CS120: Intro. to Algorithms and their Limitations Hesterberg & Vadhan

Section 9: Church-Turing & Computational Complexity

Harvard SEAS - Fall 2022 *Nov. 2, 2022*

1 Introduction

This section has three main parts: the `ExtractAssignment` algorithm from the topic on resolution, the Church-Turing thesis, and computational complexity.

2 Satisfiability Review

Below, we define Satisfiability and CNF-Satisfiability:

Input	: A boolean formula φ on n variables
Output	: An $\alpha \in \{0,1\}^n$ such that $\varphi(\alpha) = 1$, or $\perp$ if no satisfying assignment exists

Computational Problem Satisfiability

Input	: A CNF formula φ on n variables
Output	: An $\alpha \in \{0,1\}^n$ such that $\varphi(\alpha) = 1$, or $\perp$ if no satisfying assignment exists

Computational Problem CNF-Satisfiability

2.1 Simplification

We introduce the following notation:

Definition 2.1. For a (simplified) clause C, a variable v, and an assignment $a \in \{0,1\}$, we write $C|_{v=a}$ to be the simplification of clause C with v set to a. That is,

1. if neither v nor $\neg v$ appears in C, then $C|_{v=a} = C$,

2. if v appears in C and $a = 0$, $C|_{v=a}$ equals C with v removed,

3. if $\neg v$ appears in C and $a = 1$, $C|_{v=a}$ equals C with $\neg v$ removed,

4. if v appears in C and $a = 1$ or if $\neg v$ appears in C and $a = 0$, $C|_{v=a} = 1$.

(We do not need to address the case that both v and $\neg v$ appear in C, since we assume that all clauses are simplified.)

2.2 Extract Assignment

Definition 2.2. For a set $\mathcal{C}$ of clauses, a variable v, and an assignment $a \in \{0,1\}$, we write

$$\mathcal{C}|_{v=a} = \{C|_{v=a} : C \in \mathcal{C}\}.$$

Observe that the satisfying assignments of $\mathcal{C}|_{v=a}$ are exactly the satisfying assignments of $\mathcal{C}$ in which v is assigned a.

Here is pseudocode for assignment extraction algorithm:

```
ExtractAssignment($\mathcal{C}$)
Input  : A closed and simplified set $\mathcal{C}$ of clauses over variables $x_0, \ldots, x_{n-1}$ such that
         $0 \notin \mathcal{C}$
Output : An assignment $\alpha \in \{0,1\}^n$ that satisfies all of the clauses in $\mathcal{C}$
foreach $i = 0, \ldots, n-1$ do
    if $(x_i) \in \mathcal{C}$ then $\alpha_i = 1$;
    else $\alpha_i = 0$;
    $\mathcal{C} = \mathcal{C}|_{x_i = \alpha_i}$;
return $\alpha$
```

Concept Check: Let φ be a Boolean formula with set of clauses $\mathcal{C}$. Should you apply the `ExtractAssignment` direcly to $\mathcal{C}$?

Question 2.3. Apply `ExtractAssignment` to the following set of clauses $\mathcal{C}$:

$$\mathcal{C} = \{(x_3 \vee x_2 \vee \neg x_1) \wedge (x_2 \vee \neg x_4) \wedge (\neg x_3 \vee x_1 \vee \neg x_2) \wedge (\neg x_3) \wedge (x_3 \vee x_4)\}$$

3 The Church-Turing Thesis

3.1 The Standard Church–Turing Thesis

We have seen that RAM programs cannot solve many computational problems about other RAM programs. To what extent does this apply to other models of computation?

Theorem 3.1 (Turing-equivalent models)**.** *If a computational problem Π is solvable in one of the following models of computation, then it is solvable in all of them:*

- *RAM programs*

- *Word-RAM programs*
- *XOR-extended RAM or Word-RAM programs*
- *%-extended RAM or Word-RAM programs*
- *Python programs*
- *OCaml programs*
- *C programs (modified to allow a variable/growing pointer size)*
- *Turing machines*
- *Lambda calculus*
- $\vdots$

Given Theorem 3.1, we can replace both the RAM program being analyzed and the RAM program carrying out the analysis with any of the equivalent models of computation.

Concept Check: Suppose that some computational problem is unsolvable by RAM programs. Does it follow that it is also unsolvable by Python programs?

Concept Check: The Turing equivalence theorem is reminiscent about a theorem that we had already seen in class. What is it?

