CMPT 407: Computational Complexity **Summer 2017**

Lecture 1: Introduction - W2: 8 19 May

Lecturer: Valentine Kabanets *Scribe: Lily Li*

1.1 General Review

A **function problem** is a map $f : \Sigma^* \to \Sigma^*$ where Σ^* is a string. These could be rather complicated, such as taking as input a start and end vertex and outputting a shortest path (these are strings under the appropriate encoding).

A **decision problem** is a function problem where the range is the boolean values $\{yes, no\}$. And **language** associated with a decision problem is the set of all strings that map to *yes* under f i.e $\{x \in \Sigma^* : f(x) = yes\}$.

1.2 Turing Machine

You know what these are. They are Turing's attempt to simulate human computers. Formally a Turing Machine (TM) M is a tuple

$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{acc}, q_{reg})$$

where Q is the set of states, Σ are the alphabet, Γ is the tape symbols (these are a superset of Σ), q_0 is the set of start states, q_{acc} is the set of accepting states, and q_{reg} is the set of rejecting states.

$t_M(x)$ is the **running time** of a TM M on input $x \in \Sigma^*$. It represents the number of steps M takes on x before halting $t_M(x) = \infty$ if M does not halt on x. If there exists a function $t : \mathbb{N} \to \mathbb{N}$ such that $t_M(x) \leq t(|x|)$ then M is t-bounded. *This TM only needs to be "good" asymptotically, finitely many issues can be put in a look up table and fixed in constant time.*

The **space** of a TM M is the total number of work tape cells touched by M during the computation of x. Again, if there exists a function $s : \mathbb{N} \to \mathbb{N}$ with the necessary properties, then M is s-bounded.

k-tape TM consists of k infinite tapes. Of these, one is the input and one is the output leaving $k - 2$ work tapes.

Theorem 1.1 *A k-tape TM can be* efficiently *simulated by a one-tape TM with only a quadratic time slowdown.*

Proof: Allocate k parallel track on the one-tape TM. Delineate each track with the special tape symbol $*$. For each step of the k-tape TM first sweep the tape on the one-tape TM to find the track to read. ■

Remark: this quadratic time slowdown for a one-tape TM simulation is unavoidable! Surprisingly there is only a log linear slow if we were to use a two-tape TM simulation i.e. if the k-tape TM M is t-bounded then simulating M using a 2-tape TM takes $O(t \log t)$ time.

Theorem 1.2 *A k-tape TM M over tape alphabet $\Gamma = \{0, 1\}$ can be simulated by a k-tape TM M' over tape alphabet $\Gamma' = \{0, 1, \#, -\}$ with at most $O(\log |\Gamma|)$ factor slowdown (both use input alphabet $\Sigma = \{0, 1\}$) i.e. if M is t-bounded then M' is $t \log |\Gamma|$ bounded.*

The key to proving this theorem is to note that it is possible to encode each symbol in Γ by a number. There are $\lceil \log |\Gamma| \rceil$ numbers needed which results in the $\log |\Gamma|$ factor slowdown.

1.2.1 Universal Turing Machines

A **Universal TM** (UTM) is a TM U which takes as input a description of a TM M as a string and another string $w \in \Sigma^*$ and simulates M on w. The exists of UTM foretold the creation of the general purpose computer and hints at their limitations (e.g. cannot detect self replicating programs). We try to simulated a t-bounded TM M using U in $O(t \log t)$ time as follows:

Theorem 1.3 *There is a 2-tape UTM U that given $\langle M, w \rangle$, where M is a k-tape TM and w is an input to M, will simulated M on w with at most a log linear slowdown.*

Proof: We will mainly focus on the difficult of simulating a k-tape TM using a 2-tape TM (other issues involving the number of tape symbols and states of the k-tape TM can be solved by adding a constant factor).

The gist of the proof is to use one tape to hold the k tapes as k tracks. Then at some special position 0, we will keep all k symbols currently being scanned for M. For each transition step we will move each track so that the new tape cell to be scanned is placed at position 0. We can't move the whole track since that would be too expensive so we will only move a portion of the tape using the second tape to help.

