Complete problems for other complexity classes

PSPACE-complete and EXPTIME-complete problems

- A problem A is PSPACE-**hard** if for every problem A' from PSPACE there is a polynomial time reduction of A' to A.
- A problem A is PSPACE-**complete** if it is PSPACE-hard and belongs to PSPACE.

- A problem A is EXPTIME-**hard** if for every problem A' from EXPTIME there is a polynomial time reduction of A' to A.
- A problem A is EXPTIME-**complete** if it is EXPTIME-hard and belongs to EXPTIME.

$\vdots$

PSPACE-complete and EXPTIME-complete problems

Generally, for arbitrary complexity class $\mathcal{C}$ we can introduce classes of $\mathcal{C}$-**hard** and $\mathcal{C}$-**complete** problems:

Definition

- A problem A is $\mathcal{C}$-**hard** if for every problem A' from the class $\mathcal{C}$ there is a polynomial time reduction of A' to A.
- A problem A is $\mathcal{C}$-**complete** if it is $\mathcal{C}$-hard and belongs to the class $\mathcal{C}$.

So in addition to NP-complete problems, we have PSPACE-complete problems, EXPTIME-complete problems, EXPSPACE-complete problems, 2-EXPTIME-complete problems, ...

Generally speaking, $\mathcal{C}$-complete problems are always the hardest problems in the given class $\mathcal{C}$.

P-complete problems, NL-complete problems, . . .

Remark: The notions of $\mathcal{C}$-hard and $\mathcal{C}$-complete problems defined as above, where a polynomial time reductions are used, do not make much sense for the class P and other classes, which are subsets of this class (such as NL).

For such classes, the notions of $\mathcal{C}$-hard and $\mathcal{C}$-complete problems are defined in a similar way as before but instead of polynomial time reductions they use so called **logspace reductions**:

- an algorithm performing the given reduction must be deterministic and to have a logarithmic space complexity

For example, the following classes are defined this way:

- P-complete and P-hard problems
- NL-complete and NL-hard problems

Classes co-NP, co-NL, ...

For those classes where whether a given problem belongs or does not belong to a given class depends on existence or nonexistence of a **deterministic** algorithm solving this problem, it holds that:

- a problem A belongs to the given class iff its complement problem $\bar{A}$ belongs to the class.

  An algorithm solving problem $\bar{A}$ is obtained from an algorithm solving problem A by simply negating the answer of the original algorithm.

But this need not be the case for classes that refer to existence of **nondeterministic** algorithms — such as classes NP, NL, or NEXPTIME.

Classes co-NP, co-NL, ...

Therefore there are introduced classes such as:

- co-NP — the class consisting of exactly those problems that are complement problems of problem in the class NP
- co-NL — the class consisting of exactly those problems that are complement problems of problems in the class NL
- ...

For example, the class co-NP consists of those problems, for which:

- There exists a nondeterministic algorithm with polynomial time complexity accepting exactly those inputs, for which the answer is No.
- For inputs, for which the answer is No, there exist witnesses of polynomial size that can be checked by a deterministic polynomial algorithm.

co-NP-complete problems

In a similar way how NP-hard and NP-complete problems are defined, we can also define co-NP-hard and co-NP-complete problems.

Examples of co-NP-complete problems:

UNSAT

Input: Boolean formula φ.

Question: Is formula φ unsatisfiable?

The complement problem to problem IS (independent set)

Input: Undirected graph G and number k.

Question: Is it the case that the graph G does not contain any independent set of size k ?

Remark: Reduction showing co-NP-hardness of complement problems are exactly the same reductions that show NP-hardness of original problems.

co-NP-complete problems

An example of a problem where it is more natural to define it in a form that belongs to class co-NP:

TAUTOLOGY

Input: Boolean formula φ.

Question: Is formula φ a tautology?

Again, this is an example of co-NP-complete problem.

Class PSPACE

Class PSPACE

Let us recall the definition of the class PSPACE:

A decision problem A belongs to the class PSPACE iff there exists an algorithm with **polynomial space** complexity (i.e., an algorithm with space complexity $\mathcal{O}(f(n))$, where $f(n)$ is a polynomial) that solves it.

Let us recall Savitch's theorem:

- For each **nondeterministic** algorithm with space complexity $f(n)$, it is possible to construct a **deterministic** algorithm with space complexity $\mathcal{O}(f(n)^2)$ that gives answer YES for exactly those inputs, for which there exists an accepting computation of the original nondeterministic algorithm.

Therefore, to show that a given problem A belongs to PSPACE, it is sufficient to show a **nondeterministic** algorithm with a polynomial space complexity.

Transition systems

Let us say that we have a system where:

- The system can be in any state from a given set of **states**.

  Let us denote this set of states *States*.

- It is possible to from one state to another using a **transition**.

