Computational complexity

lecture 6

Homework:
available on the webpage,
deadline: 27.11.2017

Determinization

Theorem (previous lecture)

NTIME$(f(n)) \subseteq$ **DSPACE**$(f(n))$

Theorem (now)

NSPACE$(f(n)) \subseteq \bigcup_{c\in\mathbb{N}}$ **DTIME**$(c^{f(n)+log(n)})$

Determinization

<u>Theorem</u>

$\mathbf{NSPACE}(f(n)) \subseteq \bigcup_{c\in\mathbb{N}} \mathbf{DTIME}(c^{f(n)+log(n)})$

<u>Proof</u>

- We have a nondeterm. machine M working in space $g(n)=O(f(n))$. W.l.o.g. we assume that M has only one working tape.

Determinization

<u>Theorem</u>

$\textbf{NSPACE}(f(n)) \subseteq \bigcup_{c\in\mathbb{N}} \textbf{DTIME}(c^{f(n)+log(n)})$

<u>Proof</u>

- We have a nondeterm. machine M working in space $g(n)=O(f(n))$. W.l.o.g. we assume that M has only one working tape.
- A configuration of M on a fixed input of length n can be represented as:
  - ➔ contents of the working tape, with a marker over the position of the head – $(2|\Gamma|)^{g(n)}$ possibilities
  - ➔ a position of the head on the input tape – $n+2$ possibilities
  - ➔ a state (a constant number of possibilities)
- Altogether, there are $d^{g(n)+log(n)}$ configurations (for some constant d)

Determinization

Theorem

NSPACE$(f(n)) \subseteq \bigcup_{c\in\mathbb{N}}$**DTIME**$(c^{f(n)+log(n)})$

Proof

- We have a nondeterm. machine M working in space $g(n)=O(f(n))$. W.l.o.g. we assume that M has only one working tape.
- A configuration of M on a fixed input of length n can be represented as:
  - ➔ contents of the working tape, with a marker over the position of the head – $(2|\Gamma|)^{g(n)}$ possibilities
  - ➔ a position of the head on the input tape – $n+2$ possibilities
  - ➔ a state (a constant number of possibilities)
- Altogether, there are $d^{g(n)+log(n)}$ configurations (for some constant d)
- Checking that there is an accepting run amount to checking reachability in the (directed) configuration graph.
- Reachability can be solved in time polynomial in the size of the graph (i.e., in the number of configurations).

Determinization

<u>Theorem</u>

NSPACE($f(n)$)$\subseteq \bigcup_{c\in\mathbb{N}}$**DTIME**($c^{f(n)+log(n)}$)

<u>Proof</u>

<u>Remark 1</u> – If we want to generate all configurations using space $g(n)$, we have to assume that $g(n)$ is space constructible (or at least constructible in time $O(c^{f(n)+log(n)})$). But we do not need to do this – we can construct the configuration graph “on the fly”: we only need to be able to generate configurations reachable from a given configuration in a single step (to this end, we space-constructibility of $f(n)$ is not needed).

Determinization

Theorem

$\textbf{NSPACE}(f(n)) \subseteq \bigcup_{c\in\mathbb{N}} \textbf{DTIME}(c^{f(n)+log(n)})$

Proof

Remark 1 – If we want to generate all configurations using space $g(n)$, we have to assume that $g(n)$ is space constructible (or at least constructible in time $O(c^{f(n)+log(n)})$). But we do not need to do this – we can construct the configuration graph "on the fly": we only need to be able to generate configurations reachable from a given configuration in a single step (to this end, we space-constructibility of $f(n)$ is not needed).

Remark 2 – the input is not treated as a part of a configuration; thus in order to generate configurations reachable from a given configuration in a single step we have to inspect the input word.

Determinization

<u>Corollaries</u>

L$\subseteq$**NL**$\subseteq$**P**$\subseteq$**NP**$\subseteq$**PSPACE**

Supposedly, all the inclusions are strict, but we only know that some of them has to be strict (space hierarchy theorem).

Reachability in a directed graph

Input: directed graph (as a list of edges, or as an incidence matrix, does not matter), nodes x, y

Question: it is possible to reach y from x?

