Computational complexity

lecture 8

Ladner's theorem

Theorem (Ladner, 1975) – existence of NP-intermediate problems:

If **P≠NP**, then there is a problem, which is in **NP\P**, but is not **NP**-hard with respect to polynomial-time reductions (so even more with respect to logarithmic-space reductions).

Ladner's theorem

Theorem (Ladner, 1975) – existence of NP-intermediate problems:

If **P≠NP**, then there is a problem, which is in **NP\P**, but is not **NP**-hard with respect to polynomial-time reductions (so even more with respect to logarithmic-space reductions).

Proof:

Supposing that SAT$\notin$**P** we will give a language L∈**NP** such that:

- L is not in **P**, and
- SAT does not reduce to L in polynomial time

Ladner's theorem

Theorem (Ladner, 1975) – existence of NP-intermediate problems:

If **P≠NP**, then there is a problem, which is in **NP\P**, but is not **NP**-hard with respect to polynomial-time reductions (so even more with respect to logarithmic-space reductions).

Proof:

Supposing that SAT∉**P** we will give a language L∈**NP** such that:

- L is not in **P**, and
- SAT does not reduce to L in polynomial time

We create L as a variant of SAT with an appropriate amount of padding. In general, with padding we can change a problem into a simpler one. We want to add enough padding so that the SAT problem stops to be **NP**-complete, but not too much, so that still it is not in **P**.

The definition will be:

$$L=\{w01^{f(|w|)} : w\in \text{SAT}\}$$

for an appropriate function f

Ladner's theorem (⁎)

$L=\{w01^{f(|w|)} : w\in$SAT$\}$ for an appropriate function f.

We now define f

- Fix a computable enumeration $M_1,M_2,M_3,...$ of Turing machines, such that M_i works in time $O(n^i)$, and every language in **P** is recognized by some M_i
- To this end, we take a list $M'_1,M'_2,M'_3,...$ on which <u>every</u> Turing machine appears infinitely often. To M'_i we add a counter, which stops the machine after n^i steps – this results in M_i

Ladner's theorem (∗)

$L=\{w01^{f(|w|)} : w\in$ SAT$\}$ for an appropriate function f.

We now define f

- Fix a computable enumeration $M_1,M_2,M_3,\ldots$ of Turing machines, such that M_i works in time $O(n^i)$, and every language in **P** is recognized by some M_i

The function f is defined by the following algorithm:

(a) take $i=1$, $n=1$

(b) put $f(n)=n^i$

(c) if there is a word v of length $\leq log(n)$ such that M_i incorrectly recognizes whether v belongs to L, then increase i by 1

(d) increase n by 1, go back to (b)

Ladner's theorem (∗)

M_i works in time $O(n^i)$, every lang. in **P** is recognized by some M_i

$L=\{w01^{f(|w|)} : w\in\text{SAT}\}$ for f defined by:

(a) take $i=1$, $n=1$

(b) put $f(n)=n^i$

(c) if there is a word v of length $\leq log(n)$ such that M_i incorrectly recognizes whether v belongs to L, then increase i by 1

(d) increase n by 1, go back to (b)

Fact 1: It can be checked in polynomial time whether a word is of the proper form (i.e., if the number of ones is appropriate).

- In order to compute $f(n)$ we repeat the loop n times, in every repetition we check polynomially many words v (of logarithmic length)
- On every word v we run M_i, which works in time $O(log^i n)$
- We can spend this time, as the input should have length $\geq f(n)\geq n^i$ (we interrupt the loop as soon as there are not enough ones)
- Remark: i is not a constant (time $O(log^i n)$ by itself is not polynomial)
- Remark 2: the simulation time depends on $|M_i|$, but $|M_i|=|i|=log(i)\leq log(n)$, so this is OK

Ladner's theorem (∗)

M_i works in time $O(n^i)$, every lang. in **P** is recognized by some M_i

$L=\{w01^{f(|w|)} : w\in\text{SAT}\}$ for f defined by:

(a) take $i=1$, $n=1$

(b) put $f(n)=n^i$

(c) if there is a word v of length $\leq log(n)$ such that M_i incorrectly recognizes whether v belongs to L, then increase i by 1

(d) increase n by 1, go back to (b)

<u>Fact 1</u>: It can be checked in polynomial time whether a word is of the proper form (i.e., if the number of ones is appropriate).