  Let *Transitions* be the set of those transitions, i.e., those pairs (α, β) where $\alpha, \beta \in$ *States* and it is possible to go from state α to state β.

  The notation

  $$\alpha \longrightarrow \beta$$

  stands for $(\alpha, \beta) \in$ *Transitions*.

Transition systems

- We say that a state α_n is **reachable** from a state α_0 if there exists a finite sequence $\alpha_0, \alpha_1, \ldots, \alpha_n$, where $n \geq 0$, and where

$$\alpha_0 \longrightarrow \alpha_1 \longrightarrow \alpha_2 \longrightarrow \quad \cdots \quad \longrightarrow \alpha_{n-1} \longrightarrow \alpha_n$$

  The fact that state β is reachable from state α is denoted by the notation

$$\alpha \longrightarrow^* \beta$$

- Let us assume we have some **initial state** α_0 and some **final state** α_f.

  We can ask whether

$$\alpha_0 \longrightarrow^* \alpha_f$$

  (We could also consider more general case where we have a **set of initial states** and a **set of final states**.)

Transition systems

Such system can be viewed as a **directed graph**:

- **nodes** — the elements of the set *States*
- **edges** — the transitions from the set *Transitions*

Reachability corresponds to an existence of a path between given nodes.
I.e., $\alpha \longrightarrow^* \beta$ holds iff there is a path from α to β in the given graph.

Transition systems

Such system need not be given explicitly, i.e., by listing all states and transitions, but it can be described **implicitly** by some other way, e.g.:

- **States** cound be represented by some finite objects (for example as words over some alphabet Σ).

  Moreover, we can have an algorithm that determines for a given object whether this object represents a state of the given system or not(e.g., if a given word is a represntation of a state from the set *States*).

- **Transitions** could be represented by an algorithm that determines for each pair $\alpha, \beta \in \mathit{States}$ whether

$$\alpha \longrightarrow \beta$$

- **Initial states** can be represented by an algorithm that for each state α determines whether it is an initial state or not.

- **Final states** can be represented by an algorithm that for each state α determines whether it is a final state or not.

Transition systems

Consider now such system where we also assume that:

- States could be represented by objects of **polynomial** size where it is possible to check by an algorithm with polynomial space if it is a state from the set *States* or not.

  So there can exponentially many states but a polynomial space is sufficient to store one state.
- There are algorithms with polynomial space complexity that allow to check for each states α, β whether:
  - if α and β are the same state, i.e.. if $\alpha = \beta$
  - if $\alpha \longrightarrow \beta$
  - if α is an initial state
  - if α is a final state
- There is a way how to generate, using a polynomial amount of memory, all states from the set *States*, i.e., we have an algorithm with polynomial space complexity that for a describion of a state α computes the next state α' .

Transition systems

For such system, we can easily construct a **nondeterministic** algorithm with polynomial space complexity solving the problem whether some final state is reachable from some initial state,
i.e., if there exist some initial state α_0 and some final state α_f,
for which we have $\alpha_0 \longrightarrow^* \alpha_f$:

- The algorithm will rememeber:
  - A current state α
  - A value of the counter c — how many steps can be done yet (the counter c will be represented in binary)
- At the beginning, α is initialized by some nondeterministically chosen initial state α_0.

  The counter will be initialized to a value that will be greater or equal to $|States| - 1$.

  Since there is at most an exponential number of states, a polynomial number of bits will be sufficient to represent a value of the counter.

Transition systems

The algorithm will perform the following in a cycle:

- If α is a final state, it halts with answer YES.
- If the value of the counter c is 0, it halts with answer NO.
- It nondeterministically chooses a state α' such that $\alpha \longrightarrow \alpha'$.
- Is sets the value α to α'.
- The counter c is decremented by 1.
- It continues with the next interation.

So the algorithm nondeterministically guesses a path in the graph, while remembering only a current state.

It is obvious that a polynomial space is sufficient for this algorithm.

Transition systems

We can use Savitch's theorem and derive from it that there exists a **deterministic** algorithm with polynomial space complexity solving the given problem.

Alternatively, it is possible to directly construct a **deterministic** algorithm based on the same idea that is used in the proof of Savitch's theorem.

So this solution does not need to use to Savitch's theorem.

The main part of the algorithm is a recursive function

$$\textsc{Reach}(i, \alpha, \beta)$$

that returns a boolean value as the return value:

- it returns $\textsc{True}$ iff there exists a path from α to β of length at most 2^i

Transition systems

```
REACH (i, α, β):
    if i = 0 then
        if (α = β or α ⟶ β) then return True
    else
        for each γ ∈ States do
            if REACH(i − 1, α, γ) and REACH(i − 1, γ, β) then
                return True
    return False
```

So it is sufficient to call function $\textsc{Reach}(h, \alpha, \beta)$ for all initial states α and all final states β, with a number h such that

$$2^h \geq |States| - 1$$

Transition systems

It is obvious that:

- For each call of the function REACH, a polynomial amount of memory is sufficient to store its arguments and local variables.
- the initial value of variable i, the value h, is in $\mathcal{O}(\log |States|)$.

