Complexity Theory I

Markus Lohrey

Universität Siegen

Wintersemester 2022/2023

Part 1: basics

In the following we explain some basics:

- Turing machines (non-deterministic, deterministic)
- configurations
- computations,...

Most of this stuff we do not really need later, because:

- Turing machines can be defined in several equivalent ways.
- Turing machiens can be replaced by other equivalent computation models (e.g. register machines).

Turing machines: definition

Notation: With $\mathcal{P}_{\neq\emptyset}(A) = 2^A \setminus \{\emptyset\}$ we denote the set of all non-empty subsets of the set A.

Definition

A non-deterministic k-tape Turing machine is a tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, q_J, q_N, \square)$.

- Q : a finite set of state
- $q_0 \in Q$: initial state
- $q_J \in Q$: accepting state
- $q_N \in Q$: rejecting state with $q_J \neq q_N$
- Γ : finite tape alphabet
- Σ: finite input alphabet with $\triangleright, \triangleleft \notin \Sigma$
- $\square \in \Gamma$: blank symbol
- $\delta : (Q \setminus \{q_J, q_N\}) \times (\Sigma \cup \{\triangleright, \triangleleft\}) \times \Gamma^k \to \mathcal{P}_{\neq\emptyset}(Q \times \Gamma^k \times \{-1, 1\}^{k+1})$: transition function. -1 (1): move tape head to the left (right).

Turing machines: definition

For all instructions $(p, c_1, \dots, c_k, d_0, \dots, d_k) \in \delta(q, a, b_1, \dots, b_k)$ we have:

- $a = \triangleright \quad \Rightarrow \quad d_0 = 1$
- $a = \triangleleft \quad \Rightarrow \quad d_0 = -1$

For a deterministic k-tape Turing machine M we require

$$\delta : (Q \setminus \{q_J, q_N\}) \times (\Sigma \cup \{\triangleright, \triangleleft\}) \times \Gamma^k \to Q \times \Gamma^k \times \{-1, 1\}^{k+1}$$

A Turing machine with output is defined as a deterministic Turing machine, except that there is an additional output alphabet Σ' and for δ we have:

$$\delta : (Q \setminus \{q_J, q_N\}) \times (\Sigma \cup \{\triangleright, \triangleleft\}) \times \Gamma^k \to Q \times \Gamma^k \times \{-1, 1\}^{k+1} \times (\Sigma' \cup \{\lambda\})$$

(λ is the empty word).

Turing machines: configurations

Definition 1

A configuration α of the Turing machine M for input $w \in \Sigma^*$ is a tuple $\alpha = (q, i, u_1, i_1, \ldots, u_k, i_k)$ with:

- $q \in Q$: current state of the Turing machine
- $1 \leq i \leq |w| + 2$: the read head for the input tape is currently scanning the i-th symbol of $\triangleright w \triangleleft$.
- $\forall j \in \{1, \ldots, k\} : u_j \in \Gamma^+, 1 \leq i_j \leq |u_j|$: the j-th work tape has the content $\cdots \Box \Box u_j \Box \Box \cdots$ and the j-th read-write head is currently scanning the i_j-th symbol of u_j.
  If $i_j < |u_j|$ (resp., $i_j > 1$) then u_j is not allowed to end (resp., begin) with $\Box$.

The length $|\alpha|$ of the configuration $\alpha = (q, i, u_1, i_1, \ldots, u_k, i_k)$ is $|\alpha| = max\{|u_j| \mid 1 \leq j \leq k\}$.

Turing machines: start configuration, transitions, ...

1. For an input $w \in \Sigma^*$, the corresponding start configuration is

$$\mathrm{Start}(w) = (q_0, 1, \square, 1, \dots, \square, 1).$$

   Note: $|\mathrm{Start}(w)| = 1$.

2. For some $\tilde{u} \in Q \times \Gamma^k \times \{-1, 1\}^{k+1}$ and configurations

$$\alpha = (q, i, u_1, i_1, \dots, u_k, i_k) \text{ and } \beta$$

   we write $\alpha \vdash_{\tilde{u}} \beta$ if

$$\tilde{u} \in \delta(q, (\triangleright w \triangleleft)[i], u_1[i_1], \dots, u_k[i_k])$$

   an the application of the "instruction" $\tilde{u}$ to the configuration α yields the configuration β.

   Exercise: define this formally.

3. We write $\alpha \vdash_M \beta$ if there is $\tilde{u} \in Q \times \Gamma^k \times \{-1, 1\}^{k+1}$ with $\alpha \vdash_{\tilde{u}} \beta$.

Turing machines: computations, protocols

1. Accept_M (resp., Reject_M) is the set of configurations where the current state is q_J (resp., q_N).
   Note: for α there is no configuration β with $\alpha \vdash_M \beta$ if and only if $\alpha \in \mathrm{Accept}_M \cup \mathrm{Reject}_M$.
2. Note: $\alpha \vdash_M \beta \Rightarrow |\alpha| - |\beta| \in \{-1, 0, 1\}$
3. A computation of M for input w is a sequence of configurations $\alpha_0, \alpha_1, \ldots, \alpha_m$ with
   - $\mathrm{Start}(w) = \alpha_0$
   - $\forall 1 \leq i \leq m : \alpha_{i-1} \vdash_M \alpha_i$

   The computation is accepting if $\alpha_m \in \mathrm{Accept}_M$.
4. The protocol for this computation is the unique sequence

$$\tilde{u}_0 \tilde{u}_1 \cdots \tilde{u}_{m-1} \in \left(Q \times \Gamma^k \times \{-1, 1\}^{k+1}\right)^*$$

   with $\alpha_i \vdash_{\tilde{u}_i} \alpha_{i+1}$.

Turing machines: accepted set, duration and space

1. The duration (resp,. space) of the computation $\alpha_0, \alpha_1, \dots, \alpha_m$ is m (resp., $\max\{|\alpha_i| \mid 0 \le i \le m\}$).
2. On input w, the machine M uses time (resp., space) at most $N \in \mathbb{N}$, if every computation of M on input w has duration (resp., space) $\le N$.
3. Let $f : \mathbb{N} \to \mathbb{N}$ be a monotone growing function.

   M is f-time-bounded if M uses time at most $f(|w|)$ for every input w.

   M is f-space-bounded if M uses space at most $f(|w|)$ for every input w.
4. $L(M) = \{w \in \Sigma^* \mid \exists$ accepting computation of M on input $w\}$ is the set accepted by M.

Turing machines: counting configurations

The following simple lemma will be used many times:

Lemma 2

Let M be a non-deterministic Turing machine. There are constants c, d such that for all inputs w for M and all $m \geq 1$ we have:

- *There are at most $c \cdot |w| \cdot d^m$ configurations of length $\leq m$ with w as input.*
- *Let M be f-space-bounded. Then the number of configurations that can be reached from $\mathrm{Start}(w)$ is at most $c \cdot |w| \cdot d^{f(|w|)}$.*
- *In particular: if $f \in \Omega(\log(n))$ then the number of configurations that can be reached from $\mathrm{Start}(w)$ is at most $2^{\mathcal{O}(f(|w|))}$ (if $|w|$ is large enough).*

Complexity classes

Let $f : \mathbb{N} \to \mathbb{N}$ be a monotone growing function.

$$\mathsf{DTIME}(f) = \{L(M) \mid M \text{ deterministic } \& \ f\text{-time-bounded}\}$$

$$\mathsf{NTIME}(f) = \{L(M) \mid M \text{ non-deterministic } \& \ f\text{-time-bounded}\}$$

$$\mathsf{DSPACE}(f) = \{L(M) \mid M \text{ deterministic } \& \ f\text{-space-bounded}\}$$

$$\mathsf{NSPACE}(f) = \{L(M) \mid M \text{ non-deterministic } \& \ f\text{-space-bounded}\}$$

For a class $\mathcal{C}$ of languages, we define $\mathbf{Co}\mathcal{C} = \{L \mid \Sigma^* \setminus L \in \mathcal{C}\}$ as the set of complements of languages in $\mathcal{C}$.

Complexity classes

We will consider the classes DTIME(t) and NTIME(t) only for functions $t(n)$ with $\forall n \in \mathbb{N} : t(n) \geq n + 1$.

This is needed in order to be able to read the whole input.

We will consider the classes
DSPACE(s) and NSPACE(s) only for functions $s(n) \in \Omega(\log(n))$.

This allows to store a position $i \in \{1, \ldots, n\}$ in the input tape on a work tape.

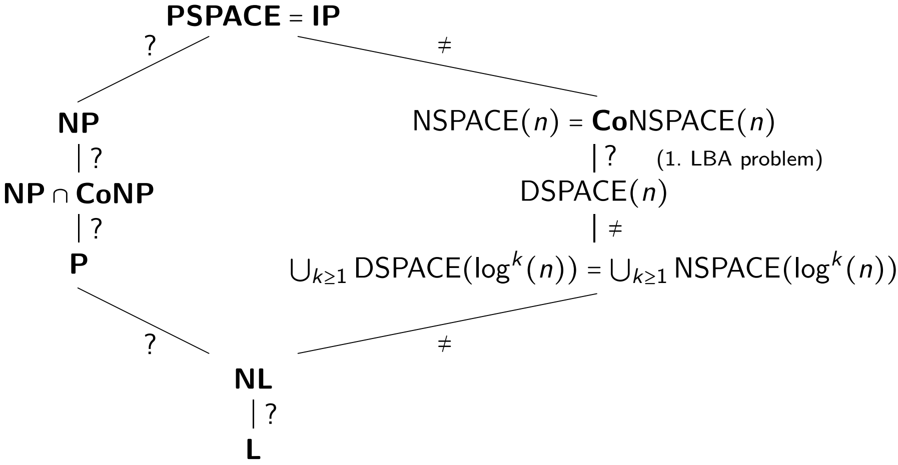
Important complexity classes

Some widely used abbreviations:

$$
\begin{aligned}
\mathbf{L} &= \mathrm{DSPACE}(\log(n)) && (1)\\
\mathbf{NL} &= \mathrm{NSPACE}(\log(n)) && (2)\\
\mathbf{P} &= \bigcup_{k \geq 1} \mathrm{DTIME}(n^k) && (3)\\
\mathbf{NP} &= \bigcup_{k \geq 1} \mathrm{NTIME}(n^k) && (4)\\
\mathbf{PSPACE} &= \bigcup_{k \geq 1} \mathrm{DSPACE}(n^k) = \bigcup_{k \geq 1} \mathrm{NSPACE}(n^k) && (5)
\end{aligned}
$$

The equality = in (5) follows from Savitch's theorem (comes later).

Relationships between complexity classes



There are many other complexity classes: visit the complexity zoo (`https://complexityzoo.uwaterloo.ca/Complexity_Zoo`)

Examples

- $\{a^n b^n c^n \mid n \geq 1\} \in \mathbf{L}$
- $\{w\$w \mid w \in \Sigma^*\} \in \mathbf{L}$
- The set PRIM $= \{p \in 1\{0,1\}^* \mid p$ is the binary encoding of a prime number $\}$ is in DSPACE(n).

  Agrawal, Kayal and Saxena proved in 2002 that PRIM $\in \mathbf{P}$, see e.g. the book *Primality Testing in Polynomial Time* of M. Dietzfelbinger, Springer 2004.

  Note: In PRIM we asl for a binary encoded integer, whether it is a prime number. For a unary encoded integer n (represented by n many a's) it is easy to check in polynomial time whether it is a prime number.

Variants of algorithmic problems

Example 1: Traveling Salesman Problem (TSP)

A traveller wants to visit a set of cities without visiting a city twice. He wants to take the shortest route. The map is represented by a directed graph, whose nodes are the cities. A street from city A to city B with distance $w \in \mathbb{N}$ is represented by an edge from A to B with weight w.

Let $G = (V, E, \gamma : E \to \mathbb{N})$ be a directed graph with set of nodes $V = \{1, \dots, n\}$, set of edges $E \subseteq V \times V$ and edge weights $\gamma(e) \in \mathbb{N} \setminus \{0\}$ for all $e \in E$.

A (Hamilton) circuit W is a sequence $W = (x_0, \dots, x_n)$, $x_0 = x_n$, $x_i \neq x_j$ for $1 \leq i < j \leq n$ and $(x_{i-1}, x_i) \in E$ for $1 \leq i \leq n$.

The cost $\gamma(W)$ of the circuit W is the sum of all edge weights in the circuit: $\gamma(W) = \sum_{i=1}^{n} \gamma(x_{i-1}, x_i)$.

Variants of algorithmic problems

(A) the decision problem:

input: $G = (V, E, \gamma : E \to \mathbb{N})$ and $k \geq 0$.

question: Does there exist a circuit with cost $\leq k$?

(B) the computation variant:

input: $G = (V, E, \gamma : E \to \mathbb{N})$ and $k \geq 0$.

output: a circuit W with $\gamma(W) \leq k$ if it exists, otherwise **no**.

(C) the optimization variant:

input: $G = (V, E, \gamma : E \to \mathbb{N})$.

output: circuit with smallest possible cost if a circuit exists, otherwise **no**.

The input size is (up to some constant factor)
$|V|^2 + \sum_{e \in E}(\lfloor \log \gamma(e) \rfloor + 1) + \lfloor \log(k) \rfloor + 1$ for (A) and (B), and
$|V|^2 + \sum_{e \in E}(\lfloor \log \gamma(e) \rfloor + 1)$ for (C).

Variants of algorithmic problems

From a practical point of view, variant (C) (optimization problem) is the most important.

But: (A) can be solved in polynomial time $\Longrightarrow$
(C) can be solved in polynomial time.

Proof:

Step 1: Check whether there exists a (Hamilton) circuit in G:

For this, we call (A) with $k_{\max} = |V| \cdot \max\{\gamma(e) \mid e \in E\}$.

Note: there is a circuit if and only if there is a circuit with cost $\leq k_{\max}$.

In the following, we assume that there is a circuit in G.

Variants of algorithmic problems

Step 2: Compute $k_{opt} = \min\{\gamma(W) \mid W \text{ is a circuit}\}$ using binary search:

FUNCTION k_{opt}
  $k_{\min} := 1$ (or alternatively $k_{\min} := |V|$)
  while $k_{\min} < k_{\max}$ **do**
    $k_{\text{mid}} := k_{\min} + \lceil \frac{k_{\max} - k_{\min}}{2} \rceil$
    if $\exists$ circuit W with $\gamma(W) \leq k_{\text{mid}}$ **then** $k_{\max} := k_{\text{mid}}$
    else $k_{\min} := k_{\text{mid}} + 1$
    endif
  endwhile
  return $k_{\min}$
ENDFUNC

Note: the number of iterations for the **while**-loop is bounded by

$$\begin{aligned}\log_2(k_{\max}) &= \log_2(|V| \cdot \max\{\gamma(e) \mid e \in E\}) \\ &= \log_2(|V|) + \log_2(\max\{\gamma(e) \mid e \in E\}) \leq \text{input size.}\end{aligned}$$

Variants of algorithmic problems

Step 3: Compute the optimal circuit:

FUNCTION optimal circuit
  Let $e_1, e_2, \ldots, e_m$ be an arbitrary enumeration of E
  $G_0 := G$
  for $i := 1$ **to** m **do**
    if $\exists$ circuit W in $G_{i-1} \setminus \{e_i\}$ with $\gamma(W) \leq k_{opt}$ **then**
      $G_i := G_{i-1} \setminus \{e_i\}$
    else
      $G_i := G_{i-1}$
    endif
  endfor
  return G_m
ENDFUNC

Variants of algorithmic problems

Claim: For all $i \in \{0, \dots, m\}$:

1. in G_i there is a circuit W with $\gamma(W) = k_{opt}$;
2. every circuit W in G_i with $\gamma(W) = k_{opt}$ contains all edges from $\{e_1, \dots, e_i\} \cap E[G_i]$ ($E[G_i]$ = set of edges of G_i).

Proof:

1. Follows directly by induction on i.
2. Assume that there is a circuit W in G_i with $\gamma(W) = k_{opt}$ and an edge e_j $(1 \le j \le i)$ with:
   - e_j belongs to G_i and
   - e_j does not belong to the circuit W.

   W is also a circuit in G_{j-1}. $\Rightarrow$
   W is a circuit in $G_{j-1} \setminus \{e_j\}$. $\Rightarrow$
   $e_j \notin E[G_j]$ and hence $e_j \notin E[G_i]$. contradiction!

Consequence: G_m has a circuit W with $\gamma(W) = k_{opt}$ and every edge of G_m belongs to W, which implies $G_m = W$. □

Variants of algorithmic problems

Example 2: vertex cover (VC)

Let $G = (V, E)$ be an undirected graph (i.e. $E \subseteq \binom{V}{2}$).
A subset $C \subseteq V$ is a vertex cover for G if for every edge $\{u, v\} \in E$ we have $\{u, v\} \cap C \neq \emptyset$.

(A) the decision variant:

input: $G = (V, E)$ and $k \geq 0$.

question: Does G have a vertex cover C with $|C| \leq k$?

(B) the computation variant:

input: $G = (V, E)$ and $k \geq 0$.

output: a vertex cover C with $|C| \leq k$ if it exists, otherwise **no**.

(C) the optimization variant:

input: $G = (V, E)$.

output: a smallest possible vertex cover for G.

Variants of algorithmic problems

Again we have: (A) can be solved in polynomial time $\Longrightarrow$ (C) can be solved in polynomial time.

Proof this as an exercise.

The graph accessibility problem

The graph accessibility problem (GAP) is a central decision problem in complexity theory:

input: a directed graph $G = (V, E)$ and two nodes $s, t \in V$.
question: is there a path in G from s to t?

GAP belongs to **P**: GAP can be solved in $\mathcal{O}(|V|)$ using breadth-first search.

Sharper statement: GAP belongs to **NL** (later we will prove **NL** $\subseteq$ **P**):

FUNCTION GAP
  var $v := s$
  while $v \neq t$ **do**
    nondeterministically choose an edge $(v, w) \in E$
    $v := w$
  endwhile
  return „there is a path from s to t."
ENDFUNC

The graph accessibility problem

This is a nondeterministic algorithm that can be easily implemented on a nondeterministic Turing machine.

Why does the algorithm only use logarithmic space?

- At every time instant, the algorithm only has to store the current node $v \in V$.
- If there are n nodes, then we can identify the nodes with the numbers $1, \ldots, n$. Therefore, the variable v only needs $\log_2(n) = \log_2(|V|)$ many bits.

Remarks:

- Savitch's theorem (comes later) implies $\mathsf{GAP} \in \mathsf{DSPACE}(\log^2(n))$.
- Omer Reingold proved in 2004 that the graph accessibility problem for undirected graphs (UGAP) belongs to the class **L**, see `https://eccc.weizmann.ac.il/eccc-reports/2004/TR04-094/index.html`

Part 2: Relationsships between complexity classes

The proofs for the theorems in this section can be found for instance in Hopcroft, Ullman; *Introduction to Automata Theory, Languages and Computation*, Addison Wesley 1979.
We will only sketch some of the proofs.

For a function $f : \mathbb{N} \to \mathbb{N}$ let $\mathsf{DTIME}(\mathcal{O}(f)) = \bigcup_{c \in \mathbb{N}} \mathsf{DTIME}(c \cdot f)$, and analogously for $\mathsf{NTIME}, \mathsf{DSPACE}, \mathsf{NSPACE}$.

