Lecture 2 - **NP**-completeness, **coNP**, $\mathbf{P}_{/\mathbf{poly}}$ and polynomial hierarchy.

Boaz Barak

February 10, 2006

Universal Turing machine A fundamental observation of Turing's, which we now take for granted, is that Turing machines (and in fact any sufficiently rich model of computing) are *programmable*. That is, until his time, people thought of mechanical computing devices that are tailor-made for solving particular mathematical problems. Turing realized that one could have a *general-purpose* machine $M_{\mathcal{U}}$, that can take as input a description of a TM M and x, and output $M(x)$. Actually it's easy to observe that this can be done with relatively small overhead: if M halts on x within T steps, then $M_{\mathcal{U}}$ will halt on M, x within $O(T \log T)$ steps.

Of course today its harder to appreciate the novelty of this observation, as in some sense the 20^{th} century was the century of the general-purpose computer and it is now all around us, and writing programs such as a C interpreter in C is considered a CS-undergrad project.

Enumeration of Turing machines To define properly the universal Turing machine, we need to specify how we describe a Turing machine as a string. Since a TM is described by its collection of rules (of the form on symbol a, move left/right, write b), it can clearly be described as such a string. Thus, we assume that we have a standard encoding of TM's into strings that satisfies the following:

- For every $\alpha \in \{0,1\}^*$, there's an associated TM M_α. Note that in the most natural representations there will be strings that are invalid as a representation of a TM, but we can just map all these strings to, say, the trivial TM that halts immediately and outputs zero.
- For every TM M, there's an infinite number of strings α that encode it. This will probably happen in any natural encoding, but we can enforce it using padding - say that if α encodes M, then so does $\alpha\#0^k$ for every k.[1] This is analogous to the "comments" mechanism (i.e., `//` or `/* .. */`) in programming languages such as C or Java.

Since every string can be also considered as a natural number in binary notation, this concept is often called an *numeration* of Turing machines. Thus, when saying something as the α^{th} Turing machine M_α, we mean the machine represented by the string/number α.

Furthermore, we'll sometimes identify the machine M with its description as a string, and so say things like $M'(M)$ (which means executing the machine M' on the string encoding the machine M).

[1]We'll often use symbols such as # to denote separators etc., pretending for a second that we have an alphabet larger than $\{0,1\}$. Of course, everything can be encoded in $\{0,1\}$. In a similar way, we'll assume that we have some way to encode pairs of strings $\langle y, z\rangle$ as a single string, and when we say $M(x,y)$ we mean the output of M when executed on the encoding of $\langle x, y\rangle$.

NP-completeness Recall our definition of a search problem as a relation π where given x, the goal is to find y such that $(x, y) \in \pi$ (or in other notation, $y \in \pi(x)$). The problem π is in **NP** if given x, y we can check in poly-time whether or not $(x, y) \in \pi$, and it is also in **P** if given x we have a poly-time algorithm to compute $y \in \pi(x)$ if $\pi(x) \neq \emptyset$ or outputs $\perp$ otherwise.

Somewhat informally, an **NP**-*complete* relation is a relation π that is **(1)** in **NP** and **(2)** has the property that if $\pi \in \mathbf{P}$ then $\mathbf{NP} \subseteq \mathbf{P}$ (the condition **(2)** is called being **NP**-*hard*). That is, such a relation is the "hardest in **NP**" in the sense that we can prove that if even one problem in **NP** doesn't have a polynomial-time algorithm then π doesn't have such an algorithm as well. A-priori, it's not at all clear that such relations should exist. However, we'll presently see one example for such a relation:

Lemma 1. *The following relation* TMSAT *is* **NP**-*complete:*

$$\langle M \circ 1^t, y \rangle \in \mathsf{TMSAT} \iff M(y) = 1 \textit{ within } t \textit{ steps}$$

where M is a string describing a Turing machine and $\circ$ encodes concatenation.[2]

Proof. TMSAT is polynomial-time verifiable from the existence of a universal Turing machine.

We now turn to proving that TMSAT is **NP**-hard. Suppose that there exists a poly-time algorithm A for TMSAT. Let π be any **NP**-relation. We'll show a poly-time algorithm B for π. Since π is an **NP**-relation, it is polynomial-time verifiable, and hence there exists a constant c and a Turing machine M_π such that for every x, y , $M(x, y) = 1$ within $|x|^c$ steps iff $\langle x, y \rangle \in \pi$.

