Computational complexity

lecture 7

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

- 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 $f\colon \Sigma^* \to \Gamma^*$ 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 <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 $w \in \Sigma^*$.

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

<u>Proof</u>

- We have a machine computing f.
- We treat it as 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.

<u>Definition</u>:

- It is a reduction between relations $R_1,R_2 \subseteq \Sigma^* \times \Sigma^*$
- R_1 is <u>Levin-reducible</u> 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 is 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 $\underline{C\text{-complete}}$ (with respect to logarithmic-space Karp reductions) if

- $L \in C$, and
- L is $\underline{C\text{-hard}}$, 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 $x_1=1, x_2=0$

<u>Theorem</u> (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

$\neg s_{qk} \vee \neg s_{q'k}$ 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

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

- 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

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

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

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

- 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):

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

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 alternatives 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.

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-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.

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

Theorem

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

Proof

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?

PSPACE-completeness of QBF

<u>QBF problem</u>

input: boolean formula $\phi(x_1,\ldots,x_n)$ with variables $x_1,\ldots,x_n$

question: is the following sentence true:

$$\exists x_1 \forall x_2 \exists x_3 \forall x_4 \ldots \phi(x_1,\ldots,x_n)$$

<u>Theorem</u>

The QBF problem is **PSPACE**-complete.

(the problem remains **PSPACE**-complete even if we require that ϕ is in the CNF)

PSPACE-completeness of QBF

QBF problem

input: boolean formula $\phi(x_1,\ldots,x_n)$ with variables $x_1,\ldots,x_n$

question: is the following sentence true:

$$\exists x_1 \forall x_2 \exists x_3 \forall x_4 \ldots \phi(x_1,\ldots,x_n)$$

Theorem

The QBF problem is **PSPACE**-complete.

(the problem remains **PSPACE**-complete even if we require that ϕ is in the CNF)

Proof

QBF is in **PSPACE**: we browse all possible valuations in lexico-graphic order... (backtracking)

for a fixed valuation, obviously we can compute the value of ϕ in **PSPACE**

PSPACE-completeness of QBF

Theorem

The QBF problem ($\exists x_1 \forall x_2 \exists x_3 \forall x_4 \ldots \phi(x_1,\ldots,x_n)$) is **PSPACE**-complete.

Proof (**PSPACE**-hardness)

- A similar trick as in the Savitch theorem.
- Let L be a language recognized by a machine M working in polynomial space
- having an input word w of length n, we want to construct a formula

PSPACE-completeness of QBF

Theorem

The QBF problem ($\exists x_1 \forall x_2 \exists x_3 \forall x_4 \ldots \phi(x_1,\ldots,x_n)$) is **PSPACE**-complete.

Proof (**PSPACE**-hardness)

- A similar trick as in the Savitch theorem.
- Let L be a language recognized by a machine M working in polynomial space
- having an input word w of length n, we want to construct a formula
- configurations of M can be encoded in $p(n)$ bits, for some polynomial p
- for every i we will write a formula $\psi_i(x_1,\ldots,x_{p(n)},y_1,\ldots,y_{p(n)})$ saying that from the configuration $x_1,\ldots,x_{p(n)}$ it is possible to reach the configu-ration $y_1,\ldots,y_{p(n)}$ in at most 2^i steps of M
- at the very end, it is enough to check whether the formula $\psi_{p(n)}(x_1,\ldots,x_{p(n)},y_1,\ldots,y_{p(n)})$ is true, where $x_1,\ldots,x_{p(n)}$ encodes the initial configuration, and $y_1,\ldots,y_{p(n)}$ encodes the accepting configuration (we can assume that it is fixed, or we can add some existential quantification)

PSPACE-completeness of QBF

Theorem

The QBF problem ($\exists x_1 \forall x_2 \exists x_3 \forall x_4 \ldots \phi(x_1,\ldots,x_n)$) is **PSPACE**-complete.

