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

Lecture 22: Computational Complexity

Harvard SEAS - Fall 2021 *Nov. 18, 2021*

1 Announcements

Recommended Reading:

- MacCormick §5.3–5.5, Ch. 10, 11
- Salil is in Oberwolfach, no office hours today

In the prior lecture, we talked about the tiling problem — covering an infinite grid with finitely many tiles. To show this is unsolvable, we want a model of computation where everything is local — we can only "move" a single step in any direction in any step of computation.

2 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 2.1 (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 `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 `mem_ptr`.

See Figure 1

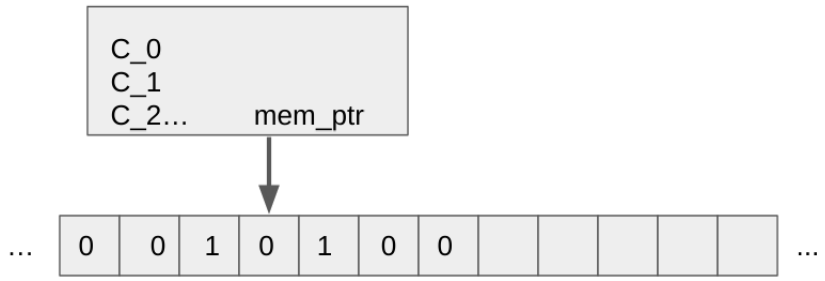

Figure 1: A TM RAM machine, with memory pointer and commands.

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 `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 2.2 (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.

Semantics of δ: $\delta(q, \sigma) = (q', \sigma', m)$ means that if the current state is q and $M[\texttt{mem\_ptr}]$ equals σ, then we transition to state q', overwrite $M[\texttt{mem\_ptr}]$ with σ' and increment/decrement/maintain `mem_ptr` according to whether $m = R$ ("move right"), $m = L$ ("move left"), $m = S$ ("stay in place").

Theorem 2.3 (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 $\text{Time}_M(x) = O(\text{Time}_P(x))$.*

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 $\text{Time}_P(x) = O(\text{Time}_M(x))$.

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?

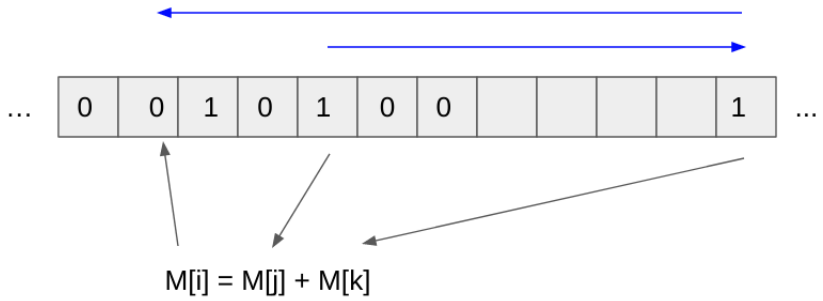

Figure 2: The requirement to move the memory pointer step by step in TM-RAM induces an up to quadratic slowdown vs RAM.

Theorem 2.4. *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.

Proof Sketch.

- Memory of P: encoded as $S \cdot w$ bits in the memory of P', at a point in the computation when P uses S bits and has a word size of w.
- Values of the variables of P: These take $O(w)$ bits and are stored in memory locations of P' near `mem_ptr`, and copied (using $O(w^2)$ steps) every time P' wants to move `mem_ptr` to a different simulated memory location of P.
- Simulating one step of P: P' scans over its entire $S \cdot w$-bit memory, doing updates (arithmetic operations, read/write operations, possibly increasing S and w, etc.) and copying the values of its variables as it goes. This takes time $O(S \cdot w^2)$.

