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 §6.0–7.6
- Participation Portfolio 2 has been returned
- Check Ed for changes to OH/Section in the coming weeks, due to holiday interruptions

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 any P as input and simulates it.

Theorem 2.1.

1. ***(Universal RAM simulator in RAM)*** *There is a RAM program U such that **for every** RAM program P and input x, U halts on input (P, x) iff P halts on x and if so, $U(P, x) = P(x)$. Moreover,* $\text{Time}_U((P, x)) = O(\text{Time}_P(x) + |P| + |x|)$.
2. ***(Universal Word-RAM simulator in Word-RAM)*** *There is a Word-RAM program U such that **for every** Word-RAM program P and input x, we have $U(P, x) = P(x)$. Moreover,* $\text{Time}_U((P, x)) = O(\text{Time}_P(x) + |P| + |x|)$.

Proof idea.
Problem Set 3 RAM simulator in Python! Plus compiling Python down to RAM.

That is, in Problem Set 3, we wrote a Python program to simulate *any* RAM program: this Python program is then compiled down to assembly, which is close to Word-RAM. If we consider the compiled version of our Python script, we essentially wrote a word-RAM program to simulate any RAM program! We could similarly have compiled that program from word-RAM into RAM to get a RAM program that simulates any RAM program.

□

We can also write a Word-RAM program that simulates **any** RAM program 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 any RAM program?

We could have U convert P to a Word-RAM program and simulate it using the Word-RAM simulator above, or convert the RAM simulator U to a Word-RAM program. Either way, though, the runtime below-up in the simulation can be much larger because a RAM program P can construct numbers whose bitlength is exponential in its runtime (as seen in PS3) and simulating computations on these numbers using finite-sized words will take exponential time. So we'd only be able to show something like:

$$\text{Time}_U((P,x)) = O(\text{Time}_P(x) + |P| + |x|)$$

for RAM simulating Word-RAM, or

$$\text{Time}_U((P,x)) = 2^{O(\text{Time}_P(x)+|P|+|x|)}$$

for Word-RAM simulating RAM. (See Theorem 7.1 from lecture 7 for more precise bounds.)

Importance of Universal Programs:

- Historically: Universal Turing Machine (Turing, 1936)
- Hardware vs. Software: Can build just one computer (U) and use it to execute any program P we want. Previously: build new hardware for every new type of problem we want to solve. (The Mark I computer in the SEC was the first such computer.)
- Inspired the development of modern computers (e.g. the "von Neumann Architecture").
- Programs vs. Data: we can think of programs P as data themselves.

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]

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:

Input	**:** A RAM program P and an input x
Output	**:** `yes` (i.e. `accept` or 1) if P halts on input x, `no` (`reject` or 0) otherwise

Computational Problem Halting Problem

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.

[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.

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*.

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.
By Lemma 4.2, it suffices to show Halting reduces to HaltOnEmpty. If HaltOnEmpty were solvable, then we could also solve Halting, but since we know Halting is unsolvable, HaltOnEmpty must also be unsolvable.

That is, we need to give a reduction A that can decide whether a program P halts on an input x using an oracle that solves HaltOnEmpty.

A template for this reduction A is as follows:

```
1  A(P, x):
   Input  : A RAM program P and an input x
   Output : yes if P halts on x, no otherwise
2  Construct from P and x a RAM program Q_{P,x} such that Q_{P,x} halts on ε if and only if P
     halts on x;
3  Run the HaltsOnEmpty oracle on Q_{P,x} and return its result;
```

Algorithm 1: Template for reduction from Halting to HaltsOnEmpty

How can we construct this RAM program $Q_{P,x}$? The idea is that $Q_{P,x}$ will ignore its own input, copy x into the input locations of memory, and then run P. (Since we can think of programs as data, colloquially we can say we "hardcode" both the original input x and the program P as the program $Q_{P,x}$, so $Q_{P,x}$ ignores any new input.) More formally, if P has commands $C_0, \ldots, C_{m-1}$ and x has length n, we construct $Q_{P,x}$ as follows:

```
   Q_{P,x}(y):
0  M[0] = x[0]
1  M[1] = x[1]
                ⋮
n-1  M[n − 1] = x[n − 1]
 n   input_len = n
n+1  C'_0
n+2  C'_1
                ⋮
n+m  C'_{m−1}
```

Algorithm 2: The RAM Program $Q_{P,x}$ constructed from P and x

Here $C'_0, \ldots, C'_{m-1}$ are the lines of P, but modified so that any "GOTO ℓ" commands are replaced with "GOTO $\ell + n + 1$," since we have inserted $n + 1$ lines at the beginning.