Theorem 3

Let $f : \mathbb{N} \to \mathbb{N}$.

1. *For* $X \in \{D, N\}$ *we have* $\mathsf{XSPACE}(\mathcal{O}(f)) = \mathsf{XSPACE}_{\textit{1-tape}}(f)$.
2. $\exists \epsilon > 0\ \forall n : f(n) \geq (1+\epsilon)n \quad \Longrightarrow \quad \mathsf{DTIME}(\mathcal{O}(f)) = \mathsf{DTIME}(f)$.
3. $\mathsf{NTIME}(\mathcal{O}(f)) = \mathsf{NTIME}(f)$.
4. $\mathsf{DTIME}(n) \subsetneq \mathsf{DTIME}(\mathcal{O}(n))$.

Point 1 combines tape reduction with tape compression.
Point 2 and 3 are sometimes called time compression.

The theorem of Hennie and Stearns (1966)

The theorem of Hennie and Stearns is a tape reduction theorem for time complexity classes.

Theorem 4

Let $k \geq 1$ and assume that $\exists \varepsilon > 0\ \forall n : f(n) \geq (1+\varepsilon)n$. Then we have $\mathsf{DTIME}_{k\text{-}tape}(f) \subseteq \mathsf{DTIME}_{2\text{-}tape}(f \cdot \log(f))$.

DTIME(f) ⊆ NTIME(f) ⊆ DSPACE(f)

Theorem 5

If $\forall n : f(n) \geq n$, *then* DTIME(f) ⊆ NTIME(f) ⊆ DSPACE(f).

Proof: We only have to show NTIME(f) ⊆ DSPACE(f).

Let $M = (Q, \Sigma, \Gamma, \delta, q_0, q_J, q_N, \Box)$ be a non-deterministic f-time-bounded Turing machine.

An input $w \in \Sigma^*$ of length n is accepted by M if and only if there is a protocol $\tilde{u}_1 \tilde{u}_2 \cdots \tilde{u}_m$ with $m \leq f(n)$ and

$$\mathsf{Start}(w) \vdash_{\tilde{u}_1} c_1 \vdash_{\tilde{u}_2} c_2 \cdots \vdash_{\tilde{u}_m} c_m \in \mathrm{Accept}_M .$$

We search systematically (e.g. in length lexicographic order) through all protocols of length at most $f(n)$ and check whether such a protocol leads to an accepting configuration.

DTIME(f) ⊆ NTIME(f) ⊆ DSPACE(f)

Note:

- Every from Start(w) reachable configuration only needs space $f(n)$.
- A protocol of length at most $f(n)$ can be stored in space $\mathcal{O}(f(n))$.

Total space needed: $\mathcal{O}(f) + \mathcal{O}(f) = \mathcal{O}(f)$.

FUNCTION protocol-search(w)
  for all protocols $\tilde{u}_1 \tilde{u}_2 \cdots \tilde{u}_m$ with $m \leq f(|w|)$ **do**
    compute the unique configuration c_m (it it exists) with
    $\text{Start}(w) \vdash_{\tilde{u}_1} c_1 \vdash_{\tilde{u}_2} c_2 \cdots \vdash_{\tilde{u}_m} c_m$
    if $c_m \in \mathrm{Accept}_M$ **then**
      return M accepts w
  endfor
  return M does not accept w
ENDFUNC

$\mathsf{DSPACE}(f) \subseteq \mathsf{NSPACE}(f) \subseteq \mathsf{DTIME}(2^{\mathcal{O}(f)})$

Theorem 6

If $f(n) \in \Omega(\log(n))$ *then* $\mathsf{DSPACE}(f) \subseteq \mathsf{NSPACE}(f) \subseteq \mathsf{DTIME}(2^{\mathcal{O}(f)})$.

Proof: We only have to prove $\mathsf{NSPACE}(f) \subseteq \mathsf{DTIME}(2^{\mathcal{O}(f)})$.

Let M be an f-space bounded non-deterministic Turing machine and $w \in \Sigma^*$ an input of length n.

By Lemma 2 the number of configurations that can be reached from $\mathrm{Start}(w)$ is bounded by $2^{\mathcal{O}(f(n))}$.

We compute the set R of all configurations that can be reached from $\mathrm{Start}(w)$.

DSPACE(f) ⊆ NSPACE(f) ⊆ DTIME($2^{\mathcal{O}(f)}$)

FUNCTION set of reachable configurations
  var $R := \{\mathrm{Start}(w)\}$
  while $\exists$ configurations $\alpha, \beta : \alpha \in R \wedge \beta \notin R \wedge \alpha \vdash_M \beta$ **do**
    $R := R \cup \{\beta\}$
  endwhile
  if $\mathrm{Accept}_M \cap R \neq \emptyset$ **then return** M accepts w
ENDFUNC

How much time does this algorithm need for an input of length n.

- R contains at most $2^{\mathcal{O}(f(n))}$ configurations of length $\leq f(n)$.
- The condition $\exists$ configurations $\alpha, \beta : \alpha \in R \wedge \beta \notin R \wedge \alpha \vdash_M \beta$ can therefore by checked in time $2^{\mathcal{O}(f(n))} \cdot 2^{\mathcal{O}(f(n))} \cdot \mathcal{O}(f(n)^2) \subseteq 2^{\mathcal{O}(f(n))}$.
- Total time needed: $2^{\mathcal{O}(f(n))}$ □

Consequences

- $\mathbf{L} \subseteq \mathbf{NL} \subseteq \mathrm{DTIME}(2^{\mathcal{O}(\log(n))}) = \mathbf{P}$
- $\mathbf{CS} = \mathbf{LBA} = \mathrm{NSPACE}(n) \subseteq \mathrm{DTIME}(2^{\mathcal{O}(n)})$

  Here, **CS** denotes the class of context-senstive languages and **LBA** the class of languages accepted by a linear bounded automaton.

Savitch's theorem (1970)

Theorem 7
If $s \in \Omega(\log(n))$ *then* $\mathrm{NSPACE}(s) \subseteq \mathrm{DSPACE}(s^2)$.

We prove Savitch's theorem under the assumption that the function s is space constructible:

- A function $s : \mathbb{N} \to \mathbb{N}$ with $s \in \Omega(\log(n))$ is space constructible, if there is a deterministic s-space bounded Turing machine that on input a^n (i.e., the unary encoding of n) computes $a^{s(n)}$ on the output tape.
- A function $t : \mathbb{N} \to \mathbb{N}$ with $t \in \Omega(n)$ is time constructible if there is a deterministic Turing machine that on input a^n terminates after exactly $t(n)$ steps.

Proof of Savitch's theorem

Let M be an s-space bounded non-deterministic Turing machine and w an input for M.

Let $\text{Conf}(M, w)$ be the set of all configurations α such that:

- the content of the input tape is w and
- $|\alpha| \leq s(|w|)$.

Hence, $\text{Conf}(M, w)$ contains all configurations that can be reached from $\mathrm{Start}(w)$.

Without loss of generality, we can assume that Accept_M contains at most one configuration α_f that can be reached from $\mathrm{Start}(w)$.

For $\alpha, \beta \in \text{Conf}(M, w)$ and $i \geq 0$ we define:

$$\text{Reach}(\alpha, \beta, i) \iff \exists k \leq 2^i, \alpha_0, \alpha_1, \ldots, \alpha_k \in \text{Conf}(M, w) : \alpha_0 = \alpha, \alpha_k = \beta, \bigwedge_{i=1}^{k} \alpha_{i-1} \vdash_M \alpha_i$$

Proof of Savitch's theorem

By Lemma 2 and $s(n) \in \Omega(\log(n))$, there is a constant c such that for all inputs w we have

$$w \in L(M) \iff \mathrm{Reach}(\mathrm{Start}(w), \alpha_f, c \cdot s(|w|)).$$

Our goal is to compute the predicate $\mathrm{Reach}(\alpha, \beta, i)$ for $\alpha, \beta \in \mathsf{Conf}(M, w)$ and $0 \leq i \leq c \cdot s(|w|)$ in space $\mathcal{O}(s^2)$ on a deterministic machine.

For $i > 0$ we will use the following recursion:

$$\begin{aligned}\mathrm{Reach}(\alpha, \beta, i) \iff \exists \gamma \in \mathsf{Conf}(M, w) : &\mathrm{Reach}(\alpha, \gamma, i-1) \wedge \\ &\mathrm{Reach}(\gamma, \beta, i-1).\end{aligned}$$

Implementation by a deterministic algorithm:

Proof of Savitch's theorem

FUNCTION Reach(α, β, i) (where $\alpha, \beta \in \mathsf{Conf}(M, w)$ and $i \leq c \cdot s(|w|)$)
  var $b :=$ FALSE
  if $i = 0$ **then**
    $b := [(\alpha = \beta) \vee (\alpha \vdash_M \beta)]$
  else
    forall $\gamma \in \mathsf{Conf}(M, w)$ **do**
      if not b **and** Reach($\alpha, \gamma, i-1$) **then**
        $b :=$ Reach($\gamma, \beta, i-1$)
      endif
    endfor
  endif
  return b
ENDFUNC

Proof of Savitch's theorem

Claim: There is a constant ϱ such that a call of $\text{Reach}(\alpha, \beta, i)$ needs space at most $\varrho \cdot (i+1) \cdot s(|w|)$.

We prove the claim by induction on $i \geq 0$:

$i = 0$: The condition $[(\alpha = \beta) \lor (\alpha \vdash_M \beta)]$ can be checked in space $\varrho \cdot s(|w|)$ for a certain constant ϱ.

$i > 0$: By induction, the 1st call $\text{Reach}(\alpha, \gamma, i-1)$ needs space $\varrho \cdot i \cdot s(|w|)$. The same holds for the 2nd call $\text{Reach}(\gamma, \beta, i-1)$.

Note: During the 2nd call $\text{Reach}(\gamma, \beta, i-1)$ one can reuse the space used for the 1st call $\text{Reach}(\alpha, \gamma, i-1)$.

In addition, we need space $3 \cdot s(|w|) + c \cdot s(|w|) \leq \varrho \cdot s(|w|)$ (if $\varrho \geq c+3$) for the configurations α, β, γ and the number i (in unary encoding). This proves the claim.

Proof of Savitch's theorem

In order to decide $w \in L(M)$ we call Reach$(\mathrm{Start}(w), \alpha_f, c \cdot s(|w|))$.

Note: in order to do this, we have to compute the unary encoding of $s(|w|)$. This is possible since we assume that s is space constructible.

Total space needed: $\mathcal{O}(c \cdot s(|w|) \cdot s(|w|)) = \mathcal{O}(s(|w|)^2)$. □

Remarks concerning Savitch's theorem

Savitch's theorem says that a non-deterministic space-bounded Turing machine can be simulated on a deterministic Turing machine with a quadratic blow-up in space. But this space efficient simulation causes a large blow-up in time.

Exercise: What is the running time of the algorithm in our proof of Savitch's theorem?

In order to get rid of the assumption that the function s is space-constructible, one has to show that the actual space needed by an s-space bounded non-deterministic Turing machine on a certain input can be computed in $\mathsf{DSPACE}(s^2)$.

Consequences of Savitch's theorem

Theorem 8
GAP belongs to $\mathrm{DSPACE}(\log^2(n))$*.*

Follows from GAP $\in$ **NL** and Savitch's theorem.

Theorem 9
$\mathbf{PSPACE} = \bigcup_{k\geq 1} \mathrm{DSPACE}(n^k) = \bigcup_{k\geq 1} \mathrm{NSPACE}(n^k)$

Follows from $\mathrm{NSPACE}(n^k) \subseteq \mathrm{DSPACE}(n^{2k})$.

Hierarchy theorems

Theorem 10 (space hierarchy theorem)

Let $s_1, s_2 : \mathbb{N} \to \mathbb{N}$ be functions, $s_1 \notin \Omega(s_2)$, $s_2 \in \Omega(\log(n))$ and assume that s_2 is space constructible. Then $\mathsf{DSPACE}(s_2) \setminus \mathsf{DSPACE}(s_1) \neq \emptyset$ holds.

Remarks:

- $s_1 \notin \Omega(s_2)$ means that $\forall \epsilon > 0\ \exists$ infinitely many n with $s_1(n) < \epsilon \cdot s_2(n)$.
  For instance, let $s_1(n) = n$ and $s_2(n) = \begin{cases} n^2, & \text{if } n \text{ is even} \\ \log n, & \text{otherwise.} \end{cases}$
  Then $s_2 \notin \Omega(s_1)$ and $s_1 \notin \Omega(s_2)$ hold.
- The space hierarchy theorem implies

$$\mathbf{L} \subsetneq \mathsf{DSPACE}(\log^2(n)) \subsetneq \mathsf{DSPACE}(n)$$
$$\subseteq \mathsf{NSPACE}(n) \subsetneq \mathsf{DSPACE}(n^{2,1}) \subsetneq \mathbf{PSPACE}$$

Proof of the space hierarchy theorem

The proof of the space hierarchy theorem is similar to the proof of the undecidability of the halting problem and is based on diagonalization.

First we fix a suitable binary encoding of deterministic 1-tape Turing machines with input alphabet $\{0,1\}$. The encoding must allow a space efficient simulation (we will make this more precise).

Every word $x \in \{0,1\}^*$ must be the encoding of a Turing machine M_x (if x is not "well formed" then x encodes some fixed defaultTuring machine).

Important convention: for all $x \in \{0,1\}^*$ and $k \in \mathbb{N}$ we have $M_x = M_{0^k x}$, i.e., x and $0^k x$ encode the same machine.

Consequence: if a Turing machine M has encoding of length k then for every $\ell \geq k$, M has an encoding of length ℓ.

Goal: a deterministic s_2-space bounded Turing machine M with $L(M) \notin \mathsf{DSPACE}(s_1)$.

$s_2 \in \Omega(\log(n)) \leadsto \exists \delta > 0 \; \exists m \; \forall n \geq m : \log_2(n) \leq \delta \cdot s_2(n)$

Proof of the space hierarchy theorem

We start with a (deterministic) universal Turing machine U.

The input for U is the binary encoding x of a 1-tape Turing machine M_x together with and input $w \in \{0,1\}^*$ for M_x.

U simulates M_x on input w.

We can choose the encoding of Turing machines and U such that for every $x \in \{0,1\}^*$ there is a constant k_x that only depends on M_x such that:
If M_x is s-space bounded, then on input $\langle x, w \rangle$ the machine U uses space at most $k_x \cdot s(|w|) + \frac{1}{1+\delta} \log_2(|w|)$.

By Lemma 2 there is a constant c such that there are at most $n \cdot c^m$ configurations of U with work space $\leq m$ and a fixed input of length n.

Our machine M works for an input $y = 0^\ell x$ (where x does not start with 0) of length $n = |y|$ as follows:

Proof of the space hierarchy theorem

1. Mark space $s_2(n)$ on the work tapes and install a counter C with initial value $2n \cdot c^{s_2(n)} + 1$ (needs space $\leq s_2(n)$ after appropriate tape compression).

   This is possible since s_2 is space constructible.
2. Execute the universal machine U with input $\langle y, y \rangle$ (has length $2n$) and set $C := C - 1$ after every transition of U.
3. If thereby U wants to leave the marked space on the work tapes, the machine M stops in the rejecting state q_N.

   This enforces M to be s_2-space bounded.
4. If C reaches the value 0 before the simulation of $M_y = M_x$ on input y terminates, then U must be trapped in a cycle.

   This implies that M_x does not terminate on input y.

   Then M accepts the input y.
5. If the simulation does terminate before C reaches 0, then M accepts the input y if and only if M_x does not accept y.

Proof of the space hierarchy theorem

Claim: $L(M) \notin \mathrm{DSPACE}(s_1)$

Proof by contradiction: Assume that $L(M) \in \mathrm{DSPACE}(s_1)$.

Let M' be an s_1-space bounded deterministic 1-tape Turing machine with $L(M') = L(M)$ (exists!).

Let $M' = M_x$.

Then, U simulates the machine $M' = M_x$ on an input of length n in space $k_x \cdot s_1(n) + \frac{1}{1+\delta} \log_2(n)$.

Here, k_x is a constant that only depends on M' (but not on n).

Since $s_1 \notin \Omega(s_2)$, there exists an $n \geq |x|$ with

$$k_x(1+\delta) \cdot s_1(n) + \log_2(n) \leq s_2(n) + \log_2(n) \leq (1+\delta) \cdot s_2(n)$$

and hence

$$k_x \cdot s_1(n) + \frac{1}{1+\delta} \log_2(n) \leq s_2(n).$$

Proof of the space hierarchy theorem

Hence, during the simulation of $M' = M_x = M_{0^{n-|x|}x}$ on input $0^{n-|x|}x$ (of length n), the machine M does not leave the space marked in step 1.

We therefore obtain:

$$
\begin{aligned}
0^{n-|x|}x \in L(M) \quad & \iff \quad M \text{ accepts } 0^{n-|x|}x \\
& \iff \quad M_x \text{ does not accept } 0^{n-|x|}x \\
& \iff \quad M' \text{ does not accept } 0^{n-|x|}x \\
& \iff \quad 0^{n-|x|}x \notin L(M') = L(M)
\end{aligned}
$$

□

Time hierarchy theorem

By the theorem of Hennie and Stearns, an arbitrary number of work tapes can be simulated with a logarithmic blow-up in time on two work tapes.

This can be used to prove analogously to the space hierarchy theorem a deterministic time hierarchy theorem.

Theorem 11 (deterministic time hierarchy theorem (without proof))

Let $t_1, t_2 : \mathbb{N} \to \mathbb{N}$, $t_1 \cdot \log(t_1) \notin \Omega(t_2)$, $t_2 \in \Omega(n \log(n))$ *and assume that* t_2 *is time constructible. Then* $\mathsf{DTIME}(t_2) \setminus \mathsf{DTIME}(t_1) \neq \emptyset$ *holds.*

As a consequence we get:

$$\begin{aligned}\mathsf{DTIME}(\mathcal{O}(n)) \subsetneq \mathsf{DTIME}(\mathcal{O}(n^2)) \subsetneq \mathbf{P} \\ \subsetneq \mathsf{DTIME}(\mathcal{O}(2^n)) \subsetneq \mathsf{DTIME}(\mathcal{O}((2+\varepsilon)^n))\end{aligned}$$

Borodin's theorem

The hierarchy theorems that we have discussed all need certain (space or time) constructibility assumptions. This is not avoidable due to the following gap theorem.

Theorem 12 (Borodin's theorem (1972))

Let r be a total, computable and monotonic function, $r(n) \geq n$ *for all* n*. Then there exists effectively a total and computable function* $s : \mathbb{N} \to \mathbb{N}$ *such that* $s(n) \geq n+1$ *for all* n *and* $\mathsf{DTIME}(s) = \mathsf{DTIME}(r \circ s)$*.*

Remarks:

- The composition $r \circ s$ is defined by $r \circ s(x) = r(s(x))$.
- That the total and computable function $s : \mathbb{N} \to \mathbb{N}$ exists effectively means that from a Turing machine that computes r one can compute a Turing machine that computes s.

Proof of Borodin's theorem

Let $M_1, M_2, \ldots$ be an enumeration of all deterministic Turing machines.

Let $t_k(n) \in \mathbb{N} \cup \{\infty\}$ be the maximal computation time that M_k needs on an input of length at most n.