- In order to compute $f(n)$ we repeat the loop n times, in every repetition we check polynomially many words v (of logarithmic length)
- On every word v we run M_i, which works in time $O(log^i n)$
- We can spend this time, as the input should have length $\geq f(n) \geq n^i$ (we interrupt the loop as soon as there are not enough ones)
- We also need to check whether $v\in L$ (where $|v|\leq log\ n$)
  - ➔ we check the number of ones in v by the induction assumption
  - ➔ we check whether prefix∈SAT in time exponential in $log(n)$

Ladner's theorem (∗)

M_i works in time $O(n^i)$, every lang. in **P** is recognized by some M_i

$L=\{w01^{f(|w|)} : w\in$ SAT$\}$ for f defined by:

(a) take $i=1$, $n=1$

(b) put $f(n)=n^i$

(c) if there is a word v of length $\leq log(n)$ such that M_i incorrectly recognizes whether v belongs to L, then increase i by 1

(d) increase n by 1, go back to (b)

Fact 1: It can be checked in polynomial time whether a word is of the proper form (i.e., if the number of ones is appropriate).

Corollary: $L\in$ **NP**

Ladner's theorem (⁎)

M_i works in time $O(n^i)$, every lang. in **P** is recognized by some M_i

$L=\{w01^{f(|w|)} : w\in\text{SAT}\}$ for f defined by:

(a) take $i=1$, $n=1$

(b) put $f(n)=n^i$

(c) if there is a word v of length $\leq log(n)$ such that M_i incorrectly recognizes whether v belongs to L, then increase i by 1

(d) increase n by 1, go back to (b)

Fact 2: if SAT$\notin$**P** then $L\notin$**P**

- If $L\in$**P**, then some M_i recognizes L, so from some moment on (i.e. for $n\geq n_0$ for some n_0) we have that $f(n)=n^i$
- Then it is easy to solve SAT in **P** (a contradiction):
  - ➔ if $|w|\geq n_0$ we append $|w|^i$ ones at the end, and we start M_i
  - ➔ for w shorter than n_0 the results can be hardcoded

Ladner's theorem (∗)

M_i works in time $O(n^i)$, every lang. in **P** is recognized by some M_i

$L=\{w01^{f(|w|)} : w\in$SAT$\}$ for f defined by:

(a) take $i=1$, $n=1$

(b) put $f(n)=n^i$

(c) if there is a word v of length $\leq log(n)$ such that M_i incorrectly recognizes whether v belongs to L, then increase i by 1

(d) increase n by 1, go back to (b)

Fact 2: if SAT$\notin$**P** then $L\notin$**P**

- If $L\in$**P**, then some M_i recognizes L, so from some moment on (i.e. for $n\geq n_0$ for some n_0) we have that $f(n)=n^i$
- Then it is easy to solve SAT in **P** (a contradiction):
  - if $|w|\geq n_0$ we append $|w|^i$ ones at the end, and we start M_i
  - for w shorter than n_0 the results can be hardcoded

Corollary: Because $L\notin$**P**, the function f grows faster than every polynomial

Ladner's theorem (∗)

M_i works in time $O(n^i)$, every lang. in **P** is recognized by some M_i

$L=\{w01^{f(|w|)} : w \in \text{SAT}\}$ for an appropriate f.

<u>Fact 3</u>: if SAT$\notin$**P** then L is not **NP**-hard

- Suppose that SAT reduces to L through a function g computable in time n^k. We will show a polynomial algorithm for SAT.