3.2 The Strong (or Extended) Church–Turing Thesis

The Church–Turing hypothesis only concerns problems solvable at all by these models of computation (Word-RAM programs, etc.). We haven't even seen any problems that are *not* solvable by Word-RAM programs—that will be a topic for the end of the course. There is, however, a stronger version of the Church–Turing hypothesis that also covers the efficiency with which we can solve problems.

More formally,

> Extended Church–Turing Thesis v2: Every physically realizable, deterministic, and sequential model of computation can be simulated by a Word-RAM program (or Turing Machine, or Python program) with only a polynomial slowdown in runtime. Conversely, reality can simulate Word-RAM programs in real time only polynomially slower than their defined runtime.

"Deterministic" rules out both randomized and quantum computation, as both are inherently probabilistic. "Sequential" rules out parallel computation.

4 Computational Complexity

In the beginning of CS 120, we were focused on proving that certain problems are *easy*. To do so, we showed the existence of an algorithm that met the desired runtime bound. For problems like Interval Scheduling and 2-Coloring, we described algorithms, proved their correctness, and gave their runtime.

Now, we've introduced a new type of objective: proving that certain problems are *hard*. Proving that a problem is hard requires a different strategy than proving that a problem is easy. For one, it's usually difficult to enumerate over the entire space of solutions and definitely prove a lower bound on runtime. Instead, we aim to show that a problem in question is *just as hard* as some reference problem that we know to be "hard." To accomplish this, we *reduce from* the known hard problem to the problem in question: we show that the problem in question can be used to solve the known hard problem with at most polynomial overhead.

In this section, we will refer to KnownHardProblem as the reference problem and ProblemInQuestion as the problem of interest. When reducing problems to others, you will treat both KnownHardProblem and ProblemInQuestion as *black boxes*: you assume that they will behave correctly, but you have no control over their implementation. What you do have control over is the *input* to the ProblemInQuestion. Given an input to KnownHardProblem, you'll work backwards to design a perfect input to ProblemInQuestion so that its output corresponds to the desired output for KnownHardProblem.

Concept Check: The direction of a reduction is crucial!

- Suppose problem X is known to be easy. If you want to show that problem Y is also easy, should you try to reduce X to Y or Y to X?
- Suppose problem X is known to be hard. If you want to show that problem Y is also hard, should you try to reduce X to Y or Y to X?

4.1 $\mathsf{P}_{\mathsf{search}}$ and $\mathsf{EXP}_{\mathsf{search}}$

Definition 4.1. Complexity classes.

- For a function $T : \mathbb{N} \to \mathbb{R}^+$, $\mathsf{TIME}_{\mathsf{search}}(T(N))$ is: the class of computational problems $\Pi = (\mathcal{I}, f)$ such that there is a Word-RAM program solving Π in time $O(T(N))$ on inputs of bit-length N. $\mathsf{TIME}(T(N))$ is the class of *decision* (i.e. yes/no) problems in $\mathsf{TIME}_{\mathsf{search}}(T(N))$.
- (Polynomial time)

$$\mathsf{P}_{\mathsf{search}} = \bigcup_c \mathsf{TIME}_{\mathsf{search}}(n^c), \qquad \mathsf{P} = \bigcup_c \mathsf{TIME}(n^c)$$

- (Exponential time)

$$\mathsf{EXP}_{\mathsf{search}} = \bigcup_c \mathsf{TIME}_{\mathsf{search}}\left(2^{n^c}\right), \qquad \mathsf{EXP} = \bigcup_c \mathsf{TIME}\left(2^{n^c}\right).$$

Concept Check: When informally we say "easy" and "hard", which is referring to $\mathsf{P}_{\mathsf{search}}$ and which to $\mathsf{EXP}_{\mathsf{search}}$?
Additionally, we covered the following:

Theorem 4.2.
$\mathsf{P}_{\mathsf{search}} \subsetneq \mathsf{EXP}_{\mathsf{search}}$, *and* $\mathsf{P} \subsetneq \mathsf{EXP}$.

4.2 Proving that a problem is in $\mathsf{P}_{\mathsf{search}}$

To prove that a problem is in $\mathsf{P}_{\mathsf{search}}$, follow the same strategy that you've used to prove that other problems are "easy."

1. Give an algorithm.
2. Prove its correctness.
3. Show that the algorithm runs in polynomial time.

Question 4.3. Consider the IndependentSet problem we've seen in lecture.