Proof (**PSPACE**-hardness)

- for every i we want to write a formula $\psi_i(x_1,\ldots,x_{p(n)},y_1,\ldots,y_{p(n)})$ saying that from the configuration $x_1,\ldots,x_{p(n)}$ it is possible to reach the configuration $y_1,\ldots,y_{p(n)}$ in at most 2^i steps of M
- For $i=0$, either the configurations are equal, or M performs a single step between them – this can be easily written using a formula (as while proving that SAT is **NP**-hard)
- The formula can be easily generated in logarithmic space

PSPACE-completeness of QBF

Theorem

The QBF problem ($\exists x_1 \forall x_2 \exists x_3 \forall x_4 \ldots \phi(x_1,\ldots,x_n)$) is **PSPACE**-complete.

Proof (**PSPACE**-hardness)

- for every i we want to write a formula $\psi_i(x_1,\ldots,x_{p(n)},y_1,\ldots,y_{p(n)})$ saying that from the configuration $x_1,\ldots,x_{p(n)}$ it is possible to reach the configuration $y_1,\ldots,y_{p(n)}$ in at most 2^i steps of M
- For $i=0$, either the configurations are equal, or M performs a single step between them – this can be easily written using a formula (as while proving that SAT is **NP**-hard)
- The formula can be easily generated in logarithmic space
- A naive idea for $i>0$: $\psi_{i+1}(x,y)=\exists z.(\psi_i(x,z) \wedge \psi_i(z,y))$
- This does not work, since the formula grows exponentially

PSPACE-completeness of QBF

Theorem

The QBF problem ($\exists x_1 \forall x_2 \exists x_3 \forall x_4 \ldots \phi(x_1,\ldots,x_n)$) is **PSPACE**-complete.

Proof (**PSPACE**-hardness)

- for every i we want to write a formula $\psi_i(x_1,\ldots,x_{p(n)},y_1,\ldots,y_{p(n)})$ saying that from the configuration $x_1,\ldots,x_{p(n)}$ it is possible to reach the configuration $y_1,\ldots,y_{p(n)}$ in at most 2^i steps of M
- For $i=0$, either the configurations are equal, or M performs a single step between them – this can be easily written using a formula (as while proving that SAT is **NP**-hard)
- The formula can be easily generated in logarithmic space
- A naive idea for $i>0$: $\psi_{i+1}(x,y)=\exists z.(\psi_i(x,z)\wedge\psi_i(z,y))$
- This does not work, since the formula grows exponentially
- One has to use ψ_i only once:

$$\psi_{i+1}(x,y)=\exists z.\forall r.\forall t.((r=x\wedge t=z)\vee(r=z\wedge t=y)\rightarrow\psi_i(r,t))$$

- This is not in QBF, but quantifiers from ψ_i can be moved to the front of the formula (assuming that variable names are unique)

PSPACE-completeness of QBF

The QBF problem ($\exists x_1 \forall x_2 \exists x_3 \forall x_4 \ldots \phi(x_1,\ldots,x_n)$) is **PSPACE**-complete.

Proof (**PSPACE**-hardness)

- for every i we want to write a formula $\psi_i(x_1,\ldots,x_{p(n)},y_1,\ldots,y_{p(n)})$ saying that from the configuration $x_1,\ldots,x_{p(n)}$ it is possible to reach the configuration $y_1,\ldots,y_{p(n)}$ in at most 2^i steps of M
- For $i=0$, either the configurations are equal, or M performs a single step between them – this can be easily written using a formula (as while proving that SAT is **NP**-hard)
- The formula can be easily generated in logarithmic space
- One has to use ψ_i only once:

$$\psi_{i+1}(x,y)=\exists z.\forall r.\forall t.((r=x \wedge t=z) \vee (r=z \wedge t=y) \rightarrow \psi_i(r,t))$$

- This is not in QBF, but quantifiers from ψ_i can be moved to the front of the formula (assuming that variable names are unique)
- Again, this can be easily created in logarithmic space: first comparisons of appropriate variables, then ψ_0
- Remark: for **PSPACE** one usually relaxes the definition of hardness, and allows for reductions in **P** (instead of “in **L**”)

Complete problems – summary

NP – SAT, Hamiltonian cycle, clique, subset sum, dominating set, ...

