CMPT 407 - Complexity Theory: Lecture 4-7

Valentine Kabanets

September 13, 2017

1 Complexity Classes

Below n will denote the input size.

For some function $f : \mathbb{N} \to \mathbb{N}$,

- $\mathsf{Time}(f(n)) = \{L \mid L$ is decided by some TM in at most $f(n)$ steps$\}$
- $\mathsf{Space}(f(n)) = \{L \mid L$ is decided by some TM touching at most $f(n)$ worktape cells$\}$

Usually, the complexity classes are defined as collections of languages over the binary alphabet $\{0, 1\}$ (i.e., inputs are binary strings). The tape alphabet may be arbitrary.

Remark 1. *In the complexity class definition above, we allow TMs with arbitrary (fixed) number of tapes, i.e., we don't require all TMs to be 1-tape (or 2-tape).*

For example,

- $\mathsf{P} = \cup_{k \geq 1} \mathsf{Time}(n^k)$,
- $\mathsf{EXP} = \cup_{k \geq 1} \mathsf{Time}(2^{n^k})$,
- $\mathsf{L} = \mathsf{Space}(\log n)$,
- $\mathsf{PSPACE} = \cup_{k \geq 1} \mathsf{Space}(n^k)$.

We will see later in the course that $\mathsf{P} \subsetneq \mathsf{EXP}$, i.e., the class of polytime decidable languages is strictly smaller than the class of exponential-time decidable languages. (Also, we'll see that $\mathsf{L} \subsetneq \mathsf{PSPACE}$.)

In general,

> *a complexity class is a collection of languages decidable within a given amount of computational resources.*

2 Reductions

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

$$x \in L_1 \Rightarrow f(x) \in L_2,$$
$$x \notin L_1 \Rightarrow f(x) \notin L_2.$$

For now, "efficiently" means "in polytime".

Interpretations of $L_1 \leq L_2$:

- L_1 is at most as hard as L_2: can solve L_1, given an algorithm for L_2 [algorithm design technique]
- L_2 is at least as hard as L_1: if L_1 is known to be hard, then the reduction $L_1 \leq L_2$ yields that L_2 is also hard [lower bound technique]

3 Reductions: Example

- $3SAT = \{\phi \mid \phi$ is a satisfiable 3CNF formula$\}$ (Recall that a 3CNF is a conjunction of clauses, where each clause is a disjunction of three literals; a literal is a variable or its negation.)
- $IS = \{(G, k) \mid$ G is a graph with an independent set of size $\geq k\}$ (Recall that an independent set is a subset of vertices such that no two vertices in the subset are connected by an edge.)

Reduction: $3SAT \leq IS$

Given $\phi = (x \vee y \vee \neg z) \wedge (\neg x \vee w \vee z) \wedge \ldots ()$ with m clauses, produce the graph G_ϕ that contains a triangle for each clause, with vertices of the triangle labeled by the literals of the clause. Plus, add an edge between any two complementary literal from different triangles. In our example, we have triangles on $x, y, \neg z$ and on $\neg x, w, z$, plus the edges $(x, \neg x)$ and $(\neg z, z)$. Finally, set $k = m$.

Theorem 2. *ϕ is satisfiable iff G_ϕ has an independent set of size at least k.*

Proof. Exercise. □

4 Completeness

For a complexity class C, we say that a language L is *C-complete* if

1. $L \in C$, and
2. every language in C is reducible[1] to L.

We say that a language L is *C-hard* if every language in C is reducible to L (i.e., L may or may not be in C).

[1] To be meaningful, the complexity of reduction is chosen in accordance with the class C in question. For example, for P, we use logspace-reductions rather than polytime-reductions, when defining P-completeness.

Interpretation A C-complete language L is a "hardest" language in C (everybody else in C is at most as hard as L).

Usefulness A C-complete language *captures* the complexity of the entire class C. So, we can reason about C by thinking about a single concrete problem from C.

5 Extended Church-Turing Thesis

There are many other models of computation (multi-head TMs, multi-dimensional tape TMs, computers with RAM, etc.) All of them can be simulated by a 1-tape TM with only polynomial slowdown. This means that if we ignore polynomial speedups, any efficient (polytime) computation on any reasonable model of computation can be performed by a polytime 1-tape TM.

