Classes L and NL

Logarithmic amount of memory

One specific kind of algorithms are algorithms that use extremely small amout of memory — asymptotically smaller than n where n is the size of an input.

In particular, we will concentrate here on problems with **logarithmic space complexity**, i.e., the space complexity $\mathcal{O}(\log n)$.

- It is obvious that an algorithm whose time complexity is smaller than n does not have enough memory to store whole input instance in memory.
- So for algorithms that work with such small amount of memory, the memory used to store an input is not counted into their space complexity.

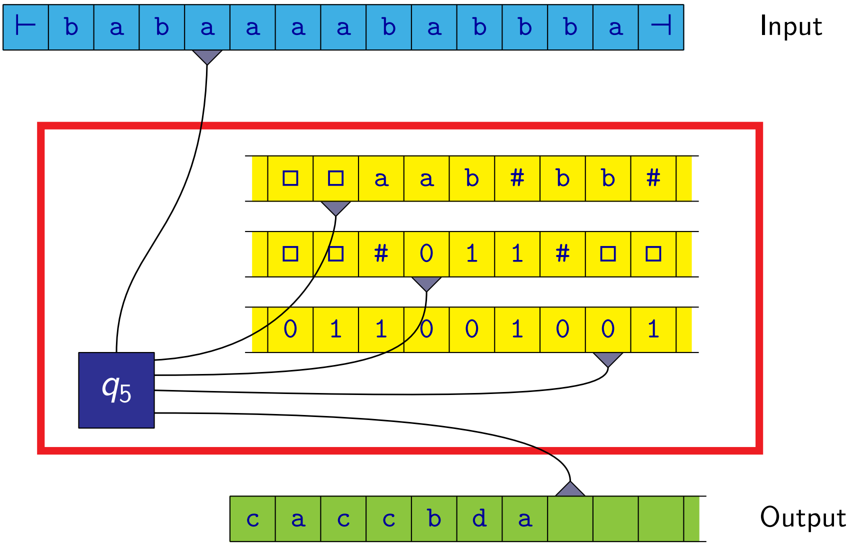
Logarithmic amount of memory

For such algorithms we assume that they are exectuted by a type of machine (e.g., a Turing machine) that has:

- **Input tape** — it contains an input word, delimeted from the left and from the right by special markers '$\vdash$' and '$\dashv$', the machine can not write on it (it is read-only), it has one head that can move in both directions
- **Output tape** — the machine can only write on it (it is write-only), it can not read from it, it is empty at the beginning of a computation, the head can move only from the left to the right
- **Working memory** — it can be read from it and written to it; e.g., in the case of Turing machines, it has a form of one or more tapes

The amount of used memory is given by the number of bits that are sufficient for storing the content of the **working memory** during computation.

Logarithmic amount of memory



Logarithmic amount of memory

If the size of an input is n, $\mathcal{O}(\log n)$ bits of memory are only sufficient to store some fixed finite number of values where each of them requires at most $\mathcal{O}(\log n)$ bits.

Using k bits, we can represent numbers in the interval from 0 to $2^k - 1$.

So logarithmic number of bits are sufficient to represent a number whose value is bounded by a polynomial (i.e., a number whose maximal value is $\mathcal{O}(n^c)$ where c is a constant).

By such numbers we can represent for example:

- an index of a cell on the input tape — basically a pointer to the input data
- a counter whose value is bounded by a polynomial
- in graph algorithms, for example an index of a node or an edge
- in algorithms working with matrices, for example an index of a row or a column

Logarithmic amount of memory

On the other hand, $\mathcal{O}(\log n)$ bits of memory are **not sufficient** to store things like:

- To store at least 1 bit of information (for exaple some flag) for each element from an input when the input consists of a sequence of n elements.
- In graph algorithms, to remember, which nodes have been visited.

Logarithmic amount of memory

Example: Consider problems where an input looks as follows:

Input: A pair of numbers x and y where these numbers are represented in binary as sequences of n bits.

There are algorithms with logarithmic space complexity for things like:

- the sum and the difference of numbers x and y (i.e., the values $x + y$ and $x - y$)
- the product of numbers x and y (i.e., the value $x \cdot y$)
- finding out whether $x = y$, $x < y$, $x \leq y$
- the maximum and minumum (the values $\max(x, y)$ and $\min(x, y)$)

Logarithmic amount of memory

Example: Consider problems where an input looks as follows:

> Input: A sequence of numbers $a_1, a_2, \dots, a_k$.

