Computational Complexity Lecture Notes

Lecture 3

The structure of NP

Draft version 0.4

3.1 Existence of NP-Intermediate Problems

After years of efforts, there are problems in **NP** without a polynomial-time algorithm or a completeness proof. Famous examples are FACTORING$_D$ (the problem of *deciding* if a given number has a factor $\leq k$), GI (Graph Isomorphism) etc. So, are there **NP** problems that are neither in **P** nor **NP**-complete? And what does that mean? The question goes even deeper:

The $\leq_T^p$*-degree* of a language A consists of all languages L such that $L \equiv_T^p A$ (that is, $L \leq_T^p A \wedge A \leq_T^p L$). So, $\leq_T^p$-degree is an equivalence relation, and hence it partitions **NP** to equivalence classes. How such a world would look like? There are three possibilities:

- **P** $=$ **NP**, thus all languages in **NP** are $\leq_T^p$-complete for **NP**, so **NP** contains *exactly one* $\leq_T^p$-degree.
- **P** $\neq$ **NP**, and **NP** contains *two different* degrees: **P** and **NP**-complete languages.
- **P** $\neq$ **NP**, and **NP** contains more degrees, so there exists a language in **NP** $\setminus$ **P** that is not **NP**-complete.

Are all these plausible views of the computational world? We will show *that the second case cannot happen*!

Theorem 3.1 (Ladner)

If **P** $\neq$ **NP**, there exists a language in **NP**, which is neither in **P** nor **NP**-complete.

Proof. (*Blowing holes in* SAT)
The idea is that we will construct a language A by taking an **NP**-complete language, and "blow holes" to it, so that it is no longer **NP**-complete, neither in **P**.

Let $\{M_i\}$ an enumeration of all polynomial-time *clocked* TMs, and $\{F_i\}$ an enumeration of all polynomial-time *clocked* functions. We define A as follows:

$$A = \{x \mid x \in \text{SAT} \wedge f(|x|) \text{ is even}\}$$

Note that if $f \in$ **FP**, then $A \in$ **NP**: just guess a truth assignment, compute $f(|x|)$ and verify.

The key of the proof is f, which, once again, will be defined in stages and assure tha diagonalization. We will define a a polynomial-time TM M_f computing f. Let also M_{SAT} be the machine that decides SAT (by assumption is not an polynomial-time machine), and $f(0) = f(1) = 2$.

On input 1^n, for n steps, M_f starts computing $f(0), f(1), \ldots$ iteratively, until it runs out of time. Let $f(x) = k$ the last value of f it was able to compute.

- If $k = 2i$:
  M_f tries to find a $z \in \{0,1\}^*$ such that $M_i(z)$ outputs the *wrong* answer to "$z \in A$" question, i.e. $M_i(z) \neq A(z)$:
  In order to do that, M_f simulates for n steps $M_i(z)$ and the machines involved in the definition of A: $M_{\text{SAT}}(z)$ and $f(|z|)$, for all z in lexicographic order, until it reaches the time limit. If such a string is found in the allotted time, output $k+1$, else output k.

- If $k = 2i + 1$:
  M_f tries to find a string z such that $F_i(z)$ is an *incorrect* Karp reduction from SAT to A, i.e. $M_{\text{SAT}}(z) \neq A(F_i(z))$:
  So, M_f simulates for n steps $F_i(z), M_{\text{SAT}}(z), M_{\text{SAT}}(F_i(z)), f(|F_i(z)|)$ for all z in lexicographic order, until it reaches the time limit. If such a string is found in the allotted time, output $k+1$, else output k.

It is clear by its definition that M_f runs in polynomial time, and $f(n+1) \geq f(n)$.

We claim that $A \notin$ P: Suppose, for the sake of contradiction, that $A \in$ **P**. This implies that there is an i such that $L(M_i) = A$. Then, in the case ($k = 2i$) above, the simulation will never find a z satisfying the desired property. The machine will be "stuck" to $f(n) = 2i$ for all $n \geq n_0$, for some n_0. Thus, $f(n)$ is even for all but finitely many n, and by its definition, A coincides with SAT on all but finitely many input sizes (*almost everywhere*). Then SAT $\in$ **P**, which contradicts our assumption that **P** $\neq$ **NP**!