> **Extended Church–Turing Thesis** *Everything* efficiently *computable on a physical computer is* efficiently *computable by a Turing machine.*

(Quantum computers pose a potential challenge to this thesis, but they haven't been built yet.)

6 Hierarchy Theorems

Hierarchy theorems were among the earliest results in complexity theory (proved in the 1960's). Informally, the time hierarchy theorem says that in more time, one can decide strictly more languages.

The proof of this result builds upon diagonalziation (a method from computability theory) and the existence of efficient universal Turing machines.

In this course, we will always use *proper* complexity functions $f(n)$. A function $f(n)$ is called *proper* if there is a TM M that, on input 1^n, outputs exactly $f(n)$ symbols and runs in time $O(f(n)+n)$ and space $O(f(n))$. The usual functions like $\log n$, $\sqrt{n}$, n^2, 2^n, $n!$ are proper. Also, if f and g are proper, then so are $f+g$, fg, $f(g)$, f^g, 2^g.

Theorem 3 (Time Hierarchy)**.** *For any proper complexity functions $t(n), T(n) \geq n$ such that $T(n) \in \omega(t(n)\log t(n))$ (i.e., T grows faster than any constant multiple of $t\log t$), we have*

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

Proof. We define a language D such that (1) $D \notin \mathsf{Time}(t(n))$, but (2) $D \in \mathsf{Time}(T(n))$.

Item (1) is argued via diagonalization. Consider an encoding $\langle M\rangle$ of a TM M. We allow M to have infinitely many encodings by allowing "padded" encodings of the form $\langle M\rangle 01^m$, for any $m \geq 0$. In other words, we allow an encoding of M to be extended by any number of symbols 1. (Allowing infinitely many such encodings of the same TM M implies that M has encoding of arbitrary large size, which in turn allows for asymptotics to kick in.)

So we enumerate all TMs $M_1, M_2, \ldots$ that run in time at most $t(n)$. The problem is we don't know how to tell if a given TM M runs in at most time $t(n)$; in fact, this problem is not possible to solve (it's incomputable). But there is an easy fix! We consider clocked TMs. Namely, we will simulate each given TM M for at most $t(n)$ steps (on inputs of size n) by keeping track of the number of steps and making sure to stop the simulation after $t(n)$ steps. (This is where we need the assumption that the time function $t(n)$ be proper, so that we can compute the value $t(n)$ and use it to count off exactly $t(n)$ steps of computation.)

Now we can define our "diagonal" language D as follows:

$$D = \{\langle M \rangle \mid \text{TM } M \text{ does not accept } \langle M \rangle \text{ within } t(n) \text{ steps}\}.$$

Claim 4. *$D \notin \mathsf{Time}(t(n))$.*

Proof of Claim 4. By contradiction, suppose that some TM M decides the language D in time $t(n)$. Since we enumerate over all possible TMs, we will have that TM in our enumeration, say as M_j for some $j \geq 1$.

Consider what happens if we simulate M_j on input $\langle M_j \rangle$.

1. Case 1: If M_j accepts $\langle M_j \rangle$ in $t(n)$ steps, then (by the definition of D) we get that $\langle M_j \rangle \notin D$, and so (since M_j decides D) that M_j does not accept $\langle M_j \rangle$. A contradiction.

2. Case 2: IF M_j does not accept $\langle M_j \rangle$ in $t(n)$ steps, then (by the definition of D) we get that $\langle M_j \rangle \in D$, and so (since M_j decides D) that M_j accepts $\langle M_j \rangle$. A contradiction.

Since we get a contradiction in both cases above, we conclude that M_j cannot exist, i.e., D is not decided by any TM in time $t(n)$. □

Claim 5. *$D \in \mathsf{Time}(T(n))$.*

Proof of Claim 5. Given input $x = \langle M \rangle$ for some TM M, where $|x| = n$, we first compute $t(n)$ symbols on an extra tape (using the assumption that t is a proper function). This takes time $O(t(n))$.

