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

One of the striking aspects of **NP**-completeness is that a surprisingly large number of **NP** problems —including some that were studied for many decades— turned out to be **NP**-complete. This phenomenon suggests a bold conjecture: every problem in **NP** is either in **P** or **NP** complete. If $\mathbf{P} = \mathbf{NP}$ then the conjecture is trivially true but uninteresting. In this section we show that if (as widely believed) $\mathbf{P} \neq \mathbf{NP}$, then this conjecture is false— there is a language $L \in \mathbf{NP} \setminus \mathbf{P}$ that is not **NP**-complete. An interesting feature of the proof is an interesting and Gödelian definition of a language SAT_H which "encodes" the difficulty of solving itself.

Theorem 3.3 *("**NP** intermediate" languages* [Lad75]*)*
*Suppose that $\mathbf{P} \neq \mathbf{NP}$. Then there exists a language $L \in \mathbf{NP} \setminus \mathbf{P}$ that is not **NP**-complete.*

PROOF: For every function $H : \mathbb{N} \to \mathbb{N}$, we define the language SAT_H to contain all length-n satisfiable formulae that are padded with $n^{H(n)}$ 1's; that is, $\mathsf{SAT}_H = \left\{\psi 01^{n^{H(n)}} : \psi \in \mathsf{SAT} \text{ and } n = |\psi|\right\}$. We now define a function $H : \mathbb{N} \to \mathbb{N}$ as follows:

> $H(n)$ is the smallest number $i < \log\log n$ such that for every $x \in \{0,1\}^*$ with $|x| \leq \log n$, M_i outputs $\mathsf{SAT}_H(x)$ within $i|x|^i$ steps.[1] If there is no such number i then $H(n) = \log\log n$.

H is well-defined since $H(n)$ determines membership in SAT_H of strings whose length is greater than n, and the definition of $H(n)$ only relies upon checking the status of strings of length at most $\log n$. In fact, the definition of H directly implies an $O(n^3)$-time recursive algorithm that computes $H(n)$ from n (see Exercise 3.6).[2] We defined H in this way to ensure the following claim:

CLAIM: $\mathsf{SAT}_H \in \mathbf{P}$ iff $H(n) = O(1)$ (i.e., there's some C such that $H(n) \leq C$ for every n). Moreover, if $\mathsf{SAT}_H \notin \mathbf{P}$ then $H(n)$ tends to infinity with n.

> PROOF OF CLAIM:
>
> **($\mathsf{SAT}_H \in \mathbf{P} \Rightarrow H(n) = O(1)$):** Suppose there is a machine M solving SAT_H in at most cn^c steps. Since M is represented by infinitely many strings, there is a number $i > c$ such that $M = M_i$. The definition of $H(n)$ implies that for $n > 2^{2^i}$, $H(n) \leq i$. Thus $H(n) = O(1)$.
>
> **($H(n) = O(1) \Rightarrow \mathsf{SAT}_H \in \mathbf{P}$):** If $H(n) = O(1)$ then H can take only one of finitely many values, and hence there exists an i such that $H(n) = i$ for infinitely many n's. But this implies that the TM M_i solves SAT_H in in^i-time: for otherwise, if there was an input x on which M_i fails to output the right answer within this bound, then for every $n > 2^{|x|}$ we would have $H(n) \neq i$. Note that this holds even if we only assumed that there's some constant C such that $H(n) \leq C$ for infinitely many n's, hence proving the "moreover" part of the claim.

Using this claim we can show that if $\mathbf{P} \neq \mathbf{NP}$ then SAT_H is neither in **P** nor **NP** complete:

- Suppose that $\mathsf{SAT}_H \in \mathbf{P}$. Then by the claim, $H(n) \leq C$ for some constant C, implying that SAT_H is simply SAT padded with at most a polynomial (namely, n^C) number of 1's. But then a polynomial-time algorithm for SAT_H can be used to solve SAT in polynomial time, implying that $\mathbf{P} = \mathbf{NP}$!