Thus if P runs for T steps, the entire simulation takes time

$$O(T \cdot (S \cdot w^2)).$$

As in the XOR-extended Word RAM problem on Problem Set 4, we have $S \leq T + n = O(T)$ and $w = O(\max\{\log n, \max_i x[i], S\}) = O(\log T)$, we get time

$$O(T \cdot (S \cdot w^2)) = O((T \log T)^2).$$

□

So TM-RAMs and Turing Machines can simulate Word-RAM programs, but with a bit more than a quadratic slowdown in runtime. This is a lot better than the relation between RAM programs and Word-RAM programs, which incurs an exponential slowdown in simulating the former by the latter (as demonstrated by your experiments on PS4).

3 The Extended Church–Turing Thesis

Last time we saw the Church–Turing Thesis, which asserted that all reasonable/intuitive/realizable models of computation are equivalent.

> Extended Church–Turing Thesis v1: Every physically realizable model of computation can be simulated by a Turing Machine (or TM-RAM, or a Word-RAM program) with only a *polynomial* slowdown in runtime.

The Extended Church–Turing Thesis is more debatable than the standard Church–Turing thesis. In particular, randomized algorithms, massively parallel computers, and quantum computers all could potentially provide an exponential savings in runtime. (For randomized algorithms, however, it is conjectured that they provide only a polynomial savings, as discussed in Lecture 8.)

If we modify the statement with some qualifiers, then these challenges no longer apply:

> Extended Church–Turing Thesis v2: Every physically realizable, deterministic, and sequential model of computation can be simulated by a Turing Machine (or TM-RAM, or a Word-RAM program) with only a polynomial slowdown in runtime.

"Deterministic" rules out both randomized and quantum computation, as both are inherently probabilistic. "Sequential" rules out parallel computation. This form of the Extended Church–Turing Thesis has stood the test of time for the approximately fifty years since it was formulated, even has computing technology has changed tremendously in that time.

4 Computational Complexity

In the past few classes, we have studied *computability theory*, where we classify computational problems according to whether they are solvable or not. *Computational complexity*, on the other hand, aims to classify the problems that are solvable according to the amount of resources (e.g. time) that they require. Indeed, this has been the goal for much of the course, where we have tried to find the fastest possible algorithms for different problems.

For example, we've seen algorithms that are:

- Linear time: Shortest Paths, 2-Coloring in time $O(n + m)$.
- Nearly linear time: Sorting, Interval Scheduling (Decision, Optimization, Coloring) in time $O(n \log n)$.
- Polynomial time: Bipartite Matching in time $O(nm)$, 2-SAT in time $O(nm)$.[1]

[1] However, thanks to one of your questions in Ed, we discovered a linear-time algorithm for 2-SAT is actually known, based on DFS (which we haven't covered) rather than BFS/Reachability.

- Exponential time: k-Coloring for $k \geq 3$, k-SAT for $k \geq 3$, Independent Set, and Longest Path in time $O(c^n)$ for constants $c > 1$.

To develop a robust and clean theory for classifying problems according to computational complexity, we make two choices:

- A problem-independent size measure. Recall that we allowed ourselves to use different size parameters for different problems (array length n and universe size U for sorting; number n of vertices and number m of edges for graphs, number n of variable and number m of clauses for Satisfiability). To classify problems, it is convenient to simply measure the size of the input by its *length N in bits*. For example:
  - Array of n numbers from universe size U: $N = \Theta(n \log_2 U)$.
  - Graphs on n vertices and m edges in adjacency list notation: $N = \Theta((n+m)\log n)$.
  - 3-SAT formulas with n variables and m clauses: $N = \Theta(m \log n)$.
- Polynomial slackness in running time: We will only try to make coarse distinctions in running time, e.g. polynomial time vs. super-polynomial time. If the Extended Church–Turing Thesis is correct, the theory we develop will be independent of changes in computing technology. It is possible to make finer distinctions, like linear vs. nearly linear vs. quadratic, if we fix a model (like the Word-RAM), and a newer subfield called *Fine-Grained Complexity* does this.

To this end, we define the following *complexity classes*.

Definition 4.1.
- 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)
$$\mathsf{EXP}_{\mathsf{search}} = \bigcup_c \mathsf{TIME}_{\mathsf{search}}\left(2^{n^c}\right), \qquad = \bigcup_c \mathsf{TIME}\left(2^{n^c}\right).$$

(Remark on terminology: what we call $\mathsf{P}_{\mathsf{search}}$ is called Poly in the MacCormick text, and is often called FP elsewhere in the literature.)

