{EQ}_{TM}}$, then $L(M_1) = L(M_2)$, so we'll never be able to find an input that gives different inputs.

So far, we've really only been talking about very theoretical self-referential problems, none of which are combinatorial in nature. We'll talk next about a combinatorial problem that is also undecidable.

16.3 Post Correspondence Problem

In the post correspondence problem, we have a collection of domino tiles, for example

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

The goal is to decide if there is a "match", i.e. a sequence of tiles, with repetitions, such that the top string is equal to the bottom string after concatenation.

For example, we can have

$$\frac{a}{ab}\frac{b}{ca}\frac{ca}{a}\frac{a}{ab}\frac{abc}{c} \implies \frac{abcaaabc}{abcaaabc}.$$

Example 16.6

Consider the following set of tiles:

$$\left\{\frac{ab}{abab}, \frac{b}{a}, \frac{aba}{b}, \frac{aa}{a}\right\}.$$

Is there a match?

Notice that we cannot start with $\frac{b}{a}$ or with $\frac{aba}{b}$, as the strings will not start with the same character. This means that we can only start with either $\frac{ab}{abab}$ or $\frac{aa}{a}$.

Suppose we start with $\frac{ab}{abab}$. The next tile should have a top string that starts with an a:

- We can't have $\frac{b}{a}$ next, as the top string doesn't match anymore
- We can't have $\frac{aba}{b}$, as now the top is $ababa$, which does not match with the bottom of $ababb$.
- We can't have $\frac{aa}{a}$, as now the top is $abaa$, which does not match with the bottom of $ababa$.

This means that the only choice we have left is the same tile $\frac{ab}{abab}$, and we'd never get two strings of the same length if we continue.

This means that we must start with $\frac{aa}{a}$.

- We can't have $\frac{aba}{b}$ next, as the top is $aaaba$ with the bottom being ab, which does not match
- We can't have $\frac{ab}{abab}$, as the top is $aaab$ with the bottom being $aabab$, which does not match

This leaves $\frac{b}{a}$, which can match; this could give something, but we'd eventually get to

$$\frac{aa}{a}\frac{aa}{a}\frac{b}{a}\frac{ab}{abab}.$$

We can see that this problem is actually quite hard. We'll show that the post correspondence problem (PCP) is actually undecidable.

To show this, we'll show how to reduce A_{TM} to PCP.

The idea here is to come up with a sequence of tiles, such that M accepts w if and only if there is a match. This match would be the sequence of configurations in M's execution on w.

Example 16.7

For example, the simulation of $L = \{w \mid w \text{ has an odd number of 1's}\}$ would be represented as

$$\#q_0 01101 \# X q_0 1101 \# XX q_1 101 \# \cdots \# XXXXXq_1 \# XXXXXq_{acc} \#.$$

That is, we separate execution steps with # symbols, and we write the state's label immediately before the location of the head of the Turing machine at that instant.

To construct the tiles in this example, we'd have the initial state $\frac{\#}{\#q_0 01101\#}$, and the next tiles are the valid transitions:

Transitions	$\frac{q_0 0}{X q_0}, \frac{q_0 1}{X q_1}, \frac{q_1 0}{X q_1}, \frac{q_1 1}{X q_0}, \frac{q_1 \#}{q_{acc} \#}$
Copy	$\frac{1}{1}, \frac{0}{0}, \frac{X}{X}, \frac{\#}{\#}$
Clean up	$\frac{X q_{acc}}{q_{acc}}$
Finish	$\frac{q_{acc} \#}{\varepsilon}$

One thing to consider though is that we could've just used all the copy tiles, but if we assert that we start with the initial state, the only match is the one that goes through the execution history.

Here is the execution of the Turing machine simulated by PCP, on the input $w = 010$ (aligned by character, not by tile, to make things a little easier to follow).

#	q_0	0	1	0	#	X	q_0	1	0	#	X	X	q_1	0	#	X	X	X	q_1	#	X	X	X	q_a	#	X	X	q_a	#	X	q_a	#	q_a	#					
#	q_0	0	1	0	#	X	q_0	1	0	#	X	X	q_1	0	#	X	X	X	q_1	#	X	X	X	q_a	#	X	X	q_a	#	X	q_a	#	q_a	#					

More generally, given $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{acc}, q_{rej})$ and $w = w_1 \cdots w_n$, we produce the following tiles:

- Type 1: $\frac{\#}{\#q_0 w_1 w_2 \cdots w_n \#}$
- Type 2: (move left) If $\delta(p, a) = (q, b, R)$, add $\frac{pa}{bq}$
- Type 3: (move right) If $\delta(p, a) = (q, b, L)$, add $\frac{\sigma pa}{q, \sigma b}$ for all $\sigma \in \Gamma$
- Type 4: (copy) add $\frac{\sigma}{\sigma}$ for all $\sigma \in \Gamma$
- Type 5: (end execution) add $\frac{\#}{\#}$ and $\frac{\#}{\_\#}$
- Type 6: (clean up) Add $\frac{a q_{acc}}{q_{acc}}$ and $\frac{q_{acc} a}{q_{acc}}$ for all $a \in \Gamma$
- Type 7: (finish) Add $\frac{q_{acc} \#}{\varepsilon}$

To solve the earlier issue that we could just use all the copy tiles, let us define a new, slightly different game, called MPCP: given a collection of tiles with a *designated left tile*, find a sequence of tiles that forms a match starting with the designated tile.

Theorem 16.8

M accepts w if and only if the tiles we produced with designated tile $\dfrac{\#}{q_0 w_1 \cdots w_n \#}$.

Proof (sketch). If M accepts w, by construction we can produce a sequence of tiles that has a match.

In the other direction, we can show by induction on the length of the partially matched sequence of tiles that they form a legal sequence of configurations of M on w. □

Next, we can show a mapping reduction from MPCP to PCP; that is, given $P = (\frac{t_1}{b_1}, \frac{t_2}{b_2}, \ldots, \frac{t_k}{b_k})$, how can we force the matches to start with $\frac{t_1}{b_1}$?

The trick is to define the following operations on strings:

- $*u = *u_1 * u_2 \cdots * u_n$
- $u* = u_1 * u_2 * \cdots * u_n*$
- $*u* = *u_1 * u_2 * \cdots * u_n*$

We then transform the tiles P into P' such that

$$P' = \left(\frac{*t_1}{*b_1*}, \frac{*t_1}{b_1*}, \frac{*t_2}{b_2*}, \ldots, \frac{*t_k}{b_k*}, \frac{*\diamond}{\diamond}\right).$$

To show that if $P \in$ MPCP, then $P' \in$ PCP, notice that we can just add stars in between each tile in the matching for MPCP to get a valid matching for P' in PCP.

To show that if $P' \in$ PCP, then $P \in$ MPCP, notice that the first tile must be $\frac{*t_1}{*b_1*}$, as the first symbol must match. After we put this tile down, it's equivalent to the MPCP instance without the stars.

10/25/2022

Lecture 17

Hierarchy of Undecidability, Complexity Theory

A natural question to ask is: how powerful are oracle TMs? It turns out that for any oracle B, there are languages that cannot be decided by any TM with oracle access to B.

Theorem 17.1

Consider

$$A'_{TM} = \{(\langle M \rangle, w) \mid M \text{ is an oracle TM with oracle access to } A_{TM} \text{ and } M^{A_{TM}} \text{ accepts } w\}.$$

A'_{TM} cannot be decided by an oracle TM with oracle access to A_{TM}.

Proof. Similar to the proof that A_{TM} is undecidable, we'll use a diagonalization technique.

Suppose for contradiction that A'_{TM} can be decided by some oracle TM $H^{A_{TM}}$. In particular, this means that

- if $(\langle M \rangle, w) \in A'_{TM}$, then $H^{A_{TM}}$ accepts $(\langle M \rangle, w)$
- if $(\langle M \rangle, w) \notin A'_{TM}$, then $H^{A_{TM}}$ rejects $(\langle M \rangle, w)$

We can define an oracle TM $D^{A_{TM}}$ where on input $\langle M \rangle$, runs $H^{A_{TM}}$ on $(\langle M \rangle, \langle M \rangle)$ and returns the opposite.

In particular, this means that

- if $M^{A_{TM}}$ accepts $\langle M \rangle$, then $H^{A_{TM}}$ accepts $(\langle M \rangle, \langle M \rangle)$, and thus $D^{A_{TM}}$ rejects $\langle M \rangle$
- if $M^{A_{TM}}$ doesn't accept $\langle M \rangle$, then $H^{A_{TM}}$ rejects $(\langle M \rangle, \langle M \rangle)$, and thus $D^{A_{TM}}$ accepts $\langle M \rangle$

Does $D^{A_{TM}}$ accept $\langle D \rangle$?

If $D^{A_{TM}}$ accepts $\langle D \rangle$, then $D^{A_{TM}}$ rejects $\langle D \rangle$ by the above reasoning—this is a contradiction. Similarly, if $D^{A_{TM}}$ doesn't accept $\langle D \rangle$, then $D^{A_{TM}}$ accepts $\langle D \rangle$, which is also a contradiction.

This means that A'_{TM} can't be decided by any oracle TM. □

We can prove further that there is a hierarchy; if we define

$$A''_{TM} = \{(\langle M \rangle, w) \mid M \text{ is an oracle TM with oracle access to } A'_{TM} \text{ that accepts } w\},$$

we can show that $A''_{TM} \not\leq_T A'_{TM}$.

This means that we have the hierarchy

$$A_{TM} < A'_{TM} < A''_{TM} < A'''_{TM} < \cdots,$$

in order of "difficulty".

17.1 Computational Complexity Theory: Time Complexity

So far in this course, we discussed

- What can/cannot be computed with finite memory?
- What can/cannot be decided on a general computer (Turing machine)?

Computational complexity theory studies "what can/cannot be computed efficiently?"

Here, "efficiently" means with limit resources; ex. time, memory, randomness, communication, qubits, etc.

Today, we'll talk about time complexity, i.e. what can be computed "quickly"? Later, we'll talk about space complexity and randomness.

As an overview, we'll talk about definitions of time complexity, and then we'll look at how with more resources, we have more power to solve problems. Next, we'll talk about differences in time complexities between different models. Lastly, we'll talk about P, the class of problems computable in polynomial time, and NP, the class of problems computable in non-deterministic polynomial time.

17.1.1 Time Complexity

Definition 17.2: Turing Machine Time Complexity

Let M be a Turing machine that always halts. The *running time* or *time complexity* of M is a function $f : \mathbb{N} \to \mathbb{N}$ defined by $f(n)$ as the maximum number of steps taken by M over *any* input of length n.

Note that by this definition, we're looking at the *worst-case* analysis of running time.

As a remark, when analyzing time complexity, we'll usually use big-oh notation. For example, we'll say that M runs in time $O(n^2)$ if M's running time $f : \mathbb{N} \to \mathbb{N}$ satisfies $f(n) = O(n^2)$.

Example 17.3

Consider $L = \{0^k 1^k \mid k \geq 0\}$.

Here is a Turing machine for L on input x of length n:

- ($O(n)$) Scan input tape and check that $x \in 0^*1^*$
- ($O(n^2)$) Repeat:
  - Find a 0 and a 1 on the tape and replace with X
  - If no 0 remains, or no 1 remains, stop
- ($O(n)$) Check that the entire string is marked X

This Turing machine then runs in time $O(n^2)$.

Definition 17.4: Time Complexity

Let $t : \mathbb{N} \to \mathbb{N}$. We define

$$\mathsf{TIME}(t(n)) = \{L' \mid \text{exists a TM } M \text{ with time complexity } O(t(n)) \text{ such that } L(M) = L'\}.$$

For example, $L = \{0^k1^k \mid k \geq 0\} \in \mathsf{TIME}(n^2)$ from our earlier example (Example 17.3).

Is n^2 the best time time complexity for L? It turns out that we can do better.

Example 17.5

Consider $L = \{0^k1^k \mid k \geq 0\}$.

Here is a Turing machine for L on input x of length n:

- ($O(n)$) Scan input tape and check that $x \in 0^*1^*$
- ($O(\log n)$ iterations) Repeat:
  - ($O(n)$) Check that the parity of # of 0's is equal to the parity of # of 1's)
  - If not, reject.
  - Cross out every other 0, and cross out every other 1.
  - Halt once all 0's and 1's are crossed out.

This Turing machine then runs in time $O(n \log n)$.

We claim that every regular language L is in $\mathsf{TIME}(n)$; this is because regular languages can be recognized by a DFA, so we can iterate once over the input and simply simulate the DFA. (This means that we can in fact solve this problem in exactly n time, plus some constant.)

As a stronger statement, we have the following theorem.

Theorem 17.6

The class of regular languages is exactly $\mathsf{TIME}(n)$.

In fact, any language that is not regular requires time $\Omega(n \log n)$.

This is quite peculiar; this means that there is no problem that takes time between $O(n)$ and $O(n \log n)$—at least, with a single-tape Turing machine.

Theorem 17.7

For "reasonable" functions $f, g : \mathbb{N} \to \mathbb{N}$ such that $f(n)\log^2(f(n)) = O(g(n))$, $\mathsf{TIME}(f(n)) \neq \mathsf{TIME}(g(n))$.

This theorem tells us that we can, for example, solve strictly more problems in time n^3 than in time n^2.

17.1.2 Time Complexity for different TM models

Example 17.8

Consider the example from before, with $L = \{0^k 1^k \mid k \geq 0\}$.

If we had two tapes, then we have the following procedure:

- ($O(n)$) Scan input tape and check that $x \in 0^*1^*$
- ($O(n)$) Copy input to second tape
- ($O(n)$) Move pointer in second tape to the first 1
- ($O(n)$) Continually advance both pointers in both tapes, checking that we have an equal number of 0's and 1's

This procedure runs in $O(n)$ time, which is better than what we had before.

Example 17.9

Consider the language $L' = \{w\#w \mid w \in \{0,1\}^*\}$.

Here is a standard Turing machine for L' on input x of length n:

- ($O(n)$) Check that $x \in (0+1)^*\#(0+1)^*$
- ($O(n)$ iterations, $O(n)$ per iteration) Repeat the following:
  - Record the first unmarked symbol in the substring preceding # and in the substring succeeding #
  - If only one symbol exists or the two symbols don't match, reject
  - Cross out these two symbols
  - When all symbols are crossed out, then stop and accept

This machine runs in time $O(n^2)$.

In fact, we claim that $L' \in \mathsf{TIME}(n^2)$, and this is optimal; any TM M solving L' will have time complexity $\geq \Omega(n^2)$. We can prove this using communication complexity; cutting the input string in half, we can look at the information passed across this halfway mark. In particular, to compare the two halves, we must exchange a lot of information.

With a two-tape Turing machine, we can do better:

- ($O(n)$) Check that $x \in (0+1)^*\#(0+1)^*$
- ($O(n)$) Copy the substring after # to the second tape
- ($O(n)$) Compare the substring before # stored on the first tape to the substring on the second tape
- Accept if and only if these two substrings are equal

These examples suggest that time complexity is not robust to different variants of the Turing machine model.

Recall that prior, we found that the multi-tape Turing machine is equivalent to the standard single-tape Turing machine in power. However, when we simulate the multi-tape Turing machine on the single-tape Turing machine, every step on the multi-tape Turing machine corresponds to a single pass over the equivalent single-tape Turing machine.

Theorem 17.10

Any multi-tape Turing machine with time-complexity $t(n)$ can be simulated by a single-tape Turing machine with time complexity $O(t(n)^2)$.

Proof. We can just execute the same simulation from prior for the multi-tape TM using a single-tape TM. Every step of the multi-tape TM can be simulated by $\leq O(t(n))$ steps on the single-tape model.

There are $t(n)$ steps total, so we need $O(t(n)^2)$ for the single-tape simulation. □

Recall also that we've shown that the two-way infinite tape Turing machine is equivalent in power to the standard one-way infinite tape Turing machine. Notice that our simulation was very efficient compared to the multi-tape simulation; we can simulate one step of the two-way infinite tape TM with a constant number of steps in the one-way infinite tape TM.

This means that any two-way infinite TM M that runs in time $t(n)$ can be simulated by a standard TM M' that runs in time $O(t(n))$.

17.1.3 Polynomial Time Computability

Definition 17.11: Polynomial Time Complexity

We define the polynomial time complexity as

$$\mathsf{P} = \bigcup_{k=1}^{\infty} \mathsf{TIME}(n^k).$$

Here, we can see that if L can be solved in polynomial time on a multi-tape TM, then L can be solved in polynomial time on a standard TM.

Proposition 17.12: Extended Curch-Turing Thesis

Turing machines can simulate any "reasonable" model of computation efficiently (with only a polynomial increase in time).

Note that this isn't a theorem or a conjecture; it can't be proven, as it's quite vague and imprecise. However, one challenge to this thesis is quantum algorithms; we know that factoring is hard (the best known algorithms are exponential in n), but there are quantum algorithms (namely Shor's algorithm) on quantum computers that can factor integers in polynomial time.