Let us say that n is the total number of bits necessary to represent numbers $a_1, a_2, \dots, a_k$.

There are algorithms with space complexity $\mathcal{O}(\log n)$ that can compute for example the following:

- to sort the elements from the smallest to the biggest
- the sum $a_1 + a_2 + \cdots + a_k$

Remark: Note that computing the sum of numbers $a_1, a_2, \dots, a_k$ in space $\mathcal{O}(\log n)$ is not a completely trivial problem since some of the numbers can require more than $\mathcal{O}(\log n)$ bits — consider for example the case where we have $\sqrt{n}$ numbers where each of these numbers has $\sqrt{n}$ bits.

Logarithmic amount of memory

Example: Also the following problem can be solved with space complexity $\mathcal{O}(\log n)$:

Matrix multiplication

Input: Matrices A and B whose elements are natural numbers.

Output: The matrix $A \cdot B$.

Remark: Similarly as in the previous case, the size of an input n is the total number of bits necessary to store matrices A and B (i.e., to write all their elements).

It is possible that some of these elements of these matrices have more than $\mathcal{O}(\log n)$ bits.

So this problem is not as simple as it may look at the first sight.

Logarithmic amount of memory

Also the following problem can be easily solved by a deterministic algorithm with space complexity $\mathcal{O}(\log n)$:

Input: A word w consisting of different kinds of parenthesis ($[_1$, $]_1$, $[_2$, $]_2$, $\ldots$, $[_r$, $]_r$).

Question: Is w a correctly parenthesised sequence?

A correctly parenthesised sequence here means a sequence belonging to the language generated by the following context-free grammar:

$$A \rightarrow \varepsilon \mid AA \mid [_1 A\,]_1 \mid [_2 A\,]_2 \mid \cdots \mid [_r A\,]_r$$

Logarithmic amount of memory

It is interesting that most of polynomial time reductions used for example in proofs of NP-hardness, PSPACE-completeness, etc., of different problems (that we have seen in the previous lectures or that are discribed in a literature) can in fact be implemented as a (deterministic) algorithm working with a logarithmic amount of memory.

Such reduction are called **logspace reductions**.

Logspace reductions

Definition

A **logspace reduction** of a decision problem A to a decision problem B is a deterministic algorithm Alg with space complexity $\mathcal{O}(\log n)$ that:

- It obtains an instance x of problem A as an input.
- It produces an instance y of problem B as an output.
- The answer for the instance y of problem B is $\textsc{Yes}$ iff the answer for the instance x of problem A is $\textsc{Yes}$.

Logspace reductions

Theorem

If there exist:

- a logspace reduction from problem A to problem B, and
- a logspace reduction from problem B to problem C,

then there is also:

- a logspace reduction from problem A to problem C.

Proof: Let us assume that:

- Alg_1 is a logspace reduction from problem A to problem B
- Alg_2 is a logspace reduction from problému B to problem C

The following simple construction, that works correctly for polynomial time reduction, does not work:

- to apply the reduction Alg_1 to an instance x of problem A, and then to apply the reduction Alg_2 to the resulting instance of problem B

Logspace reductions

The problem with this simple construction is that the resulting algorith is a reduction but not necessary a logspace reduction:

- An instance y of problem B constructed by the reduction Alg_1 can be of a polynomial size with respect to the size of the original instance x of problem A
  — a working memory of logarithmic size is not sufficient for storing this instance y

Logspace reductions

It is necessary to use a different approach — an algorithm transforming an instance of problem A to an instace of problem C will work as follows:

- It will simulate a computation of the algorithm Alg_2.
- It will remember the position of its head on its input tape — this position will be represented in binary
($\mathcal{O}(\log n)$ bits are sufficient for this).
- Whenever tha algorithm Alg_2 needs to read a symbol from its input:
    - It will start a simulation of the algorithm Alg_1 from the beginning.
    - In those steps, where the algorithm Alg_1 would write a symbols to its output, this symbol is not written anywhere. Instead, a counter of written symbols is incremented by 1.
    - At the moment when Alg_1 would write a symbol to a position that Alg_2 needs to read, the simulation of Alg_1 is stopped, and the algorithm Alg_2 obtains the corresponding symbols, and the simulation of Alg_2 continues.

