Section 11: P vs. NP, Unsolvability, and Reductions

1 The RAM model and computable functions

The RAM model is a particular model of computation, where we have a program P specified by a finite number of commands $C_0, \ldots, C_t$, each of which is a single basic operation (like reading to or writing from the RAM), and the runtime of a program on input x is the number of commants executed by P when given x as input. For RAM programs, we give input by setting the first n locations in memory equal to x before we start execution.

An important concept for subsequent parts of the course is the **universal program**. Here, we mean there is a program U that, given (P, x) as input, simulates P on input x. If $P(x)$ returns y, then $U(P, x) = y$, and if $P(x)$ never halts, neither does $U(P, x)$.

Even better, this simulation is time efficient, in that $\text{Time}(U(P, x)) = O(\text{Time}(P(x))$ (though we actually won't need this property for the computability theory part of the course).

There is a similar simulator program for Word-RAM. We can use these simulator programs to *answer questions about programs.* For instance, consider the computational problem:

Definition 1.1 (CorrectlySorts$x$Within$n\log(n)$Time)**.** Fixing a list of integers x of length n, the set of inputs $\mathcal{I}$ for this decision problem $\Pi_x = (\mathcal{I}, \{\texttt{yes}, \texttt{no}\}, f_x)$ is the set of all RAM programs. The function f_x satisfies $f_x(P) = \{\texttt{yes}\}$ if and only if P outputs a correctly sorted list on input x within $n\log(n)$ timesteps.

We do **not** allow P to run for $O(n\log n)$ steps, only the exact bound $n\log n$. Our goal is to create a program Q that correctly solves this problem.

One way we could try to do this is by looking at the code of P and trying to prove theorems about it (a la a theory problem), but theory problems can often be harder than programming ones. For a potentially easier solution, we can "check" if P correctly sorts x within $n\log(n)$ steps. If it does so, we know P is good. We can run this simulation on pencil-and-paper, and with the existence of a simulator program, we can also do it inside RAM (or Word-RAM).

Question 1.2. Give pseudocode (or describe) a program Q_x that solves CorrectlySorts$x$Within$n\log(n)$Time. Note that Q_x will be given the sorting program P as input (and we know x in advance).

Question 1.3. Write a program Q that solves CorrectlySortsWithin$n\log(n)$Time. Here, the set $\mathcal{I}$ is all pairs (P, x) where P is a RAM program and x is an integer array. The function we wish to compute, f, satisfies $f(P, x) = 1$ if P sorts x in time $n\log(n)$ where $n = |x|$.

However, a seemingly slight generalization of this problem is much harder! If we instead ask if P sorts *any input* of length n in $O(n\log(n))$ time, this problem is actually unsolvable.

2 Unsolvability

What do we mean when we say that a problem is unsolvable?

Definition 2.1. 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.*

Almost all of the computational problems we have encountered this semester (Sorting, 3-Coloring, BipartiteMatching, Satisfiability, etc.) are solvable. We know this because we have studied and derived algorithms (some more efficient than others) that can solve those problems. Let's now look at a couple of examples of unsolvable problems.

3 The Halting Problem

The halting problem is a decision problem that takes in (a) a RAM program P and (b) an input x to P. It determines whether P will halt or run forever if it was given x as input.

Input	**:** A RAM program P and an input x
Output	**:** `yes` if P halts on input x, `no` otherwise

Computational Problem Halting Problem

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

In other words, we say that the Halting Problem is unsolvable.