Ladner's theorem (∗)

M_i works in time $O(n^i)$, every lang. in **P** is recognized by some M_i

$L=\{w01^{f(|w|)} : w\in\text{SAT}\}$ for an appropriate f.

<u>Fact 3</u>: if SAT$\notin$**P** then L is not **NP**-hard

- Suppose that SAT reduces to L through a function g computable in time n^k. We will show a polynomial algorithm for SAT.
- We know that there is n_0 such that for $n\geq n_0$ it holds that $f(n)>n^k$
- For formulas w shorter than n_0 the results can be hardcoded

Ladner's theorem (∗)

M_i works in time $O(n^i)$, every lang. in **P** is recognized by some M_i

$L=\{w01^{f(|w|)} : w\in\text{SAT}\}$ for an appropriate f.

Fact 3: if SAT$\notin$**P** then L is not **NP**-hard

- Suppose that SAT reduces to L through a function g computable in time n^k. We will show a polynomial algorithm for SAT.
- We know that there is n_0 such that for $n\geq n_0$ it holds that $f(n)>n^k$
- For formulas w shorter than n_0 the results can be hardcoded
- For $|w|\geq n_0$ we consider the word $g(w)$; it has length $\leq|w|^k$.

  If $g(w)$ is not of the form $w'01^{f(|w'|)}$, then it is not in L, we reject (by fact 1, this can be checked in **P**). Otherwise $w\in\text{SAT} \Leftrightarrow w'\in\text{SAT}$

Ladner's theorem (∗)

M_i works in time $O(n^i)$, every lang. in **P** is recognized by some M_i

$L=\{w01^{f(|w|)} : w\in\text{SAT}\}$ for an appropriate f.

<u>Fact 3</u>: if SAT$\notin$**P** then L is not **NP**-hard

- Suppose that SAT reduces to L through a function g computable in time n^k. We will show a polynomial algorithm for SAT.
- We know that there is n_0 such that for $n\geq n_0$ it holds that $f(n)>n^k$
- For formulas w shorter than n_0 the results can be hardcoded
- For $|w|\geq n_0$ we consider the word $g(w)$; it has length $\leq|w|^k$.

  If $g(w)$ is not of the form $w'01^{f(|w'|)}$, then it is not in L, we reject (by fact 1, this can be checked in **P**). Otherwise $w\in\text{SAT} \Leftrightarrow w'\in\text{SAT}$

  Moreover, either $|w'|<n_0$, or we have that $|w|^k\geq|g(w)|>f(|w'|)>|w'|^k$, thus the new formula is shorter at least by 1.
- We repeat this in a loop; after a linear number of steps the input length decreases below n_0, and we obtain a result.

Ladner's theorem

We have thus proved:

Theorem (Ladner 1975)

If **P≠NP**, then there is a problem, which is in **NP\P**, but is not **NP**-hard with respect to polynomial-time reductions (so even more with respect to logarithmic-space reductions).

CSP problems and the dichotomy theorem

The CSP problem

Input: variables $x_1,\ldots,x_n$, domains $D_1,\ldots,D_n$, constraints $C_1,\ldots,C_m$ of the form (t,R), where t is a tuple of k variables, and R is a k-ary relation

Question: are there $x_1 \in D_1,\ldots,x_n \in D_n$ satisfying $C_1,\ldots,C_m$?

(a constraint (t,R) is satisfied if the tuple of variables t belong to the relation R)

Clearly CSP$\in$**NP**

CSP problems and the dichotomy theorem

The CSP problem

Input: variables $x_1,\ldots,x_n$, domains $D_1,\ldots,D_n$, constraints $C_1,\ldots,C_m$ of the form (t,R), where t is a tuple of k variables, and R is a k-ary relation

Question: are there $x_1 \in D_1,\ldots,x_n \in D_n$ satisfying $C_1,\ldots,C_m$?

(a constraint (t,R) is satisfied if the tuple of variables t belong to the relation R)

Clearly CSP$\in$**NP**

Most natural **NP**-complete problems can be easily reduced to CSP (written as CSP).