There are a couple ways to settle this challenge. One way is that maybe factoring *is* easy; just because we haven't found an efficient algorithm doesn't mean that one doesn't exist. In fact, from now on, we *won't* know whether any problems we discuss are truly hard or not—we'll be working with these uncertainties going forward, based only on what we know so far.

Another way to settle this is that quantum computers are unreasonable; we currently don't quite have the capabilities to fully implement Shor's algorithm.

One last way we can settle this is to tweak the thesis a little bit; instead of a Turing machine, we can use a quantum computer. That is, any reasonable model of computation can be simulated efficiently by a quantum computer.

As another remark, it may seem that polynomial time algorithms aren't actually that efficient; we could have something like $O(n^{100})$ and it'd still be in polynomial time, but the large exponent makes this practically slow.

One reason why we like polynomial time algorithms is because polynomials are closed under addition, multiplication, and composition. These operations correspond directly to running algorithms one after the other (addition), running algorithms in a loop (multiplication), or using an algorithm as a subroutine of another (composition). Doing these actions still gives us a polynomial time algorithm.

10/27/2022

Lecture 18

Non-deterministic polynomial time, Satisfiability

18.1 Non-deterministic Polynomial Time

Recall that we introduced non-deterministic Turing machines a little while ago. In particular, a non-deterministic TM accepts an input x if *there exists* a computation path on x that halts and accepts x. There could be many computation paths, but in each point in time, there are only finitely many options for the next move.

We say that a NTM *runs in time* $t(n)$ if all computation paths take $\leq t(n)$ steps. We also claim that any NTM that runs in time $t(n)$ can be simulated by a standard TM running in $2^{O(t(n))}$ time; this is done with the simulation we mentioned before, checking all possible paths (of which there are exponentially many).

In particular, the procedure would look something like the following:

- For $i = 1, 2, \ldots, t(n)$:
  - Run M on all inputs of length i.
  - If one of them accepts, then accept
- Otherwise, reject.

The maximum branching factor is $b = |Q| \cdot |\Gamma| \cdot 2$ (i.e. any state has at most b children; $|Q|$ for the next state of the TM, $|\Gamma|$ for the symbol on the tape, and 2 for the movement).

This means that in each iteration, we're checking at most b^i paths, each of depth i, with a total running time of $O((b^0 + b^1 + \cdots + b^{t(n)}) \cdot t(n))$; the last $t(n)$ factor is the time it takes to check each path. This is equivalent to $O(b^{t(n)}) = O(2^{t(n)})$. Further, we don't know if we can do better than this.

Formally, we have the following.

Definition 18.1: Non-deterministic Turing Machine Time Complexity

Let M be a non-deterministic Turing machine that always halts. Let $t : \mathbb{N} \to \mathbb{N}$. We say that M has time complexity $t(n)$ if for *any* $n \in \mathbb{N}$, for *any* input $x \in \Sigma^n$, *any* computation path of M on x halts in at most $t(n)$ time.

Note that again we're looking at the *worst-case* analysis of running time.

Definition 18.2: Non-deterministic Time Complexity

Let $t : \mathbb{N} \to \mathbb{N}$. We define

$$\mathsf{NTIME}(t(n)) = \{L \mid \text{exists a NTM } M \text{ that runs in time } O(t(n)) \text{ and decides } L\}.$$

Definition 18.3: Non-deterministic Polynomial Time Complexity

We define the non-deterministic polynomial time complexity as

$$\mathsf{NP} = \bigcup_{k=1}^{\infty} \mathsf{NTIME}(n^k).$$

Proposition 18.4

We claim that $\mathsf{TIME}(t(n)) \subseteq \mathsf{NTIME}(t(n))$, and that $\mathsf{NTIME}(t(n)) \subseteq \mathsf{TIME}(2^{O(t(n))})$.

A major open problem is whether $\mathsf{P} = \mathsf{NP}$; nobody knows whether this is true or not. There are people that are quite certain that the two are not equal, but we don't know for sure.

18.2 Satisfiability

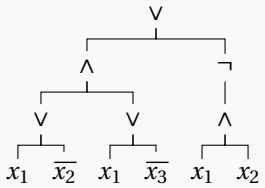
Next, we'll talk about several problems in NP; first, we will talk about satisfiability.

Definition 18.5: Boolean formula

A *boolean formula* over boolean variables $x_1, \dots, x_n$ is a tree whose leaves are marked with variables or their negations, and inner nodes are marked gates: AND ($\wedge$), OR ($\vee$), NOT ($\neg$).

Example 18.6

Here's an example of a boolean formula, represented as a tree:



Definition 18.7: Boolean Circuit

A *boolean circuit* is a looser version of a boolean formula, allowing for a DAG rather than a tree. In particular, each gate can have more than one outgoing edge (i.e. more than one parent).

Formally, a boolean circuit (with fan-in 2) over boolean variables $x_1, \dots, x_n$ is a DAG with one output gate and n input gates marked with $x_1, \dots, x_n$, such that every inner node is marked with an AND ($\wedge$), OR ($\vee$), or NOT ($\neg$) gate.

Definition 18.8: 3-CNF formula

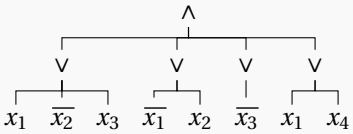
A *3-CNF formula* over boolean variables $x_1, \dots, x_n$ is an AND of clauses, each of which is the OR of at most 3 literals (i.e. variables or their negations).

Example 18.9

For example, the following is a 3-CNF formula:

$$\phi = (x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2) \wedge (\overline{x_2}) \wedge (x_1 \vee x_4).$$

As a tree, this is



Definition 18.10: Assignment

An *assignment* is a choice of values to the boolean variables $x_1, \ldots, x_n$.

Usually, $0 \equiv$ `false`, and $1 \equiv$ `true`.

Definition 18.11: Satisfiable

A formula ϕ is *satisfiable* if there exists an assignment that makes the formula evaluate to 1.

We have the following satisfiability problems:

- 3SAT: $\{\langle\phi\rangle \mid \phi \text{ is a satisfiable 3CNF formula}\}$
- FormulaSAT: $\{\langle\phi\rangle \mid \phi \text{ is a satisfiable boolean formula}\}$
- CSAT: $\{\langle\phi\rangle \mid \phi \text{ is a satisfiable Boolean circuit}\}$

The claim is that 3SAT, FormulaSAT, CSAT are all in NP.

Example 18.12

We can show that 3SAT is in NP; we want to give a NTM for 3SAT that runs in polynomial time.

On input $\langle\phi\rangle$, we can do the following:

- Check that $\langle\phi\rangle$ is a valid 3-CNF formula.
- Let n be the number of variables in ϕ.
- Non-deterministically guess an assignment for $x_1, \ldots, x_n$
- Check whether the assignment satisfies ϕ. If so, accept; otherwise reject.

If ϕ is satisfiable, then one of these paths would accept; otherwise, none of these paths will accept. This means that $\phi \in$ 3SAT if and only if this NTM accepts $\langle\phi\rangle$.

Here is an alternative definition of NP:

Theorem 18.13: Non-deterministic Polynomial Time Complexity (alt.)

A language $L \in$ NP if and only if there exists a polynomial-time Turing machine (called the "verifier") V and constants $c, k \in \mathbb{N}$ such that

$$L = \{x \mid \exists y \text{ such that } |y| \leq c \cdot |x|^k \text{ and } V(x, y) \text{ accepts}\}.$$

That is, an input x is in the language if there is a y (polynomial in the length of x) such that V accepts.

Proof. In the forward direction, if L satisfies the above, we want to show that $L \in \mathsf{NP}$. To do this, we can construct a NTM for L on input x:

- Guess a y of length $\leq c|x|^k$.
- Run V on (x, y). If V accepts, then accept; otherwise, reject.

We can see that $x \in L$ if and only if there exists a y such that $V(x, y)$ accepts, so this NTM correctly decides L.

In the opposite direction, we want to show that any language $L \in \mathsf{NP}$ has a verifier V as described above. The idea is that given a NTM, we want some kind of proof that NTM has an accepting path—this proof is just the accepting path.

Formally, if $L \in \mathsf{NP}$, there exists a NTM M that decides L running in polynomial time. This means that there exists some $c, k \in \mathbb{N}$ such that M runs in time $c|x|^k$.

We can think of y as an encoding of a path in M's computation tree. The verifier V on (x, y) is as follows:

- Interpret y as a string over the alphabet $Q \times \Gamma \times \{L, R\}$ of length at most $c|x|^k$.
- Follow the path defined by y on the computation tree defined by M.
- Return the value of the leaf. (If the leaf accepts, we accept; if the leaf rejects, we reject.)

If $x \in L$, then there would exist a y that makes $V(x, y)$ accept (i.e. the y is the accepting path), and otherwise no such y would exist. □

The takeaway here is that NP captures the class of languages that can be verified efficiently given a polynomial length solution.

Here are some more NP problems:

- Clique: $\{(G, k) \mid G$ is a graph and there exists a k-clique in $G\}$

  Here, a k-clique is a set of k fully-connected vertices.
- IndSet: $\{(G, k) \mid G$ is a graph and there exists an independent set in G with k vertices$\}$

  Here, an independent set is a set of vertices that has no edges between them.
- 3Col: $\{\langle G\rangle \mid G$ is a 3-colorable graph$\}$
- Sudoku: Given a partial assignment, is there a valid choice to the blank cells that satisfy all constraints (no repetitions in each row, column, and block).

There are many many more: Hamiltonian paths, Hamiltonian cycles, TSP, subset sum, set cover, vertex cover, etc. which we'll cover in the next couple lectures.

Again, the problem of whether $\mathsf{P} = \mathsf{NP}$ is unknown. The conjecture is that $\mathsf{P} \neq \mathsf{NP}$. This is necessary to most of cryptography as well; if $\mathsf{P} = \mathsf{NP}$, then we can guess the "secret key" and decrypt.

18.3 Polynomial-time Reductions

We can't resolve $\mathsf{P} = \mathsf{NP}$, but instead what we can do is identify the hardest problems within a class of problems. To do this, we need a notion of reductions.

Similar to the definition of a computable mapping, we can refine the definition for a *polynomial-time* computable mapping.

Definition 18.14: Polynomial-time Computable Mapping

A mapping $f : \Sigma^* \to \tilde{\Sigma}^*$ is polynomial-time computable if here exists a TM that runs in polynomial time and on input $x \in \Sigma^*$, the machine halts when the tape's content is $f(x)$.

Note that this means that the mapping shouldn't give too long of an output either, as we cannot write an exponentially long output in polynomial time.

Definition 18.15: Polyonmial-time Reducible

A language A over alphabet Σ is polynomial-time reducible to language B over alphabet $\tilde{\Sigma}$, written $A \leq_p B$, if there exists a polynomial-time computable function $f : \Sigma^* \to \tilde{\Sigma}^*$ such that for every $w \in \Sigma^*$,

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

f is called a polynomial-time reduction from A to B.

Similar to mapping reductions, we have a relation between the time complexity of A and B.

Theorem 18.16

If $A \leq_p B$ and $B \in \mathsf{P}$, then $A \in \mathsf{P}$. Equivalently, in the contrapositive, if $A \leq_p B$ and $A \notin \mathsf{P}$, then $B \notin \mathsf{P}$.

Proof. Let M_B be a polynomial-time TM for B, and let M_f be a polynomial-time TM computing f such that $w \in A \iff f(w) \in B$. We want to show that $A \in \mathsf{P}$.

We can write the TM M_A for A as follows:

- Compute $y = f(w)$ by running M_f on w.
- Run M_B on y.

With this, we have that if $w \in A$, then M_B will accept $y = f(w)$ by definition of f, and thus M_A will accept w. Similarly, if $w \notin A$, then M_B will reject $y = f(w)$ by definition of f, and thus M_A will reject w.

This TM also runs in polynomial time; computing $y = f(w)$ runs in polynomial time, since f is a polynomial-time computable mapping, and since y is of polynomial-length, M_B on y will also run in polynomial time. □

Theorem 18.17

If $A \leq_p B$ and $B \in \mathsf{NP}$, then $A \in \mathsf{NP}$. Equivalently, in the contrapositive, if $A \leq_p B$ and $A \notin \mathsf{NP}$, then $B \notin \mathsf{NP}$.

Proof. The exact same proof as with the previous theorem for P holds; we'd just use NTMs instead. □

Proposition 18.18: Transitivity of Reductions

If $A \leq_p B$ and $B \leq_p C$, then $A \leq_p C$.

18.4 NP-hardness and NP-completeness

Definition 18.19: NP-hard

A language L is NP-hard if for all $L' \in \mathsf{NP}$, $L' \leq_p L$.

Definition 18.20: NP-complete

A language L is NP-computable if $L \in \mathsf{NP}$, and L is NP-hard.

In other words, L is NP-complete if L is the "hardest" problem in NP. Although this may seem like a very restrictive definition, we'll see that there's in fact an abundance of NP-complete problems.

11/2/2022

Lecture 19

CIRCUITSAT, 3SAT

Today, we'll show that 3SAT is NP-complete; this is the starting point to show that thousands of natural combinatorial problems are NP-complete as well.

The plan for the proof that 3SAT is NP-complete is in two parts:

1. Show that CSAT is NP-complete
2. Show that CSAT $\leq_p$ 3SAT

Here, once we show that CSAT is NP-complete, we know that every language in NP reduces to CSAT, and once we show that CSAT reduces to 3SAT, we can then conclude that every language in NP reduces to 3SAT, making it NP-hard.

Recall that a boolean circuit is a DAG with one output gate and n input gates $x_1, \ldots, x_n$ such that every inner node is either an AND ($\wedge$), an OR ($\vee$), or a NOT ($\neg$) gate. (An example is shown in Fig. 19.1)

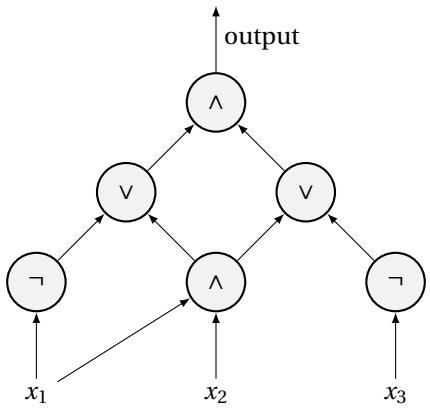


Figure 19.1: Example of a boolean circuit

A circuit C naturally defines a function $f : \{0,1\}^n \to \{0,1\}$, i.e. we have n inputs, and we evaluate the circuit to produce one output bit. This is the truth table of the circuit.

Proposition 19.1

We claim that for any boolean function $f : \{0,1\}^n \to \{0,1\}$, there exists a boolean circuit with $O(2^n n)$ gates that computes f.

Proof. There are two ways to do this. Given a truth table of f, we look at each row in the table.

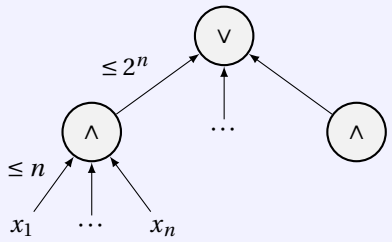
For each assignment a such that $f(a) = 1$, take the AND of the literals in the row checking for $x \equiv a$. For example, given the truth table

x_1	x_2	$\cdots$	x_3	x_4	$f(x)$
0	0	$\cdots$	0	0	0
0	0	$\cdots$	0	1	1
$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\vdots$
1	1	$\cdots$	1	1	0

.

The second row would give the formula $\bar{x}_1 \wedge \bar{x}_2 \wedge \cdots \wedge \bar{x}_{n-1} \wedge x_n$.

We then take the OR of all of these assignments such that $f(a) = 1$, giving a circuit of the following form:



Notice that we have at most 2^n wires entering the root $\vee$, and we have at most n wires entering the second level of $\wedge$. However, we'd like this to have a fan-in of 2 (i.e. each gate should have at most 2 inputs); how do we do this?

We can just replace any gate with m inputs with a binary tree with m leaves. This doesn't increase the number of edges or gates by much when doing this replacement, so we still have $O(n2^n)$ gates.

Alternatively, we can look at all the rows where $f(a) = 0$ instead, and take the OR of all the literals to check $x \neq a$. For example, the first row of the table earlier gives $x_1 \vee x_2 \vee \cdots \vee x_{n-1} \vee x_n$.

We then take the AND of all of these assignments to get our final formula. Further, this is a CNF; that is, we've shown that any function of n variables can be written as a CNF with clauses of size n. With a similar analysis, this formulation also gives us a circuit with at most $O(n2^n)$ gates. □

We can define CIRCUITSAT as the following:

$$\text{CIRCUITSAT} = \{\langle C \rangle \mid C \text{ is a satisfiable boolean circuit with fan-in 2}\}.$$

19.1 CIRCUITSAT is in NP

Firstly, we can show that CIRCUITSAT $\in$ NP.

