18.404 — Theory of Computation

Class by Michael Sipser

Fall 2023

Notes for the MIT class **18.404** (Theory of Computation), taught by Michael Sipser. All errors are my responsibility.

Contents

§1 September 7, 2023

§1.1 Introduction

This is a TCS class. Why study theory anyways? For one thing, theory is good for you; but more than that, it's important. TCS has played an important role in the development of the subject. IF you think about computers, they're amazing — Prof. Sipser has been around for a while, and when he was in high school in the 1960s, computers were amazing even then; now they're even more amazing. But the huge advances we've made in computer technology and theory doesn't take away from the fact that in some sense, the field is still in infancy — there are huge questions we don't have a clue about. For example, how does the brain work? The brain is a computational device in some sense. We do phenomenal things with machine learning and deep networks, but that makes us appreciate the brain even more — we're able to approximate what the brain does in an impressive but crude way, by throwing tremendous resources (terabytes of data, massive amounts of computation) — and we get only a small fraction of what the brain can do in our heads (it's low-powered slow software, but it manages to do all this stuff — how?). Or take evolution or the origin of life; these are also computational questions, but biologists don't have a good sense of how that really works in a deep way. Or getting back to computer science, there's quantum computing — we've been talking about it for decades. When it first became a thing people thought it'd be 20 years off; they still think so now. Will we ever really be able to build a quantum computer that can factor efficiently? The jury is still out on that.

Or sticking with theory, can we factor big numbers on a classical computer? There are some problems we can solve quickly, but factoring isn't one of them; why? More generally, there's the P vs. NP question. So there are basic questions that we don't really have any glimmer of an idea about. So we can't really say that we understand computation when such basic questions are really so far away from our understanding.

In a certain sense, you can say that computer science is almost like the way physics was at the end of the 19th century, when they thought they'd made a tremendous amount of progress in understanding physical laws (there's a quote saying that what's left is refinement), but that's all pre-relativity and pre-quantum mechanics. Prof. Sipser would suspect that there are similarly really big ideas yet to be discovered in computer science, and some of that may come out of theory.

This class will broadly speaking have two halves. The first half is *computability theory* — trying to understand what you can solve on a computer in principle. For example, suppose we have two programs; are they equivalent or not? Can we solve this problem computationally? The answer to that is not so simple — it depends on what you mean by a program, and what the underlying model of computation is. For weaker models of computation, you are able to test by an algorithm whether two programs are equivalent. But for more general models, you can't.

To develop that, we'll have to come up with what we mean by a model of computation, and give various examples; we'll start that process today.

Another question from the theory of computability concerns mathematical truth — given a mathematical statement, is it true or not? Can you compute a program to answer that question (test whether a statement is true, or even whether it's provable)? These two problems cannot be solved by any algorithm (that we can prove).

In the second half of the course, we'll look at *complexity theory*. Computability theory asks what's solvable by computers in principle; complexity theory asks what's solvable in practice, when we take into account the amount of resources (time or memory) needed to solve the problem. Then we can look at for example the factoring problem (is it solvable in a certain amount of time without searching through all possible factors, or is searching inherent to the problem)? There's a lot of theory regarding this, and more generally the P vs. NP problem — which is sort of about the ability to avoid computational search in problems that seem to require it.

As part of that, we'll develop different measures of complexity (in terms of time, memory, and randomness).

That gives a picture of what the course is about; and with that, we'll dive in.

§1.2 Finite automata

We'll begin by discussing a simple model of a computer — finite automata are a model of computation that's very simple and easy to spell out. The intuition is that it captures computational devices that only have a fixed, finite amount of memory (that doesn't depend on the input); we think of that amount as relatively small.

A finite automaton has different parts — we draw a picture with circles q_1, q_2, q_3, and arrows $q_1 \to q_1$ (labelled 0), $q_1 \to q_2$ (labelled 1), $q_2 \to q_1$ (labelled 0), $q_2 \to q_3$ (labelled 1), and $q_3 \to q_3$ (labelled $0, 1$). We also have an in-arrow to q_1, and q_3 is double-circled.

Finite automata have states, transitions, a starting state, and accepting states. The states are what appear inside the circles (q_1, q_2, and q_3). We also have transitions, which are arrows labelled by symbols of an alphabet (the set of symbols the machine works with). There's a starting state (the one with an arrow in from nowhere), and *accepting states*.

How do we think of this as a computational device? It takes as input a string of symbols that are taken from some pre-specified alphabet of symbols, in this case 0 and 1. For example, our input might be 01101.

That string is fed into this device. The device reads the symbols left to right, one by one, and it will start at the starting state and follow the transitions which correspond to the symbols it's reading off the input.

Every state should have an outgoing 1 and outgoing 0 — or more generally, some outgoing transition for each of the symbols in the alphabet.

So to simulate its behavior, we can start at q_1; then we read a 0, so we follow the 0 transition and stay at q_1. Then we read 1, and go to q_2. Then we read the next 1, and follow the transition to q_3. Now we have the symbols 0 and 1 remaining in the input, and we take those and end up in state q_3.

Now once we've reached the end of the input, the machine needs to give an output — here, that'll be the simple binary output 'accept' or 'reject'. (We'll sometimes look at more complicated output, but usually we'll just use simple binary.) That decision depends on whether or not you ended up in an 'accept' state (the ones that are double-circled). Here we ended up at the state q_3, which is an accept state, so the output is 'accept.'

In contrast, if we take the string 00101, then we start at q_1 and stay there twice; then we take the 1 to q_2, take the 0 back to q_1, and take the 1 to q_2. So we end at q_2, which is *not* an accepting state; so the output is 'reject.'

Given a finite automata, we might want to understand 'what are all the strings that the machine accepts?'

> **Definition 1.1.** The *language* of an automaton is the collection of all accepted inputs.

For this particular machine, we have to end up at state q_3; and once we get there, we'll be there forever. The way we get from q_1 to q_3 is that at some point, we need to have a 11 appearing in the input. So the collection of accepted strings is all the strings containing a substring 11 appearing somewhere. So if we name this automaton M_1, we say that the *language* of M_1 is the set

$$A = \{w \mid w \text{ has a 11 substring}\}$$

(where w is a string of 0's and 1's).

One thing we'll develop facility with is to be able to look at an automaton and determine what its language is; and given a language, either come up with a finite automaton which has that as its language or show that there is no such automaton.

Remark 1.2. We also say that M_1 *recognizes* A (to mean that A is the language of M_1).

§1.3 Formal definition

We'll now get to the formal definition, where we formally write down what a finite automaton is and how it computes. This allows us to be more precise, and develop some terminology so that we can say things more clearly.

Definition 1.3. A *finite automaton* M is a 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$ where:

- Q is the set of states.
- Σ is the input alphabet.
- δ is the *transition function* $\delta\colon Q \times \Sigma \to Q$.
- $q_0 \in Q$ is the start state.
- $F \subseteq Q$ is the set of 'accept' states.

Here Q and Σ both have to be finite.

In our example, $Q = \{q_1, q_2, q_3\}$.

In words, Σ is the set of symbols that the machine can expect to read.

In words, the transition function δ is a way to specify the arrows. It takes as input a pair consisting of a pair (q, a) where q is a state and a one of the alphabet symbols, and it gives a state r. This represents that we have a transition $q \to r$ given by reading the symbol a. We can imagine writing down all our transitions in a table (for each initial state and each symbol, we write down the resulting state); this function represents that table.

The *input* to our finite automaton will be a *string*.

Definition 1.4. A *string* is a (finite) sequence of symbols from the input alphabet.

People also study infinite strings, but we won't; we'll confine our strings to be finite.

Definition 1.5. A *language* is a set of strings.

(This comes out of the history of the subject, which had a lot of connections to linguistics.)

Strings are always finite, but languages can be finite or infinite.

Notation 1.6. The *empty string* is denoted ε, and the *empty language* is denoted $\varnothing$. These are very different objects — the empty string is the string of length 0, and the empty language is the set that contains no strings whatsoever.

We'll now talk about how these formally specified automata compute; this should match our intuition about following the arrows.

Definition 1.7. We say that M *accepts* a string $w = w_1 w_2 \cdots w_n$ (for $w_i \in \Sigma$) if there is a sequence of states $r_0, r_1, \ldots, r_n \in Q$ where:

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

In words, here we have a string whose symbols are in our alphabet, of length n. The number of states we have is one more than the number of symbols in the input string. Our machine starts off in the start state, so we want $r_0 = q_0$; and the next state r_i should come about by taking the previous state r_{i-1} and reading the next input symbol, and applying the transition function. The machine accepts if the final state r_n is an accept state.

This is a formal way of representing the path which follows the arrows reading the input.

Remark 1.8. We won't typically use this level of detail, but it's useful to see it before we start talking about things on a higher level.

Definition 1.9. We say the *language* of M, denoted $\mathcal{L}(M)$, is the set of all w such that M accepts w.

Example 1.10

In our example, the language of M_1 was A. (As mentioned earlier, another way of saying this is that M_1 recognizes A.)

Finally, we have an important definition:

Definition 1.11. A language B is *regular* if there is some finite automaton that recognizes it — i.e., if $B = \mathcal{L}(M)$ for some finite automaton M.

So regular languages are the languages we can get with a finite automaton.

Example 1.12

The set $\{w \mid w \text{ has 11 as a substring}\}$ is a regular language, since we've seen a finite automaton that gets that language.

We'd like to understand intuitively what are the regular languages; we'll do this by looking at an entirely different way of describing them, not in terms of an automaton, and proving that these two ways of describing them are equivalent.

Remark 1.13. Some of us will have seen finite automata before and may find this slow; others who haven't may find this fast. If we haven't seen finite automata, we need to read it in the book and work through examples — practicing, making up automata for various languages or reading automata and figuring out their languages. We'll have to develop some feeling for finite automata to move forward with this — in recitation we'll see a few examples, but we'll mostly be proving theorems about automata rather than giving examples, so it's important we have intuition for them.

§1.4 Regular operations

We'll now describe a different way of looking at languages, through certain kinds of operations, which we'll call *regular operations*. These are operations like $+$ and $\times$ which usually apply to numbers, but here instead apply to collections of strings.

These operations will be *union*, *concatenation*, and *star*.

Definition 1.14. The regular operations are defined as follows. Given languages A and B:

- Their *union* $A \cup B$ is
$$A \cup B = \{w \mid w \in A \text{ or } w \in B\}.$$
- Their *concatenation* $A \circ B$ (or AB) is
$$A \circ B = \{w \mid w = xy \text{ where } x \in A \text{ and } y \in B\}.$$
- The *star* A^* (this is a unary operation rather than a binary one) is
$$A^* = \{w \mid w = x_1 x_2 \cdots x_k \text{ for some } k \geq 0 \text{ and } x_i \in A\}.$$

The union is the definition we're familiar with (it's obtained by combining A and B). The concatenation is different — we get it by taking each string from A, and appending each string from B, in all possible ways. The star may be the oddest one if we haven't seen it before — we get it by taking some number of elements of A and sticking them together, in all possible ways. Note that the x_i don't have to be distinct. So we take all possible elements of A as many times as we like and stick them together; in particular, we're allowed to take 0 elements and stick them all together, in which case we get the empty string. So the empty string ε is *always* in A^*.

Example 1.15

Suppose that $A = \{0, 1\}$ and $B = \{a, b, c\}$. Then:

- Their union $A \cup B = \{0, 1, a, b, c\}$ has 5 elements.
- Their concatenation $AB = \{0a, 0b, 0c, 1a, 1b, 1c\}$ has 6 elements.
- $A^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, \ldots\}$ (since we take all possible ways of putting together elements of A). In particular A^* is infinite.

Note that the elements of A and B are strings; they don't have to be individual symbols as in this example.

Remark 1.16. We don't need to assume that A and B have the same alphabet, but we almost always do assume this (it's simpler and covers all the interesting cases we care about).

We're now going to take these regular operations and build expressions out of them, in the same way we take arithmetic operations and build arithmetic expressions — for example, we can say $6+2$, or $6+2\times7-3$.

Definition 1.17. A *regular expression* is built from the atomic elements $a \in \Sigma$ (each member of Σ is an atomic element), ε, and $\emptyset$ using $\cup$, $\circ$, and $*$.

Example 1.18

- We have $(0 \cup 1) = \{0, 1\}$. (0 and 1 are both members of our alphabet, and $\{0, 1\}$ is a language consisting of both strings.)
- $(0 \cup 1)^*$ is the set of all strings using $\Sigma = \{0, 1\}$, which we denote by Σ^*.
- Σ^*1 means we take all possible strings and stick a 1 at the end of each; this gives us all strings that end in 1.
- $\Sigma^*11\Sigma^*$ consists of all strings that have 11 as a substring; this is the language A we saw in our initial example of a finite automaton.

Remark 1.19. If we apply $*$ to any language, we always get a language which is infinite, with two exceptions — if the language just consists of the empty string (we have $\{\varepsilon\}^* = \{\varepsilon\}$), or if we have the empty language (we also have $\varnothing^* = \{\varepsilon\}$, since the only thing we can do is pick no things, i.e., take $k = 0$).

We're eventually going to show that for any language we can get from a regular expression, we can also get it from a finite automaton, and vice versa. This is our big goal — to show that finite automata and regular expressions describe the same class of languages, namely the regular languages.

What we've done so far is introduced finite automata and described them formally, defined the regular languages as the languages we can recognize using finite automata; and introduced regular expressions as an alternative way of describing languages; and we're going to argue that they're the same.

Remark 1.20. A subset of a regular language is not necessarily regular. (If the subset of a regular language were always regular, then *all* languages would be regular — Σ^* is a regular language, since we can make a finite automaton that accepts everything.) So a regular language has to be *exactly* the strings that a finite automaton accepts, not just a subset of them.

§1.5 Unions of regular languages

Question 1.21. What operations can we apply to regular languages to get regular languages?

We've seen this is false for subsets. But it's true for unions, and this will be our first theorem of the course.

Theorem 1.22

If A_1 and A_2 are regular languages, then so is $A_1 \cup A_2$.

We're doing this partly to get us into the spirit of the kinds of things we'll be doing.

Proof. Since A_1 and A_2 are regular languages, they have finite automata (by definition); so let M_1 and M_2 be the respective finite automata that recognize A_1 and A_2, and let $M_1 = (Q_1, \Sigma_1, \delta_1, q_1, F_1)$ and $M_2 = (Q_2, \Sigma_2, \delta_2, q_2, F_2)$.

Our goal is to show that $A_1 \cup A_2$ is regular too; we do this by constructing a finite automaton for the union language, built out of the two finite automata for the starting languages. We'll call this automaton $M_\cup = (Q, \Sigma, \delta, q_0, F)$, and we want it to recognize $A_1 \cup A_2$.

Prof. Sipser will give us the idea for that construction. Consider M_1 and M_2, drawn as two circles — these are two finite automata, with states drawn as smaller circles inside, and with in-arrows pointing to their starting states.

We want to take these two automata, and combine them into a new automaton which is going to do the union. This means our new automaton $M_\cup$ (the one we're trying to build) should accept an input if and only if either M_1 or M_2 would accept it (or both).

If we haven't seen this before, Prof. Sipser recommends putting ourself in the place of the machine we're building — think about, how would I do it, and then try to limit my technique to the kinds of things the automaton can do; and then implement that intuition as the machine. Putting yourself in that place helps with creativity. If we put ourself in the place of M_1 and M_2, we're getting some input w. We'd like to simulate both M_1 and M_2 — first we feed w into M_1 and see what it does; if M_1 accepts it, we should accept it. Then we feed w into M_2; if M_2 accepts, then we accept. But there's one problem with doing it that way — $M_\cup$ doesn't have the ability to rewind w and first feed it into M_1 and then M_2, because a finite

automaton just gets one shot at seeing the input from the beginning to the end — it can't look at it again. So we can't implement feeding the input into M_1 and then M_2.

This means instead, we want to do the two in parallel — we want to take w and feed it into both M_1 and M_2 at the same time, and keep track of where each automata is as it's reading the symbols of w. If we think of running M_1, we can move our finger along there to keep track of which state we're on; and we do the same inside M_2 to keep track of where it is.

When we're trying to design our automaton, the states represent memory in the sense that for each distinct possibility we need to keep track of, it'll have its own state. So the states in our new machine should be all the different pairs of being in some state in M_1 and some state in M_2 — if we have a state q in M_1 and a state r in M_2, then $M_\cup$ should have the state (q, r) — because it'll have to remember this as a possibility. So we have a state to represent each possible pair of a state in M_1 and a state in M_2. So we set $Q = \{(q, r) \mid q \in Q_1, r \in Q_2\}$; another way of writing this is $Q_1 \times Q_2$.

We'll do the transition function next — we need to define $\delta((q, r), a)$. When we're at (q, r) and read a, how do we know where to move to? We look at our two machines M_1 and M_2 and see where q goes to when we read a and where r goes to when we read a, and that pair is our new state; so we set

$$\delta((q, r), a) = (\delta_1(q, a), \delta_2(r, a)).$$

Now let's set the start state; we start at $q_0 = (q_1, q_2)$.

Finally, we'll choose the accepting states. It's tempting to set $F = F_1 \times F_2$ — but this is the set of states where *both* M_1 and M_2 are in accept, which gives a different operation (namely, the intersection). We're instead working with union; so instead, we take

$$F = (F_1 \times Q_2) \cup (Q_1 \times F_2)$$

(since we need either the first machine to accept and the second to do anything, or vice versa). □

> **Remark 1.23.** We say that the class of regular languages is *closed under union* (just like the integers are closed under addition and multiplication, but not division).

§2 September 12, 2023

§2.1 Pset Announcements

The problem set has been posted (it was posted last lecture); it is due on the next Thursday. You have to submit on Gradescope; instructions are on the homepage. (There are some idiosyncracies about how submission is done, so make sure you follow the instructions exactly and you won't have problems.) If you can't get to the homework and need to turn in late, you get an automatic 18-hour extension for a small penalty; that extension applies on a per-problem basis (for each problem you turn in late, you get a 1-point deduction out of 9 points). There are instructions on how to do this on the homepage. If you turn things in very late, that's more of a problem, because we post solutions after the late deadline. (Normal problems are due Thursday at noon, the late deadline is Friday at midnight.) If you turn in after that, we may not be able to count or grade it unless you have an excuse from S^3.

Most problems in the problem set — Prof. Sipser aims for all — we'll have enough to solve them a week in advance (meaning at the end of the Thursday lecture a week before it's due). (Sometimes a bit spills over into the Tuesday lecture.)

A week before the problem set is due, you should start actively thinking about it; don't wait until the last minute. These problems in particular, you really need to cogitate for a while — the problems are meant to be fun and interesting, but they usually involve some nontrivial idea that you'll have to think about for a bit. If you start in advance and give your brain a chance to let your subconscious creativity come into play, you'll have better luck starting a few days in advance.

All this is spelled out on the information sheet on the homepage, but to emphasize some of the key points: collaboration is okay, but it's important you spend some time thinking about the problems on your own (e.g. think about the problems on your own before the collaboration session), and you have to write them up yourself.

Every problem set will have 1–2 optional problems. The optional problems do not play a role in the grade (they don't help you, or hurrt you if you get them wrong); if you don't do some of the other problems, doing an optional problem doesn't substitute. But if you want to get an A+ you have to do some optional problems; or if we're going to ask for a letter of recommendation. Prof. Sipser probably isn't going to get to know most of us personally ('the spirit is willing but the brain is weak'), and in that case he can say that we did a bunch of the optional problems, which are hard and involve a lot of creativity — that's something he can say beyond that we got an A in the course.

§2.2 Review

Last class, we started looking at finite automata as a very simple model of computation. We introduced finite automata, gave an informal definition in terms of state diagrams and a formal definition in terms of five objects. A finite automaton may accept zero or some number or infinitely many strings; the collection of all accepted strings is the *language* of that automaton. An automaton may accept any number of strings, but it always recognizes exactly one language (the set of precisely the strings that the machine accepts).

We also introduced the regular operations $\cup$, $\circ$, and $*$, and the regular expressions you can build up using them. We mentioned that our goal over the next few lectures is to prove that these two formalisms are equivalent in terms of the languages they describe — anything you can do with a finite automaton (i.e., a regular language) you can do with a regular expression, and vice versa. We'll prove this — we'll give a conversion procedure to go from regular expressions to finite automata (today) and the reverse (next class).

In setting this up, we'll prove some closure properties for the regular languages — last time we showed the regular languages are closed under union (if we have two regular languages and we take their union, the result is also regular). We'll today show this is true for $\circ$ and $*$ as well; that will lead us to a procedure for converting regular expressions to finite automata.

Last time, we proved that regular languages are closed under $\cup$. In brief, the way this looked was that if we take two regular languages A_1 and A_2 and their associated finite automata M_1 and M_2, we take these two automata and build an associated automaton $M_\cup$ which recognizes the union language. (If we're not familiar with finite automata, there are lots of examples in the book to practice with them; you want to develop your feeling for how you construct finite automata, since they'll be the starting point for a long story.)

The idea is that $M_\cup$ is going to basically simulate the two original automata — think of our input w as going into the union machine. Then the union machine is in turn feeding that input into M_1 and M_2. And M_1 and M_2 are processing the symbols of w, so we're going to be wandering around from state to state according to their respective transition functions; $M_\cup$ is going to remember in its state which pair of states M_1 and M_2 are in at that point, having read that much of the input. So if after reading some amount of input M_1 is in state q and M_2 in r, then $M_\cup$ should be in the state (q, r) — the states of $M_\cup$ are pairs of states in M_1 and M_2 (formally, if M_1 and M_2 have set of states Q_1 and Q_2, then $Q = Q_1 \times Q_2$ is the set of pairs of an element of Q_1 and an element of Q_2).

§2.3 Closure under concatenation

Theorem 2.1

If A_1 and A_2 are regular languages, then so is A_1A_2.

(We suppress $\circ$ — A_1A_2 still means concatenation.)

We'll start off the proof in the same way — let M_1 be a finite automaton that recognizes A_1, and M_2 one that recognizes A_2. We want to construct $M_\circ$ to be a finite automaton that recognizes A_1A_2; our job is to demonstrate such an $M_\circ$.

Let's first get some intuition for what we're trying to do. In other words, for input w, what does it mean for w to be in A_1A_2? ($M_\circ$ is supposed to tell us whether w is in A_1A_2, and accept it if it is and reject it if it isn't.) This means w has to be built out of some string from A_1 concatenated with some string from A_2 — in other words, there has to be some way to break w into two pieces where the first piece is in A_1 and the second is in A_2. If *you* were given w and had to answer if $w \in A_1A_2$, you'd have to figure out, is there a way to cut w into two such pieces?

We have M_1 and M_2, and we'd like to somehow build $M_\circ$. We want to feed w in and only run M_1 on w, until maybe M_1 accepts; then we'd like to jump to M_2 and let M_2 do the rest. (So M_1 accepts the initial piece of w, and then we move to M_2 and read the rest — we need to jump somehow to the original start state.)

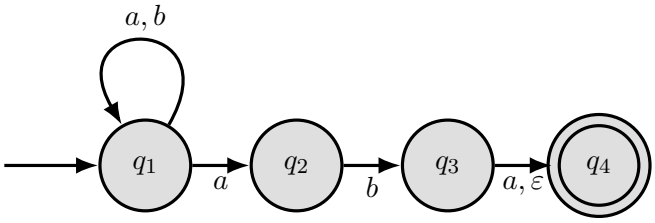
It seems like we're onto something here, but it has a basic problem — just because we've found an initial piece of w that's accepted by M_1 doesn't mean that that's the right place to cut w. It could be that there are multiple such pieces,and if we cut too early, we might get a cutting where the remainder is not in A_2. This situation doesn't imply that w is out of A_1A_2, because maybe we should have waited and found a longer initial piece that's in the A_1 language — and maybe that would have been a better place to cut, where the remainder really is in A_2.

Once you have this realization, then it seems almost hopeless — because in order to know where to make the right cut, we kind of have to know the future, and, well, you don't know the future — so you don't know whether to jump from M_1 to M_2 or stay on within M_1 and hope for a better place to cut.

We'll come back to this later; this is a perfect lead-in to one of the most important basic concepts we'll see in this course, called *nondeterminism*.

§2.4 Nondeterministic Finite Automata

We'll now shift gears — we're now going to see our second example of a model of computation, namely a nondeterministic finite automaton.



This is still a finite automaton — it has start states, accept states, and transitions. But if we look carefully, there are some key differences from the previous definition (which we'll call a *deterministic finite automaton*). The difference is in how transitions work. Before, when we were in state q_1 and had an input symbol a coming (here $\Sigma = \{a, b\}$), there was only one way to go. But here, from q_1 there are two things we can do

— go to q_2, or stay at q_1. There could be more than one ways, or just one, or even no ways (there's no a out of q_2).

There's one more difference (which is less essential, but convenient) — we also allow ε on the transitions. That does not mean we see ε as a symbol of the input; what it means is that we can take the transition for free (without reading anything from the input).

We'll first discuss how we use a NFA to process input. When a NFA is processing input, we read input one symbol at a time, but it might have several ways to go.

As an example, consider aa. When we read the first a, we can either stay at q_1 or go to q_2. To simulate that, we put a finger on q_1 and one on q_2, to represent that it's in either. Then we read another a and follow the respective transitions on our finger. The finger in q_1 again has two ways to go, so it splits in two fingers — the q_1 finger becomes a q_1 and a q_2 finger. Meanwhile, for the (original) q_2 finger reading an a, there's no way to go — so we get rid of that finger.

There could be lots of fingers; in the end some may be on an accept state, and some might not. Acceptance overrules — that's essential to how we define nondeterminism. So if at least one finger at the end of the input is on an accepting state, then the automaton accepts the input; it only rejects if all the fingers that remain are not on an accepting state (or there are no fingers).

Example 2.2

The input aa is rejected — after processing the input we had one finger on q_1 and q_2. Neither is an accepting state, so the machine rejects.

Example 2.3

Consider ab. After we read a, we're in q_1 or q_2. Then we read b. The q_1-finger stays on q_1 after reading the b. The q_2-finger advances to q_3. Now you also have to see check if there any empty-string transitions coming out of your state — because those allow further advancements. So if we're in state q_3 after processing some symbol, then we go immediately to q_3 *and* q_4 as possibilities (since we can read the empty string as a free move along that transition without processing input). So after reading ab, we're in all of q_1, q_3, and q_4 as possibilities. Since q_4 is accepting, that overrules and we accept.

(Empty-string moves can occur at any point in the string; we'll see why it's so nice to have these transitions. You can also take multiple such transitions in a row.)

(Note that an empty-string move is inherently nondeterministic, so it wouldn't make sense to have such a transition in a DFA.)

Example 2.4

Consider abb. We were in q_1, q_3, and q_4 as before, and then we read a b. The q_1 finger stays; the q_3 finger gets removed. And there's no way to go anywhere from q_4 (if you're in q_4 and any additional symbols come, you die); so this string gets rejected.

Example 2.5

Consider aba. Then we're in q_1, q_3, and q_4 (as before), but an a comes in instead of a b. Then the q_1-finger goes to q_1 and q_2; the q_4-finger disappears; but the q_3-finger can move on the a-transition to q_4. So now we're on q_1, q_2, and q_4, and so we accept.

As before, the *language* of a NFA is the set of strings it accepts. For this example, we can take any string and stay in q_1 (which we can denote at Σ^*). Then we need to somehow get to q_4; this can be done through either

ab or aba. So the regular expression describing the language of this NFA can be written as $\Sigma^*(ab \cup aba)$; we can also write this as $\Sigma^* ab(\varepsilon \cup a)$.

Here's the formal definition. Before, the transition function told us, at a particular state, if we read a symbol, what's the new state? Now it tells us, in a particular state (representing one place our finger is), on an input symbol, what's the *set* of possible states you could be in? (Now there are multiple.) The way we write this is that our transition function δ is now a function

$$\delta: Q \times \Sigma \to \mathcal{P}(Q) = \{R \mid R \subseteq Q\}.$$

There's also one other difference, which is that we have the empty string as a possible transition label. To capture this, we let $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$ — we 'enhance' the input alphabet with the empty string. We actually define $\delta: Q \times \Sigma_\varepsilon \to \mathcal{P}(Q)$.

Definition 2.6. An NFA $M = (Q, \Sigma, \delta, q_0, F)$ is a 5-tuple of a set Q of states, an alphabet Σ, a transition function $\delta: Q \times \Sigma_\varepsilon \to \mathcal{P}(Q)$, a starting state q_0, and a set of accept states F.

We'll see a lot of nondeterminism in this course. Just as an introduction, you can think about nondeterminism in different ways; depending on the situation, one or another might be a helpful way to think about it. If you want to think about this NFA processing this input, having a set of possible states it could be in, one way to model this is with a parallel computer where each of the state possibilities at a point in time corresponds to some thread of the parallelism. (So the machine as it's processing an input is following along the transitions; when there are multiple ways to go, we can think of that as the machine taking a 'fork,' deciding to go either this way or that way.) But this has the special feature that if any one of the threads ends up accepting, then overall the whole machine accepts.

Another (similar) way to think of nondeterminism is as a tree of possibilities — instead of thinking of separate devices, you can write down a tree of all the ways the machine is computing (a tree of different possibilities the machine is following, corresponding to the different possible states).

Lastly, there's one other view which is surprisingly useful, which we'll call the 'magical' view. If you think of the automaton as having different choices of which way to go, we can assume that the machine always takes the 'right' choice (which leads to acceptance, provided that there is some such way) — you can think of the machine as guessing one of the possibilities, and it always guesses right.

Remark 2.7. Is nondeterminism more powerful than determinism — can you do more things? The answer is no — we will show soon how to convert NFAs to DFAs.

Remark 2.8. Why the name 'nondeterminsm' when there's no randomness? Nondeterminism and randomization are different things — here there's no probability measure on the choices (but there are probabilistic automata we can define and explore). But it's nondeterministic because at a given point and given input, the next step is not determined — there are several possibilities.

§2.4.1 Equivalence between NFAs and DFAs

We'll now see that anything we can do with a NFA is still a regular language — in other words, we can convert NFAs to DFAs.

Theorem 2.9

If a NFA recognizes a language A, then A is regular.

Proof. Given an NFA N with $A = \mathcal{L}(N)$, we're going to construct a DFA M such that $A = \mathcal{L}(M)$ — in other words, we're going to convert our NFA to a DFA.

Instead of writing this down formally, we'll just give the idea. We want M to accept its input exactly when N accepts its input. We can again personify the automaton — if you were M, what would you need to keep track of when simulating N? You need to keep track of the set of possible states at each step — and the way a DFA does this is by having a state of its own for each of those possibilities. So M has a state corresponding to each set of states of N — if N has set of states Q, then M has set of states $Q' = \mathcal{P}(Q)$. In other words, the states of M are the subsets of states of N, corresponding to where all our fingers are in. (It's possible that some sets of states are just not reachable, but this is fine — in a DFA there might be some states you can never get to, which are still states, though they're useless in some sense; they can just live there and not bother anybody, though you could also delete them if you wanted to be more efficient.)

Now we'll quickly fill in the details. We first think about what happens if you don't have any ε transitions (you can then see what happens if you add them in); but the main idea is that M keeps track of which subsets of states N could be in. Let $N = (Q, \Sigma, \delta, q_0, F)$. Then we define $M = (Q', \Sigma, \delta', q_0', F)$ where $Q' = \mathcal{P}(Q)$, $q_0' = \{q_0\}$ (since the only possibility for where to start is the start state of N). The transition function takes each state in N and moves it along the transitions, just as we did when simulating the earlier NFA with our fingers. □

§2.5 Closure under union

To see how nice NFAs are, we'll first redo closure under union, using nondeterminism. The way we do this is: imagine we have M_1 and M_2, both with start states and a bunch of accepting states. Before, we constructed $M_\cup$ using a cross-product, where we built the union machine to keep track of which states both M_1 and M_2 are in at a given point. We'll now see an easier construction using a NFA — we construct $M_\cup$ simply by taking M_1 and M_2, throwing them together, and adding a new starting state which goes to the original start states with ε transitions. And that's it.

Why does this work? Well, when this NFA gets an input, the first thing it does — before it reads any symbols (at the start state you can't read symbols, just branch) — is go to M_1 or M_2. As we start reading the symbols of the input, we're then effectively running M_1 and M_2 in parallel (as two possibilities — there's the possibility in M_1 and the one in M_2). Then at the end of the input, we see whether one of those possibilities is on an accepting state, in which case we accept. So the definition of nondeterminism that says acceptance overrules is exactly what we need for union.

> **Remark 2.10.** We'll comment on the different perspectives. Thinking of nondeterminism as parallelism, when you split into the two possibilities, in effect you get two parallel machines, one running M_1 and the other M_2, and you accept if either accepts. This gives you one way of looking at the nondeterminism.
>
> In the magical perspective, you have your input at the beginning, and you nondeterministically guess whether it's M_1 or M_2 that's going to be accepting it; and you always guess right (that's the sense in which it's magical). Then you confirm that the guess works by continuing with the simulation according to that guess.

§2.6 Closure under concatenation

Nondeterminism allows us to resurrect the idea we started off with that didn't work — now it works. As before, we're trying to do the concatenation language, so we want to accept w if it can be split somehow into two pieces, where the first is in A_1 and the second into A_2. So we feed w into M_1 initially; if M_1 accepts, then we jump to the start of M_2 using an ε-transition. The nice thing is that this gives us the possibility of making the transition, but doesn't require it — so the nondeterminism is doing all the housekeeping for

you (at the same time as one thread of the nondeterminism is deciding to make the cut here, another cut says to hold off and stay in M_1 for longer, and if we come to another time in M_1 where we're accepting we may make the transition at that later point).

You also need to declassify the accept states of M_1 as accept states, since strings in A_1 aren't necessarily in the concatenation.

Using the magical point of view, you can say the nondeterministic automaton sort of magically guesses when is the right point to make the cut, and it jumps from an acepting state in M_1 to a start state in M_2.

§2.7 Closure under $*$

Theorem 2.11

If A is regular, so is A^*.

Proof. Again, we'll give a proof by picture. Let M recognize A; we're going to construct M_* recognizing A^*.

What does it mean for an input string w to be in A^* — pretending that we are M^*, when should we accept w? This means you can split w up into multiple segments where each one of those segments is in A.

So now suppose we have M which recognizes the original language A. To modify M, we can try to draw ε transitions from each accepting state back to the start state. This has a slight problem, which is that the star language should always include ε (that was part of the definition, corresponding to not taking any strings from the original language).

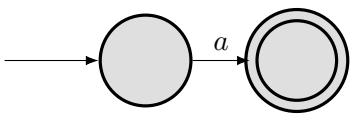
There's a not-good way to do that — since we need to accept ε, why not make the start state an accept state too? This looks tempting — now we're going to accept the empty string. But the problem is that M might reenter the start state on other occasions than just being at the start; and now suddenly those strings are going to be accepted, when maybe they shouldn't be.

So that doesn't quite work; and the way to fix this is by making a new starting state and making it an accepting state; this transitions to the original start state (with an ε). This is a way of throwing in ε into the language accepted without doing any damage. □

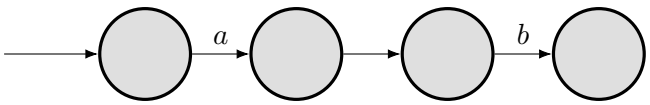
§2.8 Conversion of regular expressions to finite automata

What's nice is that this is very close to giving us a method for converting regular expressions to NFAs. (In other words, if we have a regular expression, we're going to see how to convert it to a NFA, with an automatic procedure using all the stuff we've developed so far.)

Let's say we have the regular expression $(a \cup ab)^*$. Technically this is an inductive construction; you can think of it as building up from the smaller subcomponents. We first take our atomic subexpressions a and b, and create NFAs that recognize just those languages $\{a\}$ and $\{b\}$ — for a, we take the NFA



Now we have ab, so we need to concatenate the a-machine and the b-machine. We do this as described — we take the a-machine and the b-machine, and apply the procedure for doing concatenation, where we take the accepting state of the first machine and map it to the start state of the second under an empty transition. And now this machine does exactly the language ab.



Now if we want to make $a \cup \{ab\}$, we take the machine for a and the machine for ab, add in the machine for a, and take the union construction (where we create a new start state, with ε-transitions to the two original machines).

Now for $(a \cup ab)^*$, we use the $*$ construction — add a new start state (which is accepting), map it to the old one using ε, and map all accept states back to the original start state.

§3 September 14, 2023

We've introduced DFA, NFA and proved that they're equivalent to DFA (which are a priori just a special case, never using the ability to branch) — we saw how to convert NFAs to DFAs. We used this to prove the closure of the class of the regular languages under the regular operations; that gave us one direction of proving the equivalence of regular expressions and finite automata (we showed how to convert regular expressions to finite automata using the closure property constructions). Today we'll finish the equivalence by converting finite automata to equivalent regular expressions. We'll then develop a method for proving languages are not regular (the pumping lemma); and then we'll introduce context-free grammars, another model used all over the place in e.g. CS and linguistics.

§3.1 Recap

Last time, we proved that:

Theorem 3.1

If $A = L(R)$ for a regular expression R, then A is a regular language (i.e., A has a DFA).

To prove this, we showed how to convert regulart expressions to NFAs, which can be converted to DFAs; and we gave an example for $(a \cup ab)^*$.

Today we'll do the converse:

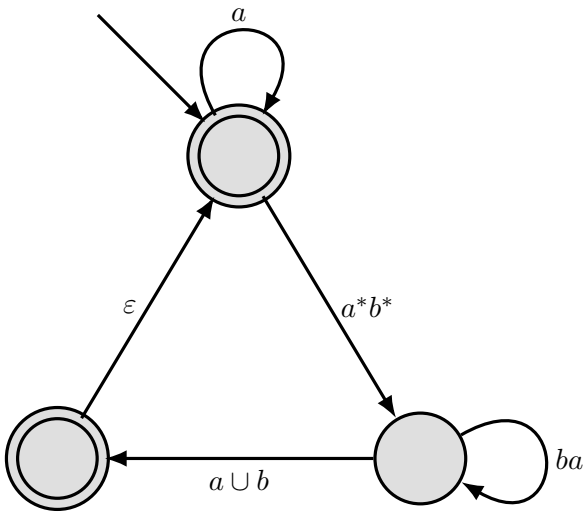
Theorem 3.2

If A is regular, then $A = L(R)$ for some regular expression R.

This means we're going to have to convert DFAs to regular expressions (that do the same language).

Before we do that, we'll make a new definition of yet another kind of finite automaton, which we call a *generalized nondeterministic finite automaton.*

Definition 3.3. A *generalized NFA* consists of states where we can write an arbitrary regular expression as a label on a transition.



How do you run a GNFA on an input? You start at the start state, and you're allowed to read a whole substring of the input at one step — you're allowed to move along one of the transitions by reading multiple symbols at once (instead of nibbling, we can take a whole bite out of the input).

There could be many ways of breaking up the input to allow us to flow through the automaton; if there is some way to end at an accept state, then the automaton accepts the input.

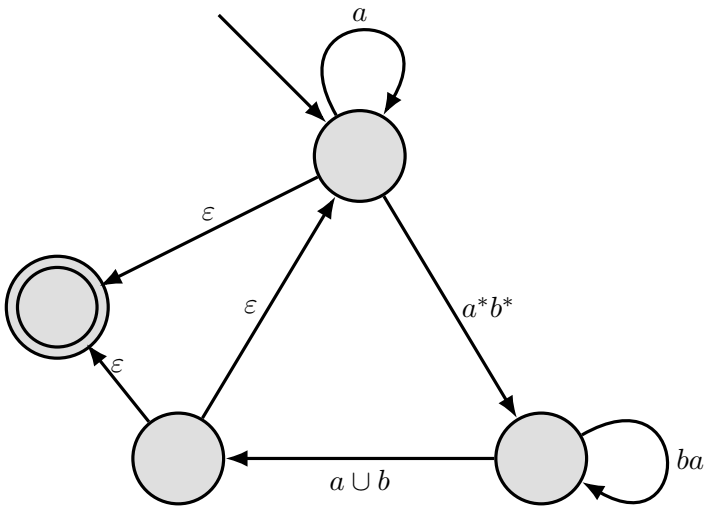
When proving you can convert a DFA to a regular expression, we'll actually work with GNFAs — we'll show how to convert any GNFA to a regular expression. This seems harder, but it'll be convenient because we'll give a proof by induction; and having a stronger statement makes it easier to do the induction.

Induction comes across in many classes as a very formal tool, but really all induction is is a recursive proof — a proof that calls itself on smaller values.

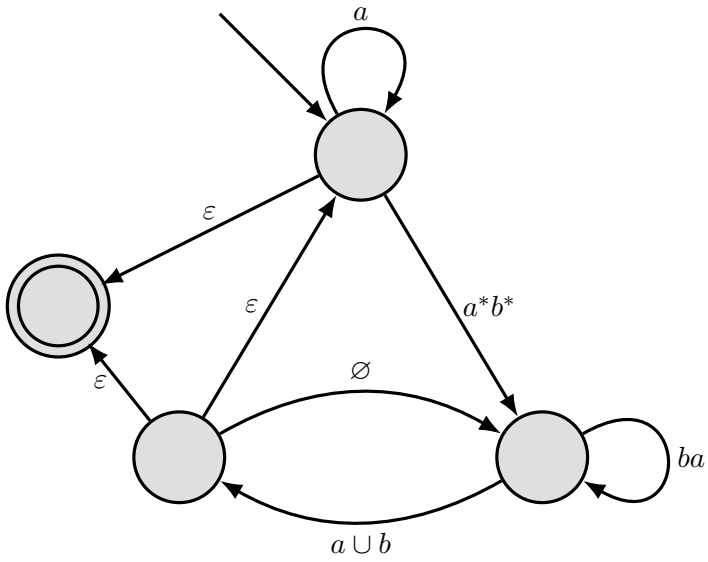
Before we get to the proof, we'll do one thing to make our lives easier — we assume GNFAs are in a convenient form, where:

- There is only a single accept state, and that single accept state is different from the start state.

  The GNFA we have violates that, but we can easily fix that by adding in a new accepting state, and having empty transitions from the original accepting states (and declassifying the original accepting states).



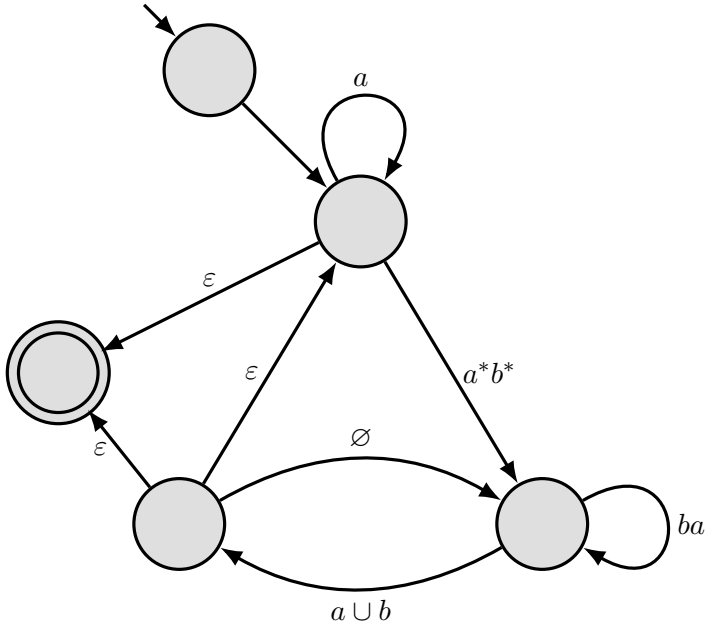
- Every possible transition is in the diagram — there is a transition between every pair of states, in both directions. For example, our original GNFA has no transition from the bottom-left to bottom-right node. We need to add a transition, but we don't want to allow ourselves to use it; so we just add it in with the empty language (so we can never use it). More generally, we can add in all missing transitions, with the empty language. (This includes self-loops.)



  (If we had multiple transitions between two states, we could union them.)

- We make one exception to that rule — we only have outgoing transitions from the start state, and incoming transitions to the accept state.

  Our start is broken, so to fix this we create a new start state that goes to the original one with an ε string (and we add new transitions going out to everything else with a $\varnothing$ label).



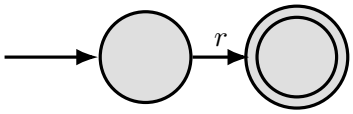
So we make these convenient assumptions; given any GNFA we can find an equivalent GNFA satisfying these assumptions.

Remark 3.4. A transition labelled $\varnothing$ is very different than one labelled ε — if we have a transition ε, we can always take it 'for free,' while we can never take a transition labelled $\varnothing$ (you'd have to read everything in the language, and you can never do that).

Lemma 3.5

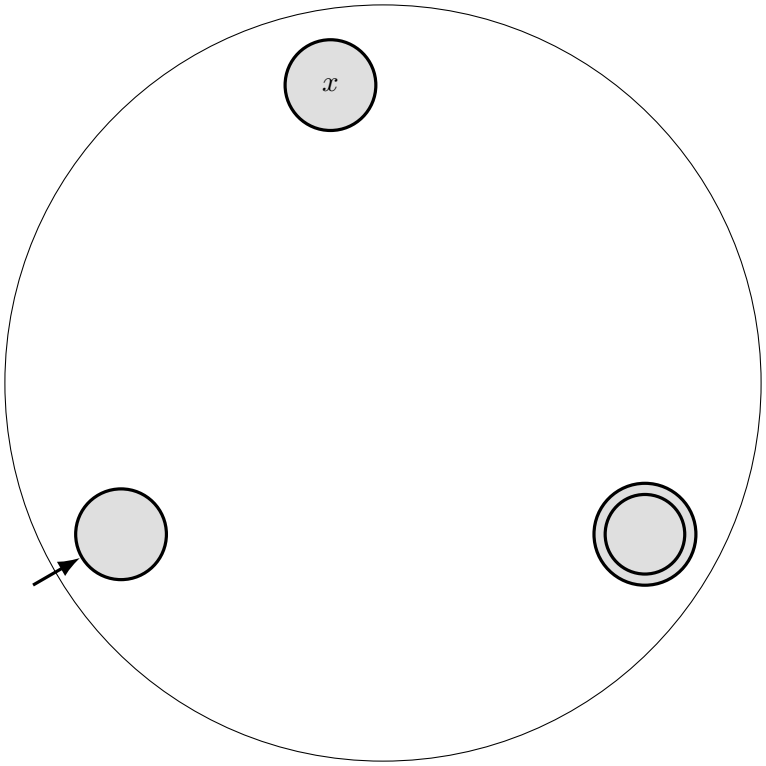
Every GNFA with k states has an equivalent regular expression R.

Proof. We'll use a proof by induction (assuming our GNFA is in the convenient form). The base case is when $k = 2$ (we have to have at least two states, since there is a single accept state that's different from the start state). In fact, we know exactly what our GNFA should look like — there should only be one arrow from the start to accept state, with a regular expression on it. The language of that GNFA is simply all the strings that regular expression satisfies; so we can take $R = r$.



Now we'll prove the induction step — we'll prove the statement for $k > 2$ assuming we've already proven it for $k - 1$.

The idea is as follows: suppose we have some k-state automaton in the convenient form.



We'll convert it to an equivalent $(k-1)$-state GNFA (also in the convenient form), by removing one of the states.

Choose some state x, and simply remove it (along with any transitions that touch it); now we have a new GNFA.

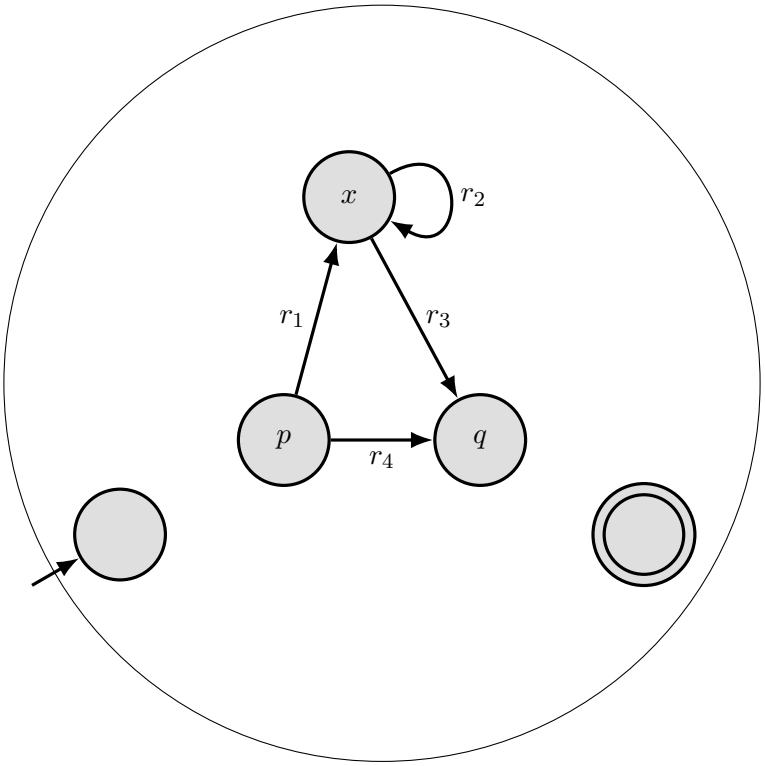
There's good news and bad news. The good news is, now we have $k-1$ states; the bad news is it might now be doing a different language (so we may have broken the automaton). So the idea is now we need to adjust all the transitions on the new GNFA to compensate for the absence of the removed state. Then we're going to use induction to say that we already know how to convert $(k-1)$-state GNFAs to regular expressions, and we're done. (If you don't like induction, you can think of it as — suppose I start out with a 4-state GNFA. This construction shows us how to convert it to a 3-state one. Then we can apply the same construction to convert it to a 2-state GNFA, and use the base case to read off the answer.)

So the name of the game is to see how to repair our automaton. We'll go through all the transitions and adjust them to give us the same behaviour we would have had if x was present.

First, we assume the state we remove is not the start or accept state (if you remove the start state, I can't fix that). Fortunately, $k > 2$; so there has to be some other state besides the start and accept state, and we pick one of those to remove.

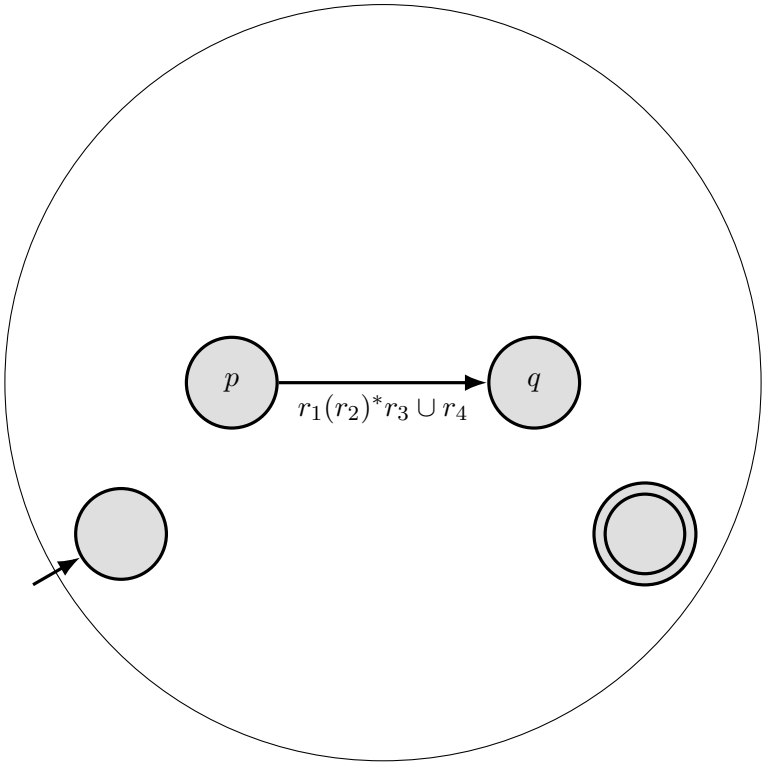
Let's suppose we have two nodes p and q, and draw in the relevant transitions — suppose the transition $p \to x$ was labelled r_1, the self-loop at x labelled r_2, the transition $x \to q$ labelled r_3; and the transition $p \to q$ labelled r_4.

(We'll do this for every pair p and q — for every transition that's left, we'll update it in this way. We might even have $q = p$.)



How do we get from p to q going through x? To do so, we have to read some string described by r_1, follow it by some string described by r_3. But actually, we might have stayed in x for several moves using the self-loop — so this could have been followed by some number of strings described by r_2. Nicely, we can capture this with regular expressions as $r_1 r_2^* r_3$; and that captures all of those strings that would have gone through x before returning to q.

So we're going to add those in as additional strings that can take us from p to q, on top of the original (r_4) — so our new label is $r_1(r_2)^* r_3 \cup r_4$.



And we do this for every transition.

This gives you a way of converting GNFAs to regular expressions (though they get very big very fast). (A GNFA is a generalization of a DFA, so if we can do GNFAs then we can certainly do DFAs.) □

Remark 3.6. What about other transitions, e.g. $q \to p$? There are a lot of states that we leave out. But when we do this, we're only looking at the relevant transitions for going from p to q. It's this particular transition that we update. We'll also have a transition $q \to p$, and that'll be updated in the same way.

There are more complicated paths, like going from p to x to p to x to q; but those strings are captured first in the new self-loop at p (we update that to capture p to x to p) and then the new transition $p \to q$ (updated with p to x to q).

Intuitively, we're just replacing the label with the strings we lost when deleting x that would take us directly from p through x to q. The other states and transitions, we handle by applying the same process to them.

Remark 3.7. Why did we need the convenient form? The real reason is for the base case — we don't want there to be reverse transitions or self-loops, because then we kind of have a mess.

Remark 3.8. What if we go from p to x to t to x to q? That's not captured by updating the p to q label; but it is captured by updating the p to t and t to q labels. The point is that it's enough to just consider the runs of x's (and remove them).

Remark 3.9. When we talk about $L(R)$, we use the notation L either on a regular expression or a finite automaton, to denote the corresponding language — L of an automaton, expression, or grammar is the set of strings it describes. Sometimes we suppress that and just say the regular expression equals the language it describes — if you have $3 + 4$, sometimes you mean 7 and sometimes you mean the expression $3 + 4$ itself, so we're trying to separate those. (Here the analog of the regular expression would be $3 + 4$, and the analog of the language it describes would be 7.)

We've now proved the equivalence between regular expression and finite automata. We'll now move on to a different topic — proving languages are *not* regular.

§3.2 Proving langauges irregular

One theme of this class is understanding the powers of these models, and that means seeing the things they *can* do, and the things they *can't* do.

Example 3.10

Let $\Sigma = \{0, 1\}$. Let $B = \{w \mid w \text{ has an equal number of 0's and 1's}\}$. Is B regular?

Given a string, can a finite automaton test whether the number of 1's equals the number of 0's? If you try for a while, you see pretty fast that you're not going to succeed — intuitively, counting the number of 0's you've seen so far requires an unbounded number of states (you can't represent the count of 0's compared with 1's that you've seen so far).

You can make that into a proof, but it's a bit dangerous; we'll illustrate that with another example.

Example 3.11

Let $C = \{w \mid w \text{ has an equal number of 01 and 10 substrings}\}$. For example, 0101 has two 01 substrings and one 10 substring, so it is not in C. Meanwhile 0110 has a single 01 and a single 10, which means it is in the language. Is C regular?

It looks like the answer is no for the same reason — how are you going to count the number of 01's to compare to the number of 10's? But surprisingly, this language is regular; you just have to look at it in a different way (left as an exercise).

So the moral of this story is — when we're asked to prove that a language is not regular, it's not good enough to say that we tried really hard to find a finite automaton and we couldn't. (That's not a proof — sometimes there's an intuition behind it that we can turn into a proof, but just because there doesn't seem to be an obvious way of building a finite automaton doesn't mean one exist. There are lots of examples where people think they've solved famous problems — I couldn't think of how to make a machine that does a certain thing, so therefore there must not be one. There is a story which Prof. Sipser will tell us later.)

So what we need is a method for proving rigorously that languages are not regular; that is called the pumping lemma. The pumping lemma gives a property that all regular languages share; it's a bit technical looking, but not too bad. To show a language is not regular, we just need to show it doesn't have that property, and then we're done.

Lemma 3.12 (Pumping lemma)

If A is regular, then there is a number p, called the *pumping length*, such that if $s \in A$ and $|s| \geq p$ (here $|s|$ denotes the length of s), then we can write $s = xyz$ such that:

1. $xy^iz \in A$ for all $i \geq 0$.
2. $y \neq \varepsilon$.
3. $|xy| \leq p$.

What is this saying? If you have s in the language (think of it as being a really long string), we can then take s and break it up into three pieces x, y, and z. Then we can take that middle piece y and repeat it as many times as we want — instead of y we can take yy, or yyy (to get e.g. $xyyyz$) — and that string is still going to be in the language. So for long strings in the language, there is some way to 'pump' them (think of taking that section, hooking up a bicycle pump, and making the string longer and longer — pumping it — by repeating that section). So this says we can expand the string by repeating this internal portion.

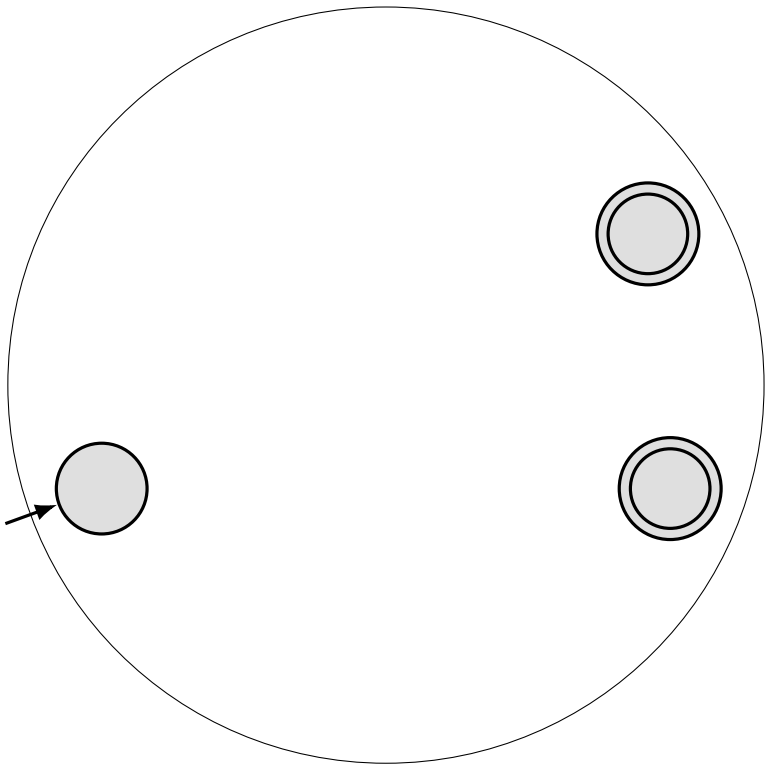
Then the pumping lemma says that in a regular language, all long strings in the language can be pumped, and we stay in the language.

The $i = 0$ case corresponds to removing y altogether; so xz is in fact also in the language.

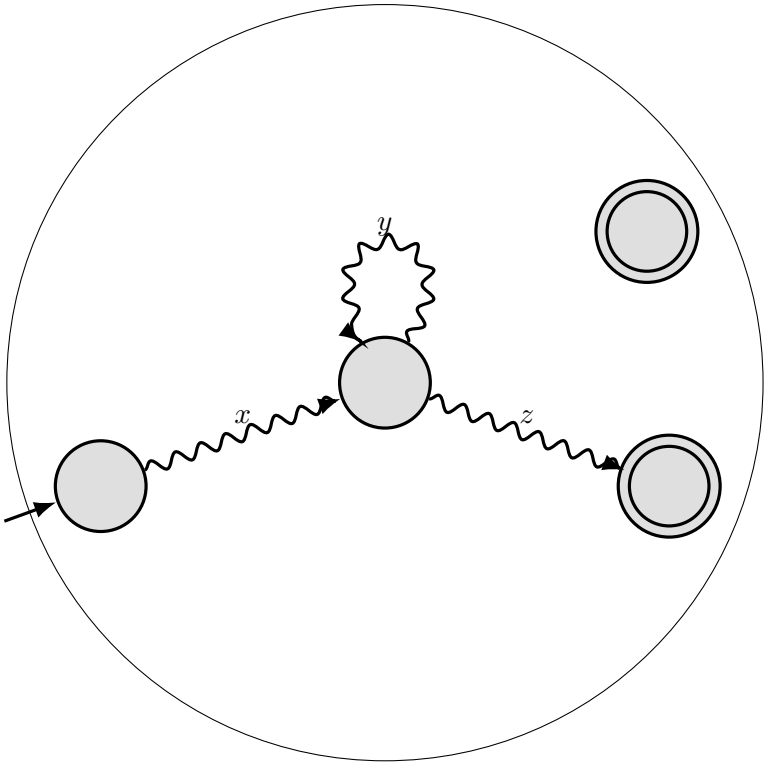
The condition $y \neq \varepsilon$ is needed — x and z can be empty, but y cannot — because if we let y be empty, then (1) wouldn't do anything for us. So we need y to be nonempty for this lemma to be at all meaningful.

Finally, $|xy| \leq p$ is mainly for convenience when we use the pumping lemma to prove languages are not regular, and is not as essential.

Proof. Let M be a DFA for A, and imagine feeding in a long string; what will happen?



If we feed in a really long string and follow the path that M takes, we'll have to repeat some state — if M has 100 states and we feed in an input which has 101 symbols, there's going to be some state that gets repeated.



We call x the portion that takes us from the start state to the first repeated state; y the portion taking us from the repeated state back to itself; and z the portion that takes us from that state on.

It's kind of clear why this automaton not only accepts $s = xyz$, but also $xyyz$ — that just keeps going around that loop as many times as you have copies of y (or even if you have no copies of y, it's still going to be accepted because you excise that cycle, and you still have z taking you from the repeated state to the accept state).

Finally, we have the question about lengths. We let p be the number of states of M (that's the minimum size we need for this reasoning to keep in — because if you have a string of length at least p, we have $p+1$ states that the machine enters, so there has to be a repetition).

Clearly y is nonempty. How can we limit $|xy| \leq p$? We take the very first repetition — even among the first p letters of the machine, we've gone through $p+1$ states, so a repetition has to already have happened. □

Example 3.13

Let $D = \{0^m 1^m \mid m \geq 0\}$ be the set of strings consisting of a run of 0's followed by the same number of 1's. Show that D is not regular.

We'll have a crisp proof (not 'I tried to make a finite automaton and couldn't') — we'll do this by showing D fails the condition of the pumping lemma.

Proof. Assume for contradiction that D is regular, so the pumping lemma applies to D; let p be the pumping length for D. Now we have to find some string s in D (our language) which has length at least p, and show that we cannot pump s; that'll give us our contradiction.