Define the set

$$N_n = \{t_k(n) \mid 1 \le k \le n\} \subseteq \mathbb{N} \cup \{\infty\}.$$

This is a finite set. Hence, for every n there is a number $s(n)$ with

$$N_n \cap [s(n), r(s(n))] = \emptyset.$$

A value $s(n)$ that would satisfy this condition would be

$$s(n) = 1 + \max\{t_k(n) \mid 1 \le k \le n,\ t_k(n) < \infty\}.$$

But this value would in general be too big and not computable.

Proof of Borodin's theorem

A better computable value $s(n)$ can be found on input n by the following algorithm:

FUNCTION $s(n)$
  $s := \max\{n+1, s(n-1)\}$
  repeat
    $s := s + 1$
  until $\forall k \leq n : [t_k(n) < s$ **or** $t_k(n) > r(s)]$
  return s
ENDFUNC

Remark: the function $n \mapsto s(n)$ is computable and monotonic.
But in general, $s(n)$ is not time constructible.

Claim: $\mathrm{DTIME}(s) = \mathrm{DTIME}(r \circ s)$

Proof of Borodin's theorem

Proof of the claim:

Since $r(n) \geq n$ for all n, DTIME$(s) \subseteq$ DTIME$(r \circ s)$ holds.

Now assume that $L \in$ DTIME$(r \circ s)$.

Let M_k be a $(r \circ s)$-time bounded deterministic Turing machine with $L = L(M_k)$.

We have $\forall n : t_k(n) \leq r(s(n))$.

The way we computed $s(n)$ implies $t_k(n) < s(n)$ for all $n \geq k$.

We therefore obtain $L \in$ DTIME(s), because for all inputs of length $< k$ (a constant) a Turing machine can directly output the correct answer after reading the input (this needs $n + 1 \leq s(n)$ steps). □

The theorem of Immerman and Szelepcsényi (1987)

The classes DTIME(f) and DSPACE(f) are closed under complement. Whether this is also true for classes NSPACE(f) was open for a long time.

Already in 1964, Kuroda asked whether the class of context-sensitive languages is closed under complement (2nd LBA problem).
Equivalently: does NSPACE(n) = **Co**NSPACE(n) hold?

After more than 20 years, this question was answered independently by R. Szelepcsényi and N. Immerman:

Theorem 13 (Theorem of Immerman and Szelepcsényi)

Let $f \in \Omega(\log(n))$ *be monotonic. Then* NSPACE(f) = **Co**NSPACE(f) *holds.*

Proof of the theorem of Immerman and Szelepcsényi

Proof technique: inductive counting

Let M be a non-deterministic f-space bounded 1-tape Turing machine and $w \in \Sigma^*$ an input word of length n.

Goal: Check non-deterministically in space $\mathcal{O}(f(n))$, whether $w \notin L(M)$.

W.l.o.g. let α_0 be the only accepting configuration; e.g. $\alpha_0 = (q_J, 1, \square, 1)$ (in particular $|\alpha_0| = 1$).

We need an enumeration $\alpha_0 < \alpha_1 < \alpha_2 < \cdots$ of all configurations of M with input w such that:

- α_0 is the smallest configuration with respect to $<$.
- $\alpha < \alpha'$ implies $|\alpha| \leq |\alpha'|$.
- $\alpha < \alpha'$ can be checked in space $|\alpha| + |\alpha'|$.

Proof of the theorem of Immerman and Szelepcsényi

We can define $\prec$ for instance as follows, where $\alpha = (q, i, u, j)$, $\alpha' = (q', i', u', j')$ are configurations of M on input w:

- If $|u| < |u'|$ then $\alpha \prec \alpha'$.
- If $|u| = |u'|$ and $u <_{\text{lex}} u'$ then $\alpha \prec \alpha'$.

  Here fix an arbitrary order on the tape symbols of M such that $\Box$ is the smallest tape symbol.
- If $u = u'$ and $j < j'$ then $\alpha \prec \alpha'$.
- If $u = u'$, $j = j'$ and $i < i'$ then $\alpha \prec \alpha'$.
- If $u = u'$, $j = j'$, $i = i'$ and $q < q'$ then $\alpha \prec \alpha'$.

  Here fix an arbitrary order on the set of states of M such that q_J is the smallest state.

Proof of the theorem of Immerman and Szelepcsényi

Let $k \geq 0$:

$$
\begin{aligned}
R(k) &= \{\alpha \mid \exists i \leq k : \mathrm{Start}(w) \vdash_M^i \alpha\} \\
r(k) &= |R(k)| \quad \text{(number of configurations that can be reached} \\
&\qquad\qquad \text{from } \mathrm{Start}(w) \text{ in } \leq k \text{ steps)} \\
r(*) &= \max\{r(k) \mid k \geq 0\} \\
&\quad \text{(number of configurations reachable from } \mathrm{Start}(w))
\end{aligned}
$$

Note: Due to Lemma 2 we have

$$r(k) \leq r(*) \in 2^{\mathcal{O}(f(n))}.$$

Since f is not assumed to be space constructible, we also need the value

$$m(k) = \max\{|\alpha| \mid \alpha \in R(k)\}.$$

Proof of the theorem of Immerman and Szelepcsényi

We will describe a non-deterministic $\mathcal{O}(f(n))$-space bounded machine with the following properties:

- If $w \notin L(M)$ then the machine will output on at least on computation path the correct value $r(*)$.
  On other computation paths, the machine can stop without output.
- If $w \in L(M)$ then the machine will stop on all computation paths without output.

Proof of the theorem of Immerman and Szelepcsényi

Computation of $r(*)$ under the assumption that $r(k+1)$ can be computed in space $\mathcal{O}(f(n))$ from $r = r(k)$ using the function compute-r$(k+1, r)$:

```
FUNCTION r(*)
   k := 0
   r := 1     (* contains r(k) *)
   while true do
      r' := compute-r(k+1, r)
      if r = r' then return r
      else k := k+1; r := r'
   endwhile
ENDFUNC
```

Space: Since $r(*) \in 2^{\mathcal{O}(f(n))}$, only space $\mathcal{O}(f(n))$ is needed to store k, r, and r'.

Proof of the theorem of Immerman and Szelepcsényi

The computation of compute-r$(k+1, r)$ is divided into three steps.

Step 1: Compute $m(k)$ from $r = r(k)$ using the function compute-m(k, r).

FUNCTION compute-m(k, r)
  $\alpha := \alpha_0$; $m := 1 (= |\alpha_0|)$
  repeat r **times**
    compute nondeterministically an arbitrary $\alpha' \in R(k)$
    if $\alpha \prec \alpha'$ **then**
      $\alpha := \alpha'$
      $m := |\alpha'|$ **(**$*$ $= \max\{m, |\alpha'|\}$ due to properties of $\prec$ $*$**)**
    else
      „FAILURE " $\Rightarrow$ computation stops
    endif
  endrepeat
  return m
ENDFUNC

Proof of the theorem of Immerman and Szelepcsényi

Note:

- If compute-m(k, r) does not stop with „FAILURE“ (and $r = r(k)$ holds), then the correct value $m(k)$ will be computed.
- If $\alpha_0 \in R(k)$ (and hence $w \in L(M)$) then compute-m(k, r) stops with „FAILURE“ on all computation paths, since $R(k)$ does not contain r many configurations that are strictly larger than α_0.

  In particular: If $w \in L(M)$, then there is a k such that compute-m(k, r) stops with „FAILURE“on all computation paths. Then also the computation of $r(*)$ stops without output.
- If $w \notin L(M)$ then there is a computation path on which compute-m(k, r) does not stop with „FAILURE“(and hence outputs $m(k)$).

Proof of the theorem of Immerman and Szelepcsényi

Space needed by compute-m(k, r)**:** the following has to be stored:

- configurations α, α' with $|\alpha|, |\alpha'| \leq f(n)$.
- $m \leq f(n)$
- binary counter up to k (in order to compute an arbitrary $\alpha' \in R(k)$ nondeterministically)
- binary counter up to $r = r(k)$ (for **repeat** r **times**).

For this, space $\mathcal{O}(f(n))$ is sufficient.

Proof of the theorem of Immerman and Szelepcsényi

Step 2: Let β be an arbitrary configuration. The procedure Reach$(r, k+1, \beta)$ tests nondeterministically, using the value $r = r(k)$, whether $\beta \in R(k+1)$ holds:

FUNCTION Reach$(r, k+1, \beta)$
  $\alpha := \alpha_0$
  repeat r **times**
    compute nondeterministically an arbitrary $\alpha' \in R(k)$
    if $\alpha' < \alpha \vee \alpha' = \alpha$ **then** „FAILURE“ $\Rightarrow$ computation stops
    elseif $\alpha' = \beta \ \vee \ \alpha' \vdash_M \beta$ **then return** true **(*** $\beta \in R(k+1)$ holds ***)**
    else $\alpha := \alpha'$
    endif
  endrepeat
  return false **(*** $\beta \notin R(k+1)$ holds ***)**
ENDFUNC

Proof of the theorem of Immerman and Szelepcsényi

Note:

- If $\text{Reach}(r(k), k+1, \beta)$ does not stop with „FAILURE", a correct answer will be produced.
- If $w \notin L(M)$ (and hence $\alpha_0 \notin R(k)$), then there is a computation path on which $\text{Reach}(r(k), k+1, \beta)$ does not stop with „FAILURE".

Space: the following has to be stored:

- configurations α, α' with $|\alpha|, |\alpha'| \le f(n)$.
- binary counter up to k (in order to compute an arbitrary $\alpha' \in R(k)$ nondeterministically)
- binary counter up to $r = r(k)$ (for **repeat** r **times**).

For this, space $\mathcal{O}(f(n))$ is sufficient.

Proof of the theorem of Immerman and Szelepcsényi

Step 3: Compute $r(k+1)$ using the function compute-r$(k+1, r)$ from $r = r(k)$.

FUNCTION compute-r$(k+1, r)$
  $r' := 0$ **(**$*$ contains $r(k+1)$ at the end $*$**)**
  $m :=$ compute-m(k, r)
  forall configurations β with $|\beta| \le m+1$ **do**
    if Reach$(r, k+1, \beta)$ **then**
      $r' := r' + 1$
    endif
  endforall
  return r'
ENDFUNC

We only have to consider configurations β with $|\beta| \le m(k)+1$, since $m(k+1) \le m(k)+1$.

Proof of the theorem of Immerman and Szelepcsényi

A successful computation of $r(*)$ is possible if and only if $w \notin L(M)$.

For this note that if $w \in L(M)$, then on every computation path the function $r(*)$ stops with „FAILURE“, since the function $m(k)$ stops on every computation path with „FAILURE“ as soon as k reaches a value such that $\alpha_0 \in R(k)$.

Therefore, as soon as the value $r(*)$ is computed, we can be sure that $w \notin L(M)$, and therefore can accept w.

Total space needed: from the previous considerations it follows that the total space needed by the algorithm is $\mathcal{O}(f(n))$. □

Translation techniques

With a translation theorem one can deduce an inclusion between small complexity classes from an inclusion between large complexity classes.

Idea: padding of languages

Let

- $L \subseteq \Sigma^*$ be a language,
- $f : \mathbb{N} \to \mathbb{N}$ a function with $\forall n \geq 0 : f(n) \geq n$, and
- $\$ \notin \Sigma$ a new symbol.

Define the language

$$\mathrm{Pad}_f(L) = \{ w\$^{f(|w|)-|w|} \mid w \in L \} \subseteq (\Sigma \cup \{\$\})^*.$$

Note: to every word from L of length n we assign a word from $L\$^*$ of length $f(n)$.

Translation theorem for time classes

Theorem 14 (Translation theorem for time classes)

Let f and g be monotone functions with $\forall n \geq 0 : f(n), g(n) \geq n$ and g be time constructible. Assume that given the unary input 1^n one can compute the output $1^{f(n)}$ in time $\mathcal{O}(f(n))$. Then, for every $L \subseteq \Sigma^*$ we have

1. $\mathrm{Pad}_f(L) \in \mathrm{DTIME}(\mathcal{O}(g)) \iff L \in \mathrm{DTIME}(\mathcal{O}(g \circ f))$,
2. $\mathrm{Pad}_f(L) \in \mathrm{NTIME}(\mathcal{O}(g)) \iff L \in \mathrm{NTIME}(\mathcal{O}(g \circ f))$.

Proof: We show the theorem only for DTIME; the proof for NTIME is analogous.

$\Rightarrow$: Let $\mathrm{Pad}_f(L) \in \mathrm{DTIME}(\mathcal{O}(g))$ and $w \in \Sigma^*$ be an input, $|w| = n$.

We decide $w \in L$ in time $\mathcal{O}(g(f(n)))$ as follows:

1. Compute the word $w\$^{f(n)-n}$ in time $\mathcal{O}(g(f(n)))$.
2. Check in time $\mathcal{O}(g(f(n)))$ whether $w\$^{f(n)-n} \in \mathrm{Pad}_f(L)$ holds.

   By definition of $\mathrm{Pad}_f(L)$ we have $w\$^{f(n)-n} \in \mathrm{Pad}_f(L) \iff w \in L$.

Proof of the translation theorem for time classes

$\Leftarrow$: Let $L \in \mathrm{DTIME}(\mathcal{O}(g \circ f))$ and let $x \in (\Sigma \cup \{\$\})^*$ be an input of length m.

We check in time $\mathcal{O}(g(m))$ whether $x \in \mathrm{Pad}_f(L)$ as follows:

1. Check in time $m \le g(m)$ whether $x \in w\$^*$ for some word $w \in \Sigma^*$.

   Let $x = w\$^{m-n}$ with $w \in \Sigma^*$, $|w| = n$.

2. Check in time $g(m)$ whether $f(n) = m$ holds:

   Compute $1^{f(n)}$ in time $\mathcal{O}(g(f(n)))$. If thereby the machine wants to do more than $g(m)$ steps (this can be detected since g is time constructible), then we can reject (since g is monotone, we have $g(f(n)) > g(m) \to f(n) > m$).

   If $1^{f(n)}$ is computed, one can compare $1^{f(n)}$ with 1^m.

   Assume now that $x = w\$^{f(n)-n}$.

3. Check in time $\mathcal{O}(g(f(n))) = \mathcal{O}(g(m))$ whether $w \in L$ holds. □

Translation theorem for space classes

Theorem 15 (Translation theorem for space classes (without proof))

Let $g \in \Omega(\log(n))$ space constructible and $f(n) \geq n$ for all $n \geq 0$. From the unary input 1^n one can compute the binary representation of $f(n)$ in space $g(f(n))$. Then, for every $L \subseteq \Sigma^*$ the following holds:

1. $\mathrm{Pad}_f(L) \in \mathrm{DSPACE}(g) \iff L \in \mathrm{DSPACE}(g \circ f)$,
2. $\mathrm{Pad}_f(L) \in \mathrm{NSPACE}(g) \iff L \in \mathrm{NSPACE}(g \circ f)$.

Consequence:

A collapse of complexity classes can be more likely to be expected at the higher end of the complexity spectrum.

It might be easier to proof the separation of complexity classes at the lower end of the complexity spectrum.

Consequences of the translation theorem for space classes

Theorem 16 (Corollary of the translation theorem for space classes)
$\mathrm{DSPACE}(n) \neq \mathrm{NSPACE}(n) \implies \mathbf{L} \neq \mathbf{NL}$.

Proof: Assume that $\mathbf{L} = \mathbf{NL}$.

Let $L \in \mathrm{NSPACE}(n) = \mathrm{NSPACE}(\log \circ \exp)$.

We get $\mathrm{Pad}_{\exp}(L) \in \mathrm{NSPACE}(\log(n)) = \mathbf{NL} = \mathbf{L} = \mathrm{DSPACE}(\log(n))$.

From the translation theorem for space classes we obtain
$L \in \mathrm{DSPACE}(\log \circ \exp) = \mathrm{DSPACE}(n)$. □

Consequences of the translation theorems

Using the translation technique one can sometimes prove that complexity classes are different.

Theorem 17 (Corollary of the translation theorems)

$\mathbf{P} \neq \mathrm{DSPACE}(n)$.

Proof: Choose a language $L \in \mathrm{DSPACE}(n^2) \setminus \mathrm{DSPACE}(n)$ (exists by the space hierarchy theorem) and the padding function $f(n) = n^2$.

We obtain $\mathrm{Pad}_f(L) \in \mathrm{DSPACE}(n)$.

Assume now that $\mathrm{DSPACE}(n) = \mathbf{P}$.

We obtain $\mathrm{Pad}_f(L) \in \mathrm{DTIME}(n^k)$ for some $k \geq 1$ and
$L \in \mathrm{DTIME}(\mathcal{O}(n^{2k})) \subseteq \mathbf{P} = \mathrm{DSPACE}(n)$.

This is a contradiction.

Consequences of the translation theorems

Remarks:

- In particular, **P** is different from the class of deterministic context-sensitive languages.
- $\mathsf{DSPACE}(\log(n)) = \mathbf{P}$, $\mathsf{DSPACE}(n) \subset \mathbf{P}$ and $\mathbf{P} \subset \mathsf{DSPACE}(n)$ are all possible according to our current knowledge.

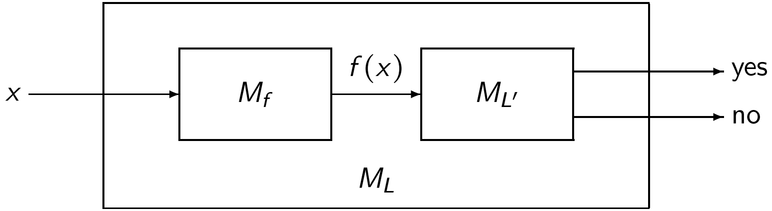
Part 3: Reductions and complete problems

Let $L \subseteq \Sigma^*$ and $L' \subseteq \Sigma'^*$ be two languages.

A reduction of L to L' is a total and computable mapping $f : \Sigma^* \to \Sigma'^*$ with $x \in L \iff f(x) \in L'$ for all x.

Assume that we have an algorithm for deciding membership in L'. Then we can check whether $x \in L$ as follows:

1. Compute the word $f(x) \in \Sigma'^*$.
2. Check, using the algorithm for L', whether $f(x) \in L'$ holds.



Polynomial time reductions

A reduction $f : \Sigma^* \to \Sigma'^*$ of L to L' is a polynomial time reduction, if f can be computed by a deterministic polynomial time bounded Turing machine.

Proposition 18

$L' \in \mathbf{P}$ and $\exists$ polynomial time reduction f of L to L' $\implies$ $L \in \mathbf{P}$.

Proof: Assume that L' belongs to $\mathrm{DTIME}(n^k)$ and that f can be computed in time n^ℓ.

For an input $x \in \Sigma^*$ of length n, we first compute $f(x)$ in time n^ℓ.

We must have $|f(x)| \le n^\ell$.

Therefore one can decide in time $(n^\ell)^k = n^{k\cdot\ell}$ whether $f(x) \in L'$ (i.e., $x \in L$) holds.

This algorithm works in time $n^\ell + n^{k\cdot\ell}$. □

Logspace reductions

Many important reductions can be computed in logarithmic space
$\Rightarrow$ logspace reductions

Definition logspace transducer

A logspace transducer is a deterministic Turing machine M with the following properties:

- M has a read-only input tape,
- M has a work tape whose length is $\mathcal{O}(\log n)$ for an input of length n,
- M has a write-only output tape.