Next we use our universal Turing machine U (from an earlier lecture) to simulate TM M on input x for $t(n)$ steps. That is, we use the $t(n)$ symbols written on an extra tape to count off exactly $t(n)$ steps of the machine M. Recall that t steps of TM M can be simulated by the universal TM U using at most $c_M \cdot t \log t$ steps, where c_M is a constant dependent on the machine M (its alphabet size, number of tapes, etc.)

Once we finish t steps of simulation of M on x, we stop and accept iff M did not accept x (and otherwise, we stop and reject).

Clearly, the described algorithm will correctly decide the language D. Its running time on an input $x = \langle M \rangle$ is dominated by $c_M \cdot t(n) \log t(n) \leq T(n)$, since $T(n)$ grows faster than any constant multiple of $t(n) \log t(n)$ by assumption. □

□

Similarly, we can prove

Theorem 6 (Space Hierarchy)**.** *For any proper complexity functions $s(n), S(n) \geq \log n$ such that $S(n) \in \omega(s(n))$, we have*

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

The space hierarchy has a smaller gap between small space bound $s(n)$ and large space bound $S(n)$ than the Time Hierarchy because, for space, the universal TM does not suffer any logarithmic-factor slowdown (unlike for time).

Remark 7. *The condition of Hierarchy Theorems that the bounds be proper functions is crucial! If $t(n), T(n)$ are allowed to be arbitrary (not proper), then we can pick some "strange" functions $t(n) \leq T(n)$ such that $\mathsf{Time}(t(n)) = \mathsf{Time}(T(n))$, i.e., in more time we don't get any more problems!*

7 Nondeterminism

Recall that for **DTM (deterministic TM)**, each next step of computation is completely determined by the current configuration of a TM. In contrast, for **NTM (nondeterministic TM)**, there may be several possibilities for a next step.

Formally. NTM's transition function is $\delta : Q \times \Sigma \to 2^{Q \times \Sigma \times \{L,R,-\}}$, i.e., a pair (state,symbol) is mapped to a set {(state,symbol,movement) }.

Thus, DTM's computation is a path; whereas NTM's computation is a tree.

Accept/Reject criteria for NTMs: $x \in L$ iff there exists some accepting computation. Note: if $x \in L$, then there may be some rejecting computations as well.

We define

$$\textbf{Time} = \max_{\text{paths } p} \{\text{length of } p \}$$

and

$$\textbf{Space} = \max_{\text{paths } p} \{\text{number of work-tape cells touched on a path } p \}$$

7.1 Nondeterministic Complexity Classes

Define

$$\mathsf{NTime}(f(n)) = \{L \mid \text{some multi-tape NTM decides } L \text{ in time at most } f(n)\}$$

and

$$\mathsf{NSpace}(f(n)) = \{L \mid \text{some multi-tape NTM decides } L \text{ in space at most } f(n)\}.$$

We also get the following classes:

- $\mathsf{NP} = \cup_k \mathsf{NTime}(n^k)$

- $\mathsf{NEXP} = \cup_k \mathsf{NTime}(2^{n^k})$
- $\mathsf{NL} = \mathsf{NSpace}(\log n)$
- $\mathsf{NPSPACE} = \cup_k \mathsf{NSpace}(n^k)$

For example, the problem of reachability in directed graphs is in NL: Nondeterministically guess a path while keeping only the current vertex in memory. It turns out that the same problem for undirected graphs is in L. Thus, the difference between L and NL is exactly the difference in the complexity of reachability for *undirected* versus *directed* graphs!

Remark 8. *We'll see later that* NPSPACE = PSPACE*, and so there is really no need to use the name* NPSPACE*.*

7.2 Alternative definition of NP

$L \in \mathsf{NP}$ if there is a language $R \in \mathsf{P}$ and a constant c such that

$$L = \{x \mid \exists y, |y| \leq |x|^c, (x, y) \in R\}$$

Lemma 9. *The two given definitions of* NP *are equivalent.*

Proof. Exercise. Hint: y in Definition 2 = accepting path in Definition 1. □

The definition of NP can also be viewed as a game between an all-powerful prover and a deterministic polytime verifier. Given an input x, the (possibly cheating) prover tries to convince the verifier that x is in the language. If x indeed is in the language, the prover can send a short witness string y to the verifier who then can check that $R(x, y)$ is true. If, however, x is not in the language, no message from the prover will convince the verifier to accept (since, for such an x, all short strings y's are such that $R(x, y)$ is false).

