16. P vs. NP, Diagonalization and its Limits

Goal: Show two results

(1) NP-intermediate languages exist, provided $P \neq NP$.

(2) P vs. NP cannot be proved with diagonalization.

16.1 Ladner's Theorem

Observation: A large number of NP-problems turned out to be NP-complete (polynomial-time reductions).

Conjecture: Every problem in NP is either in P or NP-hard (and hence complete)

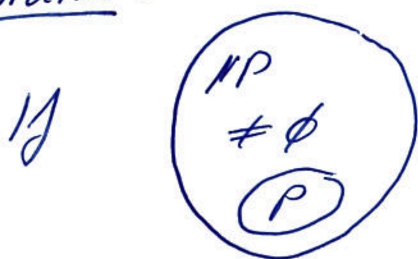
Answer:
- If $NP = P$, the conjecture trivially holds.
- If $NP \neq P$, the conjecture is false.

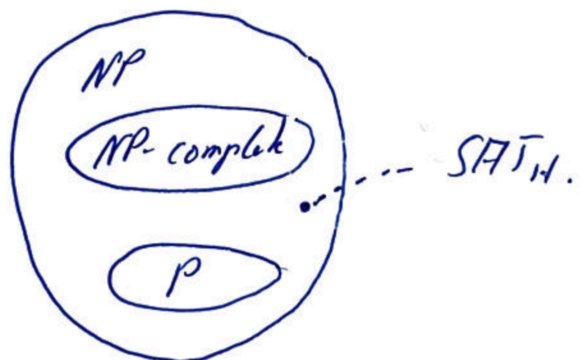
Trick: Language that encodes the difficulty of solving itself.

Theorem (Ladner '75):

Suppose that $P \neq NP$.
Then there is a language $L \in NP \setminus P$ that is not NP-complete.

Illustration:

If 

then 

For every function $h: \mathbb{N} \to \mathbb{N}$,
we define

$$SAT_h := \{ \varphi 0 1^{n^{h(n)}} \mid \varphi \in SAT \text{ and } n = |\varphi| \}.$$

All satisfiable formulas φ padded with 1s ($|\varphi|^{h(|\varphi|)}$ many).

- Define a particular such function $H: \mathbb{N} \to \mathbb{N}$ by

$H(n) :=$ smallest $i < \log \log n$ so that
for every $x \in \{0,1\}^*$ with $|x| \leq \log n$,
M_i computes $SAT_H(x)$ in $i \cdot |x|^i$-steps.

(M_i: Turing machine encoded by the binary representation of i. $SAT_H(x)$: 0 or 1)

Use $H(n) := \log \log n$ if no such i exists.

Note that the definition is recursive:
SAT_H references H and
H references SAT_H.

Still, function H is well defined:
↳ $H(n)$ determines membership in SAT_H for strings of length $> n$ (to be precise, $n + 1 + n^{H(n)}$).
↳ The definition of $H(n)$ only relies upon checking the status of strings of length $\leq \log n$.

<u>Lemma</u>: H is computable in polynomial time.

Lemma:

$SAT_H \in P$ iff $\exists c \in \mathbb{N} : H(n) \leq c$ for all $n \in \mathbb{N}$

$(H(n) = O(1))$.

Moreover, if $SAT_H \notin P$, then $\lim_{n \to \infty} H(n) = \infty$

$(\forall c \in \mathbb{N} \; \exists n_0 \in \mathbb{N} \; \forall n \geq n_0 : H(n) > c)$.

Proof:

„=>" Suppose machine M solves SAT_H in at most $c \cdot n^c$ steps.

Machine M is represented by infinitely many strings

(we can assume an encoding scheme that appends 1s).

Hence, there is a number $i > c$ with $M = M_i$.

The definition of H implies that for $n > 2^{2^i}$, we have

$$H(n) \leq i.$$

Thus, $H(n) = O(1)$.

„<=" If $H(n) = O(1)$, then H can take only finitely many values.

Hence, there is an i so that

$H(n) = i$ for infinitely many n.

But then M_i solves SAT_H in $i \cdot n^i$-time

↳ If there was an input x on which M_i fails to output the right answer within the time bound,

Then for every $n > 2^{|x|}$,
we have $H(n) \neq c$.

↯ $H(n) = c$ for infinitely many n.

Note that it is sufficient to assume the existence of a c so that

$H(n) \leq c$ for infinitely many n's.

Hence the moreover (by contraposition). □

<u>Proof (of Ladner's Theorem)</u>:

We show that, assuming $P \neq NP$,
then SAT_H is neither in P nor NP-complete.

- Assume $SAT_H \in P$.
  Then, by the lemma above, $H(n) \leq c$ for some constant c (and all $n \in \mathbb{N}$).
  This means SAT_H is SAT,
  padded with at most a polynomial (namely n^c)
  number of 1s.
  But then the algorithm for SAT_H can be used
  to solve SAT in polynomial time,
  implying $P = NP$. ↯

- Assume SAT_H is NP-complete.
  Then there is a reduction of SAT to SAT_H
  that runs in $O(n^i)$-time for some constant i.

We already showed that $SAT_H \notin P$.