Partitioning the first tape proceeds as follows: divide the chunks to the right of the 0 position into zones $R_0, R_1, ...$ (read left to right) where zone R_i has 2^{i+1} cells. Similarly, divide the space to the left of 0 into zones $L_0, L_1, ...$ (read right to left) where zone L_i has size 2^{i+1}. As follows, where P_0 is position 0, $L_{i,j}$ is the j-th cell of L_i, similarly for $R_{i,j}$:

$$\cdots, L_{1,3}, L_{1,2}, L_{1,1}, L_{1,0}, L_{0,1}, L_{0,0}, P_0, R_{0,0}, R_{0,1}, R_{1,0}, R_{1,1}, R_{1,2}, R_{1,3}, \cdots$$

Next, introduce a new *special blank* symbol different from the tape symbols of M. For zone Z_i, if Z_i has all special blank cells then it is empty, if Z_i has 2^i special blank symbols then it is half full, and if Z_i has no special blank symbols then it is full.

In a preprocessing step for U, initialize each zone i, for $i = 0, ..., \log t$ to be half full. Through out the simulation we maintain the following invariant:

1. Position 0 of U's tape contains non-"special blank" in each track (this is the k-tuple of symbols currently scanned by M.
2. For each i, $0 \leq i \leq \log t$, either both L_i and R_i are half-full, or exactly one of them is full while the other one is empty.

Next we demonstrate the procedure for moving a track of the tape to the left (this is equivalent to moving one of the heads of M to the right)

1. Scan the zones $R_0, R_1, ...$ until you find one which is non-empty. Suppose this zone is R_{i_0}.
2. Shift the contents of position 0 to L_0, L_0 to L_1, and so on until L_{i_0-1} to L_{i_0}. Since $R_0, ..., R_{i_0-1}$ are all empty, $L_0, ..., L_{i_0-1}$ are all full so will now half fill $L_1, ..., L_{i_0}$.
3. Take the first (left-most) non-special-blank cell in R_{i_0} and put it in position 0. Move the remaining non-special-blank cell of R_{i_0} down to have fill the zones $R_0, R_1, ...$

4. Observe that $R_0, ..., R_{i_0-1}$ and $L_0, ..., L_{i_0-1}$ are all half-full and L_{i_0} is full or half full depending on if R_{i_0} is empty or half full.

Finally we perform amortized analysis of the running time. The key observation from the above example is that after finding and shifting R_{i_0}, the next at least $2^{i_0} - 1$ will have to look no farther than R_{i_0-1} (all $R_0, ..., R_{i_0-1}$ are half full). Thus the total amount of shifting we need to do is:

$$\sum_{i=0}^{\log t} \frac{t}{2^i} \cdot O(2^i) \leq O(t \log t)$$

This is only the time needed for one track, but we said that all the extra time for the tracks and the symbols can be hidden behind a constant. ■

1.3 Complexity Classes

If n is the input size with function $f : \mathbb{N} \to \mathbb{N}$

- $\mathsf{Time}(f(n)) = \{L : L \text{ is decided by a TM in at most } f(n) \text{ steps}\}$
- $\mathsf{Space}(f(n)) = \{L : L \text{ is decided by a TM by tounching at most } f(n) \text{ worktape cells}\}$

Complexity classes are typically defined as collections of languages over the binary alphabet with the tape symbols made arbitrary. Also any arbitrary (fixed) number of tapes are allowed. Examples include:

- $\mathsf{EXP} = \bigcup_{k \geq 1} \mathsf{Time}(2^{n^k})$
- $\mathsf{L} = \mathsf{Space}(\log n)$

In general, a complexity class is a collection of languages decidable within a given amount of computational resources.

1.4 Linear Speedup Theorem

Theorem 1.4 *(Linear Time Speedup) If a Language L is decided by some TM M in time $t(n)$, then for any $\epsilon > 0$, there is a TM M' that decides L in time at most $n + \epsilon n + \epsilon t(n) + 2$.*

Proof: The idea is to encode k-tuples of symbols of the tape alphabet of M as one tape symbol of M'. The original input of size n compresses to an input to M' of size n/k. When we simulate M on M', every k steps of M will translate to a constant number (inparticular six) of steps in M'. The gist of the move procedure is to move the head in M': L, R, R, L to read $3k$ symbols altogether. ■

Theorem 1.5 *(Linear Space Speedup) If a language L is decided by some TM M in space $s(n)$, then for any $\epsilon > 0$ there is an TM M' that decides L in space at most $\epsilon n + \epsilon s(n) + 2$.*

As a result all constant factors can be ignored in the discussion of space and time complexity when the the bound is greater than n.