Fact: the reachability problem is in **NL**.

Proof: the machine remembers the current node, and guesses the next node (alternatively: a path in the graph can be taken as a witness)

Reachability in a directed graph

Input: directed graph (as a list of edges, or as an incidence matrix, does not matter), nodes x, y

Question: it is possible to reach y from x?

Theorem: the reachability problem is in **DSPACE**($(\log n)^2$).

Reachability in a directed graph

Input: directed graph (as a list of edges, or as an incidence matrix, does not matter), nodes x, y

Question: it is possible to reach y from x?

Theorem: the reachability problem is in **DSPACE**($(\log n)^2$).

Proof:

- consider a more general problem PATH(x,y,k): is there a path from x to y of length at most 2^k?

Reachability in a directed graph

Input: directed graph (as a list of edges, or as an incidence matrix, does not matter), nodes x, y

Question: it is possible to reach y from x?

Theorem: the reachability problem is in **DSPACE**($(\log n)^2$).

Proof:

- consider a more general problem PATH(x,y,k): is there a path from x to y of length at most 2^k?
- can be easily solved for $k=0$
- in order to solve this for some $k>0$, we browse all nodes z, and for each of them we ask whether PATH(x,z,k-1) and PATH(z,y,k-1)

Reachability in a directed graph

Input: directed graph (as a list of edges, or as an incidence matrix, does not matter), nodes x, y

Question: it is possible to reach y from x?

Theorem: the reachability problem is in **DSPACE**($(\log n)^2$).

Proof:

- consider a more general problem PATH(x,y,k): is there a path from x to y of length at most 2^k?
- can be easily solved for $k=0$
- in order to solve this for some $k>0$, we browse all nodes z, and for each of them we ask whether PATH(x,z,k-1) and PATH(z,y,k-1)
- recursion – we need a stack, on which we store triples (x,y,k)
- every triple has size $\log(n)$, and there are $\log(n)$ of them (it is enough to consider $k\leq\log(n)$) – thus memory usage is $O((\log n)^2)$

Determinization

Theorem (Savitch, 1970)

NSPACE($f(n)$)$\subseteq$**DSPACE**($f(n)^2$) whenever $f(n)=\Omega(\log n)$

(earlier, we have shown that **NSPACE**($f(n)$)$\subseteq\bigcup_{c\in\mathbb{N}}$**DTIME**($c^{f(n)+\log(n)}$))

Determinization

Theorem (Savitch, 1970)

NSPACE($f(n)$)$\subseteq$**DSPACE**($f(n)^2$) whenever $f(n)=\Omega(\log n)$

Proof

- We have a nondet. machine M working in space $g(n)=O(f(n))$.
- Consider the graph of configurations fitting in space $\leq g(n)$ – there is $d^{g(n)}$ of them, for some d, because $g(n)=\Omega(\log n)$
- Every such configuration can be stored in space $O(f(n))$
- We are interested in reachability in this graph (to every accepting configuration) – using the previous theorem, we obtain a solution working in space $O((\log d^{g(n)})^2)=O(f(n)^2)$

Determinization

Theorem (Savitch, 1970)

NSPACE($f(n)$)$\subseteq$**DSPACE**($f(n)^2$) whenever $f(n)=\Omega(\log n)$

Proof

- We have a nondet. machine M working in space $g(n)=O(f(n))$.
- Consider the graph of configurations fitting in space $\leq g(n)$ – there is $d^{g(n)}$ of them, for some d, because $g(n)=\Omega(\log n)$
- Every such configuration can be stored in space $O(f(n))$
- We are interested in reachability in this graph (to every accepting configuration) – using the previous theorem, we obtain a solution working in space $O((\log d^{g(n)})^2)=O(f(n)^2)$
- Remark 1: we do not compute and remember the whole graph; we only check single edges at the very bottom of the recursion (can y be reached from x in a single step)

Determinization

Theorem (Savitch, 1970)

NSPACE($f(n)$)$\subseteq$**DSPACE**($f(n)^2$) whenever $f(n)=\Omega(log\ n)$

Proof – Remark 2:

- It would be useful to assume that $g(n)$ is space constructible: we need to browse all accepting configurations / configurations z, fitting in space $\leq g(n)$; we need to start with appropriate $k=log(d^{g(n)})$