E.g. 3-coloring:

- $x_1,\ldots,x_n$ – represent colors of nodes $1,\ldots,n$
- $D_1,\ldots,D_n=\{1,2,3\}$
- for every edge k,l we have a constraint $x_k \neq x_l$

  (i.e., R is the binary relation $\{(1,2),(2,1),(1,3),(3,1),(2,3),(3,2)\}$)

CSP problems and the dichotomy theorem

The CSP problem

Input: variables $x_1,\ldots,x_n$, domains $D_1,\ldots,D_n$, constraints $C_1,\ldots,C_m$ of the form (t,R), where t is a tuple of k variables, and R is a k-ary relation

Question: are there $x_1 \in D_1,\ldots,x_n \in D_n$ satisfying $C_1,\ldots,C_m$?

(a constraint (t,R) is satisfied if the tuple of variables t belong to the relation R)

Clearly CSP$\in$**NP**

Most natural **NP**-complete problems can be easily reduced to CSP (written as CSP).

Problem CSP(Γ) – like CSP, but only relations from a set Γ can be used

Theorem (2017): for every set Γ we either have CSP(Γ)$\in$**P**, or CSP(Γ) is **NP**-complete

Berman's theorem

Is it the case that every problem not in **NP** is **NP**-hard?

Intuitively, **NP**-hard means hardest in **NP**, or even harder (so problems harder than **NP** should be **NP**-hard).

Berman's theorem

Is it the case that every problem not in **NP** is **NP**-hard?

Intuitively, **NP**-hard means hardest in **NP**, or even harder
(so problems harder than **NP** should be **NP**-hard).

But the definition is: L is **NP**-hard if we can reduce every problem from **NP** to L.
So: can we reduce every problem from **NP**, to every (more difficult) problem not in **NP**?

Berman's theorem

Is it the case that every problem not in **NP** is **NP**-hard?

Intuitively, **NP**-hard means hardest in **NP**, or even harder (so problems harder than **NP** should be **NP**-hard).

But the definition is: L is **NP**-hard if we can reduce every problem from **NP** to L.
So: can we reduce every problem from **NP**, to every (more difficult) problem not in **NP**?

The answer is **no** – we have the following theorem:

Theorem (Berman 1978)
If **P≠NP**, then no language over a single-letter alphabet is **NP**-hard wrt. polynomial-time reductions (so even more wrt. logarithmic-space reductions).

Berman's theorem

Is it the case that every problem not in **NP** is **NP**-hard?

No – we have the following theorem:

Theorem (Berman 1978)
If **P≠NP**, then no language over a single-letter alphabet is **NP**-hard.

Notice that there is a language language over a single-letter alphabet that requires doubly-exponential running time (i.e., surely is not in **NP**): take any language L over $\{0,1\}$ requiring triple-exponential running time, and take $\{1^{|1w|_2} : w \in L\}$, where $|1w|_2$ is the number encoded in binary as $1w$.

There is also an undecidable language over a single-letter alphabet: $\{1^k : M_k$ halts on empty input$\}$

These languages are not **NP**-hard, and not in **NP** (assuming **P≠NP**).

Berman's theorem ($\ast$)

Theorem (Berman 1978)

If **P**≠**NP**, then no language over a single-letter alphabet is **NP**-hard.

Proof

Let L be an **NP**-hard language over a single-letter alphabet. We will give a polynomial-time algorithm for SAT, contradicting **P**≠**NP**.

Berman's theorem (∗)

Theorem (Berman 1978)

If **P≠NP**, then no language over a single-letter alphabet is **NP**-hard.

Proof

Let L be an **NP**-hard language over a single-letter alphabet. We will give a polynomial-time algorithm for SAT, contradicting **P≠NP**.

By assumption there is a reduction g from SAT to L.

The algorithm is as follows:

- We are given a formula ϕ
- We will keep a list of formulas $\psi_1,\ldots,\psi_k$ such that: ϕ is satisfiable iff some of $\psi_1,\ldots,\psi_k$ is satisfiable. Initially the list contains ϕ.