We claim that A is not NP-complete: Suppose now that A is **NP**-complete. This means that there is a reduction F_i from SAT to A. Then, the case ($k = 2i+1$) will never find a z satisfying the desired property, and again, $f(n) = 2i+1$ on all but finitely many input sizes. Then A is a finite language, hence in **P**, contradiction! □

Remark 3.1

The above proof method is called *delayed diagonalization*, because in order to keep f in polynomial time, we'll have to "wait" for the diagonalization to occur. Notice that f was not increased until a stage of the diagonalization process was completed.

- Note that we proved Ladner's theorem for Karp reductions. You can adjust the proof for Cook reductions as well (*how?*).

Using the same technique, we can prove an analog of *Post's problem* in Recursion Theory:

Theorem 3.2

If **P** $\neq$ **NP**, there exist $A, B \in$ **NP** such that $A \not\leq_T^p B$ and $B \not\leq_T^p A$.

Ladner's Theorem (*generalized by Schöning*) implies also that:

Corollary 3.1

If **P** $\neq$ **NP**, then for every language $B \in$ **NP** $\setminus$ **P**, there exists a set $A \in$ **NP** $\setminus$ **P** such that $A \leq_T^p B$ and $B \not\leq_T^p A$.

Remark 3.2

So, if **P** $\neq$ **NP**, then **NP** contains *infinitely many* distinct $\leq_T^p$-degrees.

3.2 Polynomial-Time Isomorphism

As you know, all **NP**-complete problems are connected through reductions. These reductions are relating the problems based only on their computational difficulty. We would like to have some relation between languages that reflects structural similarities, ideally some sort of *isomorphism*, that would indicate that these two languages are essentially the same.[1]

Definition 3.1 (Polynomial-time isomorphism)

Two languages $A, B \subseteq \Sigma^*$ are *polynomial-time isomorphic* if there exists a function $h : \Sigma^* \to \Sigma^*$ such that:

1. h is a bijection.
2. For all $x \in \Sigma^*$: $x \in A \Leftrightarrow h(x) \in B$.
3. Both h and h^{-1} are polynomial-time computable.

Functions h and h^{-1} are then called *polynomial-time isomorphisms*.

▶ Which reductions are polynomial-time isomorphisms? Note that in the usual case of Karp reductions, e.g. $A \leq_m^p B$, we map arbitrary instances of A to very specific instances of B, so Karp reductions are usually not surjections.

[1] In general, an *isomorphism* is a bijective mapping that *preserves the structure* between two sets, spaces etc.

Definition 3.2 (Padding function)

Let $L \subseteq \Sigma^*$ be a language. We say that function $pad : \Sigma^* \times \Sigma^* \to \Sigma^*$ is a *padding function* for L if it has the following properties:

1. It is computable in polynomial time.
2. For all $x, y \in \Sigma^*$, $pad(x, y) \in L \Leftrightarrow x \in L$.
3. There is a polynomial time algorithm, which, given $pad(x, y)$ recovers y.

- Languages that have padding functions are called *paddable*.
- In addition, if pad satisfies $|pad(x, y)| > |x| + |y|$ for all $x, y \in \Sigma^*$, is called *length-increasingly paddable*.
- Function pad is essentially a reduction from L to itself that "encodes" another string y into the instance of L.

We can easily find padding functions for well-known **NP**-complete problems.

Example 3.1 (SAT)**.** *Let x an instance with n variables and m clauses, and $y \in \{0, 1\}^*$. Define $pad(x, y)$ is an instance of SAT containing all clauses of x, plus $m + |y|$ more clauses, and $|y| + 1$ more variables. We will encode y's characters encoded as clauses with one literal, after the original m clauses of x. In order to avoid decoding ambiguities (where does x end and y's encoding starts), we will add a "delimiter" of m repeated appearences of the same clause. For example:*

$$\underbrace{(x_1 \wedge \cdots) \wedge \cdots \wedge (\cdots)}_{x\text{'s } m \text{ clauses}} \wedge \underbrace{(x_{n+1}) \wedge \cdots \wedge (x_{n+1})}_{m \text{ copies of } (x_{n+1}) \text{ clause}} \wedge \underbrace{(x_{n+2}) \wedge (\neg x_{n+3}) \wedge (x_{n+4}) \wedge (x_{n+5})}_{y=1011}$$

- *The first m clauses are x's original clauses.*
- *We have m additional clauses, copies of x_{n+1} clause.*
- *The last $m + i^{th}$, $i = 1, \cdots, |y|$, are either $(\neg x_{n+i+1})$ if $y(i) = 0$, or (x_{n+i+1}) if $y(i) = 1$.*

Is that a padding function?
1. It is polynomial time computable, for sure.
2. It doesn't affect x's satisfiability, since we added satisfiable clauses with dedicated extra variables.
3. It is length increasing.
4. Given $pad(x, y)$ we can find where the "added" part begins.