  In each computation step of M, the machine
  - either writes a new symbol on the output tape and the head for the output tape moves on cell to the right, or
  - no new symbol is written on the output tape and the head for the output tape does not move.

Logspace reductions

Definition

1. A function $f : \Sigma^* \to \Sigma'^*$ is logspace computable, if the following holds:
   $\exists$ logspace transducer M $\forall x \in \Sigma^*$:
   M finally stops on input x with $f(x) \in \Sigma'^*$ on the work tape.
2. A language $L \subseteq \Sigma^*$ is logspace reducible to $L' \subseteq \Sigma'^*$ if there is a logspace computable function $f : \Sigma^* \to \Sigma'^*$ such that

$$\forall x \in \Sigma^* : x \in L \iff f(x) \in L'.$$

We briefly write $L \leq_m^{\log} L'$.

The lower index m stands for many-one. This refers to the fact that many words from Σ^* can be mapped by f to the same word from Σ'^*.

Logspace reductions

Remarks:

- Let $L, L' \in \mathbf{P}$, $L \subseteq \Sigma^*$, $L' \in \Sigma'^*$, $\emptyset \neq L \neq \Sigma^*$ and $\emptyset \neq L' \neq \Sigma'^*$.

  Then there is a polynomial time reduction of L to L' as well as a polynomial time reduction of L' to L:

  Let $x_0 \in \Sigma'^* \setminus L'$ and $x_1 \in L'$.

  Define the function $f : \Sigma^* \to \Sigma'^*$ by

$$f(x) = \begin{cases} x_0 & \text{if } x \in \Sigma^* \setminus L \\ x_1 & \text{if } x \in L \end{cases}$$

  Then, f is a polynomial time reduction of L to L'.

Logspace reductions

Remarks:

- Logspace reductions can be also used for complexity classes below **P** and lead to a finer classification than polynomial time reductions.
- Every logspace computable function $f : \Sigma^* \to \Sigma'^*$ can be also computed in polynomial time.
  In particular: $\exists k \geq 0 \ \forall x \in \Sigma^* : |f(x)| \leq |x|^k$.
- Logspace reductions and polynomial time reductions have equal power if and only if $\mathbf{L} = \mathbf{P}$ holds.

$\leq_m^{\log}$ is transitive

Proposition 19

$L \leq_m^{\log} L' \leq_m^{\log} L'' \implies L \leq_m^{\log} L''$ *($\leq_m^{\log}$ is transitive)*

Note: The corresponding statement for polynomial time reductions is trivial.

But when computing the composition of logspace reductions $f : \Sigma^* \to \Sigma'^*$ and $g : \Sigma'^* \to \Sigma''^*$ in the naive way (first compute $f(x)$, then compute $g(f(x))$) the following problem arises:

- for input $w \in \Sigma^*$ with $|w| = n$ we have $|f(w)| \leq n^k$ (k is a constant).
- The application of g to $f(w)$ therefore needs space $\mathcal{O}(\log(n^k)) = \mathcal{O}(\log(n))$.
- But: we cannot store $f(w)$ in logarithmic space on the work tape.

$\leq_m^{\log}$ is transitive

Proof of Proposition 19:

We compute $g(f(w))$ in space $\mathcal{O}(\log(|w|)$ as follows:

- Start the logspace transducer that computes g (without computing $f(w)$ before).
- When during the computation of g the i−th symbol of $f(w)$ is needed, then we run the logspace transducer for computing f starting from the initial configuration (with input w) until the i-th symbol of $f(w)$ is finally computed.

  The symbols of $f(w)$ at positions $1, \dots, i-1$ are not written on the output tape.

  To do this, we need a binary counter that is incremented each time the logspace transducer for f produces a new output symbol.
- Note: this binary counter needs space $\mathcal{O}(\log(|f(w)|)) = \mathcal{O}(\log(|w|)$ □

$\leq_m^{\log}$ is transitive

Example: Let $f(n) = n^k$.

The function $\$^n \mapsto \$^{f(n)}$ is logspace computable.

This implies that also the function $w \mapsto w\$^{|w|^k - |w|}$ for $w \in \Sigma^*$ is logspace computable.

Consequence: $L \leq_m^{\log} \mathrm{Pad}_f(L)$ for $L \subseteq \Sigma^*$ ($\$ \notin \Sigma$)

Vice versa, we also have $\mathrm{Pad}_f(L) \leq_m^{\log} L$ for $L \neq \Sigma^*$.

Complete problems

Definition

Let $\mathcal{C}$ be a complexity class and let $L \subseteq \Sigma^*$ be a language.

1. L is hard for $\mathcal{C}$, $\mathcal{C}$-hard for short, (with respect to logspace reductions) if $\forall K \in \mathcal{C} : K \leq_m^{\log} L$.
2. L is $\mathcal{C}$-complete (with respect to logspace reductions) if L is $\mathcal{C}$-hard and in addition $L \in \mathcal{C}$ holds.

GAP is **NL**-complete

Here is a first example:

Theorem 20

The graph reachability problem GAP is **NL***-complete.*

Proof: GAP $\in$ **NL** was already shown.

Let $L \in$ **NL** and let M be a non-deterministic $\log(n)$-space bounded Turing machine with $L = L(M)$.

We define a reduction f as follows: for $w \in \Sigma^*$ let $f(w) = (G, s, t)$, where:

- $G = (V, E)$ is the directed graph with

$$\begin{aligned} V &= \{c \mid c \text{ is a configuration for } M \text{ with input } w, |c| \le \log(|w|)\}, \\ E &= \{(c, d) \mid c, d \in V, c \vdash_M d\} \end{aligned}$$

- $s = \mathrm{Start}(w)$
- $t =$ is the (w.l.o.g.) unique accepting configuration of M.

GAP is **NL**-complete

The graph G is represented by its adjacency matrix.

The following holds:

$$w \in L(M) \iff \text{in } G \text{ there is a path from } s \text{ to } t.$$

The function f is logspace computable.

The following algorithm computes the adjacency matrix of G in logarithmic space.

forall $c \in V$ in length-lexicographic order **do**
    forall $d \in V$ in length-lexicographic order **do**
        if $c \vdash_M d$ **then write** 1
        else write 0
        endif
    endfor
    write $\#$
endfor

□

Part 4: **NP**-completeness

Theorem 21

If there is an **NP***-complete language, then there is an* **NP***-complete language in* NTIME(n)*:*

$$\exists L : L \textit{ is } \mathbf{NP}\textit{-complete} \Rightarrow \exists \tilde{L} \in \mathrm{NTIME}(n) : \tilde{L} \textit{ is } \mathbf{NP}\textit{-complete.}$$

Proof: Let L be an **NP**-complete language.

There is a constant $k > 0$ with $L \in \mathrm{NTIME}(n^k)$.

The translation theorem for time classes yields $\mathrm{Pad}_{n^k}(L) \in \mathrm{NTIME}(n)$.

Take any language $K \in \mathbf{NP}$.

$\Rightarrow K \leq_m^{\log} L \leq_m^{\log} \mathrm{Pad}_{n^k}(L)$

Since $\leq_m^{\log}$ is transitive, we have $K \leq_m^{\log} \mathrm{Pad}_{n^k}(L)$.

$\Rightarrow \mathrm{Pad}_{n^k}(L)$ is **NP**-complete. □

The generic **NP**-complete problem

Let $\langle w, M\rangle$ be the encoding of a word $w \in \Sigma^*$ and a non-deterministic Turing machine M.

$$L_{\text{Gen}} = \{\langle w, M\rangle\,\$^m \mid w \in \Sigma^*, M \text{ non-deterministic Turing machine}, m \in \mathbb{N}, M \text{ has on input } w \text{ an accepting computation of length} \leq m\}$$

Theorem 22

L_{Gen} is **NP***-complete.*

Proof:

$L_{\text{Gen}} \in$ **NP**:

For an input $\langle w, M\rangle\,\$^m$ one simulates M on input w non-deterministically for at most m steps.

This is a non-deterministic polynomial time algorithm for L_{Gen}.

The generic **NP**-complete problem

L_{Gen} is **NP**-hard:

Let $L \in \mathbf{NP}$ and M a n^k-time bounded non-deterministic Turing machine with $L = L(M)$ (k is a constant).

The reduction of L to L_{Gen} computes in logarithmic space on input $w \in \Sigma^*$ the output

$$f(w) = \langle w, M \rangle \$^{|w|^k}.$$

We get: $w \in L(M) \iff f(w) \in L_{Gen}$. □

Cook's theorem

Let $\Sigma_0 = \{\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow, 0, 1, (,), x\}$.

Let $\mathbb{A} \subseteq \Sigma_0^*$ be the set of all propositional formulas with variables from the set $V = x1\{0,1\}^*$.

$\mathbb{A} \subseteq \Sigma_0^*$ is a deterministic context-free language and therefore belongs to DTIME(n).

A propositional formula F is satisfiable if there is a mapping $\mathcal{B} : \mathrm{Var}(F) \to \{\mathbf{true}, \mathbf{false}\}$ from the variables that appear in F to truth values such that F evaluates to **true** when each variable $y \in \mathrm{Var}(F)$ is replaced by $\mathcal{B}(y)$.

Let SAT $= \{F \in \mathbb{A} \mid F \text{ is satisfiable}\}$.

Theorem 23 (Cook's theorem)

SAT *is* **NP**-*complete.*

Proof of Cook's theorem

(A) SAT $\in$ **NP**: For a word $F \in \Sigma_0^*$ we verifiy "$F \in$ SAT" as follows:

1. Check in time $\mathcal{O}(|F|)$ whether $F \in \mathbb{A}$ holds.
2. If "YES", guess a mapping $\mathcal{B} : \mathrm{Var}(F) \to \{\mathbf{true}, \mathbf{false}\}$.
3. Accept, if F evaluates to **true** under $\mathcal{B}$.

(B) SAT is **NP**-complete.

Let $L \in \mathbf{NP}$.

Given $w \in \Sigma^*$, we construct a formula $f(w)$ with

$$w \in L \iff f(w) \text{ is satisfiable.}$$

The mapping f must be logspace computable.

Proof of Cook's theorem

Let $M = (Q, \Sigma, \Gamma, \delta, q_0, q_J, q_N, \Box)$ be a $p(n)$-time bounded non-deterministic Turing machine with $L = L(M)$ ($p(n) > n$ is a polynomial).

Let $w = w_1 w_2 \cdots w_n \in \Sigma^*$ be an input of length n (w.l.o.g. $n \geq 1$).

W.l.o.g. M has the following properties:

1. M has only one tape, whose initial content is $\cdots \Box \Box w \Box \Box \cdots$, and the cells on the tape can be read and written during the computation.
2. The end markers $\triangleright$ and $\triangleleft$ are not needed.
3. M accepts w if and only if M is in state q_J after exactly $p(n)$ steps, and the read/write head returns to its initial position, where a $\Box$ is in the cell.
4. If $(p_1, a_1, d_1), (p_2, a_2, d_2) \in \delta(q, a)$ then $a_1 = a_2$ and $d_1 = d_2$.

Proof of Cook's theorem

Point 4 from slide 88 can be ensured as follows: define a new non-deterministic Turing machine

$$M' = (Q', \Sigma, \Gamma, \delta', q_0, q_J, q_N, \Box)$$

with

- $Q' = Q \cup (Q \times \Gamma \times \{-1, 0, 1\})$,
- for all $q \in Q, a \in \Gamma$ let

$$\delta'(q, a) = \{((p, b, d), a, 0) \mid (p, b, d) \in \delta(q, a)\},$$

- for all $(p, b, d) \in Q \times \Gamma \times \{-1, 0, 1\}$ and all $a \in \Gamma$ let

$$\delta'((p, b, d), a) = \{(p, b, d)\}.$$

We have $L(M) = L(M')$ and M' is polynomial time bounded.

Proof of Cook's theorem

Every configuration that can be reached from the start configuration can be described by a word of the form

$$\mathrm{Conf} = \{\Box u(q,a)v\Box \mid (q,a) \in Q \times \Gamma,\ uv \in \Gamma^{2p(n)}\}.$$

The start configuration is $\Box^{p(n)+1}(q_0, w_1)w_2\cdots w_n\Box^{p(n)-n+2}$.

Let $\Omega = (Q \times \Gamma) \cup \Gamma$.

Notation: For $\alpha \in \mathrm{Conf}$ we write

$$\alpha = \alpha[-p(n)-1]\alpha[-p(n)]\cdots\alpha[p(n)]\alpha[p(n)+1],$$

where

- $\alpha[-p(n)-1] = \alpha[p(n)+1] = \Box$ and
- $\alpha[-p(n)],\ldots,\alpha[p(n)] \in \Omega$.

Proof of Cook's theorem

Assume that $\alpha, \alpha' \in \text{Conf}$ with $\alpha \vdash_M \alpha'$ and $-p(n) \le i \le p(n)$.

$\alpha[i-1], \alpha[i]$ and $\alpha[i+1]$ determine which symbols are possible for $\alpha'[i]$.

Example:

If $(p, a', -1) \in \delta(q, a)$ then the following local tape modification is possible:

position			i–2	i–1	i	i+1	i+2		
α =	$\cdots$	$\cdots$	b'	b	q, a	c	c'	$\cdots$	$\cdots$
α' =	$\cdots$	$\cdots$	b'	p, b	a'	c	c'	$\cdots$	$\cdots$

If $(p, a', +1) \in \delta(q, a)$ then the following local tape modification is possible:

position			i–2	i–1	i	i+1	i+2		
α =	$\cdots$	$\cdots$	b'	b	q, a	c	c'	$\cdots$	$\cdots$
α' =	$\cdots$	$\cdots$	b'	b	a'	p, c	c'	$\cdots$	$\cdots$

Proof of Cook's theorem

We define $\Delta \subseteq \Omega^4$ as the set of all such 4-tuples $(\alpha[i-1], \alpha[i], \alpha[i+1], \alpha'[i])$:

- (a, b, c, b) with $a, b, c \in \Gamma$
- $(b, c, (q, a), (p, c))$, $(b, (q, a), c, a')$, $((q, a), b, c, b)$, where $(p, a', -1) \in \delta(q, a)$, $b, c \in \Gamma$
- $(b, c, (q, a), c)$, $(b, (q, a), c, a')$, $((q, a), b, c, (p, b))$, where $(p, a', +1) \in \delta(q, a)$, $b, c \in \Gamma$

We then obtain for all $\alpha, \alpha' \in \square\Omega^*\square$ with $|\alpha| = |\alpha'|$:

$$\alpha, \alpha' \in \text{Conf and } \alpha \vdash_M \alpha'$$
$$\Longleftrightarrow$$
$$\alpha \in \text{Conf and } \forall i \in \{-p(n), \dots, p(n)\} : (\alpha[i-1], \alpha[i], \alpha[i+1], \alpha'[i]) \in \Delta.$$

For this, point 4 from slide 88 is important!

Proof of Cook's theorem

A computation of M can be described by a matrix of the following form:

$$
\begin{array}{lcccccccc}
\alpha_0 & = & \Box & \alpha_{0,-p(n)} & \alpha_{0,-p(n)+1} & \cdots & \alpha_{0,p(n)} & \Box \\
\alpha_1 & = & \Box & \alpha_{1,-p(n)} & \alpha_{1,-p(n)+1} & \cdots & \alpha_{1,p(n)} & \Box \\
 & & & & \vdots & & & \\
\alpha_{p(n)} & = & \Box & \alpha_{p(n),0} & \alpha_{p(n),1} & \cdots & \alpha_{p(n),p(n)} & \Box
\end{array}
$$

For every triple (a, i, t) $(a \in \Omega,\ -p(n) - 1 \le i \le p(n) + 1,\ 0 \le t \le p(n))$ let $x(a, i, t)$ be a propositional variable.

Interpretation: $x(a, i, t) =$ **true** if and only if at the configuration at time t, the i-th symbol is an a.

Proof of Cook's theorem

At positions $-p(n)-1$ and $p(n)+1$ there is always a $\Box$:

$$G(n) = \bigwedge_{0 \le t \le p(n)} \Big(x(\Box, -p(n)-1, t) \wedge x(\Box, p(n)+1, t) \Big)$$

For every pair (i,t), exactly one variable $x(a,i,t)$ is true (at every time instant, a tape cell contains exactly one symbol):

$$X(n) = \bigwedge_{\substack{0 \le t \le p(n) \\ -p(n)-1 \le i \le p(n)+1}} \left(\bigvee_{a \in \Omega} \left(x(a,i,t) \wedge \bigwedge_{b \in \Omega \setminus \{a\}} \neg x(b,i,t) \right) \right)$$

Proof of Cook's theorem

At time instant $t = 0$, the configuration is $\Box^{p(n)+1}(q_0, w_1)w_2 \cdots w_n \Box^{p(n)-n+2}$:

$$S(w) = \bigwedge_{i=1}^{p(n)} x(\Box, -i, 0) \wedge x((q_0, w_1), 0, 0) \wedge \bigwedge_{i=1}^{n-1} x(w_{i+1}, i, 0) \wedge \bigwedge_{i=n}^{p(n)} x(\Box, i, 0)$$

The computation "respects" the local relation Δ:

$$D(n) = \bigwedge_{\substack{-p(n) \le i \le p(n) \\ 0 \le t \le p(n)-1}} \quad \bigvee_{(a,b,c,d) \in \Delta} \begin{pmatrix} x(a, i-1, t) \ \wedge \ x(b, i, t) \ \wedge \\ x(c, i+1, t) \ \wedge \ x(d, i, t+1) \end{pmatrix}$$

Proof of Cook's theorem

Finally, we define the formula

$$f(w) = G(n) \wedge X(n) \wedge S(w) \wedge D(n) \wedge x((q_J, \Box), 0, p(n)).$$

Then there is a natural bijection between set of all satisfying assignments for $f(w)$ and the set of all accepting computations of M on input w.

Therefore we have:

$$f(w) \text{ is satisfiable} \quad \iff \quad w \in L.$$

Number of variables in $f(w)$: $\mathcal{O}(p(n)^2)$

Length of $f(w)$: $\mathcal{O}(p(n)^2 \log p(n))$

The factor $\mathcal{O}(\log p(n))$ is needed since the indices of the variables need $\log p(n)$ many bits. $\Box$

Further **NP**-complete problems: (1) SAT ∩ KNF

Definition: literals, CNF

A literal $\tilde{x}$ is a propositional variable or a negated propositional variable.

Instead of $\neg x$, we also write $\overline{x}$. Moreover, we set $\overline{\overline{x}} = x$.

Let CNF (resp. DNF) be the set of all propositional formulas in conjunctive normal form (resp. disjunctive normal form):

$$\begin{aligned} \text{DNF} &= \{F \mid F \text{ is a disjunction of conjunctions of literals}\} \\ \text{CNF} &= \{F \mid F \text{ is a conjunction of disjunctions of literals}\} \end{aligned}$$

Fact: For every propositional formula F there is an equivalent formula $\text{DNF}(F) \in \text{DNF}$ and $\text{CNF}(F) \in \text{CNF}$.