For every x, we can define the machine M_x to be the machine with x "hardwired" as the first input. Now, the algorithm B will simply do the following: on input x, it will compute the string $M_x \circ 1^{|x|^c}$ ($\circ$ denotes concatenation) and feed it to A (note that this can be easily done in polynomial, even linear time). If A outputs $\perp$, B will conclude that $\pi(x) =$, if A outputs $y \neq \perp$ then B will output y as a solution for x. $\square$

Karp/Levin reduction Note that the structure of the algorithm B was very simple: given an input x from π it converted it into an input $M_x \circ 1^t$ for TMSAT and fed this input to A. Then, in the case A outputted a non-$\perp$ value, B converted it into a witness for x w.r.t the problem π (in fact, in this case no conversion is necessary). Such a reduction is called a *Karp* (or sometimes *Levin*, in the context of search problems) reduction, and we say that the proof shows that TMSAT is in fact **NP**-hard through a *Karp-reduction*. This is often the case. In fact, in many sources (including the textbook) the definition of **NP**-hardness and **NP**-completeness requires that the reduction will be of this form.

More NP complete problems The problem TMSAT does not seem very natural, so the next question is whether some natural problems are also **NP**-complete. On the way, we'll use a problem that also does not seem very natural, but will be a useful step to get to more natural problems.

Transitivity of reduction The key observation is that once we proved a problem π_0 to be **NP**-hard, we can prove a problem π_1 to be **NP** hard by simply showing a reduction from π_0 to π_1 (rather than showing a reduction from every possible $\pi \in \mathbf{NP}$ to π_1).

[2]We can assume for simplicity this encoding always ends with zero. Alternatively, we can define the relation by $M\#1^t$, where $\#$ encodes a separator. This is the sort of detail we'll gloss over in the future, but please stop me whenever such a point bothers you and you are not absolutely clear how it can be resolved.

Boolean circuits A Boolean circuit is a another model of computation, using gates such as AND,OR and NOT to compute an output from the input.

Boolean circuits - formal definitions:

Definition 1 (Boolean circuit)**.** A Boolean circuit with n inputs and m outputs is a directed acyclic graph (DAG) with labels on the vertices. Each vertex is labeled in one of the following labels: $\{\mathsf{in}_1, \ldots, \mathsf{in}_n, \vee, \wedge, \neg, \mathsf{out}_1, \ldots, \mathsf{out}_m\}$. For every label of the type $\{\mathsf{in}_1, \ldots, \mathsf{in}_n, \mathsf{out}_1, \ldots, \mathsf{out}_m\}$ there is exactly one vertex with this label. The vertices labeled $\mathsf{in}_1, \ldots, \mathsf{in}_n$ must be sources (i.e., have in-degree = 0), the vertices labeled $\mathsf{out}_1, \ldots, \mathsf{out}_m$ must be sinks (i.e., have out-degree = 0). Vertices labeled $\wedge$ or $\vee$ must have in-degree = 2, while vertices labeled $\neg$ must have in-degree = 1.

The *size* of a Boolean circuit is the number of vertices it contains.

If C is a Boolean circuit with n inputs and m outputs, the *function* C computes is a function $f : \{0,1\}^n \to \{0,1\}^m$ defined in the following way:let $x \in \{0,1\}^n$ be some string. For every vertex v in C we define the *value of* v with respect to x to be: **(1)** x_i, if v is labeled in_i **(2)** $a \wedge b$, if v is labeled $\wedge$ and the values of the vertices u, u' with edges into v are a and b respectively. **(3)** $a \vee b$, if v is labeled $\vee$ and the values of the vertices u, u' with edges into v are a and b respectively. **(4)** $\neq a$, if v is labeled $\neg$ and the value of the vertex u with edge into v is a. The function f maps x into $y \in \{0,1\}^m$ where y_j is the value (w.r.t. x) of the vertex labeled out_j in the circuit.

It turns out that a t-step computation of any Turing machine can be simulated using a Boolean circuit of size roughly t^2. This appears in the book, and you will also prove this in the exercises, thus establishing

Lemma 2. *The search problem* CSAT *is defined as follows: let C be a Boolean circuit with n-bit input and one-bit (i.e., binary) output,* CSAT(C) *is the set of inputs $x \in \{0,1\}^n$ such that $C(x) = 1$.* CSAT *is* **NP***-complete.*

3SAT We now define the problem 3SAT, which is in some sense "the mother of all **NP**-complete problems", since it was the first problem to be proven **NP**-complete and was used countless times since then to prove the **NP**-completeness of many natural problems.