1.5 Reduction

A language L_1 is reducible to L_2, denoted $L_1 \leq L_2$ if there is an *efficiently* computable function $f : \Sigma^* \to \Sigma^*$ such that

$$x \in L_1 \implies f(x) \in L_2$$
$$x \notin L_1 \implies f(x) \notin L_2$$

You should think of L_2 has the harder problem. Pay close attention to the word *efficiently*. For not it means polynomial time.

1.6 Completeness

In complexity class C, language L is *C-complete* if

1. $L \in C$
2. Every language in L is reducible to L. *Note that reducibility is with respect to the complexity class. There is no sense in doing poly-time reductions if we are in the class* L *for example.*

A language L is *C-hard* if every language in C is reducible to L (though L might not be in C).

Extended Church-Turing Thesis: Everything *efficiently* computable on a physical computer is *efficiently* computable on a Turing Machine.

1.7 Time Hierarchy Theorem

The following shows that giving a TM more time strictly increases the class of languages that it can decide. First a **time-constructible** function $f : \mathbb{N} \to \mathbb{N}$ is one where there exists a TM such that given the input 1^n, writes down $1^{f(n)}$ in $O(f(n))$ time. Can you think of functions which satisfies this property? In the following only consider time-constructible running times. **Proper** complexity functions is a similar notion but is a bit nicer since on input of size n we are allowed time $O(n + f(n))$ and $O(f(n))$ space. Thus $\log n$ is proper but not time-constructible.

Theorem 1.6 *If f, g are time-constructible functions satisfying $f(n) \log f(n) = o(g(n))$, then*

$$\mathbf{DTIME}(f(n)) \subsetneq \mathbf{DTIME}(g(n)) \tag{1.1}$$

a.k. for any proper $t(n), T(n) \geq n$ s.t.

$$T(n) \geq \omega(t(n) \log t(n))$$

we have

$$\mathsf{Time}(t) \subsetneq \mathsf{Time}(T)$$

Proof: Use a diagonalization technique. We consider the encoding $\langle M \rangle$ of TM M. We also allow "padded encodings $\langle M \rangle 01^m$ for all $m \geq 0$ so that M has encodings of arbitrarily large size. Put TMS $M_1, M_2, ..., M_i, ...$ that run in time at most $t(n)$ down rows of the table (consided clocked TM here only since we don't know

how to tell when a TM stops on a certain input) and encodings $\langle M_1\rangle, ..., \langle M_j\rangle, ...$ across the columns. For cell i, j run M_i on input $\langle M_j\rangle$. Add necessary timeout $t(n)$ so that M_i stops on all inputs. Define diagonal language D as follows:

$$D = \{\langle M_i\rangle : M_i \text{ does not accept } \langle M_i\rangle \text{ in } \leq t(n) \text{ steps}\}$$

thus D is not any of TM in our table. Observe that $D \notin \mathsf{Time}(t)$ by contradiction. Suppose that $D \in \mathsf{Time}(t)$ then there exists a TM M which decides D. Since we have enumerated all TM in time $t(n)$, $M = M_j$ for some $j \geq 1$. By running M_j on $\langle M_j\rangle$ we reach a contradiction. Next note that $D \in \mathsf{Time}(T)$ by the UTM theorem since we can simulate $D \in \mathsf{Time}O(t \log t)$ (including some constant factor lost due to the structure of the input machine). Since $T \gg \omega(t \log t)$, $D \in \mathsf{Time}(T)$. ■

1.8 Space Hierarchy Theorem

Similar idea as the above. First $f : \mathbb{N} \to \mathbb{N}$ is a **space-constructible** function if there is a TM M that, given any n-bit input, constructs $f(n)$ in space $O(f(n))$. The proof is essentially the same as the above except that we do not have a factor of $\log f(n)$ since the UTM for space-bounded computations has only a constant time overhead.

Theorem 1.7 *If f, g are space-constructible functions satisfying $f(n) = o(g(n))$, then*

$$\mathbf{SPACE}(f(n)) \subsetneq \mathbf{SPACE}(g(n)) \tag{1.2}$$

For any proper $s(n), S(n) \geq \log n$ s.t. $S(n) \geq \omega(s(n))$ then

$$\mathsf{Space}(s) \subsetneq \mathsf{Space}(S)$$

Theorem 1.8 *(Gap Theorem) There exist non-proper $t(n)$ such that*

$$\mathsf{Time}(t) = \mathsf{Time}(2^t)$$

(there is a special definition for this function).

Remark: if your function is weird, expect that something weird will happen.