  The depth of recursive calls of function REACH is equal to h, and so it is also in $\mathcal{O}(\log |States|)$.

  This value is at most polynomial.

So we can see that this deterministic algorithm has a polynomial space complexity.

Pebble game

Consider the following **pebble game**:

- We have a directed acyclic graph $G = (V, E)$where one of the nodes (denoted t) is selected as a target.
- We have k pebbles. These pebbles can be put on nodes of the graph.
- At the beginning, the graph is empty.
- The steps can be performed according to the following rules:
  - If a node v is empty and all predecessors of v (i.e., all nodes v' such that there is an edge from v' to v) contain pebbles, it is possible to put a pebble on node v.
  - If a node v is empty and all predecesors of v contain pebbles, a pebble from one of there predecesors can be moved to node v.
  - A pebble lying on a node of graph G can be taken away.

Pebble game

- The goal is to find a sequence of moves such that after these moves, a pebble is put on the target node t, or to find out that there is no such sequence.

Remark: This is not a game in a proper sanse since there are not two players playing againts each other. It is more like a puzzle.

The problem can be formulated as follows:

Pebble game

Input: A directed acyclic graph G with a denoted target node t, and a number of pebbles k.

Question: Is there a sequence of moves such that at the end of this sequence, a pebble is put on the node t ?

Pebble game

The pebble game can be viewed as a transition system where:

- The **states** correspond to all possibilities how up to k pebbles can be put on n nodes of the graph G.
- The **transitions** are given by the rules that specify how pebbles could be put, moved, or removed.
- The **initial state** corresponds to the situation where graph G contains no pebbles.
- The **final states** are those situations where a pebble is on the node t.

Pebble game

- States can be represented as n-tuples of bits where the value of i-th bit specifies whether a node with number i contains a pebble or not.

  (We assume that nodes are denoted by numbers $0, 1, \ldots, n-1$.)
- It is obvious that the following tests can be done in time proportional to the size of graph G:
  - a test whether states α and β are the same (i.e., if $\alpha = \beta$)
  - a test whether it is possible to go from state α to state β in one step
  - a test if α is an initial state
  - a test if α is an final test

We can see that:

Theorem

The "Pebble game" problem is in PSPACE.

Equivalence of nondeterministic finite automata

Equivalence of NFA

Input: Nondeterministic finite automata $\mathcal{A}_1$ and $\mathcal{A}_2$.

Question: Is $\mathcal{L}(\mathcal{A}_1) = \mathcal{L}(\mathcal{A}_2)$?

This problem can be solved as follows:

- For given **nondeterministic** finite automata $\mathcal{A}_1$ and $\mathcal{A}_2$, we can construct equivalent **deterministic** finite automata $\mathcal{A}'_1$ and $\mathcal{A}'_2$, and for these deterministic finite automata we will try to find a distinguishing word that is accepted by one of these automata and rejected by the other.

  If there exists such a distinguishing word, we have $\mathcal{L}(\mathcal{A}_1) \neq \mathcal{L}(\mathcal{A}_2)$ (and the answer is NO), if not, then $\mathcal{L}(\mathcal{A}_1) = \mathcal{L}(\mathcal{A}_2)$ (and the answer is YES).

Equivalence of nondeterministic finite automata

When a nondeterministic finite automaton is transformed to deterministic, the number of states can increase exponentially:

- The states of the deterministic automaton correspond to **subsets** of the set of states of the original nondeterministic automaton, and there can be up to 2^n such subsets where n is the number of the states of the original automaton.

So this approach is of an exponential space complexity.

Equivalence of nondeterministic finite automata

But it is not necessary to keep whole deterministic automata $\mathcal{A}'_1$ and $\mathcal{A}'_2$ is memory.

Moreover, we can use nondeterminism when we are looking for a distinguishing word:

- The algorithm remembers only current states of deterministic automata $\mathcal{A}'_1$ and $\mathcal{A}'_2$
  (where the first of them is a subset of the set of states of automaton $\mathcal{A}_1$ and the second a subset of the set of states of automaton $\mathcal{A}_2$).
- The algorithm nondeterministically guesses symbols of a distinguishing word.

  In one step, it guesses a following symbol, and simulates reading of this symbol on the states of automata $\mathcal{A}'_1$ and $\mathcal{A}'_2$.

Equivalence of nondeterministic finite automata

It is obvious that a polynomial amount of memory is sufficient for this nondeterministic algorithm.