P – HORNSAT

polyL – no complete problems

L – almost every language is complete

NL – reachability in directed graphs

PSPACE - QBF

It is enough to solve a complete problem

Fact

If a C-complete problem is in class D (and D is closed under composition with functions computable in **L**), then $C \subseteq D$

Proof – obvious

Corollary:

If reachability in directed graphs is in **coNL**, then **NL**=**coNL**

If SAT is in **P**, then **P**=**NP**

etc.

Plan for the nearest future

- **NL=coNL**
- existence of **NP**-intermediate problems
- difficult problems that are not **NP**-hard
- relativisation and the Baker-Gill-Solovay theorem
- decision problems vs search problems
- polynomial hierarchy
- alternating machines
- probabilistic machines

NL=coNL

<u>Theorem</u> Immerman-Szelepcseny (1987)
Unreachability in directed graphs is in **NL**.

Thus **NL**=**coNL**, since reachability in directed graphs is **NL**-complete.

<u>Remark</u>
Reachability in <u>undirected</u> graphs is in **L** (Reingold, 2004)
(this is a rather difficult theorem)

<u>Previous lecture</u>: **PSPACE**=**NPSPACE**=**coNPSPACE**

NL=coNL

Theorem Immerman-Szelepcseny (1987)

Unreachability in directed graphs is in **NL**.

Proof

- Idea: in **NL** we can not only check reachability, but also count reachable nodes

NL=coNL (∗)

<u>Theorem</u> Immerman-Szelepcseny (1987)

Unreachability in directed graphs is in **NL**.

<u>Proof</u>

- Idea: in **NL** we can not only check reachability, but also count reachable nodes
- First consider such an algorithm in **NL**: given two numbers k and q, output q different nodes reachable from node s in $\leq k$ steps, and accept (if there are less such nodes, reject)
- Solution: a loop – set a counter to 0, then for every node v in the graph, nondeterministically: either ignore v, or guess a path of length $\leq k$ from s to v, output v, and increase the counter

NL=coNL (∗)

<u>Theorem</u> Immerman-Szelepcseny (1987)

Unreachability in directed graphs is in **NL**.

<u>Proof</u>

- We can: given k and q, output q different nodes reachable from s in $\leq k$ steps, and accept (if there are less such nodes, reject)
- Main trick: using this algorithm, we will compute (by induction) q_k – a number of nodes reachable from s in $\leq k$ steps
- $q_0=1$
- Given q_k we compute q_{k+1} as follows:
  - ➔ set q_{k+1} to 1 (we include s)
  - ➔ for every other node v, output q_k nodes reachable in $\leq k$ steps from s; if among them there is a node u such that (u,v) is an edge, then increase q_{k+1} (we do not store the whole list of q_k nodes; we rather check the condition on-the-fly)
- It is now easy to finish: compute q_n, output all q_n nodes reachable in $\leq n$ steps, and check that the target node does not appear

NL=coNL

Question: why cannot we prove in a similar way that **NP=coNP**?
E.g., that SAT is in **coNP**?

NL=coNL

Question: why cannot we prove in a similar way that **NP=coNP**? E.g., that SAT is in **coNP**?

- The proof is based on counting: in **NL** we can not only check reachability, but also count (and enumerate) reachable nodes.
- However, in polynomial time, even nondeterministically, we cannot count all valuations satisfying a given formula – there are exponentially many of them, so if we would like to count them “one-by-one”, polynomial time is not enough.

NL=coNL

Corollary from the Immerman-Szelepcseny theorem:

for every space-constructible function $S(n) \geq log(n)$

NSPACE($S(n)$)=**coNSPACE**($S(n)$)

Proof: on tutorials

We use a technique called *padding*