Further **NP**-complete problems: (1) SAT ∩ CNF

Example:

$$F = \bigwedge_{i=1,\ldots,k} \left(\bigvee_{j=1,\ldots,m} \tilde{x}_{i,j} \right) \equiv \bigvee_{f \in \{1,\ldots,m\}^{\{1,\ldots,k\}}} \left(\bigwedge_{i=1,\ldots,k} \tilde{x}_{i,f(i)} \right) = F'$$

Note:

- $|F| = m \cdot k$, whereas $|F'| = m^k \cdot k$. Thus, a CNF-formula with k disjunctions of length m can be transformed into an equivalent DNF-formula consisting of m^k conjunctions of length k.
- For DNF-formulas, satisfiability can be checked deterministically in quadratic time.
- In a moment we will see that satisfiability for CNF-formulas is **NP**-complete.
  Therefore the exponential blow-up in the transformation from CNF to DNF is not surprising.

SAT $\cap$ CNF is **NP**-complete

Theorem 24

SAT $\cap$ CNF is **NP**-complete.

Proof:

(1) SAT $\cap$ CNF $\in$ **NP**: clear, because (i) SAT $\in$ **NP** and (ii) for a formula of length n it can be checked in time $\mathcal{O}(n)$ whether the formula is in CNF.

(2) SAT $\cap$ CNF is **NP**-hard:

Proof 1: In the proof of the **NP**-hardness of SAT, we have constructed a formula that is already in CNF up to subformulas of constant length.

Using a logspace transducer we can transform those constant-size subformulas into CNF and thereby obtain a CNF-formula.

SAT ∩ CNF is **NP**-complete

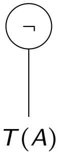
Proof 2: We show SAT $\leq_m^{\log}$ SAT ∩ CNF.

For this we have to come up with a logspace-computable mapping $f : \mathbb{A} \to \mathsf{CNF}$ such that:

$$F \in \mathsf{SAT} \iff f(F) \in \mathsf{SAT} \cap \mathsf{CNF}.$$

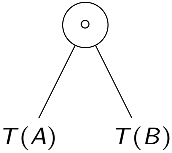
We can view a formula $F \in \mathbb{A}$ as a tree $T(F)$ that can be built recursively as follows:

1. For a variable x let $T(x) = x$.
2. If F is the negation of a formula A, i.e., $F = \neg A$, then $T(F)$ has the following form:



SAT ∩ CNF is **NP**-complete

3. If F has the form $F = A \circ B$ for formulas A, B and $\circ \in \{\Leftrightarrow, \Rightarrow, \wedge, \vee\}$, then $T(F)$ has the following form:

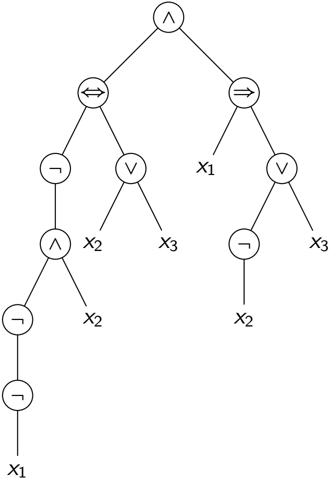


Example: For the formula

$$F = \Big(\big((\neg(\neg\neg x_1 \wedge x_2)) \Leftrightarrow (x_2 \vee x_3)\big) \wedge \big(x_1 \Rightarrow (\neg x_2 \vee x_3)\big)\Big)$$

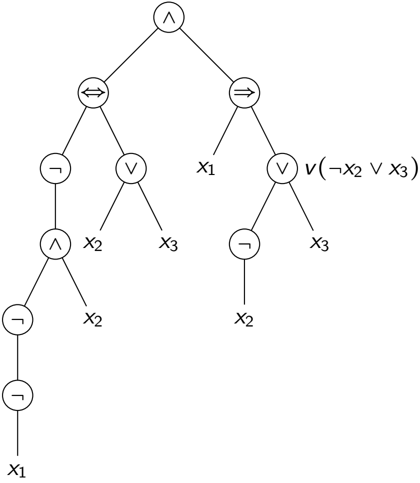
we obtain the tree $T(F)$ from the next slide.

SAT ∩ CNF is **NP**-complete



To each node of $T(F)$ we assign a new variable $v(A)$, where A is the subformula of F represented by the node.

SAT ∩ CNF is **NP**-complete



To each node of $T(F)$ we assign a new variable $v(A)$, where A is the subformula of F represented by the node.

SAT ∩ CNF is **NP**-complete

Define an auxiliary function $f' : \mathbb{A} \to \mathrm{SAT} \cap \mathrm{CNF}$ recursively as follows:

1. If $F = x$ then $f'(F) := \mathrm{CNF}\big(v(x) \Leftrightarrow x\big)$.
2. If $F = A \circ B$ with $\circ \in \{\Leftrightarrow, \Leftarrow, \wedge, \vee\}$ then

$$f'(F) := \Big(\mathrm{CNF}\big(v(F) \Leftrightarrow (v(A) \circ v(B))\big) \wedge f'(A) \wedge f'(B)\Big).$$

3. If $F = \neg A$ then

$$f'(F) := \Big(\mathrm{CNF}\big(v(F) \Leftrightarrow \neg v(A)\big) \wedge f'(A)\Big).$$

   The latter formula is equivalent to

$$f'(F) = \Big(\big(\, v(F) \vee v(A)\big) \wedge \big(\, \neg v(F) \vee \neg v(A)\,\big) \wedge f'(A)\Big).$$

SAT ∩ KNF is **NP**-complete

Note: In the definition of f' (which is not the actual reduction), we apply CNF only to formulas of constant length.

In the following, let $V(G)$ be the set of all variables that appear in a formula $G \in \mathbb{A}$.

Note: $V(G) \subseteq V(f'(G))$

Lemma

1. $f'(F)$ is always satisfiable.
2. Let $\sigma : V(f'(F)) \to \{0,1\}$ such that $\sigma(f'(F)) = 1$ and let σ' be the restriction of σ to $V(F)$. We then have $\sigma'(F) = \sigma(v(F))$.
3. For every $\sigma' : V(F) \to \{0,1\}$ there is some $\sigma : V(f'(F)) \to \{0,1\}$ with $\sigma(f'(F)) = 1$ and $\sigma'(x) = \sigma(x)$ for all $x \in V(F)$.

SAT ∩ KNF is **NP**-complete

Proof of (2): Let $\sigma : V(f'(F)) \to \{0,1\}$ such that $\sigma(f'(F)) = 1$ and let σ' be the restriction of σ to $V(F)$.

Using induction over the structure of F, we show that $\sigma'(F) = \sigma(v(F))$:

Case 1: $F = x \in V(F)$. We have

$$\sigma(f'(F)) = \sigma(\mathrm{CNF}(v(x) \Leftrightarrow x)) = \sigma(v(x) \Leftrightarrow x) = 1$$

and hence $\sigma(v(F)) = \sigma(v(x)) = \sigma(x) = \sigma'(x) = \sigma'(F)$.

Case 2: $F = A \circ B$ with $\circ \in \{\Leftrightarrow, \Leftarrow, \wedge, \vee\}$. We have

$$\begin{aligned}
\sigma(f'(F)) &= \sigma(\mathrm{CNF}\big(v(F) \Leftrightarrow (v(A) \circ v(B))\big) \wedge f'(A) \wedge f'(B)) \\
&= \sigma(v(F) \Leftrightarrow (v(A) \circ v(B))) \wedge \sigma(f'(A)) \wedge \sigma(f'(B)) \\
&= 1.
\end{aligned}$$

By induction, we have $\sigma'(A) = \sigma(v(A))$ and $\sigma'(B) = \sigma(v(B))$.

SAT ∩ KNF is **NP**-complete

Moreover, we have $\sigma(v(F)) = \sigma(v(A) \circ v(B))$.

We obtain $\sigma(v(F)) = \sigma(v(A) \circ v(B)) = \sigma'(A \circ B) = \sigma'(F)$.

Case 3: $F = \neg A$: analogous to Case 2.

Proof of (3): Let $\sigma' : V(F) \to \{0,1\}$ be arbitrary.

Define $\sigma : V(f'(F)) \to \{0,1\}$ inductively as follows:

$$\begin{aligned}
\sigma(x) &= \sigma'(x) \text{ for all } x \in V(F) \\
\sigma(v(x)) &= \sigma'(x) \text{ for all } x \in V(F) \\
\sigma(v(G)) &= \sigma(v(A) \circ v(B)) \text{ if } G = A \circ B \\
\sigma(v(G)) &= \sigma(\neg v(A)) \text{ if } G = \neg A
\end{aligned}$$

Using induction over the structure of F, we directly obtain $\sigma(f'(F)) = 1$.

Point (1) follows directly from point (3).

SAT ∩ KNF is **NP**-complete

Finally, we define our reduction $f : \mathbb{A} \to \text{CNF}$ as

$$f(F) := f'(F) \wedge v(F).$$

Claim: $f(F)$ is satisfiable if and only if F is satisfiable.

Proof of the claim:

(A) Let $\sigma' : V(F) \to \{0,1\}$ such that $\sigma'(F) = 1$.

By point (3) of the lemma there is $\sigma : V(f'(F)) \to \{0,1\}$ such that $\sigma(f'(F)) = 1$ and $\sigma(x) = \sigma'(x)$ for all $x \in V(F)$.

Point (2) implies $\sigma(v(F)) = \sigma'(F) = 1$.

Hence, we have $\sigma(f'(F) \wedge v(F)) = 1$.

(B) Let $\sigma : V(f'(F) \wedge v(F)) \to \{0,1\}$ such that $\sigma(f'(F) \wedge v(F)) = 1$.

For the restriction σ' to the variables in $V(F)$ we obtain from point (2): $\sigma'(F) = \sigma(v(F)) = 1$. □

3-SAT is **NP**-complete

Definition: 3-SAT

Let 3-CNF be the set of CNF-formulas with exactly three literals in each clause:

$$\text{3-CNF} := \{F \in \text{CNF} \mid \text{every clause in } F \text{ contains exactly three literals}\}$$

3-SAT is the set of satisfiable formulas from 3-CNF:

$$\text{3-SAT} := \text{3-CNF} \cap \text{SAT}$$

Theorem 25

*3-*SAT *is* **NP***-complete.*

Proof: Only the **NP**-hardness has to be shown.

We show: $\text{SAT} \cap \text{CNF} \leq_m^{\log} \text{3-SAT}$.

Let F be a CNF-formula. We distinguish three cases:

3-SAT is **NP**-complete

1. F contains a clause $(\tilde{x})$ with only one literal.
   We introduce a new variable y and replace the clause $(\tilde{x})$ by $(\tilde{x} \vee y) \wedge (\tilde{x} \vee \overline{y})$.
   This has no influence on the satisfiability of F.
2. F contains a clause $(\tilde{x} \vee \tilde{y})$ with two literals.
   We introduce a new variable z and replace $(\tilde{x} \vee \tilde{y})$ by $(\tilde{x} \vee \tilde{y} \vee z) \wedge (\tilde{x} \vee \tilde{y} \vee \overline{z})$.
3. F contains a clause c with more than three literals.
   Let $c = (\tilde{x}_1 \vee \tilde{x}_2 \vee \cdots \vee \tilde{x}_k)$ with $k \geq 4$.
   We introduce $k-3$ new variables $v(\tilde{x}_3), v(\tilde{x}_4), \ldots, v(\tilde{x}_{k-2}), v(\tilde{x}_{k-1})$ and replace c by

$$
\begin{aligned}
c' \quad &= \quad \big(\tilde{x}_1 \vee \tilde{x}_2 \vee v(\tilde{x}_3)\big) \wedge \bigwedge_{j=3}^{k-2} \big(\neg v(\tilde{x}_j) \vee \tilde{x}_j \vee v(\tilde{x}_{j+1})\big) \\
&\quad \wedge \big(\neg v(\tilde{x}_{k-1}) \vee \tilde{x}_{k-1} \vee \tilde{x}_k\big).
\end{aligned}
$$

3-SAT is **NP**-complete

Note: c' can be also written as

$$c' = \left(\tilde{x}_1 \vee \tilde{x}_2 \vee v(\tilde{x}_3)\right) \wedge \bigwedge_{j=3}^{k-2} \left(v(\tilde{x}_j) \Rightarrow \tilde{x}_j \vee v(\tilde{x}_{j+1})\right)$$
$$\wedge \left(v(\tilde{x}_{k-1}) \Rightarrow \tilde{x}_{k-1} \vee \tilde{x}_k \right).$$

That (3) does not change the (non)satisfiability can be seen as follows:

(A) Assume that $\sigma : V(c) \to \{0,1\}$ satisfies c.

We must have $\sigma(\tilde{x}_l) = 1$ for some $1 \leq l \leq k$.

We extend σ to σ' by:

$$\sigma'(v(\tilde{x}_p)) = \begin{cases} 1 & \text{falls } p \leq l \\ 0 & \text{falls } p > l \end{cases}$$

We then have $\sigma'(c') = 1$:

3-SAT is **NP**-complete

The unique clause, in which $\tilde{x}_l$ appears is satisfied.

In all other clauses, either a $v(\tilde{x}_p)$ with $p \le l$ or a $\neg v(\tilde{x}_p)$ with $p > l$ appears.

(B) Let $\sigma' : V(c') \to \{0,1\}$ with $\sigma'(c') = 1$.

Assume that $\sigma'(\tilde{x}_i) = 0$ for all $1 \le i \le k$.

We must have $\sigma'(v(\tilde{x}_3)) = 1$ (since $\sigma'(\tilde{x}_1 \vee \tilde{x}_2 \vee v(\tilde{x}_3)) = 1$).

By induction, we get $\sigma'(v(\tilde{x}_i)) = 1$ für all $3 \le i \le k-1$.

We obtain $\sigma'(\neg v(\tilde{x}_{k-1}) \vee \tilde{x}_{k-1} \vee \tilde{x}_k)) = 0$. contradiction! □

Integer Programming

Let $\mathrm{LinProg}(\mathbb{Z}) := \{\langle A, b\rangle \mid A \in \mathbb{Z}^{m\times n},\ b \in \mathbb{Z}^{m\times 1},\ \exists x \in \mathbb{Z}^{n\times 1} : Ax \geq b\}$.

Numbers from $\mathbb{Z}$ are coded in binary notation.

Theorem 26

$\mathrm{LinProg}(\mathbb{Z})$ *is* **NP***-complete.*

Proof:

(1) $\mathrm{LinProg}(\mathbb{Z}) \in$ **NP**:

This is the hard part of the proof, which we skip; see e.g. Hopcroft, Ullman; *Introduction to Automata Theory, Languages and Computation*, Addison Wesley 1979.

Integer Programming

(2) $\mathrm{LinProg}(\mathbb{Z})$ is **NP**-hard.

We show 3-SAT $\leq_m^{\log}$ LinProg($\mathbb{Z}$).

Let $F = c_1 \wedge c_2 \wedge \cdots \wedge c_q$ be a 3-CNF formula.

Let $x_1, \ldots, x_n$ be the variables in F.

We construct a system S of linear inequalities with variables $x_i, \overline{x_i}, 1 \leq i \leq n$ and coefficients from $\mathbb{Z}$:

1. $x_i \geq 0,\ 1 \leq i \leq n$
2. $\overline{x_i} \geq 0,\ 1 \leq i \leq n$
3. $x_i + \overline{x_i} \geq 1,\ 1 \leq i \leq n$
4. $-x_i - \overline{x_i} \geq -1,\ 1 \leq i \leq n$
5. $\tilde{x}_{j1} + \tilde{x}_{j2} + \tilde{x}_{j3} \geq 1$ for every clause $c_j = (\tilde{x}_{j1} \vee \tilde{x}_{j2} \vee \tilde{x}_{j3})$.

Integer Programming

$$\begin{aligned}
\text{(3) and (4)} &\implies x_i + \overline{x_i} = 1 \\
\text{(1) and (2)} &\implies x_i = 1, \overline{x_i} = 0 \text{ or } x_i = 0, \overline{x_i} = 1 \\
\text{(5)} &\implies \text{ in every clause } c_j \text{ at least one literal} \\
& \tilde{x}_{ij} \text{ is } 1
\end{aligned}$$

Hence: S is solvable if and only if F is satisfiable.

Size of S: $4n + q$ inequalities, $2n$ variables.

We can write S in matrix form $Ax \geq b$ so that A (resp. b) has $(4n + q) \times 2n$ (resp. $4n + q$) entires of absolute value ≤ 1. □

Remarks:

- The above proof shows that $\mathrm{LinProg}(\mathbb{Z})$ is already **NP**-hard if numbers are given in unary encoding.
- $\mathrm{LinProg}(\mathbb{Q}) \in$ **P**. This is a difficult result that was first shown by Khachiyan using his *ellipsoid method*.

Subset Sum

Subset Sum is the following problem:

input: a list of binary encoded numbers $t, w_1, \ldots, w_k \in \mathbb{N}$

question: Does there exists a subset $S \subseteq \{w_1, \ldots, w_k\}$ such that $\sum_{w \in S} w = t$?

Theorem 27 (without proof)

Subset Sum is **NP***-complete.*

Note that in Subset Sum the input numbers are given in binary representation.

This is important:

Theorem 28 (without proof)

The variant of Subset Sum, where the input numbers $t, w_1, \ldots, w_k \in \mathbb{N}$ are given in unary encoding belongs to the complexity class $\mathbf{L} \subseteq \mathbf{P}$.

Vertex Cover is **NP**-complete

A vertex cover for an undirected graph $G = (V, E)$ is a subset $C \subseteq V$ such that for every edge $\{u, v\} \in E$: $\{u, v\} \cap C \neq \emptyset$

Vertex Cover (VC) is the following problem:

input: An undirected graph $G = (V, E)$ and $k \geq 0$.

question: Does G have a vertex cover C with $|C| \leq k$?

Theorem 29

VC *is* **NP***-complete.*

Proof:

(1) VC $\in$ **NP**: Guess a subset C of vertices with $|C| \leq k$ and check in polynomial time, whether C is a vertex cover.

(1) VC is **NP**-hard:

We show 3-SAT $\leq_m^{\log}$ VC.

Vertex Cover is **NP**-hard

Let

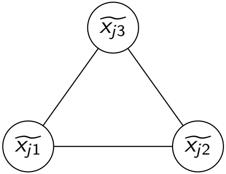
$$F = c_1 \wedge \cdots \wedge c_q$$

be a formula in 3-CNF, where

$$c_j = (\widetilde{x_{j1}} \vee \widetilde{x_{j2}} \vee \widetilde{x_{j3}}).$$

We construct a graph $G(F)$:

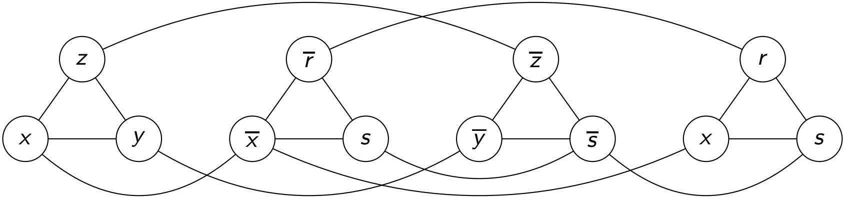
First we construct for every clause $c_j = (\widetilde{x_{j1}} \vee \widetilde{x_{j2}} \vee \widetilde{x_{j3}})$ the following graph $G(c_j)$:



Vertex Cover is **NP**-hard

The graph $G(F)$ is obtained from the disjoint union $\bigcup_{j=1}^{q} G(c_j)$ of all subgraphs $G(c_j)$ by adding all edges $(x, \overline{x})$ (x is a variable from F).