We would like to have this kind of implication:

$$(A \leq_m^p B) \wedge (B \leq_m^p A) \Leftrightarrow (A \text{ isomorphic to } B)$$

This is essentially a polynomial-time version of Schröder-Bernstein theorem:

Theorem 3.3 (Schröder-Bernstein)

If there exists a 1-1 mapping from a set A to a set B, and a 1-1 mapping from B to A, then there is a *bijection* between A and B.

But, unfortunately, this is *not* sufficient with regular reductions. To achieve this analogy, we need to "enhance" our reductions with the previous features (1-1, length increasing, and polynomial time computable and invertible). We can use padding functions to transform regular reductions to "desired" ones:

Theorem 3.4

Let f be a reduction from A to B, and pad a length-increasing padding function for B. Then, the function mapping $x \in \Sigma^*$ to $pad(f(x), x)$ is a length-increasing 1-1 reduction. Furthermore, there exists a polynomial time algorithm, which given $pad(f(x), x)$ recovers x.

Using the above Theorem 3.4, we can obtain a polynomial version of Theorem 3.3, with the extra assumption of paddability:

Theorem 3.5 (Polynomial-time version of Schröder-Bernstein Theorem)

Let A and B be length-increasingly paddable languages. If $A \leq_m^p B$ and $B \leq_m^p A$, then A and B are polynomial-time isomorphic.

- The length-increasing property in the above theorem can be removed, the result holds also for A,B paddable languages.

As noted above, finding padding functions for known **NP**-complete languages (SAT, VERTEX COVER, HAMILTON PATH, CLIQUE, , KNAPSACK etc) has been proven an easy task. The next step is to assume that maybe *all* **NP**-complete languages are isomorphic to each other. This would mean that there is only one **NP**-complete problem modulo isomorphisms, i.e. all **NP**-complete problems are relabelings of the same language! This is a conjecture stated by Berman and Hartmanis:

Definition 3.3 (Berman-Hartmanis Conjecture)

All **NP**-complete languages are polynomial-time isomorphic to each other.

But, Berman-Hartmanis Conjecture implies that $\mathbf{P} \neq \mathbf{NP}$, since if $\mathbf{P} = \mathbf{NP}$ all **NP** languages would by **NP**-complete (*why?*), and

Remark 3.3

Note that the Berman-Hartmanis Conjecture analogue in recursion theory is proven, known as Myhill's theorem: we know that all **RE**-complete problems are essentially (recursive renamings of) the Halting Problem!

3.3 Padding

3.3.1 Translation Results

Padding allows us to "transform" a language by padding every string with useless symbols. That is, we can transform a language L to:

$$L_p = \{x \underbrace{\$\$\$ \cdots \$}_{t(|x|) \text{ times}} \mid x \in L\}$$

where "\$" is a symbol *not* in L's alphabet. Then, we can transform a TM M deciding L to an M' deciding L_p, just by ignoring the \$'s. The running time of M' is measured as a function of the input length, which is now $|x| + t(|x|)$.

We can use this technique to prove *upwards translations* of equalities of complexity classes (and thus to prove downwards translations for inequalities):

Theorem 3.6

If **NEXP** $\neq$ **EXP**, then **P** $\neq$ **NP**.

Proof. We will prove the contrapositive: If **P** $=$ **NP**, then **NEXP** $=$ **EXP**. Let $L \in$ **NTIME**$[2^{n^c}]$ and M a TM deciding it. We define the language:

$$L_p = \{x\$^{2^{|x|^c}} \mid x \in L\}$$

Then, L_p is in **NP**: First, check if the input has the right format ($x\$ \cdots \$, x \in \Sigma^*$). Then, simulate $M(x)$ for $2^{|x|^c}$ steps and output the answer. Clearly, the running time of this machine is polynomial in its input size.

By our assumption, L_p is also in **P**. So, can use the machine in **P** to decide L in **EXP**: on input x, pad it using $2^{|x|^c}$ \$'s, and use the machine in **P** to decide L_p. The running time is $2^{|x|^c}$, so $L \in$ **EXP**. □

In the same way:

Theorem 3.7

If **DSPACE**$[n] \subseteq$ **P**, then **P** $=$ **PSPACE**.

