Computational Complexity September 24, 2018

Lecture 2 (Notes)

Lecturer: Ola Svensson *Scribes: Ola Svensson*

Disclaimer: These notes were written for the lecturer only and may contain inconsistent notation, typos, and they do not cite relevant works. They also contain extracts from the two main inspirations of this course:

1. The book *Computational Complexity: A Modern Approach* by Sanjeev Arora and Boaz Barak;
2. The course *http://theory.stanford.edu/ trevisan/cs254-14/index.html* by Luca Trevisan.

1 Introduction

- Recall last lecture TM and Diagonalization (Cantor, Halting, and Time Hierarchy)
- Today Non-determinism and why diagonalization alone will not solve **P** vs **NP** question.

2 Nondeterminism

2.1 Nondeterministic Turing machines

The original definition of **NP** was in using a variant of Turing machines called *nondeterministic* Turing machines (abbreviated NDTM): the only differences between NDTM and TM are

- NDTM has *two* transition functions δ_0 and δ_1, and a special state called q_{accept}

When NDTM computes a function, we envision that at each computational step M makes an arbitrary choice as to which of its two transition functions to apply.

- For every input x, we say that $M(x) = 1$ if there *exists* some sequence of these choices (which we call *nondeterministic choices*) of M) that would make M reach q_{accept} on input x.
- Otherwise, if every sequence of choices makes M halt without reaching q_{accept} then we say $M(x) = 0$.
- We say that M runs in $T(n)$-time if for every input $x \in \{0,1\}^*$ and every sequence of nondeterministic choices, M reaches either the halting state or q_{accept} within $T(|x|)$ steps

Definition 1 *For every function $T : \mathbb{N} \to \mathbb{N}$ and $L \subseteq \{0,1\}^*$ we say that $L \in \mathbf{NTIME}(T(n))$ if there is a constant $c > 0$ and a $c \cdot T(n)$-time NDTM M such that for every $x \in 0,1^*$, $x \in L \Leftrightarrow M(x) = 1$.*

Example 1 *An efficient NDTM to check whether an input n is a composite integer: guess (using nondeterminism) an integer $p : 1 < p < n$, and check whether p divides n.*

Theorem 2 (Nondetermistic time hierarchy) *If f, g are time constructible functions satisfying $f(n+1)\log(f(n+1)) = o(g(n))$, then*

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

More complex proof than deterministic time hierarchy theorem because we do not know how to simulate NDTM efficiently and flip the answer.

2.2 Definition of the class NP

Two definitions of **NP**: (i) as a polytime verifier and (ii) as a nondeterministic polynomial time Turing machine.

Definition 3 *A language $L \subseteq \{0,1\}^*$ is in* **NP** *if there exists a polynomial $p : \mathbb{N} \to \mathbb{N}$ and a polynomial-time TM M (called the verifier for L) such that for every $x \in \{0,1\}^*$,*

$$x \in L \Leftrightarrow \exists u \in \{0,1\}^{p(|x|)} \text{ s.t. } M(x,u) = 1.$$

If $x \in L$ and $u \in \{0,1\}^{p(|x|)}$ satisfy $M(x,u) = 1$, then we call u a certificate/proof *for x.*

- Examples: Graph Coloring, 3-SAT, Hamiltonian Cycle

Theorem 4 $\mathbf{NP} = \cup_{c \geq 1} \mathbf{NTIME}(n^c)$.

Proof

- Main idea: sequences of nondeterministic choices can be viewed as a certificate and vice versa.

Suppose $p : \mathbb{N} \to \mathbb{N}$ is a polynomial and L is decided by a NDTM N that runs in time $p(n)$. For every $x \in L$, there is a sequence of nondeterministic choices and that makes N reach q_{accept} on input x. We can use this sequence as a certificate for x. This certificate has length $p(|x|)$ and can be verified in polynomial time by a *deterministic* TM by simulating N using these choices. Hence $L \in NP$.

Conversely, if $L \in NP$ then we describe a polytime NDTM N that decides L. On input x, it uses the ability to make nondeterministic choices to write down a string u of length $p(|x|)$. Then it runs the deterministic verifier M to verify that u is a valid certificate for x and if so enters q_{accept}. Since $p(n) = O(n^c)$ for some $c > 1$, we conclude that $L \in \mathbf{NTIME}(n^c)$.