Example:
For the formula $F = (x \vee y \vee z) \wedge (\overline{x} \vee s \vee \overline{r}) \wedge (\overline{y} \vee \overline{s} \vee \overline{z}) \wedge (x \vee s \vee r)$ we obtain the following graph $G(F)$:



Vertex Cover is **NP**-hard

Note: Every vertex cover U for $G(F)$ must have at least $2q$ vertices, since U must contain from each of the q triangles at least 2 vertices.

Claim: $F \in 3\text{-SAT}$ if and only if $G(F)$ has a vertex cover U with $|U| = 2q$.

(A) Let σ be a satisfying truth assignment for the variables in F: $\sigma(F) = 1$.

Thus, for every clause c_j at least one of the literals $\widetilde{x_{ji}}$ is true.

Let U be a vertex set, that contains for every triangle graph $G(c_j)$ exactly two literals such that all false literals belong to U.

We have $|U| = 2q$ and U is a vertex cover.

Vertex Cover is **NP**-hard

(B) Let U be a vertex cover with $|U| = 2q$.

U must contain from every triangle graph $G(c_j)$ exactly two vertices.

Define a truth assignment σ for the variables in F:

$$\sigma(x) = \begin{cases} 1 & \text{if a copy of } x \text{ does not belong to } U. \\ 0 & \text{if a copy of } \overline{x} \text{ does not belong to } U. \\ 0 & \text{if all copies of } x \text{ and } \overline{x} \text{ belong to } U. \end{cases}$$

Note: Since U is a vertex cover and the graph $G(F)$ contains all edges of the form $(x, \overline{x})$, a variable x cannot be set to 0 and at the same time to 1.

We have $\sigma(F) = 1$! □

Hamilton Circuit and Hamilton Path are **NP**-complete

A Hamilton path in a finite directed graph $G = (V, E)$ is a sequence of vertices $v_1, v_2, \ldots, v_n$ with

- $(v_i, v_{i+1}) \in E$ for all $1 \le i \le n-1$ and
- for every vertex $v \in V$ there is exactly one $1 \le i \le n$ with $v = v_i$.

A Hamilton circuit is a Hamilton path $v_1, v_2, \ldots, v_n$ with $(v_n, v_1) \in E$.

Let

$$\begin{aligned} \text{HP} &= \{G \mid G \text{ is a finite graph with a Hamilton path}\} \\ \text{HC} &= \{G \mid G \text{ is a finite graph with a Hamilton circuit}\} \end{aligned}$$

Theorem 30

HP *and* HC *are* **NP***-complete (even for undirected graphs).*

Hamilton Circuit and Hamilton Path are **NP**-complete

Proof: We show the NP-completeness of HC.

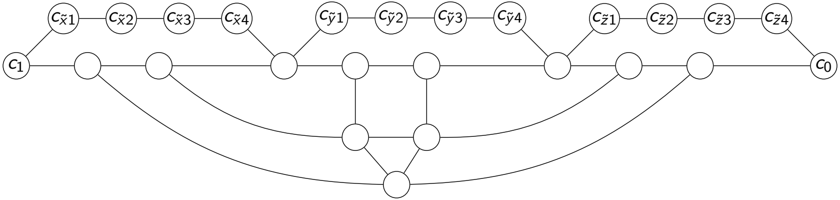
(A) HC $\in$ **NP**: trivial.

(B) 3-SAT $\leq_m^{\log}$ HC:

Let $F = \bigwedge_{c \in C} c$ be a formula in 3-CNF. Every clause $c \in C$ consists of 3 literals and we view c as a set of 3 literals.

We construct a graph $G(F)$ which contains a Hamilton circuit if and only if $F \in$ SAT.

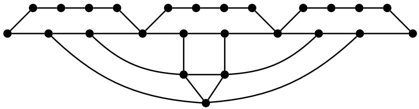
We define for every clause $c = (\tilde{x} \vee \tilde{y} \vee \tilde{z}) \in C$ the graph $G(c)$:



Hamilton Circuit and Hamilton Path are **NP**-complete

Claim 1:

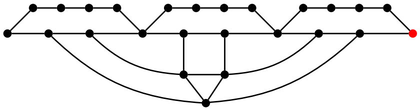
- In $G(c)$ there is no Hamilton path from c_0 to c_1.
- If one removes from $G(c)$ at least one of the paths $c_{\ell 1} - c_{\ell 2} - c_{\ell 3} - c_{\ell 4}$, $\ell \in c$, then there is a Hamilton path from c_0 to c_1.



Hamilton Circuit and Hamilton Path are **NP**-complete

Claim 1:

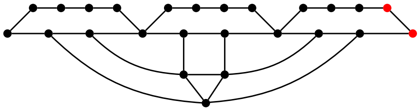
- In $G(c)$ there is no Hamilton path from c_0 to c_1.
- If one removes from $G(c)$ at least one of the paths $c_{\ell 1} - c_{\ell 2} - c_{\ell 3} - c_{\ell 4}$, $\ell \in c$, then there is a Hamilton path from c_0 to c_1.



Hamilton Circuit and Hamilton Path are **NP**-complete

Claim 1:

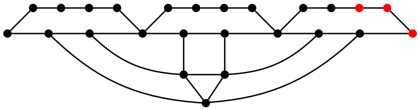
- In $G(c)$ there is no Hamilton path from c_0 to c_1.
- If one removes from $G(c)$ at least one of the paths $c_{\ell 1} - c_{\ell 2} - c_{\ell 3} - c_{\ell 4}$, $\ell \in c$, then there is a Hamilton path from c_0 to c_1.



Hamilton Circuit and Hamilton Path are **NP**-complete

Claim 1:

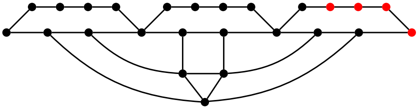
- In $G(c)$ there is no Hamilton path from c_0 to c_1.
- If one removes from $G(c)$ at least one of the paths $c_{\ell 1} - c_{\ell 2} - c_{\ell 3} - c_{\ell 4}$, $\ell \in c$, then there is a Hamilton path from c_0 to c_1.



Hamilton Circuit and Hamilton Path are **NP**-complete

Claim 1:

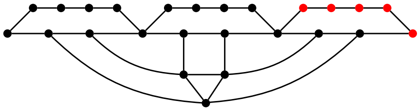
- In $G(c)$ there is no Hamilton path from c_0 to c_1.
- If one removes from $G(c)$ at least one of the paths $c_{\ell 1} - c_{\ell 2} - c_{\ell 3} - c_{\ell 4}$, $\ell \in c$, then there is a Hamilton path from c_0 to c_1.



Hamilton Circuit and Hamilton Path are **NP**-complete

Claim 1:

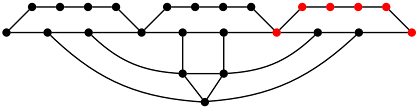
- In $G(c)$ there is no Hamilton path from c_0 to c_1.
- If one removes from $G(c)$ at least one of the paths $c_{\ell 1} - c_{\ell 2} - c_{\ell 3} - c_{\ell 4}$, $\ell \in c$, then there is a Hamilton path from c_0 to c_1.



Hamilton Circuit and Hamilton Path are **NP**-complete

Claim 1:

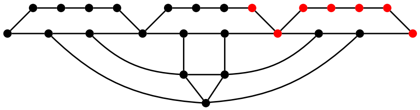
- In $G(c)$ there is no Hamilton path from c_0 to c_1.
- If one removes from $G(c)$ at least one of the paths $c_{\ell 1} - c_{\ell 2} - c_{\ell 3} - c_{\ell 4}$, $\ell \in c$, then there is a Hamilton path from c_0 to c_1.



Hamilton Circuit and Hamilton Path are **NP**-complete

Claim 1:

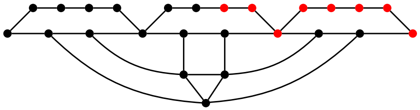
- In $G(c)$ there is no Hamilton path from c_0 to c_1.
- If one removes from $G(c)$ at least one of the paths $c_{\ell 1} - c_{\ell 2} - c_{\ell 3} - c_{\ell 4}$, $\ell \in c$, then there is a Hamilton path from c_0 to c_1.



Hamilton Circuit and Hamilton Path are **NP**-complete

Claim 1:

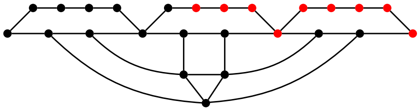
- In $G(c)$ there is no Hamilton path from c_0 to c_1.
- If one removes from $G(c)$ at least one of the paths $c_{\ell 1} - c_{\ell 2} - c_{\ell 3} - c_{\ell 4}$, $\ell \in c$, then there is a Hamilton path from c_0 to c_1.



Hamilton Circuit and Hamilton Path are **NP**-complete

Claim 1:

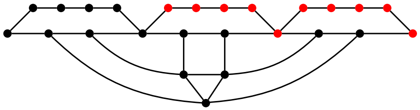
- In $G(c)$ there is no Hamilton path from c_0 to c_1.
- If one removes from $G(c)$ at least one of the paths $c_{\ell 1} - c_{\ell 2} - c_{\ell 3} - c_{\ell 4}$, $\ell \in c$, then there is a Hamilton path from c_0 to c_1.



Hamilton Circuit and Hamilton Path are **NP**-complete

Claim 1:

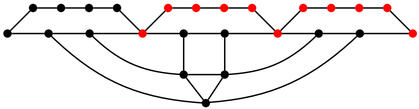
- In $G(c)$ there is no Hamilton path from c_0 to c_1.
- If one removes from $G(c)$ at least one of the paths $c_{\ell 1} - c_{\ell 2} - c_{\ell 3} - c_{\ell 4}$, $\ell \in c$, then there is a Hamilton path from c_0 to c_1.



Hamilton Circuit and Hamilton Path are **NP**-complete

Claim 1:

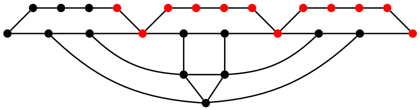
- In $G(c)$ there is no Hamilton path from c_0 to c_1.
- If one removes from $G(c)$ at least one of the paths $c_{\ell 1} - c_{\ell 2} - c_{\ell 3} - c_{\ell 4}$, $\ell \in c$, then there is a Hamilton path from c_0 to c_1.



Hamilton Circuit and Hamilton Path are **NP**-complete

Claim 1:

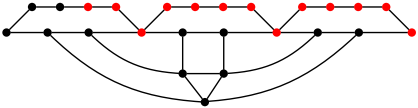
- In $G(c)$ there is no Hamilton path from c_0 to c_1.
- If one removes from $G(c)$ at least one of the paths $c_{\ell 1} - c_{\ell 2} - c_{\ell 3} - c_{\ell 4}$, $\ell \in c$, then there is a Hamilton path from c_0 to c_1.



Hamilton Circuit and Hamilton Path are **NP**-complete

Claim 1:

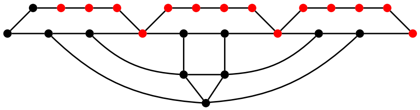
- In $G(c)$ there is no Hamilton path from c_0 to c_1.
- If one removes from $G(c)$ at least one of the paths $c_{\ell 1} - c_{\ell 2} - c_{\ell 3} - c_{\ell 4}$, $\ell \in c$, then there is a Hamilton path from c_0 to c_1.



Hamilton Circuit and Hamilton Path are **NP**-complete

Claim 1:

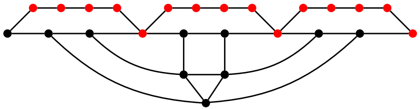
- In $G(c)$ there is no Hamilton path from c_0 to c_1.
- If one removes from $G(c)$ at least one of the paths $c_{\ell 1} - c_{\ell 2} - c_{\ell 3} - c_{\ell 4}$, $\ell \in c$, then there is a Hamilton path from c_0 to c_1.



Hamilton Circuit and Hamilton Path are **NP**-complete

Claim 1:

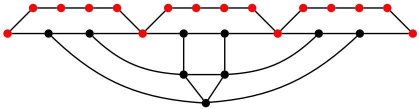
- In $G(c)$ there is no Hamilton path from c_0 to c_1.
- If one removes from $G(c)$ at least one of the paths $c_{\ell 1} - c_{\ell 2} - c_{\ell 3} - c_{\ell 4}$, $\ell \in c$, then there is a Hamilton path from c_0 to c_1.



Hamilton Circuit and Hamilton Path are **NP**-complete

Claim 1:

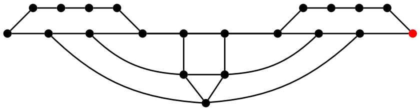
- In $G(c)$ there is no Hamilton path from c_0 to c_1.
- If one removes from $G(c)$ at least one of the paths $c_{\ell 1} - c_{\ell 2} - c_{\ell 3} - c_{\ell 4}$, $\ell \in c$, then there is a Hamilton path from c_0 to c_1.



Hamilton Circuit and Hamilton Path are **NP**-complete

Claim 1:

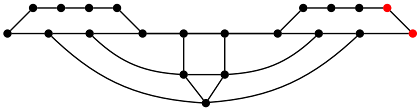
- In $G(c)$ there is no Hamilton path from c_0 to c_1.
- If one removes from $G(c)$ at least one of the paths $c_{\ell 1} - c_{\ell 2} - c_{\ell 3} - c_{\ell 4}$, $\ell \in c$, then there is a Hamilton path from c_0 to c_1.



Hamilton Circuit and Hamilton Path are **NP**-complete

Claim 1:

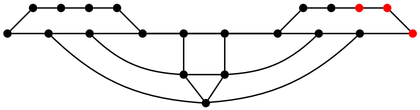
- In $G(c)$ there is no Hamilton path from c_0 to c_1.
- If one removes from $G(c)$ at least one of the paths $c_{\ell 1} - c_{\ell 2} - c_{\ell 3} - c_{\ell 4}$, $\ell \in c$, then there is a Hamilton path from c_0 to c_1.



Hamilton Circuit and Hamilton Path are **NP**-complete

Claim 1:

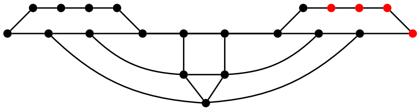
- In $G(c)$ there is no Hamilton path from c_0 to c_1.
- If one removes from $G(c)$ at least one of the paths $c_{\ell 1} - c_{\ell 2} - c_{\ell 3} - c_{\ell 4}$, $\ell \in c$, then there is a Hamilton path from c_0 to c_1.



Hamilton Circuit and Hamilton Path are **NP**-complete

Claim 1:

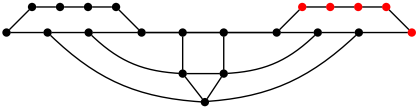
- In $G(c)$ there is no Hamilton path from c_0 to c_1.
- If one removes from $G(c)$ at least one of the paths $c_{\ell 1} - c_{\ell 2} - c_{\ell 3} - c_{\ell 4}$, $\ell \in c$, then there is a Hamilton path from c_0 to c_1.



Hamilton Circuit and Hamilton Path are **NP**-complete

Claim 1:

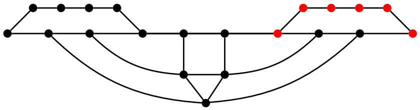
- In $G(c)$ there is no Hamilton path from c_0 to c_1.
- If one removes from $G(c)$ at least one of the paths $c_{\ell 1} - c_{\ell 2} - c_{\ell 3} - c_{\ell 4}$, $\ell \in c$, then there is a Hamilton path from c_0 to c_1.



Hamilton Circuit and Hamilton Path are **NP**-complete

Claim 1:

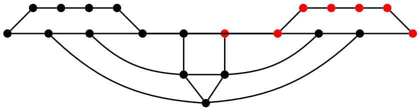
- In $G(c)$ there is no Hamilton path from c_0 to c_1.
- If one removes from $G(c)$ at least one of the paths $c_{\ell 1} - c_{\ell 2} - c_{\ell 3} - c_{\ell 4}$, $\ell \in c$, then there is a Hamilton path from c_0 to c_1.



Hamilton Circuit and Hamilton Path are **NP**-complete

Claim 1:

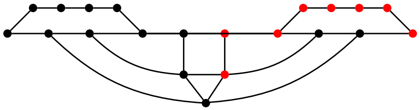
- In $G(c)$ there is no Hamilton path from c_0 to c_1.
- If one removes from $G(c)$ at least one of the paths $c_{\ell 1} - c_{\ell 2} - c_{\ell 3} - c_{\ell 4}$, $\ell \in c$, then there is a Hamilton path from c_0 to c_1.



Hamilton Circuit and Hamilton Path are **NP**-complete

Claim 1:

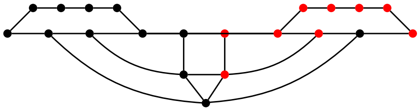
- In $G(c)$ there is no Hamilton path from c_0 to c_1.
- If one removes from $G(c)$ at least one of the paths $c_{\ell 1} - c_{\ell 2} - c_{\ell 3} - c_{\ell 4}$, $\ell \in c$, then there is a Hamilton path from c_0 to c_1.



Hamilton Circuit and Hamilton Path are **NP**-complete

Claim 1:

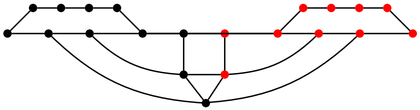
- In $G(c)$ there is no Hamilton path from c_0 to c_1.
- If one removes from $G(c)$ at least one of the paths $c_{\ell 1} - c_{\ell 2} - c_{\ell 3} - c_{\ell 4}$, $\ell \in c$, then there is a Hamilton path from c_0 to c_1.



Hamilton Circuit and Hamilton Path are **NP**-complete

Claim 1:

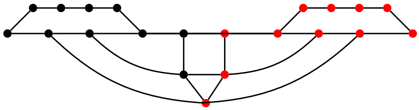
- In $G(c)$ there is no Hamilton path from c_0 to c_1.
- If one removes from $G(c)$ at least one of the paths $c_{\ell 1} - c_{\ell 2} - c_{\ell 3} - c_{\ell 4}$, $\ell \in c$, then there is a Hamilton path from c_0 to c_1.



Hamilton Circuit and Hamilton Path are **NP**-complete

Claim 1:

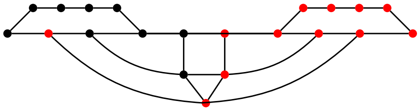
- In $G(c)$ there is no Hamilton path from c_0 to c_1.
- If one removes from $G(c)$ at least one of the paths $c_{\ell 1} - c_{\ell 2} - c_{\ell 3} - c_{\ell 4}$, $\ell \in c$, then there is a Hamilton path from c_0 to c_1.



Hamilton Circuit and Hamilton Path are **NP**-complete

Claim 1:

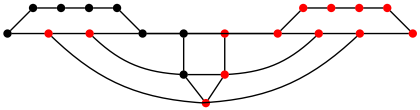
- In $G(c)$ there is no Hamilton path from c_0 to c_1.
- If one removes from $G(c)$ at least one of the paths $c_{\ell 1} - c_{\ell 2} - c_{\ell 3} - c_{\ell 4}$, $\ell \in c$, then there is a Hamilton path from c_0 to c_1.