We can define the relation

$$R = \{(\langle C \rangle, a) \mid C \text{ is a circuit, } a \text{ is an assignment such that } C(a) = 1\}.$$

R can be checked in polynomial time by some verifier V; we can just evaluate the circuit in topological order, until we get to the output.

Further, we have

$$\text{CIRCUITSAT} = \{\langle c \rangle \mid (\exists a)(V(\langle c \rangle, a) = 1)\}.$$

This means that CIRCUITSAT $\in$ NP.

19.2 CIRCUITSAT is in NP-hard

Next, we can show that CIRCUITSAT is NP-hard.

Let $L \in$ NP. We need to show that $L \leq_p$ CIRCUITSAT. We know that L is of the form

$$L = \{x \mid \exists y : |y| \leq c|x|^k, V(x,y) = 1\}.$$

Given x and y, we want to express the condition $V(x,y) = 1$ as a question about circuits.

The main idea here is that circuits can simulate Turing machines.

19.2.1 Turing machine configurations

Recall that a configuration of a Turing machine is a snapshot of the Turing machine's state at a given time. This snapshot contains the content of the tape, the head location, and the current state of the finite control.

For example, on the input "0110", the initial configuration would be "$0/q_0$ 1 1 0", and the next configuration may be "1 $1/q_1$ 1 0". WLOG, suppose the TM clears the tape at the very end, so we end up with a final configuration of $\_/q_{acc}$ or $\_/q_{rej}$.

Since we know that $L \in$ NP, there exists a verifier V that runs in time $O(|(x,y)|^\ell)$. The idea here is to consider the table of the TM V on the input $w = (x,y)$; i.e. the table of all configurations during V's execution, as shown in Fig. 19.2.

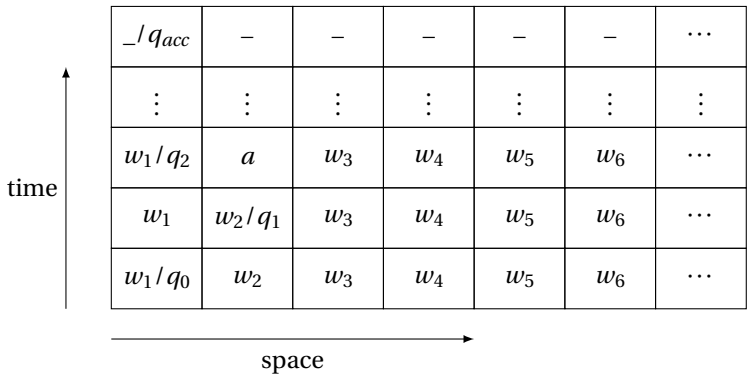


Figure 19.2: Table of the execution of a Turing machine

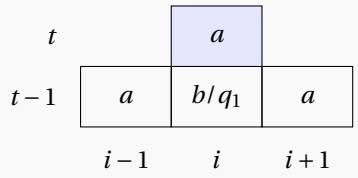
We know that the size of the table is polynomial with respect to the size of x. This is because the number of time steps we need is at most $O(|w|^\ell)$, since V runs in polynomial time, and the amount of space we need is also at most $O(|w|^\ell)$, because we can only get to as many cells as there are time steps. Since $w = (x,y)$ is polynomial with respect to $|x|$ (since $|y|$ is polynomial in $|x|$), the entire table is polynomial in $|x|$.

Crucially though, notice that the computation is local: each cell in the table depends only on the 3 cells below it. This is because the head can only move one location at a time.

Formally, let $A_{t,i}$ be the content of the ith cell in the tth configuration. We claim that $A_{t,i}$ is a function of $A_{t-1,i-1}$, $A_{t-1,i}$, $A_{t-1,i+1}$.

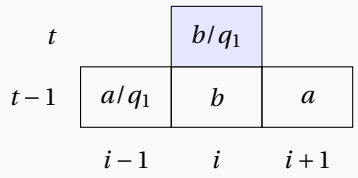
Example 19.2

Suppose we have the following, where the head of the Turing machine is below $A_{t,i}$:



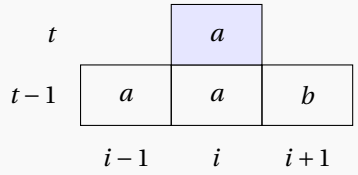
The top cell $A_{t,i}$ is determined completely by the transition function for the Turing machine; we know in the previous timestep the head is pointing at the b, in the internal state q_1, so $A_{t,i} = \delta(q_1, b)$.

Suppose we have the following, where the head of the Turing machine is diagonally below $A_{t,i}$:



Similarly, the top cell $A_{t,i}$ is determined completely by the transition function for the Turing machine. Notice that in this case, the symbol on the tape will never change, since the head was at symbol $i-1$, so it cannot write to the symbol at i. This means that we'd only need to check the transition function to see if the head will move to i.

Suppose we have the following, where the head is not in any of the cells below $A_{t,i}$:



Notice that since the head is not in any of $A_{t-1,i-1}$, $A_{t-1,i}$, or $A_{t-1,i+1}$, it is impossible for the symbol at $A_{t,i}$ to change in one step. This means that $A_{t,i} = A_{t-1,i}$. (This is what happens in most locations.)

With this in mind, we can build a boolean circuit STEP, taking in the binary encoding of 3 cells $A_{t-1,i-1}$, $A_{t-1,i}$, and $A_{t-1,i+1}$, outputting the binary encoding of the cell $A_{t,i}$.

This is a finite boolean function $\text{STEP} : \{0,1\}^{3m} \to \{0,1\}^m$, where $m = O(1)$ is a constant. This means that STEP can be implemented by $O(m \cdot m \cdot 2^{3m}) = O(1)$ size circuits (again since m is a constant).

As a remark, STEP has an output of length m, but we can consider each bit of the output as a function $\{0,1\}^{3m} \to \{0,1\}$, and putting them all together in the end.

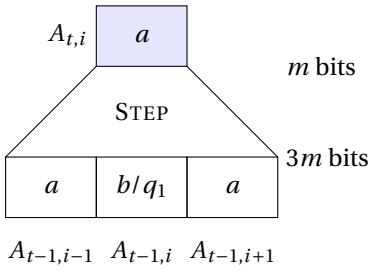


Figure 19.3: The STEP function computing the symbol $A_{t,i}$

Since the size of the STEP circuit is independent of the input length n, we can now introduce a STEP circuit for computing $A_{t,i}$ for every $t \geq 1$ and $i \geq 1$.

This means that each cell in row t will be computed by a new STEP circuit, depending on the three cells below it in row $t-1$. We also need to initialize the table in the first row; the initial configuration is just a function of the input:

$$A_{0,i} = \begin{cases} \text{INIT}(w_1, 1) & \text{if } i = 1 \\ \text{INIT}(w_i, 0) & \text{if } 2 \le i \le |w| \\ \text{INIT}(\_, 0) & \text{if } i > |w| \end{cases}.$$

Here, $\text{INIT}(\cdot, 1)$ means that the head is in the given position, and $\text{INIT}(\cdot, 0)$ indicates that the head is not in the given position. (That is, we initialize the tape such that the head of the Turing machine is in the first cell.)

Further, the final configuration accepts if and only if its first cell is $\_/q_{acc}$; we can check this using an $O(1)$ size circuit $\text{ISACCEPT}(A_{T,1})$.

The full circuit C_V that simulates the verifier on the input $w = (x, y)$ is shown in Fig. 19.4.

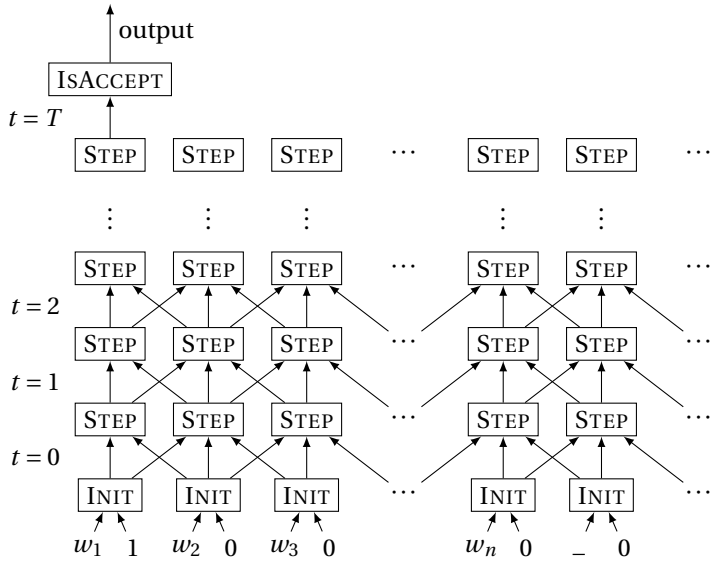


Figure 19.4: Circuit C_V simulating the Turing machine V on input w

For the full reduction, we want to reduce $L \in \mathsf{NP}$ to CIRCUITSAT. On input $x \in L$, we can construct the circuit C_V, and then hard-wire the values for x in $w = (x, y)$. This hard-wired circuit will be $C_{V,x}$, with the only input being y.

Analyzing the correctness, we have:

$$x \in L \iff \exists y : V(x, y) = 1 \iff \exists y : C_V(x, y) = 1 \iff \exists y : C_{V,x}(y) = 1 \iff C_{V,x} \text{ satisfiable.}$$

In words, since $L \in \mathsf{NP}$, we know that $x \in L$ if and only if there exists some y such that $V(x, y) = 1$. Since C_V simulates V, we know that $V(x, y) = 1$ if and only if $C_V(x, y) = 1$. We've hard-wired x into the circuit, so $C_V(x, y) = 1$ if and only if $C_{V,x}(y) = 1$. This is exactly the problem of determining whether $C_{V,x}$ is satisfiable.

With this reduction, given x, we compute $C_{V,x}$; this reduction runs in polynomial time, since the table is of polynomial size, and each cell in the table requires $O(1)$ many gates. This is because the height of the table is

$$\text{height} \le O(|w|^\ell) = O((|x| + |y|)^\ell) = O(|x|^{k\ell}).$$

Overall, the size of the circuit is at most $O(|x|^{2k\ell})$.

We've just shown that CIRCUITSAT is in NP, and it is NP-hard, so this concludes the proof that CIRCUITSAT is NP-complete.

19.3 3SAT is NP-complete

To show that 3SAT is NP-complete, we can use the fact that CIRCUITSAT is NP-complete; all we need to do is reduce CIRCUITSAT to 3SAT. That is, we want a polynomial time reduction CIRCUITSAT $\leq_p$ 3SAT.

As a first naive attempt, given a circuit C, we can try to convert it to a CNF formula. This doesn't always work—some circuits with $O(n)$ gates require exponential size CNFs, so this reduction will not be polynomial time.

For a second try, given a circuit C, we can try to find a 3CNF formula ϕ on more variables, such that C is satisfiable if and only if ϕ is satisfiable.

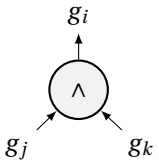
The idea is to add one variable per gate in the circuit; we can then add constraints to check that the gates are consistent with each other, and that the final output gate outputs 1.

We then have the following gate conversions:

- AND gate: check $g_i \equiv g_j \wedge g_k$

  This is a boolean condition on 3 bits, which can be expressed as a 3CNF:

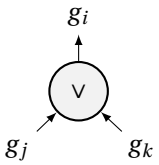
  $$(g_i \vee \overline{g_j} \vee \overline{g_k}) \wedge (\overline{g}_i \vee g_j) \wedge (\overline{g}_i \vee g_k).$$



- OR gate: check $g_i \equiv g_j \vee g_k$

  Similarly, we can express this condition as a 3CNF:

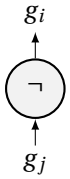
  $$(g_i \vee \overline{g_k}) \wedge (g_i \vee \overline{g}_j) \wedge (\overline{g}_i \vee g_j \vee g_k).$$



- NOT gate: check $g_i \equiv \neg g_j$

  We can express this condition as a 3CNF as well:

  $$(g_i \vee g_j) \wedge (\overline{g}_i \vee \overline{g}_j).$$



We can then reduce C to $\phi = \left(\bigwedge_{i=1}^{m} \phi_i\right) \wedge g_m$, where g_m checks that the output is a 1, and g_i is the 3CNF that checks the consistency of gate g_i with the inputs entering the gate computing g_i.

Looking at the correctness of this reduction, we firstly know that this reduction is polynomial time computable; we have constantly many clauses per gate, and the number of gates is polynomial in the input size.

If $C \in$ CIRCUITSAT is satisfiable, then there exists some x^* such that $C(x^*) = 1$, which gives us $x_1^*, \ldots, x_n^*, g_1^*, \ldots, g_m^*$ that satisfy ϕ. Here, we'd just propagate $x_1^*, \ldots, x_m^*$ to the circuit's gates—there is only one way to assign the output of a gate given its inputs.

If $x_1^*, \ldots, x_n^*, g_1^*, \ldots, g_m^*$ satisfies ϕ, then the local constraint checks enforce the outputs for all the gates in C are consistent with the inputs $x_1^*, \ldots, x_n^*$. This means that x^* satisfies the circuit C.

As such, we've shown a reduction from CIRCUITSAT to 3SAT, and since CIRCUITSAT is NP-complete, this means that 3SAT is also NP-complete.

11/8/2022

Lecture 20

NP-complete Problems

Today, we'll show the following problems are NP-complete:

- Independent set (INDSET)

- CLIQUE
- VERTEXCOVER
- SUBSETSUM

20.1 General Recipe for NP-completeness

To show that some language B is NP-complete, we have the following procedure:

1. Show that B is in NP. (This is usually the easy part.)
2. Find any NP-complete language A, and show that $A \leq_p B$. To do this, we do the following:
   2.1. Come up with a reduction f.
   2.2. Show that YES $\implies$ YES; that is, $w \in A \implies f(w) \in B$.
   2.3. Show that NO $\implies$ NO; that is, $w \notin A \implies f(w) \notin B$, or equivalently $f(w) \in B \implies w \in A$.
   2.4. Show that f is polynomial-time computable.

20.2 Independent Set

The independent set problem takes an undirected graph $G = (V, E)$ and an integer k; we want to output whether there is an independent set of k vertices in G.

Definition 20.1: Independent set

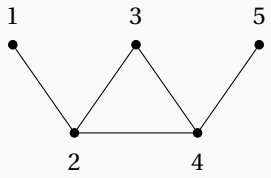
A set of vertices $S \subseteq V$ is an independent set if no two vertices in S are adjacent.

We define the independent set problem with the language

$$\text{INDSET} = \{\langle G, k\rangle \mid G \text{ is undirected graph, } k \in \mathbb{Z} \text{ such that there is an independent set of size } k \text{ in } G\}.$$

Example 20.2

Consider the following graph:



Vertices 1, 3, and 5 form an independent set; the three vertices are not adjacent in the graph. We can also show that there does not exist any independent set of size 4, because we have a triangle on vertices 2, 3, and 4. This means that we can only choose one vertex out of $\{2, 3, 4\}$ to include in our independent set, giving a maximum of 3 vertices in any independent set.

This means that $\langle G, 3\rangle \in$ INDSET, but $\langle G, 4\rangle \notin$ INDSET.

To show that INDSET is NP-complete, we need to first show that INDSET $\in$ NP.

We can define the independent set problem as

$$\text{INDSET} = \{\langle G, k\rangle \mid \exists S : V(\langle G, k\rangle, S) = 1\}.$$

We also have the relation $R = \{(\langle G, k\rangle, S) \mid G \text{ is undirected graph, } S \subseteq V(G), |S| = k, \text{ and } S \text{ is an independent set in } G\}$.

The verifier V checks that G is an undirected graph, S is a subset of the vertices of size k, and that no $u, v \in S$ share an edge. This can be done in polynomial time, so we have that INDSET $\in$ NP.

Next, we need to show that INDSET is NP-hard. It suffices to show that for some NP-hard language L, $L \leq_p$ INDSET.

At this point, we only know that 3SAT is NP-hard, so we will show a reduction from 3SAT $\leq_p$ INDSET. That is, given some 3CNF formula ϕ, we want to construct an instance of INDSET (i.e. a graph G and an integer k) such that there exists an independent set of size k in G if and only if there is a satisfying assignment to ϕ.

Recall that in 3SAT, we want to select at least one literal in each clause such that you never select both a variable and its negation. The idea is that we want to use "gadgets" to connect clauses together; each vertex is labeled by a literal from the formula ϕ. We also want to introduce edges between all literals that are negations of each other.

Figure 20.1 shows an example of this conversion.

