Introduction to Computability and Complexity Theory

Robert Robere

April 9, 2016

Contents

1 Foundations and Computability

1.1 Lecture 1: Turing Machines and the Question of Computation

Textbook Readings: Section 3.1

This is a course about *computation*. What is computation? It is a fun exercise to try and answer this question in a single sentence — the best answer that I have been able to come up with is

Computation is the process of transforming information from one form to another by a finite sequence of operations.

The salient features of computation (if it is "unbounded" in terms of the amount of computational resources used) are as follows:

1. Access to a unlimited "memory", which can be in one of a number of potentially unbounded "configurations".

2. A "transformation method" which receives as input a "bounded" amount of information about the current configuration, and uses this bounded amount of information to locally transform the configuration into a new configuration.

While the concept of an algorithm can arguably be traced back to Euclid or even earlier (e.g. the Euclidean algorithm for computing greatest common divisors), the modern study of computability (with the "correct" formal models) has its roots in two problems of David Hilbert. The first, *Hilbert's Tenth Problem*, asks for an "effective procedure" that can determine whether or not a given multivariable polynomial with integral coefficients has an integral root; the second, his *Entscheidungsproblem*, asks for a procedure that receives as input a statement in first-order logic along with a finite list of axioms and outputs *true* or *false* depending on whether or not the statement can be formally deduced from the axioms.

The Entscheidungsproblem was resolved independently (in the negative!) by both Alonzo Church and Alan Turing in 1936 by completely distinct methods. Church introduced a model of computation known as λ-*calculus* — which now forms the basis of modern *functional* programming languages — while Turing introduced his *Turing Machines*, which are intuitively more similar to modern *procedural* programming languages. Both of their respective models were shown to not be able to compute the Entscheidungsproblem, and were later proved to be *equivalent* in power (as well as being equivalent in power to numerous other models of computation). The fact that they were equivalent, as well as being equivalent to a number of other distinct models of computation, led to the general acceptance in the scientific community that they got it right, and the connection between informal algorithms and these definitions has come to being called the *Church-Turing Thesis*.

Church-Turing Thesis: Anything which can be computed by an "algorithm" can be computed by a Turing Machine.

But what is a Turing Machine? Let us develop it from the salient features that we outlined before. The *memory* of our machine will be represented by an unlimited amount of *tape*, which is divided into individual tape squares. Initially, the machine has an *input* written on the tape, represented as a sequence of symbols each chosen from a collection of symbols Σ called the *input alphabet*. Each tape square is labelled with a symbol chosen from some finite collection of symbols Γ called the *tape alphabet*. The sequence of symbols on the tape is modified by the use of a *read-write head*, which operates according to a finite *program* stored as a *state machine*. In a single step, the machine:

1. Reads a symbol $\alpha \in \Gamma$ on the tape at the location pointed to by the tape head.

2. Using the symbol α, and the current state q of its program, it writes a new symbol α' to the tape, changes the state q to a new state q', and moves the head one square to the left or the right.

The machine's program contains two "special states": q_{acc} and q_{rej}, which corresponding to halting and accepting the input or halting and rejecting the input, respectively. If the machine accepts the input, the *output* of the machine is the sequence of symbols left on the tape when the machine halts.

Note that in a single step of computation the machine receives a bounded amount of information — a symbol α from the finite tape alphabet Γ, and the current state q of the machine's finite program — and changes the memory and state configuration of the machine "locally" depending on this information. It is very important that the amount of work performed in a single step is bounded in this sense: otherwise, the machine could perform unrealistic amounts of work in a single time step.This, of course, would not capture what we intuitively believe to be computable.

Turing Machine	**Stored-Memory Computer Program**
Tape	Memory
Current Position of Read-Write Head	Current Memory Block
State Machine	Program
Current State	Program Counter
Tape Alphabet	Low-level Binary Encoding

Figure 1: Turing Machine vs. "Typical" Program Executed on a Computer

Example. Recall that if A is a set then the set A^* (pronouned "A-star") is the infinite set

$$A^* = \{a_1 a_2 \cdots a_n \mid n \in \mathbb{N} \text{ and } a_i \in A \text{ for all } 1 \leq i \leq n\}$$

of all possible strings of symbols chosen from A. Consider the collection of strings

$$L = \{w\#w \mid w \in \{0,1\}^*\}.$$

We give a Turing Machine M which, given as input any string x in $\{0,1,\#\}^*$, halts and accepts x if and only if $x \in L$. Our treatment right now is rough, and follows the treatment given by Sipser [1].

The "program" of our machine M will operate as follows. We assume the input x is written on the tape, and the read-write head is initially over the left-most symbol of x. All other symbols on the tape are assumed to be a special $\sqcup$ "blank" symbol. The program is defined as follows.

1. Move the head to the right until a $\#$ symbol or a $\sqcup$ symbol is found.

2. If no $\#$ is found, halt and **reject**.

3. Return to the beginning of the string; scan forward to the first symbol which is not marked before the $\#$ and mark it. If no such symbol exists, goto (5).

4. Move the head to the first unmarked symbol after $\#$ and mark it. If no such symbol exists, **reject**. Otherwise goto (1).

5. Move the head to the first unmarked symbol after $\#$; if it exists **reject**, otherwise **accept**.

It should be fairly easy to convince yourself that this machine does accept only the collection of strings L, and it halts and rejects on all other inputs. This example also illustrates how it is important to differentiate between the *input alphabet* (in this example it is $\{0, 1, \#\}$) and the *tape alphabet* (in this example $\{0, 1, \#, \sqcup, 0', 1'\}$, with the $0'$ and the $1'$ representing the marked symbols).

Next we give a mathematical definition of a Turing Machine following Sipser [1, Definition 3.1]. It is a bit unwieldy, and in practice we will construct Turing Machines in the high-level language that we used in the previous example. However, it is important to note that the high-level language is just a shorthand for the complete definition of a machine, and at any point a reasonably patient person could convert the high level description into a low-level description. (This is much like the method by which we write and evaluate proofs).

Definition 1.1. A *Turing Machine* (TM) is a 7-tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{acc}, q_{rej})$ where:

1. Q is a finite set of *states*.

2. Σ is a finite set of symbols called the *input alphabet*.

3. $\Gamma \supseteq \Sigma$ is a finite set of symbols called the *tape alphabet*, which contains a special "blank" symbol $\sqcup \notin \Sigma$.

4. $\delta \colon Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is a *transition function*, which takes as input a state q and a tape symbol $\gamma \in \Gamma$ and outputs a triple $\delta(q, \gamma) = (q', \gamma', D)$ where q' is a new state, γ' is a new tape symbol, and D is the direction that the head moves.

5. $q_0 \in Q$ is the *start state*.

6. $q_{acc} \in Q$ is the *accept state*.

7. $q_{rej} \in Q$, distinct from q_{acc}, is the *reject state*.

Let M be a TM, and next we formalize what it means for a TM to compute something. First, the machine M receives a string $x \in \Sigma^*$, its "input", with the read-write head pointing at the first symbol of the string x. At any point in the computation the machine M has a *configuration*, which records the current sequence of symbols on the tape, the position of the read-write head, and the current state. We represent a configuration C as a string of symbols like so:

$$C = \gamma_1 \gamma_2 \cdots \gamma_{i-1} q \gamma_i \cdots \gamma_n,$$

where the string $\gamma_1\gamma_2\cdots\gamma_n$ is the sequence of the symbols currently on the tape, and the position of the symbol q denotes that the read-write head is pointing at symbol i and the machine's program is in state q. The infinite sequence of blank symbols is omitted. We often write a configuration C as yqz where $y, z \in \Gamma^*$ and yz is the string on the tape. What is the initial configuration of a Turing Machine on input $x \in \Sigma^*$?

Suppose that $y, z \in \Gamma^*$, $\alpha, \beta, \gamma \in \Gamma$, and $q, q' \in Q$. Let C be the configuration $y\alpha q\beta z$ (note that $y\alpha\beta z$ is the string encoded on the tape at this time). Let $\delta(q, \beta) = (q', \gamma, D)$. If $D = L$, then C *yields* the configuration $yq'\alpha\gamma z$. Intuitively, the machine has replaced the symbol β under the read-write head with γ, changed its internal state to q', and moved to the left a single step. If $D = R$, then C *yields* the configuration $y\alpha\gamma q' z$. Now, the machine has replaced the symbol and changed its state like before, but it has moved the read-write head to the right.

If the read-write head is at one of the ends of the configuration we have some special cases. Suppose the configuration is of the form $q\beta z$. The read-write head is at the left end of the tape, and so if $D = L$ then $q\beta z$ yields the configuration $q'\gamma z$, since it cannot move off the left side of the tape. If $D = R$ then $q\beta z$ yields $\gamma q' z$.

Finally, suppose that the configuration is of the form $y\alpha q$. This configuration is equivalent to $y\alpha q\sqcup$ since the read-write head is pointing at the blank symbol at the right end of the string on the tape. If $D = L$, then $y\alpha q = y\alpha q\sqcup$ yields $yq'\alpha\gamma$; if $D = R$ then $y\alpha q$ yields $y\alpha\gamma q$.

Now we can define what it means for a Turing Machine to compute. A sequence of configurations $C_0, C_1, \ldots, C_m$ is a *halting computation* of M on input x if C_0 is the initial configuration, C_i yields C_{i+1} for all $i = 0, 1, \ldots, m-1$, and in C_m the machine is in the accept state or the reject state. The machine M *accepts* the input x if C_m is in an accept state, and otherwise it *rejects* x. If M accepts input x, the *output* of the machine M is the string of symbols written on the tape when it halts. Note that a Turing Machine does not need to halt on every input — it could loop forever.

A subset $L \subseteq \Sigma^*$ will be called a *language*; a language L is *semi-decidable* (or *recognizable*) if $L = \mathcal{L}(M)$ for some TM M. The *language computed by M* (or, equivalently, *the language recognized by M*) is defined to be

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

A language L is *decidable* if there is a machine M such that $\mathcal{L}(M) = L$ and M halts on all inputs in Σ^*. We let D denote the class of all languages that are decidable and we let SD denote the class of all languages that are semi-decidable. It is immediate from the definitions that $\mathsf{D} \subseteq \mathsf{SD}$; we will eventually show that $\mathsf{D} \subsetneq \mathsf{SD}$.

1.2 Lecture 2: Multi-Tape TMs, Universality and Computability

Textbook Readings: Section 3.2, 3.3

Anything that can be computed, can be computed by a Turing Machine. This, the *Church-Turing thesis*, is the basic assumption of Computer Science. From here on we take this as a given — whenever we discuss an *algorithm*, we may implicitly replace this with a *Turing Machine* and vice-versa. The Church-Turing thesis is powerful precisely because we no longer have to argue about what computation *is*, and so we are free to study computation as a phenomenon in and of itself. This allows us to replace other "fuzzy" questions with "precise" questions: for example, the fuzzy question

Can *everything* be computed?

can be replaced with the precise question

Is there an alphabet Σ and a language $L \subseteq \Sigma^*$ such that no Turing Machine recognizes L?

In this lecture we develop a major tool (*Universal Turing Machines*) that will allow us to attack this question. Along the way we define an important variant of the Turing Machine (the *multi-tape Turing Machine*), which can be used to give extremely efficient implementations of Universal Turing Machines. It is because of this that multi-tape Turing Machines are typically used as a model to define *efficient* computation (i.e. Complexity Theory). Finally, using Universal Turing Machines we will be able to answer our question about an "uncomputable" language above.

However, let us first talk about what the Church-Turing thesis means from an "algorithm designer" standpoint. We are used to describing algorithms at a fairly high level — say, using some kind of *pseudocode* — where some set of "basic" operations is taken as unit cost. For example: perhaps we allow basic arithmetic operations, random access to array indices, and the ability to move the program counter arbitrarily depending on some conditional statement, all at unit cost. It is clear that these operations are distinctly more powerful then the set of operations offered by a Turing Machine; however, it is also clear that requiring everyone to implement Turing Machines for all of their algorithms would do significantly more harm than good. To resolve this problem, we will take the view that the Church-Turing thesis is a *prescription*: in principle, any algorithm that we write *must* be able to be converted into a Turing Machine computing the same function under a reasonable encoding of the inputs and outputs.

We also note that the Church-Turing thesis does not say anything about efficiency: for example, algorithms textbooks often take for granted that we can multiply two numbers in $O(1)$ time, when in fact the best algorithms[1] known for integer multiplication that can be implemented on a multi-tape Turing Machine run in time roughly $O(n \log(n) \log\log(n))$ when given two n-bit numbers.

[1] We can actually do slightly better, but the running time of this algorithm is easier to state! Improving integer multiplication algorithms further is an interesting open problem.

There is another, somewhat sillier question. Suppose we actually want to give a formal definition of a Turing Machine M that recognizes some language L. What is the best way of writing it down? Well, we could record a table of states, the input and tape alphabet, and a (potentially giant) table recording all possible moves of the transition function δ. This would take far too much time and, what is worse, it is not easy for humans to verify that an algorithm presented in this form is actually correct.

Alternatively, we could go in the opposite direction: write down a "pseudocode" or "high-level algorithm" for Turing Machines that describe the algorithm in English at a very intuitive level (in other words, we could apply the Church-Turing thesis). This often allows us to quickly verify that a given problem is indeed computable or not computable. For example: suppose that we wanted to construct a Turing Machine M that always halts and accepts all strings in the set $PAL = \left\{xx^R \mid x \in \Sigma^*\right\}$ of palindromes, where x^R represents the string x reversed. The following would be a reasonable "high-level" algorithm: iteratively match up the first and last characters of the string until you reach the middle. The problem with this kind of description is that it is too loose: it does not describe, for example, what happens if the length of the string is odd, or exactly the mechanism by which the characters are "matched up".

To avoid both of these problems, we will try and adhere to an approach that is in the middle of these two extremes. When describing Turing Machines, we will describe the tape alphabet Γ, how new symbols introduced in the tape alphabet Γ are used during the algorithm, and also a detailed description of how the read-write head moves back and forth and edits the tape. We consider ourselves to have succeeded if it is clear that all the "edge cases" have been essentially covered by the algorithm, and also if it is reasonably clear (although it may be tedious) to convert the description of the algorithm into a collection of states Q and a transition function δ implementing the same Turing Machine.