Definition 2. A $3CNF$ formula ϕ on n variables is an expression of the following form: $\phi(x_1, \ldots, x_n) = C_1 \wedge C_2 \wedge \cdots \wedge C_m$, where each C_i (called "clause") is an expression of the form $\ell_1 \vee \ell_1 \vee \ell_3$, where each ℓ_j (called "literal") is either a variable x_k or its negation $\neg x_k$.

We evaluate the formula according to the standard rules of logic, and say that $\phi(x_1, \ldots, x_n) = 1$ if it evaluates to "true" on $x_1, \ldots, x_n$ and $\phi(x_1, \ldots, x_n) = 0$ otherwise.

Lemma 3. *Define* 3SAT *to be the following problem: for every $3CNF$ formula ϕ,* 3SAT(ϕ) *is the set of x's such that $\phi(x) = 1$ (*3SAT$(\phi) = \emptyset$ *is ϕ does not have a satisfying assignment). Then* 3SAT *is* **NP** *complete.*

Proof. TBC. □

From 3SAT, we (or more accurately, you in the exercises) can finally prove that Hamiltonian cycle is **NP**-complete:

Lemma 4. *HAM is* **NP***-complete.*

Non-deterministic Turing machine Another way to define **NP** (typically the decision version) is using the concept of a *non-deterministic Turing machine*. The idea is a Turing machine that on every step can make more than one choice of how to proceed on from its current state (without loss of generality, we can assume that each step it makes exactly two choices). Assume that the machine M can output a binary output — either zero or one. Now, if the machine runs for T steps (and hence makes T choices among two options), then it has 2^T possible computation paths. We say that the machine *accepts* the input if it outputs one in at least one of these paths, and *rejects* the input otherwise (if it outputs zero in all these possible paths). The language corresponding to a non-deterministic Turing machine M, denoted by $L(M)$, is the set of all x's that M accepts. We define $\mathbf{NP}''$ to be the class of $L(M)$ for all polynomial-time non-deterministic Turing machines. It's not hard to see that TMSAT is in $\mathbf{NP}''$ (a non-deterministic algorithm for TMSAT is given a Turing machine M and 1^t, make n choices to guess an input $x \in \{0,1\}^n$ and output one iff $M(x) = 1$ within t steps). It's also not hard to see that any language in $\mathbf{NP}''$ can be reduced to TMSAT. Thus, the class $\mathbf{NP}''$ actually coincides with the class **NP** we defined above (for decision problems).

NP as a proof system Recall the definition of **NP** as a class of decision problems:

- $L \in \mathbf{NP}$ if there's a polynomial-time verifiable relation π such that $x \in L \iff \pi(x) \neq \emptyset$.
- Equivalently, $L \in \mathbf{NP}$ if there's a polynomial-time TM M such that $x \in L \iff \exists y$ s.t. $M(x, y) = 1$. (In the non-deterministic TM formulation we can think of y as the non-deterministic choices.)

Intuitively, we can think of decision problems in **NP** as problems that we might not be able to solve, but have short and efficient proofs/certificates. That is, we might not have a polynomial-time algorithm to find a Hamiltonian cycle in a graph, but there's a way for me to *prove* to you that a graph G is Hamiltonian, by presenting the cycle.

The class coNP If **NP** is the class of statements that have short proofs, then **coNP** is the class of statements that have short *refutations*. That is, $L \in \mathbf{coNP}$ if there is a poly-time TM M such that $x \notin L \iff \exists y$ s.t. $M(x, y) = 1$. Equivalently, $L \in \mathbf{coNP}$ if there's a poly-time M such that $x \in L \iff \forall y M(x, y) = 0$. Of course we can flip the output of M to get poly-time M' such that $x \in L \iff \forall y M(x, y) = 1$.

Thus, naturally we have the following **coNP**-complete problem UN3SAT: A $3CNF$ formula ϕ is in UN3SAT iff for all x, $\phi(x) = 0$. That is, there's a Karp reduction from any language in **coNP** to UN3SAT.