Determinization

<u>Theorem</u> (Savitch, 1970)

NSPACE($f(n)$)$\subseteq$**DSPACE**($f(n)^2$) whenever $f(n)=\Omega(\log n)$

<u>Proof – Remark 2</u>:

- It would be useful to assume that $g(n)$ is space constructible: we need to browse all accepting configurations / configurations z, fitting in space $\leq g(n)$; we need to start with appropriate $k=\log(d^{g(n)})$
- However, we can succeed without this assumption: for consecutive values of S we check whether M accepts in space S, and whether M reaches a configuration in which it wants to increase the memory usage over S (if so, we increase S by 1, and we repeat)

Determinization

<u>Theorem</u> (Savitch, 1970)

NSPACE($f(n)$)$\subseteq$**DSPACE**($f(n)^2$) whenever $f(n)=\Omega(\log n)$

<u>Corollaries</u>:

NPSPACE=**PSPACE**=**coNPSPACE**

Next time, we will also prove that **NL**=**coNL**.

It seems that nondeterminism has smaller impact on space complexity than on time complexity (since probably **P**≠**NP**≠**coNP**)

(but we do not know whether **L**=**NL**; it's quite possible that they differ)

Reductions

Idea:

- problem A reduces to problem B if while knowing how to solve B it is easy to solve A as well
- if B is easy, then A is easy as well
- if A is difficult, then B is difficult as well

There are multiple kinds of reductions...

Turing reductions / Cook reductions

An oracle machine, with an oracle for a language K:

- a deterministic Turing machine
- a separate "query tape" used for writing queries to the oracle (write only, i.e., the head mover only right; its length is not included in the space complexity)
- special states $q_?$, q_{yes}, q_{no} for calling the oracle
- after entering state $q_?$, the state changes to q_{yes} if the word on the query tape is in K / to q_{no} if it is not in K; the query tape becomes empty and the head returns to its first cell (all this happens in a single step)

➔ A language L is Turing-reducible to K if there exist a machine with an oracle for K, which recognizes L

Turing reductions / Cook reductions

An oracle machine, with an oracle for a language K:

- a deterministic Turing machine
- a separate “query tape” used for writing queries to the oracle (write only, i.e., the head mover only right; its length is not included in the space complexity)
- special states $q_?$, q_{yes}, q_{no} for calling the oracle
- after entering state $q_?$, the state changes to q_{yes} if the word on the query tape is in K / to q_{no} if it is not in K; the query tape becomes empty and the head returns to its first cell (all this happens in a single step)

➔ A language L is Turing-reducible to K if there exist a machine with an oracle for K, which recognizes L

➔ By limiting the resources of M, one can talk about polynomial-time Turing reductions (often called Cook reductions), logarithmic-space Turing reductions, etc.

Turing reductions / Cook reductions

- A language L is Turing-reducible to K if there exist a machine with an oracle for K, which recognizes L
- By limiting the resources of M, one can talk about polynomial-time Turing reductions (often called Cook reductions), logarithmic-space Turing reductions, etc.

Observe that every language $L \in$ **NP** can be reduced to $\overline{L} \in$ **coNP**: it is enough to call the oracle for $\overline{L}$, and negate the answer.

But we don't know whether **NP** is contained in **coNP**.

This is rather inconvenient: we prefer not to have reductions between independent classes.

Thus Cook reductions are not so popular.

We prefer Karp reductions (next slide), having better properties.

Karp reductions

Idea: we can make only a single query to the language K, and we cannot negate the answer.

Karp reductions

Idea: we can make only a single query to the language K, and we cannot negate the answer.

A language $L \subseteq \Sigma^*$ is <u>Karp-reducible</u> to $K \subseteq \Gamma^*$ if there exists a function $f{:}\Sigma^* \to \Gamma^*$ computable in logarithmic space (sometimes: in polynomial time), such that $w \in L \Leftrightarrow f(w) \in K$ for every word $w \in \Sigma^*$.

Karp reductions

Idea: we can make only a single query to the language K, and we cannot negate the answer.