Berman's theorem (∗)

Theorem (Berman 1978)

If **P≠NP**, then no language over a single-letter alphabet is **NP**-hard.

Proof

Let L be an **NP**-hard language over a single-letter alphabet. We will give a polynomial-time algorithm for SAT, contradicting **P≠NP**.
By assumption there is a reduction g from SAT to L.

The algorithm is as follows:

- We are given a formula ϕ
- We will keep a list of formulas $\psi_1,\ldots,\psi_k$ such that: ϕ is satisfiable iff some of $\psi_1,\ldots,\psi_k$ is satisfiable. Initially the list contains ϕ.
- We alternatingly repeat two kinds of steps:

1) Replace every ψ_i by two formulas: $\psi_i[true/x]$ and $\psi_i[false/x]$, obtained by substituting true/false for one of variables.
(clearly ψ_i is satisfiable iff some of $\psi_i[true/x]$, $\psi_i[false/x]$ is satisfiable)

Berman's theorem (∗)

Theorem (Berman 1978)
If **P≠NP**, then no language over a single-letter alphabet is **NP**-hard.

Proof
Let L be an **NP**-hard language over a single-letter alphabet. We will give a polynomial-time algorithm for SAT, contradicting **P≠NP**.
By assumption there is a reduction g from SAT to L.

The algorithm is as follows:

- We are given a formula ϕ
- We will keep a list of formulas $\psi_1,\ldots,\psi_k$ such that: ϕ is satisfiable iff some of $\psi_1,\ldots,\psi_k$ is satisfiable. Initially the list contains ϕ.
- We alternatingly repeat two kinds of steps:

1) Replace every ψ_i by two formulas: $\psi_i[true/x]$ and $\psi_i[false/x]$, obtained by substituting true/false for one of variables.
(clearly ψ_i is satisfiable iff some of $\psi_i[true/x]$, $\psi_i[false/x]$ is satisfiable)
2) For every pair ψ_i,ψ_j such that $g(\psi_i)=g(\psi_j)$, remove ψ_i from the list, leave only ψ_j (notice that ψ_i is satisfiable iff some of ψ_j is satisfiable)

Berman's theorem (∗)

We alternatingly repeat two kinds of steps:

1) Replace every ψ_i by two formulas: $\psi_i[true/x]$ and $\psi_i[false/x]$, obtained by substituting true/false for one of variables.
(clearly ψ_i is satisfiable iff some of $\psi_i[true/x], \psi_i[false/x]$ is satisfiable)

2) For every pair ψ_i,ψ_j such that $g(\psi_i)=g(\psi_j)$, remove ψ_i from the list, leave only ψ_j (notice that ψ_i is satisfiable iff some of ψ_j is satisfiable)

The algorithm is correct. Why does it work in polynomial time?

- Recall that g is a polynomial-time reduction to a single-letter language. Thus $|g(\psi_i)|<p(|\psi_i|)$ for some polynomial p.
  Since there is only one single-letter word of every length, there are only $p(|\psi_i|)\leq p(|\phi|)$ possibilities for $g(\psi_i)$.
- In effect, the list has length $\leq p(|\phi|)$ after every execution of step 2, and $\leq 2 \cdot p(|\phi|)$ after every execution of step 1.
- Moreover, every step can be performed in polynomial time.

This finishes the proof.

Relativisation

Many proofs in the complexity theory uses Turing machines as "black-boxes" – the proofs are of the form:

- assume that there is a machine M working in time ... recognizing ...
- Out of it, we create M', which executes M many times in a loop...
- ... then it negates the results, executes itself on every machine ...
- at the end we obtain a machine M''''', about which we know that it cannot exist, thus M could not exist.

Such proofs relativize, i.e., they work also when every machine in the world has access to some fixed oracle (that is, it can ask whether a word belongs to a language L, and immediately obtain an answer)

Relativisation

Many proofs in the complexity theory uses Turing machines as "black-boxes" – the proofs are of the form:

- assume that there is a machine M working in time ... recognizing ...
- Out of it, we create M', which executes M many times in a loop...
- ...