1.9 Nondeterministic Time Hierarchy Theorem

1.9.1 Class NP

You know this, special since they are poly-time verifiable.

$$\mathsf{NP} = \{L : \exists V \text{ verifier s.t. } \forall x, x \in L \iff \exists y, |y| \leq poly(|x|), V(x, y) \text{ accepts}\}$$

Theorem 1.9 *If f, g are time-constructible functions satisfying $f(n+1) = o(g(n))$, then*

$$\mathbf{NTIME}(f(n)) \subsetneq \mathbf{NTIME}(g(n)) \tag{1.3}$$

Proof:

■

1.10 Ladner's Theorem: Existence of NP-intermediate problems

Theorem 1.10 *Suppose* $\mathsf{P} \neq \mathsf{NP}$. *Then* $\exists L \in \mathsf{NP}$ *such that* $L \notin \mathsf{P}$ *and* L *is not* NP*-complete.*

Proof: More diagonalization, but more special this time! Consider

$$\mathsf{SAT_H} = \{\phi 01^{n^{H(n)}}\} : \phi \in \mathsf{SAT}, |\phi| = n$$

(read through this definition carefully each element of $\mathsf{SAT_H}$ is specifically defined by $H(n)$).

First note that $\exists H(n)$ computable in P such that

1. $\mathsf{SAT_H} \in \mathsf{P} \implies H(n) \leq O(1)$; read $H(n)$ is basically a constant.
2. $\mathsf{SAT_H} \notin \mathsf{P} \implies \forall c \exists n_0 \forall n \geq n_0 : H(n) > c$; read $H(n)$ is greater than every constant.

Next we claim that $\mathsf{NP} \neq \mathsf{P} \implies \mathsf{SAT_H} \notin \mathsf{P}$. **Proof:** By contrapositive: assume that $\mathsf{SAT_H} in \mathsf{P}$. Then we can your algorithm for $\mathsf{SAT_H}$ to solve SAT! Input is $\phi \in \mathsf{SAT_H}$. Constructing the padded version (input to $\mathsf{SAT_H}$) take poly-time since $H(n)$ is bounded by a constant. Then run your poly-time algorithm $\mathsf{SAT_H}$ algorithm. This is a poly-time algorithm for SAT :(. ■

Finally we claim that $\mathsf{NP} \neq \mathsf{P} \implies \mathsf{SAT_H} \notin \mathsf{NPC}$. **Proof:** Again by contradiction: suppose that $|\phi| = m$. If you use the second then the first claim in a somewhat straight-forward way (not forgetting the definition of $\mathsf{SAT_H}$), the rest of the proof should be apparent. First suppose if $\mathsf{SAT} \leq \mathsf{SAT_H}$. ■

The remainder of the theorem is recursive (o.O, yeah weird). After each step you can square root the input, after repeated trails you get down far enough to just brute force. So if $\mathsf{SAT_H} \in \mathsf{P}$ or $\mathsf{SAT_H} \in \mathsf{NP}$ we end up with a polynomial time algorithm for SAT. That is the structure of the proof.

It remains to define $H(n)$ (recursively) as follows:

$$H(n) = \begin{cases} \min_{i < \log\log n} & \text{such that } M_i \text{ decides } \mathsf{SAT_H} \\ \log\log n & \text{otherwise} \end{cases}$$

■

The whole thing is a bit wonky, beware!

We don't know any problems which are not in P and not in NP (obviously, since that would solve the crazy problem), but we do have some candidates: such as graph isomorphism (recent result!) and integer factorization.

1. Size two clauses
2. Horn clauses
3. Dual Horn
4. XOR equations
5. Trivial clauses (satisfy all zero or all one assignments)

Theorem 1.11 *(Schaefer's Dichotomy) If* CSP*s use clauses of type (i) only, for* $1 \leq i \leq 5$*, then* $\mathsf{CSP} - \mathsf{SAT} \in \mathsf{P}$. *Otherwise* $\mathsf{CSP} - \mathsf{SAT}$ *is* NP*-complete (the numbering is from above). E.g.* $2 - \mathsf{SAT}$ *is poly-time solvable.*

Here is an example of Schaefer's Dichotomy: we will mix Dual Horn clauses ($x_1 \vee x_2 \vee x_3$) and non-Dual Horn clauses of size two to get a CSP which is NP-complete. To show this, we will reduce 3-Colorability ($3-\mathsf{COL}$ to this problem. Take an instance of $3-\mathsf{COL}$.