A language $L \subseteq \Sigma^*$ is Karp-reducible to $K \subseteq \Gamma^*$ if there exists a function $f: \Sigma^* \to \Gamma^*$ computable in logarithmic space (sometimes: in polynomial time), such that $w \in L \Leftrightarrow f(w) \in K$ for every word $w \in \Sigma^*$.

Fact: If L is Karp-reducible to K, then it is also Turing-reducible to K (with the same restrictions on resources)

Proof

- We have a machine computing f.
- We treat is a machine with oracle for K, which at the very end asks a single question.

Levin reductions

- Turing reductions and Karp reductions are for decision problems (i.e., languages – does there exist …)
- For problems in **NP** we often want to find a solution / a witness (e.g., a Hamiltonian cycle), not only decide that it exists.
- The idea of Levin reductions: additionally a witness for the first problem allows to recover a witness for the second problem.

Levin reductions

- Turing reductions and Karp reductions are for decision problems (i.e., languages – does there exist …)
- For problems in **NP** we often want to find a solution / a witness (e.g., a Hamiltonian cycle), not only decide that it exists.
- The idea of Levin reductions: additionally a witness for the first problem allows to recover a witness for the second problem.

Definition:

- It is a reduction between relations $R_1, R_2 \subseteq \Sigma^* \times \Sigma^*$
- R_1 is Levin-reducible to R_2 if there are functions $f: \Sigma^* \to \Sigma^*$, $g, h: \Sigma^* \times \Sigma^* \to \Sigma^*$ (computable in logarithmic space / polynomial time) such that:

  $R_1(x,y) \Rightarrow R_2(f(x), g(x,y))$

  $R_2(f(x),z) \Rightarrow R_1(x, h(x,z))$ (for all $x,y,z \in \Sigma^*$)

Levin reductions

- Turing reductions and Karp reductions are for decision problems (i.e., languages – does there exist …)
- For problems in **NP** we often want to find a solution / a witness (e.g., a Hamiltonian cycle), not only decide that it exists.
- The idea of Levin reductions: additionally a witness for the first problem allows to recover a witness for the second problem.

Definition:

- It is a reduction between relations $R_1,R_2 \subseteq \Sigma^* \times \Sigma^*$
- R_1 is Levin-reducible to R_2 if there are functions $f:\Sigma^* \to \Sigma^*$, $g,h:\Sigma^* \times \Sigma^* \to \Sigma^*$ (computable in logarithmic space / polynomial time) such that:

  $R_1(x,y) \Rightarrow R_2(f(x),g(x,y))$

  $R_2(f(x),z) \Rightarrow R_1(x,h(x,z))$ (for all $x,y,z \in \Sigma^*$)

Fact

The function f itself gives a Karp-reduction from $\exists R_1$ to $\exists R_2$

Reductions

Which reductions are better?

- Turing-reductions are closer to intuitions (e.g. if we can search for a Hamiltonian cycle in a single graph, then we can also search for Hamiltonian cycles in two graphs – but how to show a Karp reduction)
- but Turing reductions are too easy to find, e.g., every language can be reduced to its complement, which blurs differences between **NP** and **coNP**
- in practice, it usually possible to show a <u>Karp reduction</u>, thus since this notion is stronger, we use it
- for the same reason, we prefer reductions <u>in logarithmic space</u> over reductions in polynomial time
- in practice, we usually can even show a Levin reduction, but these are reductions between relations, not between languages, so they are not so popular

Completeness

Let C be a complexity class.

A language L is <u>C-complete</u> (with respect to logarithmic-space Karp reductions) if

- $L \in C$, and
- L is <u>C-hard</u>, i.e., every language from C Karp-reduces to L in logarithmic space

It is surprising that complete problems exist at all!

NP-completeness

<u>Theorem</u>

The following language is **NP**-complete

$TMSAT=\{(M,1^t,w) : M$ accepts w in at most t steps$\}$

(where M is a nondeterministic Turing machine)

NP-completeness

Theorem

The following language is **NP**-complete

$TMSAT=\{(M,1^t,w) : M$ accepts w in at most t steps$\}$

(where M is a nondeterministic Turing machine)

Proof

Clearly $TMSAT\in$**NP**: we simulate the run of M on w for at most t steps (this is polynomial in $|M|+t+|w|$).