With that, let us now describe *multi-tape Turing Machines*, which are Turing Machines with more than one tape and read-write head, each of which can move and edit simultaneously.

Definition 1.2. Let k be a positive integer. A *k-tape Turing Machine* is a 7-tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{acc}, q_{rej})$ where:

1. Q is a finite set of *states*.

2. Σ is a finite set of symbols called the *input alphabet*.

3. $\Gamma \supseteq \Sigma$ is a finite set of symbols called the *tape alphabet*, which contains a special "blank" symbol $\sqcup \notin \Sigma$.

4. $\delta \colon Q \times \Gamma^k \to Q \times \Gamma^k \times \{L, R\}^k$ is a *transition function*, which takes as input a state q and k tape symbols $\gamma_1, \gamma_2, \ldots, \gamma_k$ and outputs a $2k+1$-tuple

$$\delta(q, \gamma_1, \ldots, \gamma_k) = (q', \gamma_1', \gamma_2', \ldots, \gamma_k', D_1, D_2, \ldots, D_k),$$

where q' is a new state, $(\gamma_1', \ldots, \gamma_k')$ are the new tape symbols recorded on the k tapes, and D_i is the direction that the ith head moves for each $i = 1, 2, \ldots, k$.

5. $q_0 \in Q$ is the *start state*.

6. $q_{acc} \in Q$ is the *accept state*.

7. $q_{rej} \in Q$, distinct from q_{acc}, is the *reject state*.

We leave the definition of a multi-tape TM configuration and the corresponding "yields" relation as an exercise (just follow the single-tape example!). Clearly a single-tape Turing machine is just a 1-tape Turing Machine; the following (surprising!) result shows that adding more tapes does not allow us to compute more things. (We state the result without proof, see [1] for the standard construction).

Theorem 1.3. *For any multi-tape Turing Machine M there is a single tape Turing Machine M' such that $\mathcal{L}(M') = \mathcal{L}(M)$ and for all inputs $x \in \Sigma^*$, M halts on x if and only if M' halts on x. Moreover, if the machine M halts after at most $T(n)$ steps on all inputs of length n, then the machine M' halts after at most $O(T(n)^2)$ steps on all inputs of length n.*

The previous theorem shows that multi-tape Turing Machines are equivalent in power to single-tape Turing Machines; thus, in terms of absolute computational *power* the only difference between them is in the increased efficiency of the algorithms they can implement. In principle, multi-tape Turing Machines are useful primarily for two reasons:

1. They are more efficient than single-tape TMs, and are thus used as a model for efficient computation.

2. They are, in practice, much easier to program: the input is initially written on a single tape, and the other tapes can be used for free.

We will use multi-tape Turing Machines as our base model of efficient computation when we begin to study complexity theory. They also can be used to provide particularly efficient versions of Universal Turing Machines (cf. Definition 1.4), which we discuss next.

Definition 1.4. A *Universal Turing Machine* is a Turing Machine M which, when given as input any encoding $\langle M' \rangle$ of a Turing Machine M' and a string $w \in \Sigma^*$, simulates the operation of M' on the input w, and then halts and accepts (rejects) $(\langle M' \rangle, w)$ if and only if M halts and accepts (rejects) w.

Before we state the main theorem of Universal Turing Machines, we briefly discuss the role of *encoding* objects. In Computer Science there is constant tension between the mathematical objects that we manipulate in proofs, and the concrete encodings of those objects that we must operate on in algorithms. It should be well impressed on you by now that good encodings of mathematical objects (i.e. *efficient data structures*) can make or break an efficient algorithm, and in some sense the study of algorithms and the study of data structures are the same.

In this course we will often be interested in algorithms (equivalently, Turing Machines) that operate on encodings of a variety of objects. However, since we will not be "programming", per say, we will usually not be worried about the details of such encodings. As such, we introduce a new notation: if O is some mathematical object (say, a Turing Machine, a graph, or a natural number — anything!) and it is "relatively straightforward" to define a reasonable encoding of O, then we use $\langle O\rangle$ to denote the encoding of O in some fixed alphabet Σ (cf. Definition 1.4).

To illustrate, we give a detailed encoding of a Turing Machine $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{acc}, q_{rej})$ in the alphabet $\{0, 1, \#\}$ (note that our encoding may be different from the encoding in [1]). If $i \in \mathbb{N}$ is a non-negative integer we use $\langle i\rangle$ to denote the standard binary encoding of i in $\{0, 1\}$. Our encoding of M will be broken up like this:

$$\#\#\langle Q\rangle\#\#\langle\Sigma\rangle\#\#\langle\Gamma\rangle\#\#\langle\delta\rangle\#\#\langle q_0\rangle\#\#\langle q_{acc}\rangle\#\#\langle q_{rej}\rangle\#\#,$$

where we define the encodings of each of the sub-objects of M next. In fact, we will use the simplest possible encoding of each object. Writing $Q = \{q_0, q_1, \dots, q_n\}$, we let $\langle q_i\rangle = \langle i\rangle$ for each $i = 1, 2, \dots, n$. Then, we write $\langle Q\rangle = \langle q_0\rangle\#\langle q_1\rangle\#\cdots\#\langle q_n\rangle$. We similarly encode Σ and Γ by identifying each symbol σ in Σ and Γ with a positive integer i, and then letting $\langle\sigma\rangle = \langle i\rangle$, and we do this in such a way so that the encoding of $\langle\Sigma \cap \Gamma\rangle$ is consistent with the encodings $\langle\Sigma\rangle$ and $\langle\Gamma\rangle$. Finally, we represent the transition function δ as a list of tuples. For any $(q, \gamma) \in Q \times \Gamma$, let $\delta(q, \gamma) = (q', \gamma', D)$. We represent this single tuple of δ as the list

$$\#\langle q\rangle\#\langle\gamma\rangle\#\langle q'\rangle\#\langle\gamma'\rangle\#\langle D\rangle\#,$$

and concatenate each element of the list together. Substituting the encodings of each sub-object of M back into the encoding gives us the encoding of M.

This encoding is natural, but it was tedious to define. It should not be hard to convince yourself that it is easy to efficiently recognize whether or not a given string x is a valid encoding of a machine $\langle M\rangle$, and we will often be interested in languages L defined on encodings of Turing Machines M. However, in the future, when we need to encode a mathematical object O we will simply assume that such an encoding $\langle O\rangle$ is available, *if* it is reasonable to assume so. If it is *not* reasonable to assume so, then we will describe the encoding of the object in question. With this, let us return to discussing Universal Turing Machines:

Theorem 1.5. *There is a Universal Turing Machine. Moreover, there is a multi-tape, universal Turing Machine $\mathcal{U}$ such that the following holds: for any Turing Machine M and positive integer n, if M halts in at most $T(n)$ steps on inputs of length n, then U halts in at most $O(T(n)\log T(n))$ steps when simulating M on any input of length n.*

Universal Turing Machines are an important tool used in computability theory, and they also are the theoretical basis for modern programming language interpreters. In the rest of the lecture we will use Universal Turing Machines to explore the question we stated at the beginning: are there problems (i.e. languages) that *cannot* be computed by any Turing Machine? (For those familiar with set theory: is there a quick way to answer this question?) Fix an alphabet Σ, and recall the definition of the classes D and SD:

$$\mathsf{D} = \{L \subseteq \Sigma^* \mid \text{There is a TM } M \text{ such that } \mathcal{L}(M) = L \text{ and } M \text{ always halts}\}$$
$$\mathsf{SD} = \{L \subseteq \Sigma^* \mid \text{There is a TM } M \text{ such that } \mathcal{L}(M) = L\}\,.$$

Clearly $\mathsf{D} \subseteq \mathsf{SD}$, and the only difference between them is that the witnessing Turing Machine in the class SD may not necessarily halt on all of its inputs (although it will always halt and accept if the input is in the language). There is certainly an argument to be made about whether or not these classes are *realistic*, but there is no argument to be made about whether or not these two classes reasonably define the *largest classes of languages that can be computed*, under the Church-Turing thesis. Using D and SD, we can further refine our earlier question:

1. Does $\mathsf{D} = \mathsf{SD}$?

2. Are all languages in D? If not, are all languages in SD?

The answer to each of these questions is an emphatic NO. In fact, $\mathsf{D} \neq \mathsf{SD}$, and SD does not contain all languages. Furthermore, in some sense one can say that *most* languages are not computable[2].

Let us be more concrete. Fix an alphabet Σ, and recall that $\langle M \rangle$ represents an encoding of a Turing Machine M in the alphabet Σ.

Definition 1.6. The *diagonal language* is defined to be

$$\mathrm{DIAG} = \{\langle M \rangle \mid M \text{ is a Turing Machine and } \langle M \rangle \notin \mathcal{L}(M)\}\,.$$

It may take a moment to swallow the definition of the diagonal language — it consists of all encodings of Turing Machines which, when given an encoding of themselves as input, do not halt and accept. The key question to ask is: is there a Turing Machine which computes DIAG?

Theorem 1.7. *There is no Turing Machine which computes* DIAG *(i.e.* $\mathrm{DIAG} \notin \mathsf{SD}$*).*

Proof. Suppose that M is a Turing Machine that computes DIAG — so $\mathcal{L}(M) = \mathrm{DIAG}$ — and suppose that we give the machine M its own encoding $\langle M \rangle$ as input. If $\langle M \rangle \in \mathcal{L}(M)$, then by definition $\langle M \rangle \notin \mathrm{DIAG}$, and so it follows that $\langle M \rangle \notin \mathcal{L}(M)$, a contradiction. On the other hand, if $\langle M \rangle \notin \mathcal{L}(M)$, then it similarly follows that $\langle M \rangle \in \mathrm{DIAG}$, and so $\langle M \rangle \in \mathcal{L}(M)$, another contradiction. $\square$

The reason that DIAG gets its name is from Theorem 1.7, which can be viewed as a "diagonalization" argument. To see this, suppose that we made an infinite table: we label the rows of the table with $\langle M \rangle$ — all the encodings of the Turing Machines — and we label the columns of the table with M, the actual Turing Machines. At the entry of the table corresponding to the row labelled with $\langle M \rangle$ and the column labelled with M', we place a 1 if $\langle M \rangle \in M'$, and a 0 otherwise. Notice that in this (infinitely large) table, the columns of this table enumerate the outputs of each Turing Machine when given as input any encoding of a Turing Machine. With this construction, a way to view Theorem 1.7 is as constructing DIAG by looking at the diagonal of the table and flipping the values of each entry. Since we have enumerated all Turing

[2]In the sense that the set of all languages over a fixed finite alphabet Σ is an uncountable set, the set of languages in SD is a countable set, and from basic set theory we have $\aleph_0 < 2^{\aleph_0}$.

Machines in the construction of the table, the language DIAG must differ from the language of each Turing Machine in at least one entry, and so no Turing Machine can compute it.

This is a profound result, but it suffers because DIAG is, in some sense, an "artificial" language in that it was constructed precisely to not be computable. Our next theorem gives a language which is more natural (and the existence of a Universal Turing Machine is key to its definition). The theorem also illustrates a key technique (the *reduction*) which will be used repeatedly throughout the course.

Definition 1.8. The *acceptance language for Turing Machines* is defined to be

$$A_{\mathrm{TM}} = \{\langle\langle M\rangle, w\rangle \mid M \text{ is a Turing Machine and } M \text{ accepts } w\}\,.$$

Theorem 1.9. *The language A_{TM} is not decidable.*

Proof. By way of contradiction, suppose that A_{TM} is decidable, and let M be a TM which decides A_{TM}. We use M to construct a Turing Machine M' that decides DIAG, contradicting Theorem 1.7.

The machine M' computing DIAG operates as follows. First, M' is given as input a string $\langle M''\rangle$ encoding some Turing Machine. As a first step, M' checks that the input is a proper encoding, otherwise it rejects. If the encoding is proper, it then simulates (using Theorem 1.5) the Turing Machine M computing A_{TM} on the input $\langle\langle M''\rangle, \langle M''\rangle\rangle$. If the simulation of the machine M halts and accepts, then M' halts and rejects. Otherwise, if M halts and rejects, then M' halts and accepts.

If M' halts and accepts, then by definition the simulation of M must have rejected, which implies that $\langle\langle M''\rangle, \langle M''\rangle\rangle \notin A_{\mathrm{TM}}$ and thus $\langle M''\rangle \notin \mathcal{L}(M'')$. On the other hand, if M' halts and rejects, then the simulation of M must have accepted, which implies $\langle M''\rangle \in \mathcal{L}(M'')$. It follows that M' decides DIAG, which is a contradiction. □

2 Undecidability and Reducibility

2.1 Lecture 3: Uncomputability, Properties of D and SD

Textbook Readings: Section 4.2

We have now answered our original main question: there *are* uncomputable languages, such as DIAG, and even languages which *are* semi-decidable and *not* decidable. With this in mind we take a shift in perspective to *classification*: what is the structure of SD and D? Are they structurally well-behaved with respect to operations on their underlying languages, like union or complementation?

To study the structure of SD and D we should study example languages which exemplify that structure (meaning $A_{\mathrm{TM}}, \mathrm{DIAG}$). For reference, here are the definitions of the languages again:

$$\begin{aligned} A_{\mathrm{TM}} &= \{(\langle M\rangle, x) \mid \text{M is a TM and } x \in \mathcal{L}(M)\} \\ \mathrm{DIAG} &= \{\langle M\rangle \mid \text{M is a TM and } \langle M\rangle \notin \mathcal{L}(M)\}\,. \end{aligned}$$

An easy observation is that A_{TM} is like a more general version of DIAG, except the acceptance condition has been “flipped” — in A_{TM} we accept the input $(\langle M\rangle, x)$ if $x \in \mathcal{L}(M)$, while in DIAG we accept the input $\langle M\rangle$ if $\langle M\rangle \notin \mathcal{L}(M)$;in this sense A_{TM} is like the “complement” of DIAG. We formalize this next.

Definition 2.1. Let Σ be a finite alphabet, and let $L \subseteq \Sigma^*$ be a language. The *complementary language* of L (or just the *complement* of L) is the language

$$\overline{L} = \Sigma^* \setminus L = \{x \in \Sigma^* \mid x \notin L\}\,.$$

By directly applying the previous definition we see that

$$\begin{aligned} \overline{A_{\mathrm{TM}}} &= \{(\langle M\rangle, x) \mid \langle M\rangle \text{ does not encode a TM or it does and } x \notin \mathcal{L}(M)\} \\ \overline{\mathrm{DIAG}} &= \{\langle M\rangle \mid \langle M\rangle \text{ does not encode a TM or it does and } \langle M\rangle \in \mathcal{L}(M)\}\,, \end{aligned}$$

and it is now clear that A_{TM} is essentially[3] a “more general” version of $\overline{\mathrm{DIAG}}$.

Our second observation is that even though A_{TM} is undecidable (Theorem 1.9), by using the simulation idea behind the Universal Turing Machine (Theorem 1.5) we can see that A_{TM} *is* semi-decidable.

Theorem 2.2. $A_{\mathrm{TM}} \in \mathsf{SD}$

Proof. We define a Turing Machine M_0 (using the Church-Turing thesis) such that $A_{\mathrm{TM}} = \mathcal{L}(M_0)$. The algorithm is quite simple: on input $(\langle M\rangle, x)$, the machine M_0 first checks if $\langle M\rangle$ is a well-defined encoding of a TM M. If it is not, then M_0 halts and rejects. Otherwise,

[3] Up to the encoding of the machine $\langle M\rangle$ being well-formed.

using the Universal Turing Machine construction, M_0 simulates the machine $\langle M \rangle$ on x. If the simulation of M halts and accepts x, then M_0 halts and accepts $(\langle M \rangle, x)$; if the simulation of M halts and rejects x, then M_0 similarly rejects $(\langle M \rangle, x)$.

If $\langle M \rangle$ is not a well-defined encoding of a TM then M_0 halts and rejects, so we assume without loss of generality that $\langle M \rangle$ is a well-defined encoding of a TM. If $(\langle M \rangle, x) \in A_{\mathrm{TM}}$ then $x \in \mathcal{L}(M)$, and so M_0 will halt and accept in its simulation of M on x. Otherwise, if $(\langle M \rangle, x) \notin A_{\mathrm{TM}}$ then $x \notin \mathcal{L}(M)$, and so either M_0 will not halt on its simulation of M on x, or if M halts and rejects x then so will M_0. It follows that $\mathcal{L}(M_0) = A_{\mathrm{TM}}$. □

By our comment before on "complementary" languages, it is easy to modify the above algorithm to show that $\overline{\mathrm{DIAG}}$ is semi-decidable.

Theorem 2.3. $\overline{\mathrm{DIAG}} \in \mathsf{SD}$.

Proof. Exercise. □

These results illustrate the "one-sidedness" of the languages in SD: we have $\overline{\mathrm{DIAG}} \in \mathsf{SD}$ while $\mathrm{DIAG} \notin \mathsf{SD}$. Informally we can think of the one-sided definition of SD like this: a language L is in SD if it is easy to determine if a string $x \in L$, but it is perhaps impossible to determine if $x \notin L$. This suggests that we should define the other side (or the *dual*) to this: the class of languages L where it is easy to determine if a string $x \notin L$, but it is perhaps impossible to determine if $x \in L$. The next class captures this idea:

Definition 2.4. Let Σ be a finite alphabet. A language $L \subseteq \Sigma^*$ is *co-semi-decidable* if its complement is semi-decidable:

$$L \in \mathsf{coSD} \Leftrightarrow \overline{L} \in \mathsf{SD}.$$

In other words, $L \in \mathsf{coSD}$ if and only if there is a Turing Machine M such that M halts and accepts x for all $x \notin L$.

As an added bonus, introducing this new class that is dual to SD lets us express our earlier results in a convenient way: $A_{\mathrm{TM}} \in \mathsf{SD}, \overline{A_{\mathrm{TM}}} \in \mathsf{coSD}$ and $\overline{\mathrm{DIAG}} \in \mathsf{SD}, \mathrm{DIAG} \in \mathsf{coSD}$, and both $A_{\mathrm{TM}}, \mathrm{DIAG} \notin D$.

We have used the undecidable languages (and the observation that A_{TM} is essentially a more general version of $\overline{\mathrm{DIAG}}$) to expand our collection of "computability classes" to $\mathsf{D}, \mathsf{SD}, \mathsf{coSD}$. How are these classes related? It should be clear that $\mathsf{D} \subseteq \mathsf{SD}$ and $\mathsf{D} \subseteq \mathsf{coSD}$: this is equivalent to saying $\mathsf{D} \subseteq \mathsf{SD} \cap \mathsf{coSD}$. This suggests another question: is $\mathsf{D} = \mathsf{SD} \cap \mathsf{coSD}$? Or, are there languages which are semi-decidable and co-semi-decidable, but not decidable?

Theorem 2.5 (Theorem 4.16 in [1])**.** $\mathsf{D} = \mathsf{SD} \cap \mathsf{coSD}$

Proof. It is easy to see that $\mathsf{D} \subseteq \mathsf{SD} \cap \mathsf{coSD}$, so we prove that $\mathsf{SD} \cap \mathsf{coSD} \subseteq \mathsf{D}$. Let $L \in \mathsf{SD} \cap \mathsf{coSD}$, let M_1 be a TM such that $L = \mathcal{L}(M_1)$, and let M_0 be a TM such that $\overline{L} = \mathcal{L}(M_0)$. Informally, the TM M_1 witnesses the fact that $L \in \mathsf{SD}$, and the TM M_2 witnesses $L \in \mathsf{coSD}$. We need to use these two Turing Machines to construct a single TM M such that $\mathcal{L}(M) = L$ and M halts on all inputs.

We use the Church-Turing thesis, and construct M by "interleaving" the computations of M_0 and M_1. To make defining the algorithm easier on ourselves, we define M as a two-tape Turing Machine. At first, M receives an input x on its first tape, and it copies x to its second tape. For the remainder of the computation, in each step M simulates a single step of M_1 on the first tape, and a single step of M_0 on the second tape. By definition, either $x \in L$ or $x \in \overline{L}$, and so at least one of these two simulations must halt and accept for any x. If the simulation of M_1 halts and accepts, then M halts and accepts x. Otherwise, if the simulation of M_0 halts and accepts, then M halts and rejects. It follows that M halts on all inputs and $\mathcal{L}(M) = \mathcal{L}(M_1) = L$. □

As important as the previous result is for the structure of semi-decidable languages, it is even more useful as a for classifying languages: in order to show that $L \notin \mathsf{D}$ it suffices to show that $L \in \mathsf{SD}, L \notin \mathsf{coSD}$ or vice-versa. Similarly, to show that $L \notin \mathsf{SD}$, we can show that $L \in \mathsf{coSD}$ and $L \notin D$: clearly $L \notin \mathsf{SD}$ must follow since if $L \in SD \cap \mathsf{coSD}$ then $L \in \mathsf{D}$, which is a contradiction. For example, we can show that $\overline{A_{\mathrm{TM}}} \notin \mathsf{SD}$.

Theorem 2.6. $\overline{A_{\mathrm{TM}}} \notin \mathsf{SD}$.

Proof. By Theorem 2.2 we know $A_{\mathrm{TM}} \in \mathsf{SD}$. Suppose by way of contradiction that $\overline{A_{\mathrm{TM}}} \in \mathsf{SD}$. Since $\overline{A_{\mathrm{TM}}} \in \mathsf{SD}$ if and only if $A_{\mathrm{TM}} \in \mathsf{coSD}$, the assumption implies that $A_{\mathrm{TM}} \in \mathsf{SD} \cap \mathsf{coSD}$, and so $A_{\mathrm{TM}} \in \mathsf{D}$ by Theorem 2.5. But Theorem 1.9 says that $A_{\mathrm{TM}} \notin \mathsf{D}$, which is a contradiction. □

Similarly we can prove that $A_{\mathrm{TM}} \notin \mathsf{coSD}$, and $\overline{\mathrm{DIAG}} \notin \mathsf{coSD}$, enforcing the "one-sided" view of SD and coSD. We leave these as exercises.

Now, we discovered the class coSD (which turns out to play an important role in the classification of languages, thanks to Theorem 2.5) by considering how the classes behave under the "complement" operation. So, another natural question is how do D, SD, and coSD behave with respect to *other* natural operations on languages? We say that a collection of languages C is *closed* under an operation on languages if applying the operation to languages in C yields another language in C. It turns out that the classes D, SD and coSD are each closed under several natural operations on languages. We prove several of these in the next theorem, wherein we also introduce a new technique known as *dovetailing*.

Theorem 2.7. *If* L_1, L_2 *are any languages in* SD*, then*

1. $L_1^* \in \mathsf{SD}$
2. $L_1 \cup L_2 \in \mathsf{SD}$
3. $L_1 \cap L_2 \in \mathsf{SD}$.

If we know that $L_1, L_2 \in \mathsf{D}$*, then in addition to the above three properties we have* $\overline{L_1} \in \mathsf{D}$*. In other words,* SD *is closed under union, intersection, and Kleene star, and* D *is additionally closed under complement.*

Proof. We leave proving $L_1^* \in \mathsf{SD}$ as an exercise, and give proofs of the other facts. First, suppose $L_1, L_2 \in \mathsf{SD}$ and we prove $L_1 \cap L_2 \in \mathsf{SD}$. Let M_1 be the Turing Machine recognizing L_1, let M_2 be the Turing Machine recognizing L_2, and we define a TM M recognizing $L_1 \cap L_2$. This is rather easy: on input x, M simulates M_1 on x and simulates M_2 on x, accepting if both simulations accept. For simplicity we define M to be a two-tape Turing Machine. First, M receives x on its first tape, and writes x to its second tape. Then, in each step of the computation M simulates M_1 on the first tape and M_2 on the second tape, halting and accepting if both simulations halt and accept. Note that $x \in L_1 \cap L_2$ if and only if both M_1 halts and accepts x and M_2 halts and accepts x, so it follows that $L_1 \cap L_2 = \mathcal{L}(M)$.

Now we prove that $L_1 \cup L_2 \in \mathsf{SD}$. We could follow the same strategy as before (now halting and accepting if either of the parallel computations halt and accept), but we take a different approach to introduce a new technique called *dovetailing*. Roughly speaking, dovetailing "interleaves" the computations of the two machines M_1 and M_2 at the same time, see Algorithm 1 for a formal implementation. Clearly Algorithm 1 halts and accepts if and only

Algorithm 1: Dovetailing

```
Input: A string $x \in \Sigma^*$
for $i = 1, 2, 3, \ldots$ do
    Simulate $M_1$ and $M_2$ on $x$ for $i$ steps;
    Accept $x$ if either simulation halts and accepts;
end
```

if one of the machines M_1, M_2 halts and accepts x; it follows by the Church-Turing thesis that $L_1 \cup L_2$ is computable in SD. □

We leave the rest of the above theorem as an exercise, but it is useful to compare the two possible proofs that $L_1 \cup L_2 \in \mathsf{SD}$ (the first by simulating both machines in parallel on different tapes, and the second by dovetailing). A nice example of a language which shows that dovetailing is strictly more powerful than the multi-tape simulation is the following

$$L = \{(\langle \mathcal{M} \rangle, x) \mid \mathcal{M} = M_1, M_2, \ldots, M_n \text{ is sequence of TMs and } x \in \mathcal{L}(M_i) \text{ for some } i\}\,.$$

2.2 Lecture 4: The Certificate Definition of SD, Reductions

Textbook Readings: Section 5.1

We have seen now how to prove languages are not decidable, and also some of the structural properties of D, SD, and coSD. In particular, we know that these classes of languages are well-behaved with respect to some basic set operations, and also the remarkable fact that D $=$ SD $\cap$ coSD. Our goal now is to introduce *reducibility*. This is a technique which refines our earlier undecidability proofs. In the development of reducibility we will introduce a nice alternative definition of the semi-decidable languages that is closely related to logic.

We first need some new definitions in order to discuss the next results. Up until now we have only discussed what it means to "compute" a language (or, equivalently, a "yes-no" predicate on strings). We are, of course, interested in computing other things like functions and relations.

Definition 2.8. Let Σ be an alphabet, and let $D \subseteq \Sigma^*$. A function $f : D \subseteq \Sigma^*$ is *computable* if there is a Turing Machine M such that on input x, for any $x \in D$, the machine M halts and accepts with $f(x)$ on the tape.

Notice that in the definition of the function we do not care what the behaviour of the Turing Machine is if the input x is not in the domain of the function D. We also introduce what it means to compute a relation.

Definition 2.9. Let Σ be an alphabet, let $n \in \mathbb{N}$. A relation

$$R \subseteq \underbrace{\Sigma^* \times \Sigma^* \times \cdots \times \Sigma^*}_{n \text{ times}}$$

is *semi-decidable* if there exists a Turing Machine M such that $(x, y) \in R$ if and only if the machine M halts and accepts an encoding of the input (x, y). Similarly, R is *decidable* if it is semi-decidable and the machine M halts on all inputs.

Next we give what is known as the *certificate characterization* of SD. This characterization gives an intuitive picture of the languages in SD as those for which there exist "short proofs" of membership. At a high level, this characterization essentially says that, if a language L is in SD, then for any $x \in L$ there exists a string $y \in \Sigma^*$ (the *proof*) which can be used to "verify" that x is in the language. On the other hand, if $x \notin L$, then no such proof y exists (another reinforcement of the "one-sided" nature of SD).

Theorem 2.10. *Let Σ be any alphabet and let $L \subseteq \Sigma^*$. The language L is semi-decidable if and only if there is a decidable binary relation $R \subseteq \Sigma^* \times \Sigma^*$ such that for all $x \in \Sigma^*$,*

$$x \in L \Leftrightarrow \exists y \in \Sigma^* : (x, y) \in R.$$

Proof. First we prove the "if" direction. Let L be a semi-decidable language, and let M_0 be a Turing Machine such that $\mathcal{L}(M_0) = L$. For any $(x, y) \in \Sigma^* \times \Sigma^*$, define R by

$$R(x, y) \Leftrightarrow y \text{ encodes an accepting computation of } M_0 \text{ on } x.$$

Next we show that $x \in L$ if and only if $(x, y) \in R$ and that R is decidable.