A similar unsolvable decision problem is HaltOnEmpty:

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

Computational Problem HaltOnEmpty

4 Proving Unsolvability

How do we prove that the Halting Problem, HaltOnEmpty, or another problem is unsolvable? Through reductions! The use of reductions to prove unsolvability comes from the following lemma:

Lemma 4.1. *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.*

The basic structure of a reduction for proving a computational problem Γ is unsolvable is as follows:

1. Find a known computational problem Π that is unsolvable (such as Halting or HaltOnEmpty)
2. Construct a reduction A_Π from Π to Γ:

$A_\Pi(x)$:
  y = R(x) ; `/* Create a new input` y `for which` $O_\Gamma(y) = A_\Pi(x)$ `*/`
  Return $O_\Gamma(y)$;

Algorithm 1: Basic Reduction

$A_\Pi(x)$:
  $\vdots$; `/* Pre-processsing` $x \to y$ `*/`
  $z = O_\Gamma(y)$; `/* Where` O_Γ `is the oracle */`
  $\vdots$; `/* Post-processsing` $z \to$ `output */`
  Return output;

Algorithm 2: General Reduction

3. Γ is unsolvable.

Note that in a reduction from Π to Γ ($\Pi \leq \Gamma$), 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).

Question 4.2. Consider the problems A and B. Are these statements true or false?

1. If A reduces to B and B is solvable, then A is always solvable.

2. If A reduces to B and B is unsolvable, then A is always unsolvable.
3. If A reduces to B and A is solvable, then B must also be solvable.
4. If A reduces to B and A is unsolvable, then B must also be unsolvable.
5. If B is unsolvable and A is solvable, B will never reduce to A.

Question 4.3. Show that the problem ReturnsOne, which determines whether a program with a given input ever returns the digit 1, is unsolvable.

Question 4.4. Show that the following problem is solvable. Given two programs with their inputs and the knowledge that exactly one of them halts, determine which halts.

5 Search vs. Decision

In Lecture 21, we briefly covered search vs decision problems. To summarize the following definitions: *decision problems* address the existence of a solution, returning either $\{\texttt{yes}, \texttt{no}\}$. On the otherhand, *search problems* finds the solution if one exists.

5.1 Definitions

Definition 5.1. A computational problem $\Pi = (\mathcal{I}, \mathcal{O}, f)$ is a *decision problem* if $\mathcal{O} = \{\texttt{yes}, \texttt{no}\}$ and for every $x \in \mathcal{I}$, $|f(x)| = 1$.

NP consists of the problems that amount to deciding whether an instance of an $\mathsf{NP}_{\mathsf{search}}$ problem has a solution or not. Formally:

Definition 5.2 (NP)**.** A decision problem $\Pi = (\mathcal{I}, \{\texttt{yes}, \texttt{no}\}, f)$ is in NP if there is a computational problem $\Gamma = (\mathcal{I}, \mathcal{O}, g) \in \mathsf{NP}_{\mathsf{search}}$ such that for all $x \in \mathcal{I}$, we have:

$$\begin{aligned} f(x) = \{\texttt{yes}\} \quad &\Leftrightarrow \quad g(x) \neq \emptyset \\ f(x) = \{\texttt{no}\} \quad &\Leftrightarrow \quad g(x) = \emptyset \end{aligned}$$

Another view of NP: decision problems Π where a $\texttt{yes}$ answer has a short, efficiently verifiable proof. Indeed, we can prove that $f(x) = \{\texttt{yes}\}$ by giving a solution $y \in g(x)$, which is of at most polynomial length and is verifiable in polynomial time.

5.2 Examples

SAT:

- **Decision problem:** Given a formula φ, decide if φ is satisfiable
- **Search problem:** Given a formula φ, find a satisfying assignment $(\alpha_1, \ldots, \alpha_n)$ if one exists.

K-Independent Set:

- **Decision problem:** Given a graph G, decide if there exists an independent set of at least k
- **Search problem:** Given a graph G, find an independent set of at least k if one exists

5.3 Reductions

Lemma 5.3. *Satisfiability $\leq_p$ Satisfiability-Decision.*

Proof sketch. The idea is to find a satisfying assignment one variable at a time, using the Satisfiability-Decision oracle to determine whether setting $x_i = 0$ or $x_i = 1$ preserves satisfiability.