Logspace reductions

It is not difficult to see the following:

- Let us assume that problem A is logspace reducible to problem B.
- Is there would exist an algorithm with logarithmic space complexity solving problem B, there would exists also an algorithm solving problem A with logarithmic space complexity.
- So if there is no algorithm with logarithmic space complexity solving problem A, then there is also no algorithm solving problem B with logarithmic space complexity.

Logarithmic amount of memory

No **deterministic** algorithm with logarithmic space complexity is known for the following problem:

Graph Reachability

Input: A directed graph $G = (V, E)$ with two designated nodes s and t.

Question: Is there a path from node s to node t in the graph G?

But obviously there is a very simple **nondeterministic** algorithm with space complexity $\mathcal{O}(\log n)$ solving this problem:

- It rememebers only a current node v and a value of a couter c.
- It initialized $v := s$ a $c := m - 1$ where m is the number of nodes of the graph G.
- It nondeterministically guesses a path, and with every step, it decrements the value of the counter by 1.

Classes L and NL

Let us recall definitions of the following classes:

The class LOGSPACE (shortly L)

The class LOGSPACE (shortly L) consists of exactly those decision problems, for which there exists a **deterministic** algorithm with space complexity $\mathcal{O}(\log n)$.

The class NLOGSPACE (shortly NL)

The class NLOGSPACE (shortly NL) consists of exactly thost decision problems, for which there exists a **nondeterministic** algorithm with space complexity $\mathcal{O}(\log n)$.

Classes L and NL

- It is obvious that L ⊆ NL.
- Similarly as in the case of classes P and NP where it is not known whether P = NP,
  also for the classes L and NL it is not known whether L = NL.

  (It seems that probably this equality does not hold
  but there is no proof of that.)

Example: We have seen the following:

The "'*Graph Reachability*" problem is in NL.

It seems that this problem is not in L but it is not sure.

NL-complete problems

Definition

- A problem A is **NL-hard** if every problem from NL is logspace reducible to the problem A.
- A problem A is **NL-complete** if it is NL-hard and belongs to NL.

- If any NL-complete problem could be solved by a deterministic algorithm with logarithmic space complexity, it would mean that L = NL.
- If there would be at least one problem that is in NL but not in L, then there surely could not exist a deterministic algorithm with logarithmic space complexity for any NL-hard problem.

NL-complete problems

Theorem

"Graph Reachability" is an NL-complete problem.

Proof idea:

We have already seen that this problem belongs to NL.

We must show that for every problem A from NL there exists a logspace reduction from A to the graph reachability problem.

Since problem A belongs to NL, there exists a nondeterministic machine $\mathcal{M}$ (e.g., a Turing machine or other type of a machine) with logarithmic space complexity that solves it.

The number of possible configurations of the machine $\mathcal{M}$ on the given input x of size n will be polynomial.

In a logarithmic space, it is possible to generate a graph where:

- **nodes** — configurations of the machine $\mathcal{M}$
- **edges** — transitions between these configurations

NL-complete problems

Using logspace reductions from this problem, we can show NL-hardness of other problems.

Examples of some NL-complete problems:

2-UNSAT

Input: Boolean formula φ in conjunctive normal form where every clause contains exactly 2 literals.

Question: Is the formula φ unsatisfiable (i.e., is it a contradiction)?

NL-complete problems

Accepting a word by an NFA

Input: Nondeterministic finite automaton $\mathcal{A}$ and a word w.

Question: Does the automaton $\mathcal{A}$ accept the word w (i.e., does $w \in \mathcal{L}(\mathcal{A})$ hold)?

Reachable nonterminals in a context-free grammar

Input: A context-free grammar $\mathcal{G} = (\Pi, \Sigma, S, P)$ and nonterminal $B \in \Pi$.

Question: Are there some $\alpha, \beta \in (\Pi \cup \Sigma)^*$ such that $S \Rightarrow^* \alpha B \beta$?