Hence, by the lemma above, $\lim_{n \to \infty} H(n) = \infty$.

Since the reduction only needs $O(n^i)$-time,

it eventually has to map

SAT instances of size n

to SAT_H instances of size smaller than $n^{H(n)}$.

As a consequence, for large enough φ,

the reduction must map it to a string $\psi 0 1^{|\psi|^{H(|\psi|)}}$

where ψ is smaller than φ by some fixed polynomial factor, say $\sqrt{n}$.

The existence of such a reduction

implies a polynomial-time recursive algorithm for SAT.

Hence, $P = NP$. ↯ □

16.2 Limits of Diagonalization

Goal: Give oracles A and B for which

$$P^A \neq NP^A \quad \text{and} \quad P^B = NP^B.$$

Consequence: We are unlikely to show P vs. NP via diagonalization.

Why? The diagonalization method simulates one TM by another one.

Moreover, the simulator can determine the behavior of the simulated machine and behave differently.

- Suppose simulator and simulated machine both were given identical oracles.
  Whenever the simulated machine queries the oracle, so does the simulator.
  Therefore, the simulation works as before.

> Any theorem proved about TMs using only diagonalization would still hold if both machines were given the same oracle.

- In particular, if we could prove $P \neq NP$ using diagonalization we would conclude $P^C \neq NP^C$ for every oracle C.
  But $P^B = NP^B$, so the conclusion is false.
  Hence, diagonalization is not sufficient to separate the classes.
- Similarly, no proof that only relies on simulation can show that $P = NP$.
  Otherwise, this would carry over to oracles, contradicting $P^A \neq NP^A$.

<u>Consequence</u>: To solve P vs. NP, we must <u>analyze</u> computations, not just simulate them
An analysis approach via circuits will be presented next week.

Theorem:

(1) There is an oracle A with $P^A \neq NP^A$.

(2) There is an oracle B with $P^B = NP^B$.

Idea:

- For B, take any PSPACE-complete problem, say QBF.
- For A, we give an explicit construction.
  We design A so that a certain language $L_A \in NP^A$ provably requires brute-force search, so $L_A \notin P^A$.
  The trick is to consider every polynomial-time oracle machine and ensure that it fails to decide L_A.

Proof:

(2) We have

$$NP^{QBF} \subseteq NPSPACE \overset{\text{(Savitch)}}{=} PSPACE \subseteq P^{QBF}.$$

- For the first inclusion, we convert a non-deterministic polynomial-time oracle TM into a non-deterministic polynomial-space TM. Rather than querying the oracle, the new machine checks the QBF-formulas explicitly.
- The second inclusion holds because QBF is PSPACE-complete.

(1) For any oracle C, let

$$L_C := \{ w \mid \exists x \in C : |x| = |w| \}$$

All strings for which a string of equal length appears in C.

Lemma: For any C, $L_C \in NP^C$.

We now construct an oracle A so that $L_A \notin P^A$.

Let $M_1, M_2, \ldots$ be a list of all polynomial-time oracle TMs.

We assume for simplicity that M_i runs in time n^i.

The construction proceeds in stages:

Stage i constructs a part of A so that M_i^A does not decide L_A.

We construct A by declaring certain strings to belong to A and other strings to be outside A.

Each stage determines the status of a finite number of strings.

Initially, there are no constraints on A.

We begin with stage 1.

Stage i:

So far, a finite number of strings have been declared to be in or out of A.

We choose n to be

- greater than the length of any such string and
- large enough so that $2^n > n^i$, where n^i is the running time of M_i.

We extend A so that

M_i^A accepts 1^n iff $1^n \notin L_A$.

Hence, $L_A \neq L(M_i^A)$.

- We run M_i on 1^n and respond to its oracle queries as follows:
  - ↳ If M_i queries a string y whose status has already been determined, we answer consistently.
  - ↳ If y's status is undetermined, we respond "no" and require y to be outside of A.

  We continue the simulation of M_i until it halts.

- Consider the situation from M_i's perspective. The input is 1^n and M_i should compute L_A. Hence, if M_i finds a string of length n in A, it should accept because it knows $1^n \in L_A$. If M_i determines that all strings of length n are not in A, it should reject, because it knows that $1^n \notin L_A$.
- The point is that M_i does not have enough time to ask about all strings of length n, and we have answered "no" to all queries. Hence, when M_i halts and has to decide whether to accept or to reject, it does not have enough information to be sure to be correct.

- We have to make sure that M_i's decision is <u>not</u> <u>correct</u>.
  We do so by observing M_i's decision
  and extending A so that it is not correct.
  - ↳ If M_i accepts 1^n,
    we declare all strings of length n to be out of A,
    hence $1^n \notin L_A$.
  - ↳ If M_i rejects 1^n,
    we find a string of length n
    that M_i has <u>not queried</u>.
    We declare this string to belong to A.
    Hence, $1^n \in L_A$.
    Such a string exists because M_i runs in n^i
    and $n^i < 2^n$,
    the number of strings of length n.
- We finish stage i by declaring
  that any string of length $\leq n$
  whose status remains undetermined
  is outside of A.

Altogether, we have shown that <u>no</u> polynomial-time oracle TM with oracle A decides L_A. □