$R(\varphi)$:
Input : A CNF formula $\varphi(x_0, \ldots, x_{n-1})$ (and access to an oracle O solving Satisfiability-Decision)
Output : A satisfying assignment α to φ, or $\perp$ if none exists.
if $O(\varphi) = \textit{no}$ **then return** $\perp$;
foreach $i = 0, \ldots, n-1$ **do**
    if $O(\varphi(\alpha_0, \ldots, \alpha_{i-1}, 0, x_{i+1}, \ldots, x_{n-1})) = \textit{yes}$ **then** $\alpha_i = 0$;
    else $\alpha_i = 1$;
return $\alpha = (\alpha_0, \ldots, \alpha_{n-1})$

$\square$

Question 5.4. Let IS be the computational problem for finding an independent set and IS_{dec} be the computational problem for determining the existence of a independent set. Prove that $IS \leq_p IS_{dec}$

6 P vs. NP

We know the following to be true:

Lemma 6.1. $\mathsf{P} \subseteq \mathsf{NP}$.

(A proof sketch for this lemma can be found in the Lecture 19 notes.)

But one of the central open questions in mathematics and computer science is the P vs. NP question: whether or not $\mathsf{P} = \mathsf{NP}$. Or, equivalently, whether $\mathsf{NP}_{\mathsf{search}} \subseteq \mathsf{P}_{\mathsf{search}}$. The answer is widely conjectured to be no, but we do not have formal proof.

If $\mathsf{P} = \mathsf{NP}$, then:

- Searching for solutions is never much harder than verifying solutions.

- Optimization is easy.
- Finding mathematical proofs is easy.
- Breaking cryptography is easy.
- Machine learning is easy.
- Every problem in NP is NP-complete (ps9).

If $\mathsf{P} \neq \mathsf{NP}$, then:

- None of the NP-complete problems have (worst-case) polynomial-time algorithms.
- There are problems in NP that are neither NP-hard nor in P, and similarly for search problems.
- There is *hope* for secure cryptography (but this seems to require assumptions stronger than $\mathsf{P} \neq \mathsf{NP}$).

7 Diophantine Equations

The solvability of Diophantine Equations is defined as below:

Input	: A multivariate polynomial $p(x_0, x_1, \ldots, x_{n-1})$ with integer coefficients
Output	: `yes` if there are natural numbers $\alpha_0, \alpha_1, \ldots, \alpha_{n-1}$ such that $p(\alpha_0, \alpha_1, \ldots, \alpha_{n-1}) = 0$, `no` otherwise

Computational Problem Diophantine Equations

While this problem is unsolvable in the general case, what happens if restrict the search space to a finite set?

Question 7.1. Consider the problem SolutionsInK, where the program takes in a multivariate polynomial with integer coefficients, and will accept if there are integers $\alpha_0, \ldots \alpha_{n-1}$ where $\alpha_i \in [1, K]$ that solves the polynomial. Is the problem solvable? If so, how do we construct the program?

Input	**:** A multivariate polynomial $p(x_0, x_1, \ldots, x_{n-1})$ with integer coefficients
Output	**:** `yes` if there are **non-zero integers** $\alpha_0, \alpha_1, \ldots, \alpha_{n-1}$ such that $p(\alpha_0, \alpha_1, \ldots, \alpha_{n-1}) = 0$, `no` otherwise

Computational Problem Integer Diophantine Equations

While we restrict our space of solutions in the definition of Diophantine Equations to natural numbers ($\{1, 2, \ldots\}$), there are other versions of the problem in which the variables can be assigned *integer* values ($\{\ldots -2, -1\} \cup \{1, 2, \ldots\}$)[1]. Is this version of the problem significantly harder?

Question 7.2. (1) Show a reduction from Integer Diophantine Equations to Diophantine Equations. (2) If we know that Diophantine Equations is unsolvable, is this reduction enough to prove that Integer Diophantine Equations is unsolvable?

[1]We ignore 0 because often there are no constants in Diophantine Equations and so 0 is considered to be part of a 'trivial solution'.