Hamilton Circuit and Hamilton Path are **NP**-complete

Claim 1:

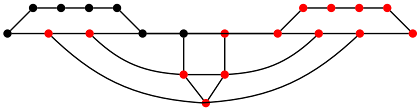
- In $G(c)$ there is no Hamilton path from c_0 to c_1.
- If one removes from $G(c)$ at least one of the paths $c_{\ell 1} - c_{\ell 2} - c_{\ell 3} - c_{\ell 4}$, $\ell \in c$, then there is a Hamilton path from c_0 to c_1.



Hamilton Circuit and Hamilton Path are **NP**-complete

Claim 1:

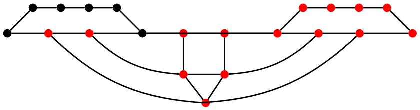
- In $G(c)$ there is no Hamilton path from c_0 to c_1.
- If one removes from $G(c)$ at least one of the paths $c_{\ell 1} - c_{\ell 2} - c_{\ell 3} - c_{\ell 4}$, $\ell \in c$, then there is a Hamilton path from c_0 to c_1.



Hamilton Circuit and Hamilton Path are **NP**-complete

Claim 1:

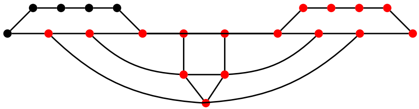
- In $G(c)$ there is no Hamilton path from c_0 to c_1.
- If one removes from $G(c)$ at least one of the paths $c_{\ell 1} - c_{\ell 2} - c_{\ell 3} - c_{\ell 4}$, $\ell \in c$, then there is a Hamilton path from c_0 to c_1.



Hamilton Circuit and Hamilton Path are **NP**-complete

Claim 1:

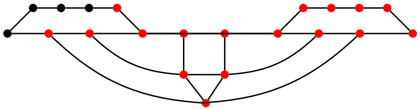
- In $G(c)$ there is no Hamilton path from c_0 to c_1.
- If one removes from $G(c)$ at least one of the paths $c_{\ell 1} - c_{\ell 2} - c_{\ell 3} - c_{\ell 4}$, $\ell \in c$, then there is a Hamilton path from c_0 to c_1.



Hamilton Circuit and Hamilton Path are **NP**-complete

Claim 1:

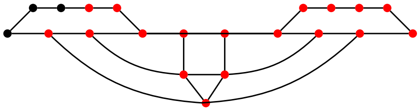
- In $G(c)$ there is no Hamilton path from c_0 to c_1.
- If one removes from $G(c)$ at least one of the paths $c_{\ell 1} - c_{\ell 2} - c_{\ell 3} - c_{\ell 4}$, $\ell \in c$, then there is a Hamilton path from c_0 to c_1.



Hamilton Circuit and Hamilton Path are **NP**-complete

Claim 1:

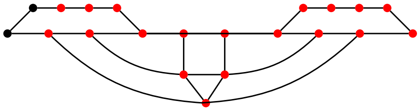
- In $G(c)$ there is no Hamilton path from c_0 to c_1.
- If one removes from $G(c)$ at least one of the paths $c_{\ell 1} - c_{\ell 2} - c_{\ell 3} - c_{\ell 4}$, $\ell \in c$, then there is a Hamilton path from c_0 to c_1.



Hamilton Circuit and Hamilton Path are **NP**-complete

Claim 1:

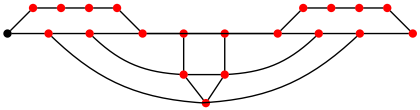
- In $G(c)$ there is no Hamilton path from c_0 to c_1.
- If one removes from $G(c)$ at least one of the paths $c_{\ell 1} - c_{\ell 2} - c_{\ell 3} - c_{\ell 4}$, $\ell \in c$, then there is a Hamilton path from c_0 to c_1.



Hamilton Circuit and Hamilton Path are **NP**-complete

Claim 1:

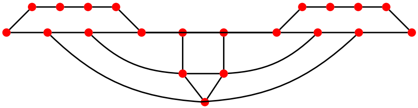
- In $G(c)$ there is no Hamilton path from c_0 to c_1.
- If one removes from $G(c)$ at least one of the paths $c_{\ell 1} - c_{\ell 2} - c_{\ell 3} - c_{\ell 4}$, $\ell \in c$, then there is a Hamilton path from c_0 to c_1.



Hamilton Circuit and Hamilton Path are **NP**-complete

Claim 1:

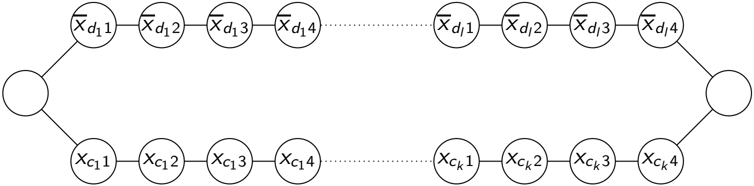
- In $G(c)$ there is no Hamilton path from c_0 to c_1.
- If one removes from $G(c)$ at least one of the paths $c_{\ell 1} - c_{\ell 2} - c_{\ell 3} - c_{\ell 4}$, $\ell \in c$, then there is a Hamilton path from c_0 to c_1.



Hamilton Circuit and Hamilton Path are **NP**-complete

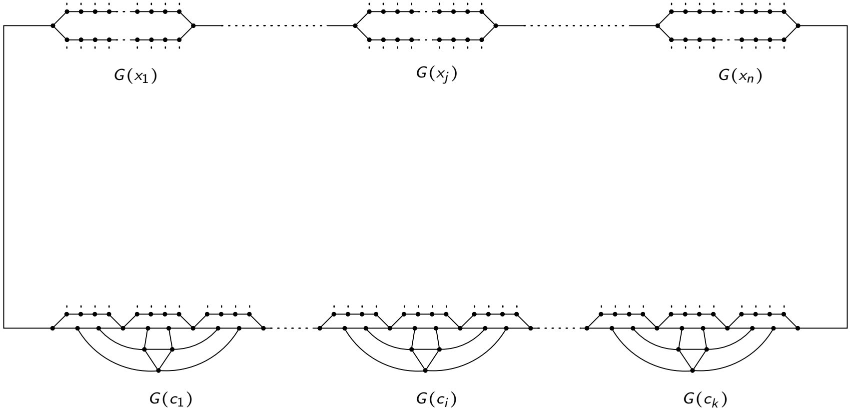
For a variable x let $\{c_1, \ldots, c_k\}$ be the set of clauses with $x \in c_i$ and let $\{d_1, \ldots, d_l\}$ be the set of clauses with $\overline{x} \in d_j$.

For every x we define the graph $G(x)$:



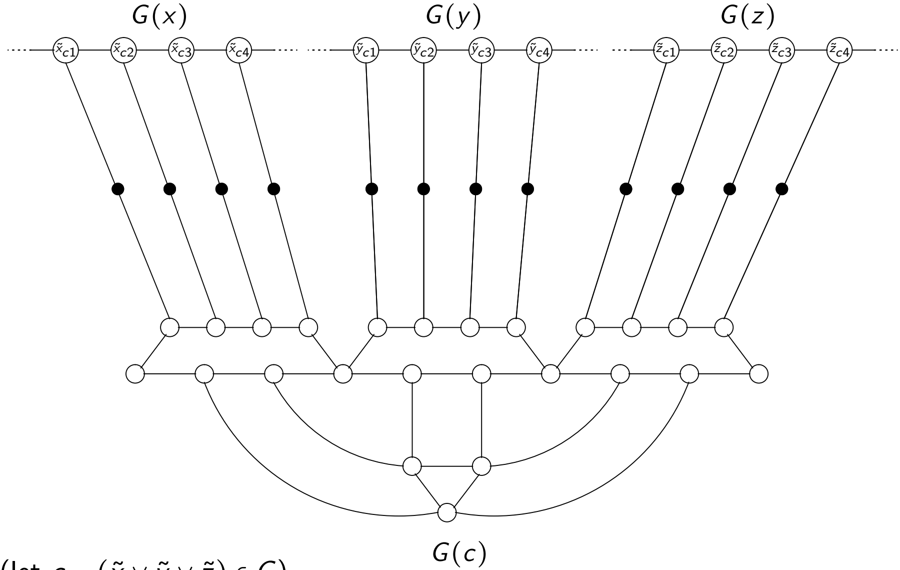
Hamilton Circuit and Hamilton Path are **NP**-complete

The graph $G(F)$ is assembled from the graphs $G(c_i)$ and $G(x_j)$, where $C = \{c_1, \ldots, c_k\}$ and $x_1, \ldots, x_n$ are the variables in F.



For every clause c, every literal $\tilde{x} \in c$, and all $1 \le i \le 4$ we connect $c_{\tilde{x},i}$ (a vertex from $G(c)$) and $\tilde{x}_{c,i}$ via an extra node.

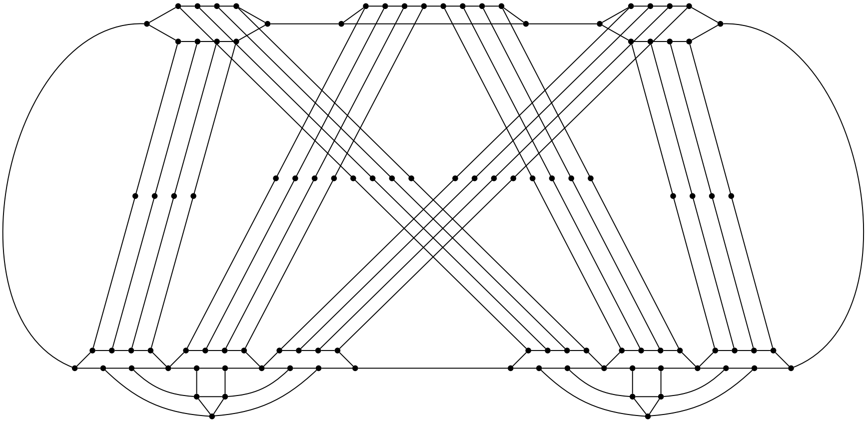
Hamilton Circuit and Hamilton Path are **NP**-complete



(let $c = (\tilde{x} \vee \tilde{y} \vee \tilde{z}) \in C$)

Hamilton Circuit and Hamilton Path are **NP**-complete

Example: The graph $G(F)$ for $F = \overbrace{(x_1 \vee \overline{x_2} \vee \overline{x_3})}^{c_1} \wedge \overbrace{(\overline{x_1} \vee \overline{x_2} \vee x_3)}^{c_2}$.



The Hamilton circuit that corresponds to $x_1 = 1$, $x_2 = 0$, $x_3 = 1$ can be found at `https://www.eti.uni-siegen.de/ti/lehre/ws2021/komplexitaetstheorie/example-hamilton.pdf`.

Hamilton Circuit and Hamilton Path are **NP**-complete

Claim 2: $F \in \mathsf{SAT} \iff G(F)$ has a Hamilton circuit.

$\Longrightarrow$: Assume σ is a truth assignment that makes F true: $\sigma(F) = 1$.

We obtain a Hamilton circuit for $G(F)$ as follows:

The circuit leads for every variable x via the x-branch (resp., the $\overline{x}$-branch), if $\sigma(x) = 1$ (resp., $\sigma(x) = 0$). Thereby it visits via the extra nodes in every graph $G(c)$ at least one of the paths $c_{\tilde{x}1} - c_{\tilde{x}2} - c_{\tilde{x}3} - c_{\tilde{x}4}$, where $\tilde{x} \in c$ is a literal with $\sigma(\tilde{x}) = 1$.

This is possible, since σ sets in each clause in each clause at least one literal to 1.

When all graphs $G(x)$ are traversed, the Hamilton circuit visits those vertices from the subgraphs $G(c)$ and $G(x)$ that have not been visited so far. This is possible by Claim 1.

The Hamilton circuit finally ends at the initial vertex.

Hamilton Circuit and Hamilton Path are **NP**-complete

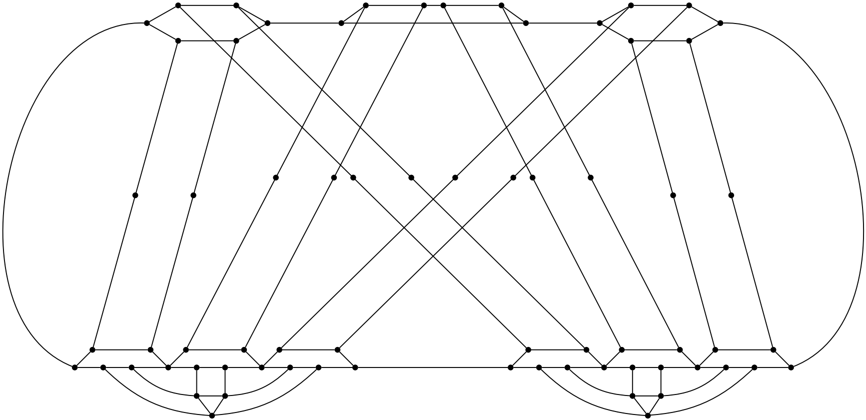
$\Longleftarrow$: Let C be a Hamilton circuit for $G(F)$.

C must traverse for each graph $G(x)$ either the x-branch or the $\overline{x}$-branch.

This defines a truth assignment for the variables in F and its not hard to see that this assignment makes F true. $\square$

Hamilton Circuit and Hamilton Path are **NP**-complete

Excercise: Would the following construction also work?



Complete problems for **P**

Let $L_{cfe} = \{\langle G \rangle | \ G$ is a context-free grammar with $L(G) \neq \emptyset\}$.
Here, $\langle G \rangle$ stands for a suitable encoding of the grammar G, *cfe* stands for "context–free–empty".

Theorem 31
*L_{cfe} is **P**-complete.*

Proof:

(A) $L_{cfe} \in \mathbf{P}$

Check for a given context-free grammar G, whether the start non-terminal S is productive.

Let P be the set of productions of G, Σ be the set of terminal symbols and N be the set of non-terminals.

A non-terminal A is productive, if there is a word $w \in \Sigma^*$ with $A \Rightarrow_G^* w$.

Complete problems for **P**

The following algorithm computes the set of productive non-terminals in polynomial time:

$M := \{A \in N \mid \text{ there is a } (A \to w) \in P \text{ with } w \in \Sigma^*\};$
$M' = \emptyset;$
while $M \neq M'$ **do**
    $M' := M;$
    $M := M' \cup \{A \in N \mid \text{ there is a } (A \to w) \in P \text{ with } w \in (M' \cup \Sigma)^*\};$
endwhile

(B) L_{cfe} is **P**-hard.

Let $L \in \mathbf{P}$ and $L(M) = L$ for a $p(n)$-time bounded deterministic Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, q_J, q_N, \Box)$, $p(n) > n$ a polynomial.

Let $w = w_1 \cdots w_n \in \Sigma^*$ be an input for M with $|w| = n \geq 1$.

Emptiness for context-free grammars is **P**-complete

We make for M similar assumptions as in the proof of Cook's theorem, where $\Omega = (Q \times \Gamma) \cup \Gamma$ (see slide 88 and 90):

1. configurations of M are represented by words from the language $\mathrm{Conf} = \{\Box u(q,a)v\Box \mid (q,a) \in Q \times \Gamma,\ uv \in \Gamma^{2p(n)}\}$.
2. The start configuration is $\alpha_0 := \Box^{p(n)+1}(q_0, w_1)w_2 \cdots w_n \Box^{p(n)-n+2}$.
3. $w \in L(M)$ if and only if M reaches the accepting state q_J after at most $p(n)$ steps from α_0.

Since M is deterministic, the relation $\Delta \subseteq \Omega^4$ from the proof of Cook's theorem (slide 92) becomes a function $\Delta : \Omega^3 \to \Omega$ such that for all words $\alpha, \alpha' \in \Box\Omega^*\Box$ with $|\alpha| = |\alpha'|$ we have:

$$\alpha, \alpha' \in \mathrm{Conf} \text{ and } \alpha \vdash_M \alpha'$$
$$\iff$$
$$\alpha \in \mathrm{Conf} \text{ and } \forall i \in \{-p(n), \ldots, p(n)\} : \Delta(\alpha[i-1], \alpha[i], \alpha[i+1]) = \alpha'[i].$$

Emptiness for context-free grammars is **P**-complete

We define the grammar $G(w) = (V, \varnothing, P, S)$ with set of variables

$$V = \{S\} \cup \{V(a,t,j) \mid a \in \Omega,\ 0 \le t \le p(n),\ |j| \le p(n)+1\},$$

an empty terminal alphabet, the start non-terminal S and the following set of productions (λ = empty word):

- $S \to V((q_J, a), t, j)$ for $0 \le t \le p(n), |j| \le p(n)+1, a \in \Gamma$
- $V(a, t+1, j) \to V(b, t, j-1) V(c, t, j) V(d, t, j+1)$
  if $\Delta(b, c, d) = a, 0 \le t \le p(n) - 1, |j| \le p(n)$
- $V(\Box, t, j) \to \lambda$ for $0 \le t \le p(n), |j| = p(n)+1$,
- $V((q_0, w_1), 0, 0) \to \lambda$,
- $V(w_{j+1}, 0, j) \to \lambda$ for $1 \le j \le n-1$,
- $V(\Box, 0, j) \to \lambda$ for $j \in \{-p(n), \dots, -1\} \cup \{n, \dots, p(n)\}$

Emptiness for context-free grammars is **P**-complete

Claim: $L(G(w)) \neq \emptyset \iff w \in L$.

Let $\alpha_0 \vdash_M \alpha_1 \vdash_M \cdots \vdash_M \alpha_{p(n)}$ ($\alpha_i \in \mathrm{Conf}$) be the unique computation that begins with the start configuration α_0.

For $-p(n) - 1 \leq j \leq p(n) + 1$ and $0 \leq t \leq p(n)$ let $\alpha(t, j) = \alpha_t[j]$.

We show the above claim by proving

$$L(V(a, t, j)) \neq \emptyset \iff \alpha(t, j) = a,$$

where $L(V(a, t, j)) \subseteq \{\lambda\}$ ist the set of all terminal words that can be derived from $V(a, t, j)$:

$\Longleftarrow$: Let $\alpha(t, j) = a$.

The cases $t = 0$ and $j \in \{-p(n) - 1, p(n) + 1\}$ follow immediately from the definition of $G(w)$.

Emptiness for context-free grammars is **P**-complete

If $t \geq 1$ and $-p(n) \leq j \leq p(n)$, then there are $b, c, d \in \Omega$ with $\Delta(b, c, d) = a$ and

- $\alpha(t-1, j-1) = b$,
- $\alpha(t-1, j) = c$,
- $\alpha(t-1, j+1) = d$.

Induction over t yields

- $L(V(b, t-1, j-1)) \neq \emptyset$,
- $L(V(c, t-1, j)) \neq \emptyset$,
- $L(V(d, t-1, j+1)) \neq \emptyset$.

Since $G(w)$ contains the production

$$V(a, t, j) \to V(b, t-1, j-1)V(c, t-1, j)V(d, t-1, j+1),$$

we get $L(V(a, t, j)) \neq \emptyset$.

Emptiness for context-free grammars is **P**-complete

$\Longrightarrow$: Let $L(V(a,t,j)) \neq \emptyset$.

The cases $t = 0$ and $j \in \{-p(n)-1, p(n)+1\}$ follow from the definition of $G(w)$.