Later in the course we'll see how this definition can be generalized by allowing randomized verifiers and allowing multiple rounds of communication between the prover and the verifier.

7.3 Importance of NP

NTMs cannot be efficiently implemented. So they are an abstraction. But, a huge number of real-life problems are in NP because they are of the form: a problem description is such that a solution to the problem is "short" and the solution is "easy" to test for correctness. (So we can nondeterministically guess a solution, and then test its correctness in polytime.)

For example, consider SAT: Given a propositional formula $\phi(x_1, \ldots, x_n)$, decide if ϕ is satisfiable. Here, if ϕ is indeed satisfiable, then there is a "short" proof of that, namely a satisfying assignment to the variables $x_1, \ldots, x_n$ which makes ϕ evaluate to True. Checking if a given assignment satisfies the given formula is "easy": one can evaluate the formula on the given assignment in polytime. Thus, SAT is in NP.

8 Ladner's Theorem

We've seen some examples of problems in NP, and most of these were either in P or NP-complete. Can the world have such a simple structure that every NP-problem is either in P or is NP-complete? The answer is no!

Theorem 10 (Ladner)**.** *If* $\mathsf{P} \neq \mathsf{NP}$*, there there is a language* $L \in \mathsf{NP}$ *such that*

1. $L \notin \mathsf{P}$ *and*
2. L *is not* NP*-complete.*

Proof. **Idea:** Assume $\mathsf{P} \neq \mathsf{NP}$. Then SAT is not P. But we can't use SAT as our language L above, since SAT *is* NP-complete. The idea is to define a variant of SAT, some "padded" version of SAT, so that this variant of SAT is no longer NP-complete, while still not in P either. The amount of padding will depend on a certain "diagonalization"-type argument, which, intuitively, will allow us to kill off all potential polytime reductions from SAT to our padded SAT, ensuring that padded SAT is not NP-complete. We provide the details next.

Padded SAT. For an arbitrary function $H : \mathbb{N} \to \mathbb{N}$, define the following padded SAT:

$$SAT_H = \{\phi 01^{|\phi|^{H(|\phi|)}} \mid \phi \in SAT\}.$$

In other words, we take $\phi \in SAT$ of size n, and place its padded version into SAT_H, with the padding of size $n^{H(n)}$. (For example, if $H(n) = 3$ (a constant function), then SAT_H is SAT with polynomial (cubic) amount of padding. It is easy to see that such SAT_H is polytime-equivalent to SAT: if one is in P then the other one is also in P.)

Our desired language L will be of the form SAT_H for the following function H: For $n \in \mathbb{N}$, define $H(n)$ to be

> the least $i < \log\log n$ such that TM M_i decides SAT_H on all input lengths $m \leq \log n$, in time $O(m^i)$; if no such TM M_i exists, then define $H(n) = \log\log n$.

Note that the definition of $H(n)$ is recursive: to compute $H(n)$, we need to consider SAT_H on input lengths $m \leq \log n$, and so we need to know the values $H(m)$ for these $m \ll n$.

One can write an algorithm for computing $H(n)$ (in the straightforward way), getting the runtime at most $O(n^3)$. (Exercise!) As a consequence of this algorithm for computing $H(n)$, we get that $SAT_H \in \mathsf{NP}$. (Exercise!)

Complexity of SAT_H vs. growth of $H(n)$. Next we will show that, assuming $\mathsf{P} \neq \mathsf{NP}$, we have $SAT_H \notin \mathsf{P}$, and SAT_H is not NP-complete. To this end, we shall use the following claim, relating the complexity of SAT_H and the asymptotic growth rate of $H(n)$.

Claim 11. *$SAT_H \in \mathsf{P} \Leftrightarrow H(n) \in O(1)$. Moreover, $SAT_H \notin \mathsf{P} \Rightarrow \forall c\ \exists n_0\ \forall n \geq n_0 \quad H(n) > c$.*

Proof of Claim 11. ($\Rightarrow$): If $SAT_H \in \mathsf{P}$, then there is some TM M_c, with runtime $O(n^c)$, such that M_c correctly decides SAT_H (on all input lengths). Then, for all $n > 2^{2^c}$, we get $H(n) \leq c$, and hence, $H(n) \in O(1)$.