If $x \in L$, then M_0 accepts x, and so there must be some string y encoding an accepting computation of M_0 on x. Thus if $x \in L$ then there is a y such that $R(x, y)$ holds. Conversely, if there is a y that encodes an accepting computation of M_0 on x, then M_0 must indeed accept x. So if $R(x, y)$ holds then $x \in L$.

The algorithm for deciding R operates as follows.

Algorithm for R

```
1. On input $(x, y) \in \Sigma^* \times \Sigma^*$.
2. Check that $y$ encodes a sequence of Turing Machine configurations, with the last configuration in an accept state. If not, reject $(x, y)$.
3. Repeat the following:
    (a) Simulate $M_0$ on $x$ for one step, and check if the configuration of $M_0$ on $x$ is encoded at the appropriate place in $y$. If not, reject.
    (b) If this is the last configuration encoded in $y$, check that $M_0$ has actually halted and accepted $x$. If so, accept. Otherwise, reject.
```

Clearly the algorithm above halts on all of its inputs, as it rejects as soon as the simulation of M_0 on x exceeds the length of the computation encoded by y. Moreover, the algorithm only accepts if and only if the string y encodes an accepting computation that M_0 accepts x, and thus if and only if $(x, y) \in R$.

Now we prove the "only if" direction in the statement of the theorem. Suppose that such a decidable relation R exists, and we use the relation R to construct a Turing Machine M_0 which recognizes the language L. Let M be the Turing Machine deciding R. The machine M_0 will run the following algorithm.

Algorithm for L

```
1. On input $x \in \Sigma^*$.
2. For each $y \in \Sigma^*$:
    (a) Simulate $M$ on the input $(x, y)$. If $M$ accepts, then accept. Otherwise continue.
```

Since R is decidable the machine M halts on all inputs, so each simulation step in the for loop will halt. The algorithm above accepts $x \in \Sigma^*$ if and only if there is a $y \in \Sigma^*$ such that $R(x, y)$ holds; by assumption, this is equivalent to $x \in L$. If no such y exists, then the algorithm will never halt. Thus $\mathcal{L}(M_0) = L$ and so $L \in \mathsf{SD}$. $\square$

If we did not require that the relation R is decidable in the previous theorem then the analogous statement is very easy to satisfy: just define, for all $y \in \Sigma^*$,

$$R(x, y) \text{ holds } \Leftrightarrow x \in L.$$

Here the relation R just ignores the proof y and accepts only those x appearing in the language. Thus the above statement is only interesting when R is decidable (and then, it is natural to think of R as a "proof-checker" — it receives as input a string x and a proof y, and then verifies that the proof y shows that $x \in L$).

In the rest of this section we switch gears and return to discussing undecidability. Recall the proof that A_{TM} is undecidable (cf. Theorem 1.9). At a high level the proof goes like this:

1. Assume that A_{TM} is decidable and get a TM M that decides A_{TM}.

2. Use M to solve DIAG. To do this, we transform inputs x of DIAG into inputs y of A_{TM} such that $x \in$ DIAG if and only if $y \in A_{\mathrm{TM}}$. Then run the algorithm M on y.

The "main step" of this proof is the transformation of inputs of DIAG into inputs of A_{TM}. We call such a proof a *reduction*, as we have reduced the problem of solving DIAG to solving A_{TM}. Intuitively, since we know DIAG is hard, it follows that A_{TM} must also be hard, since if we can decide A_{TM} then we can decide DIAG. It turns out to be very useful to formally set out a definition of such a reduction between problems. We do this next.

Definition 2.11. For any languages $L_1, L_2 \subseteq \Sigma^*$, we say that L_1 is *many-to-one reducible* to L_2 (or *mapping reducible* to L_2) if there is a computable function $f : \Sigma^* \to \Sigma^*$ such that for all $x \in \Sigma^*$,

$$x \in L_1 \Leftrightarrow f(x) \in L_2.$$

For brevity, we sometimes say that L_1 is reducible to L_2, or write $L_1 \leq_m L_2$.

To illustrate this definition we give a different proof that A_{TM} is not decidable — it is helpful to compare this with the proof of Theorem 1.9.

Theorem 2.12. A_{TM} *is not decidable.*

Proof. We mimic the proof in Theorem 1.9, but formalize it as a reduction from $\overline{\mathrm{DIAG}}$ to A_{TM}. By definition, we must give a computable function f such that for all $x \in \Sigma^*$, $x \in \overline{\mathrm{DIAG}}$ if and only if $f(x) \in A_{\mathrm{TM}}$. For any Turing Machine M, consider the machine $M'(M)$ defined by the following algorithm (note that we have one machine $M'(M)$ for every machine M.

Algorithm for $M'(M)$

1. On input x.

2. Clear the tape and write the encoding $\langle M \rangle$.

3. Simulate M on $\langle M \rangle$. Accept if it accepts, and reject if it rejects.

The function f is defined as follows, for any $x \in \Sigma^*$:

$$f(x) = \begin{cases} (\langle M'(M) \rangle, \varepsilon) & \text{if } x = \langle M \rangle \text{ for some TM } M \\ x & \text{Otherwise.} \end{cases}$$

We claim that f is computable. Given a string x is it easy to check if x encodes a TM $\langle M \rangle$. If so, f transforms $\langle M \rangle$ into the machine $M'(M)$ described above (by suitably modifying its states and transition function), which is a computable transformation. If x does not encode a TM, then $f(x) = x$, and this is also computable. Thus f is computable.

Now we prove that f is a reduction: thus, we must prove that $x \in \overline{\text{DIAG}}$ if and only if $f(x) \in A_{\text{TM}}$. If $x \in \overline{\text{DIAG}}$, then $x = \langle M \rangle$ for some Turing Machine M such that M accepts $\langle M \rangle$. By the definition of f, it follows that $f(x) = (\langle M'(M) \rangle, \varepsilon)$, where $M'(M)$ is defined by the previous algorithm and ε is the empty string. The machine $M'(M)$ accepts the input ε if and only if M accepts $\langle M \rangle$, thus $(M'(M), \varepsilon \in A_{\text{TM}})$.

On the other hand, if $f(x) \in A_{\text{TM}}$, then it is easy to see by the definition of f that $f(x) = (\langle M'(M) \rangle, \varepsilon)$ where $\langle M \rangle = x$. By the definition of A_{TM}, if $(\langle M'(M) \rangle, \varepsilon) \in A_{\text{TM}}$ then $M'(M)$ accepts ε, which implies that M accepts $\langle M \rangle$. Thus $\langle M \rangle \in \overline{\text{DIAG}}$.

Now that we have shown that f is a reduction, we prove that A_{TM} is not decidable. Suppose that A_{TM} is decidable by contradiction, and we give an algorithm that decides $\overline{\text{DIAG}}$. The algorithm is simple: given an input $x \in \Sigma^*$, first run the reduction f above and then simulate the algorithm that decides A_{TM} on $f(x)$. Since f is a reduction, $f(x)$ is accepted by the algorithm for A_{TM} if and only if $x \in \overline{\text{DIAG}}$, thus this algorithm correctly decides $\overline{\text{DIAG}}$. □

We finish by showing that a new language is not decidable, now by a reduction from A_{TM}.

Theorem 2.13. *The language*

$$\text{HB} = \{\langle M \rangle \mid M \textit{ is a TM and } M \textit{ halts on the empty string}\}$$

is not decidable.

Proof. We give a reduction from A_{TM}. Let $\langle M \rangle$ be an encoding of a TM and let x be an input. Consider the following TM $M'(M, x)$ which depends on both M and x.

Algorithm for $M'(M, x)$

1. On input $y \in \Sigma^*$.
2. Clear the tape and write x.
3. Simulate M on x. If M halts and accepts, then accept. If M halts and rejects, go into an infinite loop.

Our reduction f is defined as follows. Given a string y, if $y = (\langle M \rangle, x)$ for some Turing Machine M then define $f(y) = \langle M'(M, x) \rangle$. Otherwise, set $f(y) = \varepsilon$ (which does not encode a TM). As before, the description of the TM $M'(M, x)$ can be easily computed from $(\langle M \rangle, x)$, and so f is computable.

Now we prove that f is a reduction. If $(\langle M \rangle, x) \in A_{\text{TM}}$, then M halts and accepts x; it follows that $M'(M, x)$ must halt and accept its input. Thus $f((\langle M \rangle, x)) = \langle M'(M, x) \rangle \in \text{HB}$. On the other hand, if $\langle M'(M, x) \rangle \in \text{HB}$, then $M'(M, x)$ halts on the blank tape. By definition of $M'(M, x)$, this can only happen if M halts and accepts x. It follows that $(\langle M \rangle, x) \in A_{\text{TM}}$, and so this is a well-defined reduction.

The proof is completed as before. Suppose HB is decided by a machine M_0, and we use the reduction and M_0 to decide A_{TM}. This is quite easy — given an input $y \in \Sigma^*$, just run M_0 on $f(y)$. Since f is a reduction, M_0 accepts $f(y)$ if and only if $y \in A_{\mathrm{TM}}$. Contradiction! $\square$

2.3 Lecture 5: More Undecidable Problems

Administrative Note: As much of the lecture was spent on going over the solutions to Assignment 1, the notes are particularly short today.

We continue our discussion of reductions. Recall that a language L_1 reduces to a language L_2 if inputs of L_1 can be transformed into inputs L_2 by some computable f so that $x \in L_1$ if and only if $f(x) \in L_2$. Figure 2 illustrates: essentially, the languages L_1 and L_2 partition the set Σ^* of all strings into two sets, and a reduction is a function from Σ^* to Σ^* which "respects" these partitions.

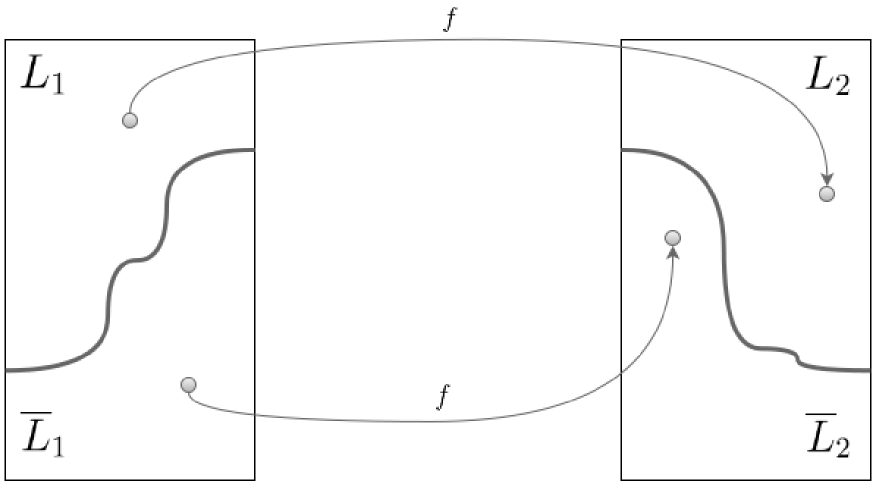

Figure 2: A reduction from L_1 to L_2. We have no arrows "crossing" the partitions.

The next theorem records the (essentially trivial) fact that if L_1 is reducible to L_2 and L_2 is computable then L_1 is also computable. By using the contrapositive, we also get an essential tool for showing that languages are *not* computable. We are now in the fortunate position where the mathematical framework that we have been building up piece by piece allows us to give a quite simple and clean proof. In fact, we only need to explicitly give one (!) algorithm: the other two facts follow from structural results.

Theorem 2.14. *Let Σ be an alphabet and let $L_1, L_2 \subseteq \Sigma^*$ be languages such that $L_1 \leq_m L_2$. Then*

1. *If $L_2 \in \mathsf{SD}$ then $L_1 \in \mathsf{SD}$.*
2. *If $L_2 \in \mathsf{coSD}$ then $L_1 \in \mathsf{coSD}$.*
3. *If $L_2 \in \mathsf{D}$ then $L_1 \in \mathsf{D}$.*

Proof. We prove (1) by giving an algorithm, and then (2) and (3) immediately follow by applying other properties of D, SD, and coSD. First assume that L_2 is semi-decidable. Let $f : \Sigma^* \to \Sigma^*$ be the reduction from L_1 to L_2, and let M_2 be any Turing Machine such that $\mathcal{L}(M_2) = L_2$. Consider the following algorithm for L_1:

Algorithm for L_1

1. On input $x \in \Sigma^*$.

2. Compute $f(x)$, simulate the algorithm for M_2 on the result. Accept if M_2 accepts, reject if M_2 rejects.

Let M_1 be the Turing Machine corresponding to the above algorithm. Clearly M_1 accepts x if and only if M_2 accepts $f(x)$, and M_2 accepts if and only if $f(x) \in \mathcal{L}(M_2) = L_2$. Since f is a reduction we have that $x \in L_1$ if and only if $f(x) \in L_2$, and so it follows that $\mathcal{L}(M_1) = L_1$.

To see (2), assume that $L_2 \in \mathsf{coSD}$. By definition, if f is a reduction from L_1 to L_2 then $x \in L_1 \Leftrightarrow f(x) \in L_2$, or equivalently $x \notin L_1 \Leftrightarrow f(x) \notin L_2$. Thus $\overline{L_1} \leq_m \overline{L_2}$. Since L_2 is co-semi-decidable, it follows that $\overline{L_2}$ is semi-decidable by definition. Therefore $\overline{L_1}$ is semi-decidable, and so L_1 is co-semi-decidable.

Finally, assume that L_2 is decidable; then L_2 is both semi-decidable and co-semi-decidable by Theorem 2.5. Applying (1) and (2) above shows that L_1 is semi-decidable and co-semi-decidable. Applying Theorem 2.5 again implies that L_1 is decidable. $\square$

As an application, let us classify the following language with respect to D, SD, and coSD. We only use reductions and later "structural" results — observe that the amount of work that we have to do is surprisingly small!

Theorem 2.15. *Let*

$$\mathrm{NE} = \{\langle M\rangle \mid M \textit{ is a TM and } \mathcal{L}(M) \neq \emptyset\}\,.$$

Then

1. $\mathrm{NE} \in \mathsf{SD}$

2. $\mathrm{NE} \notin \mathsf{coSD}$

3. $\mathrm{NE} \notin \mathsf{D}$

Equivalently, $\mathrm{NE} \in \mathsf{SD} \setminus \mathsf{D}$.

