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, U halts on input (P, x) iff P halts on x and if so, $U(P, x) = P(x)$. Moreover, $T_U((P, x)) = O(T_P(x) + |P| + |x|)$.*

Proof idea.
Problem Set 3 RAM simulator in Python! Plus compiling Python to RAM. That is, in Problem Set 3, we wrote a Python program to simulate *any* RAM program: this Python program can be compiled (using the first item of Theorem 3.1 in Lecture 7) to get a RAM program U that simulates any RAM program. □

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: The idea of a universal program was first proposed in Turing's 1936 paper, where he introduced Turing Machines and proved, by construction, the existence of a Universal Turing Machine. This had an enormous impact on practical computing. Previously, people thought that one had to build new computing hardware for each type of problem we want to solve. With universal machines, we can build just one piece of hardware (namely, one that implements U) and use it to execute any program P that we want as software. The Mark I Computer near the entrance of the SEC was one of the first physical instantiations of such a universal computer. The "von Neumann architecture" that is used in modern computers also follows the Universal Turing Machine in using the same memory to store the instructions of the software program being executed (the program P) and the data on which it is being executed (namely, the input x as well as the working memory of the program P). Thus programs can be treated as data, and we can consider solving computational problems where the inputs are programs (as we will today, and is done in practice by compilers, debuggers, etc.).

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 P halts on input x
	`no` (`reject` or 0) otherwise

Computational Problem The Halting Problem for RAM Programs

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

The theorem holds for algorithms in any model of computation - not just the RAM model - due to the Church-Turing thesis (Lecture 17).

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:

[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.

Input	**:** A Word-RAM program P and an input x
Output	**:** `yes` (i.e. `accept` or 1) if there is a word length w such that $P[w]$ halts on input x, `no` (`reject` or 0) otherwise

Computational Problem HaltingProblem-WordRAM

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.

Remarks:

- In literature, you may see other teminologies associated to solvability:
  - 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.
- 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-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:*

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

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 Q halts on the empty input ε `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. If HaltsOnEmpty-RAM were solvable, then we could also solve the Halting Problem-RAM, but since we know the Halting Problem-RAM is unsolvable, HaltsOnEmpty-RAM 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 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 $Q_{P,x}$ halts on $\varepsilon$ if and only if $P$
    halts on $x$;
3 Call the HaltsOnEmpty-RAM oracle on $Q_{P,x}$ and return its result ;
```

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, if P has commands $C_0, \ldots, C_{m-1}$ and x has length n, we construct $Q_{P,x}$ as follows:

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

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

We highlight the following aspects of this algorithm:

- $C'_0, \ldots, C'_{m-1}$ are the lines of P, but modified so that any GOTO commands are adjusted since we have inserted $n+1$ lines at the beginning.
- $Q_{P,x}$ 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.

We show the following claim:

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

Proof. By construction, executing $Q_{P,x}$ on input ε will amount to executing P on input x. Thus, the former will halt iff the latter will halt. □

Now, we see that plugging the construction of $Q_{P,x}$ from Algorithm 2 into the reduction template (Algorithm 3) gives a correct reduction from the Halting Problem-RAM to HaltsOnEmpty-RAM. That is, if the algorithm A outputs `yes`, this means that the HaltsonEmpty-RAM oracle gives the result `yes` in Line 3. For this, $Q_{P,x}(\varepsilon)$ must halt, which implies that $P(x)$ halts (by Claim 5.2). On the other hand, if $P(x)$ halts, then by Claim 5.2 $Q_{P,x}(\varepsilon)$ halts and Line 3 returns `yes`.

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 there is a word length w such that $R[w]$ halts without crashing on the empty input ε, `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.

```
1 A(Q):
  Input    : A RAM program $Q$
  Output   : yes if $Q$ halts on $\varepsilon$, no otherwise
2 Construct from $Q$ a Word-RAM program $R_Q$ such that $R_Q[w]$ halts without crashing on $\varepsilon$
    for some $w$ if and only if $Q$ halts on $\varepsilon$;
3 Call the HaltsOnEmpty-WordRAM oracle on $R_Q$ and return its result ;
```

Algorithm 3: Template for reduction from HaltsonEmpty-RAM to HaltsOnEmpty-WordRAM

The Word-RAM program R_Q is constructed based on the RAM-to-WordRAM Simulation Theorem (Thm 3.1 in Lecture 8). That theorem was stated existentially: for every RAM program Q, there *exists* is a Word-RAM program R_Q that simulates Q; in particular, if Q halts on ε, then $R_Q[w]$ halts without crashing on ε for all sufficiently large w, and if Q doesn't halt on ε, then $R_Q[w]$ halts or crashes for all w. Now we observe that this proof was actually constructive, and can be turned into an *algorithm* that constructs R_Q from Q — that is, a RAM-to-WordRAM *compiler*. This compiler instantiates Line 2 and completes the reduction.

□

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 $S$ solves the graph 3-coloring problem,
           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 $\varepsilon$, no otherwise
2 Construct from $Q$ a RAM program $S_Q$ such that $S_Q$ solves 3-coloring if and only if $Q$ halts
    on $\varepsilon$;
3 Call the Solves3Coloring-RAM oracle on $S_Q$ and return its result ;
```

Algorithm 4: Template for reduction from HaltsOnEmpty-RAM to Solves3Coloring-RAM

This time, we construct the program S_Q as follows:

```
1 S_Q(G):
  Input      : A graph G
2 Run Q on ε;
3 return ExhaustiveSearch3Coloring(G)
```

Algorithm 5: The RAM Program S_Q constructed from Q

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. If Q doesn't halt on ε, the $S_Q(G)$ will never halt, and thus cannot solve 3-coloring. On the other hand, if Q does halt on ε, then $S_Q(G)$ will always have the same output as ExhaustiveSearch3Coloring(G) and thus correctly solves 3-coloring. □

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)

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.