2 P vs. NP and Diagonalization

CS 6810 – Theory of Computing, Fall 2012
Instructor: David Steurer
Scribe: Sin–Shuen Cheung (sc2392)
Date: 08/28/2012

In this lecture, we cover the following topics:

1. 3SAT is **NP**–hard;
2. Time hierarchies;
3. Intermediate problems;
4. Limits of diagonalization.

2.1 3SAT is NP–hard

Definition 2.1 (the class **NP**)**.** $L \subset \{0,1\}^*$ is in the class of **NP** if there exists a polynomial $p : \mathbb{N} \to \mathbb{N}$ and a Turing machine M (called the *verifier* for L) such that for every $x \in \{0,1\}^*$,

$$x \in L \quad \Leftrightarrow \quad \exists y \in \{0,1\}^{p(|x|)} \text{ such that } M(x,y) = 1.$$

If $x \in L$ and $y \in \{0,1\}^{p(|x|)}$ satisfy $M(x,y) = 1$, then we call y a *certificate/witness* for x (with respect to the language L and machine M).

Definition 2.2 (CNF)**.** We call a Boolean formula over variables $u_1, \ldots u_n$ in its *CNF* form (*Conjunctive Normal Form*) if it is an AND of OR's of all variables and their negations. For instance, the following form is a *3CNF* formula:

$$(u_1 \vee \bar{u}_2 \vee u_3) \wedge (u_3 \vee \bar{u}_3 \vee u_4) \wedge (\bar{u}_1 \vee u_3 \vee \bar{u}_4).$$

Definition 2.3 (3SAT)**.** 3SAT is the language of all satisfiable 3CNF formulae; that is

$$\mathsf{3SAT} = \{ \text{3CNF } \phi : \exists \text{ satisfied assignment for } \phi\}.$$

Definition 2.4 (LP)**.**

$$\mathsf{LP} = \{\text{linear programming problem } P : \exists \text{ feasible solution for } P\}$$

Definition 2.5 (**NP**–hard)**.** L is **NP**–hard if every **NP** problem reduces to it by polynomial–time Karp reduction.

Theorem 2.6. 3SAT *is* **NP***–hard.*

Sketch of proof: To prove that 3SAT is **NP**–hard, we follow the outline below

$$\boxed{L_M} \xrightarrow{i)} \boxed{\text{TMSAT}} \xrightarrow{ii)} \boxed{\text{CircuitSAT}} \xrightarrow{iii)} \boxed{\text{3SAT}},$$

where each $\rightarrow$ denotes a polynomial Karp reduction. We use L_M to denote any language L that is verifiable by polynomial–time Turing machine M. The language TMSAT is defined as follows:

Definition 2.7 (TMSAT)**.**

$$\text{TMSAT} = \{(\alpha, x, 1^n, 1^t) | \exists y \in \{0,1\}^n \text{ s.t. } M_\alpha \text{ outputs 1 on input } (x, y) \text{ within } t \text{ steps}\}.,$$

where M_α denotes the deterministic Turing machine represented by string α.

Next, we give the definition for the language CircuitSAT.

Definition 2.8 (CircuitSAT)**.**

$$\text{CircuitSAT} = \{\text{circuit } C \,|\, \exists y \in \{0,1\}^n \text{ s.t. } C(y) = 1\}.$$

Now we proceed to the reductions.

i) From L_M to TMSAT is easy. Given an instance x for L_{M_α}, we can use $(\alpha, x, 1^n, 1^t)$ as the input instance for TMSAT.

ii) The key to this reduction is that any t–time bounded Turing machine M can be simulated by a Boolean circuit C of size roughly t^2. We use Q to denote the set of all possible configurations of M, and let z_i to denote its state at time step i. To check that z_i is correctly performed by M, we only need to know the previous configuration z_{i-1} and the cells that the head currently reads. Since M is a deterministic Turing machine, there can be only one z_i such that the z_i satisfies the correctness constraints with respect to the previous state, the input position and the head location. Therefore we can use a circuit C_M to simulate M. The size of the circuit depends on the total possible states of M.

iii) Finally, the reduction from CircuitSAT to 3SAT follows from the fact that each vertex k of a circuit can correspond to a variable z_k. The relation between vertex k and its parents i, j can be formulated as a 3CNF $\phi_{ijk}(z)$, which satisfies that

$$\phi_{ijk}(z) = 1 \text{ iff } z_k = z_i \wedge z_j.$$

2.2 Time hierarchies

Theorem 2.9 (Time hierarchy theorem)**.** *If* $t(n) \ll T(n)$*, the inclusion* **TIME**$(t) \subset$**TIME**(T) *is strict.*