Using Savitch's theorem, this nondeterministic algorithm can be transformed into a deterministic algorithm with a polynomial space complexity.

So we have the following:

Theorem

Problem "Equivalence of NFA" is in PSPACE.

Remark: Alternatively, it is possible not to refer to Savitch's theorem but instead, to construct a corresponding deterministic algorithm with polynomial space complexity directly.

Equivalence of nondeterministic finite automata

It follows from this that also the following problem (that can be viewed as a special case of "'Equivalence of NFA") belongs to PSPACE:

Universality of NFA

Input: A nondeterministic finite automaton $\mathcal{A}$.

Question: Is $\mathcal{L}(\mathcal{A}) = \Sigma^*$?

Moreover, this problem is PSPACE-**hard** and so PSPACE-**complete**.

From this we immediately obtain that the problem "Equivalence of NFA" is PSPACE-**complete**.

Theorem

Problems "Equivalence of NFA" and "Universality of NFA" are both PSPACE-complete.

Equivalence of nondeterministic finite automata

The proof of PSPACE-hardness of "Universality of NFA" can be done in such a way that we show for every problem A from PSPACE how to construct a polynomial reduction from A to the complement problem of "Universality of NFA":

- Let us assume that problem A belongs to PSPACE.
- So there exists a one-tape deterministic Turing machine $\mathcal{M} = (Q, \Sigma, \Gamma, \delta, q_0, F)$ that solves problem A, and a polynomial $p(n)$ such that if machine $\mathcal{M}$ obtains as input a word w of size n, then all configurations, through which the machine $\mathcal{M}$ goes during the computation on w, are of size at most $p(n)$.

  These configurations can be written as words over the alphabet $\Delta' = \Gamma \cup (Q \times \Gamma)$ of size exactly $p(n)$.

Equivalence of nondeterministic finite automata

- A computation of machine $\mathcal{M}$ on word w then can be represented as a word obtained by concatenating of words representing individual configurations where these configurations are separated by #.
- The machine $\mathcal{M}$ gives answer YES for input w iff its computation over this word halts in an **accepting** configuration where the state of the control unit will be the accepting final state q_{acc}.
- For the given Turing machine $\mathcal{M}$ and word w we can construct a nondeterministic finite automaton $\mathcal{A}$ that accepts exactly those words over alphabet $\Delta = \Delta' \cup \{\#\}$ that **are not** representations of an accepting computation of machine $\mathcal{M}$ over word w:
  - if $\mathcal{M}$ accepts w, then $\mathcal{L}(\mathcal{A}) \neq \Delta^*$
  - if $\mathcal{M}$ does not accept w, then $\mathcal{L}(\mathcal{A}) = \Delta^*$

Equivalence of nondeterministic finite automata

Those words over alphabet Δ that **are not** correct representations of an accepting computation of machine $\mathcal{M}$ over word w where every configuration is written as a word of length $p(n)$, look as follows:

- they are not of the correct format, i.e., they are not a sequence of configurations where every configuration is of length $p(n)$, or
- they do not start with the initial configuration of machine $\mathcal{M}$ over input w, or
- they contain some pair of consecutive configurations α_i and α_{i+1} that do not correspond to the step that would be performed by machine $\mathcal{M}$ in configuration α_i, or
- they do not end with a configuration where the state of the control unit is q_{acc}.

All of them are properties that can be easily recognized by a nondeterministic finite automaton $\mathcal{A}$ whose size is polynomial with respect to the length of word w (which is the input of the machine $\mathcal{M}$), assuming $p(n)$ is a polynomial.

Equivalence of regular expressions

In a very similar way, it is possible to prove PSPACE-completeness of the following two problems.

Equivalence of regular expressions

Input: Regular expressions α_1 and α_2.

Question: Is $\mathcal{L}(\alpha_1) = \mathcal{L}(\alpha_2)$?

Universality of regular expressions

Input: A regular expression α.

Question: Is $\mathcal{L}(\alpha) = \Sigma^*$?

Equivalence of regular expressions

- To show that these problems are in PSPACE, it is sufficient to note that for every regular expression α, it is possible to construct a nondeterministic finite automaton $\mathcal{A}$ such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\alpha)$, and this construction can be done in polynomial time (and so also in polynomial space).

  So it is sufficient to transform the given regular expressions to nondeterministic finite automata and to use the algorithms described before that work with nondeterministic finite automata and are of polynomial space complexity.

- For the proof of PSPACE-hardness, a regular expression is constructed that describes a language consisting of exactly those words that are not a correct representation of an accepting computation of Turing machine $\mathcal{M}$ on word w.

Equivalence of regular expressions

Regular expressions with squaring are defined similarly as standard regular expressions but in addition to the standard operators $+$, $\cdot$, and *, they can contain unary operator ^{2} with the following meaning:

- α^2 is a shorthand for $\alpha \cdot \alpha$.