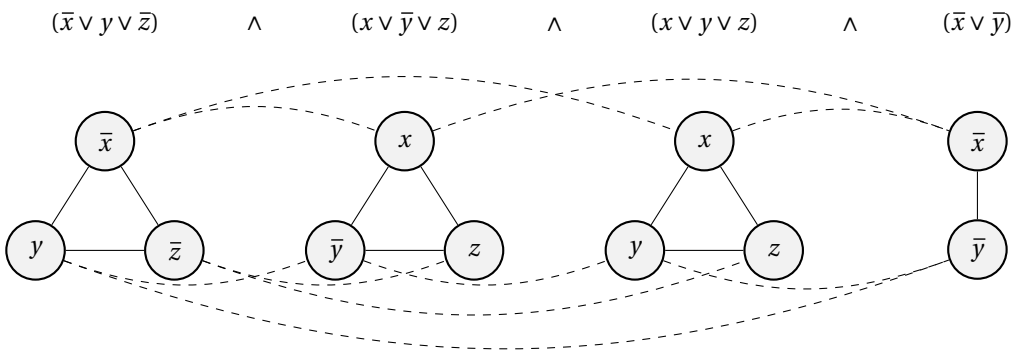


Figure 20.1: Conversion from a 3CNF formula ϕ to a graph G_ϕ for independent set

We claim that if m is the number of clauses in ϕ, then ϕ is satisfiable if and only if G_ϕ has an independent set of size m. That is, $\phi \in$ 3SAT $\iff \langle G_\phi, m \rangle \in$ INDSET.

Suppose we have a satisfiable 3CNF formula ϕ. We want to show that there exists an independent set of size m in G_ϕ. Since ϕ is satisfiable, then there exists an assignment $a \in \{0,1\}^m$ that satisfies all clauses.

This means that in every clause, we pick exactly one literal that is satisfied by a. In the gadgets for G_ϕ, we can pick a vertex corresponding to the picked literal. These m vertices we pick (one for each clause) forms an independent set—we have one vertex in each gadget, and no literal and its negation is ever picked at the same time (because otherwise we'd have $x = 1$ and $\bar{x} = 1$ in the assignment, which is impossible).

Suppose we have an independent set S of size m in G_ϕ. We want to show that the corresponding formula ϕ is satisfiable.

We know that there are m gadgets, and that the independent set is of size m. The structure of G_ϕ implies that we cannot pick more than one vertex in each gadget, because otherwise the two picked vertices are adjacent. Together, this means that S picks exactly one vertex in each gadget.

This corresponds to picking exactly one literal in each clause of ϕ. Further, we know that these literals are picked consistently—no literals are picked at the same time as their negations, because of the edges we included between literals and their negations. This means that the literals corresponding to the vertices we picked for S gives a satisfying assignment for ϕ.

As such, we've just shown that $\phi \in$ 3SAT $\iff \langle G_\phi, m \rangle \in$ INDSET.

20.3 Clique

Definition 20.3: Clique

A set S of vertices in an undirected graph G is called a *clique* if all pairs in S share an edge.

We define the clique problem with the language

$$\text{CLIQUE} = \{\langle G, k\rangle \mid G \text{ is an undirected graph with a clique of } k \text{ vertices}\}.$$

To show that CLIQUE is NP-complete, we first need to show that CLIQUE $\in$ NP. The witness is just the clique S itself, and we can verify in polynomial time (i.e. check all $\binom{n}{2}$ pairs of vertices in S) whether all vertices in S are adjacent.

Next, we'll show that CLIQUE is NP-hard through a reduction INDSET $\leq_p$ CLIQUE.

Note that for a graph $G = (V, E)$, its complement $\overline{G} = (V, \overline{E})$ is defined with $\overline{E} = \{(u, v) \mid u, v \in V, (u, v) \notin E\}$. The reduction is simply the graph $\overline{G}$.

We claim that there is an independent set in G if and only if there is a clique in $\overline{G}$. This follows almost directly from definitions, since an independent set S must have no edges between vertices in S, and a clique S' must have all edges between vertices in S'. This means that an independent set in G is a clique in $\overline{G}$, and vice versa.

As such, with this reduction from INDSET to CLIQUE, we've shown that CLIQUE is NP-hard, and thus NP-complete.

20.4 Vertex Cover

Definition 20.4: Vertex Cover

A set S of vertices in an undirected graph G is called a *vertex cover* if it touches all edges in the graph.

We define the vertex cover problem with the language

$$\text{VERTEXCOVER} = \{\langle G, k\rangle \mid G \text{ is an undirected graph with a vertex cover of size } k\}.$$

To show that VERTEXCOVER is NP-complete, we first need to show that VERTEXCOVER $\in$ NP. Here, the witness is the vertex cover S; we can verify in polynomial time (i.e. check all edges) whether or not the edge is incident to a vertex in S.

Next, we'll show that VERTEXCOVER is NP-hard through a reduction INDSET $\leq_p$ VERTEXCOVER. On the input $\langle G, k\rangle$, the output of the reduction is simply $\langle G, n - k\rangle$, where n is the number of vertices in the graph.

The idea here is that S is an independent set in G if and only if $\overline{S}$ is a vertex cover in G.

To see why, consider all the edges in the graph. We have three choices for any edge: it is between two vertices in S, between two vertices in S', or across S and S'.

If S is an independent set, then there can be no edges between two vertices in S, so all edges must either be within S', or between S and S'—this means that all edges are covered by S', and S' is a vertex cover. Similarly, if S' is a vertex cover, then there can be no edges between two vertices in S, because then these edges would not be covered by S'.

As such, with this reduction from INDSET to VERTEXCOVER, we've shown that VERTEXCOVER is NP-hard, and thus NP-complete.

20.5 Subset Sum

In the subset sum problem, also known as the knapsack problem, we're given a sequence of natural numbers $a_1, \ldots, a_m \in \mathbb{N}$, and a target $t \in \mathbb{N}$. We want to determine whether there exists a partial sum of a_i's that equals t.

Formally, we define the subset sum problem with the language

$$\text{SUBSETSUM} = \{\langle a_1, \ldots, a_m, t\rangle \mid a_1, \ldots, a_m, t \in \mathbb{N}, \exists S : \sum_{i \in S} a_i = t\}.$$

To show that SUBSETSUM is NP-complete, we first need to show that SUBSETSUM $\in$ NP. Here, the witness is the subset S; we can verify in polynomial time with respect to m (i.e. linear time by taking the sum) whether the sum of all elements in S is equal to t (along with verifying that S is indeed a subset of the given numbers).

Next, we'll show that SUBSETSUM is NP-hard through a reduction 3SAT $\leq_p$ SUBSETSUM.

Given a 3CNF formula $\phi = C_1 \wedge C_2 \wedge \cdots \wedge C_k$ on variables $x_1, \ldots, x_\ell$, we want to construct an instance of the SUBSET SUM problem $a_1, \ldots, a_m, t$ such that ϕ is satisfiable if and only if some partial sum of $a_1, \ldots, a_m$ equals t.

To do this, we can generate a number for each clause and for each literal; we'll write the numbers in decimal notation.

Example 20.5

Consider the formula from before:

$$\phi = (\bar{x}_1 \vee x_2 \vee \bar{x}_3) \wedge (x_1 \vee \bar{x}_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3) \wedge (\bar{x}_1 \vee \bar{x}_2).$$

We can construct numbers for each literal as follows:

literal	number	x_1	x_2	x_3	C_1	C_2	C_3	C_4
x_1	y_1	1	0	0	0	1	1	0
$\bar{x}_1$	z_1	1	0	0	1	0	0	1
x_2	y_2	0	1	0	1	0	1	0
$\bar{x}_2$	z_2	0	1	0	0	1	0	1
x_3	y_3	0	0	1	0	1	1	0
$\bar{x}_3$	z_3	0	0	1	1	0	0	0

Specifically for literals x_i and $\bar{x}_i$, the ith digit is a 1, denoting that the literal is associated with x_i. For the next k digits, we put a 1 in the ith digit if the literal appears in the clause C_i.

Looking at a satisfying assignment, for example $(x_1, x_2, x_3) = (0, 0, 1)$ shaded in the table above, notice that the sum $z_1 + z_2 + y_3$ is 111 1212. More generally, the first ℓ digits will always be 1's, and the next k digits will always be nonzero.

The last k digits in the sum correspond to how many of the literals in the clause are satisfied by the assignment; this means that the last digits must be $0 \leq d_i \leq 3$.

The issue here is that we can't specify "last k digits should be nonzero" with a single target as-is. To solve this issue, we can make the target (in this example) $t = 111\,3333$, and introduce new numbers for each clause, giving us the freedom to match t:

clause	number	x_1	x_2	x_3	C_1	C_2	C_3	C_4
C_1	g_1	0	0	0	1	0	0	0
	h_1	0	0	0	1	0	0	0
C_2	g_2	0	0	0	0	1	0	0
	h_2	0	0	0	0	1	0	0
C_3	g_3	0	0	0	0	0	1	0
	h_3	0	0	0	0	0	1	0
C_4	g_4	0	0	0	0	0	0	1
	h_4	0	0	0	0	0	0	1

Notice that we only give freedom to add at most 2 to the last k digits of the numbers.

In general, the construction is as follows:

- For each literal x_i (or $\bar{x}_i$) for $1 \le i \le \ell$, we create a new number y_i (or z_i for $\bar{x}_i$) such that:
  - The ith digit is a 1
  - The $\ell + j$th digit is a 1 if x_i appears in clause C_j.
- For each clause C_i for $1 \le i \le k$, create two identical numbers g_i and h_i such that the $(\ell + i)$th digit is a 1
- The target is $t = \underbrace{11\ldots1}_{\ell}\ \underbrace{33\ldots3}_{k}$

We claim that ϕ has a satisfying assignment if and only if there is a subset of $\{y_1, \ldots, y_\ell, z_1, \ldots, z_\ell, g_1, \ldots, g_k, h_1, \ldots, h_k\}$ that sums to t.

Suppose ϕ is satisfiable; this means that there exists an assignment $a \in \{0,1\}^\ell$ to $x_1, \ldots, x_\ell$ that satisfies $C_1, \ldots, C_k$.

Then, for any $1 \le i \le \ell$, we add y_i to the partial sum if $a_i = 1$, and add z_i to the partial sum if $a_i = 0$. So far, the partial sum equals $11\ldots1\ d_1 d_2 \ldots d_k$, for $d_1, \ldots, d_k \in \{1,2,3\}$. Here, d_j represents the number of literals in C_j that are satisfied by the assignment.

Next, for $1 \le j \le k$, if $d_j = 1$, then add both g_j and h_j to the partial sum, and if $d_j = 2$, add only g_j to the partial sum. This ensures that the digit equals 3 at the very end.

Together, the subset we've constructed will be of the form $11\ldots1\ 33\ldots3 = t$, so there exists a subset that sums to t.

In the other direction, suppose there exists a partial sum that gives t. First, we can notice that we'll never have a carry; of the first ℓ digits, we only have 2 ones to choose from (i.e. y_i and z_i), and in the last k digits, we only have at most 5 ones to choose from (i.e. two from g_i and h_i, and at most three more from the 3 literals in the clause). This means that we'll never reach a digit greater than 5, so no carries are possible.

This means that from the structure of the numbers, we must have picked exactly one from each pair $\{y_i, z_i\}$. As such, suppose we define $a \in \{0,1\}^\ell$ such that $a_i = 1$ if and only if y_i is picked for the partial sum.

Suppose we look at the clause C_j. We know that 3 numbers contributed to the $(\ell + j)$th digit in the partial sum; since only at most 2 of these ones came from g_j and h_j, we can conclude that at least one of the numbers corresponding to a literal in C_j must have been included in the partial sum.

This means that the assignment a satisfies the constraints in ϕ, and we've found a satisfying assignment to ϕ.

As such, with this reduction from 3SAT to SUBSETSUM, we've shown that SUBSETSUM is NP-hard, and thus NP-complete.

11/10/2022

Lecture 21

Space Complexity

Recall that when we talk about complexity, we are interested in what can be computed "efficiently". We started with considering time complexity, with the aim of determining what we can do quickly. Today, we'll talk about what we can do with limited space.

We had discussed algorithms with small space with automata and streaming algorithms, though these models have the restriction of taking the input one by one. Turing machines don't have this restriction, and we'll look at a few results of this.

21.1 Space Complexity Definitions

For a first natural attempt at the definition of space complexity, for a deterministic TM M that always halts, we may define the space complexity of M as a function $f : \mathbb{N} \to \mathbb{N}$ such that $f(n)$ is the maximum number of cells that M visits on any input of length n. We'd have a similar definition for nondeterministic Turing machines as well.

The issue with this definition comes when we start to talk about sublinear space complexity; it doesn't really make sense to talk about a Turing machine that uses something like $O(\log n)$ space or $O(\sqrt{n})$ space, because we can't even read the entire input.

This motivates the following definition of space complexity, which we'll be using for this course.

Definition 21.1: Space Complexity

Consider a two tape Turing machine, where one tape is the input tape, and one tape is a work tape. The input tape is read-only, and the work tape is read/write.

We define the space complexity of this two-tape Turing machine as the function $f : \mathbb{N} \to \mathbb{N}$ such that $f(n)$ is the maximum number of cells accessed on the work tape on any input of length n.

The tape layout in this definition is illustrated in Fig. 21.1. In contrast to the previous definition, here we're interested in the amount of *additional* space necessary for a Turing machine; this means that we can read the entire input and still have the possibility of a small sublinear space complexity.

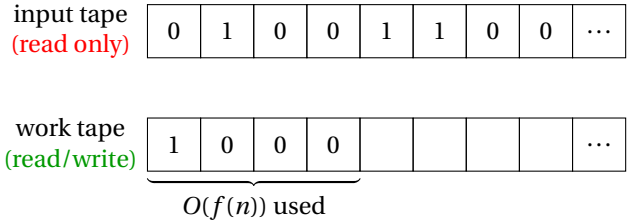


Figure 21.1: Two-tape Turing machine definition of space complexity

We can simulate DFAs without using any additional space at all, and with streaming algorithms, we'd be using some additional sublinear space (although here we're allowed to read the input however many times we want).

Example 21.2

Consider $A = \{0^k 1^k \mid k \geq 0\}$. We claim that A can be computed in $O(\log n)$ space.

The algorithm to do this is as follows:

- Iterate through the string, character by character
- As long as we see 0's, we increase the counter until we see a 1
- Now, as long as we see 1's, we decrease the counter
- We reject if a 0 comes after a 1, and reject if we ever reach a counter below 0; we accept if we finish reading the input and the counter is exactly 0

This algorithm only needs to store a number of size at most n, and as such requires $O(\log n)$ bits.

Proposition 21.3

SAT can be computed in deterministic $O(n)$ space.

Proof. Suppose we have a CNF formula ϕ. We want to check whether this formula is satisfiable—notice that we only care about the amount of space this algorithm uses.

The naive algorithm is to iterate through all possible assignments $a \in \{0,1\}^m$ over $x_1, \ldots, x_m$. For each assignment, we can just evaluate the clauses and check whether the formula is satisfied.

Here, we need $O(m)$ bits to store the assignment, and another small amount of space during the verification. This gives an algorithm using $O(m) \approx O(n)$ space, since the length of the formula (n) is larger than the number of clauses (m). □

Definition 21.4: SPACE and NSPACE

We have

$$\mathsf{SPACE}(s(n)) = \{A \mid A \text{ is a language decided by a deterministic TM using } O(s(n)) \text{ space}\}$$

and

$$\mathsf{NSPACE}(s(n)) = \{A \mid A \text{ is a language decided by a nondeterministic TM using } O(s(n)) \text{ space}\}.$$

Definition 21.5: Space Complexity Classes

We have

$$\begin{aligned}\mathsf{PSPACE} &= \bigcup_{k=1}^{\infty} \mathsf{SPACE}(n^k)\\ \mathsf{NPSPACE} &= \bigcup_{k=1}^{\infty} \mathsf{NSPACE}(n^k)\\ \mathsf{L} = \mathsf{LogSpace} &= \mathsf{SPACE}(\log n)\\ \mathsf{NL} &= \mathsf{NSPACE}(\log n)\end{aligned}$$

21.2 Small Space Computation

When analyzing machines with bounded space, a key concept is the *configuration graph* of a TM.

Definition 21.6: Configuration Graph

Given a TM M and input x, the graph $G_{M,x}$ is the graph of all possible configurations

$$\text{configuration} = (\text{state}, \text{content of work tape}, \text{head positions}),$$

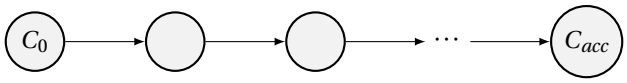
where $C \to C'$ if executing M on C for one step can yield C'.

As a remark, a configuration is a triplet of all the information necessary to resume simulation of a Turing machine. Knowing the state, content of the work tape, and the head positions, we can always continue the simulation of the Turing machine from this point in time.

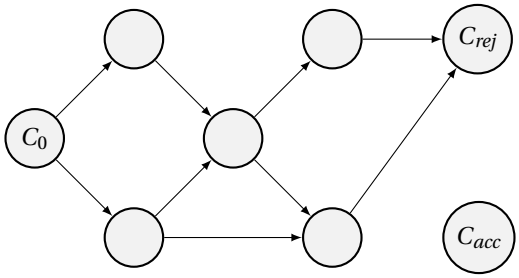
For a deterministic TM, the graph is just a path (since each configuration has only one possible next configuration given a fixed input), but for a non-deterministic TM, we have a more complex graph (since each configuration can have a set of possible next configurations).