Let $s = 0^p 1^p$ (so s consists of a bunch of 0's followed by a bunch of 1's). The pumping lemma says that there is a way to break $s = xyz$ satisfying the three conditions. In particular, x and y must be in the first part of s; so there is some way of breaking

$$s = \underbrace{00\cdots0}_{x} \mid \underbrace{00\cdots0}_{y} \mid \underbrace{0\cdots0111\cdots1}_{z},$$

such that we can repeat y and remain in D. But repeating y increases the number of 0's, without touching the number of 1's. So we get a string ($xyyz$) that has more 0's than 1's, and is therefore not in D; that's our contradiction. □

Remark 3.14. The pumping lemma says there has to be some way to break s; we showed here that there is no such way (no matter how you try to break s up, you will fail); so we have a contradiction, because the pumping lemma isn't applying the way it's supposed to. It's not enough to show that *some* way to break it up fails; you have to show that *every* way to break it up fails. (Here we chose a string for which all the cases look essentially the same — if we didn't use condition (3), then we'd have to deal with more cases, depending on where y is — if it's in the 1's then we have more 1's than 0's after pumping, and if y contains both then we'll have 0's and 1's out of order.)

Remark 3.15. Are there languages that are not regular but do satisfy the pumping lemma? There are such things; it's an interesting exercise to come up with them, and to prove they're not regular you'll have to use different methods.

There's another technique, that combined with the pumping lemma, gives you tremendous new power — and that's to use the pumping lemma together with the closure property. If I want to show that the earlier set B, consisting of strings with equal numbers of 0's and 1's, is not regular, we can go back and use the pumping lemma proof (using the same argument). But here's a different argument, that's a 1-line proof: what's the difference between B and D? Both require the number of 0's and 1's to be the same, but in B they can be in any order. So we can observe that

$$D = B \cap 0^*1^*.$$

We proved that the regular languages are under intersection (as a side remark in our first lecture, when we proved closure under union). So if B were regular, since 0^*1^* is obviously regular, then $B \cap 0^*1^*$ would also be regular. But we know D is not regular.

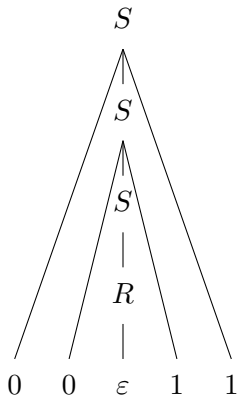
(This is very handy for getting very short answers to certain problems — by intersecting or unioning or taking complements with regular languages, we might be able to get something which is obviously not regular. So if the original was regular, doing this would keep it regular; if the end result is not regular, then we know the original could not have been regular either.)

§3.3 Intro to Context-Free Grammars

We'll now look at a different model of computation. These look like:

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

The meaning of this is that you start with the starting point S, and these are 'substitution rules' that allow you to substitute one of these capital letters (which denote variables) on the LHS by the corresponding RHS of the rule. The non-variables are constants, called *terminals*. So this substitution rule says we can do



So now we have $S \to 0S1 \to 00S11$. Then we can also substitute S for R, and R for ε; now we get $00R11 \to 0011$. Now there's nothing left to substitute; that means we've gotten something in our language.

You can see that all possible strings you get in this way are precisely the language D — strings of 0's followed by an equal number of 1's. We are encouraged to take a look at 0.4, which gives a lot of practice working with them.

§4 September 19, 2023

§4.1 PS comments

The homework is due Thursday at *noon*, on Gradescope. Hopefully we have gotten started on it.

The problems are intentionally hard. You should not feel bad if you find them hard — they are intended to not just be solved in a couple of minutes (they're not routine applications of the ideas in the course — you have to come up with some creative idea to solve them). But it's intended they should not be too hard to write up — Prof. Sipser isn't looking for long multi-page solutions, we should just try to capture the main ideas in a few paragraphs. After this problem set, we'll distribute the solutions; those should work for us as model solutions going forwards, to get a sense of the level of detail we should provide.

Today we'll cover the material for problem 6 (we've already covered the material for the other problems). In the case of the grammar, there is a grammar that's not very long; it seems hard at first, but we can play with it and try to understand the language that the problem is referring to, which will help us come up with both the grammar and the PDA. For the PDA, it's enough to give an informal description of how it works (we can write down the formal states and transitions, but it'll help the grader if we explain how it works).

There's lots of comments on Piazza; we are encouraged to look at it (there are some helpful suggestions for how to get going on some of the problems).

§4.2 Review

Last time we wrapped up our discussion of finite automata. We proved the equivalence of finite automata and regular expressions, and introduced the pumping lemma as a method for proving irregularity. Sometimes that's coupled with the closure properties (regular languages are closed under union, intersection, complement, concatenation, and star — note that irregular languages are only closed under complement, and not the other ones).

§4.3 Context-free Grammars

Today we'll continue with the discussion of context-free grammars that we started last time.

Example 4.1

One example of a CFG G_1 is given by the following three substitution rules:

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

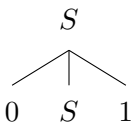
In each substitution rule, we have a *variable* mapping to a string of variables and non-variables, called *terminals*. Here the variables are S and R, and the terminals are 0 and 1. (ε is the empty string, which is neither.) We'll use the shorthand

$$S \to 0S1 \mid RR \qquad\qquad \to \varepsilon$$

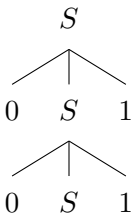
to capture the first two rules in one line (it means the same thing).

To use a CFG, it generates a language of strings; similar to regular expressions, it's a generative (rather than recognizing) grammar.

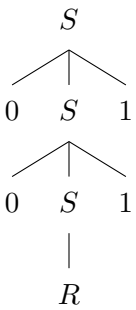
We take our grammar, and write down a designated start variable (typically the left-side variable of the topmost rule), which here is S. We then use the rules to perform substitutions on the variables we have currently. Currently we have a S, so we can replace it with either $0S1$ or R; let's do the former.



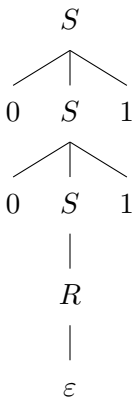
Here we still have a variable, so we can substitute it again:



Now we have $00S11$. We can now substitute this to R:



Then we have to substitute R with ε.



The resulting string 0011 has no variables left, so that's a string in our grammar.

The strings in our grammar are all that can be generated in this way. (Of course, we might have multiple possibilities for how to do a substitution, which can give multiple strings coming out.) So we have

$$\mathcal{L}(G_1) = \{0^k 1^k \mid k \geq 0\}.$$

(This was our first example of a non-regular language; we can get it with a CFG.)

Here's the formal definition:

Definition 4.2. A context-free grammar (CFG) is a 4-tuple $G = (V, \Sigma, R, S)$ where:

- V is the set of *variables*;
- Σ is the set of *terminals*;
- R is a set of *rules*;
- $S \in V$ is a designated *start variable*.

We can also think of Σ as the alphabet of the generated language — that's why we use the same symbol as before. (We're going to keep performing substitutions until we're only left with a string of terminals.)

Here's another, somewhat more interesting, example o f a grammar:

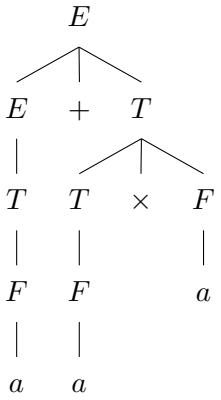
Example 4.3

Consider the CFG G_2 given by

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

Here the variables are $V = \{E, T, F\}$, and the terminals are $\Sigma = \{+, \times, (,), a\}$. (These are all formal symbols; they have no meaning in the way we normally think of these things.) We have 6 rules in this grammar (each line has two rules — you can read the first line as 'E goes to $E + T$ or T').

For example, we have $a + a \times a \in \mathcal{L}(G_2)$ — in order to generate it, we perform



Remark 4.4. CFGs are going to be another warm-up model for us, but they come up a lot in applications — in linguistics as a way of modeling human languages, and in applied computer science (e.g. compilers) as a way of formally describing a programming language. In fact, if you give a grammar that formally describes your new programming language, there are tools that will automatically generate a parser (that will develop a parse tree from the input and grammar); you can use this to generate the object code from the input code and stuff.

This kind of object is called a *parse tree*. It contains information that captures something about what you're trying to represent with the given string. In this parse tree, if we look at its structure, it kind of groups together the $\times$ more tightly than the $+$ — in a sense, this grammar and associated parse tree has an implicit precedence of $\times$ over $+$ — unless we put in parentheses (as in $(a+a)\times a$), which will give a different parse tree. That's part of what this language is doing.

We call all languages that can be generated with CFGs as *context-free languages*:

Definition 4.5. We say A is a *context-free language* if $A = \mathcal{L}(G)$ for some CFG G.

These are analogous to regular languages, but for CFGs instead of finite automata or regular expressions.

We got into a brief discussion of the parse tree and the meaning behind the strings that the language may be generating. But sometimes when you have a CFG, there may be multiple ways of getting the same string to come out, using different parse trees — that's entirely legal and possible.

Example 4.6

Take the grammar G_3 with the rules

$$E \to E+E \mid E\times E \mid (E) \mid a.$$

This grammar G_3 generates the same language as G_2. But there's an important difference here, because for G_3, the string $a+a\times a$ is generated by two very different parse trees — one that binds $+$ more tightly than $\times$, and one like our earlier tree that binds $\times$ more tightly than $+$. (There'll be a symmetric tree to this one in G_3 which has $+$ lower down than $\times$.)

Definition 4.7. We say a grammar which generates a string using multiple parse trees generates that string *ambiguously*.

If you think of the parse tree as representing meaning, then this means there's multiple meanings of the string. (A lot of this came out of linguistics, e.g. by Noam Chomsky.)

This corresponds to ordinary language — English (and other languages) is very ambiguous, even in ordinary conversation.

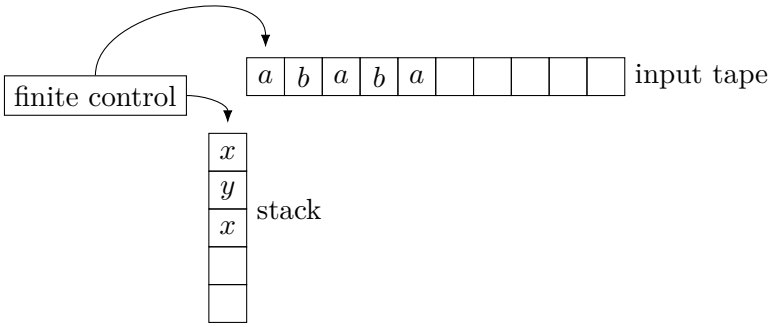
Example 4.8

If we say *I touched the dog with a bone*, there are two possible interpretations — either I used a bone to touch the dog, or there are multiple dogs and I touched the one which has the bone. There are two interpretations, and this corresponds to having two different parse trees.

§4.4 Push-down Automata

We're going to follow a similar pattern (in the other order) as we did for finite automata — we're also going to give an automaton model that captures context-free languages. (We had two ways of capturing regular languages — with finite automata and regular expressions, one of which is a recognizer and the other a generator.) We'll do the same here. We gave the generator first; next we'll give a machine model which recognizes exactly the context-free languages. This model will be called a *push-down automaton*.

First we'll draw a PDA schematically.



(The 'finite control' box represents the part of our automaton with states and transitions.)

We have an input tape, and a head that can read it one symbol at a time, the same as in a DFA or NFA. But here we have an extra device, a *stack*. This is also a kind of tape, but it's used in a specific way. We have symbols that can appear on the stack (which might be different than the input symbols).

The stack is going to be a device for the machine to be able to record information — so we can not only read from the stack, but also write on it. But we can only write in a special way — we can only write a symbol at the very top of the stack. So we can write a y at the top of the stack, and that'll sit on top of the x — you can think of this as 'pushing down' the other symbols one level, so that now the y is the symbol sitting at the top and we have xyx below it.

So the machine can only write by putting symbols at the very top and pushing the rest down. Meanwhile, it can read only by looking at the symbol at the very top and removing it; then everything else pops up.

We call reading from the stack *popping*, and writing *pushing*. When we read, we always take the symbol from the top, *remove* it, and look at it; when we write, we take the symbol we have in mind and put it there, and it pushes everything else down.

Example 4.9

Construct a PDA for $\{0^k 1^k \mid k \geq 0\}$.

Imagine we get an input; the PDA should check if it consists of a run of 0's followed by an equal number of 1's. We certainly need to use the stack for this — without the stack, we'd just be a finite automaton, and we already proved a finite automaton can't recognize this language.

So we need to use the stack, and we'll do so in the following way:

- We first read the input symbols. If we see a 0, we push on the stack.
- If a 1 appears on the input, then we read the 1's and pop 0's off the stack.
- Finally, we accept if the stack is empty.

So if we get an input of 000111, we read from left to right and push the 0's onto the stack until we see a 1. Then we start removing the 0's — that's how we compare the lengths of the run of 0's with the run of 1's. If we empty the stack, then we accept. (Acceptance only applies when you're at the end of the input — if we ran out of 0's, then we'd enter an accept state, but we wouldn't be able to read the last symbol because there's no more 0's to pop; so the machine would get stuck and not accept. The machine has to accept at the *end* in an accepting state.)

(For a string like 00011101, we don't want to accept it. This is because we can only read 0's, and then we go into the second stage where we can only read 1's — the machine reads 0's initially, and then reads 1's and matches with 0's. If it gets a 0 at this point, then it can't do anything and it just fails. It's like a NFA — if there's nowhere to go on a particular input, then the machine just dies.)

Formally:

Definition 4.10. A PDA is a collection $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$ where:

- Q is a set of *states*;
- Σ is the *input alphabet*;
- Γ is the *stack alphabet*;
- $\delta: Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to \mathcal{P}(Q \times \Gamma_\varepsilon)$ is the transition function.

The stack alphabet Γ is kind of a convenience — we'll often want to write different symbols on the stack as in the input, though we could just use the input alphabet if we wanted to.

The transition function tells us how the machine operates. So where should the function be from? It tells us what to do when we're in a given state (the element of Q), and we read an input symbol (the element of Σ) and a stack symbol (the element of Γ). So we view this as a function on $Q \times \Sigma \times \Gamma$ (since the input symbol and the symbol on top of the stack can affect what we do). Then we go into a new state, and possibly put a new symbol on top of the stack — we watn to represent this as $Q \times \Gamma$. But we're not done yet — there's several other things we should do.

First, PDA will be nondeterministic — unlike finite automata, deterministic PDA are actually weaker, and we need nondeterminism to get equivalence with grammar. So we're only going to work with nondeterministic PDA (and our PDA are always assumed to be nondeterministic). (There's a section of the book about the power of deterministic PDA, but we will skip this.)

The way we get nondeterminism is by writing down $\mathcal{P}(Q \times \Gamma)$ instead — when we're in a given state, and we read a given input symbol and pop a certain stack symbol, there might be several choices for what we do next (we might go to any one of a bunch of states, and for each of these we might push a different symbol).

There's a few more refinements. In NFAs, we allowed the machine the ability to not read an input symbol on a given move — it can do a 'free' move. We represent this by replacing Σ with Σ_ε — this means the machine can take either a symbol from Σ or the empty string, and can do things accordingly. Similarly, we allow the machine not to read anything from the stack. (This might seem excessively formal, but when we're actually using this it's useful.) Similarly, we allow the machine to not push something onto the stack if it doesn't want to.

So at any point it can read a symbol from the input or skip reading, pop a symbol from the stack or leave the stack alone, and then it can go into a new state and either push something onto the stack or not.

Example 4.11

Let $C = \{ww^{\mathcal{R}} \mid w \in \{a, b\}^*\}$. (Here $w^{\mathcal{R}}$ is the reverse of w, so C consists of strings consisting of a bunch of symbols and then the same symbols in reverse — i.e., palindromes of even length.) Find a PDA that recognizes C.

For example, the input $abaaaaba$ is in the language (taking $w = abaa$).

The way our PDA will work is: First, we read a symbol and push it onto the stack; we repeat. So on $abaaaaba$, we'll first push a, then b, then a, then a.

At some point, our machine has got to shift gears and start popping off the stack. The reversal will happen for free, since stacks always reverse stuff (if we push things in and then pop them out, they come out backwards). Here we'll have pushed $abaa$, so we have $aaba$ sitting on the stack; then we want to start comparing the stack with the rest of the input.

But we need to do this when we're in the middle. How do you tell when you're in the middle? You can't, and that's what nondeterminism is for.

So at every step, we might nondeterministically shift into the popping stage, where we start popping things off the stack and comparing with the rest of the input.

Here's the PDA description:

(1) Read an input symbol and push it onto the stack. Repeat, or nondeterministically move to (2).

(2) Read the input, and compare it with the popped symbols. Automatically reject if they're not equal, and accept if the stack is empty.

There are going to be different threads of the nondeterminism here. Whenever the machine has a nondeterministic choice, the entire machine replicates to follow the different threads — in particular, each thread has its own stack. So when we've read up to aba and have aba on the stack, now nondeterministically, the machine says 'Hmm, maybe this is the midpoint,' so there's going to be some thread of the nondeterminism that transitions to phase 2 — it'll read the next symbol, compare it to the top of the stack, and be happy for now (since both symbols are a). But its fate is not very auspicious going forwards — when it reads the next symbol a it'll see a b popped off; these are unequal, so that thread dies.

Fortunately, there's going to be some other thread which stayed in the pushing phase for longer; that thread is going to have $abaa$ on the top of its stack. (This is a different thread, and it has its own stack.) Now on that thread, when it comes off, it'll match off with the remainder of the input successfully; so the machine will end up accepting.

Remark 4.12. If we have $abaaba\cdots$ (where $\cdots$ is nonempty), there'll be some thread of the automaton that reads aba, then pops while reading aba, and enters an accept state. But acceptance only counts when you're at the end of the input (just like a NFA) — if it goes through an accept state along the way, that doesn't matter. Here we'll be in an accept state too early, so that doesn't matter — the machine won't be accepting the input.

PDA as we've defined them have no mechanism for detecting the end of the input. But actually, there are two things you can do with PDA that aren't in the definition.

The PDA can't tell natively whether the stack is empty or not — that's not one of the features we've built into it. But you can get the same effect — if you need that, when you're programming a PDA, you can start off by writing a special symbol on the stack that marks the very bottom of the stack. Every time you

see that symbol at the top, this tells you the stack is effectively empty. (This is like getting the emptiness test for the stack as software rather than hardware.) There's a similar, slightly more complicated, way of being able to tell that you're at the end of the input.

§4.5 Equivalence of CFGs and PDAs

We'll now move to our big theorem of the day — the equivalence of context-free grammars and push-down automata.

Theorem 4.13

A language A is a context-free language if and only if $A = \mathcal{L}(P)$ for some push-down automata P.

We defined CFLs in terms of context-free grammars; so another way of saying this is that A is the language of some CFG if and only if it is the language of some PDA (or the CFGs and PDAs give you the same class of languages).

Proof. We're going to convert CFGs to PDAs and vice versa. In fact we're only going to prove one direction; the other direction is a bit too much of a chore. (It's in the book; we're not responsible for it, except to know that it's true.)

We'll prove the rightwards direction by converting any CFG to a PDA. (The reverse direction is a similar idea, but we won't do it and we're not responsible for the proof; it's in the book.)

Suppose we're given some CFG G; we're going to construct a PDA P for it.

Suppose we have a grammar G given by

$$\begin{aligned} S &\to \cdots \mid \cdots \mid \cdots \\ R &\to \cdots \mid \cdots \\ T &\to \cdots \end{aligned}$$

and so on. We want to somehow convert that into a PDA.

Here's how it'll informally work. First we're going to cheat. What we're going to do is start off by writing the start symbol to the top of our stack. (That's okay, it's not cheating.) But now what we're going to do is take that stack, and every time I see a variable on that stack, I'm going to apply one of the associated substitution rules, just like you would normally do when trying to generate something with the grammar. So the PDA is going to look *anywhere* in the stack for a variable (that's where the cheating comes in), and if it sees a variable anywhere, it substitutes for that variable by squeezing in the substitution in its place (this is more cheating). Once all the variables are gone, it's going to take what remains on the stack, and compare with the input.

What we just described is inherently nondeterministic, and it does work if you allow the cheating (we'll have to figure out later how to deal with the cheating). Every time the PDA sees a variable, there might be several substitutions that could apply; it nondeterministically chooses one of them and sticks that in place of the variable. This is exactly what you do when you're using the grammar to generate the string.

Example 4.14

In the grammar G_2, we first see E on the stack; we then substitute this by $E+T$ (on the stack with E on the top — we have our order come out so that when we read it, it'll correspond eventually to the order of the input). Now another move we could do, which is cheating, is to take its head, scurry down to the T, take that T, and change it to $T \times F$ (so now we have $E+T \times F$ on the stack, with E at the top). We keep doing this; eventually we'll end up with $a+a \times a$. Now there's no variables; the machine says 'Whew, I hope no one caught me cheating,' and now takes a look at the input — does the string I generated on the stack equal the input? So it reads the input, popping off the stack, and accepts if they're the same.

So some thread will work if the input is really in the language — intuitively, the threads are looking at all the possible substitutions (this is exactly what the nondeterminism is — you can think of it as parallelism, or as guessing what is the right substitution to do).

But we're cheating; how do we not cheat? This is what we're going to do instead: we only allow ourselves to substitute a variable if it's at the very top of the stack. (So we don't have any scurrying down inside the stack — we won't need that any more. Only if there's a variable at the top of the stack will we substitute for it.)

How do we get to the guys lower down? As long as there's a variable at the top of the stack, we'll keep substituting. But if we get to a terminal symbol at the top of the stack, then we're going to compare that symbol with the input — because that terminal symbol is fixed (it's not a variable, so it's never going to change).

So as the terminal symbols appear on the top of the stack, we match them off the input until a variable appears again; and then I do a substitution. That avoids needing to go down into the stack and cheating to do the substitutions down below — because we only need to do substitutions when the variables bubble up to the top (and if there's not a variable at the top, then there's a terminal and we compare it to the input).

In other words, here's how P operates.

(1) Write the start variable on the stack.

(2) Pop the stack.

   (a) If the symbol is a variable, then substitute using some rule (chosen nondeterministically). So for example, if we have E at the top of the stack, we can replace it with either $E+T$ or T.

   (b) If the symbol is a terminal, then we compare it with the next input. We reject if they're not equal (the thread will die if the symbol it popped off doesn't match the next input symbol, since this means something has gone wrong with the nondeterministic choice — that thread is not finding our input).

   (c) If the stack is empty, then we accept (again, using the same fact that the acceptance only takes effect if we're at the end of the input).

(With nondeterminism, we don't need a special 'reject' state — we can just have the thread die, by having no transition with the corresponding information, similarly to a NFA where we 'take our finger off' — when we implement, we just only allow the thing to proceed if the symbols are equal. You *could* make a state where the machine always loops back to itself as a permanently dead state, but it's not needed.)

One detail is, how do we push multiple symbols in one step? (When we do $E \to E+T$, the way the PDA is defined, it can only push a single symbol.) This is implemented in software — we have a sequence of states the machine goes through, where at each step it doesn't read any input and just pushes the next symbol of the sequence you have in mind.

(We have this formal underlying model, but we can talk about it informally; we and the graders will not be fussy about the details, and we can just talk about pushing the whole string onto the stack.) □

§5 September 21, 2023

There's going to be a new problem set on the course homepage this evening.

To recap what we've been doing, we've been looking at CFLs and their associated models — CFGs and PDAs. We showed how to convert grammars to PDAs; we should be aware that the conversion can be done the other way as well. This has some important consequences, but we won't go through the proof (it's a nice proof; we can look at it in the book).

Today we'll develop a method for proving languages are *not* context-free. (A lot of this course is about testing the limits of models — understanding what we can and cannot do.) We'll then introduce our next model, Turing machines.

Last time, we saw that A is a CFL if and only if some PDA recognizes A (where we defined CFLs in terms of grammars — so A has a CFG if and only if A has a PDA). Last time we showed how to convert grammars to PDAs; we won't do the other direction, but it has the following consequences.

Corollary 5.1

Every regular language is a CFL.

This is because every regular language has a DFA, which is a type of PDA.

Remark 5.2. There are certainly CFLs that are not regular languages — for example $\{0^k1^k \mid k \geq 0\}$. So CFLs are definitely a bigger class, and the models are more powerful than the models for regular languages.

There's another useful fact that comes up in proving that languages are *not* context-free (in the same way we used it to prove languages are not regular):

Corollary 5.3

If A is a CFL and B is a regular language, then $A \cap B$ is a CFL as well.

Remark 5.4. In general, the context-free languages are *not* closed under intersection or complement (so the intersection of two CFLs is not necessarily a CFL). They are, however, closed under the regular operations $\cup$, $\circ$, and $*$.

§5.1 Proving non-context-free languages

As a prototypical example of a non-context-free language, let

$$B = \{a^k b^k c^k \mid k \geq 0\}.$$

If we didn't have the c's, then we'd have a CFL. But when we have runs of a's followed by b's followed by c's, it goes beyond the power of what we can do with a CFG or PDA. To prove this, we'll develop a pumping lemma for CFLs.

We will show that B is not a CFL.

Theorem 5.5 (Pumping lemma for CFLs)

If A is a CFL, then there is a pumping length p such that if $s \in A$ and $|s| \geq p$, then we can write $s = uvxyz$ such that:

(1) $uv^ixy^iz \in A$ for all $i \geq 0$.

(2) $vy \neq \varepsilon$.

(3) $|vxy| \leq p$.

This will look very similar to the pumping lemma for regular languages, but with some important differences.

As before, this says that if s is in our language and it's long, then we can split it up — here, into five pieces instead of three pieces — such that we can pump a piece (the way we break it up and do the pumping is a bit different). (The second condition states that at least one of v and y is nonempty.)

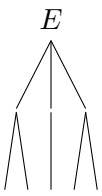
Let's get a feeling for this. If we have s in our CFL A, then we can break s up into five pieces. Then we can take the second and fourth pieces, and repeat them — but we have to repeat them the same number of times (and then we stay in the language). So for example, $uvvxyyz$ is in the language.

So as before, all long strings can be pumped; it's just that the nature of the pumping is different (we have five pieces instead of three).

Conditions (2) and (3) play the same role as they did before. For (2), we want to make sure that (1) says something interesting — if v and y were both empty, then (1) wouldn't be interesting, because repeating them wouldn't change the string. So we use (2) to tell us that at least one isn't empty. And (3) is for convenience in using the pumping lemma — this is because to show that a language is not context-free, we'll typically show that it fails the lemma (we'll argue by contradiction — we'll say that if our language were context-free it would have a pumping length, and then we'll cleverly choose some long string in the language and show there's no way to cut it up). This condition limits the number of ways our string can be cut up, giving us a simpler proof in the end.

Proof. Suppose we're given a CFL A and a long string $s \in A$ (we'll work out the numbers later, but for now think of s as being some long member of A). Our goal is to show that we can break s up into five pieces.

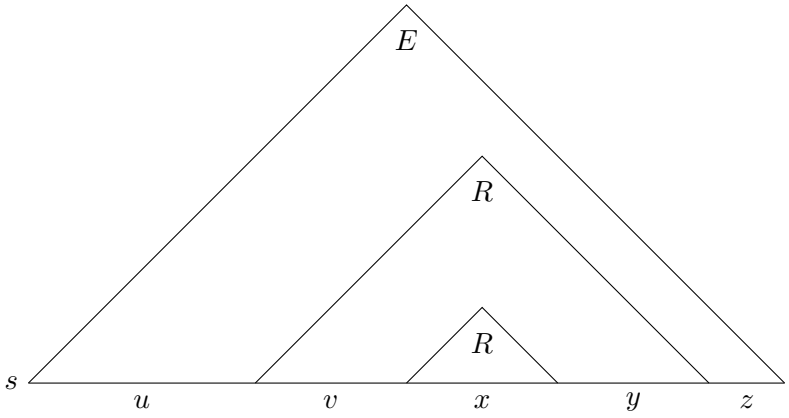
Let's take s, some really long string. We know s is in A. So we can now take some grammar for A, and look at the parse tree for S using that grammar: we start off wth some start variable E, do some substitutions, and eventually end up with s.



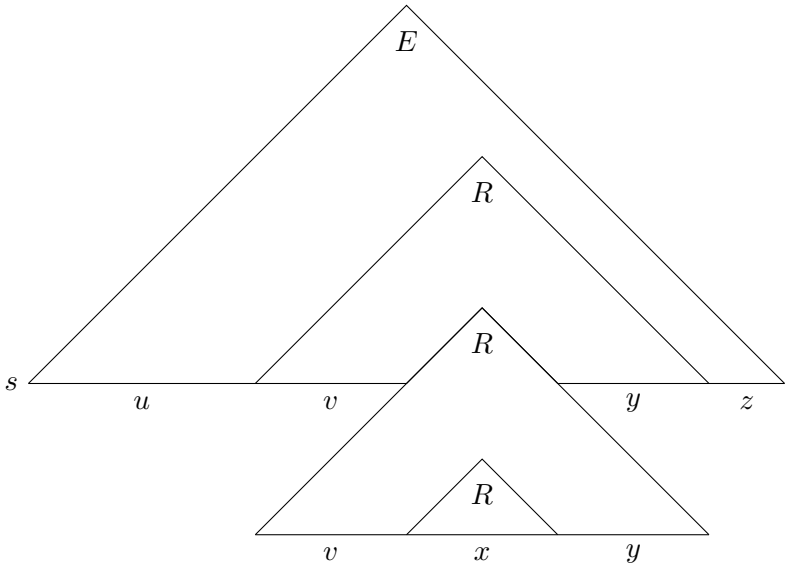
But if s is really long, the parse tree has to be really tall — because there's no way to get a really long s with a shallow tree (the tree can only branch out at a certain rate, so if the tree is really shallow, there's only a limited size s the tree could possibly generate).

Then if the tree is tall, there's got to be some long branch in this parse tree that goes through many variables. And if we arrange 'tall' to be big enough, that's going to tell us that the variables we go through have to repeat! In other words, we're going to start out with some grammar (e.g. with 20 variables); if s is long enough it'll have a parse tree longer than 20, so along this way we'll have some repetition that occurs.

Now we take this lower repeated variable, and look at the subtree we can generate just starting there — that'll be a portion of s. And then we take th upper repeated variable, and look at the larger tree it generates. This now tells us how to split s into five pieces.



How do we know that $uvvxyyz$ is still in the language? The way we'll do this is by taking the tree and doing some surgery on it, to get a new parse tree — we take our upper R and make a copy of its subtree, and instead of taking the original subtree at our lower R, we take this copy of the subtree from the upper R.



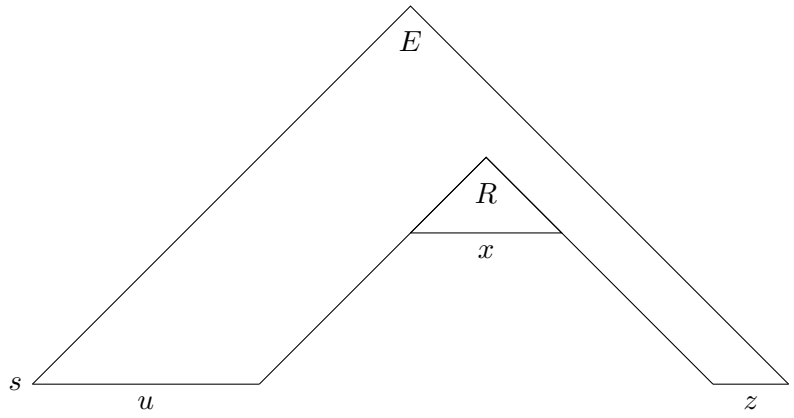
The lower subtree differs from the upper subtree just because we made different substitutions — so instead imagine we do the same substitutions for the lower R as we did for the upper R. Then we're going to change

what we get out by changing what substitutions we're doing — visually, what happens is that we took the upper subtree coming out of the upper R, and stuck it in place of the lower subtree.

Then our new tree will have u, the original v, and then another copy of vxy, and then the original y and z.

And we can do this repeatedly — we can take that bigger subtree and then stick it in for the new lower subtree, and we get a third v and a third y. And that's the whole idea.

It can also be used to get the case $i = 0$ (which is sometimes useful) — this is a reverse substitution, where we take the lower subtree and stick it in for the upper subtree. Then we just get $uxz = uv^0xy^0z$.



That's the idea; now we'll go through the details and the calculation, just to give us a sense of how it goes.

We have to calculate how big p needs to be to make this happen. How long does s really need to be? Our whole goal is to get a really tall tree, so we need to have a connection between the length of the generated string s and the height of the tree. And that has to do with the branching factor — we want to look at how much we can branch out of any one node. Then every level of the tree is multiplied by at most the branching factor, so if we have branching number b, then we have 1 node at the top, then (at most) b in the next layer, then b^2, and so on.

The branching factor comes from looking at the grammar — b is the maximum size of a right-hand side of a rule. For example, if we have $E \to E + T$, this would give us 3.

So then a tree of height h gives s of length at most b^h — every time we have another level of the tree, we can possibly multiply the result we would get by b.

So we let $p = b^{|V|} + 1$, where V is the set of variables. Then if $|s| \geq p$, we must have $h > |V|$, and so we're going to get a repeated variable R (on some path down from the top).

Now we need to prove (2) and (3). With what we've done so far, it's conceivable that v and y are empty — for example, we could have had $R \to T \to R$, in which case the two R's would generate exactly the same stuff. We don't want this to happen. The way we're going to fix this is by starting with the smallest possible parse tree for our string. If we had two R's that generated the same thing (so that v and y were empty), we could have formed a smaller parse tree (where we started off by merging these two R's to get a smaller subtree). So if we start with the smallest parse tree, then we can just prevent this.

To get (3), we start with the lowest possible repetition — if we have our long path and there are several different repetitions and we take the lowest, the most it can expand to is length p.

(The main idea in the proof is the picture; the rest is calculation.) □

Now that we've proved this, let's use it.

Example 5.6

The language $B = \{a^k b^k c^k \mid k \geq 0\}$ is not a CFL.

Proof. Assume for contradiction that B is a CFL, and get p from the pumping lemma. Now we need to find a string s where there's no way to pump it. Oftentimes we'll need to do this cleverly, but here the obvious choice works — let $s = a^p b^p c^p$. This is in B, and $|s| = 3p \geq p$. So the pumping lemma says something interesting about s (it's long enough for the breaking up to apply).

Now let's see why we can't actually break up s. Here it'll be helpful to use (3), which says that the three pieces v, x, and y together have length at most p — and we're repeating v and y. Here if altogether vxy has length at most p, then no matter where it occurs, it can only contain at most two letters — v and y together can't contain all three possible types of symbols (if they contain a's, they won't be able to reach out into the c's, since y is just not far away enough from v).

So if we repeat v and y, that'll increase at least one type of symbol, but it won't increase all types of symbols — so e.g. uv^2xy^2z will have an unequal number of a, b and c, and will therefore not be in B; this is a contradiction. □

We'll now do another example. Recall that we gave a pushdown automaton for the language $ww^{\mathcal{R}}$, and it's easy to give a grammar for it as well. But we can change that a bit:

Example 5.7

The language $C = \{ww \mid w \in \{0,1\}^*\}$ is not a CFL.

For example, C contains 11011101.

Proof. Assume that C is a CFL, and get the pumping length p. Now it's perhaps a little less obvious what the string s ought to be, so you have to think a bit about what's a good choice. One possibility is — we have to make sure s has length at least p, so as a start, we can try $s = 0^p 10^p 1$. This is in C. (This is actually a bad choice; the point of this discussion is to illustrate that we might have to try several possibilities.) The pumping lemma says that this string can be pumped, so we'd like to show it can't be. But unfortunately, this string *can* actually be pumped — suppose that vxy is the middle 010 (with $v = 0$, $x = 1$, $y = 0$). This satisfies all the conditions, but now we can repeat v and y, but we stay in the language — this isn't what we want. (If we repeat v and y, we're going to increase the number of 0's in the left half at the same time as in the right half, but that doesn't do anything to get us out of the language.)

Instead, we're going to try to use a little more of the complexity of the language C, and take the string $0^p 1^p 0^p 1^p$. Now wherever we place v, x, and y, they can only contain numbers from two consecutive blocks — v and y can't be so far apart that they're in corresponding runs. (If v is in the first run of 0's then y can't be in the second, for example.) So when we repeat the v's and y's, we'll end up out of the language (we won't go through the details carefully, but it's fairly straightforward to see this is not going to be pumpable). □

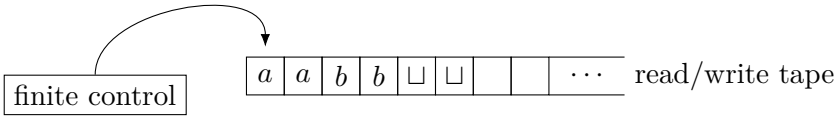
So these are examples of languages we can prove are not CFLs using the pumping lemma.

§5.2 Turing Machines

Turing machines will be the model of computation we'll spend the most time on; we'll now introduce them.

So far, all the models of computation we've looked at are restricted in some way that makes us feel like they don't capture the full power of computers. Turing machines are going to be a better approximation, in that sense, to the way we think about general-purpose computers.

We'll first describe them with a schematic diagram, as before.



As before, we have our finite control and a tape. But the main difference between Turing machines and finite automata is that we allow the Turing machine to *write* on the primary tape, not just read from it.

Then there's other modifications we need to make to build on that idea of being able to write. For one thing, we have to allow the machine to move its reading-head on the tape, in both directions — otherwise, if you can only move in one direction, then you can write stuff but you'll never be able to read it again.

So we have a 2-way head. We'll also think of this tape as being infinite, to the right side only — because the machine is going to use the tape as storage, and we don't want to limit the amount of storage it has. (Later we will talk about complexity and we will want to limit the storage, but right now we won't.)

The third difference is that since the tape is infinite, we have a special blank symbol ⊔ that fills the rest of the tape — the input sits to the left of the tape, and the rest of the tape is filled with this blank symbol. (We assume the input doesn't use this blank symbol, so you can use the first one you see as a marker telling you that the end of the input has occurred.)

The fourth feature (that follows from being able to write on this infinite tape) — in the models that we've discussed until now, when the machine enters an accept state, it only counts when it's at the end of the input. Now the machine has the ability to erase the input, so it might put other stuff there and move around and so on — so this doesn't make sense anymore, and it doesn't make sense to only look at the location where the input used to reside. So we now allow the machine to accept anywhere — when it goes into an accept state, it halts and accepts. (This is a bit different from how we treated finite automata and pushdown automata, where accepting only took effect at the end.) Also, before the machine stopped when it got to the end of the input, but now the machine can just keep going — so we now also give it the power to stop and reject anywhere.

Example 5.8

Find a Turing machine for $B = \{a^k b^k c^k \mid k \geq 0\}$.

Suppose our Turing machine is presented with $aaabbbccc \sqcup \sqcup \sqcup \sqcup \cdots$. How does our Turing machine use its ability to write in order to process this?

Suppose our Turing machine first takes its head and reads the input (it'll be able to come back and read it again), and checks that it is in $a^*b^*c^*$ (forgetting about the counts) — it can do this just with its finite control. (It can act like a finite automaton to check that the symbols are in the right order.) It rejects if not.

Then, we're going to return to the left end, and we scan across the input. But now we use our writing feature to cross off one a along the run of a's, then one b, and then one c. (We know our input is of the form a's, then b's, then c's.) By *crossing off* we mean we write a new symbol on top of the a, e.g. a crossed-out a.

We repeat until we run out of some symbol (we return back to the beginning and scan across again, crossing off an a, b, and c). We accept if at this point, we've run out of *all* symbols (meaning we've crossed out the last a, b, and c on the same scan), and reject otherwise.

This is a very simple Turing machine; of course, we can do many more complicated things. But this is an initial demonstration that Turing machines are much more powerful than what we've seen before.

§5.2.1 A formal definition

We'll now formally define Turing machines, similarly to the previous models. (This might be the last time we run through a formal definition, but it's nice to have precise definitions of these things.)

Definition 5.9. A *Turing machine* M is a tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r)$ where:

- Q is the set of states;
- Σ is the input alphabet;
- Γ is the tape alphabet (so $\Sigma \subseteq \Gamma$);
- δ is the transition function;
- q_0 is the start state;
- q_a is the accept state;
- q_r is the reject state.

As with the stack alphabet, we have Γ so that there might be symbols that you can write on the tape that aren't part of the input — such as the blank symbol.

The transition function has the following form (it essentially defines how the machine behaves). We're going to take a state and a tape symbol — so we want δ to be a function on $Q \times \Gamma$ (you can think of the machine as being at a state and reading a symbol off the tape). Then we should end up in a new state, get a new symbol to write, and also get an instruction for the head to move (left or right) — so δ is a function

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

Remark 5.10. In our example, how do you make sure you only cross out one a, b, and c every pass? You do this with your finite control. Suppose we come back to the left end of the tape (we'll see later how to detect the end of the tape). Then you'll start scanning to the right, and you'll see crossed-off a's or plain a's. You'll be in a state that keeps moving the head to the right each time it sees a crossed-off a, until it sees an a that's not crossed off. Then we transition to a state r, write a crossed-off a, and move right — so if our starting state was q, we set $\delta(q, a) = (r, \overline{a}, \mathsf{R})$. Now that state remembers that we've already seen an a, and we keep just moving right — so we skip over the other a's without changing them — we set $\delta(r, a) = (r, a, \mathsf{R})$ — until we get to a b. Then when we're at this state and get a b, something else happens.

Remark 5.11. You might want to have several reject states, but it doesn't matter — you can send all of those states to a single reject state (similar to accept states for NFAs).

When you have a Turing machine, three things can happen — it can run for a while and end up in the accept state, run for a while and end up in the reject state, or run for a while and never stop. We call the latter case *looping*. Because we defined the language of the machine to be the strings where it ends up in an accept state, we consider the looping strings 'rejected by default.' So a Turing machine can reject in two ways — either by halting and rejecting, or by looping and never halting. That distinction is going to turn out to be an important one, that we will explore.

§6 September 26, 2023

PS2 has been posted, and PS1 is almost done being graded; the grades should be out within a week of the original due date. PS1 is on Gradescope, and there's a possibility of asking for regrade requests if we disagree with the grade we got. Regrade requests will not be open forever — we have two weeks from the original due date (so we have a week from when the grades are posted) to check our grading and make sure we're comfortable with it, and we shouldn't wait until the end of the term to submit regrade requests. Recitations are optional, but if we're having difficulty with the course we're expected to go to recitation (we have to learn the material one way or another). The TAs who have been leading the recitations are looking for feedback to see how we feel about them (whether they're pitched at the right level); they're putting together an anonymous feedback form (and there'll be an incentive for getting that feedback back to the TAs, with details still being worked out).

Last time we finished up our segment on CFLs by giving a method for proving languages are not context-free (the pumping lemma for CFLs), and we introduced Turing machines. Today we'll look at different variations of the Turing machine model (e.g. nondeterminism, having multiple tapes) and proving those variations are all equivalent to the original model. This will lead to the Church–Turing thesis — the reason we're looking at these variations is to show that the Turing machine model is very natural and robust (if we tweak it in various ways within reason, we still get the same class of languages that can be recognized).

Recall that the Turing machine model — which will be our model of a computer with unrestricted and unlimited memory — is that it has a finite control (consisting of states and transitions), and input presented on a tape that is infinite to the right; the portion of the tape not containing input is marked with blank symbols $\sqcup$.

The head starts out in the leftmost cell; the machine can read on the tape and write on the tape (it can change symbols to other symbols). It can compute for a while, and it may accept or reject by entering the states q_{accept} or q_{reject}. There's another possibility — that the machine goes on and on forever without reaching one of these two halting states. Then we say that the machine is *looping* (it's not really doing the same thing over and over again, but this follows the terminology of a program being in an infinite loop). In this case, we still say the machine rejects — so the machine can reject in two ways, either by halting or by looping.

A machine that rejects by halting is 'better' in some sense than one that loops (because then we don't ever really know if it's going to eventually reach an accept state or just loop forever), which motivates the following definition:

> **Definition 6.1.** If $A = L(M)$ for some Turing machine M, we say A is *Turing-recognizable*.

> **Definition 6.2.** If a Turing machine M halts on all inputs, then we say M is a *decider*.

In other words, if M never loops — it always ends up at q_{accept} or q_{reject} — then we call it a decider, since it always makes a decision. (Obviously M halts on accepting, but in general it's allowed to loop on rejecting; if that never happens, we say M is a decider.)

> **Definition 6.3.** If $A = L(M)$ for a decider M, then we say A is *decidable*.

The distinction between Turing-recognizable and decidable will be an important distinguish for us. But before we get there, we'll talk a bit about the model.

§6.1 Variants of Turing machines

At the end of last lecture, several of us had the same question. Recall that our Turing machine has its head, and the transition function says it can look at the contents of the cell, change it, and then move our head

left or right. Is the head just allowed to stay where it is?

According to the definition we gave, the answer is no — it has to move left or right. But you can imagine giving the Turing machine that extra capability. This is in some sense an enhanced Turing machine. But does this give the machine any additional power — does a Turing machine with this extra stay-put feature have the ability to recognize any languages?

The answer is no — if you have a machine that uses the stay-put operation, we can convert it to one that doesn't by simulating that operation with an ordinary Turing machine, where every time the original Turing machine stays put, the original moves right and then left (making sure that it doesn't change the tape when it moves back left).

That'll be the spirit of all the arguments we present today — we're going to give the Turing machine some 'extra' power and then show that this power doesn't matter, in the sense that it doesn't give any additional languages. The proof idea is to simulate that extra power by a machine that doesn't have that extra power.

The reason why this is interesting will be saved to the end, but it's important — the point is that the Turing machine model is very robust, and it doesn't matter if we change it in several ways.

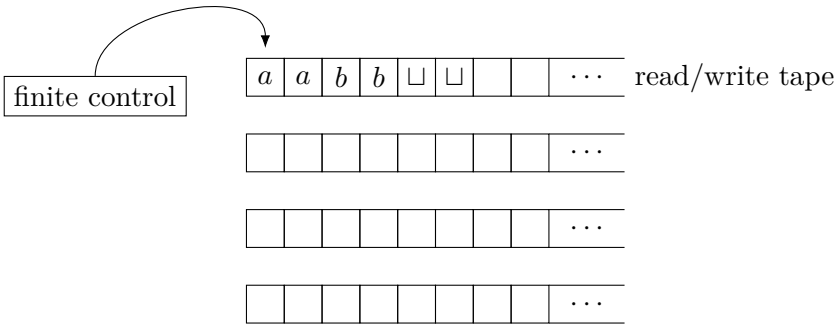
§6.2 Multi-tape Turing machines

Question 6.4. What if instead of having a single tape, the Turing machine has multiple tapes?

Conceivably, this might give the Turing machine the power to do new things. For example, if we instead had a PDA and we gave it two stacks instead of one, then it *can* do more things than an ordinary one-stack PDA. (Oddly enough, having 3 stacks is no more powerful than having 2, but 2 is more powerful than 1.)

But in the case of Turing machines, 2 tapes is no more powerful than 1 tape — and in fact any number of tapes is no more powerful than 1 tape.

A *multi-tape Turing machine* has a finite control and any number of tapes. The input is presented on the first tape; the remaining tapes are all initially blank (but may get written on as the Turing machine computes) — all these tapes are read-write. We also have heads for each of the tapes.



In the formal definition, everything is the same except the transition function — if we have k tapes, then the transition function δ is a function that takes as input a state and a symbol on each state, and produces a new state, a new bunch of symbols to write, and a left-right instruction for each one of the heads — i.e., $\delta: Q \times \Gamma^k \to Q \times \Gamma^k \times \{\mathrm{L}, \mathrm{R}\}^k$. So it can look at all the symbols under the current heads (and the current state), and it can write a new symbol at each of those places and can independently move each head left or right.

Remark 6.5. The number of tapes k is a *fixed* number — the number of tapes doesn't depend on the input. You could also define a model with a variable number of tapes, and look at that machine's properties; it'll turn out to also be equivalent, but that is more difficult. To keep things simple, we'll look only at machines which have a given number of tapes (it can't construct new tapes as it's computing).

Theorem 6.6

A is Turing-recognizable if and only if $A = L(M)$ for some multi-tape Turing machine M.

(We defined Turing-recognizable in terms of single-tape Turing machines; but now we're establishing that even if we did it by multi-tape Turing machines, we'd get the same set of languages.)

Proof. There are two directions to prove. The forwards direction is easy — if A is Turing-recognizable, then A is recognized by a single-tape Turing machine, and a single-tape Turing machine *is* a Turing machine with $k = 1$.

To prove the other direction, we need to convert any multi-tape Turing machine M to a single-tape Turing machine S.

Suppose we have a 3-tape Turing machine M, and we pick a snapshot of M after some number of steps (now we'll have stuff on all the different tapes), for example $aba\sqcup\sqcup\sqcup\cdots$, $xy\sqcup\sqcup\cdots$, $bxbx\sqcup\sqcup\cdots$ (with heads on a, y, and the third x). We'll show how the simulation by the single-tape Turing machine stores all the information M has on multiple tapes, using just a single tape.

This will be a very simple data structure — S simply takes this information and lays it out linearly on its one tape. So on S we'll have

$$aba\#xy\#bxbx\sqcup\sqcup\sqcup\cdots,$$

where # is a marker symbol separating the blocks associated with the different tapes. This is how S's tape will represent the contents of M's three tapes. (There are millions of ways we could have done this, but this is one simple way.)

Now, M also has its heads at different locations on those various tapes, and S had better keep track of where those heads are in order to simulate M. One simple way to do that is that we're going to take the tape alphabet for S (which has already been expanded to include #) to be expanded further to include a way of indicating the head locations. You could use another special symbol, but a typical way is to expand the tape alphabet by introducing new versions of each of the original tape alphabet symbols — symbols that have a dot added on them, where the dot signifies that this is the location of the head on the particulat rape. So for example, we'd have

$$\mathring{a}ba\#x\mathring{y}\#bxb\mathring{x}\sqcup\sqcup\cdots.$$

Now once S has the tape information of M written down in this way, to do one step of the simulation (where M reads from each of its heads, changes those symbols, and moves the heads left or right), what is S going to do? It can scan across the entire tape, not changing anything, to read what the symbols are corresponding to the dotted positions. Then built into our finite control in S we can have M's transition function, so that we know what M would do given this string of symbols and M's state (which we store). Then we go back and update the tape, based on what M would have done — we go back, find the dotted positions, and if we now know that M would have changed the last x to a b and moved its head left, then S changes the dotted x to an ordinary b, moves to the left, and puts a dot over that b to the left.

Here we're essentially having S do the obvious thing — what we'd do if asked to code up M using a single tape. We just represent all the information on a single tape, scan across to gather the information M would need to figure out its next step, and then update the step accordingly.

There are a few problems. What happens if M has its head on the last non-blank symbol on the second tape, and decides to move its head right? The storage of S only recorded the non-blank portion, so what do we do? We can scan through the whole tape and shift everything to the right, to open up more space — if we determine that M would have wanted to move its head to the right from that y, we see there's a # there; this is erally a representation for the infinitely many $\sqcup$'s on M's tape. So we take the portion of the tape right from there, shift it all one space at a time (we can go right to left, copying stuff over one symbol to the right and putting a blank symbol where we were), and then continue the simulation as we would have done.

Remark 6.7. The non-blank portion on each tape is always finite because each tape starts out with only a finite number of non-blank symbols (we just start out with the input on the first tape), and in every move, we can only convert at most k blanks to non-blanks — so only a finite number of blanks can be written on. So after a finite number of moves, there's only a finite number of non-blank positions. (The simulation has only gone on for a finite number of steps, so at any point during the simulation, there's only a finite amount of non-blank stuff to worry about.)

All strings are finite (unless otherwise stated, and that'll be very rare) — the inputs are finite, and we never let a Turing machine run for an infinite amount of time and then look at it.

There's one other detail that we haven't addressed — we really have to do an initialization at the beginning. Here S's tape starts out with just the input, and we need to put in #'s just representing that our tapes are blank. We need a place to put the heads as well, so we initialize as

$$\text{input}\#\mathring{\sqcup}\#\mathring{\sqcup}\#\cdots. \qquad \square$$

Remark 6.8. What if we instead interleaved our tapes, putting one letter from the first tape, then one from the second, and one from the third, and so on? So we'd have $axbbyxa \sqcup b \sqcup \sqcup x \cdots$. This would work, and we'd never have the problem where we run out of space and need to shift.

Remark 6.9. As S scans through the tape, it has to remember the values it sees on the head locations — we scan across, look up the dotted symbols, and remember those in S's finite memory. We're not going to go through and build the state diagram for that (it would be a mess to do it), but it can be done — your intuition about the finite control should be that you can store any fixed finite amount of information in the state, and here we just need to store k dotted symbols under these 'virtual heads' that S has to represent M's actual heads.

We're going to be operating at higher and higher levels of abstraction as we go on — we'll be getting away from the very formal constructions we did earlier on.

So now we've seen how to convert a multi-tape Turing machine into a single-tape Turing machine that does the same language; this proves the theorem (A is recognizable if and only if we can do it with a multi-tape Turing machine).

§6.3 Nondeterministic Turing machines

We'll now move to our second Turing machine variant. (Here we're back to single-tape Turing machines.) We're not going to write out the definition again in full detail, but here we have *nondeterministic* Turing machines. This means our transition function is actually a function

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

As with NFAs, this means that if a machine is in a particular state and it reads a particular symbol under its head, previously we went to a new state, wrote a new symbol, and moved the head in a specific direction; but now we have several possibilities. For example, we could have $\delta(q, a) = \{(r_1, b, \mathrm{L}), (r_2, c, \mathrm{R})\}$. Then the Turing machine operates nondeterministically the same way our NFAs and PDAs have operated in the past — if there are multiple ways to go, then the machine forks into multiple threads (each of which is a complete copy of the original machine, with its own control and its own tape), where in one copy we make one choice, in the next copy we make the next choice, and so on.

So in terms of parallelism, when we have a choice, we suddenly get two Turing machines — one does the first choice, and one does the second. And we can continue to have more forking or branching or threads, each of which is doing its own thing.

We continue until one accepts — if some thread leads to 'accept,' then we shut down the whole thing and say it accepts. So acceptance overrules everything — it overrules both acceptance and looping. In particular, if some thread hits q_{reject}, then that particular thread goes away; but this doesn't tell us anything about whether the machine will accept the input (since a rejecting thread is overruled by an accepting thread) — this just tells us that thread is gone. The *only* way this machine rejects is if every thread rejects (i.e., ends up at q_{reject} or goes on forever).

Nondeterministic Turing machines will be equivalent, but they're important and we'll come back to them in complexity theory (where they'll play a key role).

Remark 6.10. For NFAs, there could be multiple threads and some threads could die because there was nowhere to go. For Turing machines the same can happen — if $\delta(q, a) = \{\}$ (i.e., the machine is in a particular state and reads a particular symbol, and there's nothing to do), then that thread just dies off, as with NFAs.

(This cannot happen with normal Turing machines, because with deterministic Turing machines there's always a way to go — the transition function always gives you exactly one next move. But because nondeterminism allows for the possibility of 0 next moves, that thread then dies.)

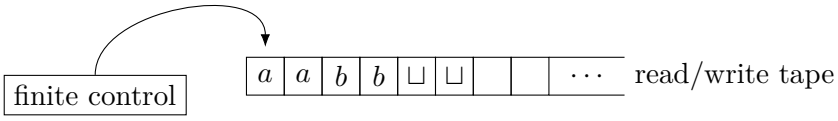
Theorem 6.11

A language is Turing-recognizable if and only if some non-deterministic Turing machine recognizes it — i.e., A is Turing-recognizable if and only if $A = L(N)$ for some nondeterministic Turing machine N.

Proof. The forwards direction is again immediate — a determinstic Turing machine (which is how we defined Turing-recognizable) is a special kind of nondeterminstic Turing machine. So we only have to do work for the backwards direction.

For the backwards direction, we want to convert a nondeterministic Turing machine N to an ordinary (deterministic) Turing machine M.

Suppose we have a nondeterministic Turing machine N, with its tape.



One way of thinking about nondeterminism is as a tree of threads — the machine starts off in one state, but then if we have a nondeterministic move we might have two possibilities, and so on.

We want to convert this to a Turing machine M.

First, why don't we add more tapes every time we have a forking? This isn't exactly the multi-tape model, since we can't add more tapes in the middle of computing. (But it is possible to use this idea.)

Instead, we'll again have M dividng its tape into blocks separated by #'s, where now each block, instead of representing the contents of a tape, is going to represent the contents of N's tape on one of its threads.

It's almost exactly the same, but there's one additional detail. Here we might have

$$a\mathring{a}bb\#x\mathring{y}xy\#ax\#\cdots\#\sqcup\sqcup\sqcup$$

with some number of threads active. But one thing that happens now: in addition to remembering where each head is, we also need to remember where all the states are (because now each thread has its own state, and M has to keep track of which state N is in on *each* of those threads). So we're just going to write this on the tape — for example, we might write

$$q_7a\mathring{a}bb\#q_3x\mathring{y}xy\#q_2ax\#\cdots\#\sqcup\sqcup\sqcup\sqcup.$$

So we add a tape symbol representing a state at the beginning of each block.

Remark 6.12. This doesn't create infinitely many symbols — M has a tape symbol for each one of N's states (not M's own states). So if N had 12 states, M is going to have 12 additional tape symbols, where we write down one of those states on the tape. We don't need M to write down its *own* states on the tape (then we'd have some kind of circular reasoning) — M is only worried about recording N's states, not its own.

Remark 6.13. Each one of N's threads is a full Turing machine, with a state, a tape, and a head (like a NFA — each thread has a separate state). So we have to record on each block the full information of that thread — M has to keep track of which state each thread of N is in, and update accordingly.

Now once we have all this information, M can operate on this block-by-block — it reads the first block and it can update this block. (It doesn't have to read the whole thing at once and collect together information, the way we did for multi-tape machines.) We read and see that in the first thread N is in q_7 and is reading an a; now suppose that it forks into two possibilities. Then we're going to have to copy this whole thing either next door or at the end, with those different possibilities realized.

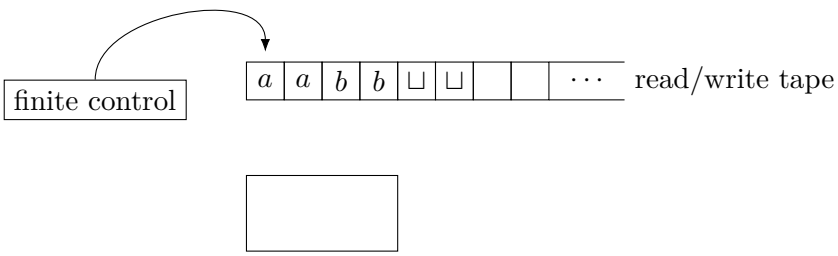
(We're not going to really need the details of the proof for now, so we'll move onto another example.) □

§6.4 Turing enumerators

Now we've seen two of the three equivalent variants of Turing machines. We're next going to do one that looks *really* different — a *Turing enumerator*. This is kind of a different 'breed' of Turing machines. When we talked about finite automata and context-free languages, we had two types of models — a 'recognizer' model (DFAs and NFAs and PDAs) and a 'generator' model (regular expressions and context-free languages). We're now going to see a generator model for Turing-recognizable languages (here this will also be an automaton) — this is going to be a way of *producing* the strings in the language, instead of *recognizing* which strings are in the language.

Definition 6.14. A *Turing enumerator* is a Turing machine with a printer.

We're being a bit informal here; we could make this precise but we won't.



Now we still have our control and a tape we can write on, but we also have an output device. We don't give our Turing enumerator any input to start with — we start out with the tape completely blank, and turn it on. The machine is then going to start writing on the tape, going from state to state. And it has a special command, which says PRINT. And there's going to be a designated part of the tape (maybe everything up to where the head is) — we have a way of designating a string that appears on the tape as the string to be printed. Then when the machine goes into its special print state, that string appears on the printer. And then the machine keeps going — it keeps on computing, and maybe it sends out another string to the tape, and that gets printed.

So we eventually end up with a list of strings that come out of the printer; this could be a finite list or an infinite list (obviously for an infinite list the machine never halts and just keeps going; it might also produce a finite list and never halt, and might just keep computing while nothing else comes out, or it might have some way of halting).

That collection of output strings is the *language* of the enumerator.

Definition 6.15. The *language* of a Turing enumerator E is the set

$$L(E) = \{w \mid E \text{ prints } w \text{ (at some point) when started on blank input}\}.$$

We could, for example, make a Turing enumerator whose language is $a^k b^k c^k$. The way it does this is that on its tape, it just has to remember k, which starts out as 0. So it initially prints out the empty string. Then it increments k to be 0, and on the tape it writes abc; then it prints that. Now we increment k to 2 and write $aabbcc$; and so on. So it keeps writing out $a^k b^k c^k$, printing that string out, and going to the next value of k. So we'll get ε, abc, $aabbcc$, ..., and that'll be the language of the enumerator (if we program it to do that).

Remark 6.16. Enumerators don't have a notion of accept or reject — it chooses to print whatever it wants (whatever it's programmed to print). The collection of printed things *is* the language of that program.

Remark 6.17. There's no input to a Turing enumerator — the output is all that happens. It's a generator, not a recognizer — it's not going to be accepting or rejecting.

Theorem 6.18

The languages we can get with Turing enumerators are exactly the recognizable languages — A is Turing-recognizable if and only if $A = L(E)$ for a Turing enumerator E.

Proof. The backwards direction is the easy direction — we're given E, and we want to convert it to a Turing machine M. On an input w, M simply does the following: we have an eumerator and string, and we have to decide whether to accept that string. Since E is a Turing machine as well, M can run E (it can simulate E on its tape) — it puts w off to the side and runs E, and every time E prints out something, we check if it's w. So we just run the enumerator until it outputs w; if it does, then we accept. If we never see w and the enumerator just keeps going, then we may be looping, but we'll never accept; so we reject w by looping.

We're out of time, but here's the idea for the other direction (which is slightly trickier) — we're given an ordinary Turing machine M, and we want to make an enumerator E. What do we *want* to do? The enumerator doesn't have any input, so it takes M and starts feeding in different strings into M — we'd like to feed in all possible strings into M one by one, and run M on all those different strings. (So we'd run M on ε, 0, 1, 00, 01, ... one by one.) If M accepts a string, then the enumerator should print it out; and if M doesn't accept the string, you don't print it out.

But we have a problem. The way we just described it, we first run M on ε; if M accepts then we print and if M rejects we don't; and then we move on. But this is bad — because if M is looping on ε, we'll never know, and we might never have a chance to run M on 0, which it might accept.

So what we really need to do is run M on all of those strings in *parallel*. More mathematically, we run M on the first i strings for i steps, and we do this for increasing i — so we run M on more and more strings for longer and longer, and whenever M accepts a string we print it out. □

§7 September 28, 2023

We've started talking about Turing machines, which will be the model we'll focus on the rest of the semester. We introduced many variations and proved that they're all equivalent; the purpose of this was to give us a sense that this model is very robust (it's not sensitive to small, or even fairly large, changes in the definition, and in fact there are many other definitions that look really different but are also equivalent). This has a consequence called the Church–Turing thesis, which we'll talk about soon.