Such proofs relativize, i.e., they work also when every machine in the world has access to some fixed oracle.

Examples of relativizing proofs: Turing theorem about undecidability, hierarchy theorems, gap theorems, Ladner's theorem, Immerman-Szelepcseny theorem, Savitch theorem, ...

On the other hand, proofs based on circuits do not relativize (it is not at all clear what is an oracle for a circuit)

The next theorem shows that using relativizing arguments we cannot solve the **P** vs. **NP** problem.

Baker-Gill-Solovay theorem

Theorem (Baker-Gill-Solovay, 1975)

There exist languages A and B such that $\mathbf{P}^A = \mathbf{NP}^A$ and $\mathbf{P}^B \neq \mathbf{NP}^B$

Baker-Gill-Solovay theorem

Theorem (Baker-Gill-Solovay, 1975)

There exist languages A and B such that $\mathbf{P}^A=\mathbf{NP}^A$ and $\mathbf{P}^B\neq\mathbf{NP}^B$

Proof

As A we can take QBF – we have:

$$\mathbf{NP}^{\mathrm{QBF}}\subseteq\mathbf{NPSPACE}=\mathbf{PSPACE}=\mathbf{P}^{\mathrm{QBF}}$$

Steps from the left:

- instead of asking the QBF oracle about a word, a machine can itself compute the answer (questions are of polynomial length, and QBF can be solved in polynomial space)
- Savitch theorem
- **PSPACE**-completeness of the QBF problem

Baker-Gill-Solovay theorem

Theorem (Baker-Gill-Solovay, 1975)

There exist languages A and B such that $\mathbf{P}^{A}=\mathbf{NP}^{A}$ and $\mathbf{P}^{B}\neq\mathbf{NP}^{B}$

Proof

As A we can take QBF – we have:

$$\mathbf{NP}^{\mathrm{QBF}}\subseteq\mathbf{NPSPACE}=\mathbf{PSPACE}=\mathbf{P}^{\mathrm{QBF}}$$

Steps from the left:

- instead of asking the QBF oracle about a word, a machine can itself compute the answer (questions are of polynomial length, and QBF can be solved in polynomial space)
- Savitch theorem
- **PSPACE**-completeness of the QBF problem

Does A=SAT work as well? – $\mathbf{NP}^{\mathrm{SAT}}\subseteq\mathbf{NP}\subseteq\mathbf{P}^{\mathrm{SAT}}$

Baker-Gill-Solovay theorem

<u>Theorem</u> (Baker-Gill-Solovay, 1975)

There exist languages A and B such that $\mathbf{P}^{A}=\mathbf{NP}^{A}$ and $\mathbf{P}^{B}\neq\mathbf{NP}^{B}$

<u>Proof</u>

As A we can take QBF – we have:

$\mathbf{NP}^{\mathrm{QBF}}\subseteq\mathbf{NPSPACE}=\mathbf{PSPACE}=\mathbf{P}^{\mathrm{QBF}}$

Steps from the left:

- instead of asking the QBF oracle about a word, a machine can itself compute the answer (questions are of polynomial length, and QBF can be solved in polynomial space)
- Savitch theorem
- **PSPACE**-completeness of the QBF problem

Does A=SAT work as well? – ~~$\mathbf{NP}^{\mathrm{SAT}}\subseteq\mathbf{NP}\subseteq\mathbf{P}^{\mathrm{SAT}}$~~

NO – an **NP** algorithm for SAT doesn't give the inclusion $\mathbf{NP}^{\mathrm{SAT}}\subseteq\mathbf{NP}$ (maybe the external algorithm „prefers” to obtain that a formula is not satisfiable, and it will incorrectly compute its satisfiability)

It is important that QBF can be solved in <u>deterministic</u> **PSPACE**

Baker-Gill-Solovay theorem (∗)

