Computational Complexity Theory **Summer HSSP 2018**

Week 1: Computability and Decision Problems

Dylan Hendrickson *MIT Educational Studies Program*

1.1 The Halting Problem

We'll start by considering whether programs can solve certain problems at all. Most problems you encounter can be solved by a program, though it may take a huge amount of time, perhaps brute forcing its way through every potential solution. But there are some problems that it's not clear how to write a program to solve. For example, the halting problem: given a program and some input, does the program eventually halt on that input? Some programs will keep running in an infinite loop, and it's not always obvious when this will happen. (Often, it will be easy to tell that a program will halt, or that it will loop, but we want an algorithm that *always* tells us whether a program halts.) One attempt would be to run the program in question; if it halts, then you can answer 'yes.' But if it doesn't halt, then you'll never find out; maybe it will keep running forever or maybe it will suddenly stop if you simulate it for a little longer.

Theorem 1.1. *There is no program that solves the halting problem.*

Proof. Suppose there were a program H which solved the halting problem; that is, for any program P and input x, $H(P, x)$ answers either 'yes' or 'no' depending on whether $P(x)$ halts. We'll show that this assumption leads to a contradiction, so it must be false. Use H to construct a new program G which does the following when input a program P:

- Run $H(P, P)$; this tells us whether P halts when given its own code as input.
- If $H(P, P)$ is 'yes,' enter an infinite loop.
- If $H(P, P)$ is 'no,' halt.

Now $G(P)$ halts exactly when $H(P, P)$ is 'no,' which is when $P(P)$ loops. Let's consider what happens when we run G on its own code, by plugging in G for P. We find that $G(G)$ halts exactly when $G(G)$ loops, which is impossible. So our assumption that H exists must be wrong. □

1.2 Decision Problems

When we talk about what programs are capable of, we usually mean the classes of yes/no questions they can answer. Such a class of yes/no questions is called a *decision problem*. We've already seen our first decision problem: given a program P and input x, does $P(x)$ halt? Notice that it's uninteresting to consider only a single yes/no question, since it has an answer of either 'yes' or 'no,' and either the program the outputs 'yes' or the program that outputs 'no' solves it (even if we don't know which one). We instead consider infinite families of yes/no questions, and look for programs that can answer all of them. When a program answers 'yes,' we say it *accepts* the input, and when it answers 'no,' we say it *rejects* the input.

If there is a program that answers every yes/no question in a decision problem, we say that the program *decides* the decision problem, and that the decision problem is *decidable*. Most decision problems you care about are decidable, but we've just seen that the halting problem isn't decidable.

If there is a program that accepts whenever the answer is 'yes,' and either rejects or runs forever whenever the answer is 'no,' we say that the program *recognizes* the decision problem, and that the decision problem is *recognizable*. The algorithm which, given inputs P and x, runs $P(x)$ until it halts and then accepts, recognizes the halting problem. If $P(x)$ halts, this algorithm will eventually accept, and if $P(x)$ doesn't halt, this algorithm runs forever. So the halting problem is recognizable but not decidable.

Deciders have to always given an answer, but recognizers only have to answer when the answer is 'yes,' and are allowed to loop when the answer is 'no.' What if we allow the program to loop when the answer is 'yes' but not when it's 'no'? Then we say that the program *corecognizes* the decision problem, and the decision problem is *corecognizable*. For example, take the negation of the halting problem: given a program P and input x, does $P(x)$ run forever? The program which runs $P(x)$ and rejects when it halts corecognizes this decision problem; if the answer is 'no,' then $P(x)$ halts and the program rejects, and if the answer is 'yes' then it runs forever.

We are now in a position to prove some results about decidability, recognizability, and corecognizability, but I'd like to introduce some notation first. If L is a decision problem, we write $x \in L$ to mean x is in L, or the answer to the question corresponding to x is 'yes.' We write $x \notin L$ if the answer to the question corresponding to x is 'no.' The *complement* $\overline{L}$ of L is the decision problem with all answers opposite those of L, so $x \in L$ exactly when $x \notin \overline{L}$ and vice-versa. Note that $\overline{\overline{L}} = L$. I'll also use the shorthand 'iff' for 'if and only if.'