1.11 Reducibility and NP-Completeness

Theorem 1.12 *Properties of reductions:*

1. *(Transitivity) If $L \leq_p L$ and $L \leq_p L$", then $L \leq_p L$".*
2. *If language L is NP-hard and $L \in P$, then $\mathsf{P} = \mathsf{NP}$.*
3. *If language L is NP-complete, then $L \in P$ if and only if $\mathsf{P} = \mathsf{NP}$.*

Proof:

■

1.12 SAT

Theorem 1.13 ***Cook-Levin Theorem***

1. SAT *is* NP*-complete.*
2. 3SAT *is* NP*-complete.*

Proof:

■

Lemma 1.14 SAT *is* NP*-hard.*

Proof:

■

1.13 Decision v.s. Search

Theorem 1.15 *Suppose that* $\mathsf{P} = \mathsf{NP}$. *Then, for every* NP *language L and verifier TM M for L, there is a polynomial time TM B that on input $x \in L$ outputs a certificate for x (with respect to L and M).*

Proof:

■

1.14 coNP, EXP, and NEXP

1.14.1 coNP

If L is a language then we define $\bar{L}$ to be the complement of L. The language $\mathsf{coNP} = \{L : \bar{L} \in \mathsf{NP}\}$. Note that coNP is not the complement of NP. In fact the two languages have P in common. An alternative definition of coNP is as follows:

Definition 1.16 *For every $L \subseteq \{0,1\}^*$, $L \in \mathsf{coNP}$ if there exists a polynomial $p : \mathbb{N} \to \mathbb{N}$ and a polynomial-time TM M such that for every $x \in \{0,1\}^*$,*

$$x \in L \iff \forall u \in \{0,1\}^{p(|x|)}, M(x,y) = 1$$

An example of a coNP-complete language is the following:

$$\mathsf{TAUTOLOGY} = \{\phi : \phi \text{ is a tautology } \}$$

1.14.2 EXP and NEXP

1.15 TM v.s. Circuit

For the TM, the algorithm M is the *same* for any length $|w|$. Circuits C_n is a fixed algorithm which only works for a *fixed* length input. For different lengths the circuits might potentially be quite different. Circuits are more "natural" than TMs.

Example 1.17 *Given $w \in \{0,1\}^*$, decide if $w \neq 0^*$ (not all zero string. For a TM: input w scan the tape. Circuit C_4: complete three of height 2 with or at each node.*

Generally when deciding with circuits you need to slice the language depending on the input size. To decide L with circuits, you need a family of circuits $C = \{C_n\}$ where C_n decides L_n. We *want* circuits to be small i.e. use only a few number of logic gates.

Circuit Minimization Task:

1. given a truth table of $f : \{0,1\}^n \to \{0,1\}$ you want the smallest circuit for f (try not encoding the entire 2^n look-up table.
2. Given a circuit $C_n : \{0,1\} \to \{0,1\}$ find the smallest equivalent circuit.

Unfortunately both problems can only be solved by exhaustive search (currently). We also cannot categorize these problems in our hardness classes NP, NP-complete etc.

Efficient Algorithms: for uniform model (TM) it is the class P, while for non-uniform models (circuits) we want the class of poly-size which is a family $\{C_n\}$ such that $C_n \leq O(n^d)$ for some constant d.

Theorem 1.18 $\mathsf{P} \subseteq \mathsf{PolySize}$.

Lemma 1.19 *For any proper $t : \mathbb{N} \to \mathbb{N}$,* $\mathsf{Time}(t) \subseteq \mathsf{Size}(t \cdot \log t)$

Proof: First show the weaker inclusion $\mathsf{Time}(t) \subseteq \mathsf{Size}(t^2)$. As follows: first consider only a finite chunk of the tape of size $2t(n)$ this is where the interesting things are going to happen. Then you want to do something similar to the SAT proof, that is you want to specify the configurations and only look at the local changes. There is at most $t(n)$ configurations and every configuration is of length $2t(n)$ so that is how you get the $O(t^2)$.

To get $\mathsf{Size}(t \log t)$ we need to introduce **oblivious** TM. These are TM that only cares about the size of the inputs; on inputs of the same size the movement of the TM is *exactly the same*! As it turns out all languages in $\mathsf{Time}(t)$ can be decided in time $O(t \log t)$ by and oblivious time TM (how do make such an oblivious machine).

Using this oblivious TM we can simplify the grid to $O(t)$. ■

There are problems which are in $\mathsf{PolySize}$ but are not in P. Consider the following: