CS1200: Intro. to Algorithms and their Limitations Anshu & Vadhan

Lecture 22: Uncomputability by Reductions

Harvard SEAS - Fall 2024 *2023-11-21*

1 Announcements

- Salil OH 11-12 pm after class in SEC 3.323, Anurag zoom OH on Friday 1:30-2:30.
- Pset 8 due Wed night.
- SRE 7 on Thursday.
- T-shirt/sticker survey on Ed.

Recommended Reading:

- MacCormick Chapter §6.0–7.6

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.

Recall the idea of a *universal simulator* for RAM programs that you saw on Problem Set 3 and was discussed briefly in the Proof Idea for Theorem 3.1 in Lecture 7: this is a program U that takes any program P as input and simulates it. The theorem below formalizes this notion.

Theorem 2.1. ***(Universal RAM simulator in RAM)*** *There is a RAM program U such that* ***for every*** *RAM program P and input x,*

Proof idea.
Problem Set 3 RAM simulator in Python! Plus compiling Python to RAM. That is,

□

Remark: We can also write a Word-RAM program that simulates **any** RAM program and vice-versa. A few aspects would change in the theorem if we wanted U to be a Word-RAM program but allow P to be any RAM program. We could convert the universal RAM simulator from Theorem 2.1 to a Word-RAM program (using Theorem 3.1 from Lecture 8). However, the runtime blow-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 PSet 3) and simulating computations on these

numbers using finite-sized words will take exponential time. So we'd only be able to show something like:

$$T_U((P,x)) = 2^{O(T_P(x)+|P|+|x|)}$$

for Word-RAM simulating RAM. (See Theorem 3.1 from lecture 8 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 near the main entrance of the SEC was one of the first such computers.)
- 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 $U'(P,x) = P(x)$.[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 `no` (`reject` or 0)

Computational Problem The Halting Problem for RAM Programs

Theorem 3.1. *There is no algorithm that solves the Halting Problem for RAM Programs.*

We'll prove this theorem in Lecture 23. For today, we'll just assume it's true. A similar result can be shown for the Word-RAM analogue of the above problem:

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

Computational Problem HaltingProblem-WordRAM

[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.

Theorem 3.2. *There is no algorithm that solves the Halting Problem for Word RAM programs.*

Theorem 3.2 can be deduced from Theorem 3.1, as we will see in the next section for a variant of the Halting Problem (namely HaltsOnEmpty).

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 RAM program P that solves Π. Otherwise we say that Π is *unsolvable*.

Note that we don't care about runtime of P in this definition; classifying problems by runtime was the subject of CS 1200'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-RAM is the first unsolvable problem we have seen.

Now that we have one unsolvable problem (the Halting Problem-RAM), we will be able to obtain more via reductions. For this, we recall Lecture 4's four-part lemma about reductions. We will use the following part

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

-

We highlight that this lemma applies to all reductions, including those whose runtime is more than polynomial, unlike the polynomial time reductions from the last two weeks.

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

5 Other unsolvable problems via reduction

The following problem is a special case of the Halting Problem-RAM problem.

Input	**:** A RAM program Q
Output	**:** `yes` if
	`no` otherwise

Computational Problem HaltsOnEmpty-RAM

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. *HaltsOnEmpty-RAM is unsolvable.*

Proof.
By Lemma 4.2, it suffices to show that the Halting Problem-RAM reduces to HaltsOnEmpty-RAM.

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
3 Call
```

Algorithm 1: Template for reduction from the Halting Problem-RAM to HaltsOnEmpty-RAM

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,

We show the following claim:

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

Proof.

□

Now, we see that plugging the construction of $Q_{P,x}$ from into the reduction template (Algorithm 3) gives a correct reduction from the Halting Problem-RAM to HaltsOnEmpty-RAM. That is,

By the unsolvability of the Halting Problem-RAM Problem (Thm. 3.1) and Lemma 4.2, we

deduce that HaltsOnEmpty-RAM is unsolvable.

□

Using Theorem 3.1 from Lecture 8, we can also conclude the unsolvability of the Word-RAM variant of the problem:

Input	: A Word-RAM program R
Output	: `yes` if
	`no` otherwise

Computational Problem HaltsOnEmpty-WordRAM

Theorem 5.3. *HaltsOnEmpty-WordRAM is unsolvable.*

Proof idea. We give a reduction from HaltsonEmpty-RAM to HaltsOnEmpty-WordRAM, as in the following template.

□

By similar reasoning (using the fact that the simulations of RAMs or Word-RAMs by TMs are given by compilers that can be implemented as algorithms), HaltOnEmpty-TM and the Halting Problem-TM are also unsolvable, and similarly for other Turing-equivalent models of computation.

Our next example of an unsolvable problem is the following:

Input	: A RAM program S
Output	: `yes` if
	`no` otherwise

Computational Problem Solves3Coloring-RAM

Theorem 5.4. *Solves3Coloring-RAM is unsolvable.*

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

```
1 A(Q):
  Input  : A RAM program Q
  Output : yes if Q halts on ε, no otherwise
2 Construct from Q
3 Call
```

Algorithm 2: Template for reduction from HaltsOnEmpty-RAM to Solves3Coloring-RAM

This time, we construct the program S_Q as follows:

Similarly to our previous reduction, we need to check:

Claim 5.5. *S_Q solves 3-coloring if and only if Q halts on ε.*

Proof.

□

Claim 5.5 implies that plugging this construction of S_Q in to Algorithm 3 gives a correct reduction from HaltsOnEmpty-RAM to Solves3Coloring-RAM, and thus completes the proof that Solves3Coloring-RAM is unsolvable.

□

We note again analogous results hold for RAMs, TMs, and all Turing-equivalent models, i.e. Solves3Coloring-WordRAM and Solves3Coloring-TM are unsolvable.

There is nothing special about 3-Coloring in this proof, and a similar proof can be used to show a very general result called the Rice's theorem (stated in the next optional section). Rice's theorem says that every non-trivial problem about the input-output behavior (semantics) of programs is unsolvable. Halting, Halting on empty input and solving 3-coloring are examples of non-trivial semantic properties. Non-semantic properties include having even number of lines of codes, running in linear time etc. There are unsolvable problems not covered by Rice's theorem; we will examples in the SRE on Thursday (about running in linear time), in Lecture 23 and also one in problem set 9.

6 Rice's Theorem (optional)