NL-complete problems

Emptiness of a language accepted by a DFA

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

Question: Does $\mathcal{L}(\mathcal{A}) = \emptyset$ hold?

Universality of a DFA

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

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

Equivalence of DFA

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

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

NL-complete problems

Generating an element by an associative operation

Input: A finite set X, an associative binary operation $\circ$ on the set X (given in a form of a table specifying values $x \circ y$ for each pair $x, y \in X$), a subset $S \subseteq X$, and an element $t \in X$.

Question: Is it possible to generate the element t from the elements of the set S ?

An element t can be generated from the elements of a set S if there exists a sequence $x_1, x_2, \ldots, x_k$ of elements of the set S such that

$$t = x_1 \circ x_2 \circ \cdots \circ x_k$$

P-complete problems

P-complete problems

Let us recall that P (resp. PTIME) is the class of decision problems solvable by an algorithm with a **polynomial** time complexity.

Definition

- A problem A is P-**hard** if every problem from P is logspace reducible to the problem A.
- A problem A is P-**complete** if it is P-hard and it belongs to the class P.

- It is obvious that NL $\subseteq$ P.
- Whether NL = P is not known. (It seems that probably NL $\neq$ P)
- If any P-complete problem would be in NL, then every problem from P would be in NL, and we would have NL = P.
- On the other hand, if there would at least one problem in P that would not be in NL, then no P-complete problem would be in NL.

P-complete problems

P-complete problems play an important role as problems that:

- They can be solved in a polynomial time.
- They are **difficult to parallelize** in the sense that it seems that there are no **efficient parallel** algorithms for them, i.e., algorithms that:
  - use a polynomial number of processors
  - work in a polylogarithmic time
    (i.e., in time $\mathcal{O}(\log^k n)$ for some constant k)

  Remark: The class of problems that can be solved by such efficient parallel algorithms is denoted NC.

  We will deal with the class NC (and parallel algorithms in general) later.

Circuits

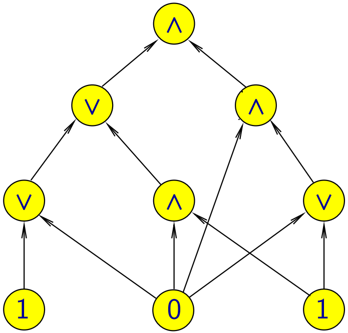
A **Boolean circuit** is a directed acyclic graph consisting of nodes of two types — **inputs** and **gates**, and edges that represent **wires**:

- **Inputs** — there are no wires going to them, they are denoted by names of boolean variables (every input with a different variable) — a value of the given input is given by an assignment of a boolean variable to the corresponding variable
- **Gates** — are of three different types:
  - NOT — negation; exactly one edge enters into this gate
  - AND — conjunction; there are always at least two edges entering this gate
  - OR — disjunction; there are always at least two edge entering this gate
- One of the nodes is denoted as an output.

Circuit Value Problem (CVP)

Input: A description of a boolean circuit G and a truth valuation ν representing values assigned to its inputs.

Question: Is the value 1 on the output of the circuit G in the given assignment ν?

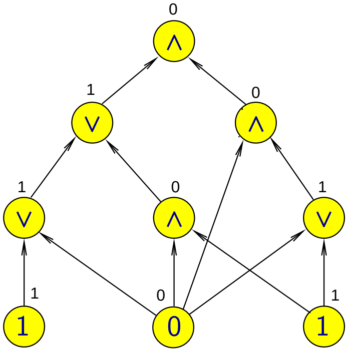


Circuit Value Problem (CVP)

Circuit Value Problem (CVP)

Input: A description of a boolean circuit G and a truth valuation ν representing values assigned to its inputs.

Question: Is the value 1 on the output of the circuit G in the given assignment ν ?



Circuit Value Problem (CVP)

Theorem

CVP is P-complete problem.

Proof idea:
The fact that CVP belongs to the class P is obvious — a straightforward algorithm evaluating values of all gates has obviously a polynomial time complexity.

We need to show that every problem from P is logspace reducible to CVP.

Let us assume we have a problem A from the class P.
There exists a polynomial algorithm that solves the problem A.
This algorithm can implemented as a machine (e.g., a Turing machine) $\mathcal{M}$ with a polynomial time complexity that can be bounded from above by some polynomial $p(n)$.

