Computational Complexity Theory **Summer HSSP 2019**

Week 2: Reductions and Completeness

Dylan Hendrickson *MIT Educational Studies Program*

2.1 Some Decision Problems

Recall that decision problems are families of yes/no questions. Last time, we saw two decision problems, which we'll name:

HALT: given a program and an input, does the program halt (i.e. stop running rather than enter an infinite loop) when given that input?

EVERACCEPT: given a program, is there any input on which the program accepts (i.e. outputs 'yes')?

Now we'll introduce a few more decision problems. First, however, we need to define the objects they ask questions about.

Definition 2.1. A *circuit* is a collection of AND, OR, and NOT *gates* connected by wires. Some of the wires are the *input*. The circuit computes a boolean function, meaning it takes some number of True or False inputs and outputs True or False. Each gate computes some local function of its inputs:

- The output of an AND gate is True exactly when *all* of its inputs are True.
- The output of an OR gate is True exactly when *at least* one of its inputs is True.
- The output of an NOT gate is True exactly when its input is False.

I might use 0/1 instead of False/True.

For this to work, we need some technical conditions:

- Wires connect to the output of one gate (or an input to the circuit) to the input of another gate.
- NOT gates have exactly one input; OR and AND gates can have any number.
- The output of a gate can be connected to any number of inputs of other gates.
- One gate is designated as the *output*, which says what the circuit outputs.
- There can't be any loops in the circuit moving from input to output.

We have some decision problems about circuits:

CIRCUITEVAL[UATION]: given a circuit and input to the circuit, is the output True?
CIRCUITSAT[ISFIABILITY]: given a circuit, is there an input such that the output is True?

Definition 2.2. A *boolean formula* is a sequence of variables connected with AND, OR, and NOT: for example, x AND (NOT y OR z). We abbreviate this as $x \,\&\, (\neg y \vee z)$. An *assignment* for a boolean formula sets each variable to True (1) or False (0). An assignment *satisfies* the formula if the formula evaluates to

True. For example, $x = 1, y = 0, z = 1$ satisfies the above formula but $x = 1, y = 0, z = 0$ doesn't. A formula is *satisfiable* if there is an assignment which satisfies it.

A boolean formula is *k-cnf* if it is a collection of *clauses* join by AND, where each clause is a collection of up to k *literals* join by OR, where each literal is either a variable or NOT a variable. For example, this is a 3-cnf formula:

$$(x \vee \neg y \vee z) \ \& \ (w \vee \neg x \vee \neg z) \ \& \ (\neg x \vee \neg w \vee y) \ \& \ (\neg z \vee w \vee \neg x).$$

Decision problems about boolean formulas:

FormulaEval: given a boolean formula and assignment, does the assignment satisfy the formula?
FormulaSat: given a boolean formula, is it satisfiable?
2SAT: given a 2-cnf boolean formula, is it satisfiable?
3SAT: given a 3-cnf boolean formula, is it satisfiable?

Definition 2.3. A *graph* is a collection of *vertices* (dots), some pairs of which are connected by *edges* (lines).

Decision problems about graphs:

Path: given a graph, is there a path (along edges) from a specified vertex to another specified vertex?
Ham[iltonian]Path: given a graph, is there a path from one vertex to another which visits each vertex exactly once? 2Color: given a graph, can you color the vertices with 2 colors such that no two vertices connected by an edge are the same color?
3Color: given a graph, can you color the vertices with 3 colors such that no two vertices connected by an edge are the same color?

Exercise 1. Write down some instance of one of the above decision problems. Is the answer to each instance yes or no?

Exercise 2. Find an algorithm to solve one of the above decision problems. Estimate whether the algorithm is fast or slow. (if I ask about an instance with 100 gates/variables/vertices, how long will it take you to answer it?)

2.2 Reductions

Suppose I want to solve some decision problem A, and I have a magic box which solves decision problem B. How can I use my magic box to solve A?

One way is using a *reduction* [1] from A to B; that is, a way to convert an instance of A into an instance of B which has the same answer. Then, to solve an instance of A, I can convert it to an instance of B, stick it in my magic box, and get the answer.

More explicitly, a *reduction* from A to B is an algorithm which takes as an input an instance x of A and outputs an instance $f(x)$ of B. We require that $x \in$ A if and only if $f(x) \in$ B; in words, the answer to the B-instance we construct is the same as the answer to the original A-instance.

We saw an example of a reduction from Halt to EverAccept last week: given an instance of halt which is a program P and an input x, output the program which runs $P(x)$ and then outputs 'yes' on input 0, and immediately outputs 'no' on any other input.

[1] We'll only talk about what are called 'many-one' reductions in this class; there are other notions of reduction which allow e.g. using the magic-B-box multiple times.

Exercise 3. For any decision problem A, find a reduction A $\to$ A.

Lemma 2.4. *If there are reductions* A $\to$ B *and* B $\to$ C*, then there's a reduction* A $\to$ C*.*

Exercise 4. Prove Lemma 2.4

If there is a reduction A $\to$ B, we think of A being *at least as easy* as B. If we know how to solve B, we can use that knowledge together with the reduction to solve A. Rephrasing, we can say B is *at least as hard* as A.