Input	**:** A graph G and a number $k \in \mathbb{N}$
Output	**:** An independent set S of size k in G, or $\perp$ if no such independent set exists

Computational Problem IndependentSet

We can show that IndependentSet is in $\mathsf{P}_{\mathsf{search}}$ when k is restricted to some constant. For example, the 3-IndependentSet problem searches for an independent set of size 3 in a graph G. Prove that 3-IndependentSet is in $\mathsf{P}_{\mathsf{search}}$.

Concept Check: Is DNF-SAT in $\mathsf{P}_{\mathsf{search}}$? What about CNF-SAT?

Reduction tip: It's helpful to think of yourself as working backwards: you're designing an input that can be fed into a black box and assuming that it will do its job correctly. You're not worried about how the black box would actually work.

4.3 Reductions

To expand on our definition of reductions from last week, we cover 3 main concepts regarding reductions here:

1. Polynomial-time Reductions
2. Closure of $\mathsf{P}_{\mathsf{search}}$ under reductions
3. Transitivity of Reductions

Definition 4.4. For computational problems Π and Γ, we write $\Pi \leq_p \Gamma$ if there is a *polynomial-time* reduction R from Π to Γ. That is, there should exist $c \in \mathbb{R}$ such that on an input of length N, the reduction runs in time $O(N^c)$, if we count the oracle calls as one time step (as usual).

Some examples of polynomial time reduction that we've seen include:

- GraphColoring $\leq_p$ SAT
- LongPath $\leq_p$ SAT

Remember our definition of a **reduction**: If Problem A *reduces to* Problem B, we can solve Problem A using an oracle for Problem B. Intuitively, you can think of this as saying that Problem A is *no harder than* Problem B. Now, we're going to apply those same principles to reduce to satisfiability problems. Typically, a reduction with one oracle call goes like this:

Algorithm for Problem A by reducing to Problem B

- Pre-process the input: $R_{pre}(x)$.
- Call the oracle, which solves Problem B on the transformed input: $B(R_{pre}(x))$.
- Post-process the output: $R_{post}(B(R_{pre}(x)))$. Typically $R_{post}(y)$ outputs $\perp$ iff $y = \perp$; we will assume this in the correctness proof below.

After specifying the reduction, you will often be asked to prove the correctness of the reduction. In other words, you need to show that a solution to Problem B on the transformed input can be used to construct a solution to Problem A on the original input.

A typical proof of correctness of a reduction from $A = (\mathcal{I}, f)$ to $B = (\mathcal{J}, g)$

- $x \in \mathcal{I}$ and $A(x) = \emptyset$ implies that $g(R_{pre}(x)) = \emptyset$. This implies that we correctly output $\perp$ when there is no solution to Problem A.
- $x \in \mathcal{I}$ and $A(x) \neq \emptyset$ implies that $R_{pre}(x) \in \mathcal{J}$ and $g(R_{pre}(x)) \neq \emptyset$. Furthermore, if $y \in g(R_{pre}(x))$, then $R_{post}(y) \in f(x)$. This implies that when there is a solution to Problem A on input x, we will find one.

And just like that, we've leveraged a solution to Problem B to solve Problem A. When we complete a reduction, we also need to bound the total pre- and post-processing runtime. If that runtime is polynomial, we say that there is a *polynomial-time reduction* from Problem A to Problem B. Lastly, to set us up for material for next week, we have the useful result

Lemma 4.5. *Let Π and Γ be computational problems such that $\Pi \leq_p \Gamma$. Then:*

1. If $\Gamma \in \mathsf{P}_{\mathsf{search}}$*, then* $\Pi \in \mathsf{P}_{\mathsf{search}}$*.*

2. If $\Pi \notin \mathsf{P}_{\mathsf{search}}$*, then* $\Gamma \notin \mathsf{P}_{\mathsf{search}}$*.*

As we proved in lecture, this reflects the closure of $\mathsf{P}_{\mathsf{search}}$ under reductions. Intuitively, if Γ is in $\mathsf{P}_{\mathsf{search}}$ and Π reduces in polynomial time to Γ, we can conclude that Π is in $\mathsf{P}_{\mathsf{search}}$ as well. Moreover, because we have defined computationally efficient to be polynomial time, we can compose reductions:

Lemma 4.6. *We can compose reductions - if* $\Pi \leq_p \Gamma$ *and* $\Gamma \leq_p \Theta$ *then* $\Pi \leq_p \Theta$*.*

5 Appendix

5.1 Turing Machines

Most courses on the theory of computation (like CS121) use Turing Machines as their main model of computation, whereas we use the (Word-)RAM model because it better suited for measuring the efficiency of algorithms. However, Turing machines can be understood as a small variant of Word-RAM programs, where we make the word size *constant*:

Definition 5.1 (TM-RAM programs)**.** A *TM-RAM* program P is like a RAM program with the following modifications:

1. *Finite Alphabet:* Each memory cell and variable can only store an element from $[q]$ for a finite *alphabet size* q, which is independent of the input length and does not grow with the computation's memory usage.