Circuit Value Problem (CVP)

The length of a computation on an input of size n is at most $p(n)$.

Individual configuration of the machine $\mathcal{M}$ can be encoded as sequences of $\mathcal{O}(p(n))$ bits.

A circuit is constructed for an input x:

- It will consist of $p(n) + 1$ "levels" that would correspond to configurations $\alpha_0, \alpha_1, \ldots, \alpha_{p(n)}$ through which the machine $\mathcal{M}$ goes in the computation on the input x.
- The inputs will represent the initial configuration α_0.
- Between levels i and $i + 1$ (where $0 \leq i < p(n)$) we add a circuit that computes a binary representation of a configuration α_{i+1} from a binary representation of a configuration α_i.
- After the last level (level $p(n)$) we add a circuit that generates output 1 iff the values on this level represent an accepting configuration.

Circuit Value Problem (CVP)

Definition

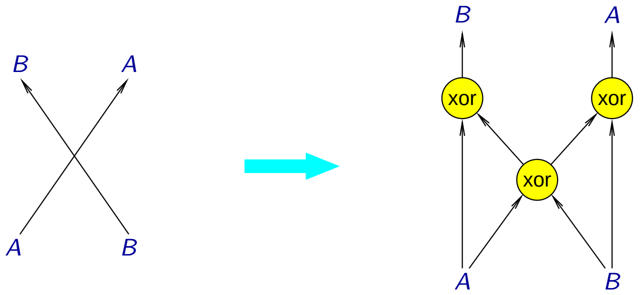
A directed acyclic graph $G = (V, E)$ is **topologically sorted** if its nodes are numbered by numbers $\{1, 2, \ldots, n\}$ in such a way that for every edge $(i, j) \in E$ we have $i < j$ (i.e., edges go only from nodes with lower indexes to nodes with higher indexes).

- It is not difficult to see that the construction in the proof of P-completeness of CVP can be done in such a way that the resulting graph of the constructed circuit is topologically sorted.
- So the CVP remains P-complete also in the special case where we require that the graph of the circuit is topologically sorted.

Circuit Value Problem (CVP) — a planar graph

Using a logspace reduction from CVP we can show P-completeness of CVP also in the special case where we require that the graph of a circuit is **planar**.

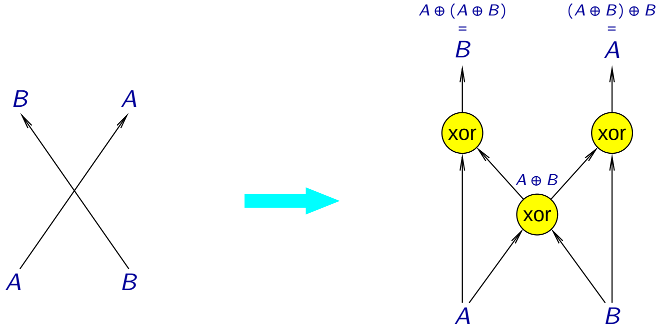
Proof idea: A crossing of wires can be replaced with three XOR gates:



Circuit Value Problem (CVP) — a planar graph

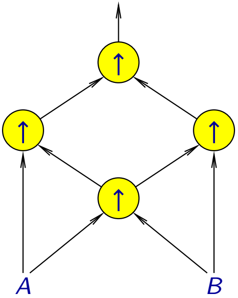
Using a logspace reduction from CVP we can show P-completeness of CVP also in the special case where we require that the graph of a circuit is **planar**.

Proof idea: A crossing of wires can be replaced with three XOR gates:



Circuit Value Problem (CVP) — a planar graph

A XOR gate can be implemented using four NAND gates in such a way that the resulting graph is planar:



$(A \uparrow (A \uparrow B)) \uparrow ((A \uparrow B) \uparrow B) \iff \neg((A \uparrow (A \uparrow B)) \wedge ((A \uparrow B) \uparrow B))$
$\iff \neg(A \uparrow (A \uparrow B)) \vee \neg((A \uparrow B) \uparrow B) \iff (A \wedge (A \uparrow B)) \vee ((A \uparrow B) \wedge B)$
$\iff (A \wedge \neg(A \wedge B)) \vee (\neg(A \wedge B) \wedge B) \iff (A \wedge (\neg A \vee \neg B)) \vee ((\neg A \vee \neg B) \wedge B)$
$\iff (A \wedge \neg B) \vee (\neg A \wedge B) \iff A \text{ xor } B$

Monotone Circuit Value Problem (MCVP)

A boolean circuit is **monotone** if it does not contain NOT gates.