(I might sometimes say 'language' instead of 'decision problem'; the terms are essentially equivalent, but I think it's easier to think in terms of decision problems. A language is a set of inputs, which you think of as the inputs for which the answer is 'yes.')

Lemma 1.2. *A decision problem L is recognizable iff $\overline{L}$ is corecognizable.*

Exercise 1. Prove Lemma 1.2. Once you understand the notation, you should be able to see that it's not saying anything particularly deep or surprising.

Lemma 1.3. *A decision problem L is decidable iff it is both recognizable and corecognizable.*

Proof. There are two directions to prove; one is easy. First suppose L is decidable. The program that decides L both recognizes and corecognizes L, so L is recognizable and corecognizable.

Now suppose L is recognizable and corecognizable. Then there are programs F and G which recognize and corecognize L, respectively. That means that whenever $x \in L$, F accepts x (and G either accepts or loops on x), and whenever $x \notin L$, G rejects x (and F either rejects or loops on x). We construct a new program P which simulates F and G in parallel. The exact nature of the simulation isn't important; maybe it alternates running single lines of code from F and G, keeping them separate from each other. If the simulation of F accepts, P accepts, and if the simulation of G rejects, P rejects. We know that exactly one of these will happen, depending on whether $x \in L$. If $x \in L$, P will accept x, and if $x \notin L$, P will reject x. Thus P decides L, so L is decidable. □

Since we know the halting problem is recognizable but not decidable, it must not be corecognizable.

There's another equivalent way to define recognizability (actually multiple, but one that is important to us). Let L be a decision problem. We will see that L is recognizable if there is a program P which always halts, such that for every $x \in L$, there is some positive integer n such that $P(x, n)$ accepts, and if $P(x, n)$ accepts, then $x \in L$. We say that n is a *certificate* for x; we can use n convince the program that the answer to the question corresponding to x is 'yes.'

Theorem 1.4. *This is an equivalent definition of recognizability. That is, a decision problem L is recognizable iff there is a decidable decision problem D such that*

- *If $x \in L$, there is a positive integer n such that $(x, n) \in D$*
- *If $(x, n) \in D$, then $x \in L$.*

Before reading the proof, make sure you understand what this theorem is saying. If you're feeling up to it, try to prove it yourself. The idea of the proof is that the certificate n that convinces you that $x \in L$ is the number of steps the program that recognizes L takes to accept on input x.

Proof. Suppose L is recognizable. Let F be a program that recognizes L. We construct a decision problem D satisfying the conditions: given x and n, does $F(x)$ accept within n steps? If $x \in L$, then $F(x)$ must accept at some point, so there is some n such that $(x, n) \in D$. If $(x, n) \in D$, then $F(x)$ accepts within n steps, so in particular $F(x)$ accepts, and thus $x \in L$. Finally, D is decidable; the program which, on input x and n, simulates $F(x)$ for n steps and accepts if the simulation accepted and rejects otherwise, decides D.

Now suppose there is a decision problem D satisfying the conditions decided by a program P. We write a program that recognizes L. On input x, for each positive integer n in order, run $P(x, n)$. If $P(x, n)$ accepts, accept; otherwise move on to the next n.

If $x \in L$, then this program will eventually find an n such that $(x, n) \in D$, and accept. If $x \notin L$, then no such n will work, and the program will run forever. □

Exercise 2. Give an equivalent definition of corecognizability along the lines of Theorem 1.4, and prove that it's equivalent (you can, but don't have to, use Theorem 1.4 in your proof).

Decision problems are organized into *complexity classes*. A complexity class contains all the decision problems that can be 'easily solved' by a program; the meaning of 'easily solved' is what distinguishes different complexity classes. When 'easily solved' means 'decided,' we have the complexity class of decidable problems, called **R** (for 'recursive,' an older word for 'decidable'). When 'easily solved' means 'recognized' or 'corecognized,' we have the classes **RE** and **coRE** (for 'recursively enumerable,' an older term describing an alternate definition of 'recognizable'). There are hundreds of named complexity classes, a few of which we'll get to know in this class. A table of some of the important ones for our purposes is at the end of these notes; if you want to see more, visit `complexityzoo.uwaterloo.ca` for descriptions of complexity classes or `https://www.math.ucdavis.edu/~greg/zoology/diagram.xml` for an interactive visualization.