We covered multitape Turing machines, nondeterministic Turing machines, and enumerators (which will be covered more in recitation).

§7.1 Church–Turing Thesis

The Church–Turing thesis says that the intuitive notion we have about algorithms (which predates computer science — we can think of it as following a recipe or procedure) is equivalent to what you get from Turing machines. So it links an informal intuitive definition with the precise, formal definition of Turing machines.

You can take this as a definition of algorithms, but it's kind of a bit more — the fact that Turing machines are equivalent to very different models suggests there's something natural about this class of algorithms, motivating the connection.

This has been really important in the history of math in terms of solving problems. In 1900, at the International Congress of Mathematicians, David Hilbert gave a lecture on unsolved problems that would challenge mathematicians for the coming century. He presented a whole bunch of problems in geometry, number theory, and many other areas of math (he was probably the last human being to know pretty much all the mathematics of his day). One of his problems had to do with algorithms and algebra:

> **Question 7.1** (Hilbert's tenth problem)**.** Given a polynomial with integer coefficients over several variables (e.g. $x^2 - 4$), find a procedure which tells us whether the polynomial has a solution in the *integers*.

No one knew of a procedure to test whether a polynomial had an integer solution or not; so his problem was to devise a procedure to do that. He didn't even offer the possibility that there wasn't one — the problem was formulated to give a procedure.

We now know that there *is* no procedure — that's an undecideable problem. But there was no hope of giving that answer when Hilbert posed the question, because we only had an informal idea of what an algorithm is — to prove there is none, you need a precise definition of what an algorithm is. That only came up in the 1930s with the Church–Turing thesis; and then it took another 30 years for us to solve this problem.

Today we'll give a bunch of examples of problems we'll show are decidable.

Last time we had two classes of languages — decidable languages and Turing-recognizable languages. The *decidable* languages are ones recognizable by a Turing machine that always halts; Turing-recognizable languages are ones recognized by a Turing machine, which may or may not halt. (We'll prove that this is a larger class next week.)

§7.2 Encoding

We'll often want to feed into Turing machines complicated objects, like graphs, polynomials, other automata, and so on. We'll do this just like you'd naturally do — to put a complicated object as an input, we'll code it into some string, and just feed the string in. We'll often talk about the encoding of an object Z by writing $\langle Z \rangle$; this means an encoding of Z that's reasonable, in a way that a machine can decode to figure out what the object was. If we have several objects we want to represent as a single string, we write $\langle Z_1, \ldots, Z_k \rangle$.

§7.3 Some decision problems on automata

We'll see an example, which has to do with whether finite automata accept their input string:

Example 7.2 (Acceptance problem for DFAs)
Let $A_{\mathsf{DFA}} = \{\langle B, w \rangle \mid B \text{ is a DFA and accepts string } w\}$.

Here I give you a DFA and a string, which are together represented as a string (that represents both of these things together); we want to test whether that DFA accepts that string.

Theorem 7.3
A_{DFA} is a decidable language.

The way we'll prove these kinds of things is by giving a decider — a Turing machine that accepts when we're in the language, and halts and rejects when we're not.

Going forwards, we're going to write our Turing machines just as informal descriptions of pseudocode (in prose). There really is an actual set of states and transitions — there's a real honest-to-goodness Turing machine there — that if we had the patience, we could build. We never will, but we could.

Proof. The pseudocode for our Turing machine M (on an input s) is as follows:

(1) We first check that s is of the form $\langle B, w \rangle$, and reject otherwise (as it definitely can't be in the language).

(2) Now that we know our input is of the right form, we're going to simulate B on w.

What do we mean by this? First, what does it mean to be of the right form? We should have $s = \langle B, w \rangle$, which means that s corresponds to a DFA presented in the usual way — with a set of states $Q = \{q_0, \ldots, q_k\}$, an input alphabet $\Sigma = \{a, b, c\}$, a transition function as a table, and so on. This whole information is written on the tape (encoded into 0's and 1's, but if we wrote it that way it'd be unintelligible). We also have a string w, e.g. 101, sitting on the tape.

And now we're going to simulate B, which is this DFA, on w, which is our input string. For convenience, we may as well use a second tape. We're then just going to do the simulation in the obvious way we'd do it if asked to write a Pascal program for this — we use the extra tape to keep track of the current state in the machine. We start off in q_0; then we start reading w (crossing off to keep track of what we've read so far). Every time we read a symbol of w, we look at the current state, look up that state in the transition function (along with the corresponding symbol), figure out what our next state should be; then we look at the next symbol in w, and so on. We continue until we've read up to the end of w.

(3) If we end in an accept state, then we accept; if we end in a non-accept state, then we reject.

Under all conditions, this machine always halts — it never has the option of running forever — so it's a decider. And it decides exactly A_{DFA}. □

Remark 7.4. On the tape is just 0's and 1's (to start with); that's some encoding of a DFA as a string. You can imagine a system of encoding a larger alphabet into 0's and 1's, plus a dividing symbol (e.g. coding # into 0's and 1's as well). So there's some way to represent several objects together as a single string of 0's and 1's.

Example 7.5

Let $A_{\mathsf{NFA}} = \{\langle B, w\rangle \mid B \text{ is a NFA and accepts } w\}$.

Theorem 7.6

A_{NFA} is also decidable.

Proof. We'll build a new Turing machine N. We could do this in the same way as above, writing down a set of states instead of a single state (the way you'd normally simulate a NFA), but we'll do it a bit differently to illustrate something.

We'll use the shorthand of 'N on input $\langle B, w\rangle$' to meant hat N expects the input to be of that form and checks that it is; if it isn't, then it just automatically rejects.

The first thing we'll do is use the fact that we can convert NFAs into equivalent DFAs; we just run that conversion process (the subset construction process).

(1) Convert the NFA B into a DFA D.

(2) Now that we have a DFA, we can use M to solve the problem — because we already know how to solve this problem for DFAs. So now we run M (as a subroutine) on input $\langle D, w\rangle$.

What would have happened if we had fed $\langle B, w\rangle$ into M? It would have rejected, because B is not a valid encoding of a DFA — M is only designed to handle DFAs. But we can convert the NFA to a DFA, and then use M. (We'll be doing lots of this kind of thing.)

(3) Accept if M accepts, and reject if M rejects. □

Remark 7.7. The brackets are because it doesn't make sense to make a language out of an automaton and a string. Here it's easy to be precise about what we're doing by using bracket notation — the brackets make the things we're proving closer to being rigorous statements. But we're not going to define how to do the encoding — you can come up with your own system of reasonable encodings (you can sort of just lay out the description of the automaton, put into 0s and 1s). But it's also possible to come up with pathological systems that are mathematically encodings but the Turing machine can't make sense of them; don't do that.

Example 7.8 (The emptiness problem)

Consider $E_{\mathsf{DFA}} = \{\langle B\rangle \mid B \text{ is a DFA and } L(B) = \varnothing\}$.

Here we're not taking a particular input; instead we have an automaton B, and we want to find if it accepts *any* input or not.

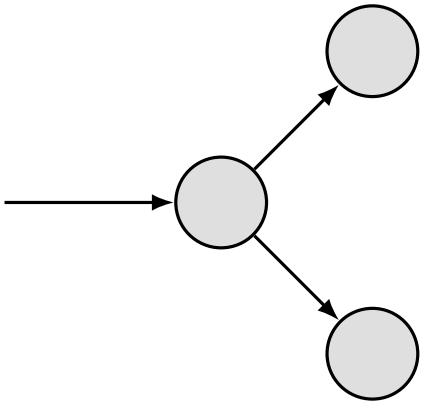
Theorem 7.9

E_{DFA} is decidable.

Proof. Our Turing machine S, on input $\langle B \rangle$, does the following.

Imagine (as a picture) that we have B; it has a start state and a bunch of other states, with some accepting states. This doesn't necessarily mean its language is nonempty — it's possible none of the accept states are reachable, in which case B doesn't accept anything and it has an empty language (meaning we're supposed to accept).

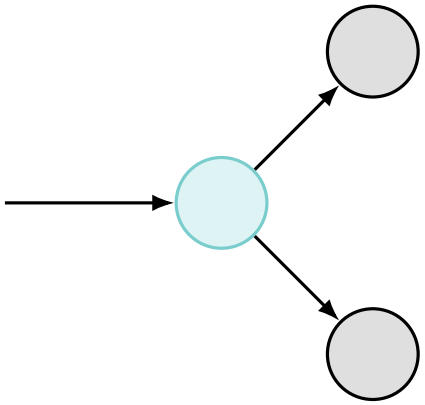
This is really just a graph traversal type problem — we want to know whether there's a path from the start symbol to one of the accepting states. You can use DFS or BFS to do this; since we're not assuming we've taken an algorithms course, we'll give a very simple (but inefficient) algorithm.



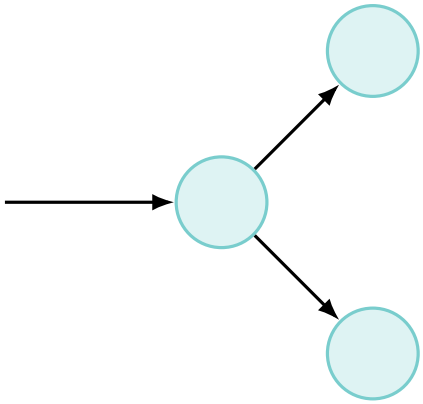
We use a marking procedure — we mark all of the states that we have shown to be reachable from the start state. And we keep on adding to that list until we can't add any more. Then we look — once we've maxed out our list of reachable states, is there an accepting state among them? If there is, then the machine's language is not empty, and we reject. If there were no reachable accepting states (that were marked), then the language is empty, and we should accept.

Here's how we do this.

(1) Mark B's start state. (Obviously the start state is reachable from the start.)



(2) Repeat until nothing new is marked: Mark every state with an incoming arrow from a previously marked state.



(3) Reject if some accept state is marked, and accept otherwise.

So we start by marking the start state, and then we go through and mark everything reachable from things we've already marked. Eventually we have to run out of things to mark (since there's only finitely many states), and then we check if we've marked any accepting state. □

Of course E_{NFA} is decidable for the same reason — once we have a conversion procedure that converts between two types of automata to show they're equivalent, we can always substitute one for the other. We could even put E_{regex} there — does my regular expression generate anything or not? — because we could convert the regular expression into a DFA, and then run the DFA testing.

Example 7.10

Consider the equivalence problem

$$\mathrm{EQ}_{\mathsf{DFA}} = \{\langle A, B\rangle \mid A,\ B \text{ are DFAs and } L(A) = L(B)\}.$$

This is somewhat less obvious how to do, but it's still decidable:

Theorem 7.11

$\mathrm{EQ}_{\mathsf{DFA}}$ is decidable.

Proof. Let our Turing machine be T. Then on input $\langle A, B\rangle$, what should we do?

We're going to take advantage of a certain new operation, *symmetric difference*:

Definition 7.12. Given two sets X and Y, their *symmetric difference* $X \,\Delta\, Y$ is all the elements in exactly one of the two sets.

Think of our two sets as $L(A)$ and $L(B)$, and consider their symmetric difference $L(A) \,\Delta\, L(B)$. We'll build a new DFA whose language is precisely this symmetric difference.

Why is this interesting? What happens if A and B were equivalent, meaning $L(A) = L(B)$? In that case, this symmetric difference is empty. And conveniently, we know how to test emptiness.

(1) Construct a DFA C recognizing $L(A) \,\Delta\, L(B)$.

How do we know we can do this? It turns out that

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

(The first term tracks terms in $L(A)$ but not $L(B)$, and the second tracks terms in $L(B)$ but not $L(A)$.) So there's a nice way to write the symmetric difference in terms of intersection, complement, and union. And we saw closure under these operations — in particular, we saw how to construct DFAs whose languages correspond to the intersections, unions, and complements of given DFAs, and we can use that here.

(2) Use S to test if $L(C) = \varnothing$.

(3) Accept if yes (i.e., if S accepts) and reject if no. □

§7.4 Algorithms for grammars

Now we'll shift gears and talk about algorithms for grammars.

Theorem 7.13

Let $A_{\mathsf{CFG}} = \{\langle G, w\rangle \mid G \text{ is a CFG and generates } w\}$. Then A_{CFG} is decidable.

Proof. Call our Turing machine F, on input $\langle G, w\rangle$.

Now things start getting more tricky — G looks something like

$$\begin{aligned} R &\to \cdots \mid \cdots \\ T &\to \cdots \mid \cdots \\ U &\to \cdots \mid \cdots \mid \cdots \end{aligned}$$

and we want to apply this grammar and see if we get w. We might try converting to a PDA, but either way we'll run into a difficulty.

That difficulty is that if we do the obvious thing — which is that we start with the start variable and start doing substitutions, there are many different substitutions you can do. And if you keep on doing substitutions (maybe trying a whole tree of possibilities), if you get on the wrong branch, it might go forever and never hit w.

So you have to be careful how you're going to organize this to make sure that your procedure actually terminates.

There are a variety of ways of doing this; we'll describe one. If you try converting to a PDA, you'll have the same problem — the PDA is nondeterministic, and some of the threads might not end (while others might end), and if we just simulate them without being careful, we might follow a thread that is not terminating and keep wasting our time on it even if there is some other thread that does accept.

So we have to be careful how we organize this; one way to do this is to take advantage of *Chomsky normal form*. (We'll use this later as well.) (This was one of the 0.X problems on PS2.)

Definition 7.14. A grammar is in *Chomsky normal form* if all rules are either of the form $A \to BC$ or $A \to a$.

Of course there can be many possibilities, but they all have to look like either two variables or one terminal. It's pretty straightforward to convert an ordinary CFG into Chomsky normal form; the procedure is given in the textbook, and we're not responsible for knowing it (it's somewhat tedious, but you just do the obvious thing).

(This is named after the linguist Chomsky, who was at MIT.)

The nice thing about Chomsky normal form is the following lemma (which is the 0.X problem):

Lemma 7.15

If G is in Chomsky normal form, then the number of steps it takes to generate a string w is exactly $2|w|-1$.

It's fairly simple why this is true — if $|w| = k$, then we need $2k-1$ steps to generate it. This is because we're starting out with a single variable. Every time we do a step $A \to BC$ we get an extra variable; so we need $k-1$ steps to stretch out into k variables. Then we need another k steps to turn those variables into terminals.

Now our Turing machine F works as follows:

(1) Convert G to Chomsky normal form; let this new grammar be H.

(2) Try all derivations (i.e., collections of substitutions) of length $2|w|-1$.

(3) Accept if (at least) one of them generates w, and reject if not. □

There is an important corollary (which will be a little different) to this theorem. What it tells us is that every context-free language is also decidable.

Theorem 7.16

Every context-free language is decidable.

Proof. Let A be a CFL generated by a CFG G (any CFL has some grammar).

We then construct the following Turing machine for A. (This gets a little slippery.)

Our Turing machine M_G will have G built into it — A has a grammar G by definition, and we use our grammar G to construct M.

Then on input w:

(1) Test if $w \in L(G)$ using our construction F on input $\langle G, w\rangle$.

(2) Accept if yes, and reject if no.

To clarify, we have a CFL; it has some grammar. We want to show that we can test whether a string is in the language, because that's what our decider needs to do — it needs to test whether a given string is in the CFL.

It's going to use the grammar for that language, and our tester — which given a grammar and a string, tests if the grammar generates the string. That'll tell us whether $w \in A$, using a decider. □

Remark 7.17. This proves that CFL $\subseteq$ Decidable. Here M_G *is* the decider — it takes a string w as input (this has to work for every w, as that's our input), and it has to figure out, is that in A or not? And it's going to use the grammar for A and the procedure above to test whether our grammar generates the string; and that'll tell that the string is in the language.

Remark 7.18. Why can't you use the fact that a CFL is accepted by some PDA, and use that to say it's decidable?

Here our proposal is that a CFL is recognized by some PDA; does that immediately give us a Turing machine? Not really, because the PDA has branches which don't halt. So the PDA is in essense not a decider (though we don't use this terminology) — it might go on forever on some branches. So we have to prove something stronger — we have to prove that we have a PDA that never has any infinite branches. But we're not going to think about it this way.

(There are also proofs using PDAs that do something to detect and terminate infinite branches.)

Remark 7.19. Inside CFLs we have the regular languages. Since we're claiming the CFLs are all decidable, we immediately get that the regular languages are also decidable.

Remark 7.20. Where did we use the fact that A is a CFL? We used the fact that because it's a CFL, we have a grammar; and we use that grammar (we feed it into F, together with the string).

Remark 7.21. What's the difference between the last two theorems? The first proves that the one language A_{CFL} is decidable, the second proves that every CFL is decidable.

Remark 7.22. Why are we working with just a single w? We aren't — here w is an input, and this has to work for every w (e.g. w could be 0011, or 010, or so on). Then M_G has to process w, whatever it is; and it does this way, and it'll get the right answer.

Remark 7.23. When you have a decidable language, its complement is also decidable — you just swap the accept and reject states in the Turing machine. So the class of *decidable* languages is closed under complement. But the class of CFLs is *not* closed under complement. There's no contradiction here — it just tells us that the class of CFLs is decidable and the class of *complements* of CFLs (which we call coCFLs) is also decidable. But that's another class of languages — we have two (potentially overlapping) circles lying inside the larger circle of decidable languages.

Remark 7.24. Prof. Sipser got a lot of questions, but not the one he expected — where did we get G from? We know we have a CFL G, but who gives me G? Because I need G to build M_G.

The answer to this question (which we didn't ask) is — we know that G *exists*, and therefore we know that M_G exists. And that's enough to tell us that A is decidable. If you don't tell me G, I can't tell you M_G; but I know it exists. (If that didn't bother you, don't let this confuse you.)

Theorem 7.25

The emptiness problem for CFLs,

$$E_{\mathsf{CFL}} = \{\langle G\rangle \mid G \text{ is a CFL and } L(G) = \varnothing\},$$

is decidable.

Proof. The proof is in the textbook; it's a similar marking algorithm to the one we used for emptiness for

DFAs. On input $\langle G \rangle$: suppose the grammar looks something like

$$R \to SaTbc$$
$$S \to Sab \mid T \mid ab$$
$$\vdots$$

We want to know, is there some way to start at the start variable and generate some string of terminals — can I get some output from this grammar? (That means the language is nonempty.)

What we're going to do is a marking procedure, where we mark starting with the *terminals*. We're going to first mark every terminal symbol. Then we look at the grammar, and mark every variable that goes to a string of fully marked symbols. So first because S, as one of its rules here, goes to a terminal string ab, now we can mark S too (because we know S can derive a string of terminals).

Now we keep doing this — we keep looking at the rules to see if we have some string on the right-hand side of completely marked symbols, and in that case we mark the variable associated to that (because we know if we start at that variable, we can derive a string of terminals). And we keep marking until we can't mark anything new. Then we check whether the start variable is marked. If it is, the grammar's language is nonempty. Otherwise there's no way to start with that variable and generate a string of terminals, and we know the grammar's language is empty.

In other words:

(1) Mark all terminals.

(2) Repeat until there are no new marks: Mark all variables S where S goes to a string of completely marked symbols.

(3) Accept if the start variable isn't marked; reject if it is. □

We'll now see a more complicated, and particularly interesting, problem — equivalence for CFGs.

Theorem 7.26

The equivalence problem for CFGs,

$$\mathrm{EQ}_{\mathsf{CFG}} = \{\langle G, H \rangle \mid G \text{ and } H \text{ are CFGs and } L(G) = L(H)\},$$

is *not* decidable.

This should be surprising — two context-free grammars are pretty simple-looking things, and you'd imagine that testing whether two grammars are equivalent (using some procedure like this) should be possible by an algorithm. But it's not; we'll show that later.

You might ask, why can't we use the same symmetric difference construction? We don't have closure under complement for CFLs (or intersection, for that matter).

Finally, a last and important example:

Theorem 7.27

The language

$$A_{\mathsf{TM}} = \{\langle M, w \rangle \mid \text{Turing machine } M \text{ accepts } w\}$$

is not decidable.

We'll prove this later. But what we will prove today is that it *is* Turing-recognizable: we can make a recognizer which might loop on rejection. This is a particularly important and famous algorithm.

Theorem 7.28

The language

$$A_{\mathsf{TM}} = \{\langle M, w\rangle \mid \text{Turing machine } M \text{ accepts } w\}$$

is Turing-recognizable.

Proof. Our Turing machine U on $\langle M, w\rangle$ works as follows:

(1) Simulate M on w (we're given the instructions of another Turing machine and an input w, and we can simulate it).

(2) If M accepts, then we accept. If M halts and rejects, then we reject.

And that's what we can say. What happens if M doesn't halt — what happens to U? It's just going to keep simulating, because it doesn't know what M is going to eventually do. So if M rejects w by looping, then U is going to reject $\langle M, w\rangle$ by looping. □

The reason this is important is that this is the *universal Turing machine* that can simulate any other Turing machine — this was in Turing's original paper.

§8 October 3, 2023

The last few lectures, we talked about Turing machines. Last lecture, we raised the level of abstraction a bit — we're not going to look at Turing machines so much in terms of their states and transitions; instead, we're going to talk about them as *programs*, and we're going to reason about them at a high level, writing the description of how they operate in pseudocode or prose (and not paying attention to the tapes and states). Underneath it all, that's really what's happening, and if we really wanted to we could convert everything to states and tapes.

But sometimes when we're proving things about Turing machines we *are* going to make use of that formalism; and that's the advantage of having the formalism of a general-purpose computing model.

Last time, we showed that a whole bunch of languages are decidable (the emptiness problems for DFAs and CFGs, and so on). Recall that a Turing machine is a *decider* if no matter what input we feed it, it always halts (it must reject by entering the reject state — a general Turing machine is allowed to reject by looping, but a decider isn't).

Last time we introduced the language $A_{\mathsf{TM}} = \{\langle M, w\rangle \mid M \text{ accepts } w\}$. We saw it's Turing-recognizable — we have a TM that can recognize it, possibly by looping. But today we'll ese that's the best we can do — today we're going to prove that it's not decidable. (In fact, later we'll also see a language that isn't even Turing-recognizable.)

§8.1 Undecidability of A_{TM}

Theorem 8.1

The language A_{TM} is undecidable.

It's recognizable, by the universal Turing machine we saw earlier; but today we'll show it's not decidable.

Before we prove this statement, we'll introduce the method we'll use, in a different context — that's called the *diagonalization* method.

§8.2 Diagonalization

The diagonalization method is more than 100 years old, and it was introduced in a totally different part of math — one that has to do with understanding the sizes of infinite sets. The sizes of finite sets is straightforward (if one set has 7 elements and the other has 5, clearly they have different sizes). But with infinite sets, this isn't so clear.

Example 8.2

Suppose we compare $\mathbb{N} = \{1, 2, \ldots\}$ to $\mathbb{Z}$. Do these have the same size or different size?

There is a nice notion of size for infinite sets:

Definition 8.3. Two sets A and B are the *same size* if there is a one-to-one, onto function $f: A \to B$.

To define these terms:

Definition 8.4. A function f is *one-to-one* if whenever $x \neq y$ we have $f(x) \neq f(y)$.

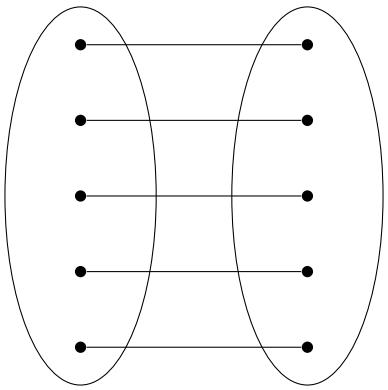
In other words, there are no collisions — f never maps two different elements to the same element. There's another, more modern, term that Prof. Sipser doesn't like; such functions are also called *injective*. (Probably many of us have seen this term before; Prof. Sipser has a hard time remembering what this means, and one-to-one seems more natural as a terminology. But we can use either.)

Definition 8.5. A function $f: A \to B$ is *onto* if its range is B, i.e., if f covers all of B.

Such functions are also known as *surjective*.

This means that if we range over the different elements of A, we end up hitting all the elements of B.

A one-to-one onto function is sometimes called a *one-to-one correspondence* or a *bijection*. But really what it is is a *pairing* up of the elements of A with the elements of B:



When f has these properties, it pairs elements of A with elements of B such that it's a reasonable pairing — we never pair one thing on the left with two on the right, and everything is in a pair.

Once we have this notion that two sets are the same size if there is a pairing between them, then it seems obvious that this is the right thing — at least, it works in the finite case. (If you have 5 oranges and 7 apples, you can't pair up the apples and oranges so that each orange gets one apple and vice versa; but it will be true when you have two finite sets of the same size. And we just extend that idea to the infinite world.)

Example 8.6

The natural numbers $\mathbb{N}$ and the integers $\mathbb{Z}$ have the same size.

Proof. To show this, we have to find a function f that pairs each natural number with each integer. We can do this by mapping $1 \mapsto 0$, $2 \mapsto -1$, $3 \mapsto +1$, $4 \mapsto -2$, $5 \mapsto +2$, $6 \mapsto -3$, and so on. (This might be a bit difficult to describe as an equation, but it's certainly a valid function, and it's one-to-one and onto.) □

So even though $\mathbb{Z}$ superficially looks bigger than $\mathbb{N}$, we can pair elements up and show they have the same size.

Example 8.7

The positive rational numbers $\mathbb{Q}^+ = \{\frac{m}{n} \mid m, n \in \mathbb{N}\}$ and the positive integers $\mathbb{N}$ have the same size.

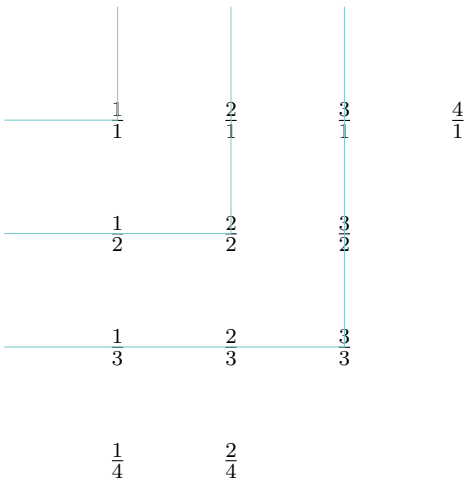
(The same is true for $\mathbb{Q}$ instead of $\mathbb{Q}^+$, but $\mathbb{Q}^+$ leads to a slightly nicer picture.)

Proof. We can put all the positive rational numbers into a table:

$\frac{1}{1}$	$\frac{2}{1}$	$\frac{3}{1}$	$\frac{4}{1}$
$\frac{1}{2}$	$\frac{2}{2}$	$\frac{3}{2}$	
$\frac{1}{3}$	$\frac{2}{3}$	$\frac{3}{3}$	
$\frac{1}{4}$	$\frac{2}{4}$		

(Some rational numbers might appear multiple times, but each is certainly included at least once.)

We'll use this table to get a pairing between the natural numbers and positive rationals: we simply walk through the table in the following way.



We first start at 1, pairing 1 with $\frac{1}{1}$. Then we start walking along the next path — we pair 2 with $\frac{2}{1}$, and 3 with $\frac{1}{2}$ (we'd naively get $\frac{2}{2}$, but we don't want 1 and 3 to map to the same thing, so we simply jump over it and take $\frac{1}{2}$). Then 4 maps to $\frac{3}{1}$, 5 to $\frac{3}{2}$, and so on ($\frac{3}{3}$ gets skipped, so 6 gets mapped to $\frac{2}{3}$). □

Remark 8.8. What if we have two functions f and g, such that f is one-to-one and g is onto? Does this imply that there exists a function that is both one-to-one and onto? Prof. Sipser doesn't know, but there should exist an answer.

This seems like a very flexible definition — so it looks like any two infinite sets might be of the same size. But that turns out to be false — there actually are infinite sets which are truly bigger than others.

In order to get there, we need the following definition:

Definition 8.9. We say A is *countable* (or *countably infinite*) if A has the same size as $\mathbb{N}$.

We just proved that $\mathbb{Z}$ and $\mathbb{Q}^+$ are both countable; and of course $\mathbb{N}$ is also countable (it has the same size as itself). This is a reasonable term to use because a function $\mathbb{N} \to A$ gives a way of assigning a label to each element of A that sort of counts it.

We'll give an example of an infinite set which is *not* countable — and that's the real numbers.

The *real numbers* $\mathbb{R}$ are those that we can express as decimals; for example, $6.27135\ldots$. Anything that we could express as a decimal number — or binary number — with some digits before the decimal point, and possibly infinitely many digits following the decimal point, we'll call a real number. We won't give a more precise definition than that; if you want to see one, you should take an analysis class.

It's clear that $\mathbb{R}$ is an infinite set.

Theorem 8.10

The set $\mathbb{R}$ is not countable.

Cantor is the first one who set this up — he showed that $\mathbb{R}$ is really a 'bigger' set than $\mathbb{N}$. The proof uses diagonalization, here in its most basic and pure form; this will be helpful to us when we use it in our particular situation about computation.

Proof. Assume for contradiction that $\mathbb{R}$ is countable, so we have a function $f: \mathbb{N} \to \mathbb{R}$ that shows $\mathbb{R}$ is countable. We can imagine building a table of $n \in \mathbb{N}$ and $f(n) \in \mathbb{R}$ — we have to be able to write down a list in which every real number appears.

n	$f(n)$
1	
2	
3	
4	

This seems innocent enough to possibly be doable. Suppose that you don't believe this is impossible, and you stay up late thinking about this instead of playing your video games; and you come up to Prof. Sipser the next day and say you came up with a correspondence, and show him your table.

n	$f(n)$
1	$3.14159\ldots$
2	$2.71828\ldots$
3	$22.22222\ldots$
4	$0.0000\ldots$

Now we're going to show that you have not succeeded, and we'll do that by finding a real number that's missing from the list. And we'll do this by actually constructing that real number, digit by digit.

We want to find some $x \in \mathbb{R}$ that's missed. We'll construct x by starting with a 0 before the decimal point, and following the decimal point, we'll look at the table. For the first digit after the decimal point, we'll look at the first table entry, and look at its digit following the decimal point, which is a 1. And we instead place anything that's *not* a 1 in that location (let's say a 2).

For the next digit, we look at the next number in the next digit. This is also a 1, so we pick some other number not equal to 1, for example 3.

(We have some flexibility because we're in decimal — we have 9 choices for our digit, as it just has to be different from the one we're comparing to.)

Then we look at the third digit after the decimal point in the third row; that's a 2, so we pick anything that's not a 2, e.g. 1.

(We won't ever pick a 9 because we have minor issues if we get e.g. $0.9999\ldots$; but this isn't important, and we won't think about it.)

And then for the fourth digit, we pick something different from a 0, e.g. a 2. And we just keep going — so we end up with some

$$x = 0.2312\ldots.$$

n	$f(n)$
1	$3.14159\ldots$
2	$2.71828\ldots$
3	$22.22222\ldots$
4	$0.0000\ldots$

The number x we've picked is certainly a real number (since we've written down its decimal expansion); but how do we know it's not in the table? What if it's in the 22nd place?

Well, we know it's not in the 22nd place, because x is different from the number appearing in the 22nd place, in the 22nd digit! By construction, x has been built to be different from each of the numbers that appears in any one of these entries. So it has to be a number left out of the table. □

Remark 8.11. You might say, am I only missing one number — can I fix this by just putting x at the top? But as this construction indicates, you're actually missing nearly *every* real number.

This is called *diagonalization* because we're going down the diagonal of our table.

§8.3 Back to Turing machines

We're now going to prove that A_{TM} is undecidable.

Proof. Assume for contradiction that A_{TM} is decidable, so a Turing machine H decides A_{TM}. (Imagine you give us a Turing machine that allegedly decides A_{TM}; we want to show that it fails.)

First, let's try to understand — what does H actually do? We're feeding in input $\langle M, w\rangle$ to H, and it should accept if M accepts w and reject otherwise — we can write this compactly as

$$H(\langle M, w\rangle) = \begin{cases} \text{ACCEPT} & \text{if } M \text{ accepts } w \\ \text{REJECT} & \text{if } M \text{ doesn't accept } w. \end{cases}$$

Note that M can reject either by halting or by looping; but H is never allowed to loop (because it's a decider, so it always halts).

This is what we're assuming we have; and now we want to get a contradiction. To do that, we're going to build another Turing machine, called D, which is going to use H as a subroutine. Last time, we built some Turing machines and used them inside others; we'll do exactly the same thing here.

We define D in the following way — D just takes the description of a Turing machine $\langle M \rangle$ as its input. Then it does the following (this might look bizarre at first):

1. D runs H on $\langle M, \langle M \rangle \rangle$.

Before we go on, let's explain what this means. Our Turing machine H expects to get two things together, which should encode to a Turing machine and a string; here our string is the description of the Turing machine itself. Then H is going to try to answer, does M accept its own description? (If I take M, write down its description, and feed that into M itself, does it accept or not?)

You might find this hard to wrap your mind around, but feeding a program into itself does happen (rarely but a nonzero number of times) in the real world, and it can even make sense. For example, suppose you have a Python program, and you have a compiler for Python, and the compiler is *written* in Python — so you can imagine running your Python compiler, and its input is the Python compiler. You might think what's the point of that, but there is a point — in C (Prof. Sipser isn't sure if this is done in Python), there's a standard C compiler, and an optimizing C compiler (that does some tricks to make the code more efficient). The optimizing C compiler is written in C. So you first feed this into the standard compiler (so you now have an optimized compiler that runs more efficiently). But now suppose you want to have an optimized optimized compiler — this can cause you to feed a program into itself.

And here we're just taking a Turing machine, running it on its own description, and seeing what it does. And H can figure this out; D is going to take H's answer and do the opposite.

(2) Accept if H rejects, and reject if H accepts.

Now we're almost there. Let's try to unpack a bit further, what is D actually doing? This Turing machine D accepts an input $\langle M \rangle$ if and only if M *doesn't* accept $\langle M \rangle$. What D does, when it gets some Turing machine, is that it imagines feeding the Turing machine description into the Turing machine, uses H to figure out whether M would accept this description, and does the opposite. So D accepts $\langle M \rangle$ if and only if M wouldn't accept this description, and it does this using H.

But now imagine running D on its *own* description $\langle D \rangle$. Then plugging in D for M, we get that D will accept $\langle D \rangle$ if and only if D doesn't accept $\langle D \rangle$. This is clearly impossible, and that's our contradiction.

This is our proof that H cannot exist — because if H existed, then D would exist, and we'd get a contradiction. □

At first glance, it looks like total magic that this works. But there's a way of looking at this *as diagonalization* that makes it clearer.

Imagine that we have a table, similar to the listing of all real numbers — where we list all Turing machines for rows (we can do this since they can be encoded as strings of 0's and 1's) and all descriptions of Turing machines in the columns, and our entries are what each Turing machine would do on each string:

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\cdots$
M_1	Acc	Acc	Acc	$\cdots$
M_2	Rej	Rej	Rej	$\cdots$
M_3	Acc	Rej	Acc	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

(We can have rejections either by looping or halting.)

If A_{TM} were decidable, then I could use H to fill out this table — H can actually output what M_1 does on the description of $\langle M_2 \rangle$, for instance. So H can report on the Accepts and Rejects in this table (and it's a decider, so it always gives the answer).

Now we build this Turing machine D; if H is a Turing machine then D is also a Turing machine, so it has to appear on this list.

What does D do? We can actually fill out the table for D — for example, what does D do on $\langle M_1 \rangle$? It checks what M_1 would do on $\langle M_1 \rangle$ (using H), and then it does the opposite. What does H do on $\langle M_2 \rangle$? It checks what M_2 would do on $\langle M_2 \rangle$; and M_2 rejects its own description, so D does the opposite.

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\dots$
M_1	Acc	Acc	Acc	$\dots$
M_2	Rej	Rej	Rej	$\dots$
M_3	Acc	Rej	Acc	$\dots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$
D	Rej	Acc	Rej	

So we can see that we're going down the diagonal and reversing things.

But if you extend this diagonal out to D, what does D do on the description of D? Something goes terribly wrong — because D is going to calculate the value in that square and then be different from it, but that value cannot be different from itself. So D is going to fail in doing the job it's designed to do; and D just depends on H, so H could not exist.

> **Remark 8.12.** The table is certainly defined mathematically; the point (that lets us get a contradiction) is that we have an algorithm to calculate it, which must actually be an entry of the table.

§8.4 An unrecognizable language

Now we'll give an example of an argument that isn't even recognizable — the *complement* of A_{TM}. (This is in a certain sense even worse.) This will be a corollary of the previous result, but it's a nice result on its own.

First we need to prove a theorem that'll enable this:

> **Theorem 8.13**
> If A and $\overline{A}$ are both Turing-recognizable, then A is decidable.

(Note that being decidable is closed under complement — your decider is always going to accept or reject, and you can invert the answer to get a decider for the complement. So it follows immediately that $\overline{A}$ is also decidable. However, we'll soon see that being Turing-recognizable *isn't* closed under complement; and we'll use this to prove that.)

Proof. Suppose that R recognizes A and S recognizes $\overline{A}$. We'll construct a Turing machine T that decides A. This Turing machine T operates in the following way: suppose we have some string w, and we want to be able to decide whether it's in A or not.

To do so, we run *both* R and S on w, until one of them accepts — R and S are recognizing complementary languages, so every string is going to be accepted by one or the other (because every string is either in the language or in its complement).

So eventually one or the other is going to accept, and that'll tell us whether we were in A or its complement — we accept if R accepted, and reject if S accepted. (Of course they both can't have accepted, because they're doing complementary languages.) □

So if A_{TM} and $\overline{A_{\mathsf{TM}}}$ were both Turing-recognizable, then A_{TM} would be decidable. We know it's not decidable — we just proved that — so one of A_{TM} and its complement must not be Turing-recognizable. But we showed that A_{TM} *is* recognizable, so it must be its complement that isn't. To write this down explicitly:

Theorem 8.14

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

Proof. If both A_{TM} and $\overline{A_{\mathsf{TM}}}$ were recognizable, then A_{TM} would be decidable, which is not the case. We know A_{TM} is recognizable, so $\overline{A_{\mathsf{TM}}}$ must be the one that's not recognizable. □

§9 October 5, 2023

We've been looking at Turing machines and undecidability; and we used diagonalization to show that A_{TM} is undecidable. Even further, we showed that there's a Turing-unrecognizable language, namely $\overline{A_{\mathsf{TM}}}$. Today we'll build on that to find a method for showing other languages are undecidable, using our knowledge that A_{TM} is undecidable (or that its complement is unrecognizable). That's called the *reducibility method* (and we'll see something similar in complexity theory).

§9.1 The Halting Problem

To illustrate the idea, we'll look at a new language, the *halting problem* (which is also very famous) — it's very similar to A_{TM}, but instead of talking about whether the Turing machine *accepts* an input, it asks whether the Turing machine *halts*.

Theorem 9.1

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

We could prove this by going through diagonalization again, but instead we'll do it by using our knowledge that A_{TM} is undecidable and showing from *that* that $\mathsf{HALT}_{\mathsf{TM}}$ is undecidable.

Proof. We'll use a proof by contradiction — we'll show that if $\mathsf{HALT}_{\mathsf{TM}}$ *were* decidable, then A_{TM} would also be decidable. We know A_{TM} is not decidable, so this will give us a contradiction.

Assume for contradiction that some Turing machine R decides $\mathsf{HALT}_{\mathsf{TM}}$. We'll use R to construct a Turing machine S that decides A_{TM} (which is impossible).

We define S in the following way. We want S to try to solve the A_{TM} problem (we can't do this, but don't let that throw you — S has built into it a $\mathsf{HALT}_{\mathsf{TM}}$ decider, which is going to help it). On input $\langle M, w\rangle$, we're trying to decide whether M accepts w (assuming we have R, which can decide if M halts on w).

How can we use R to make S? First, let's run R on $\langle M, w\rangle$. Sometimes, depending on what R's answer is, this is useful — suppose R reports back that M doesn't halt on w. Then we're done, because there's no way that M could be accepting w if it doesn't even halt on w — it'll be rejecting by looping. So in that case, we've gotten the answer — if R says that M loops on w (i.e., R rejects $\langle M, w\rangle$), we can reject (because M couldn't have accepted).

But suppose that R says that M halts on w. What do we know at that point? We don't know that M accepts w or that it rejects w. However, this is still very helpful — if we know that M halts on w, and we want to find out whether M accepts or rejects w, then knowing that M halts we can just simulate it — now we can run M on w, without having any worries that we're going to get into an infinite loop ourselves (because we're already guaranteed by M's answer that M halts on w).

So if R says M halts on w (i.e., R accepts w), then we run M until it halts (which will eventually happen); and we accept if M accepts, and reject if M halts and rejects.

Here we're using the decider for $\mathsf{HALT}_{\mathsf{TM}}$ to decide A_{TM}; and therefore the decider for $\mathsf{HALT}_{\mathsf{TM}}$ cannot exist. □

§9.2 Reducibility

This is an easy first example to illustrate the idea of reducibility. Reducibility is essentially a way to use the solution to one problem to solve another problem.

> **Definition 9.2.** Informally, we say that A is *reducible* to B if we can decide A using a decider for B.

Here, what we've done is shown that A_{TM} is reducible to $\mathsf{HALT}_{\mathsf{TM}}$ — we've shown how to decide A_{TM} using a decider for $\mathsf{HALT}_{\mathsf{TM}}$.

There are two ways to use this idea of reducibility — in a positive way and in a negative way. The positive way is that if we know that B is decidable and we also want to decide A, we can show that A is reducible to B — we can show how to use the B-decider to decide A. We've already done this (at least) once before — previously, we looked at decision problems for automata and grammars, and we showed that A_{DFA} was decidable. And then we showed that A_{NFA} was decidable. But we didn't start from scratch there — instead, we recalled that we can convert a NFA to a DFA, and we used a DFA decider. So we basically reduced the A_{NFA} problem to the A_{DFA} problem — we used the solution to the A_{DFA} problem to solve A_{NFA}. This happens often in mathematics or more generally — if you can reduce a problem to some other problem you've already solved, then you've solved the original problem. So that's the positive sense — if A is reducible to B and we know a decider for B, then we get a decider for A.

But there's a negative application of the idea as well, and that's what we'll mainly focus on — if we show that A is reducible to B and we know that A is *not* decidable, then neither is B. This is really saying the same thing (it's logically equivalent), but it might be more to digest.

So to summarize, if A is reducible to B:

- If B is doable (we'll use this in more general senses than just being decidable later), then so is A.
- If A is *not* doable, then neither is B.

It's this second direction that's going to be our main usage of reducibility.

> **Remark 9.3.** The take-home message here is that if you want to show that some problem B is undecidable, you can do so by showing that A_{TM} is reducible to B. We did this earlier — we showed that A_{TM} is reducible to $\mathsf{HALT}_{\mathsf{TM}}$.

§9.3 Emptiness problem for Turing Machines

Now we'll look at another example, the emptiness problem for Turing machines (the language consisting of all Turing machines that don't accept any input at all).

Theorem 9.4

The language $E_{\mathsf{TM}} = \{\langle M\rangle \text{TM } M \text{ with } L(M) = \varnothing\}$ is undecidable.

Proof. We'll reduce A_{TM} to E_{TM}. To do this, assume for contradiction that we have a Turing machine R that decides E_{TM}. We'll construct a Turing machine S that decides A_{TM}. (All these proofs begin in the same way; as usual, this Turing machine S gives the desired contradiction. We're trying to show E_{TM} is undecidable; we assume it *is* decidable, and use that to show how to decide A_{TM}, which we already know is undecidable.)

We'll define S as follows: S is an A_{TM} decider, so its input will be $\langle M, w\rangle$, and S wants to figure out whether M accepts w (using the E_{TM} solver).

As a first attempt, why can't we run R on $\langle M, w\rangle$? This doesn't work, because R is not expecting an input of this form — R answers questions just about a Turing machine with no input (whether that Turing machine's language is empty). So we can't do this (R will just reject it for being of the wrong form).

To experiment (and illustrate the thought process), let's try to run R on M. Suppose that R says M's language is empty (that's the kind of answer R gives — it either says M's language is empty, or it's not empty). Then we're done — we're trying to figure out whether M accepts w, and if M's language is empty then it certainly doesn't accept w, because it doesn't accept *anything*. So if R accepts (because M's language is empty), then we can reject (because we know M doesn't accept w). That's good news. But now suppose R says M's language is nonempty; what can we do then? We don't know anything about whether M accepts w — all we know is that M accepts *something*, which may be w or may be something else.

So we're going to have to do a trick to solve this problem (this will be a standard method we use many times). Instead of running R on M, we're going to first modify M.

At this point, we know what M and w are, because those have been provided as input to S. Now S is going to build a *new* Turing machine, which is first going to eliminate all inputs that are *not* w — it's going to check and reject all the non-w inputs. (So it's a modified version of M that filters out all the annoying strings that could cause the language to be nonempty in ways that aren't useful to us.) For w, we'll have this new Turing machine do the same thing as M. But now, if the new Turing machine's language is nonempty, then we know it must accept w (because it rejects everything else).

Explicitly, here's our construction (on input $\langle M, w\rangle$):

(1) We construct a new Turing machine M_w (based on M and w), which does the following on input x:
   (1) If $x \neq w$, reject.
   (2) If $x = w$, run M (on x, which is w).
   (3) Behave in the same way as M — accept if M accepts, and reject if M halts and rejects.

   (Of course, if M were to loop on w, then because M_w is simulating M, it will be looping as well — but this isn't written in its code.)

(2) Now that we've built this new machine M_w (a modified version of M that rejects everything except w, and does the same thing for w as before), we run R on $\langle M_w\rangle$.

(3) If M_w has an empty language, then the original machine must be rejecting w (because we didn't change what the behavior was on w). But now, if M_w has a nonempty language, then it must be because the original M accepted w, because we know M_w rejects everything else. So if R accepts (i.e., $L(M_w) = \varnothing$) then we reject; if R rejects (i.e., $L(M_w) \neq \varnothing$, so M_w must be accepting *something*, which must be w), then we accept. □

This shows that E_{TM} is undecidable.

§9.4 Mapping reducibility

Now we're going to do something a bit trickier — we're going to formalize the notion of reducibility in a particular way. There are many ways of formalizing the notion of reducibility, depending on what you're trying to do. The method we've seen here is fine for proving languages are undecidable; but it's inadequate for proving languages are Turing-unrecognizable. So we need to develop a more precise and refined notion of reducibility to prove languages are unrecognizable using that method. That's going to be something called *mapping reducibility*.

For this, we need the following definition (which is important anyways).

Definition 9.5. We say a Turing machine M *computes* the function $f: \Sigma^* \to \Sigma^*$ if M always halts with $f(w)$ on the tape when started on input w.

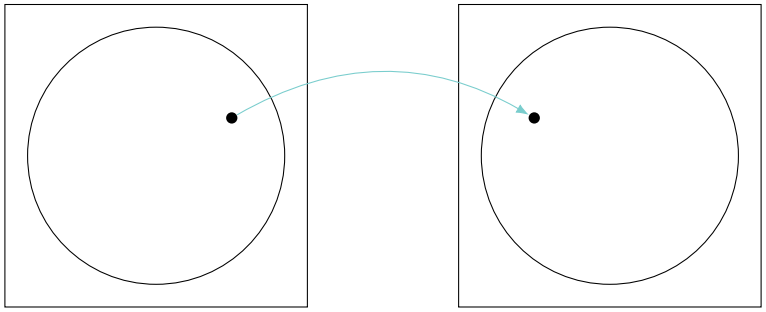
So far, our Turing machines basically had outputs which were accept or reject. But computers can compute more complicated functions than just accept and reject. For here, it'll be useful to have Turing machines that compute general functions from strings to strings — we give a string as input to the program, we turn it on and run it for a while, and eventually it halts with $f(w)$ sitting on the tape in the end. That's how we say that a Turing machine computes some function.

Remark 9.6. This applies to any function $f: \Sigma^* \to \Sigma^*$. But not every function will be computable — even at this point, we can already figure out functions related to the undecidable problems we've found that won't be computable.

But many functions are computable, and those are the ones we'll focus on. (We call functions *computable* if they can be computed by a Turing machine.)

Definition 9.7. We say that A is *mapping reducible* to B (written as $A \leq_{\mathsf{m}} B$) if there is a computable function f where $w \in A$ if and only if $f(w) \in B$.

In pictures, we have some language A, and another language B. What the mapping function f does is that if we feed it something in A, it maps it to something in B; and if we feed it something outside A, it maps it to something outside B.



We'll use this idea in the following way:

Theorem 9.8

If $A \leq_{\mathsf{m}} B$ and B is decidable, then so is A.

This is the idea of reductions that we described informally — if we have A that's reducible to B, and we can solve B, then we can solve A. But now we're doing this reduction in a precise way — we consider a mapping where strings in A go to strings in B, and strings out of A go to strings out of B.

To understand this, think of A and B as problems, and f as a transformer between problems — it transforms A-problems into B-problems. If we want to test whether something's in A, and we already know how to test if something's in B, then f allows us to map the A-problem on w to a B-problem on some other string, which we know how to solve.

Proof. Let the Turing machine R decide B; we'll construct a Turing machine S that decides A.

We have two languages A and B, and we assume we have a function that we can compute with a Turing machine, that takes things in A to things in B, and things out of A to things out of B. We want to use this to show that if we have a B-decider R, we can make an A-decider S. Here S has some w, which is either inside A or not; we don't know. How is our algorithm going to work to test if $w \in A$?

We simply apply the function and check if the result is in B — we compute $f(w)$, run R on $f(w)$, and give the same answer. □

But importantly, this theorem still works if we replace 'decidable' with 'Turing-recognizable.'

Theorem 9.9

If $A \leq_{\mathsf{m}} B$ and B is Turing-recognizable, then so is A.

Proof. We can use the same proof — if here instead of a decider for B we have a recognizer for B (something that accepts things in B, but may reject things possibly by looping) and we want the same for A, we can still take w, map to B, and run the B-recognizer; and that's going to be an A-recognizer. □

Remark 9.10. The value of this is that we get a new take-home message: to show that B is Turing-unrecognizable, we can show that some unrecognizable problem A is mapping-reducible to B. And we conveniently already have an unrecognizable language, namely A_{TM}. So we can do so by showing that A_{TM} is mapping-reducible to B.

So we have a general type of reduction that we can use to prove decidability; but to prove unrecognizability we need to use this more specific kind of reduction.

The reason is that in a general reduction, you *can* reduce an unrecognizable problem to a recognizable one — for example, you can reduce $\overline{A_{\mathsf{TM}}}$ to A_{TM}, just by flipping the answer. But mapping reducibility is a restricted form of reductions that bans taking the complement; this doesn't allow $\overline{A_{\mathsf{TM}}}$ to mapping-reduce to A_{TM}. (This is a bit slippery.)

Remark 9.11. There's an if and only if in the mapping reducibility definition, so why don't we get the theorem that if $A \leq_{\mathsf{m}} B$ and A is decidable, then B is also decidable? It looks like everything is symmetric — $w \in A$ if and only if $f(w) \in B$. But it isn't actually — because f might not even be a function in the other direction. There might be elements in B that don't come from any things in A, and we won't know how to map backwards starting from that element.

Remark 9.12. Note that a Turing machine cannot halt and produce an infinite output; we'll never have infinite strings in this class. Also note that Σ^* only consists of *finite* strings.

§9.5 The equivalence problem for Turing machines

We'll now apply mapping reducibility to show that some more problems aren't even unrecognizable.

Theorem 9.13

Let $\mathsf{EQ_{TM}} = \{\langle M, N\rangle \mid M \text{ and } N \text{ are TMs with } L(M) = L(N)\}$. Then both $\mathsf{EQ_{TM}}$ and $\overline{\mathsf{EQ_{TM}}}$ are Turing-unrecognizable.

We've talked about the equivalence problem for automata and for grammars (where we mentioned it won't be decidable). Here it's unsurprisingly not decidable; but more than that, it's not even recognizable, and neither is its complement. So this is a language that's worse than any of the ones we've seen so far — we saw earlier that $\overline{A_{\mathsf{TM}}}$ wasn't recognizable, but its complement was. But here neither $\mathsf{EQ_{TM}}$ *nor* its complement is recognizable.

Proof. Here we have two different languages we're working with. We'll work with each in turn — we'll show that $\overline{A_{\mathsf{TM}}} \leq_{\mathsf{m}} \mathsf{EQ_{TM}}$ and that $\overline{A_{\mathsf{TM}}} \leq_{\mathsf{m}} \overline{\mathsf{EQ_{TM}}}$.

We'll first show that $\overline{A_{\mathsf{TM}}} \leq_{\mathsf{m}} \mathsf{EQ_{TM}}$. To do so, we need to find a function f.

We need to define what f does on all strings, but the only interesting strings are those that look like A_{TM} problems, so let's only focus on those — suppose that f has input $\langle M, w\rangle$. We should think of f as a problem transformer, transforming $\overline{A_{\mathsf{TM}}}$ problems into $\mathsf{EQ_{TM}}$ problems — so its output should look like $\langle T, U\rangle$ for two Turing machines T and U. We want that if M *rejects* w then T and U should be equivalent (if $\langle M, w\rangle \in \overline{A_{\mathsf{TM}}}$ then we should have $\langle T, U\rangle \in \mathsf{EQ_{TM}}$).

To do this, we define T to be the always-reject Turing machine. Meanwhile U will be the one that does the main work. It'll be similar to the machine M_w we defined before: on an input x, we run M on w, and do what M would do.

Let's understand what U does — why do we have both x and w? What U does is no matter what input it gets, it ignores it, and it always runs M on w (on every input). If M accepts w, then U is going to accept every input. And if M rejects w, then U is going to reject every input. So this is a way of expanding what M does on one input into a machine that does the same on all its inputs. (This is kind of similar to M_w we defined earlier, where we erased all the non-w inputs; here we're copying what we do on w to all the inputs.)

So U accepts either everything or nothing; its language is either $\varnothing$ or Σ^*. And its language is empty exactly when M rejects w. In that case, U rejects everything, and is therefore equivalent to T (which was defined to reject everything). That's what we want.

Now if we have a solver for $\mathsf{EQ_{TM}}$, we would get a solver for $\overline{A_{\mathsf{TM}}}$. But more to the point, if we even had a *recognizer* for $\mathsf{EQ_{TM}}$ then we'd have one for $\overline{A_{\mathsf{TM}}}$; so $\mathsf{EQ_{TM}}$ is not even Turing-recognizable.

> **Remark 9.14.** If M never halts on w, then U doesn't halt on anything. But this means that U's language is still empty, and T's language is also empty. It doesn't matter how U gets to the empty language, as long as its language is still empty — however M rejects w doesn't matter.

So if M rejects w then T and U are equivalent. Meanwhile, if M accepts w, then U's language is Σ^*, and so U is not equivalent to T (whose language is nothing). So T and U are equivalent if and only if M rejects w, which is what we want.

> **Remark 9.15.** Instead of using U, we could also use M_w, the machine that rejects all the non-w things. Then if M rejects w, that machine has an empty language and is equivalent to T; and if M accepts w, then its language will be $\{w\}$, and it won't be equivalent to T. So this will work. But the reason we did it this way is to facilitate the second part (which is straightforwards now).

Now to show that $\overline{A_{\mathsf{TM}}} \leq_{\mathsf{m}} \overline{\mathsf{EQ_{TM}}}$, we want $\langle T, U\rangle$ to be equivalent if and only if M *accepts* w (the opposite of what we had before). To do this, we can replace T by the always-accept Turing machine. □

§10 October 12, 2023

§10.1 Review

We've been working on how to prove undecidability and Turing-unrecognizability using reductions. We started off by showing A_{TM} was undecidable using diagonalization; and then by taking reductions from A_{TM} and mapping reductions from its complement, we were able to show other problems were undecidable or unrecognizable.

As a quick review of how we use these techniques, we have the following template to help us get started: if we want to show a problem B is undecidable, we can reduce some known undecidable problem (like A_{TM}) to B. The form of this reduction is that we assume for contradiction we have a Turing machine R that decides B; and we use this to construct a new Turing machine S that decides A. Undecidability proofs generally all look like this.

If we want to show something stronger — that a language is *unrecognizable* — then we have to use a specific form of a reduction, called a mapping reduction. We have to give a mapping that takes strings in A to strings in B, and strings out of A to strings out of B.

§10.2 Overview

So far, we've shown various problems are undecidable or unrecognizable; and those problems have all involved Turing machines. In some sense, you might think that the undecidability of problems about Turing machines is almost to be expected — it's believable that you can't make a Turing machine that decides what other Turing machines can do. And the methods we've developed so far support that.

But many other problems that have nothing to do overtly with computation are undecidable too. Today we'll develop a method for proving their undecidability, called the *computation history method*.

§10.3 Post correspondence problem

We'll illustrate this method with a problem.

Imagine we're given a collection of *dominos*, each of which consists of a top string and a bottom string — for example,

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

A *match* is a sequence of dominos from the collection P (where repetitions are allowed) such that if we stick them next to each other, then the string we get from concatenating the tops is the same as the string we get from concatenating the bottoms.

For example, consider the above collection

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

We want to take dominos from this collection and stick them together into some sequence of dominos, so that the strings we get reading along the top and bottom are the same.

Here there's only one possible domino we can start with (since if we start off with the first domino, we start with aa on the top and ab from the bottom, so they can't possibly match) — we have to start with

$$\begin{bmatrix} ab \\ aba \end{bmatrix}.$$

Then we can take the first domino, so we now have

$$\begin{bmatrix} ab \\ aba \end{bmatrix} \begin{bmatrix} aa \\ aba \end{bmatrix};$$

we can see that we're starting to fulfill the conditions of being a match (the top and bottom strings agree). Now we need something starting with ba in the top; so we can take

$$\begin{bmatrix} ab \\ aba \end{bmatrix} \begin{bmatrix} aa \\ aba \end{bmatrix} \begin{bmatrix} ba \\ aa \end{bmatrix} \begin{bmatrix} aa \\ aba \end{bmatrix} \begin{bmatrix} abab \\ b \end{bmatrix}.$$

And we have a match.

Question 10.1. Suppose I give you a finite collection of such dominos, and I'd like to know, is there a match?

This seems like a fairly straightforward question. But it's actually undecidable, and we'll prove that today.

Theorem 10.2

Let PCP $= \{\langle P \rangle \mid P$ is a finite collection of dominos that has a match$\}$. Then PCP is undecidable.

This should be surprising — it seems to be a reasonable question that has nothing to do with Turing machines.

In order to prove this, we'll introduce a method called the *computation history method*. Before we get into the details, we'll put this on hold and look at a different problem which can be solved using that method. This problem will be less motivated (it's again about Turing machines), but it will illustrate the method in a more clear way; and then we'll see how to apply the method to this problem as well.

§10.4 Linearly bounded automata

We'll now introduce a new question about Turing machines, or rather, a variant called a *linearly bounded automaton*.

Definition 10.3. A *linearly bounded automaton* (LBA) is a (single-tape, deterministic) Turing machine that has only enough tape to hold the input.

This means that our tape is no longer infinite; instead it's a finite tape (going back to some earlier automata we've seen). The Turing machine is only able to use this finite amount of tape — it can write on the tape (just like an ordinary Turing machine), but if it tries to move off the ends of the tape, it can't. (In an ordinary Turing machine, if we try to move on the left end, we instead stay there; here the same is true for the right end.)

The tape length adjusts based on the input — we just give the machine enough tape to hold the input. Then it can work just as before; it just doesn't have available to it the originally blank portion of the tape. So it can do whatever it wants on the tape it has, but there's no other tape it can use for recording things.

Equivalently, a LBA is an ordinary Turing machine that's not allowed to go into the blank portion of the tape.

LBAs are reasonably powerful — they can still do $\mathtt{a}^k\mathtt{b}^k\mathtt{c}^k$, for instance (the original procedure we gave for this doesn't use any additional tape — we just repeatedly cross off an `a`, `b`, and `c`). But as we'll see later, it's not as powerful as a general Turing machine.

As with other machines, we can look at the acceptance problem for LBAs.

Definition 10.4. We define $A_{\mathsf{LBA}} = \{\langle B, w\rangle \mid B \text{ is a LBA and accepts } w\}$.

Theorem 10.5

The language A_{LBA} is decidable.

This is the conventional notion of decidability, for regular Turing machines — we have an algorithm to test whether a given LBA accepts its input.

Remark 10.6. Why is it called a *linearly* bounded automaton? This has more to do with the complexity section of the course. If you think of the tape as the 'memory' of the machine, the amount of memory this machine has available to it is *linear* in the size of the input. Why isn't it *exactly* the size of the input? If the input is a 2-letter alphabet, the tape alphabet might have 20 letters — so the tape can store a linear multiple of the amount of information that you're given in the input. So that's why we say *linear* — it's not exactly equal.

How would such an algorithm work? Let's try to decide — imagine we're given a LBA and an input. And let's try just running it (our decider algorithm just tries running the LBA).

This is the same thing we tried doing with Turing machines, and it sounds potentially dangerous — because a decider can't run forever. If we're simulating the LBA and the LBA doesn't halt, we're still required to halt. So what do we do?

Because the LBA has only a limited amount of tape available to it, if it goes for a really long time, it's going to have to repeat the contents of the tape — because there's only a limited number of possibilities. And not only will the contents of the tape repeat, but the location of the head and the state of the machine will repeat as well (at the same time).

And once you have a repetition, then you *know* the machine will go on forever.

So you only have to go on for long enough to see that the machine has either halted, or has repeated itself; and then it's going to go forever, and you can just reject.

The trick then is to calculate how long you have to go until you are sure the machine has repeated itself. For that, we're going to have to count the number of different possible *configurations* (consisting of the state, head-position, and contents of the tape). If we can calculate this number of configurations, then we can just go for that long; and if the machine hasn't halted by then, then we just halt and reject.

Definition 10.7. A *configuration* of an LBA on input w is a triple (q, h, t), where q is a state, h the position of the head, and t the contents of the tape.

(This can be defined similarly for general Turing machines.)

Now, how many such configurations can we have for input strings w of length n? We just have to multiply the number of possible values we can have for each of these elements. The number of possible values of q is $|Q|$; the number of possible head-positions is n (the length of the input) — because the only tape available to the machine is the n cells of the input; and the number of possibilities for the tape is $|\Gamma|^n$. So the total number of configurations is

$$|Q| \cdot n \cdot |\Gamma|^n$$

(when we start it with an input of length n).

So this gives an algorithm for an A_{LBA} decider: suppose we have input $\langle B, w\rangle$, where $n = |w|$.

(1) Run B on w for $|Q| \cdot n \cdot |\Gamma|^n$ steps.

(2) If B has accepted, then accept. If B has rejected or is still running, then reject.

Clearly if B has rejected, we know it rejects w. But if B is still running after this many steps, then it's going to loop; so it's repeating itself, and it's never going to accept.

This algorithm is a decider, because it never goes forever.

§10.5 Emptiness problem for LBAs

But we'll consider a different problem about LBAs, the emptiness test. We saw that general TM emptiness testing is undecidable, by a reduction from A_{TM}. Now we'll show that the emptiness problem for LBAs is undecidable.