($\Leftarrow$): Assume there is a constant $c > 0$ such that for all large enough n, $H(n) \leq c$. (That is, $\exists c\ \exists n_0\ \forall n \geq n_0 \quad H(n) \leq c$.) Since for all these large n's, $H(n) \in \{0, 1, 2, \ldots, c\}$, there must exist some $0 \leq d \leq c$ such that $H(n) = d$ for *infinitely many* n's.

Consider TM M_d. If M_d is wrong for SAT_H on some input length m, then for *every* $n \geq 2^m$ we would have $H(n) \neq d$. But this contradicts our earlier conclusion that $H(n) = d$ for infinitely many n's. So it must be the case that M_d correctly decides SAT_H on all input lengths, and so $SAT_H \in \mathsf{P}$ (as M_d runs in time $O(n^d)$).

Finally, for the "moreover" part of the claim, consider the contrapositive:

$$\exists c\ \forall n_0\ \exists n \geq n_0 \quad H(n) \leq c \text{ implies } SAT_H \in \mathsf{P}.$$

The assumption simply means that there are infinitely many n's where $H(n) \leq c$. For these n's, $H(n) \in \{0, 1, 2, \ldots, c\}$, and so, again there is $0 \leq d \leq c$ such that $H(n) = d$ for infinitely many n's. But we already argued above that this leads to the conclusion $SAT_H \in \mathsf{P}$. □

Finishing the proof. Next we show (in the following two claims) that, assuming $\mathsf{NP} \neq \mathsf{P}$, we have $SAT_H \notin \mathsf{P}$ and that SAT_H is not NP-complete.

Claim 12. *If $\mathsf{NP} \neq \mathsf{P}$, then $SAT_H \notin \mathsf{P}$.*

Proof. (by contradiction). Suppose $SAT_H \in \mathsf{P}$. That is, some TM M_c, with polynomial runtime, correctly decides SAT_H for all input lengths. By the proof of Claim 11 (part ($\Rightarrow$)), we get that $H(n) \leq c$, for all large enough n's.

But then SAT_H is just SAT with polynomial amount of padding: for a formula ϕ of size n, $\phi \in SAT$ iff $\phi 01^{n^c} \in SAT_H$. And so, we can use TM M_c to decide SAT:

"On input ϕ of size n, compute the instance $\phi 01^{n^c}$ and run M_c on that instance."

The runtime of the described algorithm for SAT is polynomial in $(n + n^c)$, which is still polynomial in n. □

Claim 13. *If $\mathsf{NP} \neq \mathsf{P}$, then SAT_H is not NP-complete.*

Proof. (by contradiction). Suppose SAT_H is NP-complete. In particular, there is a polytime reduction f from SAT to SAT_H. Let f be computable in time at most n^d, for some constant $d > 0$. Thus, f maps a SAT-instance ϕ of size n to a SAT_H-instance $\psi 01^{m^{H(m)}}$, where $|\psi| = m$, so that

$$\phi \in SAT \Leftrightarrow \psi \in SAT.$$

Moreover, $|f(\phi)| \leq n^d$. As the size of $f(\phi)$ is dominated by $m^{H(m)}$, we get that $m^{H(m)} \leq n^d$, and so

$$m \leq n^{d/H(m)},$$

where m is the size of ψ.

By Claim 12, we know that $SAT_H \notin \mathsf{P}$. Next, by the "moreover" part of Claim 11, for the constant $c := 2d$, we get that $H(n) > 2d$ for all large enough n's (i.e., for all $n \geq n_0$ for some constant n_0). Hence, for all large enough m's, we get $m \leq n^{d/(2d)} \leq n^{1/2}$.

This means that deciding if $\phi \in SAT$ is equivalent to deciding if $\psi \in SAT$, where *the size of ψ is at most square root of the size of ϕ!* So we're led to the following recursive algorithm $\mathcal{A}$ for SAT:

> "On input ϕ of size n, compute $f(\phi) = \psi 01^{m^{H(m)}}$, where $|\psi| = m$. If $m \leq n_0$, solve SAT for ψ by a brute-force algorithm in time $O(2^{n_0})$ (which is constant time). Otherwise, recursively call $\mathcal{A}$ on ψ."