Proof. By diagonalization. Here we list the Turing machines with time bound t as follows:

	M_1	M_2	M_3	$\cdots$	M_n	$\cdots$
M_1	$M_1(M_1)$	$M_1(M_2)$	$M_1(M_3)$	$\cdots$	$M_1(M_n)$	$\cdots$
M_2	$M_2(M_1)$	$M_2(M_2)$	$M_2(M_3)$	$\cdots$	$M_2(M_n)$	$\cdots$
M_3	$M_3(M_1)$	$M_3(M_2)$	$M_3(M_3)$	$\cdots$	$M_3(M_n)$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
M_n	$M_n(M_1)$	$M_n(M_2)$	$M_n(M_3)$	$\cdots$	$M_n(M_n)$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

Now we construct a language L by imposing the following constraints:

$$L(M_i) = 1 - M_i(M_i) \quad \forall i = 1, \ldots.$$

It is easy to see that none of the listed Turing machines with time bound t can decide this language. On the other hand, we can always use a universal Turing machine to simulate any of the listed Turing machinse with only logarithmic delay. This guarantees that $\mathcal{U}(M_i, M_i)$ can be computed in $\mathbf{TIME}(t \log t)$ and hence we can take the negation of the output to decide L. □

2.3 Intermediate Problems Exist

Many natural **NP** problem turn out to be either in **P** or **NP**–hard (e.g., 3SAT, LP). Could it be that every **NP** problem is either **NP**–hard or in **P**? The following theorem says that there has to exist intermediate problems (unless **P**=**NP**, which means every **NP** problem is in **P** and **NP**–hard).

It would be very interesting to show that a natural problem is intermediate (assuming something about **P** vs **NP**). The classical candidates are Factoring and Graph Isomorphism (both are unlikely to be **NP**–hard, but both could very well be in **P**). Some approximation problems are also candidates for intermediate **NP** problems (e.g., MAXCUT).

Theorem 2.10 (Ladner)**.** *If* $\mathbf{P} = \mathbf{NP}$*, then there exists a language* L *such that* $L \notin \mathbf{P}$*,* $L \in \mathbf{NP}$ *and* $L \notin \mathbf{NP}$*–complete (w.r.t. usual reductions).*

Proof. The proof is based on padding 3SAT instances. (There are also proofs based on a different idea. See references if you are interested.) For a function $h : \mathbb{N} \to \mathbb{N}$, let $\mathsf{3SAT}_h$ be the language $\{(\phi, 1^{h(\phi)}) \mid \phi \in \mathsf{3SAT}\}$, meaning that the satisfiable formulas padded with $h(|\phi|)$ 1's. The goal is to choose h such that $\mathsf{3SAT}_h$ is in **NP** and neither in **P** nor **NP**–hard (unless **P**=**NP**).

Note that the faster h grows (such that we get to spend more time on formulas of the same size) the easier $\mathsf{3SAT}_h$ becomes. So the idea is that we have h growing fast enough such that the problem is not **NP**–hard but slow enough so that it is

not in **P** (unless **P**=**NP**).

Claim 2.11. If h is polynomial–time computable, then $\mathsf{3SAT}_h$ is in **NP**.

Proof. We can use the same *verifier* as for 3SAT except that we have to check that the padding has the right length. We can do that efficiently because h is polynomial–time computable.

Claim 2.12. If [1] $h(n) \in n^{\omega(1)}$, then $\mathsf{3SAT}_h$ is not **NP**–hard (unless **P**=**NP**).

Proof. For the sake of contradiction, suppose $\mathsf{3SAT}_h$ is **NP**–hard and hence f is a polynomial–time Karp reduction from 3SAT to $\mathsf{3SAT}_h$. Then f maps to formula ϕ' . If we iterate this reduction, we can solve 3SAT in polynomial time.

Claim 2.13. If **P**! = **NP**, then there exists h that is polynomial–time computable, grows super–polynomially, but $\mathsf{3SAT}_h$ is not in **P**.

Note that the claims together imply the theorem.

Remark 2.14. The proof of the claim is not hard but it involves a subtle diagonalization argument (see section 3.3 in [AroraB09] or the note http://oldblog.computationalcomplexity.org/media/ladner.pdf)

Let us prove a weaker version of Claim 3, which gives the right intuition.