Definition 10.8. Define $E_{\mathsf{LBA}} = \{\langle B\rangle \mid B \text{ an LBA and } L(B) = \varnothing\}$.

Theorem 10.9

The language E_{LBA} is undecidable.

For this, we'll have to do something fancier, which will illustrate a new technique.

We'll need one more definition:

Definition 10.10. For a Turing machine M on input w, an *accepting computation history* for M on w is the sequence of configurations $C_1, C_2, \ldots, C_\ell$ the machine goes through from the start until it accepts w.

If M is accepting w, it does so in some number of steps (let's say 100 steps). Then step-by-step we watch the machine go; and for each one of those steps, we take the configuration (the state, head position, and tape contents). And we write these down in sequence, from the start (the start state, the head in position 1, and tape w); we run the machine, getting these configurations one by one, until the machine accepts. If the machine accepts, then the last sequence in that configuration will have the machine in an accept state.

If M does *not* accept w (either by halting and rejecting or by looping), then there is no accepting computation history.

We will convert the problem of testing whether M accepts w to the problem of testing whether there is an accepting computation history of M on w — and that's a problem we can test with an LBA. We'll make a LBA whose language is the set of accepting computational histories of M on w (there is one if M accepts w and none otherwise).

Remark 10.11. Computation histories exist in the real world, for algorithms — you can get a debugger to store the sequence of the contents of the variables as the machine runs, and use that to see where it's going wrong. This is a theoretical version of the same concept, for Turing machines.

Remark 10.12. LBAs for us will be deterministic. (We can talk about nondeterministic LBAs (and some people do), but for now they're deterministic.)

It will turn out to be convenient to be a bit more explicit about how we represent the computation history as a string — because we're going to make a language out of it. So we need an encoding of configurations.

Imagine that the machine is in a state q, the tape is `abcd` (followed by either the end of the tape or blanks, depending on whether we have a LBA or TM), and the head is on position 3 (the `c`). We'll encode this as

$$\texttt{ab}q\texttt{cd}.$$

Instead of writing down the number for the head position, we'll put the state in the middle of the tape, to the left of the symbol it's reading. This string represents the entire configuration — you can figure out the tape contents, the state, and the head position from here.

We'll write down a computation history as a sequence of such encodings separated by the marker # — i.e.,

$$C_1\#C_2\#\cdots\#C_\ell.$$

If we have an accepting computation history, we'll first have $C_1 = q_0 w_1 \cdots w_n$. Then we'll go through some sequence of configurations; and eventually we'll have a configuration C_ℓ with q_{acc}.

§10.5.1 Proof of undecidability

The main point is that we can make a LBA that tests whether its input is the accepting computation history of a given machine.

Assume for contradiction that a Turing machine R decides E_{LBA}; we're going to make a Turing machine S that decides A_{TM}. (So we're just going to give a reduction from A_{TM} to E_{LBA}; but this will have the extra wrinkle of involving accepting computation histories.)

Suppose that S has input $\langle M, w\rangle$. So we have a Turing machine M, and we want to know, does M accept w? Here M is a general Turing machine, not necessarily a LBA — we're trying to solve the general acceptance problem for Turing machines, using an emptiness tester for LBAs.

You might try to take M and feed it into R, and ask, is its language empty? But then R will be unhappy, because M is not necessarily a LBA — it might be using more tape. So R won't be able to give us a correct answer on anything regarding M.

Instead, what we'll have to do is take M and construct an associated new machine guaranteed to be an LBA, that we can use in R. That LBA is going to be a *computation history checker* — because *checking* a computation is easier than doing a computation. So an LBA can do that without using any extra tape. For one thing, we're *giving* it a ton of tape — because we write down the entire computation history, which is a very long thing — and it's just going to have to check that this computation history is valid.

We take M and w, and construct a LBA B (depending on M and w) which does the following: on input x, it tests if x is an accepting computation history for M on w.

So here we have B, and it takes some input; and its input is supposed to be a sequence of configurations

$$C_1\#C_2\#\cdots\#C_\ell$$

of M and w. Checking that it's legit is not hard (note that M and w are built into B — because we're constructing B out of M and w). First we need to check that C_1 is the correct start configuration — i.e., $q_0 w_1 \ldots w_n$. It also looks at the very last configuration, and checks that it has an accepting state in it. If either of those fail, then x is certainly not an accepting computation history, so we immediately reject.

For each of the in-between states, the LBA can look at the current configuration and the previous one; and by zigzagging back and forth (potentially marking the symbols to keep track of where it is), it can check that C_1 validly leads to C_2, and C_2 to C_3, and so on. This is a straightforward check — you need to check that the tape remains the same everywhere except at the head, where it gets updated according to the rules of M. And it can do this as an LBA — it doesn't need any additional tape.

> **Remark 10.13.** How do we know that the computation history is finite? If M accepts w, it does so in finite time. (All strings are finite — the input to all of our machines are finite strings. So infinite computation histories are just not a thing.) If M ran forever on w (meaning it rejects by looping), then there'd be no accepting computation history — so any finite string x we give B is going to get rejected.

Remark 10.14. How do we construct the computation history in the first place? We don't. We are only constructing a computation history *checker*. Making a *checker* is not hard — we could write code for this (that, given a computation history, checks if it is legit). We don't know whether there is one, but we can make a checker; and then we test whether the checker's language is empty. If that checker's language is empty, then there wasn't an accepting computation history, and M didn't accept w. If its language is nonempty, then this means there is some accepting computation history. So we're going to use the emptiness tester *on the checker* to see if there's some string out there that it accepts.

Now that we've constructed B, we use R to test whether $L(B) = \varnothing$. We accept if not (because if $L(B)$ is nonempty, then there was an accepting computation out there that B accepted), and reject if yes.

So that's how we use the emptiness tester for LBAs to decide A_{TM}.

§10.6 Back to PCP

We'll now come back to the post correspondence problem — we're given a bunch of dominos, and we want to know whether there is a match.

This uses a similar idea in a certain sense — we're still going to use computation histories, but in a different way. To show PCP is undecidable — i.e., that testing whether a match exists is an undecidable problem — we'll again reduce from A_{TM} to PCP, using the computation history method.

Assume that we have a Turing machine R that decides PCP; we'll construct a Turing machine S that decides A_{TM}.

As usual, S is an A_{TM} decider, so it takes input $\langle M, w\rangle$. We need to construct a PCP problem $P_{M,w}$, consisting of a bunch of dominos (which we will describe soon). The idea is that we'll construct $P_{M,w}$ so that a match corresponds to an accepting computation history for M on w. Then we'll use R to test whether P has a match. We'll accept if yes, and reject if no.

So the whole question now becomes, how do we make the dominos? This is kind of a mess, but we'll try to see the ideas.

We'll first make an assumption about the match; this is kind of cheating, but if we have time we'll discuss how to eliminate it — we'll assume that in our dominos, there is a special starting domino, and the match only counts if we start with that particular starting domino.

Our $P_{M,w}$ needs to consist of a bunch of dominos. We'll have our start domino be

$$\left[\begin{matrix}\# \\ \#q_0 w_1 \dots w_n \#\end{matrix}\right].$$

(We're going to assert that if there is a match, it has to start with this domino.)

Imagine we start building a match as we go along — as an example, let's suppose $w = \texttt{aba}$, so we start with

$$\left[\begin{matrix}\# \\ \#q_0\texttt{aba}\#\end{matrix}\right].$$

The idea is to put dominos in such a way that building a match forces you to simulate the machine — that the only way a match could occur is by actually doing the steps of the machine.

How can we do this? If $\delta(q, a) = (r, d, \mathrm{R})$ (when the machine is in state q and its head is looking at an a, then it moves to state r, writes a d on top of the a, and moves right), this will correspond to putting in a particular domino into our collection — we'll add the domino

$$\left[\begin{matrix}qa \\ dr\end{matrix}\right].$$

And we do this for every q, a, d, and r — every rule in our transition function corresponds to adding another domino.

Why? Let's turn to our example. Imagine that we have a configuration $q_0\texttt{aba}$, and $\delta(q_0, \texttt{a}) = (q_7, \texttt{d}, \text{R})$. Then according to our domino construction, we get the domino

$$\begin{bmatrix} q_0\texttt{a} \\ \texttt{d}q_7 \end{bmatrix}.$$

Now when we're trying to extend our match, we need a domino that starts with q_0a to put here; that's going to force me to put down dq_7 next.

Then in addition, for the symbols not near the head, we need additional dominos that allow us to copy them over — we add

$$\begin{bmatrix} a \\ a \end{bmatrix}$$

for all a in the tape alphabet, as well as

$$\begin{bmatrix} \# \\ \# \end{bmatrix}.$$

That means we can take

$$\begin{bmatrix} \# \\ \#q_0\texttt{aba}\# \end{bmatrix} \begin{bmatrix} q_0\texttt{a} \\ \texttt{d}q_7 \end{bmatrix} \begin{bmatrix} b \\ b \end{bmatrix} \begin{bmatrix} a \\ a \end{bmatrix} \begin{bmatrix} \# \\ \# \end{bmatrix}.$$

Now we've extended the match using our dominos; and what we have sticking out at the bottom is the configuration at the *next* step of the Turing machine. And we can do the same thing — as we build the match configuration by configuration, we're simulating the machine.

And we keep doing this.

> **Remark 10.15.** The main idea here is that you want to put in the dominos that make you simulate the machine.

Now suppose the machine accepts with the computation history we've produced so far; this means we end up with a configuration at the bottom that has q_{acc}. We want this to be a match, and we're not done — because the bottom sticks out (it's consistent with the top everywhere except it has this extra stuff, which is the accepting configuration). So now we're going to add some additional dominos, which are extra 'fake steps' of the machine that cause the q_{acc} state to 'eat' the tape.

So we'll have new 'endgame' dominoes of the form

$$\begin{bmatrix} q_{\text{acc}}a \\ q_{\text{acc}} \end{bmatrix}, \begin{bmatrix} aq_{\text{acc}} \\ q_{\text{acc}} \end{bmatrix}.$$

Now instead of simulating a move, this causes the contents of the tape to shrink — so we have q_{acc} consume the tape until there's nothing left except for q_{acc} sitting by itself. Now we'll have $q_{\text{acc}}\#$ at the bottom; and we need one last thing to finish the entire match, which is

$$\begin{bmatrix} q_{\text{acc}}\#\# \\ \# \end{bmatrix}.$$

And then we've got a match.

So there's an endgame that allows the top to catch up to the bottom if the bottom has entered an accepting state.

We'll now mention one more thing, without proving it. Let

$$\mathsf{All}_{\mathsf{PDA}} = \{\langle B \rangle \mid B \text{ is a PDA and } L(B) = \Sigma^*\}.$$

Theorem 10.16

$\mathsf{All}_{\mathsf{PDA}}$ is undecidable.

We'll prove this using the computation history method. But note that E_{PDA} is decidable, using a simple algorithm we gave a few lectures back (for grammars). Interestingly enough, we can test whether a CFG generates nothing, but not whether it generates everything.

§11 October 17, 2023

§11.1 Midterm information

The midterm is next week during class hours, in Walker (on the top floor). The exam is closed book; we can bring a (double-sided) crib sheet, as described on the bottom of the homework (and on the information page and the sample problems).

The recitation on Friday will go over some review in preparation for the midterm; there are also sample problems for the midterm (which have all been on past midterms) on the homepage.

Finally, the midterm covers everything up through today's lecture. This Thursday's and next Tuesday's lectures will not be on the midterm. (On Thursday we'll shift gears and start talking about complexity theory; the midterm covers only computability theory.)

§11.2 Review

Last time we introduced a higher-level method for proving undecidability called the computation history method. We gave two examples; we'll see one more today. We looked at the emptiness problem for linearly bounded automata (machines that have only enough tape to hold the input) — for those machines the acceptance problem is decidable, but emptiness is still undecidable. We also talked about a more combinatorial-looking problem called the PCP; and we went through a proof that testing whether a set of dominos has a match is also undecidable, even though overtly it doesn't seem like it has anything to do with Turing machines. The method we used is that given a Turing machine M and w, we built a set of dominos (an instance of PCP) for which trying to make a match forces you to simulate M on w — and the only way the set of dominos arrives at a match is if the simulation ends in an accepting state. We then had some extra dominos in there to ensure that what we're building really does work out to be a match.

This method is important, since it's one of the basic ideas for proving undecidability for more complicated objects. Where do we use the computation history method as opposed to garden-variety reductions from A_{TM}? (Both are reductions from A_{TM}.) This generally arises when the problem at hand (that we're trying to prove undecidability for) involves testing for *existence* — when we want to show that testing for the existence of some object is undecidable. In E_{LBA}, the thing you're trying to test the existence of is an accepted string (given a LBA); for PCP you're testing for the existence of a match. We'll see one more example today.

Recall that an accepting computation history is a sequence of configurations (snapshots of the Turing machine at different moments in time, holding what's on the tape, where is the head, and what is the state) such that if we combine them, we get a history from the machine starting out to eventually accepting.

Typically in applications, we'll encode the computation history in different ways depending on what we're trying to do. Last time, we encoded the computation history into a sequence of symbols — each configuration is represented with the location of the head showing the position of the state symbol in the string (for example, the start configuration is $q_0 w_1 \dots w_n$). We simply string these together with #'s to obtain the computation history.

There are other encodings that could be useful depending on context. For example, we earlier mentioned Hilbert's problem

$$D = \{\langle p \rangle \mid p \text{ is a polynomial with integer root}\}.$$

Proving the undecidability of this problem is done by a reduction from A_{TM} using the computation history method. We won't see the proof, but at a high level, given a Turing machine M and input w, you construct a several-variable polynomial; one of these variables is a special variable, such that the only way that we can find an integer solution to the polynomial is if that variable encodes an accepting computation history as a number. (There's some number theory involved in how that actual representation looks.) The other variables are there to help the polynomial verify that this particular number is in fact an accepting computation history of the machine.

When Prof. Sipser was a graduate student, Julia Robinson presented this; the entire course was doing this one reduction. Designing the polynomial involves a certain amount of number theory and fancy algebra. But the method at a very high level is the same.

§11.3 The $\mathsf{All}_{\mathsf{PDA}}$ problem

Theorem 11.1

The language $\textsc{All}_{\mathsf{PDA}} = \{\langle B \rangle \mid B \text{ is a PDA with } L(B) = \Sigma^*\}$ is undecidable.

Here we're given a PDA (or equivalently a context-free grammar), and we want to know, does it accept all possible strings? If we change the problem to find whether the PDA just accepts *some* string (i.e., is its language nonempty), then the problem is decidable (we saw this earlier). But changing to *all* strings makes it undecidable.

Proof. We'll give a reduction from A_{TM} to $\textsc{All}_{\mathsf{PDA}}$ using the computation history method. Assume for contradiction that a Turing machine R decides $\textsc{All}_{\mathsf{PDA}}$; we'll construct a Turing machine S deciding A_{TM}.

On input $\langle M, w \rangle$, S does the following — it's going to take M and w (for which we want to know, does M accept w) and convert it into a PDA $B_{M,w}$, where the PDA accepts everything if and only if M *doesn't* accept w. We'll construct $B_{M,w}$ to accept all strings that are *not* an accepting computation history for M on w.

Keep in mind that there are two machines here — M (which is a Turing machine) and $B_{M,w}$ (which is a PDA). Here $B_{M,w}$ is just being developed as a vehicle — it's constructed to accept an input if and only if it is *not* an accepting computation history for M on w. If M accepts w, then there's one accepting computation history, and a bunch of 'junk' strings that fail to be accepting computation history. Meanwhile if M doesn't accept w then all strings are junk. We're going to make a PDA that's going to accept the junk.

Why would we do this twisted thing — why not make a PDA that accepts just the accepting computation history? This is in fact impossible — if we did that, then we would be showing that E_{PDA} is undecidable, and it isn't.

But let's just try to do that and see what goes wrong, and how we can fix it.

Imagine we have $B_{M,w}$, with its input and its stack; and we want to check if the input is a valid computation history $C_1 \# C_2 \# C_3 \# \cdots \# C_\ell$. Imagine we want a PDA that accepts only this string (it's a checker, and it just checks that its input is a valid accepting computation history for M on w).

First we need to check, does C_1 start correctly? It can do this — C_1 can be stored in a table. And at the end, it can check there's q_{accept} in the last configuration.

But it also has to check that each configuration leads legally to the next one; and it's going to need to use its stack for that.

Let's imagine we push C_1 onto the stack and pop it off to compare to the next one; and make sure they're the same, except for where the update according to the Turing machine took place.

First, our PDA can certainly make sure the string starts correctly (we can build C_1 into it). Why can't we build the *whole* string into the PDA? Well, we don't even know whether this string exists — S does not yet know whether M accepts w, so it certainly cannot construct a machine to accept only this string by brute force (it doesn't even know whether this string exists). So you can't just build this string in — you have to actually do the work of doing the checking (you have to make it into an actual checker that just uses the rules of the Turing machine to check that this is a valid thing).

We'd like to check that each configuration legally leads to the next one. We can take C_2, push it onto the stack, and pop it off to compare with C_3 and see if everything agrees except around the head. (There's an issue of order, but we'll deal with that later.) But then we need to match C_3 with C_4. And now we're in trouble — we're already to the right of C_3, so we can't push it onto the stack because we've already read C_3 to match it with C_4. And there's no way to fix this.

So making a PDA which accepts only this string is hopeless. But you can make a PDA which only *rejects* this string (instead of only *accepting* this string).

Why is this easier? In order for this string to be valid, we need to check *every* pair — C_1 has to lead to C_2, C_2 to C_3, and so on — and all of these checks have to work. But in order for a string to be junk, we only need it to fail in *one* place.

And it's easy to do this using nondeterminism — we'll make $B_{M,w}$ nondeterministically scan across, and say e.g. I bet C_3 doesn't correctly yield C_4. It'll then push C_3 onto the stack, pop it off, and compare with C_4; and if that doesn't match up, it'll accept. The point is that it only needs to find a bug in one place to know that the string is *not* an accepting computation history — it's easier using nondeterminism to accept all the junk strings (and it'll never find a bug in the actual accepting computation history, so it'll accept only that).

The way $B_{M,w}$ works on its input x is as follows: Nondeterministically check that C_1 is not the starting configuration and C_ℓ is not an accepting one, and that each C_i does not yield C_{i+1}. If any of these cases happens, then accept. This guarantees $B_{M,w}$ accepts precisely the junk strings, so if M doesn't accept w, then every input to $B_{M,w}$ is a junk string, so its language is Σ^*.

So now (back to our construction of S) we test using R if $L(B_{M,w}) = \Sigma^*$. If yes, then M rejects w (since there are no accepting computation histories), so we reject. If no, then M accepts w (because the only thing $B_{M,w}$ won't accept is the legit accepting computation history for M on w, which means M did accept w).

There is a question here we brushed under the rug — how do we check that C_2 yields C_3 according to M's rules? We said we'd take C_2, push it onto the stack, and pop it off to match with C_3. The problem is that when we pop off C_2, it comes out backwards (because that's how stacks work). So trying to match C_2 backwards with C_3 doesn't work (we want things to line up so that we can match them — so that we can see that they're the same except around the heads).

So instead of using the simple encoding of an accepting computation history, we need to take every other configuration and write it backwards — we test for strings of the form

$$C_1 \# C_2^{\mathcal{R}} \# C_3 \# \dots$$

(we can choose whatever representation of the computation history we want). Now when we pop $C_2^{\mathcal{R}}$ off the stack, it'll come out in the forwards direction on the stack, and we can match it up with C_3 (and similarly C_1 popped off the stack is in reverse order, so it matches with $C_2^{\mathcal{R}}$). □

Remark 11.2. The language ww is not a context-free language; but interestingly the language xy where $|x| = |y|$ and $x \neq y$ *is* context-free. You can alternatively use this method here (avoiding the reversal issue) — you can guess that your issue is in the 17th place, look at the 17th place of C_2, and then go to the 17th place of C_3 and check it.

§11.4 Self-replication and the recursion theorem

We'll now move on to a new topic, self-replication and the recursion theorem. (This is relevant to Problem 6 on the homework — we'll need this method to solve it.)

Problem 6 asks us to make a Pascal program (or any other language) that prints out a copy of itself; we have to do this without cheating (i.e., without using a primitive that allows you to pull a description of the code). This is actually pretty tricky — because you're sort of faced with a conundrum. You can print out a string, but then there's all sorts of other things around it, and if you try to print out those, then the program gets even longer — it feels like you're chasing your tail.

And in fact, at first glance it seems making self-reproducing machines is hopeless — imagine a factory that produces cars. In a certain sense, intuitively speaking, the factory is going to be more complicated than the cars it produces — it has to be at least as complicated because it has all the instructions to produce a car, but it also has all the other machines too. Then if you want to make a factory that produces *factories*, you're seemingly doomed — you can't make the factory more complicated than the objects it's producing, because they're the same.

So this seems like a hopeless endeavor. But we know in the universe there are things that self-replicate, namely living things — somehow evolution has figured out a way around this problem. And it's essentially the method we'll see today, involving the recursion theorem.

Theorem 11.3 (Recursion theorem)

There exists a Turing machine SELF that prints its own description $\langle \mathsf{Self} \rangle$ (regardless of the input).

How will we do this? We'll say the same thing several different times, but here is the core of the idea that we'll use in the homework, and the essence of the solution to this problem:

Imagine we have our Turing machine Self and its tape (the input is irrelevant, so we can imagine just starting with a blank tape); what we want to have happen is that after the machine is done we have $\langle \text{SELF} \rangle$ on the tape.

We'll describe Self in two parts A and B; the control will first start in A, and then pass from A to B.

Before we do this, we'll need a lemma:

Lemma 11.4

There is a computable function $q\colon \Sigma^* \to \Sigma^*$ where for every string w, $q(w) = \langle P_w \rangle$ where P_w is a Turing machine that ignores its input, prints w, and halts.

Here w will be some string; and we want to get from w a Turing machine that prints out this string. This is so simple a thing that it seems absurdly obvious — certainly giving any string, you can make a program that just prints out that string, but moreover you can compute that program from the string. We call this q for *quote* — from w we want to get an object that evaluates to w in a sense. So P_w is a Turing machine that just prints out w.

Proof of theorem. We'll make A the machine that prints the description of B — i.e., A will be $P_{\langle B \rangle}$. So A will be the Turing machine that prints the string $\langle B \rangle$, which happens to be the description of B.

Now after we run A (this printing Turing machine which outputs the description of B), what is sitting on the tape? Well, this is just the description of B! (Because that's what A does — it prints out the description of B.)

So at this point here, once we pass control to B, now B is sitting on the tape. It looks like we're halfway done — we've got half of SELF already on the tape, and the only thing that's left is to put the other half of SELF on the tape.

One thing that might look tempting at first (but that you'll realize is absurdly impossible if you think about it) is to try to let $B = P_{\langle A \rangle}$ — because if we could do this, then we would be done (modulo things being a bit out of order). But if you try to do this, then you get into the same infinite regression we're trying to avoid — A is going to have $\langle B \rangle$ hardwired into it, so in a sense A is much bigger than B (A is like a factory that produces B). You can't have B produce A in the same way, because if B has a copy of A inside it, then it's going to be bigger than A. You can't have a copy of B inside A *and* a copy of A inside B. So we're going to need to define B in a different way.

We'll define B as a piece of code. First B is going to see $\langle B \rangle$ on the tape. And it's going to look at that, and from this it's going to *calculate* what A is — because A is the Turing machine that was constructed to print out what's on the tape, and B can compute what that Turing machine is by seeing what's on the tape. So B first uses q on the tape contents to get $P_{\langle B \rangle}$, which is A; and then it prints $\langle A \rangle$ to the tape and halts.

Here we've avoided just sticking A inside B — instead B actually figures out what A is. □

§11.5 An English interpretation

This method works in any programming language; it even works in English. So here's an English interpretation. Imagine we want to make an English sentence which directs the reader to write down a copy of the same sentence — instructions that if a reader follows them, will cause that sentence to reappear.

Before we get into the details, what does it mean for a reader to follow the instructions of a sentence?

Example 11.5

The sentence `Write 'hello world'` yields `hello world` (imagine you're a reader and you read this sentence; then you're going to write `hello world` on your paper).

This doesn't do the trick — we don't just want `hello world` to appear, we want `Write 'hello world'` to appear.

Here's another attempt:

Example 11.6

`Write this sentence` yields `Write this sentence`, but it's cheating — it's using a feature we don't want to assume.

Someone reading this would just write this sentence, which works. But in a sense this is cheating — it relies on the primitive *this*, which refers to the whole thing, and you don't necessarily have this in every programming language (a primitive that refers back to the code of the program). In fact, what the recursion theorem will do for us is *give* us this primitive — we will show that any programming language can have this self-referential primitive as a tool at our disposal. But we're not there yet.

So can we get the same effect without cheating? The answer is yes.

Example 11.7

The following instructions work:

```
Write two copies of the following, the second copy in quotes
'Write two copies of the following, the second copy in quotes'
```

All implementations of the recursion theorem work like this — we have a template (a string to be manipulated), and we have some code that manipulates the string. As the homework problem suggests, you have a string that serves as a template; and you print it twice, with a bit of extra stuff around it.

> **Remark 11.8.** A curiosity is that the notion of an active portion of the code and a template is something that evolution discovered — in DNA there's two copies of the information. One strand of the DNA just serves as a template for the other copy, the active part which acts on the template to both make a copy of the original *and* the template. Somehow, this notion of a program and template may be the only way of achieving this goal.

§11.6 Applications

This has two types of applications — one within the theory, and another real-world application.

First, you can take the recursion theorem (which gives a program that prints out itself) and enhance it a bit — you can generalize this theorem in a fairly straightforward way so that not only can you print out a copy of yourself, but you can then go on and use that copy of yourself, and compute with it (as if it were an input).

Simply getting a copy of yourself and doing something with it comes up in the real world in computer viruses — computer viruses have to get access to their own code. Some might take advantage of implementation details of the computer they're infecting (using some sort of referencing). But some computer viruses probably get access to their own code this way. So that's an example of a computer program that gets access to its own code.

But we'll now see applications within the theory. The moral of this story is that we now have a new primitive that we can use when making Turing machines — 'get a copy of my own description.' The recursion theorem will guarantee that you can actually make such a Turing machine.

For example, we can use this to give a new proof of the following theorem from earlier.

> **Theorem 11.9**
>
> The language A_{TM} is undecidable.

We proved this a while ago, using diagonalization; this took us half a lecture. Now we'll give a proof in two minutes.

Proof. Assume for contradiction that the Turing machine H decides A_{TM}. Using the recursion theorem, we construct a new Turing machine R, which works as follows on input w:

(1) Get its own description $\langle R \rangle$. (So now, by the magic of this code, we have $\langle R \rangle$ sitting on the tape — so R knows $\langle R \rangle$.)

(2) Use H to test whether R accepts w — we just run H on $\langle R, w \rangle$. (This is exactly what H knows how to deal with, so H will tell us, does R accept w?)

(3) Now H has told R whether it's going to accept w or not; and R just does the opposite.

In other words, if H says R accepts w, then you reject w; if H says R rejects w, then you accept. This contradicts the assumption that H decides A_{TM}. □

It's kind of amazing that you can do this so succinctly. Here are a few more fun applications.

Back when Prof. Sipser was an undergraduate learning how to program in C, he used to like to find really short solutions — you have to write code to do various different things, and Prof. Sipser liked to find short code (that was probably completely unintelligible to the TAs).

Imagine you try to write the shortest possible Turing machine that does what you want to do (measured in terms of the description size).

Definition 11.10. We say a Turing machine M is *minimal* if there is no Turing machine M' with a shorter description that has the same language.

Definition 11.11. Let $\text{MIN}_{\mathsf{TM}} = \{\langle M \rangle \mid M \text{ is a minimal } \mathsf{TM}\}$.

This is a language consisting of all minimal Turing machines — the shortest-description Turing machine in their class.

Theorem 11.12

The language MIN_{TM} is not Turing-recognizable.

Proof. Assume for contradiction that E enumerates MIN_{TM}. It's nice to think about this problem from the standpoint of an enumerator — we've seen that a language is Turing-recognizable if and only if there is a Turing enumerator for it. So there is a Turing machine that prints out all minimal Turing machine descriptions — we turn it on and it prints out some Turing machine descriptions, which are all guaranteed to be minimal.

We now make a new Turing machine R (which will not be an enumerator, but will use the enumerator inside it), which on input w works as follows:

(1) Get your own description $\langle R \rangle$.

(2) Run E. (Now we've fired up our enumerator, and it's going to start printing out minimal Turing machines.) Suppose that it outputs $\langle M_1 \rangle$, $\langle M_2 \rangle$, ... Now R is running E, and it's watching those outputs — E is printing out new outputs and R is biding its time watching these Turing machines come out. It waits until E prints out a Turing machine whose description is bigger than R. (There are infinitely many minimal Turing machines, so eventually some Turing machine which is bigger than R is going to get outputted.) Let this machine be $\langle M_\ell \rangle$, so that $|\langle M_\ell \rangle| > |\langle R \rangle|$.

(3) Simulate this machine M_ℓ (i.e., run M_ℓ on your input w).

So R waits until E outputs a bigger machine, and then simulates it. Why is this a problem? Now R is doing the same thing as M_ℓ; but it's smaller than M_ℓ, so M_ℓ isn't minimal. And this means E outputted something it shouldn't have — E was guaranteed to only produce minimal machines, so that's your contradiction. □

Remark 11.13. At a high level, the recursion theorem is kind of a compiler — you can give it code which assumes that its own description comes in as an input, and this compiler converts it to one that computes its own description rather than just getting it afterwards.

Not every Turing machine can do this, but you get this primitive when constructing Turing machines.

This also has an application in mathematical logic. You probably know the Gödel statements that in any system of axioms, there is some true but unprovable statement. The way this is done is by having the statement encode the idea that this statement is unprovable. But this a priori refers to itself; and there's a recursion theorem-type method of encoding this.

§12 October 19, 2023

§12.1 Midterm

The midterm is next week in Walker (on the third floor). There are sample problems posted; the rules for the midterm are on the sample problems. It'll cover everything up to last lecture (the end of the first half of the course, on computability theory). Today we'll start the second half, on complexity theory.

§12.2 Introduction

Computability theory is basically the theory of what you can and cannot do by algorithms. Meanwhile, complexity theory has to do with what you can do in a more practical sense, taking into account what resources you have. Computability theory was more or less wrapped up in the 1950s — it's not a very active area of research. But complexity theory is — in complexity theory there's a lot of unsolved problems, lots of people working on them, and lots of advances being made.

We'll kind of have to start over again — at the beginning of computability theory we set up a model (the Turing machine) we'd work with, and we're going to sort of have to do that again (to figure out which model we'll work with). That'll get sorted out today.

§12.3 A first example

Example 12.1

Consider the language $A = \{\mathtt{a}^k\mathtt{b}^k \mid k \geq 0\}$. How many steps does a Turing machine need to decide this language?

Going forwards, all the languages we'll look at are decidable — we won't have questions about whether things are decidable or not. But we want to understand how much time or space (or other resources) we need to decide the language. And here we consider the number of steps a Turing machine would need.

We can come up with a Turing machine and figure out how much time it takes; and we will do that. But beforehand, a few thoughts — the number of steps is going to depend on the input (longer inputs will take more steps). So the number of steps is a function of the input. We could analyze this input by input, but in general we simplify — we group all inputs of the same length together, and try to bound the amount of time we need for *all* inputs of length n. We'll typically reserve the letter n for representing the length of the input; and we'll talk about how much time we need for inputs of length n (looking at the worst-case over all inputs of length n).

Remark 12.2. You can also consider how much time on *average* you need for inputs of length n, but we won't do that; we'll look at worst-case complexity.

Another question is, which model do we want to use? In the computability section we had single-tape Turing machines, multitape Turing machines, and nondeterministic Turing machines. In general, these won't be the same; so we have to figure out which model to work with in order to set up the theory.

But for starters, let's stick with a deterministic one-tape Turing machine model, and ask, as a function of the input length n, how many steps do we need to decide the language?

Theorem 12.3

A one-tape Turing machine can decide $A = \{\mathtt{a}^k\mathtt{b}^k \mid k \geq 0\}$ within cn^2 steps, for some constant c (independent of n).

We'll often have our results expressed in this form — where there'll be some constant times some function of n. The typical way to refer to this is with the notation $O(n^2)$ — this is a shorthand for representing 'at most cn^2 for a constant c independent of n.' (This is called big-O notation and is described in the book; it essentially represents a hidden constant factor that we won't care about.)

Proof. Imagine we have some input `aaaaabbbbb` $\sqcup\sqcup\sqcup\cdots$. We'll call our Turing machine M. On input w:

(1) Read w and test whether $w \in \mathtt{a}^*\mathtt{b}^*$ (scanning across w and behaving like a finite automaton).

(2) Make repeated passes across the input, crossing off one `a` and one `b` on each pass (while `a`'s and `b`'s remain).

(3) Accept if we cross off the `a`'s and `b`'s completely on the *same* final pass; reject otherwise (if we have `a`'s remaining while all the `b`'s are crossed off, or vice versa).

We claim this procedure achieves our goal of deciding A within time $O(n^2)$. This is not surprising — we first scan. Then we cross off a `a` and `b` repeatedly; we keep doing this until all are crossed off. How much time does this take?

The first scan across the input takes $O(n)$ steps (we simply walk across and then return).

(We'll refer to the different numerals as different stages; so the first stage takes $O(n)$ steps.)

Then how many passes do we make? Each pass reduces the number of symbols by 2 (we cross off a `a` and a `b`), so the number of passes we make is at most $\frac{n}{2}$. And within each pass, we have to make a scan across the input, which takes $O(n)$ steps. So we have $\frac{n}{2}$ passes, each using $O(n)$ steps; this gives us a total of

$$O(n) + \frac{n}{2}O(n) = O(n^2)$$

steps. (The arithmetic of O notation is that you look at the biggest-order term — $\frac{n}{2}O(n)$ is $O(n^2)$, while the first term is $O(n)$; and $O(n) + O(n^2)$ is $O(n^2)$ since lower-order terms get absorbed within the higher-order terms.) □

Question 12.4. Could we do better? Can we replace $O(n^2)$ with a smaller function, or is this the best possible?

Crossing off one `a` and one `b` on each pass may seem like not accomplishing much; what if we crossed off ten `a`'s and ten `b`'s? But this wouldn't change the final calculation, since it'd only improve the runtime by a constant factor. Can we do something better than a constant factor?

The answer is yes — we can get down to $O(n \log n)$. Insteaad of crossing off one `a` and one `b`, we cross off *every other* `a` and every other `b`, and we keep track of whether we've seen an even or odd number of symbols. For example, if we have `aaaaabbbbb`, we cross out the first, third, and fifth `a` and `b`, keeping track of the fact that we've seen an odd number of `a`'s and `b`'s (in our finite memory — the number is 5, and we can't store *that* in our memory, but we don't need to). If the parities don't match, then we know there's a different number, so we reject.

Then we go back and do the same thing (crossing out alternating letters from the remaining) — this causes us to cross out the second `a` and `b`, and we see an even number of both `a`'s and `b`'s. On the third pass, we have one last `a` and `b` (the fourth ones), which we cross out; and both parities are odd.

If we stop at the same point and all the parities agree, then we know the number of `a`'s and `b`'s were the same at the beginning. (This can be seen by looking at the binary representation of the counts.) So we can accept.

If we modified our analysis for this algorithm, we'd be crossing off half the symbols on each pass, so the total number of passes would be $\log_2 n$ (since we cut the number of symbols down by half each time). Each pass still requires $O(n)$ steps, but then we'd have a total of $\log_2 n \cdot O(n) = O(n \log n)$.

Remark 12.5. Technically we mean $O(n \log_2 n)$, but changing $\log_2$ to $\log_c$ for any constant c (e.g. 10 or e) only changes the output by a constant factor; so when we write log's inside O, we don't need to specify the base.

Remark 12.6. What do we mean by a step? We mean a Turing machine step — every time the Turing machine makes a move using its transition function (moving the state and its head) is a step. That's why making a pass across the input requires n steps (and returning back requires another n steps).

First we saw a (deterministic, one-tape) Turing machine doing A in $O(n^2)$ steps, and then one in $O(n \log n)$ steps. Can we do better — can we get down to $O(n)$? The answer is no — with a one-tape Turing machine, we can't decide the language in $O(n)$ steps.

Remark 12.7. In a one-tape Turing machine, the only things we can do with $O(n)$ steps are regular languages — this is true even though the Turing machine can write. (This takes some work to prove.)

§12.4 Model dependence

But suppose that instead of a one-tape Turing machine we had a multitape Turing machine, and we wanted to decide A.

We've seen that we can decide A on a 1-tape deterministic Turing machine in $O(n^2)$ or $O(n \log n)$ steps, but not $O(n)$ steps.

Question 12.8. What about a 2-tape Turing machine?

Now we can do $O(n)$! On the first pass, we copy the `a`'s down to the second tape; and then as we continue along looking at the `b`'s, we can match off the `b`s' with the `a`'s that appear on the second tape. So with two tapes, $O(n)$ really is possible.

The point here is that the amount of time you need to decide a language will depend on which model we pick — even on which model of a Turing machine we pick (e.g. how many tapes we have).

This may not be surprising, but in a sense it's bad news. When we looked at computability theory, we had something very nice — the model we chose didn't matter (it didn't change the results), as long as we used a reasonable general-purpose computation model. One-tape TMs, multitape TMs, nondeterministic TMs, multidimensional TMs, even things that don't look like TMs but more like normal computers (e.g. the model used in an algorithms class, such as random access) all give the same class of decidable and recognizable languages — so the choice didn't matter. But now it does matter.

So for computability we had *model independence* — this is closely related to the Church–Turing thesis, which states that all reasonable models capture our intuition about algorithms (and therefore they're all equivalent to each other, since they all capture the same notion). In complexity we lose that — we have *model dependence*. Then we have to figure out which model we're going to work with when defining the complexity of a language.

This is bad news, since it's kind of unfortunate that the theory changes working with one-tape vs. two-tape vs. five-dimensional Turing machines. (This is not very nice in terms of naturalness, since we're not interested in studying Turing machines so much as algorithms — what's the power of algorithms? And Turing machines were just a vehicle for capturing that.)

What saves us is that even though we have model dependence, the dependence is not very big — we'll see that for a variety of different models, we can convert from one model to another without changing the

complexity by too much. So we'll focus on questions that are not going to be sensitive to which model we have — and we'll basically regain model independence for those kinds of questions.

So it'll turn out that although we have model dependence, we can bound the dependence for certain reasonable models.

§12.5 Some definitions

We'll look at this bounding in a moment, but motivated by this statement, we'll just pick one model and work with it going forward — the deterministic 1-tape Turing machine. We'll show that relative to this, all other deterministic reasonable models are pretty close.

So we'll continue with deterministic 1-tape Turing machines as our default model.

Definition 12.9. Let $t: \mathbb{N} \to \mathbb{N}$ be a function. We say that a Turing machine M *runs in time* $t(n)$ if M halts on all inputs w of length n within $t(n)$ steps.

This is what it means for a Turing machine to run in time $t(n)$ — it always uses at most this many steps before it halts. All our machines from now will be deciders — they always have to halt — and we care about how long they take before halting.

This will lead to an important definition, of what we call a *time complexity class*:

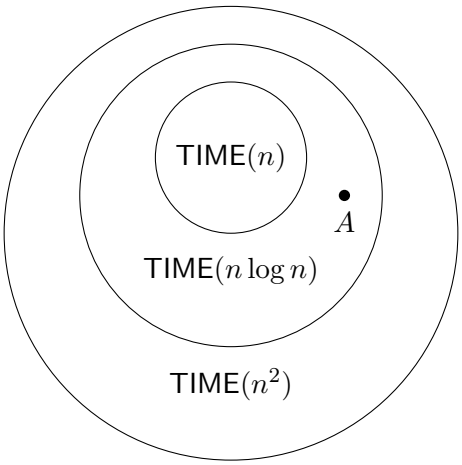
Definition 12.10. We define $\mathsf{TIME}(t(n))$ as the set of all languages that can be decided by a Turing machine that runs in time $O(t(n))$ — i.e.,

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

Like the regular languages or context-free languages, $\mathsf{TIME}(t(n))$ is a class of languages — the collection of languages we can do within $O(t(n))$ steps. Note that we're building in big-O notation into the definition of our time complexity class. There are several technical reasons for doing this, but the most important is that in our definition of time complexity, we don't care about what constant factor is used — $\mathsf{TIME}(t(n))$ and $\mathsf{TIME}(5t(n))$ will be the same class of languages.

Example 12.11

Our first theorem states that $A = \{\mathtt{a}^k\mathtt{b}^k\}$ is in $\mathsf{TIME}(n^2)$ — since there is some one-tape TM that can decide A in $O(n^2)$ steps. We also showed that $A \in \mathsf{TIME}(n \log n)$ — this is a stronger fact. And we also stated (but didn't prove) that $A \notin \mathsf{TIME}(n)$.



Each of these is a collection of languages, and A lies in $\mathsf{TIME}(n \log n)$ but not $\mathsf{TIME}(n)$. As suggested by this picture, these circles grow — increasing the allowed time can only allow you to do more things. How they grow is something we'll study — we'll show that generally, when you increase the bound, you can increase the class of languages.

§12.6 Bounded dependence

We'll now bound the differences between the different models.

> **Question 12.12.** Suppose that we can do something with a multitape Turing machine in a certain amount of time. Can we bound how long it would take to do the same with a single-tape Turing machine?

Going from multi-tape to single-tape can increase the amount of time (as in the example we saw); we'd like to bound by how much.

> **Theorem 12.13**
> For $t(n) \geq n$, if a k-tape TM decides B in $t(n)$ steps, then $B \in \mathsf{TIME}(t(n)^2)$.

We typically assume that we have $t(n) \geq n$, since otherwise we wouldn't even have enough time to read the input — and then weird things happen.

This says that if we can do something with a multi-tape TM in $t(n)$ steps, then we can do it in a single-tape TM in $t(n)^2$ steps. So going from multi-tape to single-tape at most squares the number of steps we need.

> **Remark 12.14.** Note that a multitape TM necessarily has a fixed number of tapes — its number of tapes can't grow with the input.

Proof. We'll revisit the simulation we already saw for converting multi-tape to single-tape TMs — we can use the same standard simulation.

Imagine we have a multi-tape Turing machine M, which we want to convert to a single-tape Turing machine S. To do this, we divided S's tape into blocks, where each block represented the contents of one of M's tapes — we took the tape contents for M and stored them sequentially in the tape of S.

Then to do one step of M, we needed S to scan its entire tape to see what was underneath all the virtual heads (the places with dots on them), pick up that information to see what M's heads were reading, and go back to update the tape accordingly. So each single step of M takes a whole bunch of steps for S; how many? Since S is scanning across its tape, the main question is, how big can S's tape be? It can be at most $O(t(n))$, where t is the runtime for M — this is because M could be using at most $t(n)$ space on each one of its tapes (even if it sends its heads cruising out to the right as fast as possible, the furthest they can reach is $t(n)$ — after $t(n)$ steps, the heads can only be at most $t(n)$ distance from the left end, so at most $t(n)$ stuff can be written on the tape). So each section has size at most $t(n)$, and we have only k tapes; so the total size is $O(t(n))$.

Then S has to do such a scan to simulate each one of M's steps — to do one step is $O(t(n))$, so to do $t(n)$ steps is $t(n) \cdot O(t(n)) = O(t(n)^2)$. □

Remark 12.15. Going from multi-tape to single-tape incurs a square. What if we want to go backwards — if we can do something with a single-tape TM, does this tell us we can do it with a multi-tape TM with a square root gain? No such fact is known.

You *can* show that going from single-tape to multi-tape in general gets *some* improvement, though (although not as good as a square root) — it may be $t(n)/\log t(n)$ or something like this.

§12.7 The class P

The fact that converting from multitape to single-tape involves squaring is a common cost — for reasonable deterministic computational models, converting one to the other can be done with some polynomial cost. For example, multidimensional Turing machines are a model where the tape is a section of the plane or cube instead of just a line, and the head can move in multiple directions; then converting might involve a higher-order polynomial. But converting between two deterministic models is usually polynomial cost; and we'll say two models are *polynomially equivalent* if converting from one to another only requires a polynomial increase in the number of steps.

Definition 12.16. We say two computational models are *polynomially equivalent* if time $O(t(n))$ on one model can be simulated by time $O(t(n)^k)$ on the other (for some k).

So going from one model to another only incurs a polynomial increase. All 'reasonable' deterministic models are polynomially equivalent. (We won't try to define 'reasonable.')

Remark 12.17. It may be equivalent to define polynomial equivalence as $O(n) \to O(n^k)$, but this is the way it's normally defined.

Remark 12.18. Is there one model that is the slowest or fastest among all reasonable models? Prof. Sipser doesn't think so — there's even theorems that suggest this might not be the case.

This motivates the definition of the class P (for *polynomial time*).

Definition 12.19. The class P is defined as $\mathsf{P} = \bigcup_k \mathsf{TIME}(n^k)$.

For each k, $\mathsf{TIME}(n^k)$ is the set of languages that can be decided in time $O(n^k)$; and taking the union over all k gives the languages that can be decided in polynomial time.

The class P was first proposed in the 1960s, and it turns out to be really important. The reason why it's so central in complexity theory is two-fold.

For one thing, P doesn't depend on the choice of reasonable deterministic model — if we define P in terms of one-tape Turing machines (as we did) or multidimensional TMs or random access machines, we get the same class of languages, since all these models can simulate each other with at most a polynomial cost. This might change k, but it won't take us in or out of being polynomial. So this is a recovery of some kind of model independence that we had so nicely in computability theory — P is independent for 'reasonable' deterministic models.

The second reason, that connects to the first, is that the class P roughly corresponds to the class of problems you can solve in a practical sense — these are the feasibly solvable problems. This is only an approximation to what we have in mind by 'practical' — a n^{100} time algorithm isn't very practical. But such algorithms are unusual — often if you find a polynomial-time algorithm for a problem, you can turn it into one you can really use.

When you combine these two features of P — the first being a mathematical naturalness, and the second being a kind of practical relevance — then you have a winner. It's both nice mathematically and kind of useful, and that's why P has become such an influential complexity class in complexity theory.

§12.8 An example in P

Now let's look at an example of a problem in P, to see how we start talking about these things. We'll look at languages related to computational problems; the *path* problem will be about connectivity in graphs.

Definition 12.20. We define $\text{PATH} = \{\langle G, s, t\rangle \mid G \text{ is a directed graph and has a path from } s \text{ to } t\}$.

A *directed* graph is one in which connections between nodes are arrows (they have a direction). Here s and t are nodes in our directed graph. We want to find if there's a sequence of arrows we can follow from s to t.

Theorem 12.21
$\text{PATH} \in \mathsf{P}$.

If we've taken an algorithms course, this is completely routine — there's many algorithms that test connectivity in polynomial time (e.g. depth-first search, breadth-first search). We'll use one as an illustration.

Proof. We'll define a Turing machine M on input $\langle G, s, t\rangle$. We'll use a marking algorithm, similarly to before — we start by marking the start node s; then we mark every node reachable from the start node; and so on. We keep iterating, marking nodes reachable from previously marked nodes, until we can't mark anything new. Finally, we test to see if we've marked t. So explicitly:

(1) Mark s.

(2) Repeat until we produce no new marks: mark all nodes directly pointed at by previously marked nodes.

(3) Accept if t is marked, and reject otherwise.

We can analyze this along the lines of the analyses we gave before; but when we give polynomial algorithms, typically each stage should be something you can obviously do in polynomial time, and there should be a polynomial number of stages. Here marking things directly reachable from previously marked nodes can be done just by scanning the graph; this is $O(n)$ or potentially $O(n^2)$. And then at each iteration we're marking at least one new node, so we have at most n iterations; that means the total cost is at most n^3. (We're not looking for fancy precise analyses — we can generally be pretty loose with showing that something takes polynomial time.) □

As a setup for next lecture, instead of the path problem, we can define a superficially similar-sounding problem, the *Hamiltonian path* problem:

Definition 12.22. We define HAMPATH as the set of $\langle G, s, t\rangle$ such that G is a directed graph and has a Hamiltonian path from s to t.

A *Hamiltonian path* is one that goes through every node of the graph exactly once. In a certain sense, a Hamiltonian path from s to t is the longest possible path (since it has to visit every node) — as if you're doing a newspaper route.

Question 12.23. Is $\text{HAMPATH} \in \mathsf{P}$?

The answer is not known.

§13 October 24, 2023

§13.1 Introduction

We're starting to talk about complexity theory. (This will not be on the midterm, which only covers the material on computability.)

Last lecture, we looked at various different computational models — we're doing a sort of reset to go back and figure out which models we're going to work with to measure complexity and whether it makes much of a difference. For computability the choice of model didn't matter. Here it *does* matter, but not very much — at least, up to a polynomial. This may be a lot depending on our application — the difference between solving a problem in $O(n)$ and $O(n^4)$ time has important practical implications. (In fact, n vs. $100000n$ time makes a practical difference.) But still, there are many practical questions that come up if we don't consider polynomial differences; and that's the realm we'll work with.

We'll stick with single-tape deterministic Turing machines as our base deterministic model. We use them to define $\mathsf{TIME}(t(n))$ as the set of all languages we can decide with a Turing machine that operates in at most $O(t(n))$ steps. We defined P as the union of $\mathsf{TIME}(t(n))$ over all polynomials $t(n) = n^k$. We then saw that $\textsc{Path} \in \mathsf{P}$.

Today we'll do the same for *nondeterministic* classes, where things are a bit different in important ways.

Recall that in PATH, we're given a graph and two points s and t, and we want to know if there's a path connecting s to t. The problem HAMPATH asks if there's a *Hamiltonian* path from s to t (one that goes through every node exactly once).

There's a fairly straightforward problem that shows $\textsc{Path} \in \mathsf{P}$, but for HAMPATH this isn't known. And we'll explore that today — it's possible that there *is* a polynomial-time algorithm to solve HAMPATH, but if there was one then it'd have amazing implications. For example, if it turned out HAMPATH were solvable in polynomial time, then you could factor large numbers in polynomial time (which is something we don't know how to do either). That's an amazing statement because factoring and Hamiltonian paths look like completely unrelated problems, but a fast way to solve HAMPATH immediately translates into a fast way to solve factoring. We'll develop this in the next couple of lectures.

One important feature of HAMPATH is that even though we don't know of a polynomial-time algorithm to find whether there *is* such a path, if you do through all the work of finding that there is a Hamiltonian path, then it's easy to *verify* that fact — it's easy to convince someone else of this fact (without them having to go through all the work), since you can just exhibit them the path.

So for certain problems, they may be difficult to solve, but the answer can be easy to check. That's what the following setup will be trying to capture.

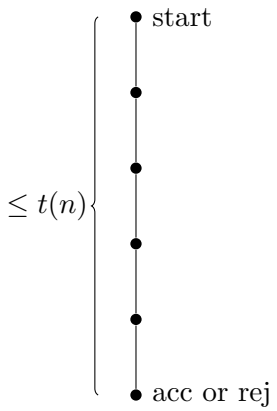
§13.2 Nondeterministic Complexity

We'll now try to set up complexity for nondeterministic machines.

> **Definition 13.1.** A nondeterministic Turing machine (NTM) runs in time $t(n)$ if for all inputs of length n, it halts within $t(n)$ steps on all branches.

For deterministic Turing machines, the machine had to take all inputs of length n and halt in $t(n)$ steps. This definition is similar, but now in a NTM there's lots of branches possible, and we require that *all* these branches have to halt within the time bound.

So for a deterministic machine M on w, we can imagine having a sequence of steps, and this sequence must have height at most $t(n)$.



For a *nondeterministic* machine, we can imagine having a tree of possibilities. We'll say a nondeterministic decider has to halt on every branch; and for a NTM to run within the time bound, *all* the branches of the tree have to halt within that bound (so every branch is going to accept or reject within the bound).

Definition 13.2. We define $\mathsf{NTIME}(t(n))$ as the set of languages B such that some (single-tape) NTM decides B and runs within $O(t(n))$ time.

Last time we talked about polynomial equivalence between various deterministic Turing machines — all reasonable deterministic models are equivalent. Once we switch to nondeterminism, we don't know if we get equivalence anymore; so we have to set up a whole new set of classes (and we don't know how exactly they correspond to the deterministic ones).

We'll stick to single-tape Turing machines as our basic model (to be definite); going from single-tape to multi-tape Turing machines might change these classes, but for different nondeterministic computational models, the change is again going to be at most polynomial. This motivates the definition of the class NP, which ignores which polynomial we have.

Definition 13.3. We define the class *nondeterministic polynomial time* as $\mathsf{NP} = \bigcup_k \mathsf{NTIME}(n^k)$.

Remark 13.4. Note that NP does *not* stand for 'not polynomial' — we don't know whether $\mathsf{P} = \mathsf{NP}$ or not (and many problems in NP can also be in P).

§13.3 Some examples

This is the same notion of nondeterminism we've been working with all along — the machine has multiple branches or threads, and accepts if and only if one of those threads ends up accepting. To develop our intuition, here's an example.

Example 13.5

We have HAMPATH $\in$ P.

Proof. Here's a nondeterministic Turing machine T — on input $\langle G, s, t\rangle$ we want to test whether there's a Hamiltonian path from s to t, and now we can use nondeterminism.

Let m be the number of nodes of G. (We reserve n for the size of the input — the number of symbols it takes down to write down the encoding of G (however we chose to encode it, e.g. a list of edges or an

adjacency matrix — for the purpose of polynomiality it doesn't matter how we chose to encode, since we can convert reasonable encodings from one to the other in polynomial time).)

Our algorithm will nondeterministically write a sequence of m nodes v_1, v_2, …, v_m in the graph. (We're not yet saying anything about what these nodes look like — there may be repetitions, and it's possible not all nodes are represented; it's just some sequence of nodes, which as we'll see soon is easy to write down nondeterministically.) The nondeterminism means we'll write down *all* different sequences of m nodes, one on each thread of the nondeterminism.

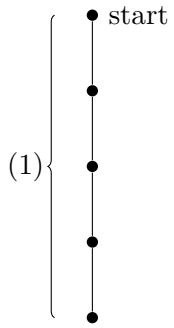
Once we have this sequence of nodes written down (in any particular thread), we now check if v_1, …, v_m is a Hamiltonian path from s to t. (We think of this sequence as actually representing the nodes we go through as we're going from s to t — as a guess on this branch of the nondeterminism. So at a high level, we're going to guess the Hamiltonian path and then check that it works — this is a sort of typical 2-step procedure.) This means we check that $s = v_1$, $t = v_m$, each (v_i, v_{i+1}) is an edge in G (so every step we take is actually following along one of the arrows in the graph), and finally that for all $i \neq j$, we have $v_i \neq v_j$ (so we have no repetitions).

Once we have this, we know that v_1, …, v_m is a Hamiltonian path (it starts at s, goes to t, is a valid path, and has no repetitions; and since we have exactly m nodes this means all are used).

> **Remark 13.6.** There's a story of a professor lecturing to his math class, and then at some point he says 'it's obvious that….' Then he steps back, starts to pace, steps out of the room; and 20 minutes later he comes back and says 'yes I was right, it is obvious.' Here Prof. Sipser was worried that we're not forbidding the case where we get from s to t, but there's some set of vertices outside that's not participating. But this can't happen, since we're only looking at sequences of m nodes.

Finally, we accept if yes (i.e., if the above check passed, so v_1, …, v_m is a Hamiltonian path), and reject if no.

We can think about this algorithm at a lower level of the functioning of the Turing machine (for illustration). In the very first stage of the algorithm, T is going to spend several steps counting up the number of nodes in the graph; this will be something like $O(n)$ if you're careful about how you do the counting, or potentially $O(n^2)$ (all we care about is that each stage is polynomial).



Then stage 2 is where the action is. Suppose our Turing machine writes down m (we may as well imagine doing this on a separate tape). Then T is going to write down every sequence of nodes using its nondeterminism — it writes down either 0 or 1, then 0 or 1, and so on. Once it's written down enough bits, that'll represent v_1. Then it writes down a #; and then it nondeterministically writes down more 0's or 1's until it has v_2; and it continues this until it has v_m.

So after the deterministic phase, we're going to start having nondeterminism in order to get v_1, and then v_2, and so on. We'll have tons of paths, and each path corresponds to a particular choice of node for v_1. (There's only one correct choice, namely $v_1 = s$, but at this stage we don't think about this — we start

nondeterministically at any node in the graph, and write that down as our first node.) By the time we have $v_1, \ldots, v_m$, we enter stage 3.

Now we're going to be doing the checking in (3), which returns to a deterministic phase; and in (4) we're simply accepting or rejecting on each one of those threads.

We might have (2) take $O(n^2)$ steps as well (maybe $O(n)$ — it doesn't really matter), and maybe (3) does too. But the point is that the whole thing will be $O(n^k)$ for some k.

That's on each thread *independently*. What we do *not* do is add up the total amount of work across all the threads. If we were going to be really implementing this in the deterministic world, we'd have to go through all the threads. But a NTM is just a theoretical model trying to capture something; and all we're doing is measuring the running time on each thread individually. □

So this is a nondeterministic polynomial-time algorithm that shows HAMPATH $\in$ NP. Why does it work? It's really dependent on the idea mentioned in the beginning — to know that a graph has a Hamiltonian path, all we need to do is find it. Checking it works is easy; finding it is the hard part. And with nondeterminism, we can find it by guessing all the paths in parallel.

Here's another example (which we won't do in very much detail).

Definition 13.7. We define COMPOSITES as the set of binary representations of composite numbers.

Theorem 13.8

We have COMPOSITES $\in$ NP.

(In fact, 23 years ago primality testing (and therefore composite testing, its complement) was shown to be in P; but this is much harder to show.)

The intuition is that if a number is composite, it's easy to check its compositeness — what we need to know in order to easily check is its factorization. (Once we have the factorization, we can just multiply to check that we get that number.)

So even though compositeness may or may not be easy to solve directly, *checking* that a number is composite is definitely easy (we can just exhibit a factor).

Proof. We'll define a Turing machine S as follows: on x, we nondeterministically write y with $1 < y < x$ as a candidate factor. We then accept if y evenly divides x (by performing division and seeing if we get 0 remainder), and reject otherwise. Each of these steps can be done in polynomial time, where n is the length of x written in binary. □

Remark 13.9. Earlier we said the encoding of the input didn't matter as long as we were reasonable (e.g. adjacency matrices, lists of edges, and so on). That's kind of true here — it doesn't matter whether we write x in binary or decimal (you can convert binary to decimal using a procedure that runs in polynomial time, so it won't matter which encoding you use). However, you're *not* allowed to use unary (which would represent 10^6 using 10^6 1's) — converting a binary number to unary would take exponential time (since we'd have to write down x 1's instead of $\log x$ digits). So we have to be careful to stick with reasonable encodings, and a unary encoding is not reasonable.

§13.4 The intuition for NP

As stated earlier, the intuition for NP is that we can check the answer. But more precisely, it's that we can check a *positive* answer. We can check membership in the language easily, but we may not be able to check *non*membership.

Saying that $B \in \mathsf{P}$ means that we can *test* membership in polynomial time. Saying that $B \in \mathsf{NP}$ means that we can *verify* membership in polynomial time.

We've written it this way purposefully — you can verify membership in B, but you can't necessarily verify *non*membership.

For example, in the Hamiltonian path problem, if we've determined that the graph does have a Hamiltonian path then you can convince someone else of this by giving them the path. But suppose that you found there *is* no Hamiltonian path; how do you convince someone of that? This is much less obvious — how do you show them there's no Hamiltonian path without showing them all your work (which is an exponential amount of effort)? And in fact, whether $\overline{\text{HAMPATH}} \in \mathsf{NP}$ is also unknown.

So there's a kind of asymmetry in NP (which is characteristic of nondeterminism in general) — a language might be in NP, but its complement conceivably may not be (whereas P is closed under taking the complement, since we can simply flip the answer).

Remark 13.10. Note that we're not enumerating all the paths in our algorithm for HAMPATH — we're selecting one in each thread, and using nondeterminism to take care of that for us. Nondeterminism means by definition that we accept if there is *some* accepting path. That'll mess you up if you try taking the complement — even if we have a nondeterministic algorithm for HAMPATH that finds the path on some thread, then suppose the input has no Hamiltonian path. Then every branch will reject, but there's no single branch you can use to see that there's no Hamiltonian path.

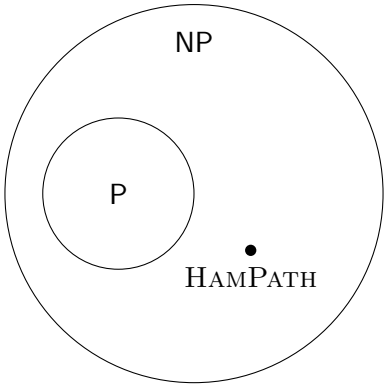
This is a bit similar to the fact that CFLs are not closed under complement — a PDA is nondeterministic, so we can't just flip the answer.

Remark 13.11. We know $\overline{\mathsf{P}} = \mathsf{P}$, since if we have a deterministic polynomial-time algorithm for B, we get one for $\overline{B}$ obtained by just flipping the answer. That doesn't work for nondeterminism because we have lots of threads, and flipping all their answers doesn't necessarily flip the answer (if we have a computation with some accepting and some rejecting threads, that's an accepting computation, but flipping the answers will keep it an accepting computation, which we don't want to do). So it doesn't seem like $\overline{\mathsf{NP}} = \mathsf{NP}$; but we don't know how to prove this.

Remark 13.12. 'Verifying membership' means there's some information you can provide (e.g. a Hamiltonian path, or a factor) that allows you to convince yourself that you're in the language; we call this a *certificate* (the extra information that allows checking). But there is a difference between HAMPATH and $\overline{\text{HAMPATH}}$ — you can verify that we *have* a Hamiltonian path (by producing it), but how can you verify that we *don't* have one? That's the distinction that we're trying to capture mathematically.

§13.5 P vs. NP

Note that $\mathsf{P} \subseteq \mathsf{NP}$: a deterministic machine is a special kind of nondeterministic machine, so anything we can do with a deterministic machine in polynomial time can also be done with a nondeterministic one.



But the question in the other direction — whether $\mathsf{P} = \mathsf{NP}$ — is a famous unsolved problem (it's been a famous open problem since the 1970s). It looks like HAMPATH is only in the outer circle, but it's possible the two circles coincide; then HamPath is in both.

But the magical thing is that if someone one day finds a polynomial time to solve HAMPATH, then we automatically get $\mathsf{P} = \mathsf{NP}$ — if HAMPATH is in P, then everything *else* sitting on the outside also gets brought along with it into P.

Remark 13.13. The Hilbert problems were posed as 23 problems to challenge mathematicians in the 20th century. The same thing was done by a committee for the 21st century, with 7 problems (called the Millenium problems); this problem was one of them.

Personally, Prof. Sipser has spent a lot of time thinking about this problem; he's not so sure it was time well spent. The question might be, how does one try to think about proving either that they're equal or not (most people believe they're not equal)? This would amount to showing that for some problem in NP, there is no polynomial-time algorithm.

