CS120: Intro. to Algorithms and their Limitations Prof. Salil Vadhan

Lecture 20: Unsolvable Problems

Harvard SEAS - Fall 2021 *Nov. 11, 2021*

1 Announcements

Recommended Reading:

- MacCormick Chapter 7
- No office hours for Salil next week
- Active learning Tuesday

2 The Approach

Last time we saw our first example of an *unsolvable problem*, for which there is no algorithm whatsoever:

Input	**:** A RAM program P and an input x
Output	**:** `accept` if P halts on input x, `reject` otherwise

Computational Problem Halting Problem

Theorem 2.1. *There is no algorithm (modelled as a RAM program) that solves the Halting Problem.*

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

Definition 2.2. Let $\Pi = (\mathcal{I}, f)$ 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 is the subject of *Computational Complexity*, our next topic. 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 $f(x) \subseteq \{\texttt{accept}, \texttt{reject}\} = \{Y, N\} = \{1, 0\}$ for all $x \in \mathcal{I}$), then *decidable problem* (and *undecidable problem*) is common terminology.

Also, people often 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.

Definition 2.3. Let $\Pi = (\mathcal{I}, f)$ and $\Gamma = (\mathcal{J}, g)$ be two computational problems. A *reduction* from Π to Γ is an algorithm that solves Π using a(ny) oracle that solves Γ as a subroutine. If there exists a reduction from Π to Γ, then we write $\Pi \leq \Gamma$.

Note that if Γ is a computational problem where there can be multiple valid solutions (i.e. $|g(x)| > 1$ for some $x \in \mathcal{J}$), then a valid reduction is required to work correctly for *every* oracle that solves Γ (i.e. no matter which valid solutions it returns).

The use of reductions to prove unsolvability comes from the following lemma:

Lemma 2.4. *Let Π and Γ be computational problems such that $\Pi \leq \Gamma$. Then:*

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

Proof.

1. Let A be the algorithm solving Π using an oracle to Γ. By assumption, there exists an algorithm solving Γ, denoted B. We then use B in the place of A's oracle and obtain a standard (oracle-free) algorithm for Π.
2. This is the contrapositive of Item 1.

□

Both of these statements are not true if we flip the $\leq$. Intuitively, this is because very easy problems can reduce to very hard problems (consider a sorting algorithm that first calls a Halting Problem oracle on the array, then discards the result). But this doesn't imply that sorting is unsolvable.

So far in the course, we have been using reductions to show (efficient) solvability of problems, i.e. using Item 1. Now we will use Item 2 to prove that problems are unsolvable. *Note that the direction of the reduction ($\Pi \leq \Gamma$ vs. $\Gamma \leq \Pi$) is crucial!*

We have already seen one example of using Lemma 2.4. Our proof of the undecidability of the Halting Problem last time can be broken into two parts:

1. The RejectSelf problem reduces to the Halting Problem.
2. The RejectSelf problem is unsolvable.

With Lemma 2.4, these two claims imply that the Halting Problem is unsolvable.

3 Unsolvable Problems

Now let's see some more unsolvable problems.

Input	**:** A RAM program P
Output	**:** `accept` if P halts on the empty input ε, `reject` 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 3.1. *HaltOnEmpty is unsolvable.*

Proof.
By Lemma 2, it suffices to show Halting reduces to HaltOnEmpty. 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 : accept if P halts on x, reject 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. 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.

By construction, executing $Q_{P,x}(y)$ on input $y = \varepsilon$ will amount to executing P on input x. Thus we have:

Claim 3.2. *$Q_{P,x}$ halts on ε if and only if P halts on ε.*

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. 2.1) and Lemma 2.4, we deduce that HaltOnEmpty is unsolvable.

□

Our next example of an unsolvable problem is the following:

Input : A RAM program P
Output : `accept` if P correctly solves the graph 3-coloring problem, `reject` otherwise

Computational Problem Solves3Coloring

Theorem 3.3. *Solves3Coloring is unsolvable.*

Proof.
By Theorem 3.1 and Lemma 2.4, 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 : `accept` if P halts on ε, `reject` 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 can need to check:

Claim 3.4. *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 3.4 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.

Theorem 3.5 (Rice's Theorem)**.** *Let $\Pi = (\mathcal{I}, f)$ 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 programs such that $Q(x) = P(x)$ for all inputs x, then $f(P) = f(Q)$.*
2. *No constant function solves Π. That is, for every possibly output y, 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:
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:

$$f(P) = \begin{cases} \{\texttt{accept}\} & if \text{P} solves 3-coloring \\ \{\texttt{reject}\} & otherwise \end{cases}$$

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 $f(P) = \{\texttt{accept}\}$ and $g(P) = \{\texttt{reject}\}$ 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.
In the next active learning exercise, you will see a problem for which Rice's Theorem doesn't apply, i.e. an unsolvable problem about programs that isn't only about the program's functional semantics.