Proof. It is easy to see that $\mathrm{NE} \in \mathsf{SD}$ by dovetailing. Let $x_0, x_1, x_2, \ldots$ be any computable enumeration of the strings in Σ^*. Consider the following algorithm.

Algorithm for NE

```
1. On input $\langle M\rangle$
2. For each $i = 0, 1, 2, \ldots$
    (a) Simulate $M$ on each string $x_0, x_1, \ldots, x_i$ for $i$ steps.
    (b) If any of the above simulations accept, accept.
```

Clearly if $\langle M \rangle \in \mathrm{NE}$ then there is an $x_j \in \Sigma^*$ such that $x_j \in \mathcal{L}(M)$; thus the above algorithm will accept x_j for some sufficiently large i. On the other hand, if $\langle M \rangle$ is accepted by the algorithm after i iterations, then for some $j \leq i$ the machine M accepts x_j, and so $\langle M \rangle \in \mathrm{NE}$. Thus the above algorithm accepts NE, and so $\mathrm{NE} \in \mathsf{SD}$.

Now we prove that $\mathrm{NE} \notin \mathsf{D}$. To see this we give a reduction from A_{TM}. Let $(\langle M \rangle, x)$ be any pair such that M is a TM, and we show how to construct a Turing Machine $M'_{(M,x)}$ such that

$$(\langle M \rangle, x) \in A_{\mathrm{TM}} \Leftrightarrow M'_{(M,x)} \in \mathrm{NE}.$$

Consider the following algorithm $M'_{(M,x)}$, which is computable from the pair $(\langle M \rangle, x)$

Algorithm for $M'_{(M,x)}$

1. On input $y \in \Sigma^*$.
2. Overwrite the tape with blanks, and then simulate M on x. Accept if and only if the simulation accepts.

If $(\langle M \rangle, x) \in A_{\mathrm{TM}}$, then the algorithm $M'_{(M,x)}$ accepts all inputs y, and so

$$\mathcal{L}(M'_{(M,x)}) = \Sigma^* \neq \emptyset.$$

This means $M'_{(M,x)} \in \mathrm{NE}$. On the other hand, if $M'_{(M,x)} \in \mathrm{NE}$, then by the definition of $M'_{(M,x)}$ it is easy to see that $M'_{(M,x)}$ accepts any input if and only if M accepts x. Thus $(\langle M \rangle, x) \in A_{\mathrm{TM}}$.

It follows that $A_{\mathrm{TM}} \leq_m \mathrm{NE}$, and so by Theorem 2.14 we have that NE is not in D since A_{TM} is not in D. This immediately shows that $\mathrm{NE} \notin \mathsf{coSD}$ as well since $\mathsf{D} = \mathsf{SD} \cap \mathsf{coSD}$ by Theorem 2.5. □

2.4 Lecture 6: Turing Reducibility, Completeness, Beyond SD and coSD

Now that we have seen many examples of reductions, you may have noticed one of two things. First, our notion of a reduction does seem a little restrictive compared to our intuition: intuitively, when we reduce solving one problem to solving another we do not have such a "rigid form" for what the reductions are allowed to do. For example, we may reduce solving some problem A to solving many copies of problem B, or maybe to solving problems B and C, and then combining the answers to B and C together in some complicated way to yield a solution to A. A many-to-one reduction only allows us to transform an input of A into an input of B such that membership across the languages is preserved — compared to the intuitive notion, it is quite inflexible. In this lecture we discuss *Turing Reductions*, which are closer in form to our intuition about how reductions work.

Second, why is A_{TM} so useful when performing reductions between problems? Well, it turns out that A_{TM} exhibits a kind of *universality* of its own that is not unlike a Universal Turing Machine: it turns out that if A_{TM} reduces to a language L, then *every* language in SD reduces to L.

Definition 2.16. Let Σ be any alphabet. Let C be a collection of languages over Σ, and let $L \subseteq \Sigma^*$ be a language. The language L is *hard* for C if $L' \leq_m L$ for all $L' \in \mathsf{C}$; it is *complete* for C if it is hard for C and also $L \in \mathsf{C}$.

Theorem 2.17. *The language A_{TM} is complete for* SD.

Proof. Let L be any language in SD and let M be the TM computing L. We give a many-to-one reduction from L to A_{TM}. The reduction is quite simple: for any $x \in \Sigma^*$ map

$$x \mapsto (\langle M \rangle, x).$$

It is quite obviously computable, so we focus on showing $x \in L \Leftrightarrow (\langle M \rangle, x) \in A_{\mathrm{TM}}$. This is also easy: if $x \in L$ then M accepts x, and so $(\langle M \rangle, x) \in A_{\mathrm{TM}}$. Conversely, if $(\langle M \rangle, x) \in A_{\mathrm{TM}}$, then $x \in L$. $\square$

Corollary 2.18. *The language $\overline{A}_{\mathrm{TM}}$ is complete for* coSD.

Proof. Let $L \in \mathsf{coSD}$. By Theorem 2.17 we have $L' \leq_m A_{\mathrm{TM}}$ for all $L' \in \mathsf{SD}$, and so since $\overline{L} \in \mathsf{SD}$ we have that $\overline{L} \leq_m A_{\mathrm{TM}}$. But this immediately implies that $L \leq_m \overline{A}_{\mathrm{TM}}$. $\square$

This theorem is useful in two ways: first, it shows that A_{TM} is a "super flexible" language, in so far as efficient algorithms for A_{TM} yield efficient algorithms for *every* language in SD. Second, it shows that A_{TM} inherits the "hardness" from every single language in the class SD — if any language in SD is not in some class C, then we immediately obtain that A_{TM} is not in C.

Keep the completeness of A_{TM} in mind as we begin to explore *Turing Reductions*, which is are a more powerful class of reductions that is more closely related to our intuition. Intuitively, in a Turing reduction from a language A to a language B, we assume the existence of some "black box" which computes B (called an *oracle*) that, when given a string y as input, immediately returns whether or not y is in B. We then give the machine that is computing A the power to query this magic black box, and so in solving A we may query membership of strings in B multiple times in an adaptive way (that is, with later queries depending on earlier queries).

In order to properly define how to access this black box, we need to introduce a new variant of Turing Machine.

Definition 2.19. Let Σ be any alphabet and let $L \subseteq \Sigma^*$ be any language. A *Turing Machine with oracle* L is a Turing Machine M equipped with a special *oracle tape*, alongside its regular read-write tapes, which can be written to and read from as normal. At any time during the computation M may enter a special oracle state, at which point the string x that is written on the oracle tape is replaced with 0 if $x \notin L$ and 1 if $x \in L$. If M is a Turing Machine with oracle L, we say that M is an *oracle Turing Machine* (OTM), and may write M^L to specify that M is receiving an oracle for the language L.

Definition 2.20. Let Σ be an alphabet and let $A, B \subseteq \Sigma^*$. The language A is *Turing reducible* to B if there is an oracle Turing Machine M with an oracle for B deciding A. We write $A \leq_T B$ if A is Turing reducible to B.

It turns out that Turing reducibility is no more powerful than many-to-one reducibility if all languages involved are decidable.

Theorem 2.21. *If $A \leq_T B$ and B is decidable then A is decidable.*

Proof Sketch. Let M^B be an oracle Turing Machine deciding A. Since B is decidable, we can just replace each call to the oracle with a simulation of the algorithm deciding B. □

However, it is important to keep in mind that similar theorems do *not* hold for SD and coSD. In fact, equipping a Turing Machine M with an oracle for A_{TM} makes it extremely powerful. Intuitively this is because of Theorem 2.17: since every language in SD is reducible to A_{TM}, giving a TM M an oracle to A_{TM} essentially allows M to **decide** any language in SD in a single step. This is much more powerful than what regular Turing Machines can do (and, in fact, is the first variant of a Turing Machine that we have seen which is *not* equivalent to a regular TM). The next proposition supports this fact: we give a language that is not in SD or coSD, but it can still be computed by a OTM with an A_{TM} oracle.

Proposition 2.22. *The language*

$$\mathrm{TOTAL} = \{\langle M \rangle \mid M \textit{ is a TM that halts on every string.}\}$$

is not in SD *or* coSD. *However, there is an oracle TM $M^{A_{\mathrm{TM}}}$ with an oracle for A_{TM} computing $\overline{\mathrm{TOTAL}}$.*

Proof. We prove that TOTAL is not in SD or coSD by many-to-one reductions from A_{TM} and $\overline{A}_{\mathrm{TM}}$. Let $(\langle M \rangle, x)$ by any TM/input pair, and consider the machine $M'_{(\langle M \rangle, x)}$ defined by the following algorithm.

Algorithm for $M'_{(\langle M \rangle, x)}$

1. On input $y \in \Sigma^*$.

2. Simulate M on x and accept if M accepts x. If M rejects x, loop forever.

Our many-to-one reduction maps $(\langle M \rangle, x) \mapsto M'_{(\langle M \rangle, x)}$, and it is clearly computable. It is fairly easy to see that $(\langle M \rangle, x) \in A_{\text{TM}}$ if and only if $\langle M'_{(\langle M \rangle, x)} \rangle \in \text{TOTAL}$. For if $(\langle M \rangle, x) \in A_{\text{TM}}$, then $M'_{(\langle M \rangle, x)}$ accepts every input string y and so it is in TOTAL. On the other hand, if $M'_{(\langle M \rangle, x)} \in \text{TOTAL}$, then it must halt on every input, and so it follows that M does not reject x. Thus $(\langle M \rangle, x) \in A_{\text{TM}}$ and the reduction is well-defined. We conclude that $\text{TOTAL} \notin \text{coSD}$ since $A_{\text{TM}} \notin \text{coSD}$.

Now we reduce $\overline{A}_{\text{TM}}$ to TOTAL, proving it is not in SD. Let $(\langle M \rangle, x)$ by any TM/input pair, and consider the machine $M^*_{(\langle M \rangle, x)}$ defined by the following algorithm.

Algorithm for $M^*_{(\langle M \rangle, x)}$

1. On input $y \in \Sigma^*$

2. If $\langle M \rangle$ is not a well-formed Turing Machine, accept.

3. Simulate M on x for $|y|$ steps. If M has not accepted x when the simulation ends, accept. Otherwise loop forever.

Our reduction maps $(\langle M \rangle, x) \mapsto \langle M^*_{(\langle M \rangle, x)} \rangle$, and we show that $(\langle M \rangle, x) \in \overline{A}_{\text{TM}}$ if and only if $\langle M^*_{(\langle M \rangle, x)} \rangle \in \text{TOTAL}$. First, suppose that $(\langle M \rangle, x) \in \overline{A}_{\text{TM}}$. It follows that either $\langle M \rangle$ is not a well-defined Turing Machine — in which case $M^*_{(\langle M \rangle, x)}$ accepts all inputs $y \in \Sigma^*$ — or $\langle M \rangle$ does not accept the input x, in which case $M^*_{(\langle M \rangle, x)}$ again accepts all inputs, and so $M^*_{(\langle M \rangle, x)} \in \text{TOTAL}$.

On the other hand, if $(\langle M \rangle, x) \notin A_{\text{TM}}$, then M must accept x. Let $i \in \mathbb{N}$ be the number of computation steps it takes for M to accept x. It follows that for all $j \geq i$ the machine $M^*_{(\langle M \rangle, x)}$ does not halt on any input y with $|y| = j$, and so $M^*_{(\langle M \rangle, x)} \notin \text{TOTAL}$. Thus $(\langle M \rangle, x) \in \overline{A}_{\text{TM}}$ if and only if $M^*_{(\langle M \rangle, x)} \in \text{TOTAL}$, and so TOTAL is not in SD.

Finally, we prove that

$$\overline{\text{TOTAL}} = \{\langle M \rangle \mid \langle M \rangle \text{ does not encode a TM or } M \text{ does not halt on all inputs}\}$$

can be computed by a Turing Machine $M^{A_{\text{TM}}}$ with an oracle for A_{TM}. Consider the following algorithm.

Algorithm for M^{TM}

1. On input $y \in \Sigma^*$.

2. If y does not encode a TM, accept. Otherwise let N be the TM such that $y = \langle N \rangle$.

3. Compute the encoding of $\langle N' \rangle$, defined by the following algorithm:

   Algorithm for N'

   (a) On input $z \in \Sigma^*$.

   (b) Simulate N on z. If the simulation halts and accepts or rejects, then accept.

4. For each string $x \in \Sigma^*$, query the oracle for A_{TM} about the pair $(\langle N' \rangle, x)$. If $(\langle N' \rangle, x) \notin A_{\mathrm{TM}}$ then accept.

We claim that $\mathcal{L}(M^{A_{\mathrm{TM}}}) = \overline{\mathrm{TOTAL}}$. Let $y \in \Sigma^*$ be any string. If y does not encode a TM, then y is accepted by M^{TM} and it is also in $\overline{\mathrm{TOTAL}}$, so assume that $y = \langle N \rangle$ does encode a TM N. By definition, the algorithm M^{TM} accepts the input $\langle N \rangle$ if and only if there is an $x \in \Sigma^*$ such that $(\langle N' \rangle, x) \notin A_{\mathrm{TM}}$. By examining the algorithm for N', it is easy to see that $(\langle N' \rangle, x) \notin A_{\mathrm{TM}}$ if and only if N does not halt on x. Combining these two steps together we get that $M^{A_{\mathrm{TM}}}$ accepts $\langle N \rangle$ if and only if there is an $x \in \Sigma^*$ such that N does not halt on x, and thus $\mathcal{L}(M^{A_{\mathrm{TM}}}) = \overline{\mathrm{TOTAL}}$. □

2.5 Computability Review

Here is a quick survey of everything we did over the first half of the course.

Turing Machines, Computational Problems Definition of a Turing Machine (our basic model of computation). Several variants of Turing Machines (nondeterministic, multi-tape, etc.) which turn out to be equivalent in power.

Computational Universality The existence of a Universal Turing Machine that can simulate other Turing Machines.

Intro to Computability Theory There exist languages (shown to exist via **diagonalization**) that can not be computed (and in fact, **most** languages are not computable). Computability classes $\mathsf{D}, \mathsf{SD}, \mathsf{coSD}$.