The following two problems are EXPSPACE-complete:

Equivalence of regular expressions with squaring

Input: Regular expressions with squaring α_1 and α_2.

Question: Is $\mathcal{L}(\alpha_1) = \mathcal{L}(\alpha_2)$?

Universality of regular expressions with squaring

Input: A regular expression with squaring α.

Question: Is $\mathcal{L}(\alpha) = \Sigma^*$?

Equivalence of regular expressions

- To show that these problems are in EXPSPACE can be done as follows:

  In a regular expression with squaring, each subexpression of the form α^2 can be replaced with an equivalent subexpression $\alpha \cdot \alpha$. This transformation increases the size of an expression at most exponentially.

  So when we use the algorithms with polynomial space complexity described before on these exponentially big regular expressions obtained by this transformation, we will obtain an algorithm with an exponential space complexity.

Equivalence of regular expressions

- The proof of EXPSPACE-hardness is very similar to the proof of PSPACE-hardness for regular expressions without squaring.

  A general reduction that can be used for every problem A from EXPSPACE is described.

  If a problem A is in EXPSPACE, there exists a one-tape deterministic Turing machine $\mathcal{M}$ that solves it, with a space complexity, which is bounded from above by a function of the form $2^{p(n)}$, where $p(n)$ is a polynomial.

  A computation of this machine $\mathcal{M}$ on word w of length n can be described as a sequence of configurations where each configuration can be written as a word over alphabet Δ of length $2^{p(n)}$.

  Again, we can construct a regular expression (now with squaring) describing exactly those words that are not representations of an accepting computation of the machine $\mathcal{M}$ on the word w.

Equivalence of regular expressions

It is possible to describe exponentially long sequences of symbols between pairs of corresponding triples of symbols in each consecutive pair of configurations with subexpressions of the form

$$((\cdots(((\alpha^2)^2)^2)\cdots)^2)^2$$

of polynomial size.

Quantified Boolean Formulas (QBF)

Quantified Boolean Formulas (**QBF**) are a generalization of standard boolean formulas.

These formulas are defined as follows:

- Formula of the form x, where x is a boolean variable, is a well-formed formula.
- Formulas of the form $\neg\varphi_1$, $\varphi_1 \wedge \varphi_2$, $\varphi_1 \vee \varphi_2$, $\varphi_1 \rightarrow \varphi_2$, $\varphi_1 \leftrightarrow \varphi_2$, where φ_1 and φ_2 are well-formed formulas, are well-formed formulas.
- Formulas of the form $\exists x.\varphi_1$ and $\forall x.\varphi_1$, where x is a boolean variable and φ_1 is a well-formed formula, are well-formed formulas.

Example:

$$\exists x_1.\forall x_2.\exists x_3.((\neg x_1 \vee x_2) \wedge (x_1 \vee \neg x_3))$$

Quantified Boolean Formulas (QBF)

- Boolean variables take values from the set $\mathsf{Bool} = \{0, 1\}$.
- A formula of the form $\exists x.\varphi$ can be viewed as a short way how to express the formula of the form

$$\varphi[0/x] \vee \varphi[1/x]$$

and similarly a formula of the form $\forall x.\varphi$ can be viewed as a short way how to express the formula of the form

$$\varphi[0/x] \wedge \varphi[1/x]$$

where notation $\varphi[b/x]$ denotes formula φ with free occurrences of variable x replaced with boolean constant b.

- The meaning of boolean connectives $\neg$, $\wedge$, $\vee$, $\rightarrow$, and $\leftrightarrow$ is the same as in propositional logic.

Quantified Boolean Formulas (QBF)

- Quantified boolean formulas can be also viewed as a special case of formulas of predicate logic where variables take values from the universe $\mathrm{Bool} = \{0, 1\}$.
- A formula φ is **closed** if it does not contain any free variables, i.e., if every occurrence of a boolean variable is bound by a quantifier.
- Truth value of formula φ depends on assignment of truth values to all boolean variables that have free occurrences in φ.

  So in the case of closed formulas, their truth value does not depend on an assignment of truth values to variables.
- We say a closed formula φ is **true** if its truth value is 1.

Quantified Boolean Formulas (QBF)

Consider the following decision problem:

Problem of quantified boolean formulas (QBF)

Input: A quantified boolean formula φ.

Question: Is formula φ true?

Remark: In literature, this problem problem is also denoted with the following names:

- **TQBF** — True Quantified Boolean Formulas
- **QSAT** — Quantified Satisfiability

Quantified Boolean Formulas (QBF)

- The problem of **satisfiability** of boolean formulas (**SAT**) can be viewed as a special case of QBF:

  Let φ be a standard boolean formula without quantifiers and assume that x_1, x_2, ..., x_n are all boolean variables occurring in the formula φ.