More generally, we have that for every $L \in \mathbf{NP}$, $\bar{L} = \{0,1\}^* \setminus L$ is in **coNP** and this characterizes exactly the languages in **coNP**. If L is **NP**-complete, then by the same reduction $\bar{L}$ is **coNP**-complete. A slightly more natural **coNP**-complete problem than UN3SAT is TAUT: $\phi \in$ TAUT iff $\forall x, \phi(x) = 1$.

Note that **coNP** is *not* the complement of **NP**. In particular, there are languages in the intersection of **NP** and **coNP**. In fact all languages in **P** are in this intersection. It is believed (although not known) that $\mathbf{P} \subsetneq \mathbf{NP} \cap \mathbf{coNP}$.

(Insert figure of world view)

Polynomial hierarchy To recap, **NP** is the class of problems reducible to the problem of deciding truth of statements of the form $\exists x \phi(x) = 1$. **coNP** is the class of problems reducible to deciding truth of statements of the form $\forall x \phi(x) = 1$. What about statements of the form $\exists x \forall y \phi(x, y) = 1$, $\forall x \exists y \forall x \phi(x, y, z) = 1$?

We can define classes $\Sigma_1, \Sigma_2, \Sigma_3, \ldots$ and Π_1, Π_2, Π_3 capturing these classes. We define the *polynomial hierarchy* **PH** to be the union of all these classes. What we know is that:

- If $\mathbf{P} = \mathbf{NP}$ then $\Sigma_1 = \Sigma_2 = \cdots = \Pi_1 = \Pi_2 = \cdots = \mathbf{P}$. (See exercise).
- If $\mathbf{NP} = \mathbf{coNP}$ then $\Sigma_1 = \Sigma_2 = \cdots = \Pi_1 = \Pi_2 = \cdots = \mathbf{NP} = \mathbf{coNP}$.
- More generally, if $\Sigma_i = \Pi_i$ then $\Sigma_i = \Sigma_{i+1} = \cdots = \Pi_i = \Pi_{i+1} = \cdots$. In this case we say that the polynomial hierarchy *collapses* to its i^{th} level.
- It is conjectured that the polynomial hierarchy does not collapse. While we seem to have some intuition and evidence for the lower level of the hierarchy (and particularly for the conjecture that $\mathbf{P} \neq \mathbf{NP}$ and $\mathbf{NP} \neq \mathbf{coNP}$), the conjecture that the hierarchy does not collapse at all seems, at least to me, to be mostly for aesthetic reasons. In any case, this assumption has been often used (typically for relatively low levels of the hierarchy), and so far had not been contradicted.

$\mathbf{P}_{/\mathbf{poly}}$, non-uniformity We can use Boolean circuits as a different model of computation: If $f : \{0,1\}^* \to \{0,1\}^*$ is a function, we denote by f_n the restriction of f to $\{0,1\}^n$. We say that $f \in \mathbf{P}_{/\mathbf{poly}}$ if there are constants c, d such that for every n, f_n is computable by a circuit of size cn^d.

Every machine can be simulated by a circuit (in a similar way to the fact that CSAT is **NP**-complete) and so we have

Lemma 5. $\mathbf{P} \subseteq \mathbf{P}_{/\mathbf{poly}}$

Proof. Exercise □

Machines with advice Let $f : \{0,1\}^* \to \{0,1\}^*$ be a function in $\mathbf{P}_{/\mathbf{poly}}$. This means that there is a family of circuits $\{C_n\}_{n \in \mathbb{N}}$ where C_n computes $f_n = f_{\restriction \{0,1\}^n}$. Note that for every n, the circuit C_n can be completely different, and there need not be a uniform rule to obtain all these circuits. This is opposed to computing f by a Turing machine, in which case there's a single machine that computes f on all input length. For this reason Turing machines are often called a *uniform* model of computation, while circuits are called a *non-uniform* model of computation, and can in fact compute more functions. Thus, we have the following result:

Lemma 6. $\mathbf{P} \neq \mathbf{P}_{/\mathbf{poly}}$

We can also capture non-uniformity by Turing machines, if we allow them to take a string, often called *advice* for each input length.