[1] Recall that M_i is the machine represented by the binary expansion of i, and $\mathsf{SAT}_H(x)$ is equal to 1 iff $x \in \mathsf{SAT}_H$.

[2] "Recursive algorithm" is a term borrowed from standard programming practice, where one calls a program "recursive" if it has the ability to call itself on some other input.

- Suppose that SAT_H is **NP**-complete. This means that there is a reduction f from SAT to SAT_H that runs in time $O(n^i)$ for some constant i. Since we already concluded SAT_H is not in **P**, the claim above implies that $H(n)$ tends to infinity. Since the reduction works in $O(n^i)$ time only, for large enough n it must map SAT instances of size n to SAT_H instances of size smaller than $n^{H(n)}$. Thus for large enough formulae φ, the reduction f must map it to a string of the type $\psi 01^{H(|\psi|)}$ where ψ is smaller by some fixed polynomial factor, say, smaller than $\sqrt[3]{n}$. But the existence of such a reduction yields a simple polynomial-time recursive algorithm for SAT, contradicting the assumption $\mathbf{P} \neq \mathbf{NP}$! (Completing the details is left as Exercise 3.6.)

■

Though the theorem shows the existence of some non **NP**-complete language in $\mathbf{NP} \setminus \mathbf{P}$ if $\mathbf{NP} \neq \mathbf{P}$, this language seems somewhat contrived, and the proof has not been strengthened to yield a more natural language. In fact, there are remarkably few candidates for such languages, since the status of most natural languages has been resolved thanks to clever algorithms or reductions. Two interesting exceptions are the *Factoring* and *Graph isomorphism* languages (see Example 2.3). No polynomial-time algorithm is currently known for these languages, and there is strong evidence that they are not **NP** complete (see Chapter 8).

3.4 Oracle machines and the limits of diagonalization

Quantifying the limits of diagonalization is not easy. Certainly, the diagonalization in Sections 3.2 and 3.3 seems more clever than the one in Section 3.1 or the one that proves the undecidability of the halting problem in Section 1.5.

For concreteness, let us say that "diagonalization" is any technique that relies solely upon the following properties of Turing machines:

I The existence of an effective representation of Turing machines by strings.

II The ability of one TM to simulate any another 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 now show a general way to define variants of Turing Machines called *oracle Turing Machines* that still satisfy the above two properties. However, one way of defining the variant results in TMs for which $\mathbf{P} = \mathbf{NP}$, whereas another way results in TMs for which $\mathbf{P} \neq \mathbf{NP}$. We conclude that to resolve **P** versus **NP** we need to use some other property in addition to **I** and **II**.

Oracle machines are TMs that are given access to a black box or "oracle" that can magically solve the decision problem for some language $O \subseteq \{0,1\}^*$. The machine has a special *oracle tape* on which it can write a string $q \in \{0,1\}^*$ and in one step gets an answer to a query of the form "Is q in O?". This can be repeated arbitrarily often with different queries. If O is a difficult language (say, which cannot be decided in polynomial time, or even undecidable) then this oracle gives an added power to the TM.

Definition 3.4 *(Oracle Turing Machines)* An *oracle* Turing machine is a TM M that has a special read/write tape we call M's *oracle tape* and three special states $q_{\mathsf{query}}, q_{\mathsf{yes}}, q_{\mathsf{no}}$. To execute M, we specify in addition to the input a language $O \subseteq \{0,1\}^*$ that is used as the *oracle* for M. Whenever during the execution M enters the state q_{query}, the machine moves into the state q_{yes} if $q \in O$ and q_{no} if $q \notin O$, where q denotes the contents of the special oracle tape. Note that, regardless of the choice of O, a membership query to O counts only as a single computational step. If M is an oracle machine, $O \subseteq \{0,1\}^*$ a language, and $x \in \{0,1\}^*$, then we denote the output of M on input x and with oracle O by $M^O(x)$.

Nondeterministic oracle TMs are defined similarly. ♢