Algebraic Properties of Computability Classes Closure properties of $\mathsf{D}, \mathsf{SD}, \mathsf{coSD}$. Relations between $\mathsf{D}, \mathsf{SD}, \mathsf{coSD}$. The certificate ("proof theoretic") definition of SD.

Undecidability and Reducibility Many-to-one reductions, and formalizing reducibility between languages. Techniques for classifying languages into D, SD, and coSD via reducibility (e.g. **dovetailing**). Lots of examples.

Turing Reducibility, Completeness, and Beyond Complete and hard languages for computability classes. Oracle Turing Machines, and how using powerful oracles allows us to compute languages beyond SD and coSD.

3 Complexity Theory

3.1 Lecture 7: Models of Efficient Computation

Let us take a brief pause, and we will begin to switch gears from *computability theory* into *complexity theory*. First of all — what is the difference? Computability theory is concerned with the study of whether or not certain languages are computable, and we have had great success with classifying languages with respect to "how computable they are" (i.e. whether or not they are in D, SD, or coSD). Realistically, though, membership in D is an *extremely weak* guarantee on how easy a language is to compute. Recall that a language L is in D if there is a Turing Machine M that decides it — the Turing Machine M could take $O(n)$ transitions in the worst case, or $O(2^n)$, or $O(2^{2^n})$, or much much worse! An algorithm is not very useful if inputs of length $n = 2$ cannot be decided before the human race is extinct, and so in this sense we should not take membership in D as being synonymous with *effectively computable*. So what should be the definition of effectively computable? What languages can we realistically compute? Can we model this in a similar way as we did in computability theory?

The study of these types of questions is known as *computational complexity theory*. Typically, in (basic) complexity theory we fix

1. A model of computation, such as a Turing machine or a boolean circuit.
2. A *computational resource*, which can either be "mundane" like the amount of time or memory used, or more exotic resources like number of random bits used, number of non-deterministic steps, or number of bits communicated between two processes in a distributed system.

Once we have fixed these two parameters, we are free to ask what kinds of things can be computed in the bounded-resource computational model that results. The theory becomes very rich when we consider more exotic computational models, and also how different computational resources can be used to simulate one another.

In this course we will be primarily interested in the study of *time*, which is the number of steps a Turing machine needs to halt.

Definition 3.1. Let M be a Turing machine that always halts (i.e. a decider), and let $t(n) : \mathbb{N} \to \mathbb{N}$ be a function. We say that M has *time complexity* $t(n)$ if for all $n \in \mathbb{N}$ the machine M halts on all inputs of length n after at most $t(n)$ transitions.

Immediately we get differences between computability theory and complexity theory. In computability theory the basic class, D, is robust to the choice of underlying model. In other words, if we replace a Turing machine with a multi-tape Turing machine, or a non-deterministic decider, or even a "faulty" Turing machine (like that seen in Assignment 1) then the collection of languages in D does not change. It is reasonable to ask whether or not this is true for complexity theory. To do this let us introduce another definition.

Definition 3.2. Let $t(n) : \mathbb{N} \to \mathbb{N}$ be a function and let $L \subseteq \Sigma^*$ be a language. The language L is in the class $\mathsf{TIME}(t(n))$ if there is a multi-tape Turing machine M accepting L with time complexity $O(t(n))$.

Now we can formalize our question as:

For every function $t(n)$, does the class $\mathsf{TIME}(t(n))$ change if we change the underlying model from a multi-tape Turing machine to something else?

Example. Let $PAL = \left\{x \in \{0,1\}^* \mid x = x^R\right\}$ be the set of palindromes. We give two algorithms for PAL: one on a single-tape Turing machine, and one on a two-tape Turing machine. The single-tape algorithm runs in time $O(n^2)$, while the two-tape algorithm runs in time $O(n)$. Moreover, one can show (although we will not) that *any* single-tape TM accepting PAL runs in time $\Omega(n^2)$ — the upper bound is tight!

Single-Tape Algorithm for PAL

1. On input $x = x_1x_2 \dots x_n \in \Sigma^*$
2. Repeat until all characters are marked:
   (a) Mark the first unmarked character in x and scan to the last unmarked character in x. If they are different, then halt and reject, otherwise mark the last unmarked character and continue.
3. Accept.

Recall that the time complexity of the algorithm is the number of transitions. On a string of length n, the single-tape algorithm above must scan from one end of the tape to the other (a step that takes $O(n)$ transitions), and it must do this $O(n)$ times. Thus the run time of the algorithm is $O(n^2)$.

Two-Tape Algorithm for PAL

1. On input $x = x_1x_2 \dots x_n$.
2. Scan to the end of the input, and copy the input in reverse to the second tape.
3. Move both heads back to the beginning of the second tapes.
4. In parallel, check if the character under each read-write head is the same. If so, move both heads to the right and continue. If not, halt and reject. If all characters are the same, halt and accept.

The two-tape algorithm for PAL uses three $O(n)$-time sweeps of the entire input, and so the algorithm runs in $O(n)$ time.

So, on single tape TMs we have that PAL requires $\Theta(n^2)$ time to compute, while on multi-tape TMs PAL can be computed in $O(n)$ time. Thus the complexity class $\mathsf{TIME}(n)$ is not robust to changing the underlying model from a multi-tape TM to a single-tape TM (and, in fact, this is why we used "multi-tape TM" in the definition in the first place).

For our purposes, in the computational complexity theory portion of the course we fix the model to be a multi-tape Turing machine. With that we can make one of the most important definitions in the entire theory.

Definition 3.3. The complexity class P is defined to be

$$\mathsf{P} = \bigcup_{c \geq 0} \mathsf{TIME}(n^c).$$

The class P consists of all languages that can be decided by a multi-tape Turing machine with time complexity $O(n^c)$ for any fixed constant c, i.e. *polynomial* time. In complexity theory the class P is essentially seen as synonymous with languages that are *efficiently computable*. There are many obvious criticisms here: would we say that a $O(n^5)$ algorithm is "efficient"? How about $O(n^{100})$, or $O(n^{1000})$? Obviously no, but pragmatically speaking the benefits of this definition outweigh the criticisms:

1. First, and perhaps most importantly, most languages that people wish compute efficiently in practice do not have these pathologically large run-times. It is possible to construct these languages, but it seems that most "natural" languages have run times like $O(n)$, $O(n \log n)$, $O(n^2)$, and so on.
2. Second, P is robust, just like D is. That is to say: the languages in P are not changed if we change the underlying from a multi-tape TM to a single-tape TM (exercise!)

Example. In this example we give a natural problem appearing in P. We also use this opportunity to change gears in how we define languages.

Problem 1. s-t Connectivity
Input: A directed graph G, two distinguished vertices s, t with $s \neq t$ in G.
Problem: Decide if there is a path from s to t using edges in G.

Implicitly this problem is equivalent to the language

$$\text{STCONN} = \{(\langle G \rangle, \langle s \rangle, \langle t \rangle) \mid G \text{ is a directed graph and there is a path from } s \text{ to } t \text{ in } G\}.$$

However, we are all comfortable at this point with understanding that Turing Machines do not receive graphs as input, they receive encodings of graphs, and so on. From now on we will suppress these details when it is convenient — this abstraction will make our lives much easier, but, if we must to return to the details of how things are encoded then we will.

So, how do we show that this problem is in P? Well, this is a fairly easy algorithm design question — we leave it as an exercise to give an $O(n)$-time algorithm.

3.2 Lecture 8: NP, coNP, and Polynomial-Time Reducibility

It is helpful to think of P as the "scaled-down" version of the class D, in the sense of bounding the running time. Indeed, it shares many of the same properties: it is robust with respect to the underlying model (for example, if we exchange the multi-tape TM with a single-tape TM), and it is *two-sided* in the sense that if $L \in P$ then $\overline{L} \in P$. So, suppose we try extend this analogy further: what are the "scaled-down" versions of SD and coSD? The answer to this question are the complexity classes NP and coNP. To define NP we "scale down" the certificate definition of SD (cf. Theorem 2.10) by requiring that the verifier runs in polynomial time.

Definition 3.4. Let L be a language. A *verifier* for L is a Turing Machine M such that

$$x \in L \Leftrightarrow \exists y \in \Sigma^* : M \text{ accepts } (x, y).$$

The string y is called a *certificate* (cf. Theorem 2.10).

A language L is in the complexity class NP if it has a verifier which runs in polynomial-time *in the length of* x on all inputs.

With the definition of the class NP we can define the class coNP analogously to the definition of coSD.

Definition 3.5. A language L is in the complexity class coNP if $\overline{L} \in \mathsf{NP}$.

You may naturally wonder wonder why we chose to scale down the certificate definition of SD rather than the regular definition of *recognizability*. It turns out that if we instead bound the run-time of "recognizers" we get an apparently different class than NP (this will be explored further in Assignment 3). The fact that these two definitions are not readily equivalent with the run-time bound, is evidence that P, NP, and coNP are much more complicated beasts than D, SD and coSD. Another natural question to ask is whether or not $\mathsf{P} = \mathsf{NP} \cap \mathsf{coNP}$, since $\mathsf{D} = \mathsf{SD} \cap \mathsf{coSD}$. Despite $\mathsf{D} = \mathsf{SD} \cap \mathsf{coSD}$ being an easy theorem, whether or not $\mathsf{P} = \mathsf{NP} \cap \mathsf{coNP}$ remains an open research problem.

By defining NP using polynomial-time verifiers we can also borrow the *intuition* of the certificate definition for SD — that is, NP contains all languages which contain efficiently verifiable "proofs of membership". It is perhaps easiest to see this through an example.

Example. The *Clique* problem is defined as follows. Recall if $G = (V, E)$ is a graph then a *clique* in G is a collection of vertices $V' \subseteq V$ such that every pair of vertices $u, v \in V'$ is connected by an edge $uv \in E$. Equivalently, a clique is a complete subgraph of G.

Problem 2. Clique
Input: A graph $G = (V, E)$, a positive integer k.
Problem: Decide if G contains a clique with at least k vertices.

Can we decide the clique problem efficiently? Some thought should convince you that it is not easy to do so without essentially checking all possible cliques of size k. Since there are $O(n^k)$ of these for $k \leq n/2$ this is an exponential time algorithm, and so should not be considered "efficient". On the other hand, it is easy to give a polynomial-time verifier for Clique.

Verifier for Clique

1. On input (G, k) and y.
2. Check that y encodes a subset of vertices S_y of G with $|S_y| \geq k$. If not, reject.
3. Check that $uv \in E$ for each pair of vertices $u, v \in S_y$. If not, reject.
4. Accept if all pairs are connected.

Clearly this verifier runs in polynomial time in the size of G — it takes polynomial time to check that y encodes a subset of vertices, and checking that each pair of vertices encoded by y is connected requires $O(|V|^2)$ time in the worst case. Moreover, if G contains a clique C of size at least k, then certainly there exists a y that makes the verifier accept (just choose the string y encoding C). Conversely, if the verifier accepts (G, k) and y then the set S_y is a clique of size at least k by definition. It follows that $\text{Clique} \in \mathsf{NP}$.

Note that if we even slightly alter the definition of Clique then it is not so easy to show that it is in NP. For example, consider the following variant:

Problem 3. MaxClique
Input: A graph $G = (V, E)$, a positive integer k.
Problem: Decide if the largest clique in G has at least k vertices.

For the MaxClique problem we cannot pull the same trick as before, since we also have to verify that the clique encoded by the string y is larger than every other clique in the graph. This is roughly analogous to what happens when you consider languages in SD vs. coSD: intuitively, languages in coSD $\setminus$ SD are hard because they require you to check "infinitely many conditions".

Next we explain where the "N" in NP comes from: it stands for an alternative definition of NP using *nondeterministic Turing Machines*. First we need to define what time complexity means for a non-deterministic machine. We make the "obvious" choice — it is the longest amount of time that the machine takes over any possible input of length n.

Definition 3.6. Let M be a nondeterministic Turing Machine. We say that M has *time complexity* $t(n)$ if for each $n \in \mathbb{N}$ and each input x of length n, the machine M halts in at most $t(n)$ steps no matter which non-deterministic transitions are chosen in the execution of M. For any function $t(n) : \mathbb{N} \to \mathbb{N}$ the complexity class $\mathsf{NTIME}(t(n))$ contains all languages L that are computable by a non-deterministic TM with time complexity $t(n)$.

Theorem 3.7. $\mathsf{NP} = \bigcup_{c>0} \mathsf{NTIME}(n^c)$.

Proof. Proof reviewed in next class. □

Example. Let us consider another problem in NP. A *vertex cover* in a graph $G = (V, E)$ is a set of vertices $V' \subseteq V$ such that every edge $e \in E$ is incident to some vertex in V'.

Problem 4. Vertex Cover
Input: A graph $G = (V, E)$, a positive integer k.
Problem: Does G contain a vertex cover of size at most k?

The Vertex Cover problem somehow seems "easier" than the Clique problem, and from an algorithmic viewpoint this seems to be correct (although, no one can give a polynomial time algorithm). For example, in class the following natural algorithm was suggested.

Greedy Algorithm for Vertex Cover

1. On input G.
2. Let $S = \emptyset$, let $G' = G$.
3. Repeat until S is a vertex cover of G:
   (a) Choose the vertex v in G' with the most neighbours, add it to S.
   (b) Delete v from G'.
4. Output S.

It can be shown (although we will not do so here) that the vertex cover output by S will be no more than twice the size of the smallest vertex cover of G, and so in this sense this problem is "easy". However, we now show that closing this gap from 2 to 1 in the approximation ratio is at least as hard as solving the clique problem!

First, observe that if $S \subseteq V$ is a vertex cover of G, then $V \setminus S$ must not contain any edges of G, as if it did then S would not be a vertex cover. Let $\overline{G}$ be the *complement graph* of G obtained by removing every edge of G and adding every non-edge of G. Since $V \setminus S$ does not contain any edges, it follows that it is a clique in the graph $\overline{G}$. Thus, G has a vertex cover of size at most k if and only if $\overline{G}$ has a clique of size at least $n - k$.