Claim 2.15. If **P**! = **NP** in a meaningful way, namely there exists polynomial–time computable function t such that t grows super–polynomially and 3SAT requires time $t^{\omega(1)}$, then $\mathsf{3SAT}_t$ is not in **P**.

Proof. For the sake of contradiction, assume $\mathsf{3SAT}_t$ is in **P**. Then, we can solve 3SAT in time $t^{O(1)}$, contradicting our assumption.

Taking Claim 2.11, Claim 2.12 and Claim 2.15 together shows that if **P** differs from **NP** in a meaningful way, then there exists intermediate problems in **NP**.

2.4 Relativization

Suppose we have the access to some oracle $O \subset \{0,1\}^*$, which is assumed to do certain computations in only one time step. Given an oracle O, we can define the counterpart complexity classes $\mathbf{P}^O$ and $\mathbf{NP}^O$ correspondingly. Now the question is *do we have* $\mathbf{P}^O = \mathbf{NP}^O$ *for any oracle* O*?* The answer is *NO!*

We say that a proof about Turing machines *relativizes* if the proof goes through even if we allow the machines access to any fixed oracle O (the oracle

[1] $f(n) \in \omega(g(n))$ means that $f(n)$ dominates $g(n)$ asymptotically, that is $\frac{f(n)}{g(n)} \geqslant k$ for all k.

corresponds to some decision problem).

Diagonalization proofs tend to relativize, e.g., the proof of the time hierarchy theorem also works relative to an oracle.

We want to show that relativizing proofs cannot settle the **P** vs **NP** question. To do so, we will show that there exist oracles A and B such that $\mathbf{P}^A = \mathbf{NP}^A$ and $\mathbf{P}^B != \mathbf{NP}^B$.

Why does this show that relativizing proofs cannot settle **P** vs **NP**? Suppose there is a relativizing proof for $\mathbf{P} != \mathbf{NP}$. Then we get a contradiction to $\mathbf{P}^A = \mathbf{NP}^A$. On the other hand, suppose there is a relativizing proof for $\mathbf{P} = \mathbf{NP}$. The we get a contradiction to $\mathbf{P}^B != \mathbf{NP}^B$.

For A, we can use an oracle that allows us to simulate exponential time computation (e.g., an oracle for an **EXPTIME** complete language). Even for such an oracle, $\mathbf{NP}^A$ machine can be simulated in exponential time. Hence, $\mathbf{NP}^B$=**EXPTIME**=$\mathbf{P}^B$. (Here, **EXPTIME** means $\mathbf{TIME}(2^{\mathrm{poly}(n)})$.)

The choice of B is more subtle. Here is the idea: we'd like to choose B such that for all n, $B \cap \{0,1\}^n$ is a tiny, random, but non–empty subset of all n–bit strings. Such an oracle would be utterly useless for **P**. It can query B only on a polynomial number of n–bit strings. So most likely none of the queries are in B and the polynomial–time machine doesn't learn anything interesting about B. On the other hand, the **NP** machine can guess which n–bit strings are contained in B and then use the oracle to verify it.

Formally, we consider the problem U_B consisting of all unary strings 1^n such that $B \cap \{0,1\}^n$ is non–empty.

Claim 2.16. U_B is in $\mathbf{NP}^B$.

Proof. The verifier $M(1^n, y)$ only needs to check that $|y| = n$ and that y is in B (which it can do using the oracle B).

Claim 2.17. There exists B such that U_B is not in $\mathbf{P}^B$.

Proof. By diagonalization. Let us enumerate all polynomial–time machines $M_1, M_2, ...$ and numbers $n_1, n_2, ...$ with $n_1 = 1$ and $n_i = 2^{n_{i-1}}$ (we just want that the numbers n_i grow very fast). We want to choose B such that for all i. (Then, the problem U_B is not solved by any of the machines M_i, which implies the claim.)

We can use the following algorithm to construct B,

– Initialize B to be 0 everywhere;

- For i from 1 to infinity,
- Invariant: for all n_i-bit strings z (and longer strings) we have $B(z) = 0$
- Run M_i^B on 1^{n_i}. Let S_i be the set of points queried in the oracle. (Note $|S_i| = \text{poly}(n_i)$.)
- If $M_i^B(1^{n_i}) = 0$, find a n_i–bit string z_i outside of S_i and set $B(z_i) = 1$.

Why does this procedure work? You should be able to convince yourself that the invariant holds. Another important point is that setting $B(z_i) = 1$ does not affect previous iterations because n_i is much larger than n_{i-1} and all considered machines are polynomially bounded.