Theorem (Baker-Gill-Solovay, 1975)

There exist languages A and B such that $\mathbf{P}^A=\mathbf{NP}^A$ and $\mathbf{P}^B\neq\mathbf{NP}^B$

Proof

We now construct an oracle B, and we consider the language

$L=\{1^n$: some word w of length n belongs to $B\}$

- Clearly $L\in\mathbf{NP}^B$ – nondeterministic machine can guess some $w\in B$
- A deterministic machine recognizing L has a problem: it can only ask the oracle for consecutive words, but it has not enough time to check all of them. We only need to choose B so that indeed it is impossible to do anything better.

Baker-Gill-Solovay theorem (∗)

<u>Theorem</u> (Baker-Gill-Solovay, 1975)

There exist languages A and B such that $\mathbf{P}^A=\mathbf{NP}^A$ and $\mathbf{P}^B\neq\mathbf{NP}^B$

<u>Proof</u>

$L=\{1^n$: some word w of length n belongs to $B\}$

We now choose B:

- Fix a list $M_1,M_2,M_3,\ldots$ of all Turing machines with oracle working in polynomial time
  - ➔ an oracle is not a part of the definition of the machine,
  - ➔ for every M_i there should exist a polynomial p_i such that for every oracle the machine M_i works in time $p_i(n)$
  - ➔ if some M with oracle C recognizes a language L in polynomial time, then some M_i with oracle C also recognizes L
  - ➔ such a list $M_1,M_2,M_3,\ldots$ is created as in the proof of Ladner's theo.
  - ➔ this time, we do not use the fact that the list is computable (conversely to the proof of the Ladner's theorem)
- We construct B gradually, cheating consecutive machines

Baker-Gill-Solovay theorem (∗)

$L=\{1^n$: some word w of length n belongs to $B\}$

We create $B=\bigcup_{i\in\mathbb{N}}B_i$ and a sequence n_i such that:

- $M_i^{B_i}$ incorrectly recognizes the word 1^{n_i}
- M_i^B agrees with $M_i^{B_i}$ on the word 1^{n_i}

We start with $B_0=\emptyset$; then for consecutive i:

- we take n_i so large that for all $j<i$, machine M_j for on the word 1^{n_j} produces only queries shorter than n_i (thanks to this the machines that were cheated earlier remain cheated), and such that M_i on the word 1^{n_i} works in less than 2^{n_i} steps

Baker-Gill-Solovay theorem (∗)

$L=\{1^n$: some word w of length n belongs to $B\}$

We create $B=\bigcup_{i\in\mathbb{N}} B_i$ and a sequence n_i such that:

- $M_i^{B_i}$ incorrectly recognizes the word 1^{n_i}
- M_i^{B} agrees with $M_i^{B_i}$ on the word 1^{n_i}

We start with $B_0=\emptyset$; then for consecutive i:

- we take n_i so large that for all $j<i$, machine M_j for on the word 1^{n_j} produces only queries shorter than n_i (thanks to this the machines that were cheated earlier remain cheated), and such that M_i on the word 1^{n_i} works in less than 2^{n_i} steps
- run $M_i^{B_{i-1}}$ on the word 1^{n_i}
- if it accepts, take $B_i=B_{i-1}$ – then $1^{n_i}\notin L$, we have cheated M_i
- if it rejects, find a word w of length n_i about which M_i haven't asked (it exists, since M_i has made $<2^{n_i}$ step) and define $B_i=B_{i-1}\cup\{w\}$ Then $1^{n_i}\in L$, and we have cheated M_i

Baker-Gill-Solovay theorem (∗)

$L=\{1^n$: some word w of length n belongs to $B\}$

We create $B=\bigcup_{i\in\mathbb{N}}B_i$ and a sequence n_i such that:

- $M_i^{B_i}$ incorrectly recognizes the word 1^{n_i}
- M_i^{B} agrees with $M_i^{B_i}$ on the word 1^{n_i}