Let us prove this formally. Suppose that G has a vertex cover $V' \subseteq V$ where $|V'| \leq k$, and we claim that $V \setminus V'$ is an independent set of size $|V| - |V'| = n - k$. If V' is a vertex cover then for each edge $uv \in E$ either $u \in V'$ or $v \in V'$. Equivalently, if neither $u \in V'$ nor $v \in V'$ then $uv \notin E$. Thus there is no edge connecting each pair of vertices $u, v \in V \setminus V'$, and so $V \setminus V'$ is an independent set.

Conversely, suppose G contains an independent set $V'' \subseteq V$ of size at least $n - k$. We claim that $V \setminus V''$ is a vertex cover. The argument is essentially the same: if $uv \in E$ then clearly $u \notin V''$ and $v \notin V''$. Thus at least one of u or v is in $V \setminus V''$, and so $V \setminus V''$ is a vertex cover. Clearly $|V \setminus V''| \leq n - k$, so the proof is complete.

The argument outlined above should look familiar: it is a *reduction*. Compare the next definition with Definition 2.11.

Definition 3.8. Let L_1, L_2 be languages. We say that L_1 is *polynomial-time many-one reducible* to L_2 (other synonyms: polynomial-time mapping reducible, Karp reducible), written

$L_1 \leq_p L_2$, if there is a function f computable by a polynomial-time, deterministic TM M such that for all $x \in \Sigma^*$,

$$x \in L_1 \Leftrightarrow f(x) \in L_2.$$

In this language we can re-state our observation about Vertex Cover and Clique. Note that the reduction we sketched above "goes both ways" — if we can solve Vertex Cover exactly then we can solve Clique exactly!

Proposition 3.9. $\mathrm{VertexCover} \leq_p \mathrm{Clique}$ *and* $\mathrm{Clique} \leq_p \mathrm{VertexCover}$.

3.3 Lecture 9: Interaction and NP, Vertex Cover Revisited, NP-Completeness

It may be helpful to think of NP in the following "interactive" setting. Suppose we had two algorithms: one, the *verifier*, wishes to decide L, but it is bounded to run in polynomial time. (Assuming that $\mathsf{P} \neq \mathsf{NP}$, this is impossible.) The verifier does, however, have a companion algorithm called a *prover*, which it can interact with. The prover has unlimited power — it can determine whether or not an input x is in L in a single computation step (in some sense it is like the oracles that we saw in Lecture 6). The verifier would like to use the prover to help him compute the language L, but there is a catch: the prover is untrustworthy. That is, if the verifier just asks the prover the question "if $x \in L$, then return 1, and if $x \notin L$, then return 0", then the prover may not be honest, and so it can not trust the single bit that the prover returns.

So, instead, the verifier must ask the prover a question such that the input x will be accepted if the prover is honest, but if the prover is dishonest and tries to convince the verifier that x is in L then the verifier can "catch the prover in the act" and reject correctly.

To be concrete, let us consider the Vertex Cover problem introduced in the previous lecture. As input, the verifier receives a string x that encodes a pair (G, k), where $G = (V, E)$ is a graph k is an integer, and it must determine whether or not G contains a vertex cover $V' \subseteq V$ with at most k vertices. To compute this, the verifier sends the prover the input string x that encodes G and k, and asks the prover to send back a string y encoding a vertex cover V' of G with at most k vertices. The prover responds with the string y instantly (this is the "certificate" y appearing in Definition 3.4). Now, the verifier knows that the prover may have lied, so it must verify that the string y does actually encode a vertex cover. So, first it checks that y encodes a subset of vertices $V' \subseteq V$ of size at most k. If not, the verifier rejects x and y instantly. Then it checks whether or not every edge $e \in E$ in the graph G has one of its endpoints in the set V'. If so, then it accepts (since the set V' is a vertex cover of size at most k); otherwise it rejects.

Let us state this verifier formally. When describing the verifier formally we ignore the exchange with the prover and instead assume that it has already received the string y.

Polynomial Time Verifier V for Vertex Cover

```
1. On input $x, y$.
2. Check that $x$ encodes a graph $G = (V, E)$ and a positive integer $k$. If not, reject.
3. Check that $y$ encodes a subset $V'$ of vertices of $G$ with $|V'| \leq k$. If not, reject.
4. For each edge $e \in E$:
    (a) Check if either endpoint of $e$ lies in $V'$. If not, reject.
5. Accept.
```

Now, following Definition 3.4, to show that Vertex Cover is in NP we must prove that for every string x we have that

$$x \in \text{VertexCover} \Leftrightarrow \exists y : V \text{ accepts } x, y$$

and V has time complexity $O(|x|^c)$ for some constant c and all inputs x. We give a detailed formal proof.

Proposition 3.10. *Vertex Cover is in* NP.

Proof. Let V be the verifier defined above. As stated before the theorem, to show that Vertex Cover is in NP we need to show that for every string x we have

$$x \in \text{VertexCover} \Leftrightarrow \exists y : V \text{ accepts } x, y$$

and V has time complexity $O(|x|^c)$ for some universal constant c.

First, let x be any string and suppose that $x \in$ VertexCover. We show that there exists a y such that V accepts x and y. By definition, since $x \in$ VertexCover there is a graph $G = (V, E)$ and a positive integer k such that x encodes the pair $(\langle G\rangle, \langle k\rangle)$, and there is a vertex cover $V' \subseteq V$ with $|V'| \leq k$ for G. Examining the definition of the verifier, we let y be the string encoding the set V'. Since V' is a vertex cover in G with at most k vertices it follows from the definition of V that V accepts (x, y).

Now, suppose that there is a string y such that V accepts x and y, and we show that $x \in$ VertexCover. By definition, this means that we need to show that x encodes a pair $(\langle G\rangle, \langle k\rangle)$ where G is a graph, k is a positive integer, and G has a vertex cover with at most k vertices.

Since V accepts (x, y), by line (2) in the definition of V we have that x encodes a graph G and a positive integer k. Similarly, by line (3) in V, we know that y encodes a set V' of at most k vertices of G. Finally, by (4) and (4a), the verifier V accepts (x, y) if and only if every edge of G has at least one endpoint in V'. This means that V' is a vertex cover of G, and so $x \in L$, as required.

Finally, we show that the algorithm runs in polynomial time. This is easy. We assume that we can efficiently encode and decode graphs and integers, and so steps (2) and (3) run in polynomial time. The loop in step (4) requires checking $O(|E'|)$ edges in G, and checking each edge requires, at worst, a loop over $O(|V'|)$ vertices. Thus the run-time of this step is $O(|E'||V'|)$, which is polynomial in $|x|$ since x encodes G. It follows that the verifier described is a polynomial time algorithm, as required. $\square$

NP-Hardness and Completeness. As we have discussed, we do not know whether or not $\mathsf{P} = \mathsf{NP}$, although it is widely believed that the two classes are different. Here we give a crucial tool — using polynomial-time reducibility — which is believed to give strong support for whether or not a language is in $\mathsf{NP} \setminus \mathsf{P}$.

Definition 3.11. Let L be a language. We say that L is NP-*Hard* if every language in NP polynomial-time many-one reduces to L (see Definition 3.8 for polynomial-time many-one reducibility). That is, for all $L' \in \mathsf{NP}$ we have

$$L' \leq_p L.$$

Further, we say that L is NP-*Complete* if $L \in \mathsf{NP}$ *and* L is NP-Hard.

This definition of completeness is identical to the definition of completeness we used in computability theory (Definition 2.16) *except* we use polynomial-time many-to-one reducibility instead of (unrestricted) many-to-one reducibility. The next proposition and corollary show why NP-Completeness is important with respect to the question of P vs. NP.

Proposition 3.12. *Let A and B be languages such that $A \leq_p B$. If B is in* P *then A is in* P.

Proof. Assume B is in P and let M_B be the polynomial-time algorithm computing B. We give a polynomial-time algorithm M_A computing A.

Since $A \leq_p B$, by Definition 3.8 there is a polynomial-time computable function f such that for all strings x

$$x \in A \Leftrightarrow f(x) \in B.$$

Here is an algorithm for A, using f and M_B.

Algorithm M_A for A.

1. On input x.
2. Simulate M_B on $f(x)$, and accept if and only if the simulation accepts.

Clearly M_A has polynomial time complexity since both M_B has polynomial-time complexity and f can be computed in polynomial time. We need to argue that M_A computes A, but this is also easy. By the definition of f, we have that $f(x)$ is accepted by M_B if and only if x is in A, and clearly M_A accepts x if and only if M_B accepts $f(x)$. Thus $\mathcal{L}(M_A) = L$. □

Corollary 3.13. *Let A be an* NP*-Hard language, and suppose that* P $\neq$ NP. *Then $A \notin P$.*

Proof. By contradiction, suppose that $A \in P$. Since A is NP-Hard, every language in NP reduces to A. Let L be any language in NP and not in P, which must exist if P $\neq$ NP. Then $L \leq_p A$, and so applying the previous proposition we get that $L \in$ P, which is a contradiction. □

Thus, under the assumption that P $\neq$ NP, if any language in NP is not in P it is the NP-Complete languages. So, what are some examples of NP-Complete languages? It may come as quite a surprise that both Clique and Vertex Cover are both NP-Complete!

Theorem 3.14. *Both Clique and Vertex Cover are* NP*-Complete.*

Proof. We have proven above that both of these languages are in NP, so we just need to prove that they are NP-Hard. We defer this to a later lecture. □

How can we show that a language L is NP-Complete? By definition, we need to show that L is in NP (by giving a poly-time verifier) and then show that L is NP-Hard. The easiest way to do this is to use the *transitivity* of reductions.

Proposition 3.15. *Let L_0 be an* NP*-Hard language, and suppose that $L_0 \leq_p L_1$. Then L_1 is* NP*-Hard.*

Proof. Let L be an arbitrary language in NP. Since L_0 is NP-Hard it follows that $L \leq_p L_0$. We show that $L \leq_p L_1$.

This follows immediately by composing the reductions. Let f be the polynomial-time computable function such that for every string x,

$$x \in L \Leftrightarrow f(x) \in L_0,$$

and let g be the polynomial-time computable function such that for every string y,

$$y \in L_0 \Leftrightarrow g(y) \in L_1.$$

It follows that for every string x,

$$x \in L \Leftrightarrow f(x) \in L_0 \Leftrightarrow g(f(x)) \in L_1.$$

The function $g(f(x))$ is clearly computable in polynomial time since both g and f are, and so it follows that $L \leq_p L_1$. Since L is an arbitrary language in NP it must be that L_1 is NP-Hard. $\square$

By Theorem 3.14 it follows that if Clique reduces to some language L or if Vertex Cover reduces to L then L is NP-Hard.

Summary. In this lecture we considered a different and (in the Lecturer's opinion, more intuitive way) to think about membership in NP via an interaction between a polynomial-time verifier and an unlimitedly powerful (though fickle) prover. However, note that giving such a verifier-prover relationship is not a *proof* of membership in NP. To show that a language L is in NP, you must follow Definition 3.4 and give a polynomial-time verifier for L. As we have seen in both the Clique and Vertex Cover example, this requires three steps:

1. Give an algorithm for the verifier V which takes as input a string x (which is the string to be tested for membership in L) and another string y (this is the "certificate" or "proof" that is used to help test membership of x.)

2. **Efficiency:** Prove that V runs in polynomial time *in the length of* $|x|$.

3. **Correctness:** Prove that for every input x,

$$x \in L \Leftrightarrow \exists y : V \text{ accepts } x, y.$$

   This itself requires two subcases: first you assume $x \in L$ and show that $\exists y$ such that V accepts x and y; second, you assume $\exists y$ such that V accepts x and y, and then show that $x \in L$.

Note that these three steps are not mutually exclusive — often you will be defining your verifier while simultaneously thinking about how to easily prove Efficiency and Correctness.

Then we defined the NP-analogue of completeness using polynomial-time reducibility. NP-Completeness is important since it gives strong evidence for being outside of P — as we have seen, if $\mathsf{P} \neq \mathsf{NP}$ then the NP-Complete languages must not be in P. By Definition 3.11, to show that a language is NP-Complete we must show that it is both in NP (by giving a verifier, as described before) and that it is NP-Hard. To show that it is NP-Hard, it is easiest to reduce from a language that we know is NP-Hard; at this point the only two examples are Clique and Vertex Cover (plus any languages covered in tutorial which were reduced to from Clique or Vertex Cover, or otherwise stated to be NP-Hard). By Definition 3.8, reducing a language A to a language B itself also requires three steps:

1. For any x in A determine a polynomial-time computable function f such that $x \in A$ if and only if $f(x) \in B$. Typically, you must pay careful attention to the requirement for membership in both A and B, and somehow "transform" legal inputs to A into legal inputs to B. This step is creative, and is best learned by examples (both reading and doing). See Tutorial 6 for several such examples.

2. **Efficiency.** Prove that the function f is computable in polynomial time.

3. **Correctness.** Prove that $x \in A$ if and only if $f(x) \in B$. This itself requires paying careful attention to the definitions of A, B, and your definition of f.

The same comment about the verifier also holds here: these three steps are not mutually exclusive, and often you will be constructing your reduction while simultaneously thinking about how to easily prove Efficiency and Correctness.

3.4 Lecture 10: More NP-Completeness. Self-reducibility.

3-SAT and More Reductions. In this lecture we prove Theorem 3.14 using Cook's Theorem about the NP-Hardness of SAT, which was introduced in Tutorial 7. For completeness we re-define SAT here. Let x, y be boolean variables, and recall the semantics of the basic operations $\wedge$ (AND), $\vee$ (OR), and $\bar{\cdot}$ (NOT):

- $x \wedge y = 1$ if and only if $x = 1$ and $y = 1$,
- $x \vee y = 1$ if and only if $x = 1$ or $y = 1$,
- $\overline{x} = 1$ if and only if $x = 0$.

A *boolean formula* ϕ on n variables is a sentence composed of the basic boolean operations and n propositional variables. For example,

$$(x \wedge y) \vee (\overline{x} \wedge \overline{y})$$

is a boolean formula which is true (outputs 1) if and only if $x = y$.

A *literal* is a variable x or a negation of variable $\overline{x}$. The formula ϕ is called a *clause* if it is an OR of a set of literals. So, the formula

$$x_1 \vee x_2 \vee \overline{x}_3$$

is a clause while

$$(x_1 \wedge x_2) \vee x_3$$

is not.