  It is obvious that formula φ is satisfiable iff the following formula is true:

$$\exists x_1.\exists x_2.\cdots.\exists x_n.\varphi$$

- In a similar way, we can express that a formula φ is a **tautology** by the following quantified boolean formula:

$$\forall x_1.\forall x_2.\cdots.\forall x_n.\varphi$$

Quantified Boolean Formulas (QBF)

It is not difficult to see that the problem QBF belongs to PSPACE:

The core of the algorithm is a recursively defined function

$$\textsc{Eval}(\varphi, \nu)$$

that obtains as arguments:

- φ — a quantified boolean formula that should be evaluated
- ν — a truth valuation assigning truth values to all boolean variables that can occur as free variables in formula φ

Remark: The notation

$$\nu[x \mapsto b]$$

denotes the truth valuation that is the same as a truth valuation ν, except that boolean value b is assigned to variable x.

Quantified Boolean Formulas (QBF)

Function $\textsc{Eval}(\varphi, \nu)$ distinguishes different cases depending on the form of formula φ.

If φ is of the form:

- x: **return** $(\nu(x))$
- $\neg\varphi_1$: **return** (**not** $\textsc{Eval}(\varphi_1, \nu)$)
- $\varphi_1 \wedge \varphi_2$: **return** ($\textsc{Eval}(\varphi_1, \nu)$ **and** $\textsc{Eval}(\varphi_2, \nu)$)
- $\varphi_1 \vee \varphi_2$: **return** ($\textsc{Eval}(\varphi_1, \nu)$ **or** $\textsc{Eval}(\varphi_2, \nu)$)
- $\varphi_1 \rightarrow \varphi_2$: **return** ((**not** $\textsc{Eval}(\varphi_1, \nu)$) **or** $\textsc{Eval}(\varphi_2, \nu)$)
- $\varphi_1 \leftrightarrow \varphi_2$: **return** ($\textsc{Eval}(\varphi_1, \nu)$ **iff** $\textsc{Eval}(\varphi_2, \nu)$)
- $\exists x.\varphi_1$: **return** ($\textsc{Eval}(\varphi_1, \nu[x \mapsto 0])$ **or** $\textsc{Eval}(\varphi_1, \nu[x \mapsto 1])$)
- $\forall x.\varphi_1$: **return** ($\textsc{Eval}(\varphi_1, \nu[x \mapsto 0])$ **and** $\textsc{Eval}(\varphi_1, \nu[x \mapsto 1])$)

Quantified Boolean Formulas (QBF)

- It is obvious that the depth of recursive calls of function EVAL is bounded from above by the size of formula φ.
- The total amount information that the algorithm must remember during a computation is obviously bounded by a polynomial.

The problem QBF belongs to PSPACE.

Quantified Boolean Formulas (QBF)

We will show now that the problem QBF is PSPACE-hard:

- We need to show that for every problem A from PSPACE there exists a polynomial reduction of problem A to QBF.
- So let us assume that A is a problem from PSPACE, i.e., that there exists an algorithm with polynomial space complexity that solves this problem.
- Let us assume that this algorithm is implemented in a form of some machine $\mathcal{M}$.

  For example, we can assume that $\mathcal{M}$ is a one-tape deterministic Turing machine.

  (This choice is not very important, and the algorithm for the reduction would work correctly also for other types of machines.)
- Let us assume that the size of configurations of machine $\mathcal{M}$ on word w is bounded from above by some polynomial $p(n)$ where n is the length of word w.

Quantified Boolean Formulas (QBF)

- Configurations of machine $\mathcal{M}$ in a computation on word w can be "encoded" using values of boolean variables.

  Let us assume that a configuration α will be encoded using values of boolean variables

  $$x_1, x_2, \ldots, x_r,$$

  where r is a natural number whose value is in $\mathcal{O}(p(n))$.

Quantified Boolean Formulas (QBF)

- If a configuration α is encoded using variables $x_1, x_2, \ldots, x_r$, the notations

$$\exists \alpha.\varphi \qquad\qquad \forall \alpha.\varphi$$

will be understood as shorthands for

$$\exists x_1.\exists x_2.\cdots.\exists x_r.\varphi \qquad\qquad \forall x_1.\forall x_2.\cdots.\forall x_r.\varphi$$

- Let us say that configuration α is encoded using variables $x_1, x_2, \ldots, x_r$, and configuration β using variables $y_1, y_2, \ldots, y_r$.