Definition 3 (Computing with advice)**.** Let $f : \{0,1\}^* \to \{0,1\}^*$ be a function, and let $T, a : \mathbb{N} \to \mathbb{N}$ be two functions. We say that $f \in \mathbf{DTIME}(T)_{/a}$ if there exists a Turing machine M, and a sequence of strings $\{\alpha_n\}_{n \in \mathbb{N}}$ with $|\alpha_n| \leq a(n)$ such that for every $x \in \{0,1\}^*$, if $n = |x|$ then $M(x, \alpha_n)$ outputs $f(x)$ within $T(n)$ steps.

I leave it to you to verify that the class $\mathbf{P}_{/\mathbf{poly}}$ is equal to the union of $\mathbf{DTIME}(n^c)_{/n^d}$ for all constants $c, d > 0$.

Note: As before, many texts define $\mathbf{P}_{/\mathbf{poly}}$ as a class containing only decision problems (i.e., functions with one-bit output).

$\mathbf{NP} \subseteq \mathbf{P}_{/\mathbf{poly}}$ implies PH collapses If $\mathbf{P} = \mathbf{NP}$ then we know that polynomial hierarchy collapses. If $\mathbf{NP} \subseteq \mathbf{P}_{/\mathbf{poly}}$ then this does not mean that $\mathbf{P} = \mathbf{NP}$ since it may be hard to find that circuit. However, finding this circuit can be posed as a Σ_2-search problem, and once we have it, we can solve Π_2-problems in Σ_2.

Practical and philosophical implications of $\mathbf{P} = \mathbf{NP}$ etc.. As far as we know, it may be the case that $\mathbf{P} = \mathbf{NP}$ and that there's even a truly efficient algorithm for 3SAT (say linear or quadratic, with a reasonable constant). This will immediately mean that we can solve all search problems essentially in the same time we can verify a solution. Such search problems include not only the Hamiltonian cycle problem and seating people in dinner parties, but a great many problems arising in scheduling/operation research, computational biology, economics and game theory, physics, and many other areas of human endeavors (there are thousands of **NP** and **NP**-complete problems in all kinds of applications[3]). Another well known application is that such an algorithm will enable breaking not just all currently used encryption and signature schemes, but all possible encryption scheme.

However, because such an algorithm will imply that actually $\mathbf{P} = \mathbf{PH}$, such an algorithm will have even more dramatic consequences.[4] For example, many questions arising in science and other applications, have the following form: given a set of observations, find the simplest theory/equation/rules that explain these observations. This may sound simple, but even in a relatively "clean" science such as physics, where eventually the theories are described by few simple equations, it took centuries to find the right ones (and we're still not done). When trying to understand DNA, develop drugs, or the human brain, the space we search in is much larger, and hence, unsurprisingly, we still have a lot more to go. However this search problem is either in NP or Σ_2 (depending on what whether checking for a good theory means takes polynomial-time or takes enumerating over an exponential space) we'll now be able to solve it automatically. It means that it is likely that we'll be able to use such an algorithm not only to resolve unsolved mathematical questions, find the unified theory of physics, and cure cancer, but also find the software that does just as well as the human brain in various tasks (visual perception, language recognition). Since as our hardware keeps improving, it seems the main barrier for such simulations is software, we may get to the point where computer can actually simulate a human brain (or maybe many brains). At that point a computer may be able to not just recognize English but also write novels and symphonies...

Of course, it's very likely that $\mathbf{NP} \neq \mathbf{P}$ and hence we won't have this science fiction fantasy world.[5] However, the fact that we cannot rule out such ridiculous possibilities is outrageous. This demonstrates how deep is our ignorance about computation. I personally believe that,

[3]In a 1997 survey paper, Papadimitriou notes that in a search he found 6000 papers each year with the term "NP-complete" in the title, abstract or keywords. The term was more popular than terms such as "compiler" , "operating system", "database", "neural network" and "expert".

[4]Note that the proof that $\mathbf{P} = \mathbf{PH}$ will turn a quadratic algorithm for 3SAT into a roughly n^{2^c} algorithm for Σ_c–3SAT. However, most of the applications below require relatively low levels of the hierarchy, and also note that as far as we know at the moment, there may be a linear or quadratic algorithm for all of **PSPACE**.

[5]Although we will fulfill some fantasies: if we do manage to prove good lower bounds on computation of problems in **NP**, we'll have provably unbreakable cryptography, and universal simulation of all randomized algorithms.

just as it was with Geometry in the 19^{th} century and physics in the 20^{th} century, as we resolve this shame and learn more about computation, there will be unexpected and far reaching consequences.