Because Prof. Sipser does complexity theory and has thought about the problem, he gets lots of mail from people claiming to have solved the problem (in the old days it was physical letters, which he kept in a file called 'crazy correspondence'). When people are trying to prove $\mathsf{P} \neq \mathsf{NP}$, they often make the same mistake — they take something like HAMPATH, and somewhere they say that 'clearly the only way you can solve the problem is by looking at all the different paths' (and then they have a long complicated analysis that shows this takes a long time). The latter statement is obvious; the first is *not* obvious — there might be a non-obvious way of solving the problem *without* searching through all the paths. The compositeness problem is such a problem (one that looks like it requires searching but actually doesn't); we'll see another example soon.

As an example, we'll consider the problem of testing whether a string is in a context-free language.

Definition 13.14. We define $A_{\mathsf{CFG}} = \{\langle G, w\rangle \mid \mathsf{CFG}\ G \text{ generates } w\}$.

Earlier we showed that A_{CFG} is decidable — on $\langle G, w\rangle$ we converted G to Chomsky normal form, and then tried all derivations of length $2k-1$ (where $k = |w|$); we accepted if any of them yielded w, and rejected otherwise. This showed decidability; but it also shows that $A_{\mathsf{CFG}} \in \mathsf{NP}$ (since instead of trying all derivations sequentially on a deterministic machine, we can nondeterministically write down all possible derivations, and accept if any one of them yields w). So this also shows $A_{\mathsf{CFG}} \in \mathsf{NP}$. (On different threads we test each possible derivation.)

You might think that this is a problem in NP that seems to require trying all these different possibilities; so by the same bogus reasoning mentioned earlier ('it seems clear the only thing you can do is trying all the possibilities') we might think this problem is not in P. But in fact it is in P. This uses something called

dynamic programming; this is one of the most basic nontrivial polynomial-time algorithms, so probably something we should know.

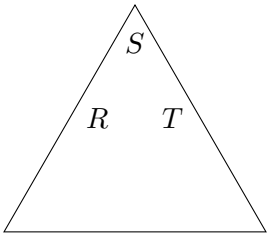
Theorem 13.15

We have $A_{\mathsf{CFG}} \in \mathsf{P}$.

Proof. As before, we first convert G to Chomsky normal form (where all rules look like $A \to BC$ or $A \to b$ — this conversion is straightforward and can be done in polynomial time; it's described in the textbook). This suggests a natural recursion — imagine we have the start variable S, and we want to know whether there's some sequence of substitutions that can give us w.

Now if there's a rule $S \to RT$, it's natural to do the substitution for S into R and T; then we want to know whether R can generate some piece of w and T can generate the rest of w.

So we're trying to test whether we can get from S to w. We'll recursively say, let's try all possible substitutions which come out of S into two variables. If one of them is RT, then we can try dividing w into all possible parts x and y, and seeing whether we can get from R to x and T to y. And we do this recursively.



The problem is if we do the analysis, this doesn't give us a polynomial-time algorithm — the recursion could have $\log n$ levels, each of which has n choices (all ways to break up w); and we get a $n^{\log n}$ algorithm, which is not polynomial.

But we'll instead do the trick of dynamic programming (which is actually a very simple idea) — it's just 'doing recursion and not being stupid.' More specifically, it's recursion plus memory. When we do a recursion like this, the general problem we're trying to solve is, within the grammar, if I have a substring of w and a variable, can I get from the variable to that substring? If we look naively at the tree we go through when we do the recursion, it may have superpolynomial size. But we can prune this down by making sure we don't repeat work that we've already done. The number of possible subproblems is at most the number of variables times the number of substrings of w; the number of variables is certainly polynomial in n, and the number of consecutive substrings of w is n^2. So there's only n^2 different subproblems; and so if you do recursion, pruning away things you've already done, you end up with a polynomial-time algorithm. (Algorithms folks call this *memoization*.) □

The point is that there's examples of nontrivial polynomial algorithms, which can solve things that look like they require searching but can be solved in a more clever way. We *think* that's not the case for HamPath, but how do you show that?

§13.6 The satisfiability problem

Finally, we'll introduce one more problem, which will play an important role.

Definition 13.16. A *Boolean formula* consists of Boolean variables (which take values in `True` and `False`, or sometimes `1` and `0`) and the Boolean operations And, Or, and Not (denoted $\wedge$, $\vee$, and $\bar{\cdot}$).

Example 13.17

For example, $(\overline{a} \vee b) \wedge (a \wedge \overline{b})$ is a Boolean formula.

Definition 13.18. A Boolean formula is *satisfiable* if there is some assignment to its variables which makes the formula evaluate to `True`.

(An assignment is just a way of setting the values of the variables.)

Example 13.19

The formula $(\overline{a} \vee b) \wedge (a \wedge \overline{b})$ is satisfiable — we can take $a =$ `True` and $b =$ `True`. (Meanwhile setting $a =$ `True` and $b =$ `False` would *not* be a satisfying assignment, as then $\overline{a} \vee b =$ `False`.)

Definition 13.20. We define $\text{SAT} = \{\langle \varphi \rangle \mid \varphi \text{ is satisfiable}\}$.

We have $\text{SAT} \in \mathsf{NP}$ — if a formula is satisfiable, there's a certificate that allows you to verify the satisfiability, namely the satisfying assignment. Once you have a satisfying assignment, you can easily check that it works; but finding one in the first place is hard.

§14 October 31, 2023

In the last few lectures, we defined deterministic and nondeterministic time complexity classes — last time we focused on nondeterministic complexity, and defined $\mathsf{NTIME}(t(n))$ as the languages we can decide with a single-tape nondeterministic Turing machine where on every input of length n, all threads halt in $O(t(n))$ steps. We defined $\mathsf{NP} = \bigcup_k \mathsf{NTIME}(n^k)$, as $\mathsf{P} = \bigcup_k \mathsf{TIME}(n^k)$. The intuition is that P are the languages where we can quickly *test* that we're in the language, and NP are the languages where we can *verify* that we're in the language — there's some short 'proof' or 'certificate' that we can quickly check.

Remark 14.1. This is similar to the notion of a certificate in real life — if you want to go to a concert, you get a ticket, and that's the proof that you paid that allows someone to quickly check. A certificate here plays a similar role — it allows a verifier to confirm that the input is in the language.

We gave several examples — HamPath, Composites, and especially Sat.

Definition 14.2. In *satisfiability problem*, we're given a Boolean formula (with variables, and AND, OR, and NOT), and we want to know whether it's possible to assign values to the variables to make the formula true — as a language,

$$\text{Sat} = \{\langle \varphi \rangle \mid \varphi \text{ is a satisfiable Boolean formula}\}.$$

We saw that $\text{Sat} \in \mathsf{NP}$. This will play an important role for us — we'll show that if we have a way of solving Sat in polynomial time, then we *also* get a way of solving HamPath in polynomial time. This is remarkable because the two problems look totally unrelated — why should a fast solution for Sat yield one for HamPath? That's the technology we'll develop today.

It'll be convenient to only look at a special case of Sat:

Definition 14.3. A formula is in CNF if it looks like

$$\varphi = (a \vee b \vee c) \wedge (\overline{a} \vee b \vee d) \wedge \cdots \wedge (\overline{x} \vee \overline{y} \vee z),$$

where the formula is an AND of *clauses*, and each *clause* is an OR of literals (variables or negated variables).

So CNF is an and of ors of variables and negated variables. It'll be convenient to work with such formulas — when you have a satisfying assignment to a CNF formula, that satisfying assignment has a particularly nice meaning when we look at the formula. If you have an assignment which makes the formula true, that means it makes every clause true (because the clauses are ANDed together); and to make a clause true, you have to make at least one of the literals true. That significance of a truth assignment in a CNF is going to be useful — a satisfying assignment in a CNF makes *at least one* literal true in *each* of the clauses.

Definition 14.4. A formula is in 3-CNF if there are exactly 3 literals in each clause.

Definition 14.5. The language 3Sat is $\{\langle\varphi\rangle \mid \varphi \text{ is a satisfiable 3-CNF}\}$.

We see that 3SAT is a special case of SAT — we're testing satsifiability, but only on formulas of a particular form. But it turns out that testing if a 3CNF formula is satisfiable will be as hard as testing whether a general formula is satisfiable (we'll work through that soon).

§14.1 Reductions

Now we're going to start to develop a method for connecting complexities with each other. To illustrate this most clearly, we'll introduce yet another computational problem which will be a good example to start with, the clique problem.

Definition 14.6. We define CLIQUE $= \{\langle G, k\rangle \mid G \text{ is a graph with a } k\text{-clique}\}$. (All graphs are undirected unless otherwise specified; a k-clique is k nodes with each pair connected by an edge.)

So a k-clique is k nodes with all possible edges present. If we have a graph G which might have lots of nodes and edges, but it has within it some such 5 vertices sitting inside it, then $\langle G, 5\rangle \in$ CLIQUE.

First, we have CLIQUE $\in$ NP. To see why, we should think, how do we verify membership in CLIQUE (if I give you a graph and the number k, what's a certificate that the graph has a 5-clique)? I can simply tell you which nodes are the 5-clique — if it's nodes 1, 3, 5, 12, 13, then you can just go look at the graph and see that in fact, yes, these nodes are all in the graph and all connected to each other by edges.

So CLIQUE $\in$ NP. Is it in P? If I give you a graph with 500 nodes and I want to know, does it have a clique with 100 nodes, how are you going to test that deterministically in a reasonable amount of time? That's not known.

But what we will see is the following:

Claim 14.7 — If CLIQUE is in P, then so is 3SAT.

The method for this will be defining a new kind of reduction, called polynomial time reductions.

Definition 14.8. We say A is polynomial time reducible to B, written as $A \leq_{\mathsf{p}} B$, if $A \leq_{\mathsf{m}} B$ and the reduction function f is computable in polynomial time.

This is a type of mapping reducibility (and this term is sometimes called 'polynomial time mapping reducible').

Recall that in mapping reduction, we had A and B, and a reduction function f that took things in A to things in B, and things not in A to things not in B. For this to be a polynomial time reduction, f has to be computable in polynomial time.

Theorem 14.9

If $A \leq_{\mathsf{p}} B$ and $B \in \mathsf{P}$, then $A \in \mathsf{P}$.

This should look familiar — we did something very similar for decidability and recognizability. Before, we saw that if $A \leq_{\mathsf{m}} B$ and B is decidable, then A is decidable — that was the whole basis for our method of proving problems undecidable. Here, we're just looking at a time complexity version of that same argument — if A is polynomial-time reducible to B and B is easy (solvable in polynomial time), then A is also easy.

Proof. Let a TM R decide B in polynomial time (since we're assuming $B \in \mathsf{P}$). We'll construct a TM S that decides A in polynomial time. The idea is that we have a way of quickly testing whether we're in B or not, and we can quickly convert A-problems to B-problems, then we can test whether we're in A quickly by mapping over to the B-world.

On input w:

(1) Compute $f(w)$.

(2) Use R to test whether $f(w) \in B$.

(3) Accept if yes, and reject if no.

(1) can be done in polynomial time by assumption. Now $f(w)$ may be a larger string than w, but it's still polynomial in $n = \texttt{len}(w)$ (since computing f takes polynomial time). So running R on $f(w)$ still takes polynomial time in n, which means the entire algorithm takes time polynomial in n. □

Remark 14.10. Isn't our definition of polynomial-time reducibility weaker than we want it to be? Before, we had two different kinds of reducibility — general reducibility and mapping reducibility. In the end, the reason mapping reducibility was useful was that it didn't allow us to flip the answer — that was important because it allowed us to distinguish between showing a language is not Turing-recognizable from its *complement* not being Turing-recognizable.

Here we're doing the same thing as the mapping reducibility case, of not allowing flipping. This is because we want to distinguish a problem being in NP from its complement being in NP — we don't want complementation to occur inside the reduction, because this would blur the distinction between NP and coNP.

What we'll actually show is that $3\textsc{Sat} \leq_{\mathsf{p}} \textsc{Clique}$ — then by this theorem, if $\textsc{Clique} \in \mathsf{P}$ then $3\textsc{Sat} \in \mathsf{P}$. The point is that we can convert $3\textsc{Sat}$ problems to $\textsc{Clique}$ problems.

Theorem 14.11

We have $3\textsc{Sat} \leq_{\mathsf{p}} \textsc{Clique}$.

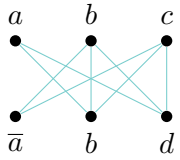
Proof. We need to give a reduction f that takes 3-CNFs to things of the form $\langle G, k \rangle$, with the property that when the 3-CNF is satisfiable, G has a k-clique, and vice versa. So our formula should be satisfiable if and only if G has a k-clique.

The actual construction itself is pretty simple, and it's useful to understand how it works (this is the pattern that defines what we'll be doing in more complicated versions soon). As an example, suppose

$$\varphi = (a \vee b \vee c) \wedge (\overline{a} \vee b \vee d) \wedge () \wedge \cdots \wedge ().$$

Then f will convert this into $\langle G, k \rangle$. First we'll define the nodes of G — each *occurrence* of a literal will become a node in G. So each clause will have three associated nodes, one for each literal.

Next, we need to define the edges. We'll connect every node to every other node, with two exceptions. First, we never connect two nodes in the same group (where a group is the three variables associated to each clause, since each clause has three literals). And we also don't draw edges between contradictory nodes — each node comes from a literal, so we can imagine labelling the nodes with the literals they came from (e.g. a, b, c, $\overline{a}$, b, d, $\dots$). We say two nodes are contradictory if they come from labels that are negations of one another — so a and $\overline{a}$ are contradictory, and we don't draw an edge between them.



Finally, we need to define k; we take k to be the number of clauses.

When we give such a reduction, we need to say a few words about why it works — here's a correctness proof which shows that φ is satisfiable if and only if G has a k-clique. First, we need to show that if φ is satisfiable, then G does have a k-clique.

If φ is satisfiable, then there is some satisfying assignment. (We might take a long time to find it, but we're not talking about finding it, just about the assignment existing.) This means it has to make true at least one literal in each clause. So we can go through each clause and pick a true literal in that clause. (If there are multiple true literals in a clause, we just pick one.) And we then look at the associated nodes in G; these form a k-clique — we can see that they're all connected to each other. (They don't violate either reason to not include an edge — we only pick one node from each group, and we can't have chosen contradictory nodes because in a satisfying assignment, we can't have a and $\overline{a}$ both be true.) So we have shown that if φ is satisfiable, G has a k-clique.

For the backwards direction, we want to show that if G has a k-clique, then the formula is satisfiable. If G has a k-clique, we first can't have two nodes in the same group, because they're not connected; so at most one node in each group is in our clique. But we have k nodes, and there are k groups, so we have exactly one node in each group.

Now we look at the corresponding literals to those nodes, and we make those literals true (either by assigning the variable `T`, or if the associated node was a complementary variable, then we assign the variable `F`, which makes the literal true). This never forces us to make a variable both true and false, because we don't connect contradictory nodes. So then out of the k-clique, we can construct an assignment that makes (at least) one true literal in every clause; so this is a satisfying assignment. □

Remark 14.12. The value of 3 isn't important here, but in other proofs it sometimes is.

Remark 14.13. Usually, it's pretty clear why the construction is correct; but we should generally say a few words about why it does work.

§14.2 NP-completeness

So we now have a way of converting 3SAT problems into CLIQUE problems — if we can solve CLIQUE quickly, then we can solve 3SAT quickly. This motivates the definition of NP-completeness.

Definition 14.14. A language B is NP-complete if $B \in \mathsf{NP}$, and for every $A \in \mathsf{NP}$, we have $A \leq_{\mathsf{p}} B$.

So the picture is that B lies in NP, and *every* other language in NP is polynomial time reducible to B. This in itself is an interesting thing to show — so far, we've only given one example of a reduction between two languages. Later we're going to show a *general* reduction that works for *every* language in NP.

One ramification of this definition of NP-complete is that if B is NP-complete and $B \in \mathsf{P}$, then $\mathsf{P} = \mathsf{NP}$. So if we imagine P sitting inside NP and B happens to itself fall inside P, then since everything in NP is reducible to it, they *all* end up falling into P.

So that sounds like a fairly strong property. But in fact, we will show tomorrow the following theorem:

Theorem 14.15 (Cook–Levin 1971)
SAT is NP-complete.

So SAT actually *is* an example of one of these problems to which everything else in NP is reducible.

Corollary 14.16
3SAT is NP-complete.

Why do we care about NP-completeness? If B is NP-complete and $B \in \mathsf{P}$, then $\mathsf{P} = \mathsf{NP}$. We can turn this on its head and say that if B is NP-complete and $\mathsf{P} \neq \mathsf{NP}$ (either because we've proven this, or because we're assuming it), then that implies $B \notin \mathsf{P}$.

Definition 14.17. We say a problem B is *intractible* if $B \notin \mathsf{P}$.

(This is because if there's no polynomial time algorithm to solve a problem, then it's hard.)

Most people believe that $\mathsf{P} \neq \mathsf{NP}$, though we don't know how to prove it. So showing that a problem is NP-complete is strong evidence that the problem doesn't have a polynomial time solution (because if it did, then that would show $\mathsf{P} = \mathsf{NP}$). So B being NP-complete is considered as evidence that B is intractible. (Until we one day prove $\mathsf{P} \neq \mathsf{NP}$ this isn't a *proof* of intractibility, but we believe it.)

Typically, you'll have a new problem and you'd like to find a polynomial time algorithm to solve it, and you're struggling with it because there's a lot of different configurations (you can easily show it's in NP, but not in P). You might try to show that the problem is NP-complete, because that means you can pretty much give up on trying to find your polynomial time algorithm — because if you did, you could give up your robotics research and just publish it. So often, you have two alternatives — you try to work on a polynomial time algorithm for your problem, and to show it's NP-complete.

How do we show a problem is NP-complete? The theorem we showed earlier, that $3\text{SAT} \leq_{\mathsf{p}} \text{CLIQUE}$, shows that CLIQUE is *also* NP-complete. And that'll be our general strategy — we'll give a reduction from some known NP-complete problem (typically 3SAT) to our problem, and this shows that our problem is also NP-complete. That's because we'll show 3SAT is NP-complete, meaning everything in NP can be reduced to it; and we can compose the two reductions to reduce everything to CLIQUE.

§14.3 Another reduction

We'll now do one more, slightly harder, reduction from 3SAT. (There are two on our problem set.)

Recall that

$$\textsc{HamPath} = \{\langle G, s, t\rangle \mid \text{directed graph } G \text{ has a Hamiltonian path from } s \text{ to } t\}.$$

(A Hamiltonian path is one that uses every node exactly once.) We observed earlier that $\textsc{HamPath} \in \mathsf{NP}$.

Theorem 14.18

$3\textsc{Sat} \leq_{\mathsf{p}} \textsc{HamPath}$.

Proof. We need to show how to convert a formula φ to $\langle G, s, t\rangle$. Here we'll try to give a sense of the strategy of how we show these things.

The idea behind these reductions from 3Sat is that φ is a formula which may or may not be satisfiable; and we're trying to build a structure which encodes the satisfiability of φ. The encoding inside this structure often involves substructures that encode the variables and the clauses — because the variables have very clear interpretations in the formula (they either get set to `T` or `F`). So inside G we're going to have a structure that can go in one of two ways; that simulates whether we set the variable to `T` or `F`. We call these substructures *gadgets*.

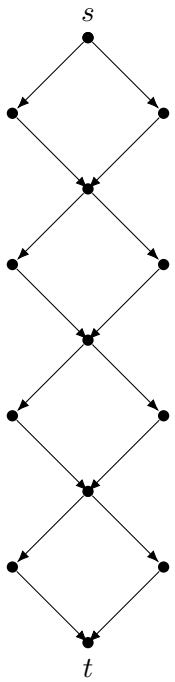
So the strategy is for G to 'simulate' φ. It's almost as if we're programming in a very weird programming language, the programming language of HamPath. So G will have variable 'gadgets' (substructures of the graph which can simulate variables), as well as clause gadgets. The variable gadgets will have a bimodal form — in a Hamiltonian path, the gadget can go one way or another. The clause gadgets will be in there to force the variable gadgets to actually be a satisfying assignment — it forces its clause to have a true literal. So we're simulating the different components of our formula inside the graph.

How do we make this work? Suppose our formula looks like

$$\varphi = (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee x_5) \wedge \cdots \wedge ().$$

We want to convert this into $\langle G, s, t\rangle$.

First, here's G. We'll have a node s (the starting node). And then we have two differet ways we can go, which merge back in; and then we have another of these, and so on. So we have a bunch of diamonds, with all edges directed down.



We also have intermediate nodes going horizontally across in each diamond. And here, there'll be a bunch of arrows that go from left to right, and also right to left (through the same nodes — so we have a path that goes in both directions).

Each of these diamond structures is a variable gadget — the first is for x_1, the second is for x_2, and so on, up to x_k.

First, what do Hamiltonian paths from s to t look like? We need to get from s to t, and we have to hit every node along the way exactly once. We can start by going down to the left. We can then try going down again, but then there's no way to go back and hit the nodes on the same level. So we have to go across, hitting all the nodes on the rightwards edges, and then go down. So we have this sort of zigzag in order to hit every node in the x_1-gadget. And we can do that for each one of the gadgets, which means there certainly is a Hamiltonian path.

We're not done, because we want a Hamiltonian path to exist *only* when the formula is satisfiable.

But let's look at this again. We can zigzag through the x_1-gadget, but instead of going left, left-to-right, and then left again, we can also go the other way — a 'zagzig' instead of a zigzag. So we have a choice, and that's the essence of doing a variable gadget — when we're looking at trying to find the Hamiltonian path or whatever structure the problem is about, we have a choice of making the path happen in two ways, for each of these variables. So this simulates a truth value for x_1.

But now we want to enforce the clause property, that every clause has at least one true literal. For that, we're going to add in the clause gadgets, which are just single nodes plus a few edges.

So for each clause, we add in a new node. The idea is that a zigzag will correspond to `T`, and a zagzig to `F`.

Now in clause C_1, if x_1 is true, that satisfies C_1. Our Hamiltonian path is also going to have to visit each of these nodes. We'll give ourselves the option to, as we're going across from left to right, instead make a detour — in the left-to-right segment, we're allowed to go to C_1 from the first node and come back to the node immediately to its right. Then if we're doing a zigzag, we have the option of taking a detour to visit C_1. But if we're doing a zagzig, then we *don't* have that option — we hit the return edge before the outbound edge, so we can't.

And we do that for every literal in C_1 — we give an opportunity to visit C_1, but only in the correct direction. (And each of these detours is separated from the other detours, so that we don't have them interfering with one another.) So if we have $\overline{x_2}$ in C_2, then we have an outbound edge to C_2 and a return on the *other* side, so that when we're taking a zagzig we can visit C_2. □

§15 November 2, 2023

§15.1 Review

We've defined P and NP (we don't know if they're equal). And we defined a connection between problems in terms of polynomial time reducibility (which lets us link time complexities — if $A \leq_{\mathsf{P}} B$ and B is easy (i.e., in P), then so is A). This lets us define the notion of NP-completeness — these are the problems in NP to which everything else is polynomial time reducible. If we can solve one of these in polynomial time, then we can solve *all* NP problems in polynomial time.

Today we'll show that SAT is NP-complete. And SAT itself is polynomial-time reducible to 3SAT (in conjuctive normal form, with 3 literals per clause) — we'll probably show this in recitation. Last time we showed that $3\text{SAT} \leq_{\mathsf{P}}$ CLIQUE and HAMPATH.

Definition 15.1. A language B is NP-complete if $B \in \mathsf{NP}$ (*membership*) and for all $A \in \mathsf{NP}$ we have $A \leq_{\mathsf{P}} B$ (*hardness*).

The second part alone is said as 'B is NP-hard' — it's possible that everything in NP reduces to B even if B isn't NP-complete.

§15.2 Cook–Levin theorem

Theorem 15.2

SAT is NP-complete.

To prove this, we've already seen that SAT $\in$ NP. So we have to show that for all $A \in$ NP we have $A \leq_{\mathsf{p}}$ SAT.

Last time, we showed polynomial time reductions from 3SAT to CLIQUE and HAMPATH. Here we're showing that *all* problems in NP are reducible to 3SAT — so it's a statement about infinitely many problems, and we really need infinitely many reductions, one for each problem. (We'll construct these reductions only using the fact that our language is in NP.)

Let A be some language in NP, so it is decided by a nondeterministic (single-tape) Turing machine N in time n^k. (Technically there should be a constant factor, but we suppress it for ease of notation.)

We have to give a reduction f to show that $A \leq_{\mathsf{p}}$ SAT, where f should be mapping strings to formulas (it's reducing A, which is a collection of strings, to an element of SAT — technically SAT is also a collection of strings, but conceptually we should think in terms of formulas). So we need to map each w to some formula φ_w, such that $w \in A$ if and only if $\varphi_w \in$ SAT.

At some very high level, what's going on here is not really very surprising — when you think about Boolean formulas, these are expressions built out of the basic Boolean operations (AND, OR, and negations). And we already know that you can simulate a computer with digital logic — that's how we build computers. That's in a sense all that's going on here — we're simulating the NTM for A with a formula built out of Boolean logic. The formula we're going to get is going to express, in Boolean logic, the fact that the decider for A accepts w — that's what it'll be designed to do.

In order to do this, it'll be convenient to define a bit of terminology — since we're going to employ a method introduced before, basically involving computation histories.

Definition 15.3. A *tableau for N on w* is an accepting computation history for N on w on some accepting thread of N on w, writen as a table.

A tableau is really just a computation history for a nondeterminstic Turing machine on some thread; for convenience when thinking about it, that computation history is written as a table, where the rows of the table represent the different steps (configurations) of the machine on that accepting thread.

C_1	q_0	w_1	w_2	$\dots$	w_n	$\sqcup$	$\sqcup$	$\dots$
C_2								
$\vdots$								
C_{n^k}								

We assume that the machine only goes for n^k steps, so we only have C_{n^k} configurations. And how big are the rows? Even if the machine sends its head out all the way, it can get at most n^k steps across. So each row only has to have length n^k; so we have a $n^k \times n^k$ table, and the bottom row should be accepting.

The first row will look like what we're used to (padded with blanks); and then we'll fire up N and fill out the tableau with the history of some accepting thread. (Of course if N doesn't accept, then there's no tableau.)

Now here's how we construct φ_w — from w (and N), we need to build up this formula φ_w. We'll informally think of φ_w as expressing something — it says that a tableau for N on w exists, or in other words that N accepts w.

We're trying to reduce A to SAT, and we have a candidate member w of A. We're producing a formula φ_w which should be satisfiable when $w \in A$ — i.e., when the machine accepts w. And this formula is just going to work by simulating the machine.

We'll do this in four parts — we'll have

$$\varphi_w = \varphi_{\text{cell}} \wedge \varphi_{\text{start}} \wedge \varphi_{\text{move}} \wedge \varphi_{\text{accept}},$$

sort of corresponding to what we've seen before when you're trying to check a computation history — it starts right, each step of the way is right, and it ends right (i.e., accepts).

Let's zoom in on some cell in our tableau. You can think of the tableau at the beginning as a blank form. And what the formula is going to say is, can I fill in symbols into that form in a way which is a legal tableau (i.e., a legal sequence of steps that the machine could follow)? So we'll imagine starting with a blank slate, and express what are the symbols that go into those cells.

To do this, for each cell, we'll have a bunch of variables that define what that cell contains. The variables of this formula will be $x_{ij\sigma}$ for $1 \le i, j \le n^k$ and $\sigma \in \Gamma \cup Q$. What does this mean? Here i and j are the coordinates of the cells — each pair (i, j) corresponds to some cell in our tableau, or blank form of the tableau. We have a bunch of variables associated to that cell, which we can think of as lights (since they're `T` and `F`) — there's a light for every possible symbol that could go inside this cell. So there's a light for every possible element of the tape alphabet — if `a` is a tape alphabet symbol, then there's some `a`-light. And if we fill in the cell with `a`, then that light will go on. There'll be another light with q_0 signifying that q_0 is what goes into that cell; and another with $\sqcup$; and so on. So for each possible contents of the cell — i.e., either a tape alphabet symbol or a state — we have a variable.

So then our variables $x_{ij\sigma}$ are for each possible tableau symbol and each possible cell — a setting of these variables corresponds to turning some lights on, which corresponds to putting symbols into the cells.

(This proof almost starts to do itself when we see how it's set up. But it's also a bit like the construction for PCP.)

> **Remark 15.4.** We'll need to have one light on in every cell — every cell is only supposed to be containing one symbol, so we can't have two lights on, since that means we tried to cram two symbols into that cell (and we can't have zero lights on either, since that means we haven't finished filling out the form). That's the first thing we're going to put into our formula, to express exactly that — that every cell has exactly one light turned on.

§15.3 Constructing φ_{cell}

We'll make φ_{cell} say that there's exactly one symbol per cell. How do we do this? First, for some particular cell — let's say cell (i, j) — how do we say that *at least one* variable is true? We can do this explicitly — as

$$x_{ij\sigma_1} \vee x_{ij\sigma_2} \vee \cdots \vee x_{ij\sigma_\ell},$$

where $\Gamma \cup Q = \{\sigma_1, \ldots, \sigma_\ell\}$. (We have exactly ℓ possible symbols that coudl go in that cell, and we OR all the variables — this tells us at least one of these is on, so the cell has at least one symbol.)

We'll take this moment to establish some notation, or else things will get completely unreadable — we write this as $\bigvee_{\sigma \in \Gamma \cup Q} x_{ij\sigma}$. (We've likely seen $\sum$ representing a sum; this is a big OR representing an OR of many things.)

Now, this says that at least one light is turned on. We also want to say we don't have *two* lights turned on. So we can write this as

$$\bigvee_{\sigma \in \Gamma \cup Q} x_{ij\sigma} \wedge \bigwedge_{\sigma \neq \tau} \overline{(x_{ij\sigma} \wedge x_{ij\tau})}.$$

Together, these conditions say (i, j) has exactly one symbol — in order to satisfy this expression, we'll need each of the ANDed parts to be true, and the only way to do that is to have exactly one $x_{ij\sigma}$ true.

Now we want to do this for *all* the cells; we do this with a big AND over all cells, i.e.,

$$\bigwedge_{1\le i,j\le n^k} \left(\bigvee_{\sigma\in\Gamma\cup Q} x_{ij\sigma} \wedge \bigwedge_{\sigma\neq\tau} \overline{(x_{ij\sigma} \wedge x_{ij\tau})} \right).$$

So once we've satisfied φ_{cell}, going forwards we can imagine that we have a *candidate* for a tableau — because we have a table that's populated with symbols. Of course there's more to say, to confirm these symbols conform to a legitimate tableau; but at least we're making progress.

§15.4 Constructing φ_{start}

Now we want to add an additional part of our expression φ (which we're building up in pieces), namely φ_{start}. This is going to say tha tthe symbols in the very top row correspond to a starting configuration for N on w — i.e., these symbols should look like q_0, w_1, …, w_n, and then $\sqcup$'s. So this'll be

$$\varphi_{\text{start}} = x_{11q_0} \wedge x_{12w_1} \wedge x_{13w_2} \wedge \cdots \wedge x_{1(n+1)w_n} \wedge x_{1(n+2)\sqcup} \wedge \cdots \wedge x_{1n^k\sqcup}$$

The first term says that q_0 is the symbol in the $(1, 1)$ spot; the second says that w_1 is in the next spot; and so on. So if we assign the variables in such a way to make all of these true, then the tableau in the very first row is the starting configuration for the machine.

§15.5 Constructing φ_{accept}

We're going to ask, does the bottom row have an accepting state sitting inside it (i.e., is it an accepting configuration)? There's a minor detail — if the machine decides to accept early (before n^k steps have gone), then for simplicity we'll assume that once the machine enters an accepting configuration, all future configurations remain the same. (So the machine just sits in that state, and we have pretend steps that follow where nothing changes.)

This means we can just look in the very last row, and see, is there an accepting state that appears there? We can write this as

$$\varphi_{\text{accept}} = \bigvee_{1\le j\le n^k} x_{n^k j q_{\text{accept}}}$$

(we look in the last row, j ranges over all cells in that row, and the symbol we're looking for is q_{accept} — one of these has to be turned on in order to satisfy the φ_{accept} part of φ_w).

§15.6 Constructing φ_{move}

The last part is perhaps the trickiest; we want φ_{move} to say that all steps in the tableau are valid, i.e., that everywhere in the middle, everything is good.

We're going to try to say this in a more simplified form than spelling out all the gory details, because we kind of did this before in the PCP — if the machine moves left then something happens, if it moves right then something happens, and this can get very messy. So we'll try to do it at a higher level.

Imagine we have a tableau, and inside our tableau is a 2×3 *neighborhood* of (i, j) (the second cell in the first row of the neighborhood).

We want to put in an expression into our formula that says that every one of these neighborhoods is legal. So in other words, we have a 2×3 window; and we want to say that if we move that window over the entire tableau, we don't see any violation of the TM's rules.

Why 2×3? We'll see this in the end. The point is that if every 2×3 neighborhood is 'legal' then the entire tableau is valid.

We'll have to capture what it means for a neighborhood to be 'legal.' For example, suppose we have the following:

a	q_7	b

This means the machine is on state q_7, and it's looking at a b. It'll have many possible moves; for example, let's suppose it can change the b to a c and move left, moving to state q_9.

a	q_7	b
q_9	a	c

If this is according to the specification of N (according to its transition function), then that's a legal neighborhood.

Meanwhile, something illegal might look like the following:

a	q_7	b
a	d	b

This could not happen — if we saw that in the tableau as a neighborhood appearing somewhere, then the tableau is completely broken — the state had to go either left or right, but it's done neither and just vanished. So this would be an illegal neighborhood.

What are the other legal neighborhoods that could occur? For example, we could trivially have the following:

a	b	c
a	b	c

If there's just tape symbols and nothing's changed, then that's fine; this can occur if the head is somewhere far away.

So we have a huge tableau and we focus on very small pieces of it; because the way the TM works is local, it's enough to check that everything is okay locally.

Here's a weird one — could the following configuration be legal?

a	b	c
d	b	c

It looks like maybe not — how in the world did a get changed to a d? But it's possible that sitting out here was q_{11}, invisible to the neighborhood; and q_{11} looking at an a might convert that a to a d and move left. So if there's some state that moves left and converts a to a d, then this would be a legal neighborhood.

q_0	a	b	c
	d	b	c

On the other hand, we can't have the following:

a	b	c
a	d	c

(There's no way the b could convert to a d, since there's no head nearby.)

As another example of an illegal neighborhood:

a	q_7	b
q_5	a	q_8

This can't be legal either — somehow we ended up with two states in the configuration, which would correspond to the head being in two locations.

The reason this is interesting is that if we had an even smaller neighborhood — using 2×2 neighborhoods instead of 2×3 — then it'd be possible that the left four symbols were okay (when q_7 is reading b, it's possible it moves to the left), and that the right four symbols were okay (since q_7 reading b might also change b to an a and move right — the machine is nondeterministic). This is why we need a 2×3 neighborhood — to prevent this kind of bad behavior.

Remark 15.5. What if

a	b	c
d	b	c

was sitting at the left end of the table? We're just not going to deal with the edges; there are some boundary things that are a bit different, that you also have to take into account. So you do have to change some of the logic to handle the boundaries; but if you understand the main idea, you can adjust the proof to account for that.

So now we're going to write down some Boolean logic to indicate that a given neighborhood is legal — we can write

$$\bigvee_{\begin{array}{|c|c|c|}\hline r & s & t \\ \hline x & y & z \\ \hline \end{array}\ \text{legal}} (x_{i(j-1)r} \wedge x_{ijs} \wedge x_{i(j+1)t} x_{(i+1)(j-1)x} \wedge x_{(i+1)jy} \wedge x_{(i+1)(j+1)z}).$$

So we OR this for every possible legal neighborhood — this expression says that the (i, j)-neighborhood is legal. And now we want this to be true for *all* the neighborhoods, so we AND this over all $1 \le i, j \le n^k$. And so that's φ_{move} —

$$\varphi_{\text{move}} = \bigwedge_{1 \le i,j \le n^k} \bigvee_{\begin{array}{|c|c|c|}\hline r & s & t \\ \hline x & y & z \\ \hline \end{array}\ \text{legal}} (x_{i(j-1)r} \wedge x_{ijs} \wedge x_{i(j+1)t} x_{(i+1)(j-1)x} \wedge x_{(i+1)jy} \wedge x_{(i+1)(j+1)z}).$$

Claim 15.6 — If all 2×3 neighborhoods are legal, then the whole tableau is valid.

This is proved in the book carefully; we won't do this. As one example, what if we had two states in one row spread out? Then one of them would have a q appearing from nowhere — so the neighborhood around that q would see something bad happening. (We have to take into account that the neighborhoods are moving all across the picture — importantly, we're starting out with just one q in the first row, and that single q is going to be forced to propagate as a single state all through the picture.)

§15.7 Conclusion

To conclude, $\varphi_w = \varphi_{\text{cell}} \wedge \varphi_{\text{start}} \wedge \varphi_{\text{move}} \wedge \varphi_{\text{move}}$. If φ_w is satisfiable, then there's some assignment of variables that meets all these conditions; that's got to correspond to a valid tableau, which means N accepts w. And conversely, if N accepts w then there's a tableau, which corresponds to a valid assignment of variables.

Remark 15.7. This theorem is what launched a big part of complexity theory in the early 1970s — the idea that you could capture the difficulty of an *entire* complexity class in a single problem was really important.

Why is this important? Proving something is NP-complete gives very strong evidence that a problem isn't solvable in polynomial time.

As another reason, let's say you're a theorist trying to prove $\mathsf{P} \neq \mathsf{NP}$ (this is not recommended). Then you might want to pick some problem in NP and show it's not in P — to prove that they're different, we want to find something in NP which we can show really isn't in P. Which problem should we pick? It may be that P and NP are really different, but you picked the wrong problem — the problem you chose to work on was one you didn't *think* was in P, but it was after all. For example, twenty years ago people thought primality testing sat on the outside — it was not known to be solvable in P. So you might try using number theory to show it's not in P; and that would have been a bad choice, because it is in P.

But the nice thing about NP-completeness is that if you try to show some NP-complete problem is not in P, you'll never fall into that sad situation. So this tells you what problems to think about.

This is supposed to be a polynomial time reduction, so we need to check that it really works in polynomial time.

How big is the formula? Well, first how many variables do we have? We have a bunch of variables for each cell in the tableau, and there are $n^k \cdot n^k = n^{2k}$ cells. And there's some *fixed* number of variables per cell, so the number of variables is $O(n^{2k})$.

Then each of the things we're constructing — φ_{start} and φ_{accept} are relatively small, and the bigger parts are things that operate over all the cells. But there's n^{2k} cells and a fixed amount of logic per cell. So the size of φ_w is going to be $O(n^{2k})$, where $n = |w|$ — we go from w of size n to a formula of polynomially bigger size.

It's good that our formula is not too big — otherwise this would be hopeless — but we also have to observe that writing down this expression isn't too costly. And that's because it's fairly simple to write down — we wrote it down using $\bigwedge$ and $\bigvee$ notation, but you'd have to actually expand it out — but this would just involve repeating the same expression over and over with different indices. So it's not too complicated to write down, and the reduction can be done in polynomial time.

§15.8 SAT to 3SAT

The next thing we'll do is show that we can convert a SAT problem to a 3SAT problem.

Theorem 15.8

We have $\text{SAT} \leq_{\mathsf{p}} \text{3SAT}$.

This sort of finishes the chain — so then we know that problems like CLIQUE and HAMPATH are really NP-complete.

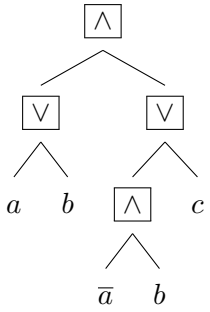
Proof. Suppose we're given a formula φ; we want to convert φ into a 3-CNF φ' preserving its satisfiability. (So far we've built an expression which is a general SAT problem. If we look at it, it's not that far from being a CNF already; but we'll do a general conversion.)

We'll do this by example — suppose that

$$\varphi = (a \vee b) \wedge ((\overline{a} \wedge b) \vee c).$$

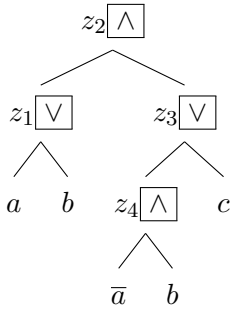
This is definitely not in CNF — the second part is very much not a legal clause (we need an AND of clauses, where each clause is an OR of literals).

So what we'll do is write this as a tree — there's a very natural tree representation of such an expression, since every AND and OR has two parts.



(Technically you can have negations in the middle, but you can push them down to the leaves using De Morgan's laws.)

We'll now make a new formula which has additional variables — one for each non-leaf node in the tree. And then we have sub-formulas checking that everything is locally correct — for each operation, we look at the value at that operation and its two leaves, and check that it's correctly assigned.



For example, let's look at z_1. We have four possibilities as a truth table for a, b, and $a \vee b = z_1$. And we put into our CNF an expression saying that these three values are set correctly. We do this for each node in the picture; and that'll tell us that the entire tree is assigned correctly.

For example, the truth table for OR says that $(\overline{a} \wedge \overline{b}) \implies \overline{z_1}$, $(a \wedge \overline{b}) \implies z_1$, and so on. These four expressions together capture the meaning of OR (if we AND them together). They're not quite in the form we need for 3CNF, but we can convert them to 3CNF using a simple trick — we can write $r \implies s$ as $\overline{r} \vee s$. So we can take $(\overline{a} \wedge \overline{b}) \implies \overline{z_1}$ and convert it to

$$\overline{\overline{a} \wedge \overline{b}} \vee \overline{z_1}.$$

And this is the same as

$$a \vee b \vee \overline{z_1}.$$

This is a clause that captures the first row of the truth table — and for each row in the truth table, we put in this clause. We AND these together and do this for each operation. And we put in one more clause saying that z_2 is true; and this gives us a CNF that exactly describes the original formula in terms of its satisfiability. □

§16 November 7, 2023

Last time, we showed the Cook–Levin theorem, that SAT is NP-complete. In some way, SAT is a bit analogous to A_{TM} from last semester — it's a 'hardest problem' within NP, just as A_{TM} is a 'hardest problem' within the recognizable languages (all recognizable languages are reducible to A_{TM}, just as we showed all NP languages are reducible to SAT).

Today we're going to shift gears — instead of talking about time complexity, we'll talk about *space complexity* (how much memory or storage an algorithm uses when it's solving a problem). We'll talk a bit about the relationships between these two notions of complexity; but first we'll make some definitions.

§16.1 Space complexity

Definition 16.1. A Turing machine *runs in space* $f(n)$ if it always halts and uses at most $f(n)$ tape cells on all inputs of length n.

You could imagine that a machine could also loop within a limited amount of space, and you could think of those machines as looping within a certain machines; but we won't say those machines are *running* in that space (because all the machines we'll be working with are deciders — this is a choice for how we set things up).

As a picture, we have a TM with its work tape; we have an input w of length n, and if all the computations of the machine (i.e., the tape cells it's using) fits within the first $f(n)$ cells, then we say it uses $f(n)$ space. We say the machine uses a cell even if it just moves its head over (without writing anything there).

Definition 16.2. A NTM runs in space $f(n)$ if it always halts and uses at most $f(n)$ cells on *each thread* independently, for all inputs of length n.

We've seen this before for time complexity — for nondeterminism, we consider all threads separately. So each thread by itself gets to use a certain amount of work tape. We don't add these up across all the threads, just as we didn't do that for time — each thread on its own gets to use our amount of tape. (But all the threads have to halt.)

Initially, everything will look analogous to what we did for time.

Definition 16.3. We define $\mathsf{SPACE}(f(n)) = \{A \mid A \text{ is decided by a TM that runs in space } O(f(n))\}$, and similarly $\mathsf{NSPACE}(f(n)) = \{A \mid A \text{ is decided by a NTM that runs in space } O(f(n))\}$.

Remark 16.4. If a TM runs in a certain amount of time, does it also run in at most that amount of space? Yes — it can't use more space than it has time. We're going to prove this as a theorem later — we'll show a relationship between the space classes and time classes. If a TM runs in a certain amount of time, we get the same bound on the amount of space it uses. If it runs in a certain amount of space it must use more time, but we can bound that too (just not by the same value).

Remark 16.5. The space includes the length of the input. For the time being, both for time and space, we're going to assume that $t(n)$ and $f(n)$ are at least n — as we've set things up right now, if a machine runs in time or space $\frac{n}{2}$ or $\sqrt{n}$, it can't even look at the whole input. Later we'll see what happens if we have a sublinear bound — at least for space — but the model is going to have to change.

Definition 16.6. We define $\mathsf{PSPACE} = \bigcup_k \mathsf{SPACE}(n^k)$ and $\mathsf{NPSPACE} = \bigcup_k \mathsf{NSPACE}(n^k)$.

§16.2 Quantified Boolean formulas

We'll now see some examples, which will turn out to be useful in another context later. The first is a generalization of the satisfiability problem — where when we write down problems, we can also write down quantifiers (e.g. *exists* and *for all*).

Definition 16.7. A *quantified Boolean formula* is a formula that looks like

$$\forall x_1 \exists x_2 \forall x_3 \dots (\exists \text{ or } \forall) x_\ell [(x_1 \vee x_2) \wedge (x_3 \vee \overline{x_2}) \wedge \cdots].$$

For simplicity, we assume the quantifiers alternate between $\forall$ and $\exists$ (we don't need this, but it'll make things simpler), and that the formula is in CNF. We'll also assume that all the variables that appear in the CNF part also are quantified by some quantifier. (So all the variables appear in the scope of one of the quantifiers — there are no unquantified variables.)

Quantifiers mean exactly what they sound like — this means for every assignment of values for x_1, there is some assignment of values for x_2, and so on. In this case, the values are going to be restricted to Boolean values, so when we say 'for all values' this means 'for both `T` and `F`.'

Example 16.8

Consider the following two formulas:

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

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

Importantly, when we have a quantified Boolean formula, we can evaluate its truth — the statement that it's making is either going to be true or false (this is because all the variables are quantified, so there are no variables floating around without a value — so we can just read out the statement, and it'll be either true or false).

For example, let's take (a). Is it true that for every x (i.e., `T` or `F`), there exists an assignment to y (possibly depending on the x-assignment, since the order of quantifiers tells us this) such that the formula is true? The answer is yes — for example, for $x = \mathtt{T}$, we are asserting that there is some value of y that makes this true. What is that value of y? We should pick $y = \mathtt{F}$ — in that case the first clause is $\mathtt{T} \vee \mathtt{F} = \mathtt{T}$, and the second is $\mathtt{F} \vee \mathtt{T} = \mathtt{T}$. But now if $x = \mathtt{F}$, we should pick $y = \mathtt{T}$. So for this expression, y is always going to need to be set to $\overline{x}$, and this will always work — so this formula (a) is true.

But for (b), does there exist y such that for all x, the expression holds? This won't be correct — because no matter what we pick for y, if we pick the same value for x, then one of the two clauses is going to be unsatisfied (as we just saw). So this second expression is false.

Question 16.9 (True quantified Boolean formula problem)**.** Given such an expression, is it true or false?

Definition 16.10. We define

$$\text{TQBF} = \{\langle\varphi\rangle \mid \varphi \text{ is a true quantified Boolean formula}\}.$$

For example, our expression in (a) is in TQBF, and the one in (b) is not.

We'll prove that this problem is solvable in PSPACE — we can solve it with a polynomial amount of space, and in fact just a *linear* amount of space.

Theorem 16.11

We have TQBF $\in$ PSPACE.

Proof. We'll give a TM M, on formula $\langle\varphi\rangle$. It'll operate by taking the formula and checking it exactly the way we would do it — it'll strip off the initial quantifier. In the case it's $\forall$, it'll plug in $x_1 = \texttt{T}$ and F, and evaluate the remaining expression in both cases (recursively); if both are true, then it'll conclude that the entire expression is true. Similarly, for an $\exists$ quantifier, it'll plug in $x_2 = \texttt{T}$ and F and evaluate everything, but in this case it'll conclude the expression is true if *either one* is true. There's a base case where there are no quantifiers and variables left. The simplest way to do this is to allow our formulas to have Ts and Fs (i.e., constants) in place of some of the variables — solving a slightly more general problem which will be helpful to us in the recursion.

If $\varphi = \forall x\, \psi$ (where ψ is the expression with the first quantifier stripped off), then we evaluate ψ with $x = \texttt{T}$ and $x = \texttt{F}$. We *accept* if both accept, and otherwise *reject*.

Similarly, if $\varphi = \exists x\, \psi$, then we do the same, but we accept if *either one* accepts, and reject if neither does. (A $\forall$ is very much like an AND, and $\exists$ is very much like an OR.)

Finally, if φ has no quantifiers, then just evaluate the expression directly (in that case there can't be any variables left, because all the variables were originally bound — and as we were setting variables, we replace them with constants; so in the end the variables are all gone and we have constants everywhere, which means we can just calculate the answer ourselves); we accept if it evaluates to T and reject if it evaluates to F.

We claim that this is a polynomial space (in fact, a linear space) algorithm. If we look at the recursion, first the *depth* of the recursion is the number of quantifiers (each time you strip off a quantifier, there's a recursive call; so for each quantifier you have another level). How much do you store at each level? You have to be slightly careful, but the only thing you really need to store is the current assignment to the variable that you're working with — for example, in the first case where $\varphi = \forall x\, \psi$, where we substitute $x = \texttt{T}$ and $x = \texttt{F}$, we have to remember what we set x to; and then we'll continue the recursion and record some other variable, and so on. So the only thing you need to store is a partial assignment to the variables. At each level of the recursion, you're just adding in a constant more amount of information (the assignment of one of the variables to T or F). And we have n levels, so the total storage required is $O(n)$.

In other words, the recursion has ℓ levels (where ℓ is the number of quantifiers), and $\ell \leq n$; so there are at most n levels. Each level stores one bit ($x = \texttt{T}$ or $x = \texttt{F}$). So the total space needed is $O(n)$. □

Remark 16.12. Why did we say the quantifiers alternate? It'll be a matter of convenience. If we didn't have alternating quantifiers, we could make them alternating by adding in dummy variables that enforce the alternation. Later we'll see that it'll be more convenient to have alternation, since we'll be working with the problem in another context. The same is true for the CNF part — if the orignal formula wasn't in CNF, you could just add more quantifiers and convert to an equivalent formula that is in CNF for the same reason you can reduce SAT to 3SAT.

Remark 16.13. Why not simplify the expression as you're recursing, to get rid of the variables that you set? You can do this — when you substitute $x_1 = \mathtt{T}$, you just go through the expression and plug in $x_1 = \mathtt{T}$, and simplify. You could do this, but then what does the base case look like? Now you've substituted everything, and you have the empty formula. Prof. Sipser thought this would be more confusing (how do you think about the empty formula?), so he thought it'd be clearer to leave constants sitting in there.

§16.3 Ladders

We'll now see a problem in NSPACE — a fun problem, somewhat like a game.

Definition 16.14. A *ladder* is a sequence of strings of the same length where consecutive strings differ in just one symbol.

(It'll matter what the allowed strings are, but we'll get there soon.)

Example 16.15

Here's a ladder of English words, that takes us from *work* to *play* — we want a sequence of English words where we change only one letter at a time, taking us from the top word to the bottom word. One such ladder is

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

We can call this an *English word ladder*; but we're not going to work with English (which is not mathematically nice to work with). Instead of English, we'll take some other specified set of legal words; for convenience, we'll use words that some DFA accepts, and we'll use that to define a language.

Definition 16.16. We define $\textsc{Ladder}_{\mathsf{DFA}}$ as the set of $\langle B, s, t\rangle$ such that there exists a ladder s to t of strings in $\mathcal{L}(B)$, where B is a DFA.

So we give a DFA B, and we want to know, is there a ladder from s to t using only strings in the language of B? So specifying the DFA determines which words are the legal words; and we're then given the start and end, and we want to know if we can get there by a ladder.

This problem is a very nice example of a problem that you can solve in *nondeterministic* polynomial space (this should also be linear).

Theorem 16.17

We have $\textsc{Ladder}_{\mathsf{DFA}} \in \mathsf{NPSPACE}$.

Proof. We'll define a Turing machine N, on input $\langle B, s, t\rangle$.

How are we going to do this? Imagine that WORK, PORK, PORT, ... are the legal words in the DFA. We're given WORK and PLAY, and we want to see, can we get from WORK to PLAY?

What's the NTM going to do? What you *don't* want to do is *write* down the entire ladder — because ladders can be really long. Here $s =$ WORK and $t =$ PLAY; the input is big enough to write down the words, but s and t can basically be length $O(n)$. And the only obvious bound we can get on how long the ladder can be might be exponential in n — because each of the positions of the symbols could potentially be any one of the symbols in the alphabet of the machine, and there could be a very long ladder.

Fortunately, nondeterminism comes to the rescue. We don't have to write down the ladder — in fact, we have to write down hardly anything. We can imagine starting with s and guessing, what's the next word? Then we just have to check two things — it's in the language, and it's different by one symbol from the previous one. Once we have that, we can throw away the first word and then guess the next word, which again has to be in the language and to differ by one letter from the previous one. And we keep doing that — guessing the next string along the ladder at each step. And if at any point we hit t, then we accept.

The algorithm we just proposed has a problem. What prevents us from looping — or doing something silly, like WORK to PORK to PORT to SORT to PORT to SORT to PORT and so on? These are all legal next steps, but we're not making progress and we'll run forever; that violates our requirement that even nondeterministic machines have to halt on every thread.

Fortunately, there's a way to fix this. If we keep track of how many steps we've gone, there's a bound on how long we could possibly go to get from s to t unless we're cycling — that's simply the number of words there are of length m, where $m = |s|$. Clearly a ladder can use at most all the words, and it can't be any longer than that unless it's getting into a cycle (which you could always eliminate). So we have to keep a count of how many steps we've taken; if we go for more than that exponential number of steps, then we just shut down the thread (it's going to reject because it didn't find a ladder within the number of steps it had allocated).

So we first let $x = s$; we reject if $s \notin \mathcal{L}(B)$. Then we nondeterministically guess a new x, and reject if it's not in $\mathcal{L}(B)$ or if it doesn't differ in one place from the previous x. We accept if $x = t$ (i.e., we got to the target). And we repeat this $|\Sigma|^m$ times, where $m = |s| = |t|$, and we reject if we haven't accepted after this number of repetitions.

This gives us a nondeterministic linear space TM — the only thing we have to keep track of is the current word we're looking at and the next word, which is no bigger than the input. □

So now we have two examples — one in polynomial space, and one in *nondeterministic* polynomial space.

§16.4 Time and space relations

Now we'll see some relations connecting time and space.

Theorem 16.18

For $t(n) \geq n$, the following two statements hold:

(1) $\mathsf{TIME}(t(n)) \subseteq \mathsf{SPACE}(t(n))$.

(2) $\mathsf{SPACE}(t(n)) \subseteq \mathsf{TIME}(2^{O(t(n))}) = \bigcup_c \mathsf{TIME}(c^{t(n)})$.

(The constant in the exponent has to run over all possible values; and we can move that constant into the base, which will make the base run over all possible values. What the constant is depends on the size of the tape alphabet of the machine — if we have a bigger tape alphabet, the machine can go for longer without repeating.)

Proof. To prove (1), we can use the same machine — a machine that runs in time $t(n)$ can only use space $t(n)$, since in each step it can use at most one more cell.

For (2), if we have a machine that runs in $t(n)$ space, and it halts (which we assumed that it does), there's a limit on how long it can run — this is the same argument we saw before when talking about LBAs, since when you have limited space, you can only go for a certain amount of time before you end up repeating; and that'll be exponential in the amount of space you have. □

That immediately tells us some consequences.

Corollary 16.19

We have $\mathsf{P} \subseteq \mathsf{PSPACE}$.

This follows immediately from (1). The moral here should be that for a given bound, space seems to be more powerful than time — because you can reuse space, but you can't reuse time. (But there's lots of things we don't know how to prove here, just like P vs. NP — we'll get there soon.)

But we can prove something even stronger than this.

Theorem 16.20

We have $\mathsf{NP} \subseteq \mathsf{PSPACE}$.

Unlike before (where we just used the same machine), here we actually have to do something (i.e., change the machine) — we have a nondeterministic polynomial time machine, and we have to show that we can use a *deterministic* polynomial space machine.

Proof. We'll show this in two parts. The first part is that $\textsc{Sat} \in \mathsf{PSPACE}$ — we can solve SAT in PSPACE, since it's a special case of TQBF (where we just put an $\exists$ for each variable). (All satisfiability is asking is, does there exist an assignment which makes the formula true?)

(We could also do this directly — if we just wanted to solve SAT in limited space, we could take our formula and try plugging in all possible assignments sequentially, and see if any of them work.)

Second, because SAT is NP-complete, for all $A \in \mathsf{NP}$ we have $A \leq_{\mathsf{p}} \textsc{Sat}$ — so A is reducible to SAT by a polynomial time reduction. And anything we can do in polynomial time can also be done in polynomial space — so in polynomial space, we can do the reduction from A to SAT, and then use our solution to SAT to solve A. □

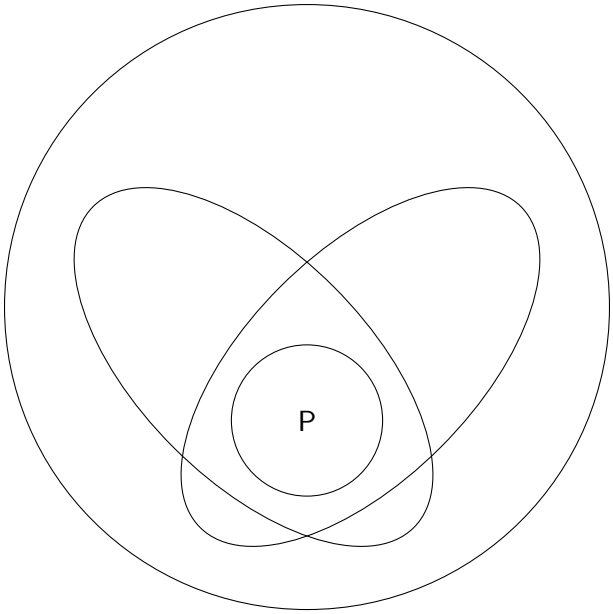
Remark 16.21. This shows the advantage of knowing that SAT is NP-complete — then what we know about SAT can be applied to *all* NP problems.

This is an opportune time to also define coNP.

Definition 16.22. We define $\mathsf{coNP} = \{\overline{A} \mid A \in \mathsf{NP}\}$.

We don't know whether $\mathsf{coNP} = \mathsf{NP}$, but we do know both are in PSPACE — because PSPACE and its complement are the same (any language in PSPACE is solvable by a deterministic algorithm, and we can always invert the answer for a deterministic algorithm to get an algorithm for the complement).

So that means we have the following picture:



Question 16.23. Is P = PSPACE?

This is also not known. In a sense, this is perhaps even more surprising than our inability to prove P ≠ NP, since PSPACE seems *way* more powerful — and we're still not able to prove there's anything in PSPACE but not in P.

Question 16.24. Is PSPACE = NPSPACE?

This we can answer, and the answer turns out to be yes — this was quite surprising (polynomial space and nondeterministic polynomial space are in fact equal).

If you think about trying to appreciate that, if we go back to our ladder problem — which we showed pretty simply is in nondeterministic polynomial space — we just claimed that this problem can also be solved in *deterministic* polynomial space. This is not obvious, but follows from the general theorem.

The idea for PSPACE = NPSPACE is one that'll turn out to be important. And what's nice for the ladder problem is that the deterministic procedure for it is in a sense general — it really captures the general case, and is more concrete (making it easier to think about).

Like a lot of things, this is a one-idea algorithm, but the idea is not easy to come up with.

Theorem 16.25

We have $\text{LADDER}_{\text{DFA}} \in$ PSPACE.

What does this mean? This means we're given a DFA defining the legal words, and s and t; and we want to know, can we get from s to t by a sequence of legal words (where the sequence itself might be very long)?

Proof. This is an example of what algorithms folks call 'divide and conquer' — we're going to take our problem, and look at the midpoint. To do this in full detail, we'll have to define how long our runs can be, so there's going to be a new parameter coming in (a bound on the length of the ladder). But for the moment, imagine we can somehow identify what point will be the middle of the ladder. We don't know what word will go in there; so instead of using our nondeterminism to get from s to t, we'll take the middle and sequentially try *every possible word*. So we'll write down AAAA, look at it and say that's not even a legal English word; and we'll do the same for AAAB, AAAC, and so on. Eventually we'll come to some actual English word, like ABLE. Now this is a *candidate* for a midpoint.

And now what we're going to do is recursively solve the problem — can we get from WORK to ABLE, and from ABLE to PLAY, in half the number of steps? If we can, then we're going to accept. (We're recursively using the same procedure — again guessing the midpoints in both cases — and we'll have to do the analysis of all this, but we'll get there.)

If we find that we can get from WORK to ABLE (by the recursion) and from ABLE to PLAY, then we know to accept. If this doesn't work out, then we go to the next word — ABLF, ABLG, and so on, until we come to another English word. This is again a candidate, and we do the same recursion. And we do that for every possible middle.

This is going to be very slow if we're measuring time, but we're not; we're measuring space, and this will turn out to give us a polynomial bound. □

§17 November 9, 2023

We'll continue our discussion of space complexity, which we defined in terms of the number of cells that a Turing machine uses on an input. We had the space classes $\mathsf{SPACE}(t(n))$ and $\mathsf{NSPACE}(t(n))$, analogous to time; and we defined PSPACE and NSPACE. We saw last time the examples $\mathrm{TQBF} \in \mathsf{PSPACE}$, and $\textsc{Ladder}_{\mathsf{DFA}} \in \mathsf{NSPACE}$.

Today we're going to prove three theorems, but all proved with the same proof — so it's really a 3-for-1 sale. We will say that proof three times as we're proving the three theorems. These three proofs, though they have the same underlying idea, are at different levels of abstraction in a way; but having in mind that they're all the same idea will help us understand what's going on.

§17.1 The ladder problem

The simplest, most concrete one, is the one we started last time:

Theorem 17.1

We have $\textsc{Ladder}_{\mathsf{DFA}} \in \mathsf{PSPACE}$.

Let's remind ourselves of what the ladder problem is. A *ladder* is a sequence of strings where consecutive strings differ in only one symbol; and we require that each string is in some language (the language associated to some finite automaton, for this example).

For example, in English we have a ladder

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

The computational problem is, given the start and end, is there a ladder between them? And to specify the legal strings, we'll give you a DFA — the legal strings are the one the DFA accepts.

We use the notation $s \to t$ to denote that there exists a ladder from s to t. We'll also add a parameter denoting the maximum length of the ladder — for example, the above ladder has length 8 (since we take 8 steps from WORK to PLAY). We write $s \xrightarrow{b} t$ to denote that there exists a ladder from s to t of length at most b. (For example $\text{WORK} \xrightarrow{8} \text{PLAY}$ and $\text{WORK} \xrightarrow{9} \text{PLAY}$).