Formula $\mathrm{EQ}(\alpha, \beta)$ expresses that α and β are one and the same configuration:

$$(x_1 \leftrightarrow y_1) \wedge (x_2 \leftrightarrow y_2) \wedge \cdots \wedge (x_r \leftrightarrow y_r)$$

Quantified Boolean Formulas (QBF)

The behaviour of machine $\mathcal{M}$ will be represented by the following three formulas (where configuration α is represented for example by variables $x_1, x_2, \ldots, x_r$, and configuration β by variables $y_1, y_2, \ldots, y_r$)

— details of how precisely these formulas look will be similar as for example in the case of the proof of NP-hardness of problem SAT:

- $\textsc{Step}(\alpha, \beta)$ — expresses that machine $\mathcal{M}$ can go in one step from configuration α to configuration β
- $\textsc{Is-Init}(\alpha)$ — expresses that α is the initial configuration of machine $\mathcal{M}$ on input w
- $\textsc{Is-Acc}(\alpha)$ — expresses that α is an accepting final configuration of machine $\mathcal{M}$

Sizes of all these formulas are polynomial (or even linear) with respect to value $p(n)$.

Quantified Boolean Formulas (QBF)

If we assume that configurations are encoded by r boolean variables, it is obvious that the number of distinct configurations is at most 2^r.

Configurations can not repeat during a computation of machine $\mathcal{M}$ — this means that the length of such computation is at most 2^r.

We will create a sequence of formulas

$$\textsc{Reach}_0, \textsc{Reach}_1, \ldots, \textsc{Reach}_r$$

where each formula

$$\textsc{Reach}_i(\alpha, \beta)$$

where $0 \leq i \leq r$, will express that the machine $\mathcal{M}$ can go by at most 2^i steps from configuration α to configuration β.

Quantified Boolean Formulas (QBF)

These formulas are defined inductively:

- $\textsc{Reach}_0(\alpha, \beta)$: is defined as

$$\textsc{Eq}(\alpha, \beta) \vee \textsc{Step}(\alpha, \beta)$$

- $\textsc{Reach}_{i+1}(\alpha, \beta)$, where $i \geq 0$:

  simple direct definition could look like this

$$\exists \gamma.(\textsc{Reach}_i(\alpha, \gamma) \ \wedge \ \textsc{Reach}_i(\gamma, \beta))$$

  However, the problem with this definition is that the size of formulas $\textsc{Reach}_i$ here grows exponentially with the value of i.

  Equivalently, we can express the same thing as follows:

$$\exists \gamma. \forall \sigma_1. \forall \sigma_2. (\\ ((\textsc{Eq}(\sigma_1, \alpha) \wedge \textsc{Eq}(\sigma_2, \gamma)) \ \vee \ (\textsc{Eq}(\sigma_1, \gamma) \wedge \textsc{Eq}(\sigma_2, \beta))) \ \rightarrow \\ \textsc{Reach}_i(\sigma_1, \sigma_2))$$

Quantified Boolean Formulas (QBF)

- The resulting formula φ is constructed as follows:

$$\exists\alpha.\exists\beta.(\textsc{Is-Init}(\alpha) \wedge \textsc{Is-Acc}(\beta) \wedge \textsc{Reach}_r(\alpha,\beta))$$

- It is not difficult to check that the formula φ is true iff there exists an accepting computation of the machine $\mathcal{M}$ on the word w.
- It is also not difficult to check that the size of this formula is polynomial with respect to the length of the word w — this size is $\mathcal{O}(p(n)^2)$ where $n = |w|$.

  This formula can be easily constructed in polynomial time.

So we can see that QBF is PSPACE-hard problem.

So the proof of the following theorem is finished:

Theorem

Problem QBF is PSPACE-complete.

Quantified Boolean Formulas (QBF)

Let us recall that problem SAT is often used for proving of NP-hardness of different problems where NP-hardness of a problem A is shown by describing a polynomial reduction from SAT to A.

In a similar way, QBF is often used to show PSPACE-**hardness** of other problems:

- To show that a problem A is PSPACE-hard,
  is is sufficient to show polynomial reduction from QBF to A.

Quantified Boolean Formulas (QBF)

By providing a reduction from QBF, it is possible to prove PSPACE-hardness (and so also PSPACE-completeness) of the following problem described before:

Pebble game

Input: An acyclic directed graph G with a distinguished target node t, and number of pebbles k.

Question: Is there a sequence of moves such that at the end of this sequence a pebble is put on node t?

(This reduction is quite complicated, so we will not discuss it in detail here.)

Quantified Boolean Formulas (QBF)

In proofs of NP-hardness, SAT (where input is an arbitrary formula) can be used but it is often easier to use 3-SAT (a restricted variant of SAT):

- it is assumed that formulas are of a specific form
  — they are in conjunctive normal form where every clause has exactly 3 literals

In a similar way, it is ofter convenient to assume in a proof of PSPACE-hardness of some problem A done by a polynomial reduction from A to QBF that formula that is an instance of QBF is of a specific form:

$$\mathcal{Q}_1\, x_1 .\, \mathcal{Q}_2\, x_2 .\, \cdots\, \mathcal{Q}_n\, x_n \,.\, \psi$$

- $\mathcal{Q}_1$, $\mathcal{Q}_2$, ..., $\mathcal{Q}_n$ are quantifiers ($\exists$, $\forall$) that alternate, i.e., $\mathcal{Q}_i = \exists$ for i odd, and $\mathcal{Q}_i = \forall$ for i even
- the subformula ψ does not contain quantifiers and is in conjunctive normal form where every clause contains exactly 3 literals

Quantified Boolean Formulas (QBF)

Quantified formulas can be transformed to equivalent formulas in some specific form using **equivalent tranformations**.