NP-hardness: Consider a language $L\in$**NP**, recognized by a nondet. machine M working in polynomial time $T(n)$. Then for every w, $w\in L\Leftrightarrow(M,1^{T(|w|)},w)\in TMSAT$, and the word $(M,1^{T(|w|)},w)$ can be computed in logarithmic space.

NP-completeness

Theorem

The following language is **NP**-complete

$TMSAT=\{(M,1^t,w)$: M accepts w in at most t steps$\}$

(where M is a nondeterministic Turing machine)

Proof

Clearly $TMSAT \in$ **NP**: we simulate the run of M on w for at most t steps (this is polynomial in $|M|+t+|w|$).

NP-hardness: Consider a language $L \in$ **NP**, recognized by a nondet. machine M working in polynomial time $T(n)$. Then for every w, $w \in L \Leftrightarrow (M,1^{T(|w|)},w) \in TMSAT$, and the word $(M,1^{T(|w|)},w)$ can be computed in logarithmic space.

TMSAT is not a very useful problem.

Are there natural problems that are **NP**-complete?

NP-completeness of the SAT problem

SAT problem: for a given boolean formula with variables (written in the infix notation, with full bracketing, variables written as numbers) decide whether it is satisfiable (i.e., whether there is a valuation of variables under which the formula evaluates to true)

e.g., $((x_1 \vee x_2) \wedge ((\neg x_1) \vee (\neg x_2)))$ is true for $x_1=1, x_2=0$

Theorem (Cook, 1971)

The SAT problem is **NP**-complete.

NP-completeness of the SAT problem

SAT problem: for a given boolean formula with variables (written in the infix notation, with full bracketing, variables written as numbers) decide whether it is satisfiable (i.e., whether there is a valuation of variables under which the formula evaluates to true)

e.g., $((x_1 \vee x_2) \wedge ((\neg x_1) \vee (\neg x_2)))$ is true for $x_1=1, x_2=0$

Theorem (Cook, 1971)

The SAT problem is **NP**-complete.

Proof

- It is easy to see that SAT$\in$**NP** – we guess a valuation which makes the formula true
- It remains to prove **NP**-hardness

NP-completeness of the SAT problem

- Fix a language L recognized by a nondeterministic machine M in time bounded by a polynomial $p(n)$
- Basing on the input word w, we need to construct (in logarithmic space) a formula ϕ such that $w \in L \Leftrightarrow \phi$ is satisfiable
- Idea: variables store a run of M on the word w, the formula ensures correctness of the run. [somehow similarly as when converting a machine into a circuit]

NP-completeness of the SAT problem

- Fix a language L recognized by a nondeterministic machine M in time bounded by a polynomial $p(n)$
- Basing on the input word w, we need to construct (in logarithmic space) a formula ϕ such that $w \in L \Leftrightarrow \phi$ is satisfiable
- Idea: variables store a run of M on the word w,
  the formula ensures correctness of the run.
  [somehow similarly as when converting a machine into a circuit]
- Three kinds of variables:
  - t_{ick} – in step k, the letter in the i-th cell of the tape is c
  - s_{qk} – in step k the machine is in state q
  - h_{ik} – in step k the head is on position i
- we have polynomially many variables – $O((p(n))^2)$

NP-completeness of the SAT problem

Variables:

- ➔ t_{ick} – in step k, the letter in the i-th cell of the tape is c
- ➔ s_{qk} – in step k the machine is in state q
- ➔ h_{ik} – in step k the head is on position i

The formula – a conjunctions of things to check (of polynomial size):

- the initial tape contents, head position, and state are as expected:

$$s_{q_0 1} \wedge h_{01} \wedge t_{0 \triangleright 1} \wedge t_{1 w_1 1} \wedge \ldots \wedge t_{n w_n 1} \wedge t_{(n+1) \perp 1} \wedge \ldots \wedge t_{p(n) \perp 1}$$

NP-completeness of the SAT problem

Variables:

- ➔ t_{ick} – in step k, the letter in the i-th cell of the tape is c
- ➔ s_{qk} – in step k the machine is in state q
- ➔ h_{ik} – in step k the head is on position i