Monotone Circuit Value Problem (MCVP)

Input: A description of a monotone boolean circuit G where moreover exactly two wires enter to each gate of type AND and OR, and a truth valuation ν.

Question: Is the output value of the circuit G for the valuation ν the value 1 ?

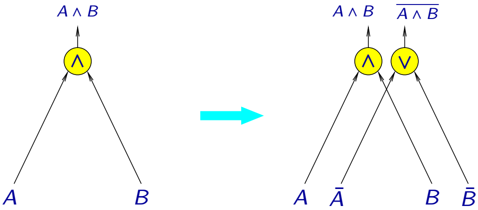
By a logspace reduction from CVP we can show the following:

Theorem

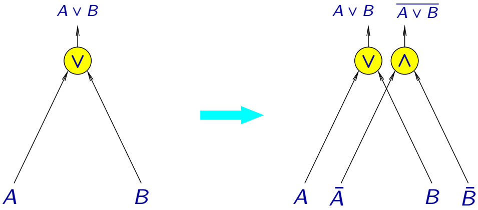
MCVP is P-complete problem.

Monotone Circuit Value Problem (MCVP)

- Replacement of AND gate:



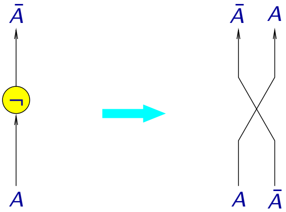
- Replacement of OR gate:



Monotone Circuit Value Problem (MCVP)

- Replacements of inputs:

- Replacement of NOT gate:



Examples of P-complete problems

Using logspace reductions from MCVP, we can show P-hardness of other problems.

Several examples of P-complete problems are described.

Combinatorial game

Input: A combinatorial game of two players where a graph of the game is given explicitly, i.e., where all positions and possible moves are listed explicitly.

Question: Does *Player* I have a winning strategy in this game?

Examples of P-complete problems

Generating a word by a context-free grammar

Input: A context-free grammar $\mathcal{G}$ and a word $w \in \Sigma^*$.

Question: Does the word w belong to the language generated by the grammar $\mathcal{G}$
(i.e., does $w \in \mathcal{L}(\mathcal{G})$ hold)?

Emptiness of a language generated by a context-free grammar

Input: A context-free grammar $\mathcal{G}$.

Question: Does $\mathcal{L}(\mathcal{G}) = \emptyset$hold?

Infinity of a language generated by a context-free grammar

Input: A context-free language $\mathcal{G}$.

Question: Is the language $\mathcal{L}(\mathcal{G})$ infinite?

Examples of P-complete problems

Generating of an element by a binary operation

Input: A finite set X, a binary operation $\circ$ on the set X (given as a table specifying value $x \circ y$ for each pair $x, y \in X$), a subset $S \subseteq X$, and an element $t \in X$.

Question: Is it possible to generate the element t from elements of the set S?

An element t can be generated from elements of a set S if there exists an expression consisting of constants representing the elements form the set S, on which the operation $\circ$ can be applied, where the value of this exapression is 1.

Another way how to say this, is to specify that the element t belongs to the smallest Y (where $Y \subseteq X$), satisfying two following conditions:

- $S \subseteq Y$
- for each two elements $x, y \in Y$ it holds that $x \circ y \in Y$.

Examples of P-complete problems

Maximum flow problem

Input: A network G with capacities of edges, with a source s and a sink t, and a number k.

Question: Has the k-th bit of the number representing the maximal flow in G from the source to the sing the value 1 ?

Depth-first seach

Input: A directed graph $G = (V, E)$ where it is specified for each node a particular ordering of edges going out of this node, the initial node $s \in V$, and a pair of nodes $u, v \in V$.

Question: In the depth-first search of the graph G that starts in the node s, and that goes through edges going out of each node in the specified order, is the node u visited before the node v ?