We now have a formal notion of relative difficulty of decision problems. If a decision problem is at least as hard as every decision problem in some complexity class **C**, we say that it's **C**-hard. If the decision problem is also in **C**, we say it's **C**-complete. Formally:

Definition 2.5. Let B be a decision problem and **C** be a complexity class. If for every A $\in$ **C**, there's a reduction A $\to$ B, we say B is ***C-hard***. If in addition B $\in$ **C**, we say B is ***C-complete***.

The **C**-complete problems are the hardest problems in the class **C**; if you can solve one of them, you can solve every decision problem in **C**.

Theorem 2.6. HALT *is* ***R****E-complete.*

Proof. We've seen HALT $\in$ **R**E (simulate $P(x)$ until it halts; then accept), so we need to show that HALT is **R**E-hard. Let A $\in$ **R**E; we need a reduction from A to HALT.

Since A $\in$ **R**E, we know there's a program P which recognizes A (accepts when the answer is 'yes' and rejects or loops forever when the output is 'no'). We construct a program Q which does the following:

- Run P on your input.
- If P accepts, halt; if P rejects, loop forever.

Given an instance x of A, our reduction outputs the pair (Q, x) of a program and an input, which is an instance of HALT.

If the answer to the A-instance is 'yes,' $P(x)$ accepts, so $Q(x)$ halts and thus the answer to the HALT-instance is 'yes.' If the answer to the A-instance is 'no,' $P(x)$ rejects or loops, so $Q(x)$ loops forever and thus the answer to the HALT-instance is 'no.' So this is a reduction A $\to$ HALT, as desired. □

Exercise 5. Suppose two decision problems A and B are both in **R**; that is, they're decidable. Show that there's a reduction A $\to$ B.

2.3 Smaller Complexity Classes

Some problems are decidable, but not in a practical amount of time. So we want to consider the complexity class of problems which can be solved quickly. How can we define 'quickly'? Some observations:

- The time solving a problem takes should depend on how complicated the instance is, so we should allow more time for bigger instances.
- I could, in advance, make a giant table of the answers to all instances with size less than 1000, and thus answer all such instances very quickly. So we should care about what happens for 'very large' instances, since any lookup table you devise has to stop working eventually.

- Computers vary in speed, and keep getting faster. So we shouldn't care much about a problem taking twice as long to solve: we'll be able to solve it just fine a few years from now.

It turns out that a good way to define 'quickly' is *polynomial time*. This means that your program must take time at most cn^k for some fixed c and k. Here n is the 'size' of the input; e.g. the number of gates and wires in a circuit or the number of vertices and edges in a graph. I might write $O(n^k)$ to mean 'on the order of n^k, ignoring a constant factor.'

Roughly speaking, a polynomial-time algorithm is allowed to run through each piece of the input, but not over each *subset* of the input. For example, consider the following algorithms:

To solve CircuitEval:

- Go through each gate in the circuit, in the order from inputs to outputs.
- For each one, look it its input values to compute the output value, and send it to the gates connected to the output.
- Accept or reject based on the output gate.

To solve CircuitSat:

- Try each setting of the input wires.
- For each setting, run the algorithm to CircuitEval. If it accepts, accept.
- Reject.

The CircuitEval algorithm requires iterating over gates, and for each gate, iterating over the inputs. If there are at most n gates and each has at most n inputs, this takes time $O(n^2)$, which is polynomial. On the other hand, if the circuit for CircuitSat has n input wires, there are 2^n settings for them, which is bigger than polynomial, [2] so this algorithm takes longer than polynomial time. That doesn't mean there isn't a smarter algorithm that solves CircuitSat in polynomial time, but it's not this one.

We define more complexity classes:

P: decision problems which have a polynomial-time algorithm.
NP: decision problems which can be checked in polynomial time. Formally: A $\in$ **NP** if there's an polynomial-time algorithm P such that $x \in A$ exactly when there's a (polynomial-size) certificate c such that $P(x, c)$ accepts. Note the similarity to the alternative definition of **RE** from last week; we could use the name **NR** instead of **RE**.

We just saw that CircuitEval $\in$ **P**. Also CircuitSat $\in$ **NP**, since the input which makes the output true can serve as a certificate. Most likely CircuitSat isn't in **P**, but nobody knows for sure.

Exercise 6. Of the decision problems defined in Section 2.1, which are in **P**? Which are in **NP**?

Since **P** and **NP** are inside **R**, Exercise 5 tells us reductions aren't very useful. Instead we use *polynomial-time reductions*. Similarly to Exercise 5, there are polynomial-time reductions between all problems in **P**.

Exercise 7. Look for polynomial-time reductions between decision problems defined in Section 2.1. The easiest interesting ones are probably 3SAT $\to$ FormulaSat and FormulaSat $\to$ CircuitSat; I encourage you to try your hand at the harder ones.

[2] To illustrate that 2^n is bigger than polynomial, let's compare it to a big polynomial like n^{10}. At $n = 1000$, 2^n is a number with over 300 digits, but n^{100} has only 30 digits. In general 2^n is much bigger than n^k for large enough n, when k is fixed.