For example, we can transform every formula φ to an equivalent formula φ' satisfying the following restrictions:

- the only logical connectives that are used are $\neg$, $\wedge$, and $\vee$ (it does not contain connectives $\leftrightarrow$ and $\rightarrow$):

$$A \leftrightarrow B \iff (A \rightarrow B) \wedge (B \rightarrow A)$$
$$A \rightarrow B \iff \neg A \vee B$$

- negations ($\neg$) are applied only to atomic formulas, i.e., on boolean variables (e.g., $\neg x$):

$$\neg\exists x.A \iff \forall x.\neg A$$
$$\neg\forall x.A \iff \exists x.\neg A$$

$$\neg\neg A \iff A$$
$$\neg(A \wedge B) \iff \neg A \vee \neg B$$
$$\neg(A \vee B) \iff \neg A \wedge \neg B$$

Quantified Boolean Formulas (QBF)

Moreover, the given quantified formula φ can be transformed by equivalent transformations to so called **prenex normal form**:

$$\mathcal{Q}_1 x_1 . \mathcal{Q}_2 x_2 . \cdots \mathcal{Q}_n x_n . \psi$$

where $\mathcal{Q}_1$, $\mathcal{Q}_2$, ..., $\mathcal{Q}_n$ are quantifiers ($\exists$, $\forall$) and where subformula ψ contains no quantifiers.

If x does not occur in formula A as a free variable, then we have for example:

$$\begin{array}{lcl} A \wedge \exists x.B & \iff & \exists x.(A \wedge B) \\ A \vee \exists x.B & \iff & \exists x.(A \vee B) \\ A \wedge \forall x.B & \iff & \forall x.(A \wedge B) \\ A \vee \forall x.B & \iff & \forall x.(A \vee B) \end{array}$$

(Occurrences of variables can be renamed in such a way that variables bound by different quantifiers have different names.
This will ensure that for example x does not occur in A as required above.)

Quantified Boolean Formulas (QBF)

In a formula in prenex normal form

$$\mathcal{Q}_1\, x_1.\, \mathcal{Q}_2\, x_2.\, \cdots\, \mathcal{Q}_n\, x_n\, .\, \psi$$

the subformula ψ (that does not contain quantifiers) can be transformed to subformula of the form

$$\exists y_1.\exists y_2.\cdots.\exists y_k.\psi'$$

where:

- ψ' is in **conjunctive normal form** where every clause contains exactly 3 literals
- $y_1, y_2, \cdots, y_k$ are new auxiliary variables correponding to individual subformulas of formula ψ

The construction is done is a similar way as in the reduction of SAT to 3-SAT.

Quantified Boolean Formulas (QBF)

A formula in a prenex normal form can be easily transformed to the form where quantifiers **alternate** regularly:

- it is sufficient to add new quantifiers with new variables that are not used anywhere else in the formula

Example: A formula of the form

$$\forall x_1.\forall x_2.\forall x_3.\exists x_4.\exists x_5.\exists x_6.\forall x_7.\forall x_8.\,\psi$$

can be transformed to formula

$$\exists y_1.\forall x_1.\exists y_2.\forall x_2.\exists y_3,\ \forall x_3.\exists x_4.\forall y_4.\exists x_5.\forall y_5\exists x_6.\forall x_7.\exists y_6.\forall x_8.\,\psi$$

where $y_1, y_2, \ldots, y_6$ are new, so far unused, variables.

Quantified Boolean Formulas (QBF)

Note that most of these transformations do not change the size of a given formula much:

- the size of formula φ' is linear with respect to the size of original formula φ
  (i.e., if the size of the original formula φ is n, then the size of the resulting formula φ' is in $\mathcal{O}(n)$)

The only problematic transformation where the size of the formula could increase exponentially, is the replacement of formula of the form $A \leftrightarrow B$ with formula

$$(A \rightarrow B) \wedge (B \rightarrow A)$$

The only occurrence of logical connective $\leftrightarrow$ in formulas that are created in the construction described in the proof of PSPACE-hardness of QBF, is in the formulas $\mathrm{EQ}(\alpha, \beta)$where it is applied only to atomic formulas.
So the size of such formulas increases only linearly by this transformation.