We saw a nondeterministic algorithm that started at the first string and guessed strings one by one, making sure that each step changed a single symbol (where the only thing you store on your tape is the current string — you don't want to store the whole ladder, since that might be exponentially long).

Proof. We'll give an algorithm to test $s \xrightarrow{b} t$ for any s, t, and b (where the ladder has to be in the language of the DFA — we won't keep repeating this). This does more than we need — we not only test whether we can get from s to t, but whether we can do so within a certain number of steps.

The idea is what we sketched before (we'll use English as our stand-in for the language of the DFA) — we're given b, and we want to know, can we get from WORK to PLAY within b steps? You can come up with various strategies for doing this using depth-first search, backtracking, and so on which might superficially look like they're going to work. But you have to be careful — if you just do a brute-force search through the space of possible changes you can make, the amount of stuff that you would have to *store* is going to be based on the length of the ladder, which could be exponential. So you're going to have to do something more clever.

What we're going to do is — the algorithm is going to divide the problem in two, and solve each half recursively. So we take our midpoint — we need to get from WORK to the midpoint in $b/2$ steps, and from the midpoint to PLAY in $b/2$ steps. And we're going to do this recursively, but we have to know what the middle is to break the problem in two — and we just try every possible middle sequentially. We simply try every possible string for that middle string, by using a straightforward odometer-type sequence, e.g. AAAA, AAAB, AAAC, and so on — we go until we find a string that's an English word, e.g. ABLE. Now that we've found an English word as the candidate for the midpoint, we ask, can we get from WORK to ABLE in half the number of steps, and from ABLE to PLAY in half the number of steps? Importantly, that second half is going to reuse the space.

Within each recursion, once we have the first-level problem defined (from WORK to ABLE), we can get a second-level problem defined by trying all possible midpoints of that subproblem. Maybe one of the words we'll try is CALL; we'll see if we can get from WORK to CALL in $b/4$ steps and from CALL to ABLE in $b/4$ steps. And once we know we can do that, we see we can get from WORK to ABLE, and once we know that we proceed with ABLE to PLAY.

> **Remark 17.2.** How do we choose the intermediate word? We try them all, one after the next. This is very slow, but we're measuring space, not time — so we don't care how much time it takes. The point is that we can cycle through all the possible candidate words without using a whole lot of space — we just try string after string until we find a legitimate word, and test it out. Chances are it doesn't work; then we go on to the next word. And we keep doing this until we've tried every single word. If we ever find a way to get from the top to middle and middle to bottom, we can accept; and if after trying everything nothing has worked, then we reject.

To think about space requirements, every time I'm doing the recursion, what do I have to remember? And how deep does the recursion have to go? (Because I'm going to stack up all the intermediate subproblems — each one of those levels is going to require us to store everything.)

Imagine we have WORK and PLAY written on the input, and suppose I've now found ABLE as a word. I leave WORK and PLAY, and I go to the next place on the tape and write down WORK, ABLE, and my new bound (e.g. 1000). Then to solve that I'm going to have to try every midpoint here; I leave (WORK, ABLE, 1000) alone, and next to that I write down (WORK, CALL, 500) on my tape. (We're sort of getting ahead of ourselves; we'll write down the algorithm first and then look at the analysis.)

Here's our recursive algorithm. Given s, t, and b (with $m = |s| = |t|$):

(1) Try all strings w with $|w| = m$ sequentially.

Recursively test whether

$$s \xrightarrow{b/2} w \text{ and } w \xrightarrow{b/2} t.$$

Accept if yes; else continue with the next w.

(2) If $b = 1$, check whether we can get from s to t by changing at most one letter — accept if s and t differ in at most 1 symbol (and both are in the language of B, of course).

(3) If after we've gone through all strings w we haven't yet accepted, then reject.

There are two things we should mention. First, to test if $s \to t$ without any bound, we just have to look at the longest possible bound — we can just test $s \xrightarrow{b} t$ where b is the length of the longest possible ladder. The worst-case ladder could have at most every possible string appearing once — you never have a reason to have a repeating string in a ladder (you could just cut the intermediate portion out). So the worst-case ladder is $|\Sigma|^m$, which means we can let $b = |\Sigma|^m$. (Here Σ is the alphabet of the DFA.) If we can get $s \to t$ within this distance the we know we can get there, and if we can't then we know we can't get there ever. So this is good enough.

Now we'll analyze space. Here's one way to think about this — we have the original problem (s, t, b) written on our tape (with B also sitting somewhere, let's say on the left).

As we do a subproblem, we're going to write $(s, w_1, b/2)$ to the right, trying every possible w here. And for each one of these problems, we're going to use some portion of the rest of the tape to solve that problem. First we're going to solve this subproblem; and then we're going to solve the $(w_1, t, b/2)$ subproblem (on the same space).

To the right we'll have a subproblem for the subproblem — e.g. $(s, w_2, b/4)$, and then we'll overwrite that with $(w_2, w_1, b/4)$. And when $(w_1, t, b/2)$ becomes active then here we'll get $(w_1, w_2, b/4)$ and $(w_2, t, b/4)$.

$$
\begin{array}{lll}
s, t, b & s, w_1, b/2 & s, w_2, b/4 \\
 & w_1, t, b/2 & w_2, w_1, b/4 \\
 & & w_1, w_2, b/4 \\
 & & w_2, t, b/4
\end{array}
$$

(The w_2's for the $s, w_1, b/2$ and $w_1, t, b/2$ subproblems are not the same word.)

What's happening is that we write (s, t, b) as two problems, and solve those problems sequentially. In the first, we try every possible w_1, and see if we can get from s to w_1 in half the number of steps. We try to solve that problem. Once we've solved that problem and said yes, then we have to do the other subproblem, which is $(w_1, t, b/2)$ — and we use the same space to do it.

> **Remark 17.3.** How do we remember what words we've already tried? We try them in sequence — AAAA, AAAB, AAAC, and so on. Once we get to XXXX we know we've tried everything alphabetically before it.
>
> When we're choosing w_2, we don't care if we're repeating a word that we've used before — it's not going to matter. (We don't have to worry about remembering what we've tried and trying to exclude them here.)

First, how many levels of recursion do we have? We're cutting b in half every time, and we stop when we get to 1; so the number of levels is going to be at most $\log b = \log |\Sigma|^m = O(m) = O(n)$ (since for a given finite automaton, Σ is fixed).

How much am I remembering for each level? That's going to be $O(m) = O(n)$ again. So the total space is $O(n^2)$. □

§17.2 PSPACE and NPSPACE

Theorem 17.4 (Savitch's theorem)
We have $\mathsf{NSPACE}(f(n)) \subseteq \mathsf{SPACE}(f(n))$.

Here we assume that $f(n) \geq n$, as usual.

So eliminating nondeterminism for space complexity costs you only a square — not an exponential, like what seems to be necessary for time. And when you have only a squaring cost, that tells you that PSPACE = NPSPACE (since this just changes which polynomial).

Proof. We'll convert a NTM N to a TM M, only squaring the space used.

Consider a tableau for N on w — it's a table of the sequence of configurations the machine goes through on some accepting thread (as in the proof of Cook–Levin). Let's say N runs in space $f(n)$, so each of our configurations is of length $f(n)$ (since that's how much space N uses). And the height is going to be $d^{f(n)}$, because as observed last time, if you use more than an exponential amount of steps, you're going to have a cycle (which means some thread runs forever). So $d^{f(n)}$ is a bound on how many possible configurations there are, which is a bound on the runtime of the machine on each of the threads. (The value of d is going to depend on N, and really on the tape alphabet of N — if N has a large tape alphabet there's going to be a larger exponential here. We talked last time about how $\mathsf{SPACE}(f(n)) \subseteq \mathsf{TIME}(2^{O(f(n))})$, where the base of the exponential depends on the details of the machine.)

How is M going to work? It's going to test, can N get from its starting configuration (which we know — it looks like $q_0 w_1 \dots w_n \sqcup\sqcup\cdots$) to some accepting configuration? To make our lives easier, we'll assume there's a unique accepting configuration — when the machine enters its accepting state, we further program the machine to clean up its tape. So when it's accepting, the machine has to first erase its tape back to all $\sqcup$'s, and move its head (in the accept state) over to the leftmost cell. So there's a unique particular configuration which is our target.

This is a long way — it's an exponential number of steps. So if we try to do the obvious thing — e.g. a backtracking algorithm where we search the nondeterministic tree — it's a very tall tree (of height $d^{f(n)}$). So doing the obvious depth-first search is going to require a ton of space (an exponential amount). That's not going to be good.

So we're going to do something else. First, we're going to solve a more general problem. Instead of just saying, can we get from this configuration to that configuration, we'll solve the more general problem: given two configurations and a number of steps to get there, can I get from configuration c_i to c_j? Give configurations c_i and c_j and some b, we say that

$$c_i \xrightarrow{b} c_j$$

if N can go from c_i to c_j in at most b steps. The picture here would be that instead of START and ACCEPT, we have c_i at the top and c_j at the bottom; and instead of the particular value $d^{f(n)}$, we have b.

This should look familiar. The way we're going to solve this problem is by breaking it into two subproblems in the middle. The machine M (our deterministic simulator) is going to say, well, I can go from c_i to c_j if there is some intermediate configuration c_{mid} such that I can get from c_i to c_{mid} and c_{mid} to c_j. And I'm going to try every possible configuration — I cycle through every possible string as a candidate configuration, and do this test recursively.

Here's our algorithm to test whether $c_i \xrightarrow{b} c_j$:

(1) For $b = 1$, we want to see if we can get from c_i to c_j using the rules of N. We can simply look at the string c_i and the string c_j, and make sure that the rules of N have been obeyed (except around the head everything is the same, and around the head it gets updated correctly). So we just check whether c_i yields c_j in one step according to n's rules.

(2) If $b > 1$, for all configurations c_{mid} (of size $f(n)$ — of course we only cycle through configurations that fit within the correct amount of space), test whether

$$c_i \xrightarrow{b/2} c_{\text{mid}} \text{ and } c_{\text{mid}} \xrightarrow{b/2} c_j.$$

If yes (for both questions), then we *accept* — it means we can get from c_i to c_j in b steps. Otherwise, we go to the next c_{mid}.

(3) Reject if we have never accepted.

(This is the same procedure as for the ladder problem.)

How much space does this use? The number of levels of the recursion is at most $\log d^{f(n)} = O(f(n))$, since d is a constant. Each one is going to store $O(f(n))$ stuff — think of the configurations as 'words' in the ladder problem. We have to store the subproblem, which consists of our two configurations which are each of $f(n)$ length.

So in total, this takes $O(f(n)^2)$, and we're done. □

§17.3 PSPACE-completeness

As we mentioned last time, we don't even know if $\mathsf{P} = \mathsf{PSPACE}$, which is even more shocking than the fact that we don't know if $\mathsf{P} = \mathsf{NP}$. The situation is kind of analogous — we'll show there's an example of a PSPACE-complete problem, which will serve the same role that if we can solve it in P then we can solve *all* of PSPACE in P.

Definition 17.5. We say B is PSPACE-complete if:

(1) $B \in \mathsf{PSPACE}$.

(2) For all $A \in \mathsf{PSPACE}$, we have $A \leq_{\mathsf{p}} B$.

What kind of reduction do we use here? Now that we're talking about PSPACE, you might think, do we want to use a PSPACE reduction? But that would be a bad idea. If we use a PSPACE reduction, then *every* two problems in PSPACE would be reducible to each other. So within our reductions, we always have to use a model which is *weaker* than the model that defines the class — for example, in NP we used polynomial-time reductions, since that seems to be weaker than nondeterministic polynomial time. And it'll be convenient for this definition to use polynomial-time reductions again.

Theorem 17.6

TQBF is PSPACE-complete.

So TQBF is going to play the role of SAT, but for PSPACE.

Proof. First, we showed last class that TQBF $\in$ PSPACE (we gave an algorithm — stripping off quantifiers — that took linear space).

Now consider some $A \in \mathsf{PSPACE}$, and let A be decided by a TM M in space n^k. We're going to show that $A \leq_{\mathsf{p}}$ TQBF. To do so, we have to give a polynomial-time reduction f from A to TQBF — this means f maps strings to QBFs. We'll write $f(w) = \varphi_w$; then we want $w \in A$ if and only if φ_w is true. The way we'll do this is to make φ_w 'say' that M accepts w.

This is identical to what we did in the Cook–Levin theorem, and at the beginning it's the same plan (the whole thing is quite similar, but involves a trick).

First, here's a convenient way of looking at things: imagine a tableau for M on w. This is going to have width n^k (since that's how much space N uses) and height d^{n^k}. What the formula is supposed to say is that 'there is a tableau for M on w.' (It doesn't matter whether M is deterministic or nondeterministic — but in this case we'll assume it's deterministic.)

As before, the tableau consists of a sequence of steps that M goes through when processing w, ending up at the accepting configuration (we can again assume it is unique). This formula φ_w is going to say that this tableau exists.

Here's a first proposal, which doesn't work (but it's useful to see why) — why don't we just do the same thing as in Cook–Levin? There we produced a formula that described the statement that there is a tableau. So why not use the same construction to get a formula for whether M accepts w? But that fails for a big reason.

In the Cook–Levin construction, we introduced variables for every cell in the tableau template form, and we had extra logic to set those variables in a way to ensure that the tableau starts and ends right, and follows the rules for the machine. But critically, we had a bunch of variables for each cell, and those variables all together went into the formula.

That doesn't work here because our tableau is enormous — this gives a formula that's exponentially large, and there's no way we can produce that tableau in exponential time. So the Cook–Levin construction is too big.

But it's good to appreciate that except for that it's fine — you could essentially write the formula using the Cook–Levin construction, and you would get a SAT formula (which is a type of QBF); so this would be totally fine if it weren't too big.

So what we're going to do is solve a more general problem. We're not only going to write a formula to go from the start to the end, but we'll write a formula that says we can go from c_i to c_j within b steps. (This should sound familiar.)

So we're going to construct a formula $\varphi_{c_i,c_j,b}$ which 'says' that

$$c_i \xrightarrow{b} c_j.$$

(We're solving a more general problem — and in the end we're going to apply this general case to make the formula going from START to ACCEPT within d^{n^k} steps — that's the formula we want for the original M on w, but we're solving this more general problem so that we can build the formula recursively.)

We're going to do this in two steps. The first idea won't work; then we'll fix it.

Here's a first idea — we can try to write

$$\varphi_{c_i,c_j,b} = \exists c_{\text{mid}}[\varphi_{c_i,c_{\text{mid}},b/2} \wedge \varphi_{c_{\text{mid}},c_j,b/2}].$$

Intuitively, this says, 'there exists c_{mid} such that we can get from c_i to c_{mid} in $b/2$ steps (this is what the first formula says) and from c_{mid} to c_j in $b/2$ steps (this is what the second says).'

Note that we will use the Cook–Levin construction to decribe c_{mid} — we'll describe our configuration, which is just one row, with a bunch of variables per cell. So c_{mid} is going to consist of $x_1, \ldots, x_\ell$ which describes the configuration, using the Cook–Levin configuration. So we're saying, is there some way to set those variables (corresponding to setting some configuration) so that these work out?

And as the base case, when $b = 1$, our two configurations are next to one another; and in that case we can just put in logic the fact that we can get from one configuration to the next configuration in a single step. This can be done in the same way as Cook–Levin (since this picture is small — it's just two rows.)

This doesn't work. It cuts b in half at each level, but it doubles the formula at each level — so it's not saving you anything, and you're going to get an exponentially big formula. And in some sense it's unsurprising that this doesn't work, because it only employs exist quantifiers. (This is creating a SAT problem — it's just the Cook–Levin construction in disguise.)

To fix this, we get to the 'trick.' For $b > 1$, we're instead going to use our $\forall$ quantifier to take the AND and collapse it into a single formula. So we can write

$$\exists c_{\text{mid}} \forall (c_a, c_b) \in \{(c_i, c_{\text{mid}}), (c_{\text{mid}}, c_j)\}[\varphi_{c_a,c_b,b/2}].$$

And now the recursion is going to work out — because we have a log of the number of levels, and each introduces a constant amount of stuff. (The $\forall$ quantifier says that for all pairs of configurations c_a and c_b which are either the first pair or the second pair, for both of those settings we can get from c_a to c_b in half the number of steps. It's possible to write this properly in Boolean logic.) □

§18 November 14, 2023

Last lecture, we proved three theorems that basically had the same proof. One was the algorithm for showing LADDER is in PSPACE — it's obviously in NSPACE, but by using a divide and conquer approach you can actually get a *deterministic* polynomial-space algorithm. The same idea is behind Savitch's problem, which states that if we have *any* problem we can solve in nondeterministic space with some bound, we can solve it in *deterministic* space with the square of the bound. In particular NSPACE = PSPACE. (Of course the corresponding problem for time is a famous open problem.)

We've seen that $\mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PSPACE}$; we don't know whether any of these inclusions are strict. (It could be that $\mathsf{P} = \mathsf{NP} = \mathsf{PSPACE}$, or that $\mathsf{P} \subsetneq \mathsf{NP} = \mathsf{PSPACE}$, or so on; most people believe they're all different, but we don't know how to show this.)

We defined PSPACE-completeness and saw that TQBF is PSPACE-complete. Today we'll see another example of a PSPACE-complete problem and look at some games; and then we'll talk about a different complexity class.

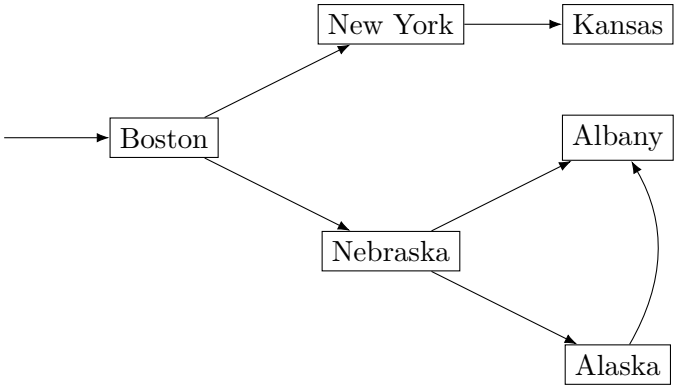
§18.1 Generalized geography

When Prof. Sipser was a kid, they didn't have video games to play, so they'd play a game called Geography. In the two-person version, you take turns naming places, where each place has to start with the same letter that the previous place ends in. So for example, if we start with *Boston*, the next place would have to start with an N, for example *New York*. Then I'd have to say a place starting with a K, e.g. *Kansas*, and then *San Francisco*, and then *Oklahoma*. Of course, you don't allow the same place to be used more than once (or else you could say *Alaska*, *Alaska*, *Alaska*, ...). At some point, you're going to run out of places, and someone's going to get stuck; the person to get stuck is the loser.

We're going to model this game in a more abstracted way, which we'll call *generalized geography*. We'll show that testing which player wins if both sides play optimally is a PSPACE-complete problem.

There aren't international competitions in playing geography; but there are well-known games (e.g. chess and checkers) for which you can show various hardness results as well (PSPACE-completeness or PSPACE-hardness — sometimes you can't show the game is in PSPACE, but you can show it's hard). Technically these games are played on a finite board, so you can't talk about complexity (you could test the winner using a very large lookup table); so you generalize to an $n \times n$ board. These $n \times n$ versions of checkers, chess, and Go are all known to be PSPACE-hard. Their proofs are more complicated; generalized geography captures the idea but is much simpler.

We can model the game of Geography as a graph. We'll assume that we have a designated starting node (e.g. *Boston*); then we ca model the game as a graph, where we draw an edge Boston → New York because that's a legal next move.



Once we have the graph written down, we can erase the labels; then it becomes a problem of picking nodes to form a simple path through the graph. We'll generalize the game to arbitrary graphs.

At some point you'll have used up too many nodes in the graph, so the game can't go on forever; and the first person to get stuck is the loser. So one side or the other is going to have a *winning strategy* — a forced win under optimal play.

Question 18.1 (Generalized geography)**.** We have two players, Players 1 and 2. They take turns picking nodes extending a simple path in the directed graph G, starting at a designated node a. If you get stuck (i.e., can't move), then you lose.

Definition 18.2. We define $\text{GG} = \{\langle G, a\rangle \mid \text{Player 1 has a winning strategy}\}$.

Our goal is to show that determining, for any graph and starting node, whether Player 1 has a winning strategy is PSPACE-complete.

§18.2 The formula game

Before that, we'll need to make a little connection between games and languages. The way we'll do that is by defining a game, which we'll call the *formula game*.

Suppose we're given a quantified Boolean formula φ of the form

$$\varphi = \exists x_1 \forall x_2 \exists x_3 \cdots \exists / \forall x_m [(\cdots) \wedge \cdots \wedge (\cdots)].$$

We define a game played on this formula, which works as follows. We have two players, who'll be named $\exists$ and $\forall$. These players are going to take turns picking values for the variables (where a value is either `T` or `F`). They take turns picking values, starting at the beginning of the formula — $\exists$ is going to pick variables bound by a $\exists$ quantifier, and $\forall$ is going to pick values bound by the $\forall$ quantifier. So in this game, $\exists$ would pick a value for x_1, then $\forall$ would pick a value for x_2, and so on. (They alterate taking turns, since we assumed for simplicity that we have alternation.)

After they've finished picking all these values, they've created an assignment. The game is now over, and we have to see who won. To do so, we pick the assignment that the two players together selected, and plug it in. If the formula is satisfied, we say $\exists$ won; if it's not satisfied, we say $\forall$ won.

So they have an oppositional nature — $\exists$ is trying to satisfy the formula, and $\forall$ is trying to falsify it. But they're together picking the values for the variables (at odds with each other).

Question 18.3. For a given formula, who wins (if both play optimally)?

Example 18.4

Let's consider $\forall x \exists y[(x \vee y) \wedge (\overline{x} \vee \overline{y})]$.

In this game $\forall$ goes first and gets to pick x (they're trying to make $(x \vee y) \wedge (\overline{x} \vee \overline{y})$ false, and $\exists$ is trying to make it true). Let's imagine $\forall$ makes $x = \mathtt{T}$. Now it's $\exists$'s turn to satisfy this formula.

Now $\exists$ can pick the value opposite to what $\forall$ played; that'll satisfy both the clauses of the formula, and will make our expression true.

This looks like a potentially complicated computational question (to decide whether $\exists$ or $\forall$ has a winning strategy if both play optimally). But it's almost immediate that $\exists$ wins exactly when the quantified Boolean formula is true. You can just see why that's the case by saying what it means for $\exists$ to have a winning strategy (i.e., a forced win) — there is some value $\exists$ can give to x_1 such that no matter what $\forall$ gives to x_2, there is some way for $\exists$ to reply with x_3 such that no matter what $\forall$ plays for x_4 and so on, the expression is true. That's exactly the semantics of the original quantified Boolean formula.

Claim 18.5 — The player $\exists$ has a winning strategy in the formula game for φ exactly when φ is true.

This follows immediately from the semantics of what it means to have a winning strategy, and the meaning of $\exists$ and $\forall$.

§18.3 Back to geography

Taking this as a starting point, we are going to now show how to prove that generalized geography is PSPACE-complete, by a reduction from TQBF — we're going to think of TQBF as a game, and that TQBF game is going to be modelled inside a specially constructed generalized geography game.

So given a formula, we're going to construct an instance of generalized geography where playing the geography game will in effect correspond to playing the formula game.

Theorem 18.6

Generalized geography is PSPACE-complete.

Proof. First we need to show that GG $\in$ PSPACE. This can be done by a routine recursive algorithm, very similar to the one we gave for TQBF itself. (We won't write out the details.)

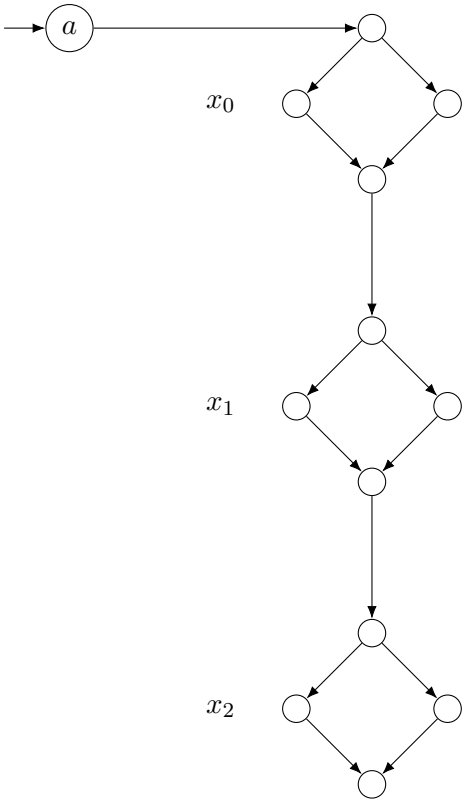
Next, we're going to show that TQBF $\leq_{\mathsf{p}}$ GG. The idea is that given some quantified Boolean formula φ, we'll construct G and a where play on G simulates the formula game.

Now we'll do the construction — here's how we construct $\langle G, a \rangle$ from φ. We'll give it by example — say

$$\varphi = \exists x_1 \forall x_2 \exists x_3 \cdots \exists x_m[(x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee \overline{x_2} \vee x_4) \wedge \cdots \wedge ()],$$

where we call the clauses $c_1, c_2, \ldots, c_k$.

Then G will look as follows:



We'll have a sequence of diamond structures, faintly reminiscent of the proof of HAMPATH being NP-complete. Each of these diamond structures corresponds to a variable.

We'll have Player 1 simulating $\exists$ and Player 2 simulating $\forall$.

At the beginning, $\exists$ starts at a, so $\forall$ is forced to go to the top node of x_1. Now $\exists$ has a choice — it can go either left or right (and either way $\forall$ is going to have to go to the bottom node) — so the only variability is whether $\exists$ goes left or right.

In our minds, we're going to think of going left as setting x_1 to `T`, and going right to setting x_1 to `F`. (That's in our mind; we'll use that later.)

Now once we have gotten down to this point, it's $\exists$ who picks the first node of x_2; and now $\forall$ gets to make a choice, of whether to go left or right. This is how we capture the idea that in the formula game, the players are taking turns assiging values for the variables (in order of the quantifiers). Here that's mirrored in the geography game we're building — we've arranged things so that the $\exists$ player gets to choose x_1, and the $\forall$ player gets to choose x_2, and so on.

(We can always adjust in case we start with $\forall$, by adding an extra step at the start; similarly if we had two $\exists$ quantifiers in a row, then we could contract the edge between the two gadgets, which is designed just to shift the initiative.)

Once we've gotten down to the bottom, all of the variables have been assigned to `T` or `F`. That corresponds to the end of the formula game, but here we need to build in a check of who won. Here's how we'll do that.

Once these variables have all been assigned, we turn to the part inside the brackets — the CNF part.

The way to think about this is that now $\forall$ is going to try to assert that this CNF formula is false under this assignment, and $\exists$ is going to try to assert that it's true. One of those is going to be incorrect. But we are going to build that into the game as follows.

Suppose $\exists$ has just picked a value for x_m. Now it's $\forall$'s turn. And now there's going to be a larger fanout — we're going to have a bunch of nodes down here, with one for every clause in the CNF part. The meaning

of this is that a CNF is *not* satisfied if and only if some clause is false; so if ∀ is trying to assert that the expression is false, then by picking e.g. c_1 they're asserting it's false because c_1 is false.

Now it's ∃'s turn, and they're going to say, 'I disagree, c_1 is in fact true.' And so we want ∃ to get to pick which one of the literals is true. So for each of these clause nodes, we'll have edges out to each of its literals. The point is that ∀ gets to pick the allegedly untrue clause, and ∃ gets to contradict that by picking the true literal (according to that assignment) in the clause. So one of these two players is lying; now we need to figure out who.

The way we'll do this is by having each one of these nodes connect back into the diamond — if we have an x_1 node it connects into the left side of the x_1 diamond, while if we have an $\overline{x_2}$ node it connects into the right side of the x_2 diamond.

Suppose that the ∀ player was actually lying. So it selected c_1, but c_1 is actually true in the assignment; and now ∃ picks the true literal in the clause, say x_1. Now x_1 is connected to the left side. If in fact x_1 was true in the earlier part of the play, then this node has already been used; so there's no place for ∀ to go, and ∀ is going to lose. (This move is forced — once ∃ picks x_1, the only place for ∀ to go is that node; and it's already been used because earlier we assigned x_1 to be `T`.) On the other hand, suppose the opposite is the case, so ∀ is telling the truth. Then c_1 is unsatisfied, so all of its literals are false. So no matter what ∃ does, it's going to be picking a false literal — if it picked x_1, then this being false would mean that play has gone down the false direction, so this node is still available and ∀ can move there. But now it's ∃'s turn; the only place it can go is the bottom of the diamond, and that's already been used. So ∃ is the one who's stranded and loses.

So if ∃ is lying then they're going to lose; if ∀ is lying then they're going to lose. And that's the construction; we've shown that play on this graph really simulates the formula game. □

> **Remark 18.7.** We could eliminate the last row by having c_1 go directly to the diamonds, on the opposite side. We just do it this way for clarity.

§18.4 Log-space complexity

PSPACE is a very large complexity class — it includes P, NP, and lots of complicated languages (that might use a ton of time).

Now we'll look at much more restricted complexity classes, inside P — the ones associated with only using a logarithmic amount of space.

In order to define this, we'll need to introduce a new model on which to base our definitions — because as we've described earlier, in our standard model you can't really have less than n space, because you won't even be able to read the entire input. (The mere act of reading the input uses up space n, and we're going to want to talk about algorithms that use only $\log n$ space.) To fix this, we'll introduce a slight variant of the model we've been using so far, which is not going to count the space of the input. That model is going to be a 2-*tape* TM with *read-only input*.

> **Definition 18.8.** A 2-*tape Turing machine with read-only input* has an input tape, which is read-only, and a *work tape*, which is read-write.

We're only going to count the cells used on the work tape towards the space complexity.

> **Definition 18.9.** A 2-tape TM of this form runs in $f(n)$ space if it only uses $f(n)$ cells on the *work tape*.

Remark 18.10. You can think about this as corresponding to the 'real-world' case — you can imagine that the input is presented on a CD-ROM (where you can't write, but can read); you can think of the CD as having a very large amount of information, much bigger than what you can store in the workspace of your machine. So you're not going to be counting the information in the CD.

A reasonably updated version of this is that you can think of the input as the entire internet. That's certainly going to be bigger than your workspace (you can't download the entire internet onto your workspace); and you can't change random things on the internet, only things local to your own machine.

So you can think of the input tape as being the internet, and the work tape as being the local space on your machine.

We're going to consider what happens when $f(n) = O(\log n)$ — this is a very natural place to pick for us because it gives us exactly enough memory to be able to have *pointers* (indices) into the input, since each index in the input can be represented as a binary number with $O(\log n)$ bits. So with an $O(\log n)$-sized work tape, you can have a constant number of pointers into the input, which lets you store at least where you are (you can also have other things, such as counters).

Definition 18.11. We define $\mathsf{L} = \mathsf{SPACE}(\log n)$ and $\mathsf{NL} = \mathsf{NSPACE}(\log n)$.

§18.5 Some examples

Example 18.12

Consider the language $A = \{ww^{\mathcal{R}} \mid w \in \Sigma^*\}$. We have $A \in \mathsf{L}$.

Imagine we have *abaa aaba* on our input tape. To check that the input is of the correct form, the old-fashioned way we would do this (if we could write on the tape) is by crossing off corresponding places. But now we can't write on the input anymore (you can think of it as being really long and read-only).

But being able to maintain a few pointers into the input allows you to keep track effectively of where you are on the two sides; so you have pointers keeping track of where you are on both sides, and then you can zigzag back and forth and make sure the corresponding places agree.

Example 18.13

Consider the path problem

$$\textsc{Path} = \{\langle G, s, t\rangle \mid \text{directed graph } G \text{ has a path } s \text{ to } t\}.$$

We have $\textsc{Path} \in \mathsf{NL}$.

Here's an algorithm: imagine we have our graph, s, and t, and we want to know if there's a path from s to t. Now we only have logarithmic space, but we do have nondeterminism; we want to make a nondeterministic machine that accepts only when there's a path s to t. The idea is to try to guess the path, node by node, keeping track of which node it's *on* using its work tape. We're not going to guess the entire path up front, since you can't write that down in the work tape. But we can write down the nodes one by one, only keeping track of the current node — we start at s, and if we ever arrive at t then we accept. (Of course some branches will go in the wrong direction and fail, but that's fine.)

One thing to be careful of is that we need every branch to halt. There might be some path which ahs a loop in it, and there'll be some branches in that case which might potentially go forever (because you keep guessing around and around). There's a trick we've seen before to make sure we don't go on forever —

we keep a count of how many nodes we've seen on the current path we're guessing (how many steps we've guessed so far). If that count exceeds m (the number of nodes in the graph), then we might as well reject on that branch — because that branch is wasting its time (if we've gone through more than m nodes and haven't hit t yet, then you've repeated nodes).

On input $\langle G, s, t\rangle$:

(1) Nondeterministically guess nodes, starting at s.

(2) *Accept* if we hit t.

(3) Keep a count of how many nodes we've gone through, and *reject* if it exceeds m (the number of nodes in G).

Remark 18.14. How do we keep a counter? We have log space, and within log space you can count up to n (because we have $\log n$ bits).

§18.6 L and NL

Question 18.15. Is PATH $\in$ L?

We had Savitch's theorem for converting nondeterministic space to deterministic space, at the cost of a squaring. Could we use that here?

First you have to check that Savitch's theorem works at this level, and it does — we proved it for space bounds at least n, but it actually works for space bounds down to $\log n$ (with the same proof). So Savitch's theorem does work. But that's not good enough — that would show that nondeterministic log space is contained in deterministic $\log^2$ space, which is different (or at least, not obviously the same) from deterministic log space.

So Savitch's problem doesn't imply PATH $\in$ L; and we don't know whether that's true or not (it's a big open problem). Imagine you have a huge graph (e.g. a map of the United States), and a small number of pointers (you and your friends walking around with cellphones). If you're nondeterministic it's easy; but if you're deterministic how do you coordinate a search through the graph? (DFS or standard searching algorithms use more than $\log n$ space, as far as we know.)

This suggests the general problem of L vs. NL.

Question 18.16 (Open problem)**.** Is L = NL?

We will prove next lecture a surprising fact about NL — so surprises do happen sometimes. Before we do that, we'll prove a few basic facts about L and NL.

Theorem 18.17

We have L $\subseteq$ P.

We actually already kind of proved this — in a slightly different model, we proved that anything we can do in space $f(n)$ we can do in time exponential in $f(n)$, and exponential in $O(\log n)$ is polynomial. So the machine running in log space is also guaranteed to run in polynomial time — we don't even have to change the machine.

Proof. What's happening is that you have to count the number of different configurations of the machine, and observe that it's polynomial. A configuration with a two-tape machine is slightly different. For a given input w of length n, how long can the machine run? It's at most the number of different configurations,

which is the number of different possibilities for the tape, state, and the two head positions. The dominating thing here is the possibilities for the tape, which is $2^{O(\log n)} = n^k$ (for some k).

So a deterministic log-space TM runs in polynomial time. □

Here's a slightly stronger fact.

Theorem 18.18

We have NL ⊆ P.

For this we're going to have to do some work — because if we have a nondetermistic machine we can't just say it runs in polynomial time and so we're done, since we need a *deterministic* machine for P.

Proof. Let $A \in$ NL be decided by a NTM N in $O(\log n)$ space. We'll now give a polynomial time algorithm for A.

To do that, we'll need a definition.

Definition 18.19. The *computation graph* for N on w is a graph with nodes given by the configurations of N on w, and edges given by $c_i \to c_j$ if N can go from c_i to c_j (in one step of the machine).

A *configuration* consists of the work tape contents, the two head locations, and the state. So c_i and c_j are two snapshots of the machine (fixing w) — with work tape contents, head locations, and state; and you can now inspect these two configurations to see if we can get from one to the other in one step (by just looking at c_i and seeing how it updates).

Now here's our polynomial time algorithm, on w:

(1) Construct the computation graph for N on w (writing down all the configurations and checking edges).

(2) Test whether there is a path from c_{start} to c_{acc}.

(3) Accept if yes, and reject if no.

We have to confirm that the computation graph has only a polynomial number of nodes in it — and that's because the number of configurations is polynomial, as we just showed. So you can write down all the possible configurations of the machine, and put down the edges connecting configurations that correspond to legal steps of the machine. Then the machine ends up accepting the input exactly when there's a path from the start configuration to the accept configuration. □

§19 November 16, 2023

§19.1 Review

Recall that we've been looking at space complexity. (This will be the last lecture focused explicitly on space complexity, though it'll still be part of our discussion going forwards.) To review, we discussed PSPACE, and we showed that the TQBF problem is PSPACE-complete (everything is polynomial-time reducible to TQBF). We also showed generalized geography is PSPACE-complete, making a connection between quantifiers and games (which is a useful perspective mathematicians often use — to think of quantifiers in terms of games).

Then we shifted gears and looked at the smaller complexity classes L (log space) and NL (nondeterministic log space). We saw $\{ww^{\mathcal{R}} \mid w \in \Sigma^*\} \in$ L and PATH ∈ NL. As mentioned, it's not known whether L = NL (they could collapse down). The situation is in some ways analogous to P vs. NP — we have complete problems for NL, and solving any complete problem in NL would collapse the entire class down.

To review, we had to introduce a different model in order to talk about space complexity where our bound is less than n (otherwise just reading the input would cost n space). We separated off the input into a read-only tape, where we don't charge for accessing it — it's not part of the cost of the memory of the machine. Separately, we have a work tape, where we will charge for usage; looking at $\log n$-bounded classes, the work tape should be $O(\log n)$ in size.

That's how we've set things up; our goal in the short term is to review the following theorem.

Theorem 19.1

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

We showed earlier that $\mathsf{L} \subseteq \mathsf{P}$; that was immediate, since a log-space machine only has polynomially many configurations so has to run in polynomial time. The same reasoning trivially shows $\mathsf{NL} \subseteq \mathsf{NP}$, but we want something stronger.

Proof. Suppose we have a language $A \in \mathsf{NL}$ decided by a NTM N running in log space. We'll make a *deterministic* Turing machine M. We don't care how much space it uses, but we care that it runs in polynomial time — so we want to convert nondeterministic log space to deterministic polynomial time.

Suppose M has input w. How is it going to work? It has to simulate N; each of the *threads* of N can run in polynomial time, so if we did DFS on the non-determinstic threads of N, we could get an exponentially large tree, and then we'd have problems.

So we have to do something more clever than a brute-force DFS of the nondeterminstic tree of N on w. Instead, what we do is we write down the *configuration graph* of N on w. (In some ways, what we're doing is just dynamic programming; but it's clearer to think about it directly.)

First, we write down the configuration graph of N on w. We'll review what that is.

For a NL machine, we have to think back to our notion of a configuration — a snapshot of the machine at a given point in time. We're not going to include the input in our configuration, since we want to count the number of configurations relative to a particular input (there's many inputs, and we don't want to include that).

After we've turned our machine on, it's going to be in some state, and the heads are going to be in some positions (which we need to record); and we also need to record the contents of the *work* tape. The input will be in a different category — it's part of what we mean when we say a configuration for N on w.

And the *configuration graph* is what you get by taking all the configurations as nodes. This seems big, but we only have polynomially many different configurations for N on w — the number of states is a constant factor, the number of head positions is $O(n) \cdot O(\log n)$, and the number of possibilities for the work tape is $|\Gamma|^{O(\log n)}$, which is polynomial in n.

So we can write the configuration graph for N on w in polynomial time; it's going to take a lot of space, but we're not measuring the space for this algorithm. Call this graph G.

And then we test if G has a path from the starting configuration c_{start} to c_{acc} (the nodes correspond to configurations, and we want to test if there's a path from the start configuration to the accept configuration).

(It's convenient to think about the machine as cleaning up its tape, erasing it, and moving its head to the leftmost position before accepting; so there's a single accepting configuration to think about.)

There's a path from c_{start} to c_{acc} if and only if N accepts w — if N accepts w then there's a seqence of moves it makes taking us from one to the other, corresponding to a path in the configuration graph.

Finally, we accept if yes, and reject if no.

Note that writing down the configuration graph is polynomial time because we have $\mathsf{poly}(n)$ configurations; and testing whether G has a path is also polynomial time, because we proved earlier that $\textsc{Path} \in \mathsf{P}$. And we're done. □

Thinking about this proof, we realize that PATH is playing a special role — we're using the fact that PATH is solvable in some way, to show that a problem in general is solvable in some way. So we're kind of reducing a general problem to the PATH problem. That sort of motivates why the PATH problem is complete for NL (we have to set up the definitions for this, but that's where we're going).

§19.2 Log space reductions

Now we'll set up NL-completeness. These definitions will look very familiar, but there's a bit of a wrinkle.

Definition 19.2. We say B is NL-complete if:

(1) $B \in \mathsf{NL}$.

(2) For all $A \in \mathsf{NL}$, we have $A \leq B$.

As usual, we want B to be in NL, and we want all languages in NL to be reducible to B. But the wrinkle comes in what type of reduction we want to use.

As a first guess, we might try using polynomial-time reductions — that's served us well so far. But that's not going to be good. The reason is that as we just proved, we have $\mathsf{NL} \subseteq \mathsf{P}$. So any NL problem is polynomial-time, which means if A and B are both in NL, then they're both automatically in P. And in the homework, we essentially showed that any two polynomial-time algorithms are reducible to each other — if we use a type of reduction more powerful than the underlying class, then the reduction can just solve the problem of membership in A, and target $f(w)$ to be either in B or not in B according to what it's determined about whether w is in A. So using polynomial-time reducibility, we'd conclude that every problem in NL would be NL-complete; this is not interesting.

To make this interesting, we need to choose a type of reduction weaker than the class. For that we'll need to define a new type of reduction, deterministic log space reductions. (We used polynomial time reductions to capture NP-completeness; similarly we'll use log space reductions to capture NL-completeness.)

Definition 19.3. We say A is *log space reducible* to B, written $A \leq_{\mathsf{L}} B$, if the reduction function is computable in log space.

Polynomial time reductions were just mapping reductions where the reduction was computable in polynomial time; here we look at reductions computable in log space. But we have to define what that means — you have to do some work here. When we talked about a log space decider, we didn't want to count the input — the input is sort of offline or separate. Now when we have a function operating in log space, we don't want to charge for the input *or* the output, because both of these can be large.

For example, you might have a log-space computable function which is just the identity — but the input is length n and the output is length n (while the work tape is essentially nothing). If you charged for the space to write the output you'd have problems; so we need to remove *both* the input and output from the cost.

To do that, we define a new type of object.

Definition 19.4. We say a function $f\colon \Sigma^* \to \Sigma^*$ is *log space computable* if it can be computed by a log space transducer.

A *log space transducer* is yet another model. It'll have three tapes — an input tape, a work tape, and an output tape. The output is write-only (it's like a printer). The input is read-only. And the work tape is read-write, as before. Only the work tape has to be $O(\log n)$ in size; we're not going to charge for either the input or the output.

You turn the machine on with its input, and it starts to put stuff on its work tape, and symbols start to come out on the output tape. It's slightly more convenient to think of the output tape as being one-way — it can only move its head from left to right. That actually turns out not to matter (a homework problem asks us to convert a log space transducer with a two-way head to one with a one-way head; that'll involve a method we'll introduce in a second).

So for the machine to compute f, when w appears on the input, when the machine halts, $f(w)$ appears on the output. The machine is deterministic, and it always has to halt; otherwise it's not computing a log space computable function.

Here's something that'll be useful (both on the homework and for us): just like a log space machine can only run for a polynomial amount of time, a log space transducer can still only run for a polynomial amount of time — having an output tape doesn't change things (you can just throw away the output, and you get the case we had before). That tells you the useful fact that the output is at most polynomial in size in the input — you're not going to have humongous outputs coming out of a log space transducer.

Theorem 19.5

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

We showed the exact same thing for P; that underlay the whole motivation for looking at reducibility (if a NP-complete problem ended up in P, then all problems in NP would). This will be the same rationale.

We can't just use the same proof, though. What would happen if we tried to?

Fake proof. Let R decide B in log space; we want to make S that decides A in log space.

We have a translation from A-problems to B-problems, so why don't we solve our A-problem by translating it into a B-problem, which we already know how to decide?

Let S do the following, on input w:

(1) Compute $f(w)$.

(2) Test whether $f(w) \in B$ using R.

(3) Accept if yes, and reject if no. □

The proof is wrong; why? Here, we need to keep $f(w)$ around — we compute it, and we've got to put it somewhere, and then we have to feed $f(w)$ into R to test whether it's in B. But $f(w)$ itself could be quite large — it could be polynomially large, so there's no way we can squeeze it down into our log-space work tape. So what do we do?

There's a trick, called the *recomputation trick*. Instead of writing down $f(w)$, we're going to compute the individual symbols of $f(w)$ on an as-needed basis when we're running R. Recall that R has its input tape, where it really wants $f(w)$ to be sitting there; but R is inside S and doesn't have its own input tape (because it's being simulated), so we can't write down all of $f(w)$. But instead we just write down one symbol of $f(w)$ — we can keep track of where R's head is on its 'virtual input tape.' If it's on symbol 20, we have to know what's on symbol 20 to know what R does. And to figure this out, we fire up our transducer, throwing away every symbol that shows up on its output tape until symbol 20 shows up. (It's convenient we have a one-way head on the output tape — this lets us just throw away symbols 1, 2, ... until we get to 20.) Then we can feed that symbol in and simulate R for one more step. Now suppose it moves its head a step left; now it'll need to know what's in the 19th location on the input tape.

We just computed that (right before we found the 20th), but we threw it away — and that means we just need to do it over again. So every time R takes a step, you have to rerun the entire transducer. This seems quite inefficient, but sometimes you save space by using extra time, and that's exactly what's happening here.

So we can fix our fake proof by using recomputation of the transducer output.

§19.3 NL-completeness

Now we have log-space computable functions, and therefore log-space reducibility; that gives us our definition of NL-completeness. Now we're going to prove that PATH is NL-complete.

Theorem 19.6

PATH is NL-complete.

So PATH plays an analogous role to SAT for NP, or TQBF for PSPACE.

Proof. First, we have PATH $\in$ NL.

To prove hardness, let $A \in$ NL be decided by a NTM N in log space. We have to give a reduction from A to PATH.

Here's the (log space) reduction f from A to PATH — we're reducing A to PATH. Suppose we have a string w (a candidate for membership in A); we want to output an object of the form $\langle G, s, t\rangle$ (i.e., a graph with two nodes, since that's what a PATH problem looks like). Unsurprisingly, we'll take G to be the computation graph for N on w (the collection of all the configurations, as seen before, with edges corresponding to legal one-step moves of N). We'll take s to be c_{start}, and $t = c_{\text{accept}}$. And that's the reduction.

What's left is to prove that this reduction is really in log space. (In general, we're not going to have to prove these statements in gory detail; a high level argument will suffice.)

To show f is log space computable, we'll give a log space transducer F — it'll take as input w, and its output should consist of

$$G = (c_1, c_3), (c_2, c_4), \ldots, (c_{12}, c_{22}); s = c_{\text{start}}; t = c_{\text{accept}}$$

(we're imagining the graph as a list of edges; we may also want to list the nodes). And F has to do this using only log space.

How does this happen? What F is going to do is take its work tape, and write down on the work tape a pair of configurations of N on w. (So it writes down a pair of nodes on this computation graph for N on w.) If it has two configurations written down, e.g. c_1 and c_2, it can inspect them and see whether c_2 follows from c_1. (The configuration says the location of the head on the input tape, and the location of the head and the contents in the work tape (built into the configuration), and the state; given this, it can just see whether N can legally go from one to the other.) If the answer is yes, it prints that pair out. And it cycles through every possible pair of configurations (in the most straightforward way, calculating like an odometer one symbol after the next — most of the time these won't even be legal configurations, but when we do have two legal configurations it'll check whether one yields the next; if yes it prints it out and moves on). This is kind of slow, but it operates in log space, which is what we care about. In the end, it's printed out the configuration graph; then it prints out the starting and accepting configurations, and it's done. □

Now to show that some other problem is NL-complete, you'll typically give a reduction from PATH to that other problem (we'll see an example in recitation, where we give a reduction from PATH to some other problem in NL to show that other problem is also NL-complete).

§19.4 NL and coNL

Now we'll move on to the second part of today's lecture, a bizarre theorem that amazed people when it came out — that NL = coNL. For comparison we don't think NP = coNP — for example, for the unsatisfiability problem (the complement of SAT), it doesn't seem like there's any short certificate for showing that a Boolean formula is *unsatisfiable*.

Theorem 19.7

We have $\mathsf{NL} = \mathsf{coNL}$.

Definition 19.8. We define $\mathsf{coNL} = \{\overline{A} \mid A \in \mathsf{NL}\}$.

For example, one consequence of this theorem is that the complement of PATH is in NL — you can make a nondeterministic log space machine that'll accept exactly when there is *no* path from s to t in the graph. We might think, why isn't that obvious — why not run the NL algorithm for PATH and do the opposite? But that's not how nondeterminism works — you can't just invert the answer of a nondeterministic machine. The whole way we set up the theory of nondeterminism means you can't just invert the output — because if you take every branch and flip the answer, if there was a mix of accepting and rejecting branches at the start, then there's still a mix when we flip, and the machine will still accept.

So we can't just do the trivial thing.

The way we're going to prove this is by working with our NL-complete problem — we will show that $\overline{\text{PATH}} \in \mathsf{NL}$ (which is sufficient — you have to show that if $A \in \mathsf{NL}$ then $\overline{A} \in \mathsf{NL}$, and you can do that using this statement).

What do we need to do? Imagine we have G, and two nodes s and t (this is our input). We're a nondeterministic log space machine; we're supposed to accept the input when there's no path from s to t (that's the only case you're supposed to accept). If there is a path from s to t, all of our threads have got to reject.

So the machine, when it's accepting on some thread, has to be *sure* that there's no path. How is that going to be?

There's two ideas to this proof. First, we're going to see how to solve the problem if we are given certain information for free. And then we'll show how to compute that information. This actually recaps exactly how this proof was discovered.

Proof, Part 1. The information we'll first be given for free is — suppose that R is the set of nodes reachable from s. Then t might be reachable or not reachable; we don't know. And we're trying to figure out whether t is in R or not. The special information we'll want is the *number* of reachable nodes — imagine I tell you that G has 100 reachable nodes. (I don't tell you what there are, just that there are 100 of them.) Once we know that, it's actually not too bad to come up with an algorithm that solves this problem — accepting when t is not reachable, and rejecting when t is reachable.

Let $R(G, s) = \{u \mid \text{path } s \text{ to } u\}$ and $c(G, s) = |R|$ (we'll just write R and c when it's clear). Imagine we're given c for free (for now; we'll later see how to calculate c).

First, here's the idea. Imagine $c = 100$; so I know there are 100 reachable nodes. But I'm given t, and I want to know, is it one of the reachable ones or not? How do you tell? What we're going to do is we're going to take the graph and go through every node in the graph, one by one. Every time we come to the next node, we're going to guess a path from s. (We're going to cut the path off at length m, so that we don't get into a loop.) So every time we get to the next node u, we guess a path coming from s. If that path hits t, then this branch of the nondeterminism has figured out that t really is reachable, so it's going to reject on its thread (since you're only supposed to accept if it's not reachable).

If the path hits u, we know that u is one of the 100 reachable nodes. We might have guessed the wrong path to get to u, and even though u was reachable, we might not have seen that; but for every node in the graph, if u *is* reachable, there will be *some* thread of the nondeterminism that finds the path going to u.

So as we're going through the nodes one by one, we're going to guess a path; if it takes us to u, we add one to the count, and move on to the next node. (If the path doesn't take us to the node, then we move on and don't add 1 to the count.)

At the end, we've gone through all the nodes. If we've ever seen t, we're going to have rejected on this branch already; that means we've never seen t. And at this point, we have some count of the nodes that we have *shown* are reachable from s.

If that count equals 100, that means this thread miraculously actually managed to find them all. Some thread is going to make all the right guesses for every one of the nodes in R (it's going to luckily find the right path). And that thread has kind of hit the jackpot — it has found out that it has found *every single node* reachable from s, and it knows t isn't one of them (because otherwise it would have rejected).

And I've found every node reachable from s, and none of them is t! So now I can accept, because I've found all the nodes that were reachable, and none of them were t.

We've assumed there were 100 reachable nodes; we go through all the nodes one by one, guessing a path to that node; if we find it we increment the count, while if we don't we just go to the next node. In the end, if our count hits 100, that means on that thread we found all the reachable nodes — we've actually managed to identify every node that's reachable (there are no others, because we know exactly how many the graph has). And none of them are t.

This is kind of twisted, but it's really cool.

So to recap, we've shown that if we can compute $c(G, s)$ on some thread (such that we know we've computed it correctly), then we can get what we want — that $\textsc{Path} \in \mathsf{NL}$ — as follows. On $\langle G, s, t\rangle$:

(1) Compute $c = c(G, s)$. Keep a count k, which we initialize to 0.

(2) For each node u in G, nondeterministically guess a path from s of length at most m (the number of nodes). If this path hits t, then *reject* (on this thread, of course). If it hits u, then increment k (replacing k with $k + 1$).

(3) After we've gone through every node, if $k = c$, then we have found all the nodes that are reachable, and none were t; so then we *accept*. Otherwise (if the count was wrong) we *reject*.

If the count was wrong, that means we were unlucky when guessing paths from s to u — we didn't find all the actually reachable nodes (there were 100 and we only found 95, for example); then we just give up on that branch, meaning that branch just rejects — because some other branch is going to get the right count, and we want that branch to be the one making the decision. □

Now how do we actually compute c? This is where things start to get hairy. Amazingly, you can compute c using the same idea, in a sense.

Proof. We're going to stratify R by distance — let R_i be the set of nodes reachable from s within i steps (i.e., the nodes at distance at most i from s). In particular $R_1 \subseteq R_2 \subseteq \cdots$. And let $c_i = |R_i|$ — so c_i tracks how many nodes are reachable of distance at most i from s.

What we're going to do is use c_i to compute c_{i+1}. If we know how many nodes are reachable from s of distance at most i, we're going to be able to use that, in a very similar way, to figure out how many nodes are reachable from s of distance $i + 1$. And then we're going to do that step by step, from 1 to 2 to 3 to $\ldots$ to i to $i + 1$ to $i + 2$ all the way up to m. And that'll be the number of nodes reachable within m steps, which is the total number of reachable nodes.

How do we do this? We're going to give a test for determining if a node is in R_{i+1}; and then we're going to apply that test, one by one, to all the nodes in the graph, to count up what c_{i+1} is. (We're trying to go from c_i to c_{i+1}; given c_i, we're going to calculate c_{i+1}. So given c_i, the first thing we're going to do is get a test for whether a node is in R_{i+1}; once we have that test, we'll apply it to every node in an outer loop, to check if it's in R_{i+1}. And we keep a count of those, and when we're done with that count, this tells us c_{i+1}.)

This is going to be a sort of souped-up version of what we did earlier. Imagine we have some node v; we want to test, is v in R_{i+1} or not? (And we know c_i.)

Using the same strategy as earlier, we're going to go through all the nodes u in the graph, one at a time; and hope to find all of the nodes which are reachable from s in at most i steps, since we know how many there are. So (in our inner loop) we go through the nodes u one by one. For each one of those nodes, we guess a path. If that path hits u, then we increment our count — I found another node of distance at most i — and at the end, I have a consistency check to see if I found them all. If I didn't, then this branch of the nondeterminism just gives up and rejects. But some branch of the nondeterminism will find all the nodes that are reachable, and it'll know that because it knows what c_i is.

And every time it finds a node that is reachable, it says, hmm…I know I got a node reachable within i. And it checks, does it have an edge to v? Because that would mean v is reachable within $i+1$.

So it finds all the nodes reachable within i, checks to make sure that it found the right number. Assuming the count did work, we found all the nodes. And while we were doing that (as we were going), we've checked, does any one of them point at v? If yes, we say, I know v is reachable within $i+1$. If at the end, you've found all the nodes and none of them pointed at v (but you've found them all), you know v is *not* reachable; and now the test says no, v is not reachable.

Some branch of the nondeterminism will guess everything correct; that branch will know the right answer, and it will know that it knows. □

Remark 19.9. The point is that if c_i is correct, then c_{i+1} is correct. And $c_0 = 1$, since R_0 is just $\{s\}$. From c_0 we can get c_1, and we'll know that c_1 is correct — because otherwise we reject.

The argument is essentially induction — assuming that c_i is correct (the inductive assumption), our algorithm guarantees c_{i+1} is correct.

In our algorithm, suppose $c_i = 100$; we go through all the nodes, and by luck we find all 100 reachable nodes in i steps, and we know we found them all because we know $c_i = 100$. While we were doing that, for each node we found that was reachable, we check whether it connects to v; if it does we know v is reachable, and if we've found all 100 nodes and none did, then we know v is not reachable.

We're not storing anything about which ones are reachable — we just track the current v and the current u. And every time we find a reachable u, we see whether it connects to v — we don't have to remember who the reachable ones are, just the count. If none of them connected to v, then v is not reachable within $i+1$ steps. And we just do this for each v and determine, is v reachable within $i+1$ steps, using this whole process over and over again for each v? And we add up the ones that we've determined are reachable; we know these answers are all correct, because otherwise we'd have rejected.

§20 November 21, 2023

§20.1 Review

Last time, we finished up the focused section on spaced complexity with our discussion of L; we defined NL-completeness and showed that PATH is NL-complete. And then we proved the bizarre theorem that NL = coNL. This was perhaps a bit hurried, and Prof. Sipser was thinking of redoing it today; but he doesn't think there's time to do both that and today's main topic — the hierarchy theorems (we can ask about it in office hours).

§20.2 Motivation

So far, we've seen the classes

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

As we've mentioned, none of these containments are known to be proper containments — most complexity theorists believe that *all* of them are proper (i.e., there are examples of languages — for example, the complete languages — that are in NL but not L). But we don't know how to prove that.

But we do know *something*. Each one of these containments might be an equality, but they're not *all* equalities — we know there *are* separations if you go far enough apart among these classes. In particular, the best thing that's known is that $\mathsf{NL} \neq \mathsf{PSPACE}$.

So there's got to be *some* inequalities — the containments $\mathsf{NL} \subseteq \mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PSPACE}$ cannot all be equalities.

> **Remark 20.1.** Could we go even higher? Yes, there are more classes after PSPACE we'll see today or next week — EXPTIME (exponential time), EXPSPACE, and so on (you can take higher and higher bounds).

We haven't proven that there's any decidable language outside of P in this course so far, so given what we know, everything could collapse down to L. But we'll show today that's not the case. The hierarchy theorems tell you that if you focus on a particular resource, like time or space, then as you increase your bound you're *guaranteed* to get new languages you can do. So for example, $\mathsf{TIME}(n^2) \subsetneq \mathsf{TIME}(n^3)$ — if you give the machine more resources, we can actually *prove* it can do more things. And the same is true for space — $\mathsf{SPACE}(n^2) \subsetneq \mathsf{SPACE}(n^3)$.

In particular, the reason we know $\mathsf{NL} \neq \mathsf{PSPACE}$ is that $\mathsf{NL} \subseteq \mathsf{SPACE}(\log^2 n)$, and we'll see that

$$\mathsf{SPACE}(\log^2 n) \subsetneq \mathsf{SPACE}(n)$$

(and of course $\mathsf{SPACE}(n) \subseteq \mathsf{PSPACE}$).

We'll have two hierarchy theorems, showing that bigger bounds give new things; we'll see the one for space first.

§20.3 Space hierarchy theorem

Theorem 20.2

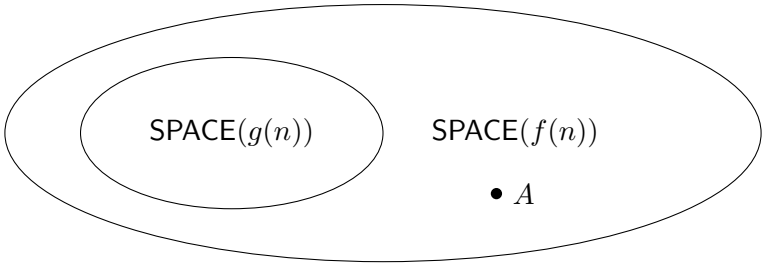
For any function $f : \mathbb{N} \to \mathbb{N}$ (where f satisfies a certain condition), if $g(n) = o(f(n))$, then $\mathsf{SPACE}(g(n)) \subsetneq \mathsf{SPACE}(f(n))$.

We'll specify the certain condition that f has to satisfy later. Here $g(n)$ is going to be our smaller space bound — e.g. n^2 — and $f(n)$ is going to be our larger space bound — e.g. n^3. Recall that the notation $g(n) = o(f(n))$ means that for large enough n, g is going to fall behind f by larger and larger constant factors — for *every* ε, for sufficiently large n we have $g(n) \leq \varepsilon f(n)$. (You pick any ε; then $g(n) \leq \varepsilon f(n)$ if you make n big enough.)

§20.3.1 The proof idea

We're now going to prove this. At a high level, the idea of the proof is simple; and we should at least take home the high-level idea. Making it all work requires fussing around with a bunch of important details, but we shouldn't lose the big picture for the main idea of why this is true.

Here's the proof idea. First, we want to show that there is some A such that $A \in \mathsf{SPACE}(f(n))$ and $A \notin \mathsf{SPACE}(g(n))$.



It would be nice if we could describe A in a simple way — e.g., $A = \mathtt{0}^k\mathtt{1}^k$ — those are the kinds of examples we've used in the past to show e.g. that the set of CFLs is a larger class than the regular languages.

But we're not going to be able to do that — we're going to describe A by giving a *machine* that describes it, and whatever that machine does is going to be A (it's not going to have a clearer description).

So what we need to do is give a TM D where $A = \mathcal{L}(D)$, which is going to have two properties:

- D runs in $O(f(n))$ space — this will give us the first condition we want.
- D is going to guarantee that $L(D) \neq L(M)$ for any M that runs in $O(g(n))$ space.

So if we have an M that uses a small amount of space, then D is going to *ensure* that its language is not the same as M's language; and this means D could not be decided in $O(g(n))$ space, because D's program is going to make sure that its language is different from any language that uses a small amount of space. That's going to be the hard part.

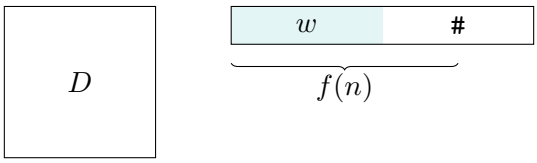
We'll say that D is designed to *block* every M that runs in small space — we'll say that D *blocks* M to mean that $\mathcal{L}(D) \neq \mathcal{L}(M)$.

At first glance this looks tough — there's infinitely many machines M that run in space $O(g(n))$. How can D make sure that its language is different from *any* of those infinitely many languages?