■

3 Reducibility and NP-Completeness

- It turns out that the Independent Set problem is at leas as hard as any other problem in NP: if it has a polynomial algorithm then so do all the problems in NP!
- How can we prove that a language C is at least as hard as some other language B? The crucial tool we use is the notion of *reduction.*

Definition 5 (Reductions) *A language $L \subseteq \{0,1\}^*$ is* polynomial-time *Karp reducible to a language $L' \subseteq \{0,1\}^*$ (often shortened to "polynomial-time reducible"), denoted by $L \leq_p L'$, if there is a* polynomial-time computable *function $f : \{0,1\}^* \to \{0,1\}^*$ such that*

$$\text{for every } x \in \{0,1\}^*,\ x \in L \text{ if and only if } f(x) \in L'.$$

Definition 6 (NP-hardness and NP-Completeness) *We say that L' is NP-hard if if $L \leq_p L'$ for every $L \in NP$. We say that L' is NP-complete if L' is NP-hard and $L' \in NP$.*

Examples of NP-complete problems are 3-SAT, Hamiltonicity, Graph Coloring, . . .

3.1 Showing NP-Completeness of Independent Set

We assume that 3SAT is NP-complete (Cook and Levin Theorem that we will show later in course).

Recall the definition of 3SAT and INDSET:

- 3SAT is the language containing all satisfiable 3CNF formulas. A boolean formula is a 3CNF formula if it is the ANDs of ORs of 3 literals:

$$(x \vee y \vee \neg z) \wedge (u \vee \neg v \vee z) \wedge \dots$$

- INDSET is the language containing all pairs $\langle G, k \rangle$ such that the graph G has a subgraph of at least k vertices with no edges between them.

To prove that INDSET is NP-complete we need to prove that

1. $INDSET \in NP$
2. $L \leq_p INDSET$ for all $L \in NP$ (but since $3SAT$ is NP-complete it suffices to show 3SAT $\leq_p$ INDSET; one of the beauties of the concept of completeness).

Lemma 7 $INDSET \in NP$

Proof Consider the following polynomial-time algorithm M: Given a pair $\langle G, k \rangle$ and a string $u \in \{0,1\}^*$, output 1 if and only if u encodes a list of k vertices of G such that there is no edge between any two members of that list. This can clearly be done in polynomial-time. Moreover, by definition, $\langle G, k \rangle$ is in INDSET if and only if there exists a string u such that $M(\langle G, k \rangle, u) = 1$ and hence INDSET is in NP. Note that the length of u is $O(k \log n)$, i.e., polynomial, where n is the number of vertices. ■

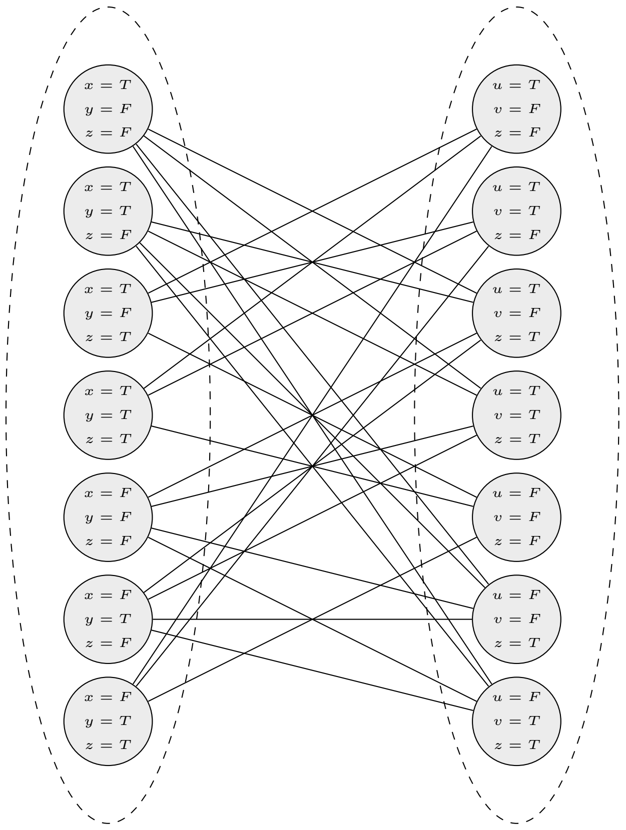
Lemma 8 $3SAT \leq_p INDSET$

Proof We show how to transform in polynomial-time every m-clause 3CNF formula φ into a $7m$-vertex graph G such that φ is satisfiable if and only if G has an independent set of size at least m. The graph G is defined as follows:

- We associate a cluster of 7 vertices in G with each clause of φ.
- The vertices in a cluster associated with a clause C correspond to the seven possible satisfying partial assignments to the three variables on which C depends.
- We put an edge between two vertices of G if they correspond to *inconsistent* partial assignments. (Two partial assignments are consistent if they give the same value to all the variables they share.)

Example of the transformation (each cluster forms a clique and those edges are not depicted for clarity):

$$(x \vee y \vee \neg z) \wedge (u \vee \neg v \vee z) \quad \Rightarrow$$



Clearly transforming φ into G can be done in polynomial time and so all that remains to show is that φ is satisfiable iff G has an independent set of size m:

- Suppose that φ has a satisfying assignment u. Define a set S of cardinality m of G's vertices as follows: for every clause C of φ put in S the vertex in the cluster associated with C (note that such a clause exists since u satisfies all clauses). Because we only choose vertices that correspond to assignment u there are no vertices in S that correspond to inconsistent assignments. Hence S is an independent set of size m.

- Suppose G has an independent set S of size m. We will use S to construct a satisfying assignment u for φ. We define u as follows: for every $i \in [n]$, if there is a vertex $u \in S$ that gives a value a to u_i, then set $u_i = a$; otherwise, set $u_i = 0$. This is well defined because S is an independent set and hence does not contain any inconsistent assignments. Moreover, since S contains exactly one vertex per cluster, our assignment will satisfy each clause.

■

3.2 Some comments

- One of the beauty of NP-completeness is that to prove a problem to be NP-complete we only need to show a reduction from an already known NP-complete problems.

- Since the Cook-Levin Theorem, there has been an explosion of NP-completeness results: 6000 paper/year with NP-completeness in title, keyword, or abstract.

- Almost complete classification of problems in NP into P or NP-complete. Notable exceptions are Graph Isomorphism (recent quasi-poly algorithm), factoring, 3 machine scheduling. One may wonder if all problems in NP are either complete or in P? The answer is NO assuming $P \neq NP$ (Ladner's theorem) and we will prove it today if we have time.
- The P vs NP question is today one of the most important problems in science; a solution is rewarded with eternal fame (and one million dollars :)).
- Today we will show that diagonalization alone will not resolve it.

4 Diagonalization alone is not sufficient to resolve P vs NP

The result covered in this section is originally due to Baker, Gill and Solovay'75.

The diagonalization proofs that we saw use two facts about TM:

1. The existence of an effective representation of Turing machines by strings.
2. The ability of one TM to simulate any other without much overhead in running time or space.

Any argument that only uses these facts is treating machines as *black boxes*: The machine's internal workings do not matter.

We will define on variant of "TM"s for which $P = NP$ and another variant where $P \neq NP$. We conclude that to resolve P vs NP we need to use some other property in addition to the two above.

To define our variants we will use the notion of oracle Turing machines that we define in the next subsection. We then continue to prove the result.

4.1 Oracle Turing Machines

Oracle Turing machines are TMs that are given access to a black box or "oracle" that can magically solve the decision problem for some language $O \subseteq \{0,1\}^*$. (Think of a magic constant time function in your favorite programming language.)

Specifically, an *oracle* Turing machine is a TM that has

- a special *oracle tape* on which it can write a string $q \in \{0,1\}^*$ and in *one step* execute the oracle to get an answer to the query "is q in O?".

This can be repeated arbitrary often with different queries. If O is a difficult language (e.g. Halt) then this oracle gives added power to the TM.

- Oracle NDTMs are defined similarly.

Definition 9 *For every $O \subseteq \{0,1\}^*$, $\mathbf{P}^O$ is the set containing every language that can be decided by a polynomial-time deterministic TM with oracle access to O.*

Similarly $\mathbf{NP}^O$ is the set containing every language that can be decided by a polynomial-time nondeterministic TM with oracle access to O.