We say that ϕ is in *conjunctive normal form* (or CNF) if it consists of an "AND of ORs" of variables (equivalently, a conjunction of clauses), where the only negations in the formula appear on the variables. For example, the formula

$$\phi(x_1, x_2, x_3) = (x_1 \vee x_2 \vee \overline{x}_3) \wedge (\overline{x}_2 \vee x_3) \wedge (\overline{x}_1 \vee x_3)$$

is in conjunctive normal form, while neither of the formulas

$$\begin{aligned} \phi_0(x_1, x_2, x_3) &= x_1 \vee (x_2 \wedge x_3), \\ \phi_1(x_1, x_2, x_3) &= \overline{(x_1 \vee x_2)} \wedge (\overline{x}_2 \vee x_3) \end{aligned}$$

are in CNF: the first is not a conjunction of clauses, and in the second there is a negation which is not applied to a variable.

A boolean formula ϕ is *satisfiable* if there is an assignment z to the variables of ϕ such that $\phi(z) = 1$. The satisfiability problem (or SAT) problem asks whether or not a given boolean formula ϕ is satisfiable.

Problem 5. CNF-SAT
Input: A boolean formula ϕ in CNF.
Problem: Decide if ϕ is satisfiable.

Remarkably, this problem is NP-Complete (historically, it is the *first* NP-Complete problem). The fact that CNF-SAT is NP-Complete is known as Cook's theorem, named after Stephen Cook.

Theorem 3.16 (Cook's Theorem)**.** *CNF-SAT is* NP*-Complete.*

(If we have time we will give a proof of this theorem!) In Tutorial 7 we used Cook's Theorem to show that if we bound the number of literals in each clause of a CNF formula ϕ then the CNF-SAT problem remains NP-Complete.

Problem 6. k-SAT
Input: A boolean formula ϕ in CNF where every clause has at most k variables.
Problem: Decide if ϕ is satisfiable.

Theorem 3.17. *For every $k \geq 3$, k-SAT is* NP*-Complete.*

The case where $k = 3$ (i.e. 3-SAT) is a *very important* problem. The inputs are of an extremely simple form (a CNF where every clause has at most three literals), and as a result it is often used as a starting point for reductions to show that languages are NP-Hard. Using 3-SAT we will "properly" show that Clique is NP-Hard, and then using Proposition 3.9 we immediately get that Vertex Cover is also NP-Hard.

Theorem 3.18. *3-SAT $\leq_p$ Clique.*

Proof. Let $\phi = C_1 \wedge C_2 \wedge \cdots \wedge C_m$ be any 3-SAT formula, and for each $i = 1, 2, \ldots, m$ let $C_i = a_i \vee b_i \vee c_i$, where each of a_i, b_i, c_i is a variable or its negation. Given two literals ℓ_1, ℓ_2 we say that ℓ_1 and ℓ_2 are *consistent* if $\ell_1 \neq \overline{\ell_2}$. The main observation of the reduction is as follows: any satisfying assignment to ϕ must choose, for each $i = 1, 2, \ldots m$, a literal from a_i, b_i, c_i and set it to true, and each of the true literals must be consistent with one another. From ϕ we construct a graph G which has a group of vertices for each clause, and we connect vertices together corresponding to consistent literals; from this, a clique in the graph G will correspond exactly to a satisfying assignment to ϕ.

Let us describe the reduction in more detail. For each clause $C_i = a_i \vee b_i \vee c_i$ in ϕ, create three new vertices $v(a_i), v(b_i), v(c_i)$ in G. Then for each pair of vertices $v(x_i), v(y_j)$ where x_i, y_j occur in different clauses, add an edge to G connecting $v(x_i), v(y_j)$ if and only if the literals x_i and y_j are consistent. Note that the only edges in the graph G connect vertices whose underlying literals are in different clauses, and not vertices within the same clause. Finally, set $k = m$, the number of clauses. The graph G can clearly be constructed from ϕ in polynomial time, since we create three vertices for each clause in ϕ and add, at worst, $O(m^2)$ edges.

Now we show that the formula ϕ is satisfiable if and only if the graph G has an m-clique. First, assume that ϕ is satisfiable, and we show that G has an m-clique. Let $\vec{x}$ be a satisfying assignment to ϕ, and for each clause $C_1, C_2, \ldots, C_m$ choose a literal $y_1, y_2, \ldots, y_m$ from the clause, respectively, which is set to true by $\vec{x}$. Clearly each of these literals are consistent since they are all simultaneously satisfied by $\vec{x}$. By the definition of G each pair of vertices

$v(y_i), v(y_j)$ are connected if and only if y_i and y_j are consistent, and so it follows that the set of vertices $\{v(y_1), v(y_2), \ldots, v(y_m)\}$ is an m-clique in G.

Conversely, assume that G contains an m-clique $\{v(z_1), v(z_2), \ldots, v(z_m)\}$. By definition of G, each of the underlying literals $z_1, z_2, \ldots, z_m$ are consistent, and also must be chosen from different clauses. Since they are each consistent, there is an assignment $\vec{x}$ to the variables of ϕ which simultaneously sets each of these literals to true. Moreover, since each must be chosen from different clauses (and since there are m of these literals), it follows that $\vec{x}$ satisfies all of the clauses ϕ. Thus ϕ is satisfiable. $\square$

Afterwards, we gave a nice example of a reduction which is significantly more involved than the reductions you have seen so far. We state the problem, but leave the reader to study the proof in the Sipser textbook (see Theorem 7.35). Let $G = (V, E)$ be a *directed* graph. A *Hamiltonian path* in G is a path in G which visits every vertex in G exactly once (the famous *Travelling Salesperson Problem* essentially asks for a minimum-cost Hamiltonian path in an edge-weighted graph). We show that just deciding if a graph G contains a Hamiltonian path is NP-Complete by a reduction from 3-SAT.

Problem 7. Hamiltonian Path
Input: A directed graph $G = (V, E)$.
Problem: Decide if G contains a Hamiltonian path.

Theorem 3.19. *Hamiltonian Path is* NP*-Complete.*

Proof. See Sipser, Theorem 7.35. $\square$

Search-to-Decision Reductions. By now, you have probably asked yourself why we focus so much on *decision problems* (i.e. determining whether or not an input x is in some language L), when in practice we usually want to output "something", rather than just decide whether or not something exists. For example, it is one thing to say whether or not a graph G contains a Hamiltonian path, but what if we actually wanted to *find* a Hamiltonian path in G? It may be the case that the decision problem is "easier" than the search problem, in that you could do some clever mathematical tricks to determine if a Hamiltonian path exists in a graph G without actually needing to find it.

Another obvious computational task is *optimization*. For another example, suppose that instead of simply deciding whether or not a graph G has a big clique, we actually want to find the *largest* clique in G? This problem has many applications in social networks and bioinformatics.

In this section we introduce a framework for these kinds of problems (which we group together as *search problems*). In fact, we will be able to show that for many problems in NP, the "search variant" is no harder than the "decision variant", in that if we can solve the decision problem then we can use the algorithm to solve the search problem as well!

Definition 3.20. Let Σ be an alphabet. A *search problem* is a relation $S \subseteq \Sigma^* \times \Sigma^*$. A Turing Machine M *computes* a search problem S if for all input strings $x \in \Sigma^*$ the machine M either

1. Halts and accepts with a string $y \in \Sigma^*$ on the tape for which $(x, y) \in S$, or

2. Halts and rejects if no such string y exists.

The search problem S is *polynomial-time computable* if there is a Turing Machine M with polynomial time complexity computing S.

A natural example of a search problem is the one associated with CNF-SAT.

Problem 8. CNF-SAT-Search
Input: A Boolean formula ϕ in CNF.
Problem: Output a satisfying assignment to ϕ, or halt and reject if no such assignment exists.

Using *oracles* (cf. Definition 2.19), we introduce a notion of reduction between search problems and decision problems. The idea is to think of the oracle as a very efficient "subroutine" or "function" computing L, and then show that if such a function existed then we can use it to compute the search problem.

Definition 3.21. Let $S \subseteq \Sigma^* \times \Sigma^*$ be a search problem and let $L \subseteq \Sigma^*$ be a language. A *polynomial-time search-to-decision reduction* from S to L is any polynomial-time Turing Machine M deciding S with an oracle for L.

The next proposition shows that if $L \in \mathsf{P}$ and S search-to-decision reduces to L then we can also compute S in polynomial time.

Proposition 3.22. *Let S be a search problem, let L be a language, and suppose that S polynomial-time search-to-decision reduces to L. If $L \in \mathsf{P}$ then there is a polynomial-time algorithm computing S.*

Proof. Suppose that M_0 is the polynomial-time algorithm computing L, and let c_0 be a constant such that the time complexity of M_0 is $O(n^{c_0})$. Similarly, let M_1 be the search-to-decision reduction from S to L, and let c_1 be a constant such that the time complexity of M_1 is $O(n^{c_1})$. Using M_0 and M_1 we create a new TM M computing S running in polynomial-time. The idea is very simple: we just replace every oracle query that M_1 makes with a simulation of M_0.

Let us define M. On input $x \in \Sigma^*$, simulate M_1 until M_1 makes a call to the L oracle on a string $z \in \Sigma^*$. When M_1 makes this oracle call, stop simulating M_1 and instead simulate M_0 on z. When the simulation of M_0 halts, continue the simulation of M_1 as if the oracle returned whatever M_0 returns. Since M_0 computes L it clearly follows that M computes S. Moreover, since M_1 runs in time $O(n^{c_1})$ it can make at most $O(n^{c_1})$ calls to the oracle, so the time complexity of M is at most $O(n^{c_0 c_1})$. Thus S is polynomial-time computable. $\square$

Using search-to-decision reductions we will be able to show that finding a solution is no harder than deciding if a solution exists for many natural NP problems. (In fact, on the next assignment, you will show that this is true for *every* NP-Complete problem.)

Proposition 3.23. *There is a search-to-decision reduction from the CNF-SAT-Search problem to the CNF-SAT problem.*

Proof. Following the definition of search-to-decision reductions, we give a polynomial-time algorithm computing CNF-SAT-Search using an oracle for CNF-SAT. The idea of the algorithm is quite simple: given a boolean formula ϕ in CNF, we set the variables of the formula one at a time, using the oracle for CNF-SAT after each assignment to determine whether or not the remaining formula is still satisfiable.

Algorithm for CNF-SAT-Search

```
1. On input $\phi$.
2. Let $x_1, x_2, \ldots, x_n$ be the variables of $\phi$.
3. Query the CNF-SAT oracle for $\phi$. If it is not satisfiable, halt and reject.
4. For $i = 1, 2, \ldots, n$.
    (a) Set $x_i = 1$, and let $\phi'$ be the formula obtained from $\phi$ after setting $x_i = 1$ everywhere in $\phi$.
    (b) Query the CNF-SAT oracle for $\phi'$. If $\phi'$ is satisfiable, set $\phi = \phi'$ and continue. Otherwise, set $x_i = 0$, and simplify $\phi$ by setting $x_i = 0$ everywhere $x_i$ appears in $\phi$.
5. Output the assignment to $x_1, x_2, \ldots, x_n$.
```

Clearly the above algorithm runs in polynomial time: the oracle calls take $O(1)$ time, there is a single $O(n)$ loop, and each simplification step can be implemented in polynomial time. Thus we just need to prove that the above algorithm solves CNF-SAT-Search.

By construction, the algorithm halts and rejects if and only if the formula ϕ is unsatisfiable, so assume that ϕ is satisfiable. In this case we must prove that the assignment output by the algorithm is a satisfying one. At the end of the ith iteration of the loop, it is easy to see that we obtain a formula ϕ_i on variables $x_{i+1}, x_{i+2}, \ldots, x_n$ which is satisfiable by the definition of the algorithm (in particular, the correctness of the CNF-SAT oracle). Moreover, the formula ϕ_i is obtained from ϕ by restricting the first i variables $x_1, x_2, \ldots, x_i$ to certain values. Since ϕ_i is satisfiable for each $i = 1, 2, \ldots, n$ by restricting the first i variables it follows that the assignment output by the loop is a satisfying assignment to ϕ. □

Summary. In today's lecture we reviewed some more complicated reductions and also had a first look at search problems. The more complicated reductions related a problem in logic (3-SAT) to problems in graph theory (Clique, Hamiltonian Path), and so in general it is important to observe that reductions can be made between problems which are superficially very different.

We also reviewed search-to-decision reductions. The general framework for constructing such a reduction from a search problem S to a language L is quite simple:

1. Construct an algorithm M computing S which has access to an oracle for L.

2. Prove that M runs in polynomial time (assuming the oracle queries take $O(1)$ time each).

3. Prove that M is correct.

3.5 Complexity Review

Here is a survey of everything we touched on in the second half of the course.

Models of Efficient Computation How we measure the running time of a Turing Machine. Definition of the complexity class P — the languages computable in polynomial time.

NP **and Reducibility** Polynomial-time verification. Definition of the class NP. Definition of the Clique and Vertex Cover problems. Introduction to reductions (reduction from Clique to Vertex Cover and vice-versa).

NP **and Interaction. Hardness and Completeness.** NP as an interaction between a verifier and a prover. Formal verifier for Vertex Cover. NP-Hardness and NP-Completeness. Overview of proving NP-Hardness and membership in NP.

Self-reducibility Search problems and search-to-decision reductions.

For the purpose of NP-Hardness reductions it is best to memorize a few "flexible" languages and use them repeatedly; for this purpose I would recommend 3-SAT, Vertex Cover, and Clique. Also remember the summaries of reductions presented in Lecture 9 and Lecture 10.

Problem 9. 3-SAT
Input: A boolean formula ϕ in CNF where every clause has at most 3 variables.
Problem: Decide if ϕ is satisfiable.

Problem 10. Vertex Cover
Input: A graph $G = (V, E)$, a positive integer k.
Problem: Does G contain a vertex cover of size at most k?

Problem 11. Clique
Input: A graph $G = (V, E)$, a positive integer k.
Problem: Decide if G contains a clique with at least k vertices.

References

[1] Michael Sipser. *Introduction to the Theory of Computation.* PWS Publishing (1997).