It's going to work using *diagonalization* — D will use diagonalization to get a language $A \neq \mathcal{L}(M)$ for all such M.

Here's our definition of D. On input w:

(1) First compute $f(n)$, and mark off $f(n)$ tape cells. We'll impose the condition on all the rest of the computation that if we ever try to use a piece of the tape beyond the marked part, then we automatically halt and reject.



This automatically ensures that D runs in $f(n)$ space; that's going to satisfy the first requirement. Now D has to do the hard work of making sure that its language is different from the language of any machine running in significantly less space.

The basic idea is that D is going to *run* all the different machines that use less space and figure out what they do, and then do the opposite. And the way it's going to do all of those different machines is that it's going to choose which machine to run based on the *input* it's given — the input is going to be taken as the description of a machine, and D is going to run the machine described by w on the input w (this is the diagonalization element). And D is going to see what that machine does, and then do the opposite; so its language is going to be different.

The only way D will be able to run that machine on w is if that machine uses less space than what D has allocated — otherwise D won't have enough space to run the machine, and is going to shut down. But that's fine — we only care about what happens for machines that use little space.

(2) If w is not the description of some machine M, then reject. (We're only interested in the w which are machine descriptions — because for those machines described, we're going to run those machines and try to be different.)

(2) Let $w = \langle M \rangle$, and simulate M on w.

  (a) If M accepts, then reject (because we're trying to be different).

  (b) If M halts and rejects, then accept.

(There are some details we're glossing over that we'll fill in later — this proof doesn't exactly work as written.)

This definition of D has the main idea; we'll see later that some issues will come up. But first let's see why D is a good candidate for doing what we need.

As pointed out earlier, D runs within space $f(n)$. How do we know that D's language is different from that of any machine running in much less space? If there is some machine running in a lot less space, then when w is the description of that machine, D is going to run that machine on w and do the opposite of what that machine does on w — so D's language will be different from M's language. And as w varies over all possible machine descriptions, we're going to block every machine that we can simulate within our limited space bound, and make sure we're different from them all.

Remark 20.3. As we vary w, we're going to be covering *all* possible TMs. Some are going to be running in an $O(g(n))$ bound; some are going to run in a lot more space, and there's no way we can figure out what they're doing in $O(f(n))$ space and be different; but we don't care about those.

Remark 20.4. Is it a problem if the complement of M is also in $\mathsf{SPACE}(g(n))$? No. If M runs in $\mathsf{SPACE}(g(n))$, then its complement does *necessarily* does too. But that's a different machine, and it'll be picked up by some other w — every machine has a description, and when we run that description, we do the opposite. So the complementary machine will be dealt with at some other point.

Remark 20.5. Would it work to run M on a well-sized random input and negate it instead? Why do we need to use this particular w?

We need to make sure that D differs from every single small space M. But here we're trying to say what D does on w. It's not going to be helpful to run M on some other string x, because D has got to decide whether to accept w or not — so it's going to feed w into M. Here w is serving two purposes here. On one hand it's describing the machine we're trying to block. And on the other hand, it's the actual input we're trying to differ on. This is inherent in these diagonalization arguments.

With that said, there's some problems with doing things the way described here. But we need to understand this outline in order to understand the issues and their fixes.

Remark 20.6. What happens if w is the description of D — how can D be different from itself? We have to look carefully and see what's going to happen when we run D on $\langle D \rangle$.

Let's suppose $w = \langle D \rangle$. Then the first thing that D does is it allocates its amount of space $f(n)$. Then it starts to go on and do the rest — it tests whether w is the description of a TM, and the answer is yes. And now it's going to simulate D on w.

And the very first thing that this simulation is going to do is, it's going to lay off $f(n)$ cells. But there's going to be a bit of simulation overhead — when we're doing a simulation, we can't fit $f(n)$ with a marker inside $f(n)$ with a marker. So as soon as we start to simulate D on w, in the simulation stage (1) it's again going to try to lay out that much tape. But when we're marking off this tape, we have to add in a new tape alphabet symbol, and so we're encoding a larger alphabet into a smaller alphabet, and there's going to be some overhead.

So D's tape has to simulate M's tape, but it'll be using a constant factor more in order to do that simulation. And that means D won't be able to simulate itself within exactly the same amount of tape — it's going to reject because it goes outside the space bound. So D on $\langle D \rangle$ is going to reject, because it doesn't have enough space to simulate itself on the same input.

§20.3.2 First issue — asymptotics

There are some issues with this proof.

Here's one: suppose that $g(n)$ is $o(f(n))$, but $g(n) > f(n)$ for small n. This could happen — it's only asymptotically that we know $g(n)$ falls behind $f(n)$.

And this is bad — suppose that in the place where we were trying to block M, the asymptotic behavior of M's space usage hasn't really kicked in yet, and M's space usage there exceeds what we can simulate in our $f(n)$ space. Then M might evade our attempts to block it — it masquerades as a large-space machine for small n and it's only for big n that it becomes a small-space machine, and the input w isn't big enough for that to appear.

So we have a problem — we need to block M, but our one opportunity to block M is going to lead to failure because we won't have enough space to simulate M on that one small input w.

The way to fix that is that instead of running M on w, we're going to run the same M on *infinitely* many w's — including some long w's. And we'll get that effect by having a redundant description of M, such that there'll be many w's that describe M (short w's as well as long w's).

We'll fix this in the following way: instead of checking if $w = \langle M \rangle$, we'll check if $W = \langle M \rangle \mathtt{01}^k$. So we'll take w, strip off all trailing `1`'s and the `0` before them; and what's *left* is what we check to be a machine description.

This means we can find a very large input on which to run the machine M; and since we're taking any value of k, there's going to at some point be a k at which the asymptotics have kicked in and $g(n) \ll f(n)$, which will allow us to run the simulation to completion. And then we'll be able to be different.

Said slightly differently, this modification means that instead of having a single opportunity to block M, we'll have infinitely many (on arbitrarily long w's). And if M's space usage is $o(f(n))$, then for some large w we'll see this asymptotic behavior and really be able to simulate M within our bound.

§20.3.3 Second issue — looping machines

Here's another issue. We've left out one of the possibilities in stage (3) — we said that if M accepts w within our space bound then we reject; and if M halts and rejects within that space bound, then we accept.

But what if M loops within that space bound? This is a possibility — you have a limited amount of space, and the machine can just keep going. If it uses too much space we'll shut it off, but if it sticks within the space bound but keeps going forever, then that's bad — because then D is going to loop on that w too (and D won't be a decider, which is bad because we defined our classes in terms of deciders only).

In other words, our issue is if M on w loops in $O(g(n))$ space.

So what we're going to do is add a counter, to shut M off if it runs for too long. Why is that okay? This goes back to ideas we've seen before — if M is using a limited amount of space and it's going for a really long time, then it's got to be repeating a configuration. So once it's gone more than some time exponential in its allocated space, it's going to be repeating a configuration, and then it'll *never* halt. So we can actually *detect* this — if M is running within $O(g(n))$ space but it has gone on for more than $2^{O(g(n))}$ time, then we're sure that it's not going to halt. So we can just reject in that case (though it doesn't really matter) — because M itself is not a decider, and we only need to be different from the small-space deciders (so if M is looping on some input, we have no obligations to be different from it, we just have an obligation to halt).

So what we do is we simulate M on w for $2^{f(n)}$ steps. What's important is that we can write down a counter up to $2^{f(n)}$ within $f(n)$ space (everything we're doing has to fit within our space bound). So we can count up to $2^{f(n)}$; and if M is really running in $o(f(n))$ space, then that means for large values of n, this will guarantee that if M has gone for this long then it's definitely looping.

So to summarize, our problem is that if M loops in $O(g(n))$ space then the simulation would loop. And the way we fix this is by adding a counter up to $2^{f(n)}$ keeping track of how many steps M has taken; if M hasn't halted after $2^{f(n)}$ steps, then we reject.

§20.3.4 Third issue — computability of f

There's one other issue that we swept under the rug when we wrote down D. In step 1, we said that D computes $f(n)$ and marks off that much tape. But it's conceivable that simply computing $f(n)$ might require more than $f(n)$ space — there are bizarre functions you can demonstrate such that just computing the value of that function cannot be done within that function's space bound. So knowing how much space you have to work with would require more space than what you have to work with.

There's actually no way around that problem — it's an inherent problem in this theorem. And the only way to deal with that is to impose a condition on f — that we can compute f within $f(n)$ space. Without that requirement on f, the theorem actually becomes false.

Definition 20.7. A function f is *space-constructible* if a TM can compute the mapping $1^n \mapsto f(n)$ in $O(f(n))$ space.

All the familiar large enough functions are space-constructible — for example $\log n$, $\log^2 n$, n^2, n^3, 2^n are all space-constructible.

Remark 20.8. Why use 1^n here and not a binary representation of n? This is how it's done in the textbook, and the reason is that we're given n as the length of the input (in the $O(f(n))$ space bound). If we wrote n in binary, then n wouldn't be the length of the input anymore; we just need a string of length n.

Remark 20.9. If $f(n)$ is sublinear, then we use the two-tape model for sublinear space complexity that we introduced.

But actually, there are examples of functions which are not space-constructible — for example, $\log\log\log n$ is *not* space-constructible. Things get weird when you have very little space available — in particular $\mathsf{SPACE}(\log\log\log n) = \mathsf{SPACE}(1)$, which is equal to the set of regular languages. So here's where the

hierarchy theorem fails — if instead of a constant amount of space I give you a tiny bit more (specifically, $\log\log\log n$), then you don't get anything new — a TM with only $\log\log\log n$ space available can only do the regular languages. (We're not going to prove this.)

So this is an example of why we need some condition on f — essentially, that the machine can figure out how much space it has, in order to utilize that space to be different from smaller-space machines. If it can't even figure out how much space it has, then the construction fails.

> **Remark 20.10.** If computing $f(n)$ uses more space than you have available, why don't you just put the boundary mark at $f(n)$ once you know where it is and then just move on?
>
> The problem is that D is supposed to run within $f(n)$ space; if we're using more than $f(n)$ space from the get-go, then we haven't succeeded. So we have to be able to compute $f(n)$ within $O(f(n))$ space, or else D is just not going to run within the space bound.

§20.4 Hierarchy theorem for time

The time hierarchy theorem has a similar idea, but there's a difference that sheds light on both of them.

> **Theorem 20.11**
>
> For $f:\mathbb{N}\to\mathbb{N}$ such that f is time-constructible, if $g(n) = o(f(n)/\log f(n))$, then $\mathsf{TIME}(g(n)) \subsetneq \mathsf{TIME}(f(n))$.

What *time-constructible* means is that f can be computed in $O(f(n))$ time (similarly to the definition of space-constructible).

What's going on with the division by $\log f(n)$? This is proving something weaker in terms of bounds — it's not enough that $g(n) = o(f(n))$, it has to be less than even $f(n)/\log f(n)$. So we don't get quite as tight a hierarchy for time as we did for space — for space, it's enough to go just a shade more to get new things, but for time you have to go up by a log factor to provably get more things. That's an artifact of the proof; we don't know whether tighter bounds are true or not.

§20.4.1 Proof idea

The idea is going to be the same — we're going to make a machine D giving some language A such that $A \in \mathsf{TIME}(f(n))$ but $A \notin \mathsf{TIME}(g(n))$, because D is constructed to make sure that its language differs from any language that we can do in $g(n)$ time.

So we're going to exhibit $A \in \mathsf{TIME}(f(n))$ where $A \notin \mathsf{TIME}(g(n))$, where $A = \mathcal{L}(D)$ for a machine D, that roughly works as follows.

First, instead of marking off $f(n)$ tape cells, we're going to start a counter — a 'clock' ticking down $f(n)$ steps. And we're going to make sure that D shuts off after $f(n)$ steps have gone by.

(1) Compute $f(n)$, and start a counter to ensure that the following steps ((2) and (3)) use at most $f(n)$ steps; *reject* if not (i.e., if (2) and (3) use more than $f(n)$ steps).

Remark 20.12. Why do we need f to be time-constructible? Suppose $f(n)$ is something like n^2 times a small extra factor, and it takes 2^n time to figure out what that extra factor is. Now when D is operating, if it computes $f(n)$, then it's already running in 2^n steps just to figure out what $f(n)$ is.

What if you start the counter before you compute $f(n)$ — including the time we take to compute $f(n)$ in the count? We could do that, but does that help? What's going to happen if we have that weird function for f described above? If we do that this way, we're going to start the counter at stage 1 (we don't know what we're counting up to, but we can start from 1).

But for that $f(n)$, this machine is not going to figure out what $f(n)$ is — it's always going to reject, and the A it's going to be producing is the empty language. So it's just going to die on phase 1.

So instead of laying out a maximum amount of space to use, we lay out a maximum amount of *time* to use.

(2) If $w \neq \langle M \rangle \mathtt{01}^k$, then reject; we'll only focus on the w's that actually encode machines.

(2) Run M on w, and do the opposite: *accept* if M halts and rejects, and *reject* if M accepts.

Here we don't have to deal with M looping, because we're going to catch that with the counter.

Why do we need the extra log? Incrementing the counter has an overhead — it's all because of the counter. So every time the machine wants to do one step of work, it's going to increment the counter, which adds a log factor overhead — this is because you have to drag the counter along with you as you move on your tape (if you leave it at a fixed location that's even worse, since you have to run back and forth a long distance to there). Actually incrementing the counter isn't hard, but keeping the counter nearby is the cost we pay in terms of time — and the counter has length $\log f(n)$, so dragging it around adds an extra log factor.

So this is only guaranteed to succeed if M runs in $o(f(n)/\log f(n))$ time, since we need this in order to have enough time to simulate M along with the counter.

Remark 20.13. If the machine left the counter in one spot — for example, at the left end of the tape — as the machine is doing the simulation, every time it does a step it'll have to run back to the beginning of the tape to increment the counter. That's a pain — instead of giving you a log factor, that's going to give you a squaring (every step costs you $f(n)$ steps to simulate, because you have to run back and take care of the counter).

So instead you carry the counter around (for example, using a different alphabet). But the counter is still $\log f(n)$ size, and so just moving it around is going to cost you.

§21 November 28, 2023

Before the break, we proved the hierarchy theorems — that $\mathsf{SPACE}(o(f(n))) \subsetneq \mathsf{Space}(f(n))$ (the things we can do on a TM with a certain amount of space grow if we give a bit more space), and $\mathsf{TIME}(o(f(n)/\log n)) \subsetneq \mathsf{TIME}(f(n))$. We proved this using diagonalization — we first used diagonalization to prove the undecidability of A_{TM}, and used it again here. There's a lot of similarity between the two proofs — we used diagonalization in a similar way.

Today we'll build on the hierarchy theorems (on Thursday we'll shift to probabilistic computation).

§21.1 Some corollaries

First, we'll get some corollaries of the hierarchy theorems.

Corollary 21.1
We have $\mathsf{NL} \subsetneq \mathsf{PSPACE}$.

So there's stuff in PSPACE that's not in NL.

Proof. We know from Savitch's theorem that $\mathsf{NL} \subseteq \mathsf{SPACE}(\log^2 n)$; and $\mathsf{SPACE}(\log^2 n) \subsetneq \mathsf{SPACE}(n)$ by the hierarchy theorem (we get new stuff as we increase the bound from $\log^2 n$ to n); and of course $\mathsf{SPACE}(n) \subseteq \mathsf{PSPACE}$. □

And in fact, not only do we get *something* new in PSPACE, but we can conclude that a *particular* language is in PSPACE but not NL — a natural choice is a PSPACE-complete problem.

Theorem 21.2
We have $\mathrm{TQBF} \notin \mathsf{NL}$.

Proof. The point is essentially that we've proved that everything in PSPACE is reducible to TQBF. But there's a subtle point here: we proved this using polynomial time reductions, which itself wouldn't be enough to give this conclusion. We actually need everything in PSPACE to be reducible to TQBF using a *log space* reduction (because then if $\mathrm{TQBF} \in \mathsf{NL}$, then you could use the reduction to get everything in PSPACE into NL, then contradicting the first corollary — if $\mathrm{TQBF} \in \mathsf{NL}$ then $\mathsf{NL} = \mathsf{PSPACE}$). In fact, we gave polynomial time reductions in this class for a bunch of things (the Cook–Levin theorem and a bunch of individual problems), but all those reductions can be actually done in log space — the reductions are very simple manipulations that don't require a lot of power or memory (e.g., they don't involve dynamic programming).

(We're not claiming that *every* polynomial time reduction can be done in log space — you can artificially construct ones that are at least not obviously doable in log space. But all standard examples of polynomial time reductions can also be done in log space, since the reductions tend to be simple translations.) □

So not only is NL properly contained in PSPACE, but the particular language TQBF is an example of a language in PSPACE but not NL. We'll expand on this idea for much of today.

§21.2 Intractability

The next corollary follows from the TIME hierarchy theorem.

Corollary 21.3
There is some decidable B which is not in P.

Proof. By the time hierarchy theorem, there exists a language that can be done in 2^n time, but not time significantly less than 2^n (and certainly not in polynomial time). □

Definition 21.4. We call decidable languages $B \notin \mathsf{P}$ *intractable*.

In this class, we won't just say that B *exists*, but give a nice example. The way the proof of the hierarchy theorem works, it demonstrates the *existence* of such a B, but that B is not in any way nicely described (it's the language of some Turing machine, but it's not a 'nice' language in the sense of being like $\mathrm{EQ}_{\mathsf{DFA}}$ — it's not 'natural' in a certain sense).

Today we'll do something like what happens in the second corollary — we'll give a particular example of a language which is decidable but not solvable in polynomial time (or even space). It'll follow the same kind of logic that was behind why TQBF is an example of a language not in NL — we'll give an example that's complete, but for a much larger class than P.

§21.3 The exponential classes

Definition 21.5. We define $\mathsf{EXPTIME} = \bigcup_k \mathsf{TIME}(2^{n^k})$ and $\mathsf{EXPSPACE} = \bigcup_k \mathsf{SPACE}(2^{n^k})$.

We'll give an example of a problem that's complete for EXPSPACE.

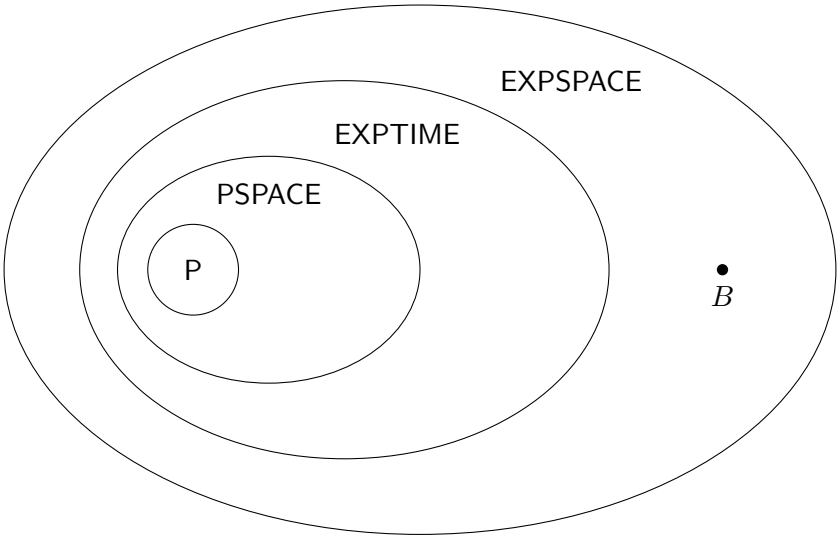
The hierarchy theorem immediately tells us that $\mathsf{P} \subsetneq \mathsf{EXPTIME}$ and $\mathsf{PSPACE} \subsetneq \mathsf{EXPSPACE}$.

Definition 21.6. We say B is EXPTIME-complete (or EXPSPACE-complete) if the following hold:

1. $B \in \mathsf{EXPTIME}$ (or EXPSPACE).
2. For all $A \in \mathsf{EXPTIME}$ (or EXPSPACE) we have $A \leq_{\mathsf{p}} B$.

So that's what it means to be complete for these very large complexity classes — the things we can do with an exponential amount of resources.

We're going to give an example of an EXPSPACE-complete problem B.



§21.4 An EXPSPACE-complete problem

Definition 21.7. *Exponentiation* of a regular expression R means repeating using concatenation a certain number of times — i.e.,

$$R^i = \underbrace{R \circ R \circ \cdots \circ R}_{i}.$$

If we allow ourselves to use exponentiation in a regular expression (together with union, concatenation, and star), that doesn't increase the power of what we can do — because we could just expand out the exponents. But it might make the regular expression much shorter, and that's going to be important.

Definition 21.8. We define $\mathrm{EQ}_{\mathsf{REX}^*}$ as the language consisting of all $\langle R_1, R_2\rangle$ such that R_1 and R_2 are regular expressions with exponentiation, and $\mathcal{L}(R_1) = \mathcal{L}(R_2)$.

Unsurprisingly, this problem is decidable (this is a straightforward exercise). But it actually turns out to be complete for EXPSPACE! And the proof of that is kind of interesting.

Theorem 21.9

$\mathrm{EQ}_{\mathsf{REX}^*}$ is EXPSPACE-complete.

This in particular implies that $\mathrm{EQ}_{\mathsf{REX}^*}$ is not in P (or even in PSPACE) — if it were in PSPACE, then since *everything* in EXPSPACE is polynomial time reducible to this language (as we will show), we would be able to solve *every* problem in EXPSPACE in PSPACE (by composing the reductions), contradicting the hierarchy theorems.

Remark 21.10. When we have a regular expression with exponentiation, we're going to write down the numbers of the exponents — e.g. 17 or a million — *in binary* (or decimal, or any numbering system — but not unary). This is important — we're not going to prove this, but when proving that $\mathrm{EQ}_{\mathsf{REX}^*} \in$ EXPSPACE, as a first step you're going to get rid of the exponents by expanding them out. Because those exponents are written in binary, that's going to blow up the size of the regular expressions by an exponential amount — you can write $R^{1000000}$ with a few bits, but if we wanted to eliminate the exponents we'd have to write R a million times. Then we get the problem of testing equivalence of two *ordinary* regular expressions; it's a nice exercise to prove that you can do this in PSPACE. So first we blow up our expression by eliminating our exponents (giving an exponentially large regular expression); and then we only need polynomial space on top of that (in the new length).

Proof. First, we have to show that $\mathrm{EQ}_{\mathsf{REX}^+} \in$ EXPSPACE; this is left as an exercise. (It's actually quite interesting — it uses Savitch's theorem.)

Next, we'll do the hardness part. Let $A \in$ EXPSPACE be decided by a Turing machine M in space 2^{n^k}; we want to show that $A \leq_{\mathsf{P}} \mathrm{EQ}_{\mathsf{REX}^*}$. So we want to give a map f reducing A to $\mathrm{EQ}_{\mathrm{REX}^*}$ — so for a string w (which may or may not be in A), $f(w)$ should be of the form $\langle R_1, R_2\rangle$ (two regular expressions with exponentiation) such that $w \in A$ if and only if $\mathcal{L}(R_1) = \mathcal{L}(R_2)$.

Remark 21.11. For the purposes of considering EXPSPACE-completeness we could use polynomial *space* reductions, but this is overkill. In fact this reduction can even be done in log space; we'll talk about polynomial time as a convenience so that we don't have to fuss around with log space transducers. It's worth looking at e.g. the reduction in the Cook–Levin theorem and seeing that there's nothing fancy there — we could do that with a log-space transducer — and the same is true here.

So we're given w, and we want to produce two regular expressions; we've got to say what those regular expressions look like. The idea here is that $\mathcal{L}(R_1)$ is going to be equal to all strings; so R_2 is what's doing all the work. And what we want is that if $w \in A$, meaning that M accepts w, then $\mathcal{L}(R_2)$ should *also* be all strings. If M *rejects* w, then we want $\mathcal{L}(R_2)$ to *not* be all strings. And the string that's going to be missing from $\mathcal{L}(R_2)$ is the *rejecting computation history* for M on w.

So we'll construct R_2 such that $\mathcal{L}(R_2)$ ends up being all strings *except* rejecting computation histories for M on w.

To check that this does what we want, if M accepts w, then R_1 generates all strings; and R_2 generates all strings except for rejecting computation histories for M on w, and if M accepts w then there *is* no rejecting computation history for M on w, so there are no exceptional strings — and R_2 generates all strings as well.

(Note that M is a decider, so it's either going to end up accepting or rejecting.)

Meanwhile, if M rejects w, then there *will* be a rejecting computation history for M on w, and so R_2 will *not* generate all strings, and it *won't* be equivalent to R_1.

Remark 21.12. Would it have been possible to take $\mathcal{L}(R_1)$ to be empty and $\mathcal{L}(R_2)$ to just consist of rejecting computation histories of M on w?

We're not doing it that way because constructing R_2 to generate *only* a rejecting computation history would be impossible, for a 'small' R_2. We've seen this kind of thing before — it's often easier to describe strings which *fail* to be computation histories, because they only have to fail in one place. Describing strings which *are* valid computation histories is more complicated — because those strings have to be good *everywhere*, while strings that fail only have to fail in one place (making them easier to describe).

So now we're going to construct R_2. This will consist of three parts, mirroring what we've done before. We want R_2 to describe all the strings that are *not* computation histories — a string is not a computation history if it starts bad, ends bad, or computes bad somewhere along the middle. So we'll construct R_2 as

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

Remark 21.13. Part of the reason it's easier to do bad strings instead of good strings is because our regular operations include union, but not intersection. If they included intersection, then looking for good strings would work — because then we'd have a way of intersecting a bunch of requirements, whereas here we can only union a bunch of requirements (making the complementary thing easier).

In our minds, let's have a picture of what a rejecting computation history looks like — because we have to describe all strings except for that one. A rejecting computation history looks like a bunch of configurations of the machine, step by step — something of the form

$$q_0 w_1 w_2 \dots w_n \sqcup \sqcup \dots \# \cdots \# \cdots\cdots \# \dots q_{\text{reject}} \cdots.$$

So we have configurations $C_1 = C_{\text{start}}$, C_2, …, C_{reject}. (This is how computation histories look — the sequence of steps the machine goes through, in this case until rejection.)

How big are these configurations? It'll be convenient to pad out these configurations so that they're all the same size. A configuration consists of the tape and the head location and state. The tapes are of exponential size — we assumed that M runs in space 2^{n^k}, so that's going to be the size of the configurations. So these configurations are pretty big — each has size 2^{n^k}.

Here's another question. How big is the *entire* rejecting computation history? That's going to be related to how much time the machine is taking — because that's how many configurations the machine is going through. And that could be doubly exponential — when we're running a machine that uses space s, it can use $\mathsf{exp}(s)$ steps (if it runs for longer than that, it's going to repeat a configuration, but that's the best bound we have). So this entire computation history could have length $2^{2^{n^k}}$.

So this string, which is the rejecting computation history for M on w (if it exists) is a *really* long string — we have a machine that has exponential space and could be going for doubly exponential time. So we should appreciate the job of our reduction — R_2 has to describe every string except for this enormously long string. But R_2 itself has to be polynomial in size (since f is supposed to be a polynomial time reduction, so it can only be producing polynomial sized objects). So R_2 is supposed to only be a polynomially large expression,

and its job is to describe all possible strings except this doubly exponentially long string. That's one of the features of this theorem — that it's somehow possible to accomplish this.

Let's take a look at how we're going to accomplish that in terms of the three parts. We're going to describe all the strings except for this one, as a union of three possibilities. We're going to describe every string that starts bad (i.e., does not start with $q_0 w_1 w_2 \ldots$), ends bad, or is bad along the way; then the only thing we *won't* describe is this string (since it's the only one that starts right, ends right, and is good everywhere along the way).

> **Remark 21.14.** Prof. Sipser remembers seeing this lecture when he was an undergraduate at Cornell, and the instructor was presenting this proof; he was totally lost, and there were tons of regular expressions on the board and he had no idea what was going on.
>
> Hopefully we're not in that state, but the main point is to understand the big picture; the details of the regular expression are not so important, but we have to do them to see what about regular expressions and in particular exponentiation is relevant here.

We'll first define $R_{\text{bad-start}}$. It'll itself be a union of a bunch of pieces — we'll define

$$\mathrm{R}_{\text{bad-start}} = S_0 \cup S_1 \cup \cdots \cup S_n \cup S_{\text{blanks}} \cup S_{\#},$$

where these pieces correspond to the elements of C_{start}.

First, why not list every string except for C_{start} of the correct length, and just attach Σ^* at the end? That'll be a regular expression that describes every string that doesn't start this way. But that'll be huge — C_{start} is already exponentially long, so if we want to list all strings except for this one, we'll end up with a huge regular expression. That's not good — we have to work harder to create a *small* expression.

And S_0 will be all strings that don't start with q_0, S_1 will be all strings that don't have w_1 in the second position; S_2 all strings that don't have w_2 in the third position; and so on, up to S_n. And then S_{blanks} will be all the strings that fail to have all blanks in the places they're supposed to be; and $S_{\#}$ all strings that don't have a # in the correct position. When we union them, we'll get all strings that start wrong.

First, the alphabet for our regular expressions will need to consist of the tape alphabet of M, together with our state symbols and # — so we'll define $\Delta = \Gamma \cup Q \cup \{\#\}$ to be the computation history alphabet. So in particular $R_1 = \Delta^*$.

What is S_0? We want S_0 to be everything that does not start with q_0 — so

$$S_0 = (\Delta - q_0)\Delta^*$$

(we really need to union together all elements of Δ other than q_0, but we'll write it this way for shorthand). For S_1, we can have anything happening in the first symbol, and we want something wrong in the second; so that's going to be

$$S_1 = \Delta(\Delta - w_1)\Delta^*.$$

And similarly

$$S_2 = \Delta^2(\Delta - w_2)\Delta^*,$$

and so on up to

$$S_n = \Delta^n(\Delta - w_n)\Delta^*.$$

By unioning these all together, we get all strings that don't start with our $n+1$ symbols $q_0 w_1 \ldots w_n$ — we've captured all the ways they can be wrong, with a fairly small expression.

Now we've got to do the blanks. We *could* continue on — we could say $S_{n+1} = \Delta^{n+1}(\Delta - \sqcup)\Delta^*$, and keep doing that for each one of these positions. But we don't want to do that — that would make an exponentially long expression, because we have exponentially many places to deal with.

So instead, we're going to use a bit of a trick. Instead of doing that, we're going to do them all at once — we define

$$S_{\text{blanks}} = \Delta^{n+1}(\Delta \cup \varepsilon)^{2^{n^k}-(n+1)}(\Delta - \sqcup)\Delta^*.$$

Why does this work? We first skip over $n+1$ places, which could be anything, because we've already taken care of those. Now we're going to skip over some number of places, so we want a 'variable spacer' — and we do that by using $(\Delta + \varepsilon)^x$ — we have Δ or an empty string, so that'll give us some number of Δ's between 0 and x. And then we're going to say we have a non-blank. The right value of x is how long this section is — 2^{n^k} minus how much we've already done, which is $2^{n^k} - (n+1)$. And then we have a non-blank; and then after that we can have anything. And lastly, we want

$$S_{\#} = \Delta^{2^{n^k}}(\Delta - \#)\Delta^*.$$

If $R_{\text{bad-start}}$ is built in this way, then it's going to describe every string that does not start correctly.

Now we want to get every string that doesn't *end* correctly; we'll say that's any string that does not have a q_{reject} symbol appearing somewhere. So we define

$$R_{\text{bad-reject}} = (\Delta - q_{\text{reject}})^*.$$

This is going to turn out to be enough — it describes all strings that don't have a q_{reject} appearing somewhere. Some strings might come up in multiple parts of our regular expression, but that's fine — as long as we capture all strings that don't have a q_{reject} appearing at the end, we're happy (and if q_{reject} doesn't appear anywhere, it's certainly not at the end). What would be bad is that we want to make sure this doesn't capture strings we *want* to have; but all rejecting computation history strings do have q_{reject} appearing, so they're not described by this. So this is an adequate way of describing all strings that don't end with a rejecting computation history.

Finally, $R_{\text{bad-move}}$ is the hardest; in order to understand it, we have to zoom in on what's going on in the middle. So let's take a look at that.

Imagine we have some

$$\cdots \#C_i\#C_{i+1}\# \cdots$$

somewhere in the middle of our rejecting computation history. We want to say that if C_{i+1} doesn't legally follow C_i, then we want $R_{\text{bad-move}}$ to describe that string. We're going to take advantage of something we've done before, which is neighborhoods — in the Cook–Levin construction we used 2×3 neighborhoods. Here we're going to look at neighborhoods abc and def appearing horizontally (rather than in a tableau picture), but the point is still that abc and def should be a 2×3 neighborhood. What we'll do is look at all the illegal neighborhoods — ones that are not valid according to the rules of the machine — and we want to capture all those illegal neighborhoods.

The tricky thing is the spacing between these neighborhoods — we want to say that abc does not validly lead to def, but def is supposed to be much later on in the string. So we have to see how big this gap is; and it's going to be 2^{n^k-2} (or something like that — essentially 2^{n^k}, with a bit of adjustment). So we're going to define

$$R_{\text{bad-move}} = \bigcup(\Delta^* abc \Delta^{2^{n^k}-2} def \Delta^*)$$

where the union is over all *illegal* tableau abc and def. This is going to describe *all* strings that have a bad neighborhood appearing somewhere. And in doing that, we're going to capture all the strings which have a bad step going from C_i to C_{i+1} in the rejecting computation history — they'll be included in what R_2 does.

So that's it — that's the construction! We have to double-check that this can be done in polynomial time, but this is true — all these regular expressions are pretty small (they just depend on the machine, and nothing complicated is going on).

Remark 21.15. Aren't there an exponential number of (abc, def)? No. These are the tableau

a	b	c
d	e	f

where each of a, b, c, d, e, f is either an element of Γ or a state — so the number of possible neighborhoods is a *constant* depending only on M (and not on $n = |w|$ at all). So this is *not* an exponential construction — if it was, we would be doomed.

This is the end of the proof — to repeat, this implies $EQ_{\mathsf{REX}\uparrow}$ is not in PSPACE, and therefore not in P. □

§21.5 Oracles

We'd *like* to say that e.g. SAT $\notin$ P, so that we could solve P = NP. Could we use the same idea or something like this? Pretty quickly on, people realized there's a meta-theorem saying that that approach is hopeless.

There's something called an *oracle Turing machine* — an oracle Turing machine has access to a language, in the sense that that language is provided as information for free to the Turing machine.

Definition 21.16. For a language A, a *Turing machine with oracle* A (written M^A) can get answers to questions 'is $x \in A$' for any x the machine comes up with, for free.

We won't define the model in detail, but you can imagine there's a special tape (or section of the tape) where it writes down a string x; and then it goes into a special oracle query state, and comes back with either a yes or no on whether $x \in A$. So we're given information about A without the machine needing to compute it.

We implement this by magic — it's an oracle, giving you useful information using divine connections in some way. So we should think of the oracle as a black box, which somehow produces the right answer.

How good is that? It depends on A. For example, if we have an oracle for SAT, then we can do all of NP in polynomial time — so an oracle for SAT would be super useful (you take your NP-language, reduce your input to a SAT problem, and ask the oracle about the SAT problem; and based on what the oracle says, you just decide whether to accept your input).

Definition 21.17. We define P^A as the set of languages B for which we can do B in polynomial time using some machine M^A (i.e., a machine with an A-oracle).

What we just proved is that $\mathsf{NP} \subseteq \mathsf{P}^{\text{SAT}}$. And we also have $\mathsf{coNP} \subseteq \mathsf{P}^{\text{SAT}}$, since P with a SAT oracle is a deterministic procedure with a magical black box; and if the black box says that the formula is *not* decidable, *then* you can accept your input.

Question 21.18. Is $\mathsf{NP}^{\text{SAT}} = \mathsf{P}^{\text{SAT}}$?

This is not known. But there *is* a theorem:

Theorem 21.19

For some A, we have $\mathsf{P}^A = \mathsf{NP}^A$.

This is weird — it says there is some oracle A where relative to that oracle, NP and P become equal. When you have that particular strange language A in place, nondeterminism doesn't help you.

This is too important to rush through; we'll state the language for now, and recap it next lecture.

Proof. Take $A = \text{TQBF}$ — for the TQBF oracle, we have $\mathsf{P}^{\text{TQBF}} = \mathsf{NP}^{\text{TQBF}}$. □

Not only is that a weird curiosity, but it also has very interesting philosophical implications, which we'll see next time.

§22 November 30, 2023

Today we'll start looking at probabilistic computation — we'll set up two models, one an analog of P and one of NP.

§22.1 Review

Last time we finished our discussion of the hierarchy theorems, and gave an explicit natural example of a problem $\text{EQ}_{\mathsf{REX}\uparrow}$ that is EXPSPACE-complete — and from that we can conclude that this language is intractable (it's not in P, or even PSPACE) — everything in EXPSPACE is efficiently reducible to it, so if it were in a lower class then it'd bring all of EXPSPACE down with it, and we know by the hierarchy theorems that EXPSPACE is strictly bigger than PSPACE.

Then we started our conversation around oracles, which is kind of an interesting topic in and of itself; but we're just going to briefly look at it because it has some relevance to P vs. NP and what kinds of methods you might be able to use to solve it.

§22.2 Oracles

A TM with an oracle for some language A (written M^A) is a TM that can answer questions of the form 'is $x \in A$' for free (it can ask many strings if it wants, and the oracle will give the answer with no cost to the machine — it doesn't take any steps to get the answer).

We define P^A as the set of languages which can be decided in polynomial time given an oracle for A (this is sometimes called *relativization*). Similarly, we can define NP^A (we can give a NP machine an oracle as well).

One interesting case is if we have an oracle for SAT — if I allow you to get answers to whether formulas are satisfiable, and I don't charge you for that answer, then you can do anything in NP — we have

$$\mathsf{NP} \subseteq \mathsf{P}^{\text{SAT}},$$

since we can reduce any NP problem to SAT and then ask the oracle. And we also have $\mathsf{coNP} \subseteq \mathsf{P}^{\text{SAT}}$ (P^{SAT} is closed under complement).

Question 22.1. Is $\mathsf{NP}^{\text{SAT}} = \mathsf{P}^{\text{SAT}}$?

This is open; in recitation we'll give another example of a language which is known to be in NP^{SAT}, but is not known whether to be in P^{SAT}. You might think that for *any* oracle you could give, a nondeterministic machine may be more powerful than a deterministic one. But as stated at the end of the last lecture, this is false:

Theorem 22.2
We have $\mathsf{NP}^{\text{TQBF}} = \mathsf{P}^{\text{TQBF}}$.

So if we have TQBF for free, then nondeterminism doesn't help you. This is kind of strange, but it's a one-line proof.

Proof. Anything we can do in NP with a TQBF oracle we can also do in NPSPACE — because if we have PSPACE, we can solve TQBF ourselves (we use our nondeterminism to simulate the nondeterminism of the NP machine, and our space to solve TQBF). But we know by Savitch's theorem that NPSPACE = PSPACE. And since everything in PSPACE is reducible to TQBF, we have $\mathsf{PSPACE} \subseteq \mathsf{P}^{\mathrm{TQBF}}$. So $\mathsf{NP}^{\mathrm{TQBF}} \subseteq \mathsf{P}^{\mathrm{TQBF}}$, and the reverse is of course true. □

This is a curiosity, but it actually also has significance — it tells you something about which methods you might not be able to use, at least in isolation, to solve the P vs. NP question.

We already know that for example P ≠ EXPTIME — we proved this using the hierarchy theorem. What we'd *like* to show is that P ≠ NP. The fact that we've been able to separate P and EXPTIME gives us perhaps some hope that we can use the same method to answer the P vs. NP question.

But we're going to argue that the same method won't work to show P ≠ NP.

Imagine you have two Turing machines, and one is simulating the other. Let's say M is the machine being simulated, and S the simulator. We have in mind a basic simulation, where every time M does a step, S basically carries out the same step on its tape as well.

Now if we have a simulation of this kind, and we give both M and S access to the same oracle, then the simulation is still going to work — every time M asks the oracle a question, S still has access to the same oracle, so it can ask the oracle the question too.

So if we have one machine simulating another, and we give them both access to the same oracle, the simulation is still going to work.

But if we think about the underlying method for proving P ≠ EXPTIME, we used diagonalization, and the core of that argument was really a simulation — inside the diagonalization, we have a machine which is simulating the faster machines to make sure it's different from what they do. (That was at a high level how the hierarchy theorem works.)

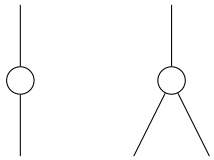
Now suppose that we could prove that P ≠ NP using a diagonalization-type argument, which has at its core a simulation. And now we put on both sides the same oracle — we put P^A and NP^A. Then the argument will still work — because inside the proof there's a diagonalization, which is just a simulation, and the simulation will still go through even though the oracle is there (the oracle is kind of invisible to the simulation).

So if you could prove P ≠ NP by a method which is at its heart a simulation, it would also prove that $\mathsf{P}^A \neq \mathsf{NP}^A$ for every oracle A. But that's false — we already know an oracle for which they're equal.

So this tells us something as simple as a simulation-diagonalization argument won't be enough to prove P ≠ NP. This was an important guide early on in the subject — people were trying to come up with complicated diagonalizations, and that shut down this approach. Even today, people have ideas that if you unpack them are basically diagonalizations; and this tells you it's not going to work.

§22.3 Probabilistic computation

A *probabilistic Turing machine* (written PTM) is a NTM where each step has one or two choices for the next move. Having one or two steps is not really an essential detail, but it'll make our lives easier. So a PTM can either have a deterministic step, or a nondeterministic branching, which we'll call a *coin toss*.



The essential difference between a PTM and a NTM is that we'll associate a probability with the two different ways to go. A NTM doesn't really take a random branch — it's really a collection of *all* branches, and we want to know if there *exists* one that ends up accepting. But with PTMs, we really *are* taking a random branch. In our computation, every time the machine comes to a nondeterministic step, there's going to be two possible ways to go; and it'll toss a coin to decide which one it takes.

So we can then talk about the probability that a machine takes a particular thread. The point is that the probability the machine takes a particular thread depends on how many coin tosses there were on that thread — every time there's a coin toss, there's a $\frac{1}{2}$ chance the machine would have taken the other track and ended up off the path. So the probability we take a particular thread is 2^{-k}, where k is the number of coin tosses. So for a PTM M on input w, we define

$$\mathbb{P}[M \text{ on } w \text{ takes thread } b] = \mathbb{P}[b] = 2^{-k},$$

where k is the number of coin tosses on b.

We also define

$$\mathbb{P}[M \text{ accepts } w] = \sum_{b \text{ accepting}} \mathbb{P}[b].$$

Each thread is either an accepting or rejecting thread (we assume we have a decider); so we take all the accepting threads and add up the probabilities that those threads actually happen — we just add up the probabilities of all the accepting threads.

The intuition here is that we have a machine, and some of its threads end up at an accept and some at a reject. And now imagine we run the machine randomly — every time it has a choice, we flip a coin on which way to go. Then we'll have some probability that we end up at an accept; and that's what we define as $\mathbb{P}[M \text{ accepts } w]$.

Of course, we can also define

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

§22.4 Languages in probabilistic computation

Now let's talk about languages. We want to think of a probabilistic machine as deciding certain languages. You can talk about the possibility that the machine might actually get the wrong answer; you presumably want that probability to be low on all inputs.

Definition 22.3. For a language A, we say a PTM M *decides* A *with error probability* ε (where $0 \leq \varepsilon \leq 1$) if for all inputs w, we have

$$\mathbb{P}[M \text{ on } w \text{ wrong}] \leq \varepsilon.$$

In other words, this means that if $w \in A$ then

$$\mathbb{P}[M \text{ rejects } w] \leq \varepsilon$$

(here M is getting the wrong answer on w — because for $w \in A$ you want M to be accepting w, so you want to bound the probability M is getting the wrong answer by rejecting it). Similarly, if $w \notin A$ then we should have

$$\mathbb{P}[M \text{ accepts } w] \leq \varepsilon.$$

We typically want ε to be a *small* value — if $\varepsilon = \frac{1}{2}$ the machine could just be tossing a coin instead, and it'd get the right answer half the time. So the problem isn't really interesting unless $\varepsilon < \frac{1}{2}$.

Definition 22.4. The class BPP is defined as the set of all languages A such that some polynomial time PTM decides A with error probability $\frac{1}{3}$.

This is sort of the probabilistic analog of P. We need some 'small' error probability, and here we use the cutoff as $\frac{1}{3}$. The idea is that these are languages where we have a probabilistic machine that runs in polynomial time — the notion of time is just as for nondeterminism, meaning that *all* threads have to halt within n^k steps — and it has to have a low error probability on *all* input.

Why $\frac{1}{3}$? This seems like a kind of arbitrary value that doesn't really belong in a nice, robust definition. But it turns out it doesn't matter — we could have picked $\frac{1}{4}$, or 1%, and so on — and we'd get the same class BPP. The point is that we can just run the machine several times, and we can improve the error probability down to something tiny by just taking the majority vote.

Lemma 22.5 (Amplification lemma)

If M_1 is a polynomial time PTM with error $0 < \varepsilon_1 < \frac{1}{2}$, and $0 < \varepsilon_2 < \frac{1}{2}$, then there is another polynomial time M_2 with error ε_2.

Here we think of ε_2 as being much smaller than ε_1 — we might have a machine with $\frac{1}{3}$ error, and we might want to improve it to some tiny ε_2.

(We need $\varepsilon_1 < \frac{1}{2}$, or else the machine would not be doing anything interesting.)

So you can always convert ε_1 error to ε_2 error, and maintain the fact that our machines are polynomial time.

Proof. We define M_2 on input w as follows: we run M_1 on w for k times (repeating its whole computation k times). Each time we'll get some answer from M_1; and we take the majority answer.

We're not going to go through the calculations, but for an appropriate value of k, we can make the new error probability as small as we like. The intuition is that M_1 is getting the right answer e.g. $\frac{2}{3}$ of the time and the wrong answer $\frac{1}{3}$, so if we run it once we don't have a whole lot of confidence. But we can run it a million times, and then chances are very high that out of those million runs, about $\frac{1}{3}$ of the time the machine will get the wrong answer, and $\frac{2}{3}$ of the time it'll get the right answer. So the majority answer is very likely to be right. □

Remark 22.6. We can improve the error probability not only to a tiny constant, but even to something *decreasing* — we can actually get $\varepsilon_2 = 2^{-\mathsf{poly}(n)}$. So we can improve from $\frac{1}{3}$ error to 2^{-n} error, or 2^{-n^2} error — so the probability of error goes down as the size of the input increases, and it goes down very fast. This is useful.

So we can convert any reasonable error to any other error.

§22.5 An example — branching programs

We'll now do an example, which will take us a while; but it's worth it. What we'll be going towards is an example of a language that's in BPP, but not known to be in P. All such examples are pretty nontrivial to prove; for this one we'll do the full argument, but it'll take us the rest of today (and into next week).

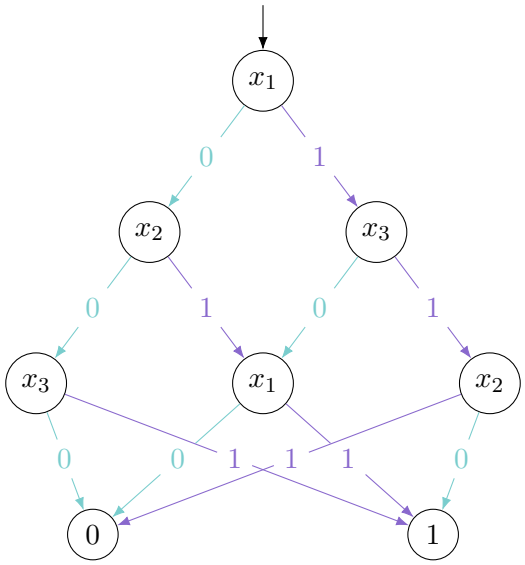
It'll involve a new model, called *branching programs*.

Intuitively, a branching program has a graph and nodes; there's a starting node, and every node has out-degree 2 except for two special nodes, the *output nodes*. And all the other nodes are labelled with variable names (we can have the same variable appearing more than once).

And the edges emanating from a node are going to be labelled with 0 and 1.

Finally, we don't have any directed cycles.

So there's a *start node*; then there's *query nodes*, each with a variable name on them and with outdegree 2, with edges labelled 0 and 1; and finally there are output nodes, also labelled 0 and 1.



We think of a branching program as describing a Boolean function. Imagine we have some assignment to the variables — for example, $x_1 = 1$, $x_2 = 0$, $x_3 = 1$. Then the branching program gives us an output — we put our finger on the start node, look at the label (here x_1), look at its value (here 1), and follow the corresponding outgoing edge. So this particular input setting will output 1.

So a branching program (BP) defines a function $f\colon \{0,1\}^m \to \{0,1\}$ (where m is the number of variables).

So we have a graph with no cycles, and to get the output from a given input setting, we just follow the path it determines — and the output node gives us the output value.

The computational question we'll look at is the *equivalence problem for branching programs*:

Question 22.7. Given two branching programs, do they compute the same function?

Definition 22.8. We define $\mathrm{EQ}_{\mathsf{BP}} = \{\langle B_1, B_2\rangle \mid B_1, B_2 \text{ are BPs that compute the same function}\}$.

We'll write $B_1 \equiv B_2$ to mean that B_1 and B_2 compute the same function (this doesn't mean they're identical, but they compute the same underlying function).

Fact 22.9 — $\mathrm{EQ}_{\mathsf{BP}} \in \mathsf{coNP}$.

In other words, this states there's a certificate for showing two branching programs are not equivalent. We can take that certificate to be an input setting on which they disagree. (So we can just guess, for our two branching programs, the place that shows they're not equivalent.)

It turns out that it's coNP-complete — we haven't exactly defined this (you can define coNP-completeness in two equivalent ways — either the complement of a NP-complete problem, or a language in coNP such that all languages in coNP reduce to it).

We're looking for an example of a language that's in BPP. It's unlikely to be this one — because if we had a coNP-complete language in BPP, then *all* coNP languages would be in BPP. And BPP is closed under complement, so that would mean NP $\subseteq$ BPP as well. Explicitly, if $\mathrm{EQ}_{\mathsf{BP}} \in \mathsf{BPP}$ then we would have coNP $\subseteq$ BPP, and NP $\subseteq$ BPP. So this is probably not true.

Remark 22.10. BPP stands for *bounded probabilistic polynomial time* — the *bound* refers to the fact that $\varepsilon < \frac{1}{2}$.

But the point is that BPP is a *practical* class, because we can make the error probability tiny — if you show a language is in BPP, then you can practically solve the language, because you can make the error probability really tiny. So if you can solve things in BPP, that's great news.

And so all of NP being in BPP is unlikely.

Instead, what we're going to do is look at a special case of this problem and show *that's* in BPP. And that special case has to do with restricting these branching programs.

Definition 22.11. A branching program is *read-once* (abbreviated ROBP) if it's not allowed to reread any variable on any given path.

In particular, our earlier example is *not* read-once — it has one violation, namely the path $x_1 \to x_2 \to x_1$. The fact that we have multiple instances of the same variable on different paths (e.g. the reuse of x_3) is not a problem; it's just being on the same path that's prohibited.

Theorem 22.12

We have $\mathrm{EQ}_{\mathsf{ROBP}} \in \mathsf{BPP}$.

This is nontrivial, and introduces a really important method; today we'll get ourselves warmed up, and we'll finish the proof next week.

First, here is an *attempt* to prove this.

Proof attempt. Imagine we have our two branching programs, and we do what at first seems to be the natural thing to try. We're trying to figure out, are these two branching programs computing the same function (i.e., do they agree on all inputs)? So a reasonable thing to do is, let's try running them. We'll use our probabilistic power to pick random values for the x_i and come up with some random assignment; and then we can feed it into the two branching programs and see what we get. And maybe we do this 100 times. And then what? If they agree every time, we'll accept; we'll say it *looks* like the two are equivalent. If they ever disagree, then we're *sure* they're not equivalent, so we can reject with total confidence.

So on $\langle B_1, B_2 \rangle$, we pick k random assignemnts to $x_1, \ldots, x_m$. (This is sort of just like nondeterminism — we can nondeterministically pick k assignments. But now we think of ourselves as tossing coins to actually make those choices.) And then we run B_1 and B_2 on each of these assignments. We reject if they ever disagree, and accept if they always agree.

What we *need* to have happen is that we have low error. Does this work?

We have to consider the cases where we're in the language and where we're not in the language. If the two branching programs are actually equivalent, then what's the probability this gets the wrong answer? We pick k random assignments and run both on those assignments; if they're really equivalent, then we get the same answer on all of them, so we accept with probability 1. (We're starting out with two equivalent branching programs, that compute the same function; and our algorithm randomly runs them both on the same assignments, and accepts if we always get the same answer. And they're equivalent, so of course we get the same answer all the time; so we accept for sure.)

So for w in the language, the probability we reject is 0.

But now suppose we're not in the language — we have two branching programs that don't agree everywhere. And we're picking k random places, and we reject if we find a place where they disagree. We need to make sure that if they really are different branching programs, we have to reject with high probability — so the

probability they agree on all our random sample points has to be low — if they really are different, we want to find that difference.

But we're not going to — imagine the two branching programs are not equivalent, but they just differ in one place (i.e., on just one input — they agree on all other inputs). Then we'd have to choose random samples enough times that we're going to find that rare case where they disagree. And to do that, we'd have to take an exponential number of samples.

So this doesn't work — we'd have to pick a value of k that's too large to make this polynomial time. □

Remark 22.13. Note that we're not saying to accept if all the *branches* agree, but if all our *random assignments* agree. The branches of our PTM don't talk to each other.

Suppose our machine is M; on input $\langle B_1, B_2 \rangle$, its nondeterminism will look like some huge tree. And these different threads don't talk to each other — we can't do something that goes across the threads.

But here we're not doing that — within an individual thread, we're picking k choices.

So let's start on how we're going to fix this — there's a really cool idea which we're going to use as the starting point.

The idea is that we're going to take these two branching programs and, instead of randomly selecting an assignment to run them both on where the assignment sets the variables to 0's and 1's, we'll actually pick a random assignment where we set the variables to non-boolean values — for example, 3, 4, and 5. So we're going to take our branching program, and assign it inputs which are *not* 0's and 1's. Then we're going to get an output, and we have to make sense of that.

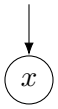
In order to get a sense of what we mean by this, we'll have to take our branching programs and look at them in a different way.

Imagine we have some branching program B. Normally we'd run the branching program by just following some path given by the answers to the queries we got from the variables. But we're now going to give an alternate way, which will be equivalent but will allow us to generalize to non-boolean inputs.

First, what we're going to do is — imagine we have an execution path of the branching program for a particular setting of the input, taking us from the start to some output. We're going to label *all* the nodes and edges of the branching program in a particular way. This will be simple — we label all the nodes and edges on this path with 1, and all the other nodes and edges (not on the path) with 0.

After we're done with that labelling, we look to see, what is the label on the output-1 node? The label of that node is going to be the output of the branching program. Here that's obvious — if the path goes through that node it'll have label 1, while if it doesn't it'll have output 0.

But what's going to be handy is that we can give another way of defining this without making reference to a path.



We always label the start node with a 1. Now suppose we've labelled x with an a. Then we're going to use the nodes to label the edges, and the edges to label the nodes. So we label our 1-edge with $a \wedge x$ (because we only take this edge if $a = 1$ and $x = 1$). And we label our 0-edge with $a \wedge \overline{x}$ (because we only take this edge if $a = 1$ and $x = 0$).

Similarly, if we have a node with edges labelled a_1, a_2, a_3 coming into it, then the label we put on this node is $a_1 \vee a_2 \vee a_3$.

And by doing this labelling, we can propagate the labels from the start node all the way through the branching program until we get to the output nodes; and we look at the label on the 1-node, and that'll be the output of the branching program.

Now what we're goign to do is develop a method called *arithmetization* — this means we're going to simulate $\vee$ and $\wedge$ with $+$ and $\times$ — AND is going to be multiplication, and OR is going to be $a + b - ab$.

Then we can convert this picture from boolean to arithmetic; and then it'll make sense to plug in values other than 0 and 1. And we'll get answers out of this; it'll turn out they'll be useful.

§23 December 5, 2023

§23.1 Review

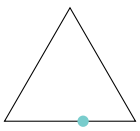
We're discussing probabilistic computation; we defined BPP, and we saw amplification and an example problem involving read-once branching programs. Today we'll prove that the language EQ_{ROBP} is in BPP.

Recall that BPP is the set of languages for which there is a polynomial time probabilistic Turing machine which decides the language. But the notion of acceptance is a bit different than before, because we have a probabilistic machine, which can sometimes make mistakes. What we want is that the mistakes are unlikely. We formalize that by saying that when inputs are in the language, the machine should accept with high probabilty; and when inputs are not in the language, it should reject with high probability. We arbitrarily set our cutoff at $\frac{1}{3}$; but the amplification lemma lets us change $\frac{1}{3}$ to a very tiny value (even one that decreases expnentially as n grows). The amplification lemma is going to be useful on our problem set.

§23.2 Probabilistic computation

Here's a somewhat different perspective on probabilistic computation than what we've had before; but a useful one. We can compare the computation trees for a nondeterministic machine M on w — imagine drawing a triangle representing the nondeterministic branching of a machine on an input, where the different paths through the tree are the threads of the computation.

Let's look at NP — imagine M running on w, and consider the computation tree of all the threads we'd see running M on w. When we're in the language, we want there to be at least one accepting thread — that's how we defined nondeterministic computation to work, that the machine accepts when there's at least one accepting thread. And it rejects if all the threads are rejecting.



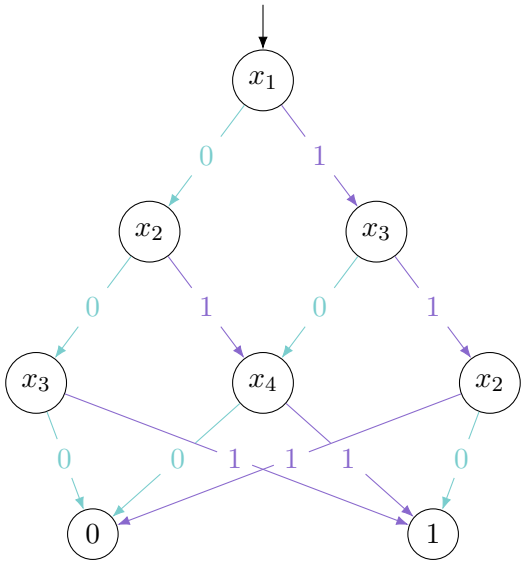
For BPP we can draw a similar tree, but the conditions are different. When we're in the language, we want *most* threads to accept — at least $\frac{2}{3}$ should be accepting, and $\frac{1}{3}$ rejecting. (We're cheating a bit here by assuming all threads have equal numbers of coin tosses — but we can always pad out threads with fewer coin tosses by making additional coin tosses that have no effect.) And most threads should be rejecting when we have an input that's *not* in the language.

So this is a more pictorial way of illustrating the same definition we saw last time for BPP.

> **Remark 23.1.** On the homework, we introduce another class RP, which is sort of a hybrid between NP and BPP. (BPP stands for *bounded probabilistic polynomial time*, and RP for *randomized polynomial time*.) This combines the rules that *all* branches have to reject when it's rejecting, while *at least* $\frac{2}{3}$ of the branches have to accept when it's accepting.

§23.3 Equivalence of branching programs

Recall that a *branching program* consists of a network of nodes, each labelled with variables. The output of the branching program on a certain assignment of variables is given by starting at the start node and following the edges given by this assignment, until we get to an output node.



The equivalence problem for branching programs is coNP-complete. But if we look at a special case, where we're not allowed to read a variable more than once on a path — such branching programs are called *read-once branching programs* — then we'll see that the problem is solvable in BPP.

Notation 23.2. We write $B_1 \equiv B_2$ to denote that B_1 and B_2 are equivalent.

We're given two branching programs; we want to accept if they're equivalent and reject if not.

§23.4 A naive attempt

Last class we saw a proof attempt, where we run our two branching programs on a bunch of random assignments of the variables. Iin terms of the threads, each thread of the algorithm (which we're thinking of as a probabilistic algorithm, since it picks assignments at random) will be picking k assignments to all the variables, at random. We'll be flipping coins for the values of x_1, x_2, x_3, ..., and we pick e.g. 20 different assignments. And once we have those 20 assignments, we plug them into the two branching programs, and see if we get the same answer all the time.

If we ever get a different answer, then we know the branching programs can't be equivalent. But if you get the same answer lots of times, you're never really sure you've just not found the possibly rare place they disagree.

And that's bad here — imagine we have 100 variables, so there's 2^{100} different assignments we could be making. And it's possible that the two programs agree on all but one assignment (it's easy to make such examples). Then by picking random assignments, chances are that we're not going to find the one rare place where they differ. And so in such cases where the two programs agree almost everywhere but not everywhere, the programs will not be equivalent, but they'll *look* equivalent to this algorithm. And then this algorithm is going to have a very high probability of error; that won't meet the condition we need to show the problem is in BPP.

So we'll have to do something different.

This is a fairly naive try — we just keep probing the two branching programs to see if we can ever find a place they're different. But we can only make a polynomial number of such probes.

What we're going to do instead is to introduce an idea called *arithmetization*, which is going to allow us to run these two branching programs on *non-Boolean inputs*. This is a kind of strange idea, because they're only defined if we assign the x_i's to 0's and 1's — but we want to make sense of assigning the x_i's to 2, or 3, or 4, and still getting an output. But that's going to be the key.

We're going to use the same idea in the next topic (interactive proof systems), where we'll do the same thing for satisfiability problems — we'll be plugging in non-Boolean variables into SAT and seeing what comes out. This sounds like gibberish, but there's a sense in which it's useful.

§23.5 Some algebra

Suppose we have a polynomial $p(x) = c_1x^d + c_2x^{d-1} + \cdots + c_{d+1}$; we say d is the *degree* of the polynomial.

Definition 23.3. We say a is a *root* of a polynomial p if $p(a) = 0$.

In other words, a root is an assignment of our variable which makes the polynomial evaluate to 0.

Theorem 23.4

A (nonzero) polynomial of degree d has at most d roots.

(There's a proof in the textbook; it can be done by induction.)

An easy corollary to this, and the form in which we're going to use it, is the following:

Corollary 23.5

If p_1 and p_2 are distinct polynomials of degree at most d, then p_1 and p_2 agree in at most d places.

Proof. This follows immediately from the theorem — we can let $p = p_1 - p_2$. If p_1 and p_2 agree in many places, then p will have lots of roots, which can't happen. So p_1 and p_2 can't agree very often, unless they agree everywhere. □

There's one more important fact we need, which is an extension of this — it's still true in any *field* $\mathbb{F}$. (A *field* is just a set with $+$ and $\times$ operations that work as you expect — satisfying associativity, distributivity, inverses, and things like that.) We're going to work over finite fields — we'll work with fields $\mathbb{F}_q$ with $+$ and $\times$ mod q, for prime q.