Example 2 *Let $\overline{SAT}$ denote the language of* unsatisfiable *formulas. Then $\overline{SAT} \in \mathbf{P}^{SAT}$. (Just query the oracle if input φ is satisfiable and output the opposite answer.)*

Another example is that $\mathbf{P} = \mathbf{P}^O$ for any $O \in \mathbf{P}$. (Any query to the oracle can be replaced by a polynomial time execution. As the number of queries are polynomial, the total running time is still polynomial.)

4.2 An oracle $EXPCOM$ for which $P^{EXPCOM} = NP^{EXPCOM}$

Let $EXPCOM$ be the following language

$$\{\langle M, x, 1^n \rangle : M \text{ outputs 1 on } x \text{ within } 2^n \text{ steps}\}.$$

Lemma 10 $\mathbf{P}^{EXPCOM} = \mathbf{NP}^{EXPCOM} = \mathbf{EXP} := \cup_c \mathbf{DTIME}(2^{n^c})$.

Proof

- An oracle query to EXPCOM allows one to perform an exponential-time computation. So $\mathbf{EXP} \subseteq \mathbf{P}^{EXPCOM}$.
- If M is a poly-time nondeterministic oracle TM, we can simulate its execution with a $EXPCOM$ oracle in exponential time: Such time suffices both to enumerate all of M's nondeterministic choices and to answer the $EXPCOM$ queries.
- It follows that $\mathbf{EXP} \subseteq \mathbf{P}^{EXPCOM} \subseteq \mathbf{NP}^{EXCOM} \subseteq \mathbf{EXP}$.

■

4.3 An oracle B for which $\mathbf{P}^B \neq \mathbf{NP}^B$

For any language B, let U_B be the unary language

$$U_B = \{1^n : \text{some string of length } n \text{ is in } B\}.$$

- For every oracle B, the language U_B is in $\mathbf{NP}^B$: a nondeterministic TM can make a nondeterministic guess for the string $x \in \{0,1\}^n$ such that $x \in B$.
- We now construct an oracle B such that $U_B \notin \mathbf{P}^B$, implying that $\mathbf{P}^B \neq \mathbf{NP}^B$.

For every i, we let M_i denote the oracle TM represented by the binary expansion of i. We construct B in stages where stage i ensures that M_i^B does not decide U_B in time $2^{n/10}$. Each stage determines the status of a finite number of strings (i.e., whether or not these strings will ultimately be in B).

Stage i:

- So far we decided on a finite number of strings (whether or not they are in B).
- Select n large enough so that it exceeds the length of any such string, and run M_i on input 1^n for $2^n/10$ steps.
- Whenever M_i queries the oracle about strings whose status has been determined, we answer consistently.
- When M_i queries the oracle about strings whose status has been undetermined, we declare that the string is *not* in B.
- After M_i finishes computation, we want to make sure that its answer on input 1^n (whatever it was) is incorrect.
- The main point is that we have only decided the fate on at most $2^n/10$ strings in $\{0,1\}^n$, and all of them were decided to not be in B.

- So if M_i accepts 1^n we declare all the remaining strings of length n are also not in B, thus ensuring $1^n \notin U_B$.
- Conversely if M_i rejects, we pick any string $x \in \{0,1\}^n$ that M_i has not queried and add it to B, thus ensuring that $1^n \in U_B$.
- Since every polynomial $p(n)$ is smaller than $2^n/10$ for large enough n, and every TM M is represented by infinitely many strings, our construction ensures that M does not decide U_B if it runs in time $p(n)$.
- Thus we have shown that $U_B \notin \mathbf{P}^B$. In fact we have shown that it is not in $\mathbf{DTIME}^B(f(n))$ for every $f(n) = o(2^n)$.

4.4 Some comments

- It can be shown that for a random oracle O, $\mathbf{P}^O \neq \mathbf{NP}^O$ with high probability.
- A proof technique that holds for all oracle Turing machines are said to relativize.
- An example of a nonrelativizing result is the Cook-Levin Theorem (establishing NP-completeness) that heavily relies on computation being local on a standard TM.