Without loss of generality, we can also assume that there is only one accepting configuration and only one rejecting configuration. This means that our goal is to figure out whether the last node is accepting or rejecting; we want to determine whether there is a path from the initial configuration to the accepting configuration.

This connects questions about space complexity to questions about graphs.



(a) Deterministic TM



(b) Nondeterministic TM

Figure 21.2: Examples of configuration graphs for deterministic and nondeterministic Turing machines

Proposition 21.7

Let M be a TM/NTM that uses space $f(n)$. Let $x \in \{0,1\}^n$. Then, $G_{M,x}$ has at most $n \cdot 2^{O(f(n))}$ configurations.

Proof. Let $V_{M,x}$ be the vertices of $G_{M,x}$, i.e. the set of all possible configurations (including ones that are not possible).

Here, we have

$$\begin{aligned} V_{M,x} &= \underbrace{|Q|}_{\text{states}} \cdot \underbrace{|\Gamma|^{f(n)}}_{\substack{\text{work tape}\\ \text{content}}} \cdot \underbrace{n}_{\substack{\text{input head}\\ \text{position}}} \cdot \underbrace{f(n)}_{\substack{\text{work head}\\ \text{position}}} \\ &\leq O(1) \cdot 2^{O(f(n))} \cdot n \cdot f(n) \\ &\leq O(n 2^{O(f(n))}) \end{aligned}$$

□

Proposition 21.8

$\mathsf{NTIME}(f(n)) \subseteq \mathsf{SPACE}(f(n))$

Proof. We can simply run DFS on the computation tree to search for the accepting state.

More formally, if $L \in \mathsf{NTIME}(f(n))$, then there exists some NTM N that decides L in time $f(n)$. We'd like to construct a deterministic TM M for L; this Turing machine can traverse the computation tree for N on some input x, checking whether the path accepts or rejects. If we find some accepting path, the accept; otherwise, if no paths accept, then reject.

Since N runs in time $f(n)$, all paths are of length $O(f(n))$, so the DFS only needs to have $O(f(n))$ memory to keep track of the current traversed path.

This means that M decides L in $O(f(n))$ space, i.e. $L \in \mathsf{SPACE}(f(n))$. □

Note that trivially, $\mathsf{TIME}(f(n)) \subseteq \mathsf{SPACE}(f(n))$, since a deterministic TM that runs in $f(n)$ time can only ever reach $f(n)$ cells.

Proposition 21.9

$\mathsf{SPACE}(f(n)) \subseteq \mathsf{TIME}(n \cdot 2^{O(f(n))})$

Proof. We claim that if $L \in \mathsf{SPACE}(f(n))$, decided by a TM M, then the same machine must run in time $n \cdot 2^{O(f(N))}$.

This is because M cannot repeat the same configuration twice—otherwise, it would loop forever and would not accept. We've shown that the total number of possible configurations is on the order of $n \cdot 2^{O(f(N))}$ for a Turing machine that uses $f(n)$ space, so M must take at most $n \cdot 2^{O(f(n))}$ steps before halting.

This means that $L \in \mathsf{TIME}(n \cdot 2^{O(f(n))})$. □

Proposition 21.10

$\mathsf{NSPACE}(f(n)) \subseteq \mathsf{TIME}(\text{poly}(n) \cdot 2^{O(f(n))})$

Proof. If $L \in \mathsf{NSPACE}(f(n))$, then there exists a NTM that decides L in $f(n)$ space. We'd like to construct a deterministic TM M that decides L in time $\text{poly}(n) \cdot 2^{O(f(n))}$.

The Turing machine M given an input x can construct the configuration graph $G_{N,x}$, which requires time polynomial in the size of the graph, i.e. $\text{poly}(f(n))$. We can then check whether there exists a path from C_0 to C_{acc} (i.e. from the initial configuration to the accepting configuration).

This algorithm would take time polynomial in the number of vertices, i.e.

$$\text{poly}(|V_{N,x}|) = \text{poly}(n \cdot 2^{O(f(n))}) = \text{poly}(n) \cdot 2^{O(f(n))},$$

which dominates the $\text{poly}(f(n))$ term for constructing the graph.

This means that $L \in \mathsf{TIME}(\text{poly}(n) \cdot 2^{O(f(n))})$. □

Note that this algorithm is not efficient in terms of space, since we've constructed the entire graph in memory, but we only care about the time complexity.

With these claims, we now have the following relations:

$$\mathsf{L} \subseteq \mathsf{NL} \subseteq \mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PSPACE} \subseteq \mathsf{NPSPACE} \subseteq \mathsf{EXPTIME} = \bigcup_k \mathsf{TIME}(2^{n^k}).$$

Here is the reasoning for why:

- $\mathsf{L} \subseteq \mathsf{NL}$, since we can only have more power when using non-determinism.
- $\mathsf{NL} \subseteq \mathsf{P}$, since Proposition 21.10 gives

$$\mathsf{NSPACE}(f(n)) \subseteq \mathsf{TIME}(\text{poly}(n) \cdot 2^{O(f(n))}) \implies \mathsf{NSPACE}(\log n) \subseteq \mathsf{TIME}(\text{poly}(n) \cdot 2^{O(\log n)}) = \mathsf{TIME}(\text{poly}(n)).$$

- $\mathsf{NP} \subseteq \mathsf{PSPACE}$, directly from Proposition 21.8.
- $\mathsf{NPSPACE} \subseteq \mathsf{EXPTIME}$, directly from Proposition 21.10.

We don't know whether most of these subsets are strict or not, but we do know that more resources gives us more power—for a reasonable $f : \mathbb{N} \to \mathbb{N}$, such that $f(n) \geq \log(n)$, then

$$\mathsf{SPACE}(f(n)) \neq \mathsf{SPACE}(o(f(n))).$$

For example, $\mathsf{SPACE}(n \log n) \neq \mathsf{SPACE}(n)$. This means that intuitively, you can solve more problems with more space.

Theorem 21.11: Savitch's Theorem

For any $s(n) \geq \log n$, $\mathsf{NSPACE}(s(n)) \subseteq \mathsf{DSPACE}(s(n)^2)$.

Proof. Suppose $L \in \mathsf{NSPACE}(s(n))$, decided by a non-deterministic Turing machine N. We'd like to construct a deterministic TM that decides L.

Here, it suffices to check whether there exists a path from C_0 to C_{acc} in the configuration graph of N, of length at most $n \cdot 2^{O(s(n))}$. We can't do this via a standard BFS or DFS, since these algorithms require space polynomial in the number of vertices, which will be exponential (as there are $n \cdot 2^{O(s(n))}$ possible configurations).

The idea here is to use divide and conquer; to check whether there exists a path from C_0 to C_{acc} of length at most $t = n \cdot 2^{O(s(n))}$, we can iterate through all intermediate vertices (configurations) and check whether there exists a path from C_0 to C_{mid} of length at most $\frac{t}{2}$, and whether there exists a path from C_{mid} to C_{acc} of length at most $\frac{t}{2}$.

Formally, we have the following algorithm $\textsc{Path}(C_1, C_2, t)$ that checks whether there exists a path in $G_{N,x}$ between C_1 and C_2 of length at most t.

- If $t = 1$, return true if and only if C_1 and C_2 are consecutive
- For all $C_{mid} \in V_{N,x}$:
  - If $\textsc{Path}(C_1, C_{mid}, \lfloor \frac{t}{2} \rfloor)$ and $\textsc{Path}(C_{mid}, C_2, \lceil \frac{t}{2} \rceil)$, then return true
- Return false

The base case can be done in small space (since we can simply run the Turing machine for one step to check whether two configurations are consecutive).

We first have the following bound on the number of configurations:

$$\#\text{ configurations} \leq n \cdot 2^{O(s(n))} \leq n \cdot 2^{d \cdot s(n)} \leq 2^{s(n)+d(s(n))} \leq 2^{(d+1)s(n)}.$$

The first bound was shown earlier; the second bound is due to the fact that $f(n) \in O(s(n))$ implies that there exists some constant d such that $f(n) \leq d \cdot s(n)$. The third bound comes from the fact that we're assuming that $s(n) \geq \log n$, so $n \leq 2^{s(n)}$.

This means that we should call $\textsc{Path}(C_0, C_{acc}, 2^{(d+1)\cdot s(n)})$ and return the answer.

To analyze the space complexity, let us define $M(i)$ as the amount of memory used by the subroutine on input $(C_1, C_2, 2^i)$.

We know that in the base case $M(0) = O(s(n))$, since we need $O(s(n))$ bits to describe the configuration, and we have a constant number of next configurations.

For the general case, we need to call the subroutine for $\frac{t}{2} = 2^{i-1}$, which needs $M(i-1)$ memory; we can reuse this space after calling the first subroutine, so we don't need to multiply by two here. In addition, we need to describe C_{mid}, which needs an additional $O(s(n))$ space.

This recurrence solves to $M(i) = O(i \cdot s(n))$. Since we're calling $\textsc{Path}$ with $t = 2^{(d+1)\cdot s(n)}$, we're interested in $M((d+1) \cdot s(n)) = O((d+1) \cdot s(n) \cdot s(n)) = O(s(n)^2)$.

This means that using $\textsc{Path}$ to decide L runs in $O(s(n)^2)$ time, as desired.

Intuitively, the recursive algorithm $\textsc{Path}$ has a depth of $O(s(n))$, since we're halving t each recursive call. Each time, we'd use up $O(s(n))$ space for storing the current configuration, so in total, we'd have $O(s(n)^2)$ space. □

This gives us the hierarchy in Fig. 21.3.

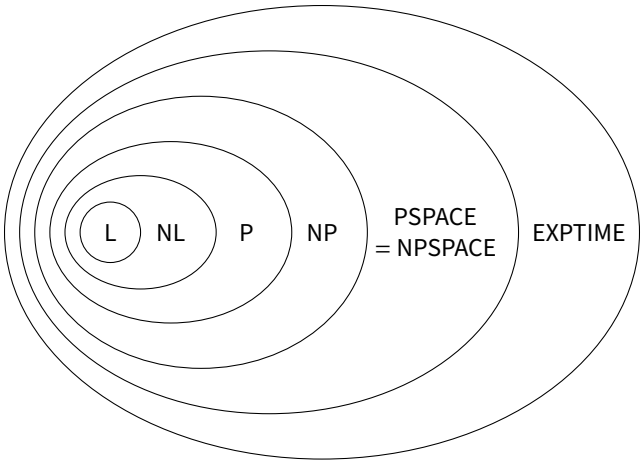


Figure 21.3: Hierarchy of time and space complexity

11/15/2022

Lecture 22

PSPACE-completeness

It's still an open question for whether $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$, and it's also an open question for whether $\mathsf{P} \stackrel{?}{=} \mathsf{PSPACE}$; it's believed that all three classes are distinct, but we don't know for sure. (Though we know that $\mathsf{P} \neq \mathsf{EXPTIME}$, since we're comparing time complexities.)

Today we'll be talking about the hardest problems in PSPACE.

Definition 22.1: PSPACE-hard

A language L is PSPACE-hard if for all $L' \in \mathsf{PSPACE}$, $L' \leq_p L$. That is, all other problems in PSPACE reduce to L in polynomial time.

Definition 22.2: PSPACE-complete

A language L is PSPACE-complete if $L \in \mathsf{PSPACE}$ and L is PSPACE-hard.

Proposition 22.3

Let L be a PSPACE-complete language. Then, $\mathsf{P} = \mathsf{PSPACE}$ if and only if $L \in \mathsf{P}$.

Proof. In the easy direction, suppose $\mathsf{P} = \mathsf{PSPACE}$; then if $L \in \mathsf{PSPACE} = \mathsf{P}$, then we trivially have $L \in \mathsf{P}$.

In the other direction, suppose $\mathsf{L} \in \mathsf{P}$. We want to show that $\mathsf{PSPACE} = \mathsf{P}$. To do this, suppose we have some $L' \in \mathsf{PSPACE}$; it suffices to show that $L' \in \mathsf{P}$. (This means $\mathsf{PSPACE} \subseteq \mathsf{P}$, and we've shown earlier that $\mathsf{P} \subseteq \mathsf{PSPACE}$.)

We know that $L' \leq_p L$, since L is PSPACE-complete. Since $L \in \mathsf{P}$, to solve L', we can simply run the reduction, and then run the algorithm for L. This means that $L' \in \mathsf{P}$ as well. □

This means that the question of $\mathsf{P} = \mathsf{PSPACE}$ boils down to whether we can find some PSPACE-complete language that is in P. The exact same proof can be used to show that $\mathsf{NP} = \mathsf{PSPACE}$ if and only if a PSPACE-complete language is in NP.

This means that the problem of $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$ can also be solved in this way—if we know $\mathsf{P} = \mathsf{PSPACE} = \mathsf{NP}$, then $\mathsf{P} = \mathsf{NP}$.

22.1 Quantified Boolean Formulas

In logic, the quantifiers $\forall$ ("for all") and $\exists$ ("there exists") are used to describe mathematical statements. For example, the statement $\forall x : (x + 1 > x)$ is true.

The statement might also depend on the "universe" that the quantified variables are defined over. For example, $\exists y : (y + y = 3)$ is true over the reals, but false over the natural numbers.

Order of the quantifiers also matters. For example, working over the reals, the expression $\forall x \exists y : (y > x)$ is true; however, it is not true that $\exists y \forall x : (y > x)$.

Definition 22.4: Quantified Boolean Formula

A *quantified boolean formula* (QBF) is a formula with a sequence of quantifiers on $x_1, \dots, x_n$, followed by a boolean formula on $x_1, \dots, x_n$. That is, a QBF takes the form

$$Q_1 x_1 Q_2 x_2 \cdots Q_n x_n (\phi(x_1, \dots, x_n)),$$

where $Q_i \in \{\exists, \forall\}$, and φ is a boolean formula. Here, the universe is $\{0, 1\}$ or $\{\text{False}, \text{True}\}$.

As an example, the following is a QBF:

$$\phi = \forall x \exists y ((x \wedge y) \vee (\overline{x} \wedge \overline{y})).$$

Note that since we've quantified over all of the variables, ϕ will always be either true or false; it does not depend on anything else.

Definition 22.5: Language of True Quantified Boolean Formulas

We define

$$\text{TQBF} = \{\langle \phi \rangle \mid \phi \text{ is a true QBF}\}.$$

Formally,

$$\text{TQBF} = \{\langle \phi \rangle \mid \phi = Q_1 x_1 Q_2 x_2 \cdots Q_n x_n (\varphi(x_1, \dots, x_n)),\ Q_i \in \{\exists, \forall\} \text{ and } \phi \equiv \text{True}\}.$$

Example 22.6

As a first remark, we claim that TQBF is at least as powerful as SAT. This is because any SAT formula can be expressed as a quantified boolean formula and posed as a question for TQBF:

$$\varphi \in \text{SAT} \iff \phi = \exists x_1 \exists x_2 \cdots \exists x_n (\phi) \in \text{TQBF}.$$

This means that in one extreme, where we have only existential quantifiers, we get a NP-complete language.

We can also solve UNSAT with TQBF by simply taking the negation; that is,

$$\varphi \in \text{UNSAT} \iff \phi = \neg \exists x_1 \exists x_2 \cdots \exists x_n (\phi) \in \text{TQBF} \iff \phi = \forall x_1 \forall x_2 \cdots \forall x_n (\neg \phi) \in \text{TQBF}.$$

This means that in the other extreme, where we have only universal quantifiers, we get a coNP-complete language.

What we're interested in with TQBF is like an intermediate between these two extremes, where we can possibly have mixed existential and universal quantifiers.

We claim that TQBF is PSPACE-complete; this is the canonical PSPACE-complete language, much like 3SAT is the canonical NP-complete language.

Firstly, we can notice that any QBF in TQBF can be written in the following "nice" form, alternating between existential and universal quantifiers:

$$\phi = \exists x_1 \forall x_2 \exists x_3 \forall x_4 \cdots \exists x_{m-1} \forall x_m (\varphi(x_1, \ldots, x_m)),$$

for an even m. To do so, we can simply add dummy variables and quantifiers in between consecutive existential quantifiers or in between consecutive universal quantifiers.

We first show that TQBF can be solved in PSPACE. Observe that to determine whether $\phi \in$ TQBF, we want to evaluate the following tree of depth m shown in Fig. 22.1.

