CS120: Intro. to Algorithms and their Limitations Hesterberg & Vadhan

Lecture 22: Uncomputability by Reductions

Harvard SEAS - Fall 2022 *2022-11-17*

1 Announcements

Recommended Reading:

- MacCormick Chapter 7

2 Universal Programs

Today we will study algorithms for analyzing programs. That is, we'll consider computational problems whose inputs are not arrays of integers, or graphs, or logical formulas, but ones whose inputs are themselves programs.

This is not quite the same as when we, in discussing computational models, said that for every word-RAM program P, there was a RAM program P' which was equivalent: halted on the same inputs, and gave the same answers. Rather, today we want to go one step beyond: instead of building, for each program P, a different program P' which does the same thing as that P, we want one program U which takes P as input and simulates it.

Theorem 2.1.

Proof idea.

□

We can also write a Word-RAM program that simulates RAM programs and vice-versa.

Q: What would change in the theorem if we want U to be a Word-RAM program but allow P to be a RAM program?

Importance of Universal Programs:

3 The Halting Problem

Our definition of a universal program needed the condition "U halts on input (P, x) iff P halts on x" before we could say that $U(P, x) = P(x)$. We might like to design a universal program U that *doesn't* run forever even if the program it's simulating would: it would be even better to have a simulator that could figure out that P would never halt on x and just report that.

For instance, one might make a simulator U' which simulates P for only, say, n^{120} steps, and give up if P hasn't halted by then. But some programs P run for longer than n^{120} steps before halting, so it would not be true that $U'(P, x) = P(x)$ even for some programs P which halt on x.[1]

What if we instead calculated an even bigger function than n^{120}, the maximum number of steps that *any* program of length at most n runs

In fact, that better simulator doesn't exist, because there's *no* program which can figure out that an input program P would never halt on an input x. Formally:

Theorem 3.1. *There is no algorithm (modelled as a RAM program) that solves the Halting Problem.*

We'll prove this theorem next time. For today, we'll just assume it's true.

4 Unsolvability

Now we will see several more examples of unsolvable problems. But first some terminology:

Definition 4.1. Let $\Pi = (\mathcal{I}, \mathcal{O}, f)$ be a computational problem. We say that Π is *solvable* if there exists an algorithm A that solves Π. Otherwise we say that Π is *unsolvable*.

[1] Is the problem just that n^{120} isn't big enough? If we call the longest time that *any* program of size n runs on an input of length n "$BB(n)$" (the "busy beaver function"), just simulating for $BB(n)$ steps would be enough. Unfortunately, BB is an uncomputable function.

Note that we don't care about runtime of A in this definition; classifying problems by runtime was the subject of CS 120's previous topic, Computational Complexity. Almost all of the computational problems we have seen this semester (Sorting, ShortestPaths, 3-Coloring, BipartiteMatching, Satisfiability, etc.) are solvable; the Halting Problem is the only unsolvable problem we have seen so far.

Other terminology that is often used:

- If Π amounts to computing a function, i.e. $|f(x)| = 1$ for every $x \in \mathcal{I}$, then *computable function* (and *uncomputable function*) is common terminology used (instead of solvable and unsolvable).
- If Π is further restricted to a decision problem (i.e. $|f(x)| = 1$ and $f(x) \subseteq \{\texttt{yes}, \texttt{no}\} = \{\texttt{accept}, \texttt{reject}\} = \{1, 0\}$ for all $x \in \mathcal{I}$), then *decidable problem* (and *undecidable problem*) is common terminology.

Also, computability theorists sometimes require that $\mathcal{I}$ contains all possible sequences of numbers or symbols, while we allow restricting to a subset (like connected graphs or sorted arrays). When the inputs are restricted, the problem Π is often referred to as a *promise problem* (since the input is "promised" to be in $\mathcal{I}$) or *partial function* (in the case that Π amounts to computing a function).

Now that we have one unsolvable problem (the Halting Problem), we will be able to obtain more via reductions.

Recall, again, one part of lecture 3's four-part lemma about reductions:

Lemma 4.2. *Let Π and Γ be computational problems such that $\Pi \leq \Gamma$. Then:*

1. *If Π is unsolvable, then Γ is unsolvable.*

(Note that this lemma applies to all reductions, including those whose runtime is more than polynomial, unlike the reductions from the last two weeks, and ones which call the oracle more than once, unlike the mapping reductions we did last week.)

Just like we last week proved that problems were not *efficiently* solvable by reducing *from* other problems that were (suspected to be) not efficiently solvable, we can prove that problems are not solvable *at all* by reducing from other unsolvable problems: the Halting problem plays the same role for us as a font of unsolvability as SAT did for NP-hardness.

Now let's see some more unsolvable problems.

5 Other unsolvable problems via reduction

Input	**:** A RAM program P
Output	**:** `yes` if P halts on the empty input ε, `no` otherwise

Computational Problem HaltOnEmpty

Here the empty input ε is just an array of length 0. (Recall that inputs to RAM programs are arrays of natural numbers.)

Theorem 5.1. *HaltOnEmpty is unsolvable.*

Proof.

□

Our next example of an unsolvable problem is the following:

Input	**:** A RAM program P
Output	**:** `yes` if P correctly solves the graph 3-coloring problem, `no` otherwise

Computational Problem Solves3Coloring

Theorem 5.2. *Solves3Coloring is unsolvable.*

Proof.

□

There is nothing special about 3-Coloring in this proof, and a similar proof can be used to show the following very general result.

Theorem 5.3 (Rice's Theorem)**.** *Let $\Pi = (\mathcal{I}, \mathcal{O}, f)$ be a computational problem where $\mathcal{I}$ is the set of all RAM programs. Assume that:*

1. *$f(P)$ depends only on the "semantics" of P. That is, if Q is another RAM program such that for all inputs x, Q halts on x if and only if P halts on x and if so, $Q(x) = P(x)$, then $f(P) = f(Q)$.*
2. *No constant function solves Π. That is, for every $y \in \mathcal{O}$, there is some RAM program P such that $y \notin f(P)$.*

Then Π *is unsolvable.*

We won't prove Rice's Theorem, but here's how the Solves3Coloring example is a special case:

Example:

Not every unsolvable problem is about semantics of programs, so not every unsolvable problem can be proven unsolvable by Rice's theorem. We'll see some such problems next lecture, and in the active learning exercise next week.