We define the complement $\overline{\mathbf{S}}$ of a complexity class **S** as the collection of the complements of languages in **S**. Lemma 1.2 can then be expressed as $\mathbf{coRE} = \overline{\mathbf{RE}}$. Lemma 1.3 can be expressed as $\mathbf{R} = \mathbf{RE} \cap \mathbf{coRE}$ (the symbol $\cap$ is for intersection; $A \cap B$ is the collection of things that are in both A and B). The relation between **R** and **RE** is analogous to the relation between **P** and **NP** (complexity classes we'll see more of later), so it would be reasonable to use **NR** for **RE**, the N standing for 'nondeterministic,' which sort of means 'with guessing.'

1.3 Where we're going

In this class, we'll get to know some complexity classes including **P**, **NP**, **PSPACE**, and maybe **L**, **NL**, and others. These are all defined by replacing 'easily solved' with various precise definitions, such as only allowing a relatively small (polynomial) amount of space or time with which to answer yes/no questions. We'll introduce the concept of *completeness* for a complexity class, which is a way of getting at 'as hard as possible' among the decision problems in the class. We'll find complete problems for various complexity classes, and learn how to use them to find more complete problems. We've already seen one: the halting problem is **RE**-complete.

In order to prove that problems are complete for **NP** or other complexity classes, we'll introduce the concept of Turing machines, which is a simple model of computation that is as powerful as any computer, but that it's relatively easy to prove things about. You can think of Turing machines as a bare-bones, hard to work with programming language.

Ultimately, we'll be able to take puzzles like Sudoku of TipOver, games like Mario or Chess or puzzles like Rush Hour, and games like Braid or Recursed, and show that they're **NP**-complete, **PSPACE**-complete, and **RE**-complete, respectively.

Aside: It's pretty easy to show many inclusions of complexity classes, such as

$$\mathbf{L} \subset \mathbf{NL} \subset \mathbf{P} \subset \mathbf{NP} \subset \mathbf{PSPACE} \subset \mathbf{EXP} \subset \mathbf{NEXP} \subset \mathbf{R} \subset \mathbf{RE}.$$

However, aside from a few examples, notably $\mathbf{L} \neq \mathbf{PSPACE}$, $\mathbf{P} \neq \mathbf{EXP}$, and the results we already have about **RE** and **R**, it's extremely difficult to show that complexity classes are different. The famous **P** vs **NP** problem conjectures that $\mathbf{P} \neq \mathbf{NP}$, and it seems very likely to be true, but nobody has a good idea of how we could prove it (see `https://www.scottaaronson.com/papers/pnp.pdf` for a detailed, somewhat accessible survey). In fact, it's possible given our current knowledge that many of the listed complexity classes our the same. If $\mathbf{P} = \mathbf{PSPACE}$, then most of what we do in this class would be silly in hindsight, since we're mostly studying problems somewhere between **P** and **PSPACE** and trying to classify them, but it's possible this classification is meaningless. However, this seems extremely unlikely, and hopefully this class will help you understand why that is.

Name	Stands for	Meaning of 'easily solved'	Complete problem
L	log space	answered in logarithmic space	undirected reachability
NL	nondeterministic **L**	verified in logarithmic time	directed reachability
P	polynomial	answered in polynomial time	formula evaluation
NP	nondeterministic **P**	verified in polynomial time	formula satisfiability
coNP	complement of **NP**	verified false in polynomial time	formula unsatisfiability
PH	polynomial hierarchy	generalization of **NP**	
PSPACE	polynomial space	answered in polynomial space	quantified boolean formula
EXP	exponential	answered in exponential time	halt in k steps
NEXP	nondeterministic **EXP**	verified in exponential time	succinct Hamiltonian path
EXPSPACE	exponential space	answered in exponential space	regex equivalence
R	recursive	answered at all	
RE	recursively enumerable	verified at all	program halts
coRE	complement of **RE**	verified false at all	program loops
AH	arithmetical hierarchy	generalization of **RE**	