The formula – a conjunctions of things to check (of polynomial size):

- the initial tape contents, head position, and state are as expected:

$$s_{q_0 1} \wedge h_{01} \wedge t_{0 \triangleright 1} \wedge t_{1 w_1 1} \wedge \ldots \wedge t_{n w_n 1} \wedge t_{(n+1) \perp 1} \wedge \ldots \wedge t_{p(n) \perp 1}$$

- at most one state at a moment

$$\neg s_{qk} \vee \neg s_{q'k} \quad \text{when } 1 \leq k \leq p(n),\ q \neq q'$$

NP-completeness of the SAT problem

Variables:

- t_{ick} – in step k, the letter in the i-th cell of the tape is c
- s_{qk} – in step k the machine is in state q
- h_{ik} – in step k the head is on position i

The formula – a conjunctions of things to check (of polynomial size):

- the initial tape contents, head position, and state are as expected:

$$s_{q_0 1} \wedge h_{01} \wedge t_{0 \triangleright 1} \wedge t_{1 w_1 1} \wedge \ldots \wedge t_{n w_n 1} \wedge t_{(n+1) \perp 1} \wedge \ldots \wedge t_{p(n) \perp 1}$$

- at most one state at a moment

$$\neg s_{qk} \vee \neg s_{q'k} \quad \text{when } 1 \le k \le p(n),\ q \ne q'$$

- at most one head position at a moment
- at most one symbol in a cell at a moment
- symbols not under the head remain unchanged

$$h_{jk} \wedge t_{ick} \rightarrow t_{ic(k+1)} \quad \text{when } 1 \le k \le p(n),\ q \ne q',\ i \ne j'$$

NP-completeness of the SAT problem

Variables:

- ➔ t_{ick} – in step k, the letter in the i-th cell of the tape is c
- ➔ s_{qk} – in step k the machine is in state q
- ➔ h_{ik} – in step k the head is on position i

The formula – a conjunctions of things to check (of polynomial size):

- the initial tape contents, head position, and state are as expected:

$$s_{q_0 1} \wedge h_{01} \wedge t_{0 \triangleright 1} \wedge t_{1 w_1 1} \wedge \ldots \wedge t_{n w_n 1} \wedge t_{(n+1) \perp 1} \wedge \ldots \wedge t_{p(n) \perp 1}$$

- at most one state at a moment

$\neg s_{qk} \vee \neg s_{q'k}$ when $1 \leq k \leq p(n)$, $q \neq q'$

- at most one head position at a moment
- at most one symbol in a cell at a moment
- symbols not under the head remain unchanged

$h_{jk} \wedge t_{ick} \rightarrow t_{ic(k+1)}$ when $1 \leq k \leq p(n)$, $q \neq q'$, $i \neq j'$

- a transition is performed (an alternative over possible transitions):

$$t_{ick} \wedge s_{qk} \wedge h_{ik} \rightarrow \bigvee (t_{ic'(k+1)} \wedge s_{q'(k+1)} \wedge h_{(i \pm 1)(k+1)})$$

NP-completeness of the SAT problem

The formula – a conjunctions of things to check (of polynomial size):

- the initial tape contents, head position, and state are as expected:

$$s_{q_0 1} \wedge h_{01} \wedge t_{0 \triangleright 1} \wedge t_{1 w_1 1} \wedge \ldots \wedge t_{n w_n 1} \wedge t_{(n+1) \perp 1} \wedge \ldots \wedge t_{p(n) \perp 1}$$

- at most one state at a moment

$$\neg s_{qk} \vee \neg s_{q'k} \quad \text{when } 1 \leq k \leq p(n),\ q \neq q'$$

- at most one head position at a moment
- at most one symbol in a cell at a moment
- symbols not under the head remain unchanged

$$h_{jk} \wedge t_{ick} \rightarrow t_{ic(k+1)} \quad \text{when } 1 \leq k \leq p(n),\ q \neq q',\ i \neq j'$$

- a transition is performed (an alternative over possible transitions):

$$t_{ick} \wedge s_{qk} \wedge h_{ik} \rightarrow \bigvee (t_{ic'(k+1)} \wedge s_{q'(k+1)} \wedge h_{(i \pm 1)(k+1)})$$