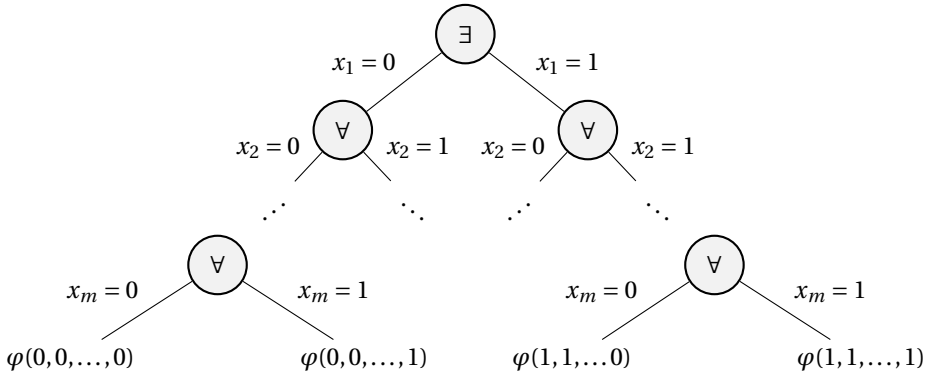


Figure 22.1: Tree to evaluate for a QBF

To evaluate this tree for some formula ϕ, we have the following algorithm EVALTREE(ϕ):

- If the top node is $\exists x_i$, then we know $\phi = \exists x_i \psi(x_i)$ for some other QBF ψ.
  - Run EVALTREE($\psi(0)$) and EVALTREE($\psi(1)$), i.e. substitute $x_i \leftarrow 0$ and $x_i \leftarrow 1$ respectively
  - If either evaluation is true, then accept.
- If the top node is $\forall x_i$, then we know $\phi = \forall x_i \psi(x_i)$ for some other QBF ψ.
  - Run EVALTREE($\psi(0)$) and EVALTREE($\psi(1)$).
  - IF both evaluations are true, then accept.
- If there are no quantifiers, then ψ consists only of constants, so accept only if $\phi \equiv$ True.

This algorithm needs only $O(1)$ memory per level of recursion (i.e. storing the value of x_i), so over all m levels of recursion, we only need $O(m)$ memory, which means that we can decide TQBF with polynomial space; TQBF $\in$ PSPACE.

Next, we want to show that TQBF is PSPACE-hard. That is, for any $L \in$ PSPACE, $L \leq_p$ TQBF.

Let $L \in$ PSPACE, and let M be a Turing machine using space $O(n^c)$ that decides L. Given an input x, we want to compute a mapping $f(x)$ in polynomial time such that $x \in L \iff f(x) \in$ TQBF.

WLOG suppose the configuration graph $G_{M,x}$ has only two final configurations: $C_{acc} = (q_{acc}, "", (1,1))$ and $C_{rej} = (q_{rej}, "", (1,1))$. That is, the final configurations are when the Turing machine is at the accept/reject states, the work tape is blank, and the heads are at the beginning of both the input tape and work tape.

This means that we have

$$x \in L \iff \text{there exists a path of length} \leq 2^{O(n^c)} \text{ from } C_0 \text{ to } C_{acc}..$$

We want to express the RHS as a polynomial-length QBF; this will be our mapping f.

As a first attempt, we can define $\text{PATH}(C_1, C_2, i)$ representing whether there exists a path of length $\leq 2^i$ from C_1 to C_2. This means that we can rewrite

$$x \in L \iff \text{PATH}(C_0, C_{acc}, O(n^c)).$$

Defining this recursively, in a similar way that we proved Savitch's theorem (Theorem 21.11), we have

- $\text{PATH}(C_1, C_2, 0) = 1$ if and only if $C_1 = C_2$ or $C_1 \to C_2$ (i.e. C_1 and C_2 are adjacent in $G_{M,x}$).

  This can be checked with a polynomial size boolean formula, in much the same way as we did in showing CIRCUITSAT was NP-complete.

- $\text{PATH}(C_1, C_2, i) = \exists C_{mid} : \text{PATH}(C_1, C_{mid}, i-1) \wedge \text{PATH}(C_{mid}, C_2, i-1)$

  That is, we can express the existence of a path as determining if there exists a middle node C_{mid} and there exists a path from C_1 to C_{mid} and from C_{mid} to C_2.

The issue with this recurrence though is that it blows up. The QBF will have an exponential size (i.e. of size $2^{O(n^c)}$), since we're just quantifying the entire path.

Recall that the way Savitch's algorithm worked is that we can reuse the memory when computing the subproblems—can we do the same here? It turns out that we can.

The solution is to replace the $\wedge$ with a $\forall$ quantifier over two possibilities:

$$\begin{aligned}\text{PATH}(C_1, C_2, i) &= \exists C_{mid} \forall (C_3, C_4) \in \{(C_1, C_{mid}), (C_{mid}), C_2)\} : \text{PATH}(C_3, C_4, i-1) \\ &= \exists C_{mid} \forall C_3 \forall C_4 : ((C_3 = C_1 \wedge C_4 = C_{mid}) \vee (C_3 = C_{mid} \wedge C_4 = C_2)) \implies \text{PATH}(C_3, C_4, i-1)\end{aligned}$$

The second equivalence expands a little bit to get the quantifiers to be in our desired form. The boolean formula can be expanded further as well to eliminate the $=$ and $\implies$, but is omitted here for brevity.

With this new recurrence, the formula no longer blows up; we do only one recursive step. Analyzing this recurrence, let $\ell(i)$ be the length of the formula for $\text{PATH}(C_1, C_2, i)$.

We know that $\ell(0) = O(n^c)$ to specify the configurations of C_1 and C_2 (along with some other smaller space to simulate one step of M). In general, $\ell(i) = \ell(i-1) + O(n^c)$; we perform the recursion, and the additional quantifiers use $O(n^c)$ bits to specify C_1, C_2, C_{mid}, etc.

This recurrence solves to $\ell(i) = O(i \cdot n^c)$, so $\text{PATH}(C_0, C_{acc}, O(n^c))$ has length $O(n^c \cdot n^c) = o(\text{poly}(n))$. This means that we've successfully shown a polynomial time reduction from L to TQBF, showing that TQBF is indeed PSPACE-hard, and thus PSPACE-complete.

22.2 TQBF Game

Given a boolean formula $\varphi(x_1, x_2, \ldots, x_m)$, for even m, Alice and Bob plays the following game. For $i = 1, 2, \ldots, m$, if i is odd, then Alice picks x_i; if i is even, then Bob picks x_i. Alice wins if and only if $\varphi(x_1, x_2, \ldots, x_m)$ is true.

A *strategy* for this game defines the next move given any sequence of prior moves. Alice has a *winning strategy* if no matter how Bob plays, Alice can ensure that she wins.

Example 22.7

Suppose we have $\varphi(x_1, x_2, x_3, x_4) = (x_1 \vee x_2) \wedge (x_2 \vee x_3) \wedge (\bar{x}_2 \vee \bar{x}_3)$. Who has the winning strategy?

Note that it must be the case that either Alice has a winning strategy, or Bob has a winning strategy; we'll show this along the way later.

If we want to satisfy this formula (i.e. we want to find a winning strategy for Alice), then for x_1, Alice should pick `true`, to satisfy the first clause. Now, for x_2, note that it also appears in the next two clauses. Bob can't do anything about the first clause, but no matter what Bob chooses for x_2, Alice can choose the opposite to satisfy both clauses.

That is, if Bob chooses x_2 to be `true`, then the second clause is already satisfied, and Alice can choose x_3 to

be `false` to satisfy the third clause. If Bob chooses x_2 to be `false`, then the third clause is already satisfied, and Alice can choose x_3 to be `true` to satisfy the second clause.

This means that Alice has a winning strategy for this specific φ.

Proposition 22.8

We claim that Alice has a winning strategy if and only if the QBF $\exists x_1 \forall x_2 \exists x_3 \cdots \forall x_m \varphi(x_1, \ldots, x_m) \equiv \texttt{true}$.

Proof. This intuitively holds, because a winning strategy means that there is some x_1 that Alice chooses, such that for any x_2 that Bob chooses, there exists an x_3 that Alice can choose, etc., such that in total the formula is `true`.

More formally, we can use induction on m to show that if the QBF is `true`, then Alice has a winning strategy. Suppose Alice has a winning strategy for m variables; we want to show that Alice has a winning strategy for the QBF in $m+2$ variables.

Here, the QBF in $m+2$ variables would look like the following:

$$\exists x_1 \forall x_2 \exists x_3 \forall x_4 \cdots \exists x_{m+1} \forall x_{m+2} \varphi(x_1, \ldots, x_{m+2}) \equiv \texttt{true}.$$

If this holds, then suppose we fix x_1 and x_2; this gives us a smaller QBF

$$\exists x_3 \forall x_4 \cdots \exists x_{m+1} \forall x_{m+2} \varphi(x_1, x_2, \ldots, x_{m+2}) \equiv \texttt{true}$$

in only m variables—by the IH, we know that since this QBF is `true`, Alice has a winning strategy after her second move. This means that if she picks our fixed x_1 and Bob picks any x_2, then she will always win—this is a winning strategy for Alice in $m+2$ variables.

In the other direction, we want to show that if Alice has a winning strategy, then the QBF is true, equivalently, we can show that if the QBF is false, then Bob has a winning strategy.

This means that we have

$$\forall x_1 \exists x_2 \forall x_3 \cdots \forall x_{m-1} \exists x_m \varphi(x_1, \ldots, x_m) \equiv \texttt{false},$$

and by a very similar reasoning, we can inductively show that Bob has a winning strategy if this is the case. □

22.3 Generalized Geography

Given a digraph G and a start node s, Alice and Bob take alternating steps, where in each step we extend the path by one edge, maintaining a simple path, until there are no possible next steps. The player who is "stuck" loses.

Let us define

$$\text{GG} = \{\langle G, s\rangle \mid \text{Alice has a winning strategy in the generalized geography game defined by } G \text{ and } s\}.$$

Theorem 22.9

GG is PSPACE-complete.

Proof. We proceed by a reduction TQBF $\leq_p$ GG.

A first observation is that TQBF' is PSPACE-complete with the modification

$$\text{TQBF}' = \{\langle \phi \rangle \mid \phi = \exists x_1 \forall x_2 \cdots \forall x_m (\psi(x_1, \ldots, x_m)),\ \psi \text{ is a 3CNF}, \phi \equiv \texttt{true}\}.$$

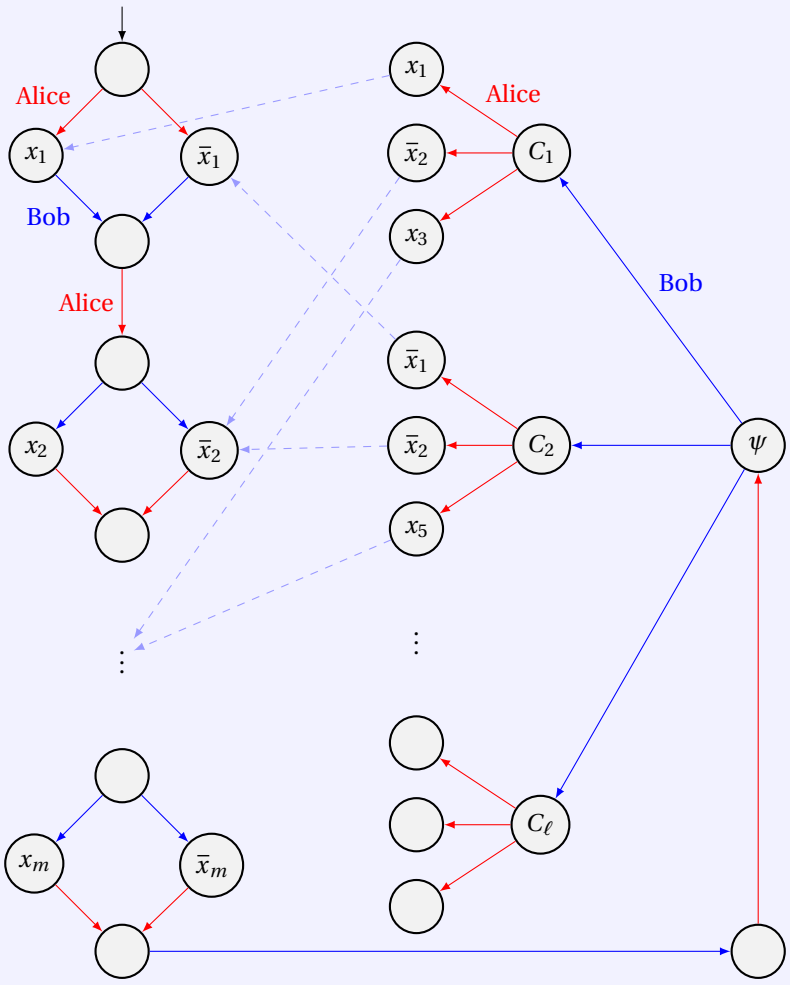
This is because we can convert any boolean formula $\varphi(x_1, \ldots, x_m)$ into 3CNF form, which gives our reduction from TQBF to TQBF'.

Next, we want a reduction TQBF$' \leq_p$ GG; that is, given

$$\phi = \exists x_1 \forall x_2 \cdots \forall x_m \psi(x_1, \ldots, x_m),$$

where ψ is a 3CNF of the form $\psi = C_1 \wedge C_2 \wedge \cdots \wedge C_\ell$, we want to construct a graph G_ϕ such that $\langle \phi \rangle \in$ TQBF$'$ if and only if $\langle G_\phi, s \rangle \in$ GG. This would show that Alice has a winning strategy if and only if $\phi \equiv$ `true`.

The graph would be as follows:



Here, the left side of the graph is a sequence of vertices forcing Alice to choose the value of x_1, then forcing Bob to choose the value of x_2, etc. If the path goes through the x_i node, then we assign $x_i =$ `true`; otherwise if the path goes through the $\bar{x}_i$ node, then we assign $\bar{x}_i =$ `true`.

When we get through the entire left of the graph, we get to the node for ϕ, and it's Bob's turn to choose an edge. This part of the graph is a tree allowing Bob to choose a clause, then Alice to choose a literal in the clause, followed by Bob choosing a forced edge. Here, the game will always end after either 2 or 3 more turns; let's analyze what the next turns will consist of.

Suppose the QBF was true; we want to show that Alice has a winning strategy. Since the QBF is true, this means that no matter what Bob chooses, there will always exist some set of assignments for Alice's variables that satisfies the 3CNF formula.

As such, no matter which clause Bob chooses at the end, Alice can always choose a literal that has been assigned `true`. At this point, it will be Bob's turn with only one possible next edge—this edge will always go to a vertex that is on the path (as the literal is connected to a node in the left part of the graph, which we used to determine which literals are true in the assignment). This means that Bob will always get stuck, and Alice wins the game; Alice always has a winning strategy.

Suppose on the other hand that the QBF was false; we want to show that Bob has a winning strategy (i.e. Alice does not have a winning strategy). Since the QBF is false, this means that no matter what Alice chooses, there will always exist some set of assignments for Bob's variables that makes the 3CNF formula false—some clause will be left unsatisfied.

As such, Bob can pick this clause that is not satisfied, and no matter which literal Alice picks, it will never be part of the path in the left side of the graph (if it had been, then the literal would have been assigned `true`, and the clause would have been satisfied). This means that Bob can still choose a valid vertex in his next turn, but this leaves Alice with no other vertices to go to. As such, Alice will always get stuck, and Bob wins the game; Bob always has a winning strategy. □

11/17/2022

Lecture 23

Logarithmic Space

Recall that we defined $\mathsf{L} = \mathsf{LogSpace} = \mathsf{SPACE}(\log n)$, and we defined $\mathsf{NL} = \mathsf{NSPACE}(\log n)$. We showed that $\mathsf{L} \subseteq \mathsf{NL} \subseteq \mathsf{P}$.

Currently, it's still an open question of whether $\mathsf{P} = \mathsf{L}$, or whether $\mathsf{NL} = \mathsf{L}$, or whether $\mathsf{NL} = \mathsf{P}$.

Today, we'll talk about log-space reductions. That is, we want a notion of reductions such that if $A \leq_{logspace} B$, and $B \in \mathsf{L}$, then $A \in \mathsf{L}$.

Since reductions are functions mapping instances of problem A to instances of problem B, we need a notion of log-space computable functions.

However, if we think about reductions, they naively use a lot of space—we have to somehow write the output, and if we use the work tape to write the output, then we'll never be able to use only logarithmic space (as we'd need to write $O(n)$ space just for the output).

This means that we'll need to modify the model a little bit for log-space reductions.

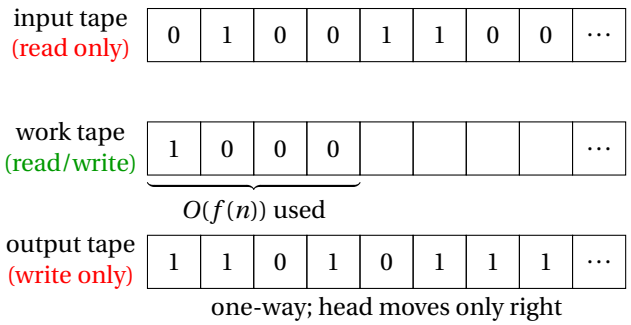


Figure 23.1: Three-tape Turing machine for reductions in space complexity

Here, the only difference is that we now have an extra output tape. To prevent exploiting this extra tape, we make

the tape write only, with the head moving only to the right; this means that the output tape is only used for the final output and nothing else.

Example 23.1

Suppose we want to flip all the bits of the input. How much space do we need?

In this case, we don't even need the work tape; we can read from the input tape and write the flipped bit directly to the output tape.

Example 23.2

Suppose we want to reverse a string. How much space do we need?

Here, we also do not need the work tape; we can first move to the end of the input tape, and move left in the input tape while copying the bits to the output tape.

Definition 23.3: Space computable

A function $f : \Sigma^* \to \Sigma^*$ is $\mathsf{SPACE}(s(n))$ computable if there exists a deterministic Turing machine M which on input x outputs $f(x)$.

Here, M has three tapes: a read-only input tape, a read-write work tape that uses $O(s(n))$ space, and a write-only one-way output tape.

As a remark, anything we can do in logarithmic space, we can do in polynomial time. This is because anything that uses logarithmic space must only have a polynomial number of configurations (otherwise, we'll never halt). In the worst case, we'd only go through a polynomial number of configurations, which takes polynomial time.

Definition 23.4: Log-space reduction

A language A log-space reduces to a language B, denoted as $A \leq_{\mathsf{L}} B$, if there exists a log-space computable $f : \Sigma^* \to \Sigma^*$ such that $x \in A \iff f(x) \in B$.

Theorem 23.5

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

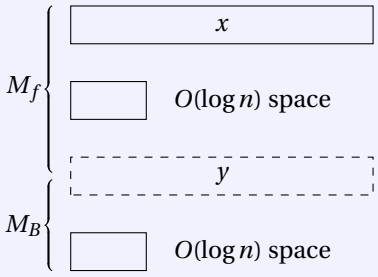
Proof. Suppose there is a log-space reduction $f : \Sigma^* \to \Sigma^*$ from A to B. Let the Turing machine that computes f be M_f. Suppose also that there is a log-space Turing machine M_B that decides B.

We want to show that there exists a log-space Turing machine M_A that decides A. This machine will do the following:

1. Compute $y = f(x)$ using M_f
2. Simulate M_B on y

However, this will not work—we have nowhere to store y. To solve this problem, we will lazily evaluate $y = f(x)$.

Visually, we have the following chain of machines:



Here, M_f takes in the input x and outputs $y = f(x)$ using a small $O(\log n)$ space. We then use y as input to M_B, which using a small $O(\log n)$ space decides B (and thus deciding A).

The idea is that we'll treat the intermediate y as a virtual buffer; since M_B is a Turing machine, there will be an associated pointer in its input buffer for y, and it'll query for a bit in the buffer. This means that we want some way to return the ith index of y in a small amount of space (i.e. $O(\log n)$ space).

To do this, we can simulate M_f and simply ignore all the output and keep a counter for how many bit it has tried to print so far. If we want the ith bit, we can ignore all output until the counter gets to i, and then we stop and return the ith bit.

In total, we need the work space for M_f, the work space for M_B, and the index i in computing the ith bit of y. Each of these auxiliary work spaces takes $O(\log n)$ space, so in total we have a reduction from A to B in $O(\log n)$ space. □

Theorem 23.6

If $A \leq_{\mathsf{L}} B$ and $B \leq_{\mathsf{L}} C$, then $A \leq_{\mathsf{L}} C$.

Proof. This proof follows in a very similar manner to the previous proof, using a virtual buffer and computing only the ith bit of the intermediate output. □

23.1 s-t Connectivity

Definition 23.7: NL-complete

A language B is NL-complete if $B \in \mathsf{NL}$, and for every other $A \in \mathsf{NL}$, we have $A \leq_{\mathsf{L}} B$.

The s-t connectivity problem (also known as PATH) will be our canonical example of an NL-complete language. Here, given a directed graph $G = (V, E)$ and two vertices $s, t \in V$, we want to determine whether there exists a path from s to t in G.

Formally, we have

$$\text{STCONN} = \{\langle G, s, t\rangle \mid G = (V, E) \text{ is a directed graph, } s, t \in V, \text{ there exists a path in } G \text{ from } s \text{ to } t\}.$$

Firstly, we can see that STCONN $\in$ P; we have many algorithms that can traverse the graph in polynomial time and check whether there exists a path from s to t (ex. BFS, DFS).

Next, we claim that STCONN $\in$ NL.

Theorem 23.8

stCONN $\in$ NL.

Proof. Here, we have to be a little careful on what we store, since even storing a single vertex will take up $O(\log n)$ space, since the encoding for a vertex depends on how many vertices there are.

Utilizing non-determinism, it turns out that we can decide stCONN while only storing one vertex in our work tape. To do so, we can simply guess which vertex to go to next, with a limit of $m = |V|$ possible vertices in our path.

If there exists a path from s to t, it must have length at most $m = |V|$, so there must exist some computation path in the NTM that guesses all the correct vertices and edges to follow to get from s to t. On the other hand, if there does not exist a path from s to t, then no computation path through this graph will ever reach t in at most $m = |V|$ steps.

This means that $\langle G, s, t\rangle \in$ stCONN if and only if this NTM accepts.

How much space does this machine take? We only need to store the current vertex in our memory in order to be able to guess what the next vertex should be; this takes $O(\log n)$ space. We also need to remember how long our path currently is (because we want to stop when our path is $|V|$ steps long), which also takes $O(\log n)$ space. This means that we can decide stCONN with logarithmic space, so stCONN $\in$ NL. □

A direct result of Savitch's theorem then says that since stCONN $\in$ NSPACE$(\log n)$, then stCONN $\in$ SPACE$(\log^2 n)$.

Note that this algorithm does not run in polynomial time; it takes $2^{O(\log^2 n)} = n^{O(\log n)}$ time. This means that there's a tradeoff in using small amount of space or a small amount of time. It's an open question whether we there is an algorithm that uses $O(\log^2 n)$ space and poly(n) time, and it's even an open question for whether there is an algorithm that uses sublinear $O(n^{0.99})$ space and poly(n) time.

A natural followup question would be: can we do better than $O(\log^2 n)$? Is stCONN $\in$ L?

It turns out that this is a very hard question to answer, and we don't know; if we can show that stCONN $\in$ L, then we'd have shown that L = NL. This is because stCONN is NL-complete, as we'll show next.

Theorem 23.9

stCONN is NL-complete.

Proof. We've already showed that stCONN $\in$ NL, so it suffices to show that every other language in NL reduces to stCONN. The idea is to use configuration graphs.

Let $A \in$ NL, and let M be the NTM using space $O(\log n)$ that decides A. We know that an input $x \in A$ if and only if M accepts x, which happens if and only if there exists a path in the configuration graph $G_{M,x}$ from C_{init} to C_{acc}.

This means that we want to map to $\langle G_{M,x}, C_{init}, C_{acc}\rangle$; we now just need to show that this mapping takes log-space.

First, we can show that the configuration graph is of size polynomial in n. We know that if $G_{M,x} = (V_{M,x}, E_{M,x})$, then the number of configurations is $|V_{M,x}| \le n2^{O(\log n)}$ (there are a constant number of states, n possible positions for the head of the input tape, and the work tape is of size $O(\log n)$ so there are $2^{O(\log n)}$ possible contents of the work tape).

This means that $|V_{M,x}| \le$ poly(n), and the graph is of size polynomial in n.

Next, we want to show that we can compute $G_{M,x}$ in logarithmic space. To do this, suppose we represent our graph as an adjacency matrix, with a 1 in index (i, j) if $(C_i, C_j) \in E_{M,x}$. It suffices to show that we can compute every element of the adjacency matrix in logarithmic space; computing the entire matrix would

consist of chaining a bunch of these computations together, each using log-space.

To check whether (C_i, C_j) are adjacent, we can simply load C_i in memory (taking $O(\log n)$ space, since there are $\text{poly}(n)$ vertices in the graph, so an encoding would take $\log(\text{poly}(n)) = O(\log n)$ space), run the Turing machine for one step, and check whether we reached C_j. Running the Turing machine also takes at most $O(\log n)$ space, so in total we've only used $O(\log n)$ space to compute the (i, j)th entry in the adjacency matrix.

In total, this means that we're iterating through all i and j, computing the (i, j)th entry in the adjacency matrix in $O(\log n)$ space, using a total of $O(\log n)$ space to create the mapping. □

23.2 NL and coNL

Definition 23.10: coNL

We define

$$\text{coNL} = \{\overline{A} \mid A \in \text{NL}\},$$

where $\overline{A} = \{x \in \Sigma^* \mid x \notin A\}$.

It turns out that NL = coNL; this is a theorem by Immerman–Szelepcsenyi.

In other words, NL is closed under complement. More generally, for all space constructible $s(n) \geq \log(n)$, we have $\text{NSPACE}(s(n)) = \text{coNSPACE}(s(n))$.

Before we prove this theorem, let us give an alternative definition of NL. Recall that we had an alternative definition of NP involving verification of a certificate (or witness) in polynomial time. That is, if a language is in NP, then we there exists a certificate that can be verified in polynomial time, and otherwise no certificate is verifiable.

Similarly, we can define NL through this idea of verifying certificates. Here, we have our input tape and work tape, defined normally, but instead of an output tape, we have a read-once one-way certificate tape. That is, the certificate tape can only be read once, and the head only moves to the right.

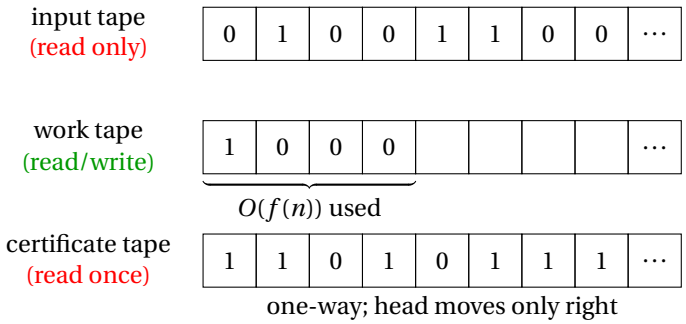


Figure 23.2: Alternative definition of NL with read-only certificates

Definition 23.11: NL (alt.)

A language $A \in \text{NL}$ if and only if there exists a deterministic Turing machine M using logarithmic space with a read-once certificate tape such that $x \in A \iff \exists y : M(x, y)$ accepts.

To show that NL = coNL, it suffices to show that $\overline{\text{STCONN}} \in \text{NL}$.

To see why, since STCONN is NL-complete, this means that $\overline{\text{STCONN}}$ is coNL-complete. That is, all $A \in \text{coNL}$ has a reduction $A \leq_{\text{L}} \overline{\text{STCONN}}$. If we've shown that $\overline{\text{STCONN}} \in \text{NL}$, then A reduces to a language in NL, so $A \in \text{NL}$ as well.

This would show $\mathsf{coNL} \subseteq \mathsf{NL}$.

In the other direction, if we've shown that $\overline{\text{STCONN}} \in \mathsf{NL}$, then $\text{STCONN} \in \mathsf{coNL}$, and since STCONN is NL-complete, this means that every language $A \in \mathsf{NL}$ reduces to $\overline{\text{STCONN}} \in \mathsf{coNL}$, and thus $\mathsf{NL} \subseteq \mathsf{coNL}$.

Together, this would show that $\mathsf{NL} = \mathsf{coNL}$.

As such, the crucial question is: how do we show $\overline{\text{STCONN}} \in \mathsf{NL}$?

Formally, recall that we have

$$\overline{\text{STCONN}} = \{\langle G, s, t\rangle \mid G \text{ is not a graph or } s, t \notin V \text{ or there is no path from } s \text{ to } t \text{ in } G\}.$$

We can easily check whether G is a graph, or whether s and t are vertices in G, so let us assume our input is valid, and the only point of contention is whether there exists a path from s to t in G. That is, to show that $\overline{\text{STCONN}} \in \mathsf{NL}$, we want to verify in log-space using read-once certificates that s and t are not connected.

Example 23.12

As a warm-up, how would we prove that there *is* a path from s to t of length at most k in G?

Our certificate can simply be the path $(v_0 = s, v_1, \ldots, v_\ell = t)$ for some $\ell \le k$. The verifier can then verify that there is an edge from v_i to v_{i+1} in the graph.

It's a little bit trickier to prove that there *is no path* from s to t of length at most k in G.

For each $i = 0, 1, \ldots, m = |V|$, let us define

$$A_i = \{v \in V \mid \text{there is a path of length} \le i \text{ from } s \text{ to } v \text{ in } G\}.$$

Here, we'd have

$$\begin{aligned} A_0 &= \{s\} \\ A_1 &= \{s\} \cup \{\text{neighbors of } s\} \\ &\vdots \\ A_m &= \{v \in V \mid \text{there is a path from } s \text{ to } v\} \end{aligned}$$

The idea is that we'll design certificates for:

1. $v \notin A_i$, assuming we already know $|A_i|$
2. $|A_i| = c_i$, assuming we already know $|A_{i-1}|$

That is, we'll be building up our knowledge of the sets A_i inductively; we start out with some information about A_0 (i.e. its size), and in the end, we'll be able to say something about A_m that allows us to prove that $t \notin A_m$, i.e. there is no path from s to t.

These certificates will be given one after the other; at any point, we'll keep track of i and $|A_i|$. Lastly, we'll give a certificate that $v \notin A_m$, given $|A_m| = c_m$ (verified in the previous certificate).

Let's now go through the two points above:

1. First, we want to give a certificate to prove that $v \notin A_i$, given $|A_i|$. This certificate would just be the set of vertices $u \in A_i$ (in ascending order), each with another certificate to prove that $u \in A_i$.

   The certificate to show $u \in A_i$ would just be the path from s to u. We also give these vertices in ascending order to prove that we have no duplicates (otherwise, we could have a duplicate at the beginning and the end, and the verifier cannot remember all the vertices to check).

   With these certificates, we can verify the following:

- There are $|A_i|$ certificates.

  This can be done in $O(\log n)$ space by keeping track of a count.

- Each certificate proves that $u \in A_i$ (i.e. each path is valid).

  This can be done in $O(\log n)$ space by traversing the path and verifying edges, keeping track of only the current vertex from s until we reach u.

- v is not in this list.

  This can be done in no additional space by comparing v to the vertex in the certificate..

- The list is sorted

  This can be done in $O(\log n)$ space by comparing the vertex in the previous certificate to the current vertex.

This shows that $v \notin A_i$, since we are given the exact vertices that *are* in A_i, and none of them are v. In addition, we can verify everything in $O(\log n)$ space as well.

1.5 Before we move on to showing $|A_i| = c_i$, we need an intermediate certificate; we want to show that $v \notin A_i$, assuming we know $|A_{i-1}|$.

The certificate would be the same as before; we can list all the vertices in A_{i-1} in ascending order, and provide certificates to attest that $u \in A_{i-1}$.

With these certificates, we can again verify the same things, but now we check that v is not in the list *and* that for every u in the list, there is no edge from u to v. This would ensure that it is impossible to get to v using another edge in the graph, so this shows that $v \notin A_i$.

2. Now, we want to show that $|A_i| = c_i$, assuming we know $|A_{i-1}| = c_{i-1}$

   Here, we will give a certificate for each $v \in V$ in ascending order; each certificate would either show $v \in A_i$ or $v \notin A_i$. If $v \in A_i$, we'd just give the path from s to v, and otherwise we can use the certificate in the previous point.

   The verifier can then verify each certificate (i.e. verify the path if $v \in A_i$, and verify using the previous point if $v \notin A_i$), and count up the number of positive certificates (i.e. count the number of certificates that show some $v \in A_i$). This count will show $|A_i| = c_i$. Again, everything here can be done in logarithmic space.

This would conclude the proof—these are all the certificates necessary to show that there is no path from s to t. In summary, we'd give certificates for the following:

- $|A_0| = 1$ since it is always the case that $A_0 = \{s\}$
- $|A_1| = c_1$, using certificates for $v \in A_1$ or $v \notin A_1$, assuming $|A_0| = 1$
- $|A_2| = c_2$, using certificates for $v \in A_2$ or $v \notin A_2$, assuming $|A_1| = c_1$
- etc.
- $|A_m| = c_m$, using certificates for $v \in A_m$ or $v \notin A_m$, assuming $|A_{m-1}| = c_{m-1}$
- $t \notin A_m$

Throughout this entire process, the verification of each certificate takes only $O(\log n)$ space, and in the end, we would have verified that $t \notin A_m$, i.e. there is no path from s to t of length $m = |V|$. This means that $\overline{\text{STCONN}} \in \mathsf{NL}$.

11/22/2022

Lecture 24

Circuit Complexity

Recall our definition of a boolean circuit (Definition 18.7) from a previous lecture; it's a DAG of gates, with n input variables and one output gate. A boolean circuit naturally defines a function $f : \{0,1\}^n \to \{0,1\}$, since our inputs are our n variables $x_1, \ldots, x_n$, and our output is the value of the output gate.