The *correctness* of the algorithm $\mathcal{A}$ is obvious: $\phi \in SAT \Leftrightarrow \psi \in SAT$ because f is a reduction. For the runtime, observe that each recursive call reduces the input size from n to $\sqrt{n}$. So, after at most $\log \log n$ recursive calls, the original formula ϕ of size n gets reduced to some formula ψ of size less than n_0 (and we decide if $\psi \in SAT$ by a constant-time brute-force computation). Thus, the overall runtime is dominated by $\log \log n$ invocations of the reduction f, which is $O(n^d \cdot \log \log n) \leq O(n^{d+1})$.

Thus, we get that $SAT \in \mathsf{P}$. A contradiction. □

This completes the proof of Ladner's theorem. □

8.1 "Natural" problems?

Ladner's example of a problem in NP that is neither in P nor NP-complete is quite artificial (constructed using a diagonalization argument). It may still be possible that the wished-for naive dichotomy is true for all "natural" problems. Some candidate problems are: Graph Isomorphism (GI) and Factoring.

In the GI problem, we are given two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, and need to decide if they are *isomorphic* (i.e., the same up to renaming the nodes). More formally, an isomorphism between G_1 and G_2 is a 1-1 mapping $f : V_1 \to V_2$ such that for every pair of nodes $u, v \in V_1$, we have $(u, v) \in E_1 \Leftrightarrow (f(u), f(v)) \in E_2$.

It is easy to see that GI is NP: simply guess a candidate isomorphism mapping f, and then check that it is indeed an isomorphism. It is unknown if GI is NP-complete. There is some evidence (based on other complexity-theoretic conjectures) that GI is not NP-complete. On the other hand, no (even quantum) polytime algorithm is known for GI.

Similarly, Factoring is not known to be in P, yet it is in quantum polytime. Factoring is unlikely to be NP-complete, as then we would get that all of NP can be solved in quantum polytime, which appears quite unlikely.

9 Schaefer's dichotomy theorem

It's tricky to define what "natural" means. However, one can define classes of constraint-satisfaction problems (CSP) as generalizations of 3-SAT, where one is a given a number of constraints on few (constantly many) variables each, and the task is to check if the constraints are satifiable simultaneously.

We can define classes of CSPs based on the *type* of constraints they are allowed to use. For example, if constraints are linear equations (mod 2), then the resulting CSP is in P (thanks to the method of Gaussian elimination). For another example, if each constraint is a clause with at most one positive literal, the resulting class of CSPs is the class Horn-SAT, which is also known to be in P. If the constraints are arbitrary clauses, then we get SAT, which is NP-complete.

Schaefer proved that for various versions of such CSPs, we indeed have the dichotomy: either in P or NP-complete. More precisely,

Theorem 14 (Schaefer's dichotomy theorem: Simplified)**.** *Any satisfiability problem restricted by the type of clauses in CNF is either in* P *or* NP*-complete.*

Actually, Schaefer's theorem provides a more detailed partitioning for the SAT problems inside P.

Theorem 15 (Schaefer's dichotomy theorem: Full version)**.** *Any satisfiability problem restricted by the type of clauses in CNF is either trivially solvable or complete for one of the following complexity classes under logspace reductions:*

1. $\mathsf{L}(= \mathsf{SL})$*, if the clauses are of the form* $(x \oplus y), (\neg x \oplus y)$*.*
2. NL*, if the clauses are of the form* $(x \vee y), (\neg x \vee y), (\neg x \vee \neg y)$*.*
3. $\oplus\mathsf{L}$*, if the clauses are of the form* $(x \oplus y \oplus z \oplus \dots), (\neg x \oplus y \oplus z \oplus \dots)$*.*
4. P*, if either all clauses are Horn or all clauses are dual Horn (at most one negative literal per clause).*
5. NP*-complete, otherwise.*

The class $\oplus\mathsf{L}$ mentioned above is the class of counting problems of the type: Given a nondeterministic logspace TM M and an input x, decide if the number of accepting computations of M on x is odd. This problem is in P. (Exercise!)