By this definition, Shortest Paths, 2-Coloring, Sorting, Interval Scheduling, Bipartite Matching, and 2-SAT are all in $\mathsf{P}_{\mathsf{search}}$ (as well as P for decision versions of the problems). However, all we know to say about 3-Coloring, 3-SAT, Independent Set, or Longest Path is that they are in $\mathsf{EXP}_{\mathsf{search}}$. Can we prove that they are not in $\mathsf{P}_{\mathsf{search}}$?

The following seems to give some hope:

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

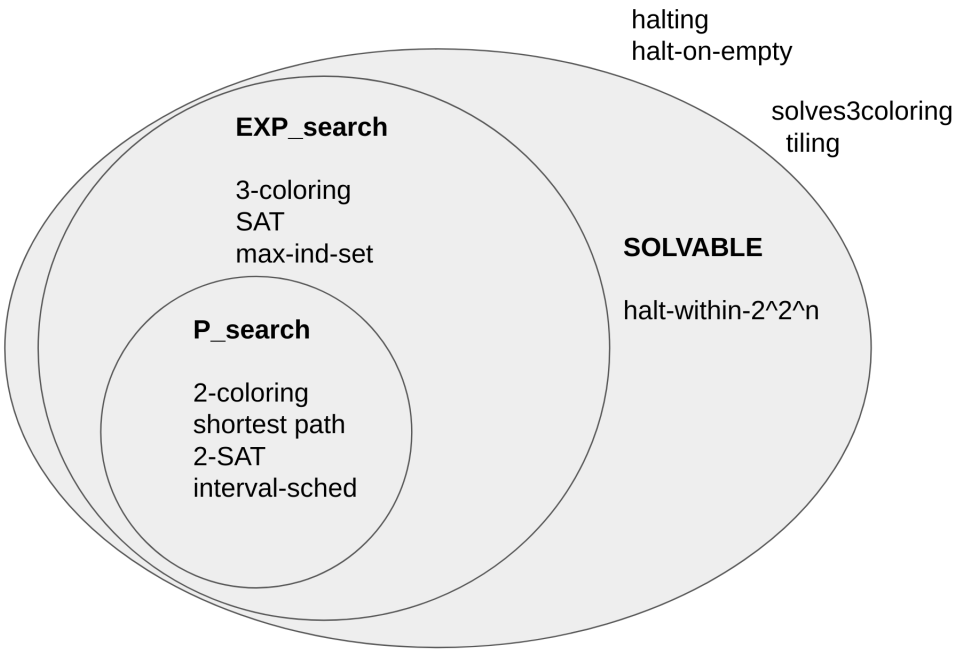

Figure 3: Complexity classes, and the positions of problems we've encountered so far.

We won't give the proof of this theorem, but it is proven using an argument similar to our proof that the Halting Problem is not Solvable. Indeed, an example, of a problem in $\mathsf{EXP}_{\mathsf{search}} - \mathsf{P}_{\mathsf{search}}$ (in fact $\mathsf{EXP} - \mathsf{P}$) is the problem of deciding whether a Word-RAM program halts on an input x of length n within 2^n steps.

Next we might try to obtain more intractable problems via reductions, similarly to what we did for solvability.

Definition 4.3. 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, on an input of length N the reduction should run in time $O(N^c)$, if we count the oracle calls as one time step (as usual).

Lemma 4.4. *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* $\Gamma \notin \mathsf{P}_{\mathsf{search}}$*.*

Unfortunately, we don't know how to reduce the problems we know in $\mathsf{EXP}_{\mathsf{search}} - \mathsf{P}_{\mathsf{search}}$ (like Bounded Halting) to many of the problems we care about (like Independent Set, 3-Coloring, and Longest Path). These problems have additional structure, which will require us to define and study a different complexity class, NP, next time.