The language B is computable, but in this theorem this is meaningless

We start with $B_0=\emptyset$; then for consecutive i:

- we take n_i so large that for all $j<i$, machine M_j for on the word 1^{n_j} produces only queries shorter than n_i (thanks to this the machines that were cheated earlier remain cheated), and such that M_i on the word 1^{n_i} works in less than 2^{n_i} steps
- run $M_i^{B_{i-1}}$ on the word 1^{n_i}
- if it accepts, take $B_i=B_{i-1}$ – then $1^{n_i}\notin L$, we have cheated M_i
- if it rejects, find a word w of length n_i about which M_i haven't asked (it exists, since M_i has made $<2^{n_i}$ step) and define $B_i=B_{i-1}\cup\{w\}$ Then $1^{n_i}\in L$, and we have cheated M_i

Search problems

The **NP** class was defined for decision problems („yes/no”),

e.g., does there exist a valuation satisfying a formula,
does there exist a Hamiltonian cycle, ...

We can also consider search problems,

e.g., find a valuation satisfying a formula,
find a Hamiltonian cycle, ...

- Of course search problems are not easier than decision problems. Thus if **P**≠**NP**, then search problems cannot be solved in polynomial time as well.
- And what if **P**=**NP**? Maybe it is possible to decide quickly whether there is a Hamiltonian cycle, but it is impossible to quickly find it?

Search problems

The **NP** class was defined for decision problems („yes/no”),
e.g., does there exist a valuation satisfying a formula,
does there exist a Hamiltonian cycle, ...

We can also consider search problems,
e.g., find a valuation satisfying a formula,
find a Hamiltonian cycle, ...

- Of course search problems are not easier than decision problems. Thus if **P**≠**NP**, then search problems cannot be solved in polynomial time as well.
- And what if **P**=**NP**? ~~Maybe it is possible to decide quickly whether there is a Hamiltonian cycle, but it is impossible to quickly find it?~~
- Then it possible to solve also search problems in polynomial time.

Search problems

Theorem

If **P**=**NP**, then for every language $L \in$ **NP** there is a polynomial algorithm that reads $v \in L$ and finds a witness for v.

We refer here to the definition of **NP** using witnesses:

NP contains languages of the form $\{v : \exists w.\ v\$w \in R\}$, where R is a relation recognizable in polynomial time and such that $v\$w \in R$ implies $|w| \leq p(|v|)$ for some polynomial p.

Search problems

<u>Theorem</u>

If **P**=**NP**, then for every language $L \in$ **NP** there is a polynomial algorithm that reads $v \in L$ and finds a witness for v.

We refer here to the definition of **NP** using witnesses:
NP contains languages of the form $\{v : \exists w.\ v\$w \in R\}$, where R is a relation recognizable in polynomial time and such that $v\$w \in R$ implies $|w| \leq p(|v|)$ for some polynomial p.

<u>Proof</u>

Consider first the SAT problem – we assume that there is a polynomial-time algorithm A for SAT, we want to find a valuation:

- Using A we check whether the formula is satisfiable
- If yes, we set $x_1=1$ and we check whether it is still satisfiable
- Yes ⇒ keep $x_1=1$ and continue for a smaller formula
- No ⇒ set $x_1=0$ and continue for a smaller formula
- In this way we eliminate consecutive variables, and we obtain a whole valuation

Search problems

Theorem

If **P**=**NP**, then for every language $L \in$ **NP** there is a polynomial algorithm that reads $v \in L$ and finds a witness for v.

We refer here to the definition of **NP** using witnesses:

NP contains languages of the form $\{v : \exists w.\ v\$w \in R\}$, where R is a relation recognizable in polynomial time and such that $v\$w \in R$ implies $|w| \leq p(|v|)$ for some polynomial p.

Proof

- For SAT we already know, consider now an arbitrary problem from **NP**
- It is enough to see that the reduction from the Cook-Levin theorem (**NP**-hardness of SAT) is actually a Levin reduction (i.e., it allows to recover witnesses)