(Here, $Q_{P,x}$ takes some input y, but nothing in the pseudocode uses y at all—this program runs the same thing regardless of the input, so we can choose whichever input we like—in this case, the empty input.) By construction, executing $Q_{P,x}(y)$ on input $y = \varepsilon$ will amount to executing P on input x. Thus we have:

Claim 5.2. *$Q_{P,x}$ halts on ε if and only if P halts on x.*

With this claim, we see that plugging the construction of $Q_{P,x}$ from Algorithm 2 into the reduction template (Algorithm 3) gives a correct reduction from Halting to HaltOnEmpty. By the unsolvability of the Halting Problem (Thm. 3.1) and Lemma 4.2, we deduce that HaltOnEmpty is unsolvable.

□

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 5.3. *Solves3Coloring is unsolvable.*

Proof.
By Theorem 5.1 and Lemma 4.2, it suffices to show that HaltOnEmpty reduces to Solves3Coloring, i.e. there is an algorithm A that solves HaltOnEmpty given an oracle for Solves3Coloring. We follow the same reduction template as before:

1 $A(P)$:
  Input : A RAM program P
  Output : `yes` if P halts on ε, `no` otherwise
2 Construct from P a RAM program Q_P such that Q_P solves 3-coloring if and only if P halts on ε;
3 Run the Solves3Coloring oracle on Q_P and return its result;

Algorithm 3: Template for reduction from HaltOnEmpty to Solves3Coloring

This time, we construct the program Q_P as follows:

1 $Q_P(G)$:
  Input : A graph G
2 Run P on ε;
3 **return** *ExhaustiveSearch3Coloring(G)*

Algorithm 4: The RAM Program Q_P constructed from P

Similarly to our previous reduction, we need to check:

Claim 5.4. *Q_P solves 3-coloring if and only if P halts on ε.*

To verify the claim, note that if P doesn't halt on ε, the $Q_P(G)$ will never halt, and thus cannot solve 3-coloring. On the other hand, if P does halt on ε, then $Q_P(G)$ will always have the same output as ExhaustiveSearch3Coloring(G) and thus correctly solves 3-coloring.

Claim 5.4 implies that plugging this construction of Q_P in to Algorithm 3 gives a correct reduction from HaltsOnEmpty to Solves3Coloring, and thus completes the proof that Solves3Coloring is unsolvable.

□

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

6 Rice's Theorem (optional)

Theorem 6.1 (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, there is some RAM program P such that $f(P) \neq \emptyset$ (so returning $\perp$ always doesn't solve Π) and for every $y \in \mathcal{O}$, there is some RAM program P such that $y \notin f(P)$ (so returning y always doesn't solve Π).*

Then Π is unsolvable.

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

Example:
Let's see that Solves3Coloring meets the conditions of Rice's Theorem. Its set $\mathcal{I}$ of inputs is indeed the set of all RAM programs. The function f is as follows:

$$f(P) = \begin{cases} \{\texttt{yes}\} & \text{if P solves 3-coloring} \\ \{\texttt{no}\} & \text{otherwise} \end{cases}$$

To verify this meets the conditions required to apply Rice's theorem, we must check two conditions. First, for Q and P that have the same behavior for all x, it must be the case that if P correctly solves 3-Coloring for all x then so does Q (since Q will produce the same answer). Analogously, if P does *not* solve 3-coloring, neither does Q. Thus f depends only on the semantics of P. Finally, to check that f is nontrivial, there is some program P that solves 3-Coloring (for instance, ExhaustiveSearch), and some program Q that does not (an infinite loop). Since $f(P) = \{\texttt{yes}\}$ and $g(P) = \{\texttt{no}\}$ are disjoint, no constant function can solve f. Since Solves3Coloring satisfies the conditions, we can apply Rice's Theorem and conclude that it is unsolvable.

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 sender–receiver exercise on 11/29.