Proof. Fix a $k > 0$, and let $L \in \mathbf{DSPACE}[n^k]$. Then, define:

$$L_p = \{x\$^\ell \mid x \in L \land |x\$^\ell| = |x|^k\}$$

so $L_p \in \mathbf{DSPACE}[n]$, thus $L_p \in \mathbf{P}$, by our assumption. We conclude that $L \in \mathbf{P}$ as well, because we can pad the input x to $x\$^\ell$, and use the $\mathbf{P}$ machine for L_p. □

3.3.2 Separation Results

We can use padding to separate complexity classes:

Theorem 3.8

$\mathbf{E} \neq \mathbf{PSPACE}$

Proof. Assume, for the sake of contradiction, that $\mathbf{E} = \mathbf{PSPACE}$. Let $L \in \mathbf{DTIME}[2^{n^2}]$. We define:

$$L_p = \{x\$^\ell \mid x \in L \land |x\$^\ell| = |x|^2\}$$

So, $L_p \in \mathbf{DTIME}[2^n]$, and from our assumption we have that $L_p \in \mathbf{PSPACE}$, that is $L_p \in \mathbf{DSPACE}[n^k]$, for some $k \in \mathbb{N}$. We can convert this n^k-space-bounded machine to another, deciding L: Given x, add $\ell = |x|^2 - |x|$ \$'s, and simulate the n^k-space-bounded machine on the padded input. We used $|x|^{2k}$ space, so $L \in \mathbf{PSPACE}$.

Thus, we proved that $\mathbf{DTIME}[2^{n^2}] \subseteq \mathbf{PSPACE}$. But, $\mathbf{E} \subsetneq \mathbf{DTIME}[2^{n^2}]$, due to the Time Hierarchy Theorem, and therefore $\mathbf{E} \neq \mathbf{PSPACE}$. □

► Note that we don't know whether $\mathbf{E} \subseteq \mathbf{PSPACE}$ or $\mathbf{PSPACE} \subseteq \mathbf{E}$!

3.4 Density of Languages

Languages are sets of strings, and they often inherit notions from other areas of mathematics. A very useful one is *density*. In the context of complexity theory, where we 're interested in input lengths and polynomial sizes, a dense language contains a *superpolynomial* number of strings for some string lengths. If, on the other hand, every "slice" of strings of length n in the language is bounded by a polynomial of n, then the language is called *sparse*. This leads us to the following definition:

Definition 3.4 (Sparse Sets)

A language $L \subseteq \Sigma^*$ is called *sparse* if $|L \cap \Sigma^n| = poly(n)$ for every $n \in \mathbb{N}$.

This means that for every input length n, the number of strings of length n in L is at most polynomial in n. Notice that this definition could be stated equivalently as $|L \cap \Sigma^{\leq n}| = poly(n)$, where $\Sigma^{\leq n} = \{x \in \Sigma^* : \ |x| \leq n\}$. Sparse sets could be considered as sets of low-information content. In general,

Definition 3.5

Let $L \subseteq \Sigma^*$ be a language. We define its *density* as the following function from $\mathbb{N} \to \mathbb{N}$:

$$dens_L(n) = |\{x \in L : |x| \leq n\}|$$

- $dens_L(n)$ is the *number of strings* in L of length up to n.
- By Definition 3.4, a language L is *sparse* if there exists a polynomial q such that for every $n \in \mathbb{N} : dens_L(n) \leq q(n)$.

Theorem 3.9

If a language A is paddable, then it is *not* sparse.

Proof. Let $A \subseteq \Sigma^*$ with padding function $pad_A : \Sigma^* \times \Sigma^* \to \Sigma^*$. Suppose, for the sake of contradiction, that A is sparse, i.e. $\exists$ polynomial q $\forall n \in \mathbb{N} : dens_A(n) \leq q(n)$. Since $pad_A \in$ **FP**, there exist an $r \in poly(n)$:

$$|pad_A(x, y)| \leq r(|x| + |y|)$$

(*a function computed in polynomial time can print at most a polynomial size output*).
Fix a $x \in A$, since pad_A is 1-1 :

$$2^n \leq |\{pad_A(x, y) : |y| \leq n\}| \leq dens_A(r(|x| + n)) \leq q(r(|x| + n))$$

for all $n \in \mathbb{N}$, which is a contradiction. □

Theorem 3.10

If the Berman-Hartmanis conjecture is *true*, then all **NP**-complete and all *co***NP**-complete languages are not sparse.