We shouldn't let this boggle us. The only importance of this thing about fields is that all our arithmetic is going to be done mod some prime q. So you can imagine some prime q, like 37; and all the arithmetic we're going to be talking about is done mod 37. But the value of this is that it's going to allow us to convert these statements into probabilistic statements.

Corollary 23.6

For a polynomial $p \neq 0$ of degree d (or at most d) over $\mathbb{F}_q$, for a *random* $r \in \mathbb{F}_q$,

$$\mathbb{P}[r \text{ is a root of } p] \leq \frac{d}{q}.$$

So we're going to pick a random value $r \in \mathbb{F}_q$ — i.e., a number between 0 and $q-1$ (all our arithmetic is mod q, so we can think of these as all the remainders when we divide by q).

Proof. There are q choices for r (from 0 to $q-1$). And p has at most d roots; so we have at most d out of q choices which can be a root. □

Now what we're going to do is extend this to multiple variables.

Theorem 23.7 (Schwartz–Zipple)

Consider a polynomial $p(x_1, \ldots, x_m)$ where each variable x_i has degree at most d. Then for random $r_1, \ldots, r_m \in \mathbb{F}_q$, we have

$$\mathbb{P}[p(r_1, \ldots, r_m) = 0] \leq \frac{md}{q}.$$

So now our polynomial has several variables; and each one can appear with exponents running from 0 to d. And we imagine choosing a random assignment; this states that the probability we land on a root is small. This generalizes the case where we had just one variable, where $m = 1$.

The important thing here is that when you have a polynomial, either in one variable or multiple variables, when you pick a random assignment the chance you land on a zero is very small. The way we're going to use this is that instead of talking about a single polynomial having a low probability of having a zero, we'll use that when we have two polynomials that are different, there's a low probability they'll agree.

We can already see why that's relevant — so far there's nothing about polynomials, but we have two things, and we want that when they're different there should be a high probability of disagreement. And we can get that from polynomials — so if we can massage this into a question about polynomials, then we'll be good.

Remark 23.8. What does it mean for two polynomials to agree? This means an assignment where the two polynomials give the same value. If we have some polynomial p_1 where $p_1(5) = 7$, and another polynomial p_2 where $p_2(5) = 8$, then they don't agree at 5. What we have in mind is that if $p_1 \neq p_2$, then they cannot agree often — most of the time they're going to be different, meaning that for most values you plug in, they'll give you different answers. The places where they agree — i.e., $p_1 = p_2$ — correspond to places where the polynomial $p_1 - p_2$ is zero; and we can't have too many zeros in our polynomial.

We're trying to show that two polynomials are not going to be equal to each other very often when we plug in the same input. Every time they are equal, the difference polynomial $p_1 - p_2$ is going to be 0 — if $p_1(5) = p_2(5) = 7$, then $p(5) = 0$ (where $p = p_1 - p_2$). But our theorems say that you can't have very many zeros in a polynomial, so there can't be very many places where p_1 and p_2 agree.

Remark 23.9. Does q have to be a prime? You can get fields of prime power size, but for our purposes we'll require q to be prime (that's simpler).

Remark 23.10. Why do we have q in the denominator and not q^m? You might imagine that because there are m choices, each ranging from 1 to q, there might be q^m different choices. But that's not what the theorem says; the theorem gives you something much weaker.

Remark 23.11. Both of these theorems are proved in the book.

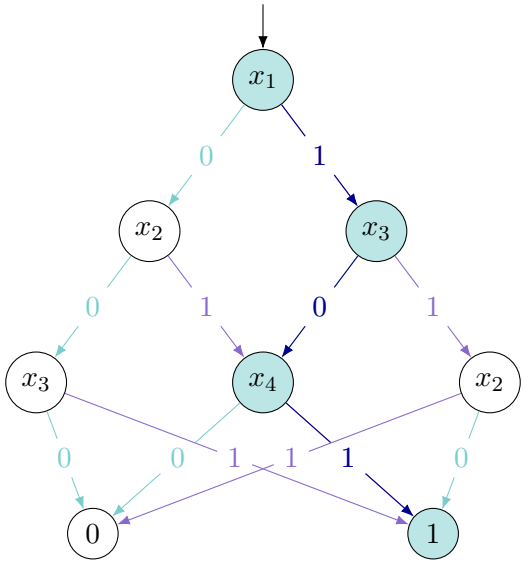
Remark 23.12. If q has to be prime, how are we going to pick our q in polynomial time? We'll have q be a prime, but not a very big prime. (We'll have to figure out how big m and d are; they'll just have to be big enough to make this $\frac{1}{3}$. It'll turn out that $d = 1$ and m is the number of variables; so this is a fairly small value. And picking a prime number around $3m$ is easy to do — you try them all until you find one, and we can test primality in polynomial time.)

§23.6 Reframing branching programs

So how do we massage our problem into polynomials?

The way we interpreted the execution of the branching program is by following a path. That interpretation is going to be less convenient for finding a polynomial version of branching programs; so we'll give a different way of evaluating a branching program, which will be equivalent when we have Boolean inputs but which will allow for non-Boolean inputs.

Imagine we have our branching program, with some execution path (the path that we're following for some given assignment of the variables).



We'll instead have an alternative view where we assign a value to all the nodes and edges — we put a 1 on all the nodes and edges on the path, and a 0 at all the nodes and edges off the path. (These values are not related to the edge labels — they instead correspond to the execution of the program on an input.)

What's nice about this is that we'll be able to define this without making reference to a path. We'll sort of take an inductive labelling of these nodes. We always label the starting node with 1. If we have a node x_i labelled with a (which is either 0 or 1), we label the 1-edge coming out with $a \wedge x_i$, and the 0-edge coming out with $a \wedge \overline{x_i}$ — and this does the right thing.

That tells us how to label edges from nodes. And to label nodes from edges, suppose we have a node with edges labelled a_1, a_2, a_3 coming in; then the label we want on our node is $a_1 \vee a_2 \vee a_3$.

And doing this, we can propagate labels down from the start to the end, until we get labels on the output nodes. And we look at the label on the 1-node (because if the 1-node has a label of 1 then the output is 1, and if it has a label of 0 then the output is 0). And the output is going to be a, the label on the 1-node.

§23.7 Arithmetization

The idea of arithmetization is to simulate $\wedge$ and $\vee$ with $+$ and $\times$. If we have $a \wedge b$, we can rewrite this as $a \cdot b$. (This is okay because it gives the same value that $\wedge$ gives you, when we assume `T` is 1 and `F` is 0 — so this is a faithful simulation.)

If we have $a \vee b$, we could try saying $a + b$. Is this a faithful simulation? No — because otherwise $1 \vee 1$ (which should be 1) would not be the same as $1 + 1$ (which is 2). So we have to subtract off ab — and then we get a good simulation of

$$a \vee b \mapsto a + b - ab.$$

And finally $\overline{a} \mapsto 1 - a$.

§23.8 The solution

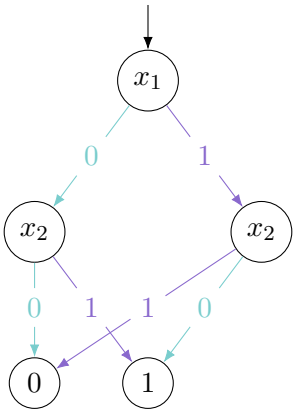
Now we can give a new labelling based on this arithmetization. Instead of $a \wedge x_i$, we're going to label the edge with $a \cdot x_i$. And instead of $a \wedge \overline{x_i}$, we're going to label the edge with $a(1 - x_i)$.

This is going to work the same as before when we plug in 0's and 1's for the x_i. However, this will start to make sense if we plug in e.g. $x_i = 2$ — something will come out. (That wouldn't make any sense for the Boolean operation.)

For going from edges to nodes, we're going to use $a_1 + a_2 + a_3$, without the correction of the subtractions. You might think this is not going to be a faithful simulation. But actually it will — the branching program is required to be acyclic, which means that when you enter a node, you can never *reenter* the node. So at most one of these a_i can be a 1, and the others have to be 0's. Maybe they're all 0's, but there can't be two 1's — that would mean the path went twice through that node, which can't happen. So that's why in this particular case it'll be good enough to just use addition — we never have more than one of these equal to 1.

If we do this to a branching program, then if we put in a Boolean input we'll get the same output coming out.

As an example, consider the following branching program (which happens to compute the XOR function).



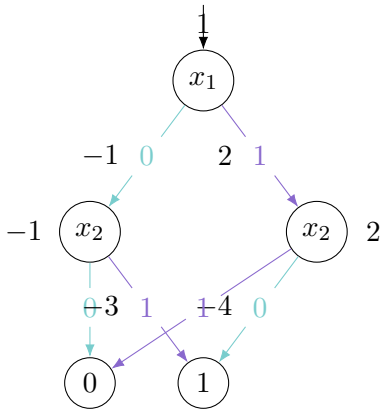
Imagine we consider the assignment $x_1 = 1$ and $x_2 = 0$. We start off by labelling x_1 with 1. Then on the edge $x_1 \to x_2$, we write $1 \cdot 1$ — which is what we want ($x_1 = 1$, so the path goes down this way). There's only one edge coming in, so we add them up and get 1; so we just have a 1 on x_2. And now what's going to be the label on the 1-edge out of x_2? We'll have to use the edge labelling rule here — the node is labelled with a 1 and we have a 1-edge coming out, but x_2 is assigned to 0. So we get $1 \wedge 0$ (or rather, $1 \cdot 0$), which is 0. But this is what we want — because the path on $x_2 = 0$ will actually go down the other way, giving a 1.

So if we have a Boolean input, everything works the way you'd expect it to — because we're just calculating, using arithmetic, the Boolean values.

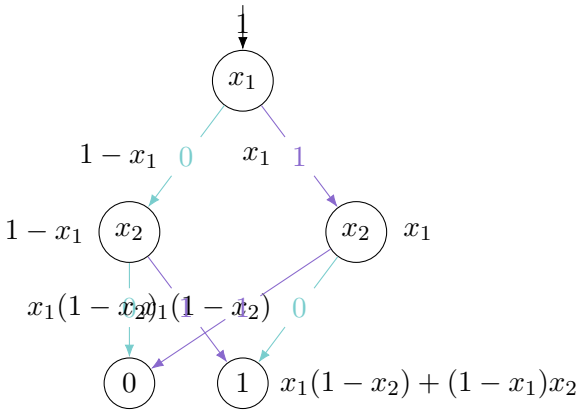
But now suppose we have a non-Boolean input — for example $x_1 = 2$ and $x_2 = 3$ — and now we're trying to compute the XOR of 2 and 3, whatever that means. What do we get?

We always put 1 on the start. But now on the 1-edge coming out of x_1, we can't use the Boolean rule; instead we use arithmetic to get $1 \cdot x_1 = 1 \cdot 2 = 2$. What's the label that goes on the 0-edge? That's $1 \cdot (1 - x_1) = 1 \cdot (-1) = -1$.

And then on the 0-outgoing edge from the right x_2, we get $2 \cdot (1 - x_2) = 2 \cdot (1 - 3) = -4$. And on the 1-edge from the left x_2 we get $-1(3) = -3$. And we have two incoming edges with -4 and -3, so we get -7. So this says the XOR of 2 and 3 is -7.



How do we go from this to polynomials? Suppose we do the same thing, but instead of plugging in values for the variables, we do a *symbolic* version of this execution. So we always put 1 at the start, but now we leave the variables as variables — we're not going to put in values for them anymore.



And so the output is going to be $x_1(1-x_2)+(1-x_1)x_2$. The point is that this is a polynomial representing the output of the branching program.

And the polynomial is going to be of a particular form, which is really important for our probability bounds. You'll always get a polynomial coming out when you do the symbolic execution of this arithmetizatiom. And the polynomial will always look something like

$$x_1(1-x_2)(1-x_3)x_4x_5\cdots x_m + (1-x_1)(1-x_2)x_3x_4(1-x_5)\cdots(1-x_m) + \cdots.$$

Why? Every time, we're either multiplying by an x_i or a $1 - x_i$, or we're adding things up. So the end result is going to be a product of x_i's and $1 - x_i$'s, and a sum of those.

There's a little detail here that we need to mention — if some branches thorugh the branching program don't read a variable at all, then there'll be a missing term here. That's a very slight complication, but let's ignore it for now and say that the branching program reads every variable exactly once. (You can fix this in two seconds, but we won't do it right now.)

So this is what the polynomial is. What does this polynomial do?

Call this polynomial $p(x_1, \dots, x_m)$. First, what happens if we put in 0's and 1's for the x_i into the polynomial? Then what we get out is the same value as the branching program — because this polynomial, on Boolean values, is a faithful simulation of the branching program (it does exactly what the branching program does), because we just simulated the ands and ors with $+$ and $\times$. But if we have other values besides 0 and 1, this is still going to give us *something*; we'll interpret that in a second.

So here's our revised algorithm. On $\langle B_1, B_2 \rangle$:

(1) Let q be a prime with $q \geq 3m$, where m is the number of variables in the branching programs.

(2) Pick $r_1, \dots, r_m \in \mathbb{F}_q$ randomly.

(3) Evaluate the branching program polynomials p_1 and p_2 on $r_1, \dots, r_m$.

(4) Accept if they agree (i.e., if $p_1(r_1, \dots, r_m) = p_2(r_1, \dots, r_m)$).

(5) Reject if not.

Claim 23.13 — If $B_1 \equiv B_2$, then $\mathbb{P}[\text{accept}] = 1$; and if $B_1 \not\equiv B_2$, then $\mathbb{P}[\text{reject}] \geq \frac{2}{3}$.

Proof. First suppose $B_1 \equiv B_2$. Why do the polynomials we get agree everywhere — not only on Boolean values, but also on the non-Boolean values that we might be picking from the field?

For each of these branching programs, we'll be getting some polynomial coming out, and we'll be plugging in these non-Boolean r_i's into them; why do they have to agree there? The idea is that you can look at the polynomials as a truth table for the Boolean function the branching program computes. For example, for the first row to be 1 we need $x_1 = 1$, $x_2 = x_3 = 0$, $x_4 = x_5 = 1$, and so on. So this corresponds to the different assignments to the variables. And only one of them is going to be a 1 — we're adding up all these possible assignments, and one of them is going to be a 1 if and only if the branching program outputs 1.

So we've got a row for each thing the branching program does. And so the branching program's values on the boolean inputs determines the polynomial — two branching programs which agree on all the inputs will have the same polynomials, which means the polynomials agree everywhere.

Now let's look at the other case. If $B_1 \not\equiv B_2$, then the two polynomials we get out are not the same polynomial. And now Schwartz–Zippel is going to apply. We have $d = 1$, and we picked $q \geq 3m$; so the probability we've chosen a place where the two different polynomials agree is at most $\frac{1}{3}$. So the probability of having an *agreement* when the two polynomials are not the same is at most $\frac{1}{3}$; this means the probability they *disagree* is at least $\frac{2}{3}$, which is what we want. □

Remark 23.14. Where are we using the read-once property? This is what ensures that every variable appears at most once in each product term. If we could read multiple times, then we'd have higher powers — for example $x_1(1 - x_2)(1 - x_3)^2 x_4(1 - x_4)x_5^3 \cdots$. And then the argument would fail — in particular, even the first part would fail, in the sense that we could have two polynomials that agree on Boolean places but disagree elsewhere.

§24 December 7, 2023

§24.1 Review

Today we're starting our last topic of the term — a really cool topic that's a launching point for a lot of more advanced work in complexity — called interactive proof systems. In some way this is a nondeterministic analog of probabilistic computation for deterministic machines — it's kind of nondeterministic probabilistic computation. This won't be immediately apparent, but it is in some sense a good analogy.

Over the last two lectures, we proved a theorem about branching programs — that if we have two read-once branching programs, we can test whether they're equivalent (i.e., whether they compute the same function) in probabilistic polynomial time. That was a fairly nontrivial method and analysis. The plan is for us to go over some of that in recitation tomorrow, if we feel we didn't really get it.

The main idea we introduced (which we'll see again) is the notion of arithmetization — where we take an object defined to operate on Boolean values, and we somehow extend it in a meaningful way to operate on other values too (and get information out of that).

One of the key ingredients — which we will see another time — is the Schwarz–Zippel theorem, which states (either in the single-variable or multi-variable version) that if we have two polynomials that are not the same, then their values have to disagree in many places. We used this to change our two branching programs (which we were trying to test for equivalence) into equivalent polynomials. If those branching programs were actually not equivalent, then their associated polynomials had to differ in many places, and so by probing the polynomials you could determine whether the branching programs were equivalent (meaning they agree everywhere) or not equivalent (meaning their polynomials disagree in many places).

Remark 24.1. As a sidenote, there's a notion of error-correcting codes — where you want to send messages such that if the messages become corrupted a bit, you can still recover the original message entirely. That can be implemented using something like the Schwarz–Zippel theorem — what you want to have in messages with error correction is that the class of messages you could send should have the property that two different messages are going to differ in many places. So among the class of messages we want to send, we want to forbid two messages which are very similar — all messages should differ in some large number of places from each other. And this gives us an error-correcting code — then by corrupting in a small number of places, you cannot convert it into another valid message (since all valid messages differ in many places). So you can recover the original message from the corrupted one (information-theoretically and algorithmically), since there's only one nearby message you can get by taking the corrupted message and changing it in a small number of places.

§24.2 Graph isomorphism problem

Today's model is going to be interactive proofs. It's a really remarkable thing — at its core it's super simple, but it does something kind of surprising. And it's good to illustrate it with an example. So we'll look at a new problem — the graph isomorphism problem, where we're given two graphs, and we want to know, are they really the same graph with one just being a permuted version of another?

So given two graphs, we want to test whether they're underlyingly the same graph — is there an isomorphism that maps the nodes of one graph onto the other, preserving edge relationships?

Definition 24.2. We define $\text{ISO} = \{\langle G, H\rangle \mid G \text{ and } H \text{ are isomorphic graphs}\}$.

For our purposes, these graphs will be undirected. We say two graphs are *isomorphic* if they're the same up to permuting vertices; we'll sometimes write $G \equiv H$ to denote that G and H are isomorphic.

What can we say about the complexity of this problem?

First, we immediately know $\textsc{Iso} \in \mathsf{NP}$ — the certificate is just the permutation (if I tell you which nodes map to which nodes, you can check that it works).

Question 24.3. Is $\textsc{Iso} \in \mathsf{P}$?

This is a very famous open problem, about which thousands of papers have been written (this is known as the 'graph isomorphism disease'). Part of the reason why this problem is so intriguing is that the Iso problem is known to be in NP, but it's not known to be in P or NP-complete. There's a phenomenon where combinatorial problems about graphs or sets or so on pretty much all end up being either in P or NP-complete — there's a dichotomy, and very few problems have an in-between status. And the graph isomorphism problem is the most famous among those. So it's kind of a target — to try to find a polynomial-time algorithm for it. There has been some progress, and some surprising sub-exponential algorithms (you can do better than brute force).

Question 24.4. Is $\textsc{NonIso} = \overline{\textsc{Iso}} \in \mathsf{NP}$?

This is also not known. (So we don't know any short certificate for showing two graphs are *not* isomorphic.)

But here's where interactive proofs come in. If two graphs really are isomorphic, there's a way for one party to convince another party that they're really isomorphic — just hand over the certificate. If the graphs are not isomorphic, we don't know if there's a short certificate. But there *is* a way for one party to convince another party that two graphs are not isomorphic, even without the notion of a certificate. This method is not infallible — it has a small probability of failing (in a way that we'll see) — but nevertheless, one party can still convince another party that two graphs are not isomorphic using a certain kind of probabilistic protocol, that's very simple and very cool.

§24.3 An interactive solution

First we have to understand the setup. The setup is that there'll be two parties — a prover and a verifier. (You can think of NP in this way as well — the prover comes up with the certificate, and the verifier checks it. We think of the prover as having unlimited computation (it has to find the certificate), but the verifier should run in polynomial time.)

Here we'll again have the prover be unbounded and the verifier running in *probabilistic* polynomial time, but there'll be another wrinkle — they can talk back and forth.

Imagine that Prof. Sipser is the verifier — he operates in probabilistic polynomial time. And he has two graphs, and he really wants to know, are they isomorphic or not? Fortunately, he has an army of research assistants, who have unlimited capabilities — they can stay up all night and know how to use computers, and they can do computations unbounded.

How is the verifier (Prof. Sipser) going to use the research assistants to help determine whether or not the two graphs are isomorphic?

Imagine Prof. Sipser has the two graphs G and H, and he gives them out to his students and says, I need to know, are they isomorphic or not? Let's imagine the students come back the next day and say 'isomorphic.' Prof. Sipser says 'great,' but knowing how students are, he doesn't fully trust that they actually did the work; so he needs to be convinced that they really are isomorphic.

And then the students say 'no problem,' and show him the isomorphism — because they worked hard and used their unlimited capabilities to figure out the isomorphism, and they've demonstrated it and Prof. Sipser can check it, so he's convinced.

But suppose the graphs are *not* isomorphic. Prof. Sipser still needs to be convinced — he won't take the students' word for it, he needs them to convince him that the graphs are not isomorphic.

So we go through the following protocol. Prof. Sipser takes the two graphs G and H, and privately turns around, flips a coin, and picks either G or H at random. And he doesn't disclose which one he picked; let's call the graph he chose K. And then furthermore, he's going to take K and randomly permute it.

So now K came from either G or H, and so it's a permuted version of G or a permuted version of H.

Now he takes that graph K, and sends it to the graduate students, and says, which one did it come from — did it come from G, or did it come from H? And then he does this 100 times.

What he's expecting the graduate students to be able to do is to identify the correct source every time. Why is that a reasonable expectation? If the graphs were really not isomorphic — Prof. Sipser really started out with two different graphs G and H — then the graduate students with their unlimited computation can figure out whether K is a permuted version of G or of H. But if G and H were the same, they'd have no way of telling — Prof. Sipser took one of them, further permuted it, and handed it over, and since the two possibilities look the same there'd be no way for the students to do better than a coin toss.

So Prof. Sipser runs this 100 times — sometimes picking G and sometimes H, and handing over those choices randomly scrambled to the prover — and the prover has to correctly identify each time whether it came from G or from H, which it can do with its unlimited computation (unless the two graphs really are the same — then there's no hope, since if you can't read Prof. Sipser's mind then you can't tell).

If the students ever fail — if the students ever says the graph K came from G, but Prof. Sipser had actually picked H — then he knows something fishy is going on, and he's not convinced anymore and won't trust the answer. But if the students get it right every time for 100 times, then it's pretty convincing that the two graphs really were not isomorphic. It's not 100% convincing, but all Prof. Sipser knows at the end of the day is that either the two graphs are really not isomorphic, or the students were *really* lucky — even though there's no way to tell, they were right 100 times. That could happen, but it's really unlikely.

There is a chance that the prover can incorrectly get the verifier to accept, but that's very unlikely no matter what the prover does. So at the end of the day, the verifier knows that either what the prover's trying to do is legit, or they got extremely lucky.

So that's the protocol; we're now going to formalize the model and restate the protocol in the formal model, and then go on to do something else with it.

§24.4 Interactive proofs model

We have two parties — a verifier and a prover. The verifier runs in probabilistic polynomial time (think of it as a Turing machine); the prover has unlimited computation. We'll typically refer to them as $\mathcal{V}$ and $\mathcal{P}$.

Both the verifier and prover are going to get to see an input; in the end, we're trying to have the prover and verifier work together to decide some language. So they're both going to see the input w.

Then they exchange a polynomial number of polynomial-size messages (polynomial in $n = |w|$). And then the verifier outputs either *accept* or *reject*.

So that's the setup.

When we have a verifier and prover of this kind, we define

$$\mathbb{P}[(\mathcal{V} \leftrightarrow \mathcal{P}) \text{ on } w \text{ accepts}]$$

(the notation $\mathcal{V} \leftrightarrow \mathcal{P}$ denotes the verifier interacting with the prover) as the probability that the verifier outputs *accept* when interacting in the above way with the prover.

So we're given a verifier, which can toss coins. And we're given a prover; let's assume the prover is deterministic (it doesn't really matter), so the prover has unlimited computation, but whenever the verifier sends a message it'll reply with some other message. They take turns exchanging things — for example, in the graph non-isomorphism protocol the verifier was picking G or H, scrambling, and sending the result to the prover; and then the prover was responding with another message, on whether it came from G or H. And in the end the verifier decides to accept or not.

If the underlying language was NonIso, what we want is for $\mathcal{V}$ to accept when the graphs are not isomorphic.

Definition 24.5. We define the class IP as the collection of languages B such that for some verifier $\mathcal{V}$ and prover $\mathcal{P}$, for all $w \in B$ we have

$$\mathbb{P}[(\mathcal{V} \leftrightarrow \mathcal{P}) \text{ accepts } w] \geq \frac{2}{3},$$

and for all $w \notin B$, for *every* prover $\mathcal{P}^*$ we have

$$\mathbb{P}[(\mathcal{V} \leftrightarrow \mathcal{P}^*) \text{ rejects } w] \geq \frac{2}{3}.$$

When $w \notin B$, we require that not only does the verifier interacting with the prover reject with high probability, but the verifier interacting with *any* potential prover should reject with high probability.

So when we're in the language, $\mathcal{V}$ and $\mathcal{P}$ are working together, and $\mathcal{P}$ should be able to make $\mathcal{V}$ accept with high probability. When we're not in the language, what we want is not only does *that* prover make the verifier reject, but no matter *what* the prover tries to do — even if it's a crooked prover trying to cheat. In the case for NP, the prover can be sending over a certificate; when we're in the language, the verifier will accept with probability 1. If we're not in the language, there is no certificate, but the prover could potentially try to send over a bogus certificate. And what we want is that no matter what the prover tries to do, if we're not in the language the verifier is going to be able to detect something is fishy, and so reject with high probability.

Remark 24.6. There's some probability of an error occurring — namely $\frac{1}{3}$ — but by repeating multiple times and taking a majority vote, you can make the probability of error very tiny (again using amplification).

We'll call $\mathcal{P}$ — the one showing something's in the language — the *honest prover*. And you should think of $\mathcal{P}^*$ as trying to outfox the verifier and make it accept when it shouldn't; we'll call it a *crooked prover*. Together, we'll call the verifier $\mathcal{V}$ with the honest prover $\mathcal{P}$ the *protocol*. So $\mathcal{V}$ together with $\mathcal{P}$ are going to follow the protocol. When you're not in the language, you can try to follow the protocol, but $\mathcal{V}$ should end up accepting with low probability.

§24.5 Formalization of the NonIso protocol

Theorem 24.7

We have NonIso $\in$ IP.

To prove this, we need to give $\mathcal{V}$ and $\mathcal{P}$ satisfying the desired condition. They'll work as follows, on input $\langle G, H\rangle$: what we want to do is that when G is not isomorphic to H the prover should make the verifier accept with high probability. And when $G \equiv H$, the prover should be unable to make the verifier accept — no matter what the prover tries to do, the verifier should reject with high probability.

First, here's the protocol (for the verifier and honest prover).

(1) Repeat the following twice.

(2) The verifier $\mathcal{V}$ sends a message to the prover $\mathcal{P}$ (written as $\mathcal{V} \to \mathcal{P}$): first $\mathcal{V}$ randomly picks either G or H. Then it randomly permutes the vertices and sends the result K to the prover.

(3) $\mathcal{P} \to \mathcal{V}$: first $\mathcal{P}$ tests whether $G \equiv K$ and whether $H \cong K$. In the legitimate case (where $\langle G, H\rangle$ is in the language, so G and H are not isomorphic), one of these is going to be a 'yes' and the other a 'no.' So the prover can figure out which, and it sends which is 'yes' to the verifier.

(4) $\mathcal{V}$ accepts if *both* answers are correct (i.e., on both repeats the prover has chosen G or H according to the one that the verifier picked at random — so $\mathcal{V}$ accepts if $\mathcal{P}$ was correct on both responses); otherwise it rejects.

For $\langle G, H\rangle \in$ NonIso (so in other words $G \not\equiv H$), what's the probability $(\mathcal{V} \leftrightarrow \mathcal{P})$ accepts? We can go back to our story — imagine that Prof. Sipser ran the protocol twice, of picking G or H at random, randomly scrambling it, and handing it to these computationally powerful students. If G is really non-isomorphic to H, what's the probability the students are going to get the right answer both times? It's 1 — because if G and H are really not isomorphic, then the prover can figure out which one K came from. So the honest prover is going to get the right answer both times, and it is guaranteed to make the verifier accept — no matter what coin tosses the verifier ended up flipping, the prover will find the correct answer both times, and so

$$\mathbb{P}[(\mathcal{V} \leftrightarrow \mathcal{P}) \text{ accepts}] = 1 \geq \frac{2}{3}$$

(which is what we want).

Now what happens if $\langle G, H\rangle \notin$ NonIso, meaning that $G \equiv H$ — then what's the probability that $(\mathcal{V} \leftrightarrow \mathcal{P}^*)$ accepts? First let's consider the honest prover — when it's testing whether $K \equiv G$ and $K \equiv H$, what result is it going to get? It'll get back two yeses, since G and H are themselves isomorphic, so K is isomorphic to both of them. If it sends 'well, it's isomorphic to both of them,' then the verifier will say, 'well, I don't believe the two graphs are isomorphic, so I'll reject.' So the best the prover can do is pick one of them at random; and then $\mathcal{V}$ will have a $\frac{1}{2}$ chance of catching it — $\mathcal{P}$ has no idea what $\mathcal{V}$ picked, and so it'll guess each one right with probability $\frac{1}{2}$, and so it'll guess both right with probability $\frac{1}{4}$. This means

$$\mathbb{P}[(\mathcal{V} \leftrightarrow \mathcal{P}^*) \text{ accepts}] \leq \frac{1}{4}.$$

(We have an inequality here because the prover $\mathcal{P}^*$ could just be misbehaving, and it could just say 'the two graphs are isomorphic, I'm not even going to try to give you one or the other and I'll just party tonight.' In that case, you'll have even less than a $\frac{1}{4}$ probability of succeeding.)

Remark 24.8. There's two ways for K to be isomorphic to G. It could be that the verifier picked G in the first place; then K has got to be isomorphic to G, because that's how it was designed. But if the verifier picked H, then K will be isomorphic to G if G is isomorphic to H. (It'll be isomorphic to both of them.) And so the prover has to guess, and they'll have a $\frac{1}{2}$ chance of being wrong.

Remark 24.9. If the prover gives an answer that isn't just a single graph, then the verifier can just reject.

§24.6 Some properties of IP

Fact 24.10 — We have NP $\subseteq$ IP and BPP $\subseteq$ IP.

To see that NP $\subseteq$ IP, we can do a protocol like this — the verifier doesn't even need to be probabilistic. In this case, the prover just sends over the certificate, and the verifier checks it. Then the verifier doesn't

need its probabilisticness, and it has a very simple interaction — it just gets a single message, and accepts or rejects.

To see that $\mathsf{BPP} \subseteq \mathsf{IP}$, we can just ignore the prover, and the verifier can decide the language by itself.

There's one more inclusion we'll give.

Fact 24.11 — $\mathsf{IP} \subseteq \mathsf{PSPACE}$.

This we're not going to prove and it's not so obvious, but it's kind of standard — if you go through and think about what you need to do to analyze the verifier and the prover, in polynomial space you can go through all possible interactions and figure out the probability that the verifier would accept with a honest prover. We won't prove this, but we should know it's true.

Remark 24.12. The *prover* is all-powerful. The prover could be deciding something more powerful than PSPACE, but the prover never actually *needs* to be more powerful than that. In cryptography you often have a model with limited-capability provers (and you want to have the prover convince the verifier that the prover e.g. knows the secret key) — that fits within this model, except with limited provers.

It turns out you only need provers with PSPACE capabilities, but that is a separate matter.

The *really* surprising result is that this goes the other way.

Theorem 24.13

We have $\mathsf{PSPACE} = \mathsf{IP}$.

In fact, it's quite amazing this is true — it says that for any PSPACE problem (for example, various $n \times n$ board games), you can have an interactive proof — remember that to actually test which side has a winning strategy, as far as we know, you effectively have to go through the entire game tree. But a powerful prover can convince a PPT $\mathcal{V}$ that e.g. white has a win in chess, without actually going through the entire game tree. So the Martians with a supercomputer can actually convince a PPT player that white has a win in chess, without going through the game tree (which you can't do in probabilistic polynomial time).

When this came out, this blew everyone's minds. It's unexpected, and the method is interesting.

We will not prove this (though we should know it is true). But we will prove the following weaker form, that still has much of the surprise element to it, and has basically the same type of proof (though it needs to be taken to a slightly higher level to get the stronger statement).

Theorem 24.14

We have $\mathsf{coNP} \subseteq \mathsf{IP}$.

This is not obvious at all. For example, how would you do an interactive proof showing that a formula is *not* satisfiable? This is not at all obvious. To show it *is* satisfiable, the prover can just hand over the satisfying assignment. But showing that it is *not* satisfiable seems a lot harder.

§24.7 Proving coNP ⊆ IP

We'll look at the following specific problem.

Definition 24.15. We define $\#\textsc{Sat} = \{\langle \varphi, k \rangle \mid \text{cnf } \varphi \text{ has exactly } k \text{ satisfying assignments}\}$.

Fact 24.16 — #SAT is coNP-hard.

Proof. We have $\overline{\text{SAT}} \leq_{\mathsf{P}} \#\text{SAT}$, since testing whether a formula is unsatisfiable is the same as saying the formula has zero satisfying assignments — we can use the mapping reduction $f(\varphi) = \langle \varphi, 0 \rangle$. □

Theorem 24.17

We have #SAT $\in$ IP.

This will imply that coNP $\subseteq$ IP.

First, we need some notation.

Notation 24.18. For a formula φ, let $\#\varphi$ denote the number of satisfying assigments to φ.

(So this is the real number; k may or may not be this number.)

Notation 24.19. We write φ_0 to mean φ with $x_1 = 0$ preset (meaning we substitute 0 for x_1 and simplify), and similarly φ_1 to mean φ with $x_1 = 1$ preset. Similarly φ_{101} means φ with $x_1 = 1$, $x_2 = 0$, and $x_3 = 1$ preset, and so on — $\varphi_{a_1 \dots a_i}$ is φ with $x_1 = a_1, \dots, x_i = a_i$ preset.

So we're going to take some of the variables and set them to constants and simplify, and this is our notation for that — $\varphi_{a_1 \dots a_i}$ means we take the first i variables, set them to given constants $a_1, \dots, a_i \in \{0, 1\}$, and simplify.

We want to do the same thing for counting the number of satisfying assignments: we use $\#\varphi_{a_1 \dots a_i}$ to denote the number of satisfying assignments once we have preset the first i variables to $a_1, \dots, a_i$. So we can write

$$\#\varphi_{a_1 \dots a_i} = \sum_{a_{i+1}, \dots, a_m \in \{0,1\}} \varphi_{a_1 \dots a_m}.$$

What this means is — we want to count the number of satisfying assignments once we've preset the first i variables. And we can calculate that by just plugging in all possible settings of the *other* variables to be 0's and 1's. Sometimes that'll give us a satisfying assignment (when we add 1) and sometimes it won't (when we add 0); so this sum is counting up the number of satisfying assignments with these presets.

There'll be two useful facts. First, we have

$$\#\varphi = \#\varphi_0 + \#\varphi_1.$$

And similarly, we always have

$$\#\varphi_{a_1 \dots a_i} = \#\varphi_{a_1 \dots a_i 0} + \#\varphi_{a_1 \dots a_i 1}.$$

(We're just extending by one place by taking the next variable and setting it to either 0 or 1, and adding up the number of satisfying assignments in those cases.)

And lastly, once we've set everything, we have

$$\#\varphi_{a_1 \dots a_m} = \varphi_{a_1 \dots a_m}$$

(since the number is 0 if this isn't a satisfying assignment — i.e., the formula evaluates to 0 — and 1 if this is — i.e., the formula evaluates to 1).

Proof attempt. First, here's a proof attempt (this is not going to work). We need to give $\mathcal{V}$ and $\mathcal{P}$, with input φ and k; we want $\mathcal{P}$ to make $\mathcal{V}$ accept when k is the correct number of satisfying assignments.

(0) $\mathcal{P} \to \mathcal{V}$ sends $\#\varphi$ (i.e., the prover sends to the verifier the actual number of satisfying assignments), and $\mathcal{V}$ checks whether $k = \#\varphi$.

In other words, the prover sends over the truly right number, and the verifier checks if that's what's in the input. (There'll be a number of checks; if at any point a check fails, the verifier is going to reject.)

A way to think about this is that the prover sends a claim, the verifier says 'okay, so far so good, but how do I know you're not lying here?' So the prover is going to try to expand this, using the fact we saw above.

(1) $\mathcal{P} \to \mathcal{V}$ sends $\#\varphi_0$ and $\#\varphi_1$, and $\mathcal{V}$ checks that $\#\varphi = \#\varphi_0 + \#\varphi_1$.

To make this concrete, suppose the input is some formula and $k = 100$. And the prover (the honest prover when we're in the language, so this is the correct value) is going to send 100, saying 'yup there's 100 satisfying assignments.' And $\mathcal{V}$ says 'okay; but how do I know that's really right?' And then $\mathcal{P}$ says 'the reason 100 is right is because if I take the first variable and set it to 0 now there's 60 satisfying assignments, and if I set it to 1 now there's 40.' And $\mathcal{V}$ says 'okay, $60 + 40 = 100$, so I'm convinced 100 is right provided that *these* two values are right.'

(2) We continue — $\mathcal{P} \to \mathcal{V}$ sends $\#\varphi_{00}$, $\#\varphi_{01}$, $\#\varphi_{10}$, and $\#\varphi_{11}$, and $\mathcal{V}$ checks that $\#\varphi_0 = \#\varphi_{00} + \#\varphi_{01}$ and $\#\varphi_1 = \#\varphi_{10} + \#\varphi_{11}$.

So we're going one level further here, and the verifier is checking the values they received last time are right (assuming these are).

(3) And so on. If we're following, we should start getting worried — this is starting to look exponential, and we don't like that. But let's go with it anyways.

(m) $\mathcal{P} \to \mathcal{V}$ sends $\#\varphi_{a_1 \dots a_m}$ for all strings $a_1 \dots a_m$ (i.e., all the ways to set *all* the variables), and $\mathcal{V}$ checks that this stage justifies the previous stage — i.e., that

$$\#\varphi_{0^{m-1}} = \#\varphi_{0^m} + \#\varphi_{0^{m-1}1}$$

and so on. (It takes the previous stage where we had the first $m-1$ values preset, and it extends the last value to both a 0 and a 1 and checks, using the value it was given.)

$(m+1)$ At the very end, the verifier says, 'I'm still with you but I'm not convinced.' And now that we have these exponentially many values that the prover sent, it can check itself that these values are correct — so $\mathcal{V}$ *accepts* if $\#\varphi_{0^m} = \varphi_{0^m}$ and so on. In other words, it plugs things directly into the formula (we don't have to be counting anymore since we set all the variables), and it does that for all of these assignments; otherwise it rejects.

The important take-home message is that this protocol works. But its only problem is that it's exponential. So we have to figure out how to fix that, and that's where the big idea comes in. □

§25 December 12, 2023

Today we'll finish our discussion on interactive proofs, which we started discussing last time.

> **Remark 25.1.** There are some sample exams for the final posted on the website. The final exam is the last day of finals week, with the same rules as for the midterm. (The instructions are on the website, as well as on the sample exam.) All the questions appearing on the sample exams have appeared in previous final exams.

§25.1 Review

Recall that we've defined interactive proofs — we have a prover and a verifier, and you can think of the prover as trying to convince the verifier that a certain string is in the language. When the string is actually in the language, there is some prover — the *honest prover* — who should exchange messages with the verifier for some polynomial number of time (where the verifier has a source of randomnes); and after this exchange, the verifier should accept with high probability. And when the string is *not* in the language, there *is* no prover that can make the verifier accept with high probability, no matter what the prover does. So when we're not in the language, *every* prover makes the verifier accept with low probability.

(We'll describe our protocols on an intuitive level; you can map it onto the formal model, but it's good enough to think of it informally, as having a prover interact with a verifier to try to convince the verifier of some information, namely that the string is in the language.)

The big theorem in this area is that $\mathsf{IP} = \mathsf{PSPACE}$ — you can actually do TQBF with an interactive proof. We'll do something a little weaker than that, but still using the same basic idea:

Theorem 25.2

We have $\mathsf{coNP} \subseteq \mathsf{IP}$.

The language we'll work with, which is coNP-hard, is $\#\textsc{Sat} = \{\langle \varphi, k\rangle \mid k = \#\varphi\}$. When we're in the language, the prover should make the verifier accept with high probability; when we're not, no matter what the prover does the verifier should accept with low probability.

We'll want to take a formula and preset some variables to certain values; initially these will be Boolean values, but we'll later use arithmetization and consider non-Boolean values. For example, we might set $x_1 = 1$, $x_2 = 0$, $x_3 = 1$, and so on. We preset the first i variables using the notation $\varphi_{a_1 \dots a_i}$ — for example φ_{101} is φ with x_1 set to 1, x_2 to 0, x_3 set to 1, and all other variables still as variables. This will be a new formula with its own number of satisfying assignments. If we think of the formula itself as evaluating to 0 or 1 (with 1 if the formula was satisfied), then the number of satisfying assignments is obtained by plugging in all possible values and adding up the number of 1's — so we have

$$\#\varphi_{a_1 \dots a_i} = \sum_{a_{i+1}, \dots, a_m \in \{0,1\}} \varphi(a_1 \dots a_m).$$

(The first i variables have been preset, and we consider all the ways to plug in the remaining variables.)

We can use this to make an observation — we have

$$\#\varphi = \#\varphi_0 + \#\varphi_1$$

(since 0 and 1 are the only ways to set x_1). In general, we have

$$\#\varphi_{a_1 \cdots a_i} = \#\varphi_{a_1 \dots a_i 0} + \#\varphi_{a_1 \dots a_i 1}$$

(we can imagine setting the $(i+1)$th variable to either 0 or 1). And lastly, when we preset *all* the variables and ask for the number of satisfying assignments, that's just 0 or 1 depending on whether the formula was satisfied, so it's just the formula itself — we can write this as

$$\#\varphi_{a_1 \dots a_m} = \varphi_{a_1 \dots a_m}.$$

(At this point the formula is not even a formula anymore — it's just true (1) or false (0).)

§25.2 A first pass

Let's do a first pass at the protocol (as described last time) — the exponential protocol for #SAT. We need a protocol where $\mathcal{P}$ convinces $\mathcal{V}$ if k is the correct value. (We'll later think about what happens if k is the wrong value.)

The verifier first sees some formula φ and some k (e.g., 100); at the start, they have no way of telling if that's right. But the prover is going to help.

(0) First, $\mathcal{P} \to \mathcal{V}$ sends $\#\varphi$; $\mathcal{V}$ checks that this matches k, and rejects if not. (The prover is trying to convince $\mathcal{V}$ that the input is correct; so $\mathcal{P}$ says that 100 is the right value, $\mathcal{V}$ confirms that that's what the input says.)

(1) Then $\mathcal{V}$ says, 'okay, but how do I know you're telling me the truth?' And $\mathcal{P}$ says, well, that's because there's 40 satisfying assignments when I set $x_1 = 0$, and 60 when I set $x_1 = 1$. And $\mathcal{V}$ says, well, that's good because $40 + 60 = 100$, but how do I know 40 and 60 are correct?

(2) And then $\mathcal{P}$ goes down one level further — it says 30 and 10 for $\#\varphi_{00}$ and $\#\varphi_{01}$, and 55 and 5 for $\#\varphi_{10}$ and $\#\varphi_{11}$. And so on — at each stage, the prover gives the verifier information that says the previous information was correct if the information at this step was.

(m) $\mathcal{P}$ sends $\mathcal{V}$ the counts for *all* assignments — i.e., $\#\varphi_{00\ldots0}$, $\ldots$, $\#\varphi_{11\ldots1}$. And $\mathcal{V}$ uses this information to confirm the previous stage — it checks that $\#\varphi_{0^{m-1}} = \#\varphi_{0^m} + \#\varphi_{0^{m-1}1}$ (where the left-hand side is from the previous stage, and the right-hand side is the two values it got in this stage), and so on, all the way down to $\#\varphi_{1^{m-1}} = \#\varphi_{1^{m-1}0} + \#\varphi_{1^m}$. So this is the same idea — using the current stage to confirm the previous stage.

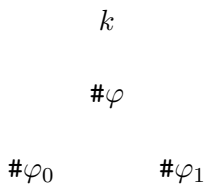
($m+1$) Finally, we need to know that the current stage is correct, but $\mathcal{V}$ can check this directly — $\mathcal{V}$ checks directly that $\#\varphi_{0\ldots0} = \varphi_{0\ldots0}$, and so on, up to $\#\varphi_{1\ldots1} = \varphi_{1\ldots1}$.

If all the checks work out, then the verifier accepts (because they have been convinced).

If $\langle \varphi, k \rangle$ is in the language — i.e., k is the correct value — then the honest prover can send all the correct values all along, and everything will check out. So the probability that the verifier ends up accepting is 1 — it doesn't even need probability here (all of this is deterministic).

> **Remark 25.3.** If we're checking all the assignments at the end, why do we even do the middle steps? We'll see. We're going to show how to take this algorithm and modify it so that it's not exponential, using non-Boolean values; but the algorithm will have the same structure.

We can imagine this protocol in a diagram — we get k from the input. And there's going to be a bunch of equalities that need to be checked — the prover supplies the actual value $\#\varphi$, which the verifier checks is equal to k. And then the verifier checks that that's equal to the sum of $\#\varphi_0$ and $\#\varphi_1$; and then we further expand those, and so on.



(At some point we will transform this into a polynomial algorithm, using a very interesting idea; but for now we're laying the groundwork.)

If k is correct, then the prover will send all the correct values, and all these equalities will work.

If k is wrong, then it's clear this is going to fail, but there's a particular way of thinking about it that's going to be useful. Suppose that k is wrong — i.e., k is *not* equal to the number of satisfying assignments. Then the verifier is going to reject for sure (no matter what the prover does), and we can see this in the following way.

Imagine you're the prover, trying to make the verifier accept. On the first step, if the prover sends the correct value of $\#\varphi$, that's going to disagree with k (because k is wrong), so the verifier is going to immediately reject. So the only way to keep the pretense going is for the prover to send the wrong value here — so if k comes in at 99 instead of the right value 100, then the prover is going to have to say 99.

Now this is a lie. That's going to force at least one of the values on the next level to be lies as well, because you can't have two correct values adding up to the wrong value. (This is like if you're a kid and you lie to your parents, and your parents ask you more questions, then the lie forces other lies.) And so there's going to be a lie propagating itself down until at some point the verifier sees either that the addition is wrong, or that at the end something doesn't match up with what the formula says.

So the verifier is guaranteed to reject.

§25.3 Idea for the fix

Now k is going to come in as before. The prover is going to start out, as before, by sending in $\#\varphi$. But now, instead of justifying that by having *two* values here, the prover is going to justify that with a kind of complicated indirection — it's going to take φ, and instead of plugging in 0 or 1 as a preset, it's going to plug in a *random value from a field* (which is going to be a non-Boolean value). You can kind of imagine that this non-Boolean value has elements of 0 and 1 put together somehow; but the point is that it's just one value.

And to justify this value, it's going to now set the first *two* values to non-Booleans. And that'll continue until we've set all the values to non-Booleans — so we have k, $\#\varphi$, $\#\varphi_{a_1}$, $\#\varphi_{a_1a_2}$, …; and in the end the verifier checks by plugging in the formula. So now instead of a tree, we just have a line.

§25.4 The protocol

First, we're going to do an arithmetization — we turn $x \wedge y$ into xy, $x + y$ into $x + y - xy$, and $\overline{x}$ into $1 - x$. (This is just simulating $\wedge$ and $\vee$ with $+$ and $\times$.)

In particular, then we can take our whole formula φ, and that becomes a polynomial p_φ in $x_1, \ldots, x_m$ (these are the variables of the formula). And importantly, $\deg p_\varphi \leq |\varphi|$. (This is a simple thing to check; when you do $\wedge$ or $\vee$ you add the degrees of the two pieces, since you have multiplication; this increases the degrees linearly.) (Here we can think of $|\varphi|$ ass the number of symbols — i.e., the length of the formula.)

Now $\varphi_{a_1 \ldots a_i} = p_\varphi(a_1, \ldots, a_i)$, with $x_1, \ldots, x_i$ preset — instsead of talking about this polynomial everywhere, we'll talk about the original formula, but plugging in non-Boolean values into the formula. When the a_j's are Boolean, this polynomial is a faithful simulation of the Boolean operations, so they agree. But we can put in non-Boolean values here; so we're extending our formula to operate on non-Boolean values, by looking at what the polynomial does. So we're now thinking about the formula as a Boolean formula but plugging in non-Boolean values; and that's fine because we can just look and see what the polynomial does.

And lastly, when we're counting the number of solutions, we define $\#\varphi_{a_1 \ldots a_i} = \sum_{a_{i+1}, \ldots, a_m \in \{0,1\}} p_\varphi(a_1 \ldots a_m)$, as before. So all we're doing is extending what we did before to non-Boolean values.

> **Remark 25.4.** The tricky thing here is that when we put a non-Boolean value in, the 'number of satisfying assignments' no longer makes sense. But if we want to think about $\#\varphi_2$, for instance (normally we'd only be putting in 0 or 1 for x_1, but now we're assigning it to 2), we can just imagine setting the remaining variables to 0's and 1's in all possible ways, and just adding up the polynomial.

Now here's our polynomial protocol to show that $\#\textsc{Sat} \in \mathsf{IP}$. Suppose we have input $\langle \varphi, k \rangle$; we want to give $\mathcal{V}$ and the honest prover $\mathcal{P}$.

We'll need one more thing. When we're doing this arithmetization, all of the arithmetic is done in a finite field (just like we did with ROBPs) — everything will be done mod some prime q. This time we'll need a larger prime, though — we'll take $q > 2^n$. (We won't worry about how to get this prime; it can be done, and it takes n bits to write down.)

(0) First $\mathcal{P} \to \mathcal{V}$ sends the prover's claim $\#\varphi$ about the number of satisfying assignments; and as before, $\mathcal{V}$ checks that $k = \#\varphi$.

(1) Here's where the magic happens — the idea is short, but a bit of a doozy. Again $\mathcal{P} \to \mathcal{V}$ will send something. Before $\mathcal{P}$ was unravelling by one level, so they'd send $\#\varphi_0$ and $\#\varphi_1$ as two values. We don't want to do that; we're going to send something which looks even worse, but the important thing is that it won't grow.

What it'll do is that it'll send $\#\varphi_z$ as a polynomial in z. What does this mean? If we take $\#\varphi_z$, we can keep z as a variable; and we get this polynomial by plugging in z for x_1, and 0's and 1's for $x_2, \ldots, x_m$. So we'll get a polynomial in the single variable z, which is going to look something like $z^3 + 5z^2 + 2z + 7$ — it's just going to write down this polynomial which is the original polynomial p_φ, but where we plug in z for x_1, and add up all possible ways to plug in 0's and 1's for the remaining values.

Importantly, $\deg \varphi_z$ is itself not too big — it's at most as big as the original polynomial was, so the degree of this is at most n.

Now $\mathcal{V}$ has to check something — they have to check the previous value, and all they have now is this polynomial. Before it had $\#\varphi_0$ and $\#\varphi_1$ and it added them together. But now it has this polynomial instead. And the nice thing about the polynomial is that you can plug in values into it — in particular, you can plug in 0 and 1, and you can see what $\#\varphi_0$ is and what $\#\varphi_1$ is by evaluating the polynomial.

So $\mathcal{V}$ checks that $\#\varphi = \#\varphi_0 + \#\varphi_1$, and it gets these values by evaluating $\#\varphi_z$. The nice thing is that we now have just one object capturing the information we needed to check the previous stage — it's a more complicated object, but it's just one object.

(2) Now $\mathcal{V}$ is going to do something that was absent from the previous protocol, in that $\mathcal{V}$ gets to speak. In the exponential protocol we only had messages from $\mathcal{P}$, but now $\mathcal{V}$ gets to say something.

What $\mathcal{V}$ wants to do is be convinced that this polynomial is the right polynomial — $\mathcal{P}$ could have just sent a bogus polynomial that works with the previous stage. There *is* a right polynomial, namely $\#\varphi_z$, but the prover might be lying.

So what the verifier does is that they evaluate this polynomial at a random point, and say 'convince me that this value is right' — $\mathcal{V} \to \mathcal{P}$ sends a random $r_1 \in \mathbb{F}_q$, and $\mathcal{P}$ needs to show that $\#\varphi_{r_1}$ is correct.

Here's the whole thing — if $\#\varphi_z$ is a lie (it's a different polynomial than the true one), then those two polynomials can only agree on a small number of places. So when $\mathcal{V}$ picks a random place, the probability that it's going to be an agreement is low. And so now the prover's probably going to show $\#\varphi_{r_1}$ is correct, even though it's probably *not* the right value.

(2) Now the protocol continues in the same way — $\mathcal{P} \to \mathcal{V}$ sends $\#\varphi_{r_1 z}$ where they now plug in r_1 and put z into the *next* value, as a polynomial. And $\mathcal{V}$ checks that $\#\varphi_{r_1} = \#\varphi_{r_1 0} + \#\varphi_{r_1 1}$, where it gets these two values by evaluating the given polynomial.

And then as before, $\mathcal{V} \to \mathcal{P}$ sends a random r_2l

And this continues.

(m) $\mathcal{P} \to \mathcal{V}$ sends $\#\varphi_{r_1 \dots r_{m-1} z}$ (again, as a polynomial in z); and $\mathcal{V}$ checks that the previous stage was right — i.e., that $\#\varphi_{r_1 \dots r_{m-1}} = \#\varphi_{r_1 \dots r_{m-1} 0} + \#\varphi_{r_1 \dots r_{m-1} 1}$ (again, by evaluating the polynomial it just received at 0 and 1).

And for the last time, $\mathcal{V}$ picks a random $r_m \in \mathbb{F}_q$ and sends it to $\mathcal{P}$.

($m+1$) $\mathcal{V}$ checks that $\#\varphi_{r_1 \dots r_m} = \varphi_{r_1 \dots r_m}$.

And that's the whole protocol.

§25.5 Analysis

Imagine looking at this protocol from the perspective of our linear diagram; if we can understand the first step, the rest is just the same over and over again.

On step 0 we have $\#\varphi$; on step 1 we have $\#\varphi_{r_1}$. The prover is trying to show that $\#\varphi$ is correct; and instead of unpacking this into both 0 and 1, it expresses the whole function capturing $\#\varphi_0$, $\#\varphi_1$, $\#\varphi_2$, ... as a polynomial; and now the verifier can recover the two desired values by evaluating that polynomial.

So that's how the honest protocol works.

What happens if k is wrong? Let's imagine that k is the incorrect value. (This is where all the algebra comes in — because we're again using the Schwarz–Zippel lemma, but here we only need to do so for single variable polynomials, so we can use the simple bound.)

Suppose that k is wrong. We want $\mathcal{V}$ to end up rejecting, no matter what the prover does.

Initially, let's say k is 99 instead of the correct value 100. On step 0 the prover had better send 99 as its claimed value of the number of satisfying assignments, or else the verifier will reject, just as before. So the prover lies by saying 99.

And the verifier says, how do I know 99 is right? What the prover *wants* to send is the number of satisfying assignments with the first variable set to 0 and to 1; but the prover's not going to send 40 and 60. Instead it's going to send a whole function — even more information. But it's not sending that whole function as a table (saying $\varphi(0) = 40$, $\varphi(1) = 60$, $\varphi(2) = 73$, ...); instead it represents that whole function as a polynomial, and it gets that polynomial by plugging z into the original polynonmial.

So the prover needs to send a polynomial representing $\#\varphi_z$ for any z. This is just as good for the verifier, since the verifier can plug in $z = 0$ and $z = 1$ themselves, do the addition, and make sure it works out.

But the prover now has a harder job — they don't just have to justify that $\#\varphi_0$ and $\#\varphi_1$ are correct, but that this entire polynomial is correct.

So the verifier evaluates this polynomial at a random place r_1. And if the prover sent a bogus polynomial, it's almost certainly going to disagree — two polynomials can't agree very often, so if we pick a random place, they'll almost certainly e in disagreement. So if k is wrong then $\#\varphi$ is wrong for sure, and then $\#\varphi_{r_1}$ is probably wrong.

Is there a possibility that the prover can get lucky, and convince the verifier to accept when it shouldn't? How could that happen? It could be that the incorrect polynomial that the prover has just sent over, and the correct polynomial (the actual polynomial you get by using the arithmetization and putting in z) do agree. And if the verifier happens to pick a place where they agree, then the prover says, *a miracle, I'm saved!* (Imagine you're totally lost in this course and you decide, well, I'm just going to study one theorem and that's it. And miraculously that's the theorem on the final exam. That's what would be happening here — we just happen to probe in the place where the prover's polynomial agrees with the actual polynomial.) Then the protocol could continue on as if we had a honest prover, since now the prover is trying to demonstrate that something's true when it actually is true.

So if there's one point at which the prover gets lucky, then the prover is going to be able to make the verifier accept. But the point is that that's a very rare possibility. It could happen at each one of the n steps of the protocol, so we have to make sure the probability of it happening at any one of the steps is much less than $\frac{1}{n}$. But it's actually something like $\frac{1}{2^n}$; so even though there's m chances for it to happen, this'll still be exponentially low probability.

Remark 25.5. How did we define the polynomial $\#\varphi_z$? We define

$$\#\varphi_{a_1 \dots a_i} = \sum_{a_{i+1},\dots,a_m \in \{0,1\}} p_\varphi(a_1, \dots, a_m).$$

So we take our preset a_i's, and then we consider all ways to plug in the remaining variables as 0's and 1's, and we add up all the possibilities. And now we do the exact same thing, but using the arithmetization instead — so now we're going to plug in non-Boolean values (e.g. $a_1 = 7$, $a_2 = 2$, $a_3 = 5$) for the first i values. The remaining values are still set to 0's and 1's. Why? Because we want our two definitions to agree on Booleans — we need a legitimate simulation of the Boolean case.

Suppose we just want $\#\varphi_z$ (where we're setting the first 1 value). So now we're setting $a_2, \dots, a_m$ to 0's and 1's in all possible ways. And then we're taking $\varphi(a_1 \dots a_m)$ (the polynomial we got from this arithmetization) and adding up all these values.

And that's our polynomial. If we plug in $z = 0$ then we get $\#\varphi_0$, and if we plug in $z = 1$ then we get $\#\varphi_1$. But now we have a polynomial in z representing not just 0 and 1, but everything.

So that's how you get the polynomial; and that's what happens with the other values. ($a_2, \dots, a_m$ go away, since they're set to 0 and 1 in all possible ways; so we have a single variable left, z, which went into the x_1-slot.)

That's what happens in the very first stage — $\mathcal{P}$ sends $\#\varphi_z$ as a polynomial in z, and $\mathcal{V}$ checks that plugging in 0 and 1, this agrees with the previous stage.

Then $\mathcal{V}$ evaluates this polynomial at some random r_1, and so on.

Suppose $\langle \varphi, k \rangle \notin \#\text{SAT}$, so k is wrong. That implies that $\#\varphi$ is wrong (assuming $\mathcal{V}$ doesn't catch $\mathcal{P}$ right away — then if k is wrong, the prover has to send the wrong claim). So that implies that $\#\varphi_0$ or $\#\varphi_1$ is wrong, which implies that $\#\varphi_z$ is the wrong polynomial — i.e., $\widetilde{\#\varphi_z}$ (we'll put tildes on to emphasize that these are the wrong values) is different from the correct polynomial $\#\varphi_z$. So this polynomial is guaranteed to be wrong, or else the verifier would have already caught the prover.

Now for random r_1, the probability that $\widetilde{\#\varphi_{r_1}} = \#\varphi_{r_1}$ is low — what's the probability that we have two different polynomials that agree at r_1? That's at most the number of roots of their difference, which is at most $d/q \approx n/2^n$ (where d is the degree). This is a very small value. So the probability that we had the wrong polynomial, but evaluated at r_1 it gave the right value, is

$$\mathbb{P}[\widetilde{\#\varphi} \text{ wrong and } \#\varphi_{r_1} \text{ correct}] \le \frac{n}{2^n}.$$

And that'll be the same at every stage —

$$\mathbb{P}[\widetilde{\#\varphi_{r_1 \dots r_i}} \text{ wrong and } \#\varphi_{r_1 \dots r_{i+1}} \text{ correct}] \le \frac{n}{2^n}.$$

So all together, we have

$$\mathbb{P}[k \text{ is wrong and } \#\varphi_{r_1 \dots r_m} \text{ correct}] \le n \cdot \frac{n}{2^n} < \frac{1}{3}.$$

Remark 25.6. Why did we use $q = 2^n$ — all we need is n^2?

There's another reason we'd like a large value of 2^n — potentially k could be a value between 0 and 2^m, and if our field is too small we could have two of these colliding to the same value. So we need to take $q > 2^m$, where m is the number of variables; this means we might as well take 2^n.

Remark 25.7. We don't know what the actual correct polynomial $\#\varphi_z$ is; all we need to know is that there is an actual one. What's happening at every stage is that if the prover starts out lying, it'll be forced to lie and lie and lie, but there's a small chance that the prover'll get away with it. At every step the prover is sending some polynomial, which is either a wrong polynomial or the right polynomial. If it's the wrong polynomial, evaluating it at a random place is probably going to be different from the right polynomial. We don't need to know what the right polynomial is; we just need to know there *is* one. We don't know what the right polynomial says at that point.

In the end, how do we catch the prover for lying? At every stage, a lie is going to cause the next stage to be a lie, with a small chance the prover gets away with it. So we have a lie forcing another lie, and so on. We don't yet know this is a lie; the verifier is just going along, and the prover keeps sending polynomials. We don't know whether they're right or not until the end.

In the end, we have $\#\varphi_{r_1 \dots r_m}$ by following along the protocol. And that the verifier can check by itself — it just takes the polynomial, plugs in $r_1, \dots, r_m$, and sees what it gets. If the prover was lying, then almost certainly these won't agree. And that's the point the verifier catches the prover — at the end. (This is just like what happened in the exponential protocol. The prover could be sending bogus values all along, but the verifier doesn't know until the end.)

In between, the verifier isn't checking that we have the 'right polynomial'. It's just checking that Stage 3 justifies Stage 2 — i.e., Stage 2 is correct *if* Stage 3 is correct. And so on; finally Stage $m+1$ is where it actually finds out the truth, since at that point it can check by itself. (There's no checking of actual truth along the way, just local consistency.)

Remark 25.8. How do we find q in polynomial time (where q is the prime)?

Using the fact that primality testing is in P and the primes are reasonably dense, you can find it by randomly sampling.

But an even simpler answer is that the prover can send over q (if you want, even together with a proof that q is prime — it's a (number theory) fact that primality is in NP).

Remark 25.9. On the stage $m + 1$, $\mathcal{V}$ has to do the arithmetization to confirm that $\#\varphi_{r_1 \dots r_m}$ equals $\varphi_{r_1 \dots r_m}$. So here they're actually plugging in these values.