2. *Memory Pointer:* In addition to the variables, there is a separate $\texttt{mem\_ptr}$ that stores a natural number, pointing to a memory location, initialized to $\texttt{mem\_ptr} = 0$.

3. *Read/write:* Reading and writing from memory is done with commands of the form $\texttt{var}_i = M[\texttt{mem\_ptr}]$ and $M[\texttt{mem\_ptr}] = \texttt{var}_i$, instead of using $M[\texttt{var}_j]$.

4. *Moving Pointer:* There are commands $\texttt{mem\_ptr} = \texttt{mem\_ptr} + 1$ and $\texttt{mem\_ptr} = \texttt{mem\_ptr} - 1$ to increment and decrement $\texttt{mem\_ptr}$.

Philosophically, TM-RAM programs are appealing because one step of computation only operates on constant-sized objects (ones with domain $[q]$). However, as we will discuss below, the ability to only increment and decrement $\texttt{mem\_ptr}$ by 1 does make TM-RAM programs somewhat slow compared to Word-RAM programs.

Note that the number of possibilities for the state of a TM-RAM's computation, excluding the memory contents is: $q^k \cdot \ell$, if there are k variables and ℓ lines in the program

Thus, the computation can be more concisely described as follows:

Definition 5.2 (Turing machine)**.** A *Turing machine* $M = (Q, \Sigma, \delta, q_0, H)$ is specified by:

1. A finite set Q of states.

2. A finite alphabet Σ (e.g. $[q]$).

3. A transition function $\delta : Q \times \Sigma \to Q \times \Sigma \times \{L, R, S\}$.

4. An initial state $q_0 \in Q$.
5. A set $H \subseteq Q$ of halting states.

Theorem 5.3 (Equivalence of TMs and TM-RAMs)**.** *1. There is an algorithm that given TM-RAM program P, constructs a Turing Machine M such that $M(x) = P(x)$ for all inputs x and* $\mathrm{Time}_M(x) = O(\mathrm{Time}_P(x))$.

2. There is an algorithm that given a Turing Machine M, constructs a TM-RAM program P such that $P(x) = M(x)$ for all inputs x and $\mathrm{Time}_P(x) = O(\mathrm{Time}_M(x))$.

Thus Turing Machines are indeed equivalent to a restricted form of RAM programs. The appeal of Turing machines is their mathematically simple description, with no arbitrary set of operations being chosen (allowing any "constant-sized" computation to happen in one step).

What about Turing Machines vs. Word-RAM Programs?

Theorem 5.4. *There is an algorithm that given a Word-RAM Program P constructs a TM-RAM program P' such that $P'(x) = P(x)$ for all inputs x and*

$$\mathrm{Time}_{P'}(x) = O\left((\mathrm{Time}_P(x) \cdot \log \mathrm{Time}_P(x))^2, \right).$$

provided that $\mathrm{Time}_P(x)$ *is at least* $n \cdot \max_i x[i]$ *for an input array x of length n.*

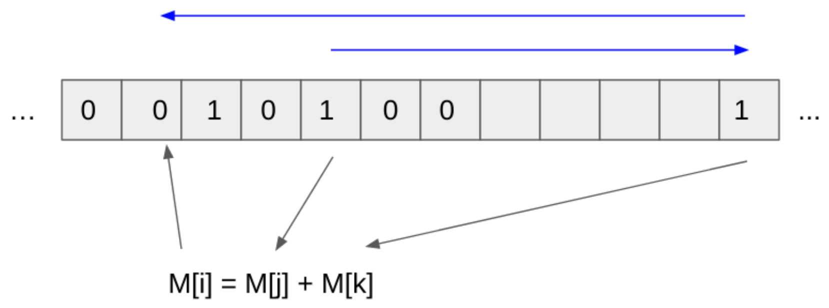

Figure 1: The requirement to move the memory pointer step by step in TM-RAM induces an up to quadratic slowdown vs RAM.