Proof. If Berman-Hartmanis conjecture is true, every **NP**-complete language A is polynomial-time isomorphic to SAT. Let f be this isomorphism, and pad_{SAT} a padding function for SAT.

Define:

$$p_A(x, y) := f^{-1}(pad_{\text{SAT}}(f(x), y))$$

Then:

$$x \in A \Leftrightarrow f(x) \in \text{SAT} \Leftrightarrow pad_{\text{SAT}}(f(x), y) \in \text{SAT} \Leftrightarrow f^{-1}(pad_{\text{SAT}}(f(x), y)) \in A$$

By definition, pad_{SAT}, f and f^{-1} are polynomial time *computable*. So, p_A is a padding function for A, hence A is paddable, and by the above Theorem 3.9, A is *not sparse*.

Also, the complements of paddable languages are paddable (*why?*), so *co***NP**-complete languages are also not sparse.

□

We can relax the assumption of the above theorem to (the weaker) $\mathbf{P} \neq \mathbf{NP}$. This result, known as Mahaney's theorem, states that if $\mathbf{P} \neq \mathbf{NP}$, all **NP**-complete languages are *dense*.

Theorem 3.11 (Mahaney)

For any sparse $S \neq \emptyset$, SAT $\leq_m^p S$ *if and only if* $\mathbf{P} = \mathbf{NP}$.

Proof. (*Ogihara-Watanabe*)

($\Leftarrow$) This direction is trivial, since if $\mathbf{P} = \mathbf{NP}$, then any **NP**-complete language reduces to the sparse language $\{1\}$ (*why?*).

($\Rightarrow$) For this direction, assume that SAT $\leq_m^p S$ for a *sparse* non-empty set S. Define the language LSAT:

$$\text{LSAT} = \{\langle \phi, \sigma \rangle \mid \phi \text{ boolean formula, and } \exists \tau, \tau \preceq \sigma : \ \phi|_\tau = T\}$$

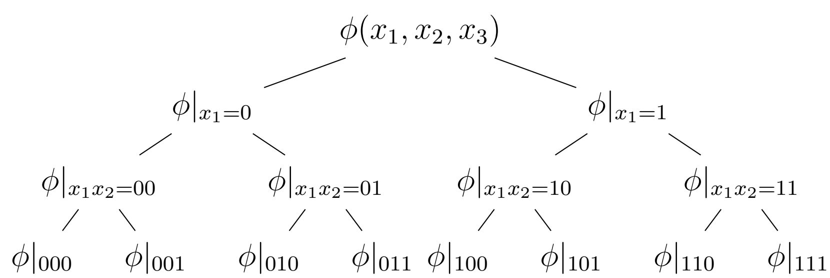
That is, LSAT contains tuples of formulas ϕ and partial truth assignments σ, such that there exist some truth assignment τ, which precedes lexicographically σ and satisfies the restriction of ϕ to these variables. Essentially, it is a "lexicographically bounded" SAT.

Notice that $\langle \phi, 1^n \rangle \in$ LSAT $\Leftrightarrow \phi \in$ SAT, so we can easily reduce SAT to LSAT, thus LSAT is **NP**-complete. Also, if $\sigma_1 \preceq \sigma_2$ and $\langle \phi, \sigma_1 \rangle \in$ LSAT, then $\langle \phi, \sigma_2 \rangle \in$ LSAT.

So, LSAT $\leq_m^p S$, and let f be this *reduction*. By definition:

$$\langle \phi, \sigma \rangle \in \text{LSAT} \Leftrightarrow f(\langle \phi, \sigma \rangle) \in S$$

Consider now the self-reducibility tree of ϕ as a *partial assignments tree*, where at each intermediate node we fix a variable, creating a partial assignment, and at the leaves all the variables are fixed, forming full truth assignments.



Example 3.2.
Example of self-reducibility tree of ϕ with 3 variables. Notice that the leaves contain all possible truth assignments, lexicographically ordered. *The intermediate nodes contain all the partial truth assignments, and the root is considered as the empty truth assignment ε.*

Using this framework, we will exploit f as a subroutine to create an algorithm for SAT (under the theorem's assumption, of course). If this algorithm is in polynomial time, then $\mathbf{P} = \mathbf{NP}$.