- acceptance:

$$\bigvee s_{qk}$$

<u>This formula can be easily generated in logarithmic space.</u>

NP-completeness

There is a long list of **NP**-complete problems:

- Hamiltonian path problem
- Traveling salesman problem
- Clique problem
- Knapsack problem
- Subgraph isomorphism problem
- Subset sum problem
- Vertex cover problem
- Independent set problem
- Dominating set problem
- Graph coloring problem

NP-hardness shown by reduction from some other **NP**-complete problem (e.g., from SAT).

Theorem

If L_1 reduces to L_2, and L_2 reduces to L_3, then L_1 reduces to L_3.

Proof

Functions computable in logarithmic space can be composed.

P-completeness of HORNSAT

HORNSAT problem: satisfiability of CNF formulas in which every clause has at most 1 positive literal

e.g., $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge x_2 \wedge (\neg x_1 \vee \neg x_2)$ is of this form

formulas of this form can be seen as implications (without alterna-tives on the right): $(x_2 \wedge x_3 \rightarrow x_1) \wedge (\top \rightarrow x_2) \wedge (x_1 \wedge x_2 \rightarrow \perp)$

e.g., $(x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)$ is not of this form

(there is an alternative on the right of an implication)

Theorem

The HORNSAT problem is **P**-complete.

P-completeness of HORNSAT

HORNSAT problem: satisfiability of CNF formulas in which every clause has at most 1 positive literal

e.g., $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge x_2 \wedge (\neg x_1 \vee \neg x_2)$ is of this form

formulas of this form can be seen as implications (without alterna-tives on the right): $(x_2 \wedge x_3 \rightarrow x_1) \wedge (\top \rightarrow x_2) \wedge (x_1 \wedge x_2 \rightarrow \bot)$

e.g., $(x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)$ is not of this form

(there is an alternative on the right of an implication)

<u>Theorem</u>

The HORNSAT problem is **P**-complete.

<u>Proof</u>

HORNSAT is in **P**: saturation (as in Prolog) – initially, we suppose that all variables are false; then we change false to true according implications in the formula

P-completeness of HORNSAT

HORNSAT problem: satisfiability of CNF formulas in which every clause has at most 1 positive literal

e.g., $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge x_2 \wedge (\neg x_1 \vee \neg x_2)$ is of this form

formulas of this form can be seen as implications (without alterna-tives on the right): $(x_2 \wedge x_3 \rightarrow x_1) \wedge (\top \rightarrow x_2) \wedge (x_1 \wedge x_2 \rightarrow \bot)$

e.g., $(x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)$ is not of this form

(there is an alternative on the right of an implication)

Theorem

The HORNSAT problem is **P**-complete.

Proof

HORNSAT is in **P**: saturation (as in Prolog) – initially, we suppose that all variables are false; then we change false to true according implications in the formula

P-hardness: if a machine is deterministic, the formula from the previous proof is (almost) in the HORN-CNF form

(an alternative of positive literals was appearing only while choosing a transition)

polyL-completeness

Tutorials: the class **polyL** has no complete problems.

Corollary: **P≠polyL**

- however, we don't know any specific problem on which they differ
- we do don't even know whether they are incomparable,
  or whether some of them is contained in the other

L-completeness

Almost every language in **L** is complete
(except the empty language, and the language containing all words)

NL-completeness

Theorem

Reachability in a directed graph is **NL**-complete

NL-completeness

<u>Theorem</u>

Reachability in a directed graph is **NL**-complete

<u>Proof</u>

It belongs to **NL**: we just walk in the graph

Hardness:

- Let L be recognized by a nondeterministic machine M working in logarithmic space
- we can assume that at the end M erases the contents of the tape, so that there is only one accepting configuration
- we get a word w of length n, we want to construct a graph
- as nodes we take configurations (there are polynomially many, as they are of logarithmic size)
- for every configuration, it is easy to write (in **L**) its successors,
- it is also easy to enumerate (in **L**) all configurations
- question to REACHABILITY: is there a path from the initial configuration (for word w) to the accepting configuration?