We also define the *size* of a boolean circuit to be the number of gates in the circuit, and we define the *depth* of a boolean circuit to be the maximum number of gates on any path from the input to the output.

Recall also that we've shown in Proposition 19.1 that any boolean function $f : \{0,1\}^n \to \{0,1\}$ can also be represented as a boolean circuit of size $O(n2^n)$.

It turns out that through an inductive argument, this bound can be lowered to $O(2^n)$, and a slightly more clever argument shows that this bound can be lowered even further to $O(2^n/n)$.

The bound of size $O(2^n/n)$ is also the best we can do for almost all functions, from the following theorem by Shannon.

Theorem 24.1

Most (at least 99%) of boolean functions $f : \{0,1\}^n \to \{0,1\}$ have circuit size $\Omega(2^n/n)$.

Proof. We can prove this through a counting argument.

There are a total of 2^{2^n} possible boolean functions from $\{0,1\}^n \to \{0,1\}$; looking at the truth table for the function, we have 2^n rows, and each row can have a function value of either 0 or 1.

On the other hand, how many circuits of size s on $x_1, \ldots, x_n$ exist? We'll give an upper bound on this count.

We can specify a circuit by listing its gates in topological order; for each gate, we can specify its type (i.e. AND, OR, NOR) and also specify which previous gates feed it as incoming wires (i.e. what its inputs are).

For each gate, we have 3 options for the type, and we have $s+n$ options for each input. This means that there are $3(s+n)^2$ ways to specify a gate. With s gates, we have $(3(s+n)^2)^s = O(s)^{2s} = 2^{2s\cdot\log(O(s))}$ possible circuits (if $s \gg n$).

Here, when $s = 2^n/100n$, this value is much much smaller than 2^{2^n}. This means that almost all functions have circuit size at least $2^n/n$; combined with the upper bound from before, this means that almost all functions have circuits of size on the order of $2^n/n$. □

Next, we want to be able to associate languages with boolean functions (and thus with boolean circuits).

Consider a language $L \subseteq \{0,1\}^n$. We can identify L with a sequence of boolean functions $f_0, f_1, f_2, \ldots$, where $f_n : \{0,1\}^n \to \{0,1\}$ is defined as

$$(\forall n \forall x \in \{0,1\}^n) f_n(x) = \begin{cases} 1 & \text{if } x \in L \\ 0 & \text{otherwise} \end{cases}$$

In particular, we have (countably) infinitely many boolean functions, and each function f_n tells us exactly which inputs of length n are in the language L.

This view on languages leads us to a notion of "uniform" and "non-uniform" models of computation.

With a *uniform* model of computation, we have one machine that decides the entire language (ex. Turing machines). With a *non-uniform* model of computation, we have one machine *per input length* (ex. circuits, one for each input length as we just described).

We're interested in how we can convert between these two models of computation; we'd need an infinite number of circuits to decide languages (at least one per input length), as opposed to the one Turing machine we've been

talking about so far.

24.1 The class P/poly

We've seen that every boolean function can be computed with circuits of size on the order of 2^n, but we're more interested in looking at functions that can be computed with small circuits, i.e. circuits of size polynomial in n.

Definition 24.2: SIZE$(s(n))$

Given a function $s : \mathbb{N} \to \mathbb{N}$, a language $L \subseteq \{0,1\}^*$ is in $\mathsf{SIZE}(s(n))$ if and only if there exists a family of boolean circuits $\mathcal{C} = \{C_i\}_{i=1}^{\infty}$ that decides L (i.e. for inputs of length i, C_i decides whether the input is in L), and where the size of C_n is at most $s(n)$, for all n.

Note that this definition has no $O(s(n))$ like we had before when talking about Turing machines; we're working with very precise definitions of circuits and circuit sizes, so constant factors like $3n$ or $10n$ are meaningful here.

Definition 24.3: P/poly

We define

$$\mathsf{P/poly} = \bigcup_{k=1}^{\infty} \mathsf{SIZE}(n^k).$$

That is, P/poly is the set of functions that can be computed with a circuit of size polynomial in n.

Recall that most functions require size $\Omega(2^n/n)$; as such, which languages are actually in P/poly?

Theorem 24.4

$\mathsf{P} \subseteq \mathsf{P/poly}$. That is, all problems that are decidable in polynomial time can be decided by a circuit of polynomial size.

In particular, if $L \in \mathsf{TIME}(T(n))$, then $L \in \mathsf{SIZE}(O(T(n))^2)$.

Proof. We've already shown this when proving that CIRCUITSAT is NP-complete with the Cook–Levin theorem. See Section 19.2.1 for more details.

The essence of the proof is that we can write out a configuration table for the Turing machine for L, where each row is a snapshot of the state of the Turing machine at a given moment in time. Each row would contain the contents of the tape, the head location, and the current state of the finite control. An example is shown in Fig. 19.2.

The configuration table would have at most $O(T(n))$ rows, since the Turing machine runs for at most $O(T(n))$ steps. Further, this means that the Turing machine will never reach beyond $O(T(n))$ cells on the tape, so there are at most $O(T(n))$ columns in the table. This means that any run of the Turing machine on a given input can be specified in space $O(T(n))^2$.

The content of each cell can also be identified with a constant size circuit (i.e. STEP), so in total, the entire Turing machine on an input x of length n can be simulated with a circuit of size $O(T(n))^2$ (as shown in Fig. 19.3). □

As a remark, it can actually be shown that any Turing machine that runs in time $T(n)$ can be simulated with a circuit of size $T(n)\log(T(N))$. The proof involves converting any Turing machine into an *oblivious* Turing machine (a Turing machine whose head position does not depend on the contents of the input, but rather only on the input length). In doing this transformation, the oblivious Turing machine would run in time $T(n)\log(T(n))$; it can then be shown that such an oblivious Turing machine can be simulated with a circuit of size $T(n)\log(T(n))$.

Notice that with the fact that $\mathsf{P} \subseteq \mathsf{P/poly}$, if $\mathsf{P} = \mathsf{NP}$, then $\mathsf{NP} \subseteq \mathsf{P/poly}$ as well. In the contrapositive, if $\mathsf{NP} \not\subseteq \mathsf{P/poly}$, then $\mathsf{P} \neq \mathsf{NP}$.

This means that in order to show $\mathsf{P} \neq \mathsf{NP}$, it suffices to show that some problem in NP cannot be solved with a polynomial-size circuit. This connection allows us to approach the P vs. NP problem through another domain—rather than working with Turing machines, with lots of moving parts, we can instead think about circuits, which are just static configurations of gates, and becomes a more combinatorial problem involving graphs.

For example, if we can show that any circuit for CLIQUE has size at least $n^{\log n}$, then we've shown that $\mathsf{P} \neq \mathsf{NP}$.

24.1.1 Turing Machines Taking Advice

Another view of $\mathsf{P/poly}$ is through Turing machines taking advice.

Definition 24.5: $\mathsf{DTIME}(T(n))/a(n)$

Given $T : \mathbb{N} \to \mathbb{N}$ and $a : \mathbb{N} \to \mathbb{N}$, the class $\mathsf{DTIME}(T(n))/a(n)$ is the set of all languages L such that there exists a Turing machine M and a sequence $\{\alpha_n\}_{n=1}^{\infty}$ of binary strings $\alpha_n \in \{0,1\}^{a(n)}$ such that

$$x \in L \iff M(x, \alpha_{|x|}) = 1,$$

and M runs in time $O(T(|x|))$ given $(x, \alpha_{|x|})$.

Here, we can think of this as Turing machines that have some advice α_n for inputs of length n. Note that this string of advice is the same for all inputs of length n. Put another way, this can be seen as doing some preprocessing of the inputs, and storing the result of the preprocessing in a string α_n, which we give to the Turing machine M to aid its computations.

We then have the following alternative definition of $\mathsf{P/poly}$:

Proposition 24.6: P/poly (alt)

We have

$$\mathsf{P/poly} = \bigcup_{c,d \geq 1} \mathsf{DTIME}(n^c)/n^d.$$

That is, $L \in \mathsf{P/poly}$ as defined above if and only if L has polynomial size circuits.

Proof. In the backward direction, suppose L has polynomial size circuits. This means that the advice could be the description of the circuit (for input of length n). The circuit is of polynomial size, so α_n would be of polynomial size.

The Turing machine can then just evaluate the circuit on the input, deciding L for inputs of length n.

In the forward direction, suppose $L \in \mathsf{P/poly}$, i.e. any input $x \in L$ if and only if there exists a Turing machine M and advice $\alpha_{|x|}$ such that $M(x, \alpha_{|x|}) = 1$.

We want to show that there exists a polynomial size circuit $C_{|x|}$ (dependent on the input size) such that $C_{|x|}(x) = M(x, \alpha_{|x|})$.

This circuit can simply be the circuit simulating M with advice $\alpha_{|x|}$, since for all inputs of the same length, the advice $\alpha_{|x|}$ is constant. Since M runs in polynomial time, the circuit for M on input x would be of size polynomial in $|x|$; after hard-wiring the value of $\alpha_{|x|}$ where needed, we have our resulting circuit $C_{|x|}$ that decides L by simulating $M(x, \alpha_{|x|})$. □

Note that we'll still usually refer to $\mathsf{P/poly}$ and work with $\mathsf{P/poly}$ through the definition of polynomial-size circuits.

As an addendum, note that $\mathsf{P/poly}$ isn't the same as P nor NP, since it actually contains undecidable languages.

24.2 Circuit Lower Bounds

Going back to the ideas of lower bounds on circuit sizes to work toward showing $\mathsf{P} \neq \mathsf{NP}$, remember that we want to show that some problem in NP is not in P/poly.

Can we show that $\mathsf{SAT} \notin \mathsf{P/poly}$? Currently, the best known circuit lower bound for a language in NP is $5n$, which is quite weak—we don't even know if all problems in NP require more than linear size circuits.

An alternative approach toward proving $\mathsf{NP} \not\subseteq \mathsf{P/poly}$ is by first showing $\mathsf{NP} \not\subseteq \mathcal{C}$ for more restricted classes of circuits, and extend $\mathcal{C}$ gradually, removing restrictions until we get to P/poly.

Let's talk about a couple of these more restricted classes

Definition 24.7: NC^i (Nick's Class)

For $i \in \mathbb{N}$, a language $L \in \mathsf{NC}^i$ if there exists a constant α, a polynomial p, and a circuit family $\{C_n\}_{n=1}^{\infty}$ deciding L such that:

- $\forall n : \text{size}(C_n) \leq p(n)$
- $\forall n : \text{depth}(C_n) \leq d \cdot (\log n)^i$
- All gates in C_n have fan-in ≤ 2

Definition 24.8: NC

We define

$$\mathsf{NC} = \bigcup_{i=1}^{\infty} \mathsf{NC}^i.$$

Here, NC^i essentially models parallel computation, giving us languages that can be computed efficiently in parallel. In contrast to our definition of P/poly, with NC^i, we have a restriction on the depth of the circuit. Viewing the circuit as a bunch of parallel computations (i.e. each branch is computed and propagated in parallel), the depth of the circuit tells us the latency of the circuit.

Definition 24.9: AC^i (Alternating Circuits)

For $i \in \mathbb{N}$, a language $L \in \mathsf{AC}^i$ if there exists a constant α, a polynomial p, and a circuit family $\{C_n\}_{n=1}^{\infty}$ deciding L such that:

- $\forall n : \text{size}(C_n) \leq p(n)$
- $\forall n : \text{depth}(C_n) \leq d \cdot (\log n)^i$
- All $\wedge$, $\vee$ gates in C_n have unbounded fan-in ($\neg$ gates have only one input)

Definition 24.10: AC

We define

$$\mathsf{AC} = \bigcap_{i=1}^{\infty} \mathsf{AC}^i.$$

The only difference here is that AC^i allows for an unbounded fan-in for $\wedge$ and $\vee$ gates; there is still the same restriction on the depth of the circuits.

Example 24.11

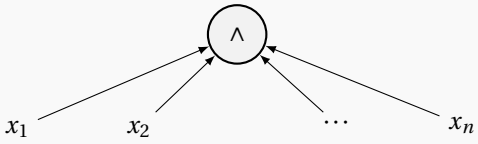
The class NC^0 contains circuits of depth $O(1)$, and fan-in 2. If the depth of the circuit is a constant $d = O(1)$, then there would be at most 2^d leaves in the circuit, which is also a constant.

This means that languages in NC^0 are local functions; even though we take an input of n bits, the output depends only on a constant d bits. These are very simple functions, and cannot even compute languages that depend on the entire input.

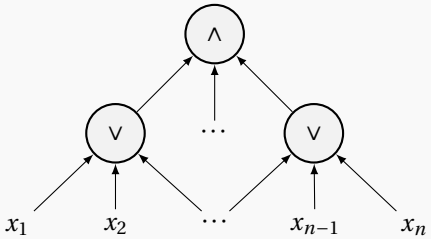
Example 24.12

The class AC^0 contains circuits of depth $O(1)$, size $\text{poly}(n)$, but with unbounded fan-in.

For example, the function that takes the AND of all the input bits is in AC^0:



The function that computes any CNF formula is also in AC^0:



However, it turns out that the parity function (i.e. counting the number of 1's in a bitstring modulo 2) is not in AC^0.

We know a lot about AC^0, and have established circuit lower bounds on the class.

Example 24.13

The class NC^1 contains circuits of depth $O(\log n)$, size $\text{poly}(n)$, and with fan-in 2. These are equivalent polynomial size formula; we can convert any such circuits into a binary tree of logarithmic depth, which would have polynomial size.

This is the first class in which we don't know too much about. In particular, it's still an open question as to whether $\mathsf{NP} \subseteq \mathsf{NC}^1$, or whether $\mathsf{NEXP} \subseteq \mathsf{NC}^1$.

We have the following sequence of complexity classes:

$$\mathsf{NC}^0 \subsetneq \mathsf{AC}^0 \subsetneq \mathsf{NC}^1 \subseteq \mathsf{AC}^1 \subseteq \mathsf{NC}^2 \subseteq \mathsf{AC}^2 \subseteq \cdots.$$

Here, we know that $\textsc{And}(x) = x_1 \wedge x_2 \wedge \cdots \wedge x_n$ is in AC^0 but not in NC^0, and we also know that $\textsc{Parity}(x) = \sum_{i=1}^n x_i \pmod 2$ is in NC^1 but not in AC^0.

It's still an open problem for whether the rest of the containments are strict.

24.2.1 Parity

Theorem 24.14

For all n, any AC^0 circuit of depth d computing PARITY must be of size $2^{\Omega(n^{\frac{1}{d-1}})}$.

This is also a tight bound; for all n, there exists an AC^0 circuit of depth d and size $2^{O(n^{\frac{1}{d-1}})}$ computing PARITY.

We'll show a much simpler case of the above theorem; that any AC^0 circuit of depth 2 for PARITY must be of size at least 2^n.

Let C be a circuit for PARITY of depth 2. This means that C can either be an OR of ANDs, (i.e. a DNF), or an AND of ORs (i.e. CNF). Consider the first case, i.e. we have an OR of ANDs; the second case is symmetric.

Here, we have $C = C_1 \vee C_2 \vee \cdots \vee C_m$, where each C_i is an AND of literals. We want to show that $m \geq 2^{n-1}$.

Firstly, we claim that any C_i is either never satisfied, or satisfied by exactly one assignment to $x_1, \ldots, x_n$. Suppose by constriction that C_i is satisfied by assignments $a, b \in \{0,1\}^n$, where $a \neq b$.

We can write

$$C_i = \ell_{i,1} \wedge \ell_{i,2} \wedge \cdots \wedge \ell_{i,k},$$

where each $\ell_{i,j}$ are literals. Since $a \neq b$, then there must exist some index j such that $a_j \neq b_j$. This means that no literal can depend on x_j. Otherwise, there would be contradicting evaluations from a and from b; one assignment has x_j be false (making C_i be false if x_j is included), and the other assignment has x_j be true (making C_i false if $\bar{x}_j$ is included).

With this information, suppose we consider a', which is the same as a but with the jth bit flipped. Since C_i does not depend on x_j, this means that a' also satisfies C_i. Further, since C is an OR of these clauses, a' also satisfies C. This means that both a and a' are assignments that satisfy C, but with one bit flipped.

This is a contradiction—since a and a' differ by only one bit, we must have $\text{PARITY}(a) \neq \text{PARITY}(a')$. This means that it must be the case that each clause is satisfied by at most one assignment.

There are 2^{n-1} inputs for which $\text{PARITY}(x) = 1$. If $C = C_1 \vee C_2 \vee \cdots \vee C_n$, then C accepts at most m inputs. Since we must accept all of the 2^{n-1} possible inputs, this means that $m \geq 2^{n-1}$, and the size of C must be exponential in n.