Observe that since $f \in \mathbf{FP}$, $|f(x)| \leq p(|x|)$, for a polynomial p and every $x \in \Sigma^*$. In addition, f maps strings of LSAT to strings of S, and S is sparse, i.e. it contains at most polynomially many strings for every string length, hence the number of strings with length at most $p(n)$ is also polynomial in n. Let this polynomial be $q(n)$, where $q(n) = |S \cap \Sigma^{\leq p(n)}|$.
The algorithm will work on the partial assignment tree by pruning some nodes at each level:

- Start from root.
- If the next level has $> q(n)$ nodes, run a *pruning* procedure until the nodes will be $\leq q(n)$.
- Output 1 if there is a satisfying truth assignment.

At the end, there will be n levels with at most $q(n)$ nodes each, so the tree is polynomial.

Pruning Procedure The pruning will work in two stages:

- At the first stage, we will compute $f(\langle \phi, \sigma_i \rangle)$, for all nodes σ_i at this level. If there are σ_1, σ_2 such that $f(\langle \phi, \sigma_1 \rangle) = f(\langle \phi, \sigma_2 \rangle)$ and $\sigma_1 \preceq \sigma_2$, then we throw away σ_2.
- If there are $> q(n)$ nodes left, we apply the second stage: *remove leftmost node*. That is, remove the leftmost partial assignment, until there are $q(n)$ nodes left.

Why is this correct? We must assure that the above pruning procedure does not affect the satisfiability of ϕ. Here, we can take advantage of LSAT:

> *If ϕ satisfiable, at the end of iteration on each level, there is an ancestor of the leftmost satisfying truth assignment of ϕ.*

We can prove the above claim using induction on the depth of the tree:

- For the root, it is trivial.
- Suppose of the claim holds for level $k-1$, so it contains an ancestor of the leftmost satisfying truth assignment for ϕ. Of course, it holds for level k, before the prunning.

  For the duplicates removal (*first stage*), since $f(\langle \phi, \sigma_2 \rangle) \in S \Rightarrow f(\langle \phi, \sigma_1 \rangle) \in S$, ϕ has a satifying truth assignment smaller than σ_1. Hence, the leftmost satifying truth assignment has *not* σ_2 as ancestor (or else we would have the contradiction $\langle \phi, \sigma_1 \rangle \notin$ LSAT and $\langle \phi, \sigma_2 \rangle \in$ LSAT!)

  For leftmost nodes removal (*second stage*), if the level contains more than q nodes, there will be at least one σ such that $f(\langle \phi, \sigma \rangle) \notin S$, because S is sparse, and has at most $q(n)$ strings (recall that due to first stage, all nodes are distinct). Then, ϕ will *not* have a satisfying truth assignment smaller than σ, so all partial truth assignments to the left of σ can be pruned.

At the end of the pruning of each level we have at most $q(n)$ nodes, so at the next level there will be at most $2q(n)$ nodes, before the prunning. The application of f to $2q(n)$ nodes is overall polynomial, and at the leaves we will have at most $q(n)$ (full) truth assignments to check, thus the above algorithm functions in polynomial time. □

3.5 Summary

- Classes like **NP**, **PSPACE** or **FP** can be *effectively enumerated*.
- If $\mathbf{P} \neq \mathbf{NP}$, there exist problems in **NP** which are not **NP**-complete neither in **P**.
- We can obtain polynomial-time isomorphisms between languages, given they are interreducible and paddable.
- Berman-Hartmanis Conjecture postulates that all NP-complete languages are polynomial-time isomorphic to each other.
- We can use padding to *translate upwards* equalities between complexity classes.
- If $\mathbf{P} \neq \mathbf{NP}$, then a *sparse* set *cannot* be $\leq_m^p$-hard for **NP**.

Acknowledgements

Special thanks to grad student Orestis Milolidakis for his useful comments.

References

[AB09] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 1st edition, April 2009.

[BC94] Daniel Pierre Bovet and Pierluigi Crescenzi. *Introduction to the Theory of Complexity*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1994.

[Cai03] Jin-Yi Cai. Lectures in Computational Complexity, 2003.

[DK00] Ding-Zhu Du and Ker-I Ko. *Theory of Computational Complexity*. Wiley-Interscience, January 2000.

[For00] L. Fortnow. Diagonalization. 71:102–112, June 2000. Computational Complexity Column.

[Kat11] Jonathan Katz. Notes on Complexity Theory, 2011.

[Pap94] Christos H. Papadimitriou. *Computational Complexity*. Addison Wesley, 1994.