If $t \geq 1$ and $-p(n) \leq j \leq p(n)$, then there must exist a production

$$V(a,t,j) \to V(b,t-1,j-1)V(c,t-1,j)V(d,t-1,j+1)$$

(in particular $\Delta(b,c,d) = a$) such that

- $L(V(b,t-1,j-1)) \neq \emptyset$,
- $L(V(c,t-1,j)) \neq \emptyset$,
- $L(V(d,t-1,j+1)) \neq \emptyset$.

Induction $\Rightarrow$ $\alpha(t-1,j-1) = b$, $\alpha(t-1,j) = c$, $\alpha(t-1,j+1) = d$.

Since $\Delta(b,c,d) = a$, we get $\alpha(t,j) = a$. □

Boolean circuits

Definition of a boolean circuit

A boolean circuit C is a directed labelled graph $C = (\{1, \ldots, o\}, E, s)$ for some $o \in \mathbb{N}$ with the following properties:

- $\forall (i,j) \in E : i < j$, i,e., C is acyclic.
- $s : \{1, \ldots, o\} \to \{\neg, \wedge, \vee, 0, 1\}$, where

$$
\begin{aligned}
s(i) \in \{\wedge, \vee\} &\quad\Rightarrow\quad \text{indegree}(i) = 2 \\
s(i) = \neg &\quad\Rightarrow\quad \text{indegree}(i) = 1 \\
s(i) \in \{0, 1\} &\quad\Rightarrow\quad \text{indegree}(i) = 0
\end{aligned}
$$

  $s(i)$ is the type (or sort) of vertex i.

The vertices are also called gates.

The gate o is the output gate of C.

Boolean circuits

We can evaluate the circuit C in the intuitive way (see example) and thereby assign to every gate i a truth value $v(i) \in \{0, 1\}$.

A circuit is called monotone, if it does not contain $\neg$-gates.

Circuit Value (CV) is the following problem:

input: A boolean circuit C

question: Does the output gate of C evaluate to 1?

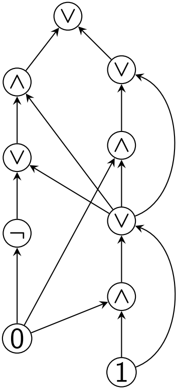
Monotone Circuit Value (MCV) is the following problem:

input: A monotone boolean circuit C

question: Does the output gate of C evaluate to 1?

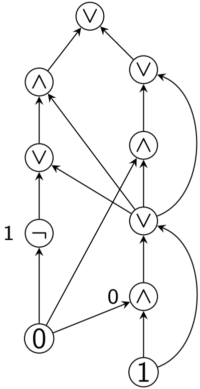
Boolean circuits

Example:



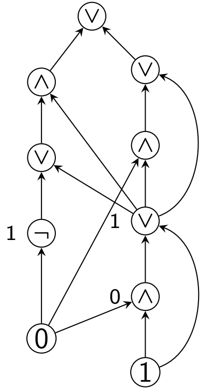
Boolean circuits

Example:



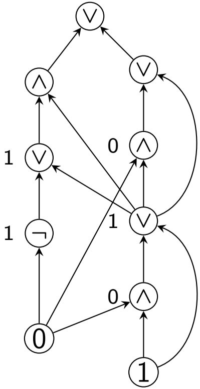
Boolean circuits

Example:



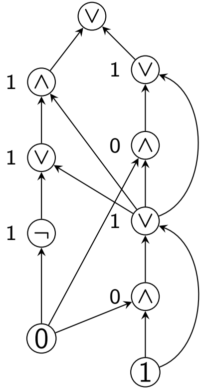
Boolean circuits

Example:



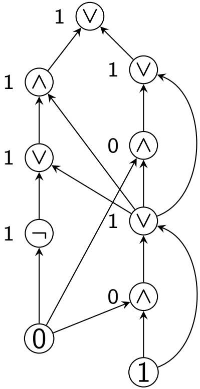
Boolean circuits

Example:



Boolean circuits

Example:



Circuit Value is **P**-complete

Theorem 32

CV *and* MCV *are* **P**-*complete.*

Proof:

(i) CV $\in$ **P**: evaluate the gate in the order $1, 2, \ldots, o$.

(ii) MCV is **P**-hard:

Recall the proof of the **P**-hardness of L_{cfe}.

For a language $L \in$ **P** and every input $w \in \Sigma^*$ we constructed a context-free grammar $G(w)$ with: $w \in L$ if and only if $\lambda \in L(G(w))$.

All productions of $G(w)$ have the form $A \to \alpha$, where α is a (possibly empty) sequence of non-terminals.

Moreover, $G(w)$ is acyclic (there are no derivations of the form $A \Rightarrow^+ uAv$).

Circuit Value is **P**-complete

Step 1: Replace every production $A \to A_1 A_2 \cdots A_n$ with $n \geq 3$ by

$$A \to A_1 A_2',\ A_i' \to A_i A_{i+1}'\ (2 \leq i \leq n-2),\ A_{n-1}' \to A_{n-1} A_n$$

for new non-terminals $A_2', \ldots, A_{n-1}'$.

Step 2: Replace every production $A \to B$ by $A \to BB$.

Now, all productions are of type $A \to \lambda$ or $A \to BC$.

Step 3: for every non-terminal A, which is the left-hand side of at least two productions, i.e., $A \to \alpha_1 | \alpha_2 | \cdots | \alpha_n$ for some $n \geq 2$, we replace these n productions by

$$A \to A_1 | A_2,\ A_1 \to \alpha_1,\ A_2 \to \alpha_2$$

if $n = 2$ (A_1, A_2 are new non-terminals), respectively

$$A \to A_1 | A_2',\ A_i' \to A_i | A_{i+1}'\ (2 \leq i \leq n-2),$$
$$A_{n-1}' \to A_{n-1} | A_n,\ A_i \to \alpha_i\ (1 \leq i \leq n).$$

if $n \geq 3$ ($A_1, \ldots, A_n, A_2', \ldots, A_{n-1}'$ are new non-terminals).

Circuit Value is **P**-complete

Then, for every non-terminal A one of the following 4 cases holds:

1. There is no production for A.
2. For A there is exactly one production with A on the left-hand side. This production is $A \to \lambda$.
3. For A there is exactly one production with A on the left-hand side. This production is of type $A \to BC$.
4. A is the left-hand side for exactly two productions and these productions are of type $A \to B$: $A \to B|C$

The new grammar produces λ if and only if the old grammar produces λ.

We denote this new grammar again with $G(w)$.

Circuit Value is **P**-complete

We define the circuit $C(w)$ as follows:

Every non-terminal of $G(w)$ is a gate of $C(w)$.

The start non-terminal of $G(w)$ is the output gate of $C(w)$.

1. A non-terminal A of type 1 becomes a 0-input gate.
2. A non-terminal A of type 2 becomes a 1-input gate
3. A non-terminal A of type 3 becomes a $\wedge$-gate with entries B and C.
4. A non-terminal A of type 4 becomes a $\vee$-gate with entries B and C.

Circuit Value is **P**-complete

The circuit $C(v)$ produced in this way is acyclic because $G(w)$ is acyclic.

We have: $L(A) \neq \emptyset \iff$ gate A evaluates in $C(w)$ to 1.

Hence, $L(G(w)) \neq \emptyset \iff$ the output gate of $C(w)$ evaluates to 1. $\square$

Remark: In a boolean circuit, a gate may have outdegree > 1. This seems to be important for the **P**-hardness:

The set of all (variable-free) boolean expressions is defined by the following grammar:

$$A ::= 0 \mid 1 \mid (\neg A) \mid (A \wedge A') \mid (A \vee A')$$

Boolean expressions can be viewed as tree-shaped boolean circuits.

Buss 1987: The set of all boolean expressions that evaluate to the truth value 1 is complete for the complexity class $\mathbf{NC}^1 \subseteq \mathbf{L}$.

Complete problems for **PSPACE**: quantified boolean formulas

Quantified boolean formulas

The set M of quantified boolean formulas is the smallest set with:

- $x_i \in M$ for all $i \geq 1$
- $0, 1 \in M$
- $E, F \in M, i \geq 1 \implies (\neg E), (E \wedge F), (E \vee F), \forall x_i E, \exists x_i E \in M$

Alternatively: M can be defined by a context-free grammar with terminal alphabet $\Sigma = \{x, 0, 1, (,), \neg, \wedge, \vee, \forall, \exists\}$.

Variables can be encoded by words from $x1\{0,1\}^*$.

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

Satisfiability of boolean formulas

Satisfiability of quantified boolean formulas is defined by the existence of a satisfying assignment.

An assignment is a function $b : \{x_1, x_2, \ldots\} \to \{0, 1\}$.

For a given formula F, the assignment can be restricted to those variables that occur in F.

For $z \in \{0, 1\}$ and an assignment b let $b[x_j \mapsto z]$ be the assignment with

- $b[x_j \mapsto z](x_i) = b(x_i)$ for $i \neq j$ and
- $b[x_j \mapsto z](x_j) = z$.

Satisfiability of boolean formulas

Inductive definition of the satisfiability of the formula F with respect to the assignment b:

The assignment b satisfies the formula F if and only if one of the following conditions holds:

$F = 1$,		
$F = x_j$	and	$b(x_j) = 1$,
$F = (\neg E)$	and	b does not satisfy E,
$F = (F_1 \wedge F_2)$	and	b satisfies F_1 and F_2,
$F = (F_1 \vee F_2)$	and	b satisfies F_1 or F_2,
$F = \exists x_j E$	and	$b[x_j \mapsto 0]$ or $b[x_j \mapsto 1]$ satisfies E,
$F = \forall x_j E$	and	$b[x_j \mapsto 0]$ and $b[x_j \mapsto 1]$ satisfy E.

If F is satisfied by every assignment then F is called valid.

Satisfiability of boolean formulas

The set $\mathrm{Free}(F)$ of free variables of F is defined as follows:

- $\mathrm{Free}(0) = \mathrm{Free}(1) = \emptyset$
- $\mathrm{Free}(x_i) = \{x_i\}$
- $\mathrm{Free}(\neg F) = \mathrm{Free}(F)$
- $\mathrm{Free}((F \wedge G)) = \mathrm{Free}((F \vee G)) = \mathrm{Free}(F) \cup \mathrm{Free}(G)$
- $\mathrm{Free}(\exists x_j F) = \mathrm{Free}(\forall x_j F) = \mathrm{Free}(F) \setminus \{x_j\}$

A formula F with $\mathrm{Free}(F) = \emptyset$ is called closed.

Note: The satisfiability of a closed formula F does not depend on the assignment. In other words: if there is a satisfying assignment for F then F is already valid.

QBF is the set of all closed quantified boolean formulas that are valid.

QBF is **PSPACE**-complete

Theorem 33

QBF *is* **PSPACE**-*complete.*

Proof:

(i) QBF $\in$ **PSPACE**:

Let E be a closed quantified boolean formula in which the variables $x_1, \ldots, x_n$ occur.

W.l.o.g. E is build from $1, x_1, \ldots, x_n, \neg, \wedge, \exists$ and there is no variable x_i that is quantified twice in E (the algorithm on the next slide would for instance not yield a correct result for the formula $\exists x((\exists x\, 0) \vee x))$.

The following recursive deterministic algorithm algorithm uses $x_1, \ldots, x_n$ as global variables and checks in polynomial space whether E is valid.

QBF is **PSPACE**-complete

```
FUNCTION check(F)
  if F = 1 then return(1)
  elseif F = x_i then return(x_i)
  elseif F = (¬G) then return(not check(G))
  elseif F = (F_1 ∧ F_2) then return(check(F_1) and  check(F_2))
  elseif F = ∃x_i G then
    x_i := 1
    if check(G) = 1 then
      return(1)
    else
      x_i := 0
      return(check(G))
    endif
  endif
ENDFUNC
```

QBF is **PSPACE**-complete

(ii) QBF is **PSPACE**-hard:

Let $L \in$ **PSPACE** and $L(M) = L$ for a $p(n)$-space bounded deterministic Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, q_J, q_N, \Box)$, $p(n) > n$ is a polynomial.

Let $w = w_1 \cdots w_n \in \Sigma^*$ be an input for M with $|w| = n \geq 1$.

We assume for M conventions similar to those from the proof of Cook's theorem, where $\Omega = (Q \times \Gamma) \cup \Gamma$:

1. configurations of M will be described by words from the language $\text{Conf} = \{\Box u(q,a)v\Box \mid (q,a) \in Q \times \Gamma,\ uv \in \Gamma^{2p(n)}\}$.
2. $\mathrm{Start}(w) = \Box^{p(n)+1}(q_0, w_1) w_2 \cdots w_n \Box^{p(n)-n+2}$.
3. $\alpha_f = \Box^{p(n)+1}(q_J, \Box)\Box^{p(n)+1}$ is w.l.o.g. the unique accepting configuration, that is possibly reachable from $\mathrm{Start}(w)$.

QBF is **PSPACE**-complete

There exists a function $\Delta : \Omega^3 \to \Omega$ such that for all $\alpha, \alpha' \in \Box\Omega^*\Box$ with $|\alpha| = |\alpha'|$ we have:

$$\alpha, \alpha' \in \text{Conf and } \alpha \vdash_M \alpha'$$
$$\iff$$
$$\alpha \in \text{Conf and } \forall i \in \{-p(n), \dots, p(n)\} : \Delta(\alpha[i-1], \alpha[i], \alpha[i+1]) = \alpha'[i].$$

Moreover, there is a constant c such that at most $2^{c \cdot p(n)}$ configurations are reachable from $\mathrm{Start}(w)$ (Lemma 2).

Consider the approach from the proof of Savitch's theorem:

$$\mathrm{Reach}(\mathrm{Start}(w), \alpha_f, c \cdot p(n)) \iff w \in L$$

$$\mathrm{Reach}(\alpha, \beta, i) = \exists \gamma \left(\mathrm{Reach}(\alpha, \gamma, i-1) \wedge \mathrm{Reach}(\gamma, \beta, i-1) \right) \quad \text{for } i > 0$$

$$\mathrm{Reach}(\alpha, \beta, 0) = \alpha \vdash_M^{\leq 1} \beta$$

QBF is **PSPACE**-complete

An iterated application of this would lead to a formula of exponential length.

Solution: We introduce configuration variables $X, Y, U, V, \dots$ that take values from Conf and define for $i > 0$:

$$\mathrm{Reach}(X, Y, i) :=$$
$$\exists U\, \forall V\, \forall W \left(\begin{array}{l} ((V = X \wedge W = U) \vee (V = U \wedge W = Y)) \\ \to \mathrm{Reach}(V, W, i-1) \end{array} \right)$$

Step 1: Compute from the input w by iterated application of the above recursion, starting with $\mathrm{Reach}(\mathrm{Start}(w), \alpha_f, c \cdot p(n))$, a formula F of size $\mathcal{O}(c \cdot p(n))$ in which configuration variables $X, Y, \dots$ occur.

F contains atomic formulas of the form $\mathrm{Reach}(X, Y, 0)$ and $X = Y$ as well as the constants $\mathrm{Start}(w)$ and α_f.

QBF is **PSPACE**-complete

Step 2: We transform F into a closed quantified boolean formula:

- We encode a configuration X by an assignment for boolean variables $x_{a,i}$ for $a \in \Omega$ and $|i| \leq p(n) + 1$.

  Intuition: $x_{a,i} = 1$ if and only if in the configuration X the symbol a is at position i.

- There is a boolean formula $\gamma((x_{a,i})_{a \in \Omega, |i| \leq p(n)+1})$ of size $\mathcal{O}(p(n))$ that is satisfied for an assingment for the variables $x_{a,i}$ if and only if the assignment describes a correct configuration.

- The constants $\mathrm{Start}(w)$ and α_f can be replaced by concrete truth values for the corresponding boolean variables.

QBF is **PSPACE**-complete

- $\forall X \cdots$, respectively $\exists X \cdots$, is replaced by the following block of quantifiers:

  $$\forall x_{a,i}\ (a \in \Omega, |i| \leq p(n)+1) : \gamma((x_{a,i})_{a \in \Omega, |i| \leq p(n)+1}) \to \cdots \text{ resp.}$$
  $$\exists x_{a,i}\ (a \in \Omega, |i| \leq p(n)+1) : \gamma((x_{a,i})_{a \in \Omega, |i| \leq p(n)+1}) \wedge \cdots$$

- $X = Y$ is replaced by the formula $\bigwedge_{a \in \Omega, |i| \leq p(n)+1} (x_{a,i} \leftrightarrow y_{a,i})$.
- The atomic formula $\mathrm{Reach}(X, Y, 0)$ becomes $X = Y \vee X \vdash_M Y$, where $X \vdash_M Y$ is finally replaced by

  $$\bigwedge_{|i| \leq p(n)} \ \bigvee_{(a,b,c) \in \Delta} (x_{a,i-1} \ \wedge \ x_{b,i} \ \wedge \ x_{c,i+1} \ \wedge \ y_{\Delta(a,b,c),i})$$

In this way, we obtain a closed quantified boolean formula that is valid if and only if $w \in L$. □

Equivalence of regular expressions is **PSPACE**-complete

Recall: for a finite alphabet Σ, $\mathrm{Reg}(\Sigma)$ denotes the set of all regular expressions of Σ. It is defined inductively as follows:

- $\emptyset, \varepsilon \in \mathrm{Reg}(\Sigma)$,
- $\Sigma \subseteq \mathrm{Reg}(\Sigma)$,
- if $\alpha, \beta \in \mathrm{Reg}(\Sigma)$ then $(\alpha \cup \beta), (\alpha \cdot \beta), \alpha^* \in \mathrm{Reg}(\Sigma)$.

The language L defined by a regular expression α is inductively defined by

- $L(\emptyset) = \emptyset$, $L(\varepsilon) = \{\lambda\}$,
- $L(a) = \{a\}$ for $a \in \Sigma$,
- $L(\alpha \cup \beta) = L(\alpha) \cup L(\beta)$, $L(\alpha \cdot \beta) = L(\alpha)L(\beta)$, $L(\alpha^*) = L(\alpha)^*$.

Let

$$\begin{aligned}
\mathrm{RegEquiv}(\Sigma) &= \{(\alpha, \beta) \mid \alpha, \beta \in \mathrm{Reg}(\Sigma), L(\alpha) = L(\beta)\} \\
\mathrm{RegUniv}(\Sigma) &= \{\alpha \mid \alpha \in \mathrm{Reg}(\Sigma), L(\alpha) = \Sigma^*\}
\end{aligned}$$

Equivalence of regular expressions is **PSPACE**-complete

Theorem 34

RegEquiv(Σ) *and* RegUniv(Σ) *are* **PSPACE***-complete for every finite alphabet* Σ *with* $|\Sigma| \geq 2$.

Proof:

(1) RegEquiv(Σ) ∈ **PSPACE**.

Let $\alpha, \beta \in \mathrm{Reg}(\Sigma)$.

First, we transform α, β into equivalent nondeterministic finite automata A, B with $L(A) = L(\alpha)$, $L(B) = L(\beta)$.

This can be done in polynomial time (see the construction from GTI).

We check in polynomial space whether $L(A) \subseteq L(B)$ and $L(B) \subseteq L(A)$.

We only show how to check $L(A) \subseteq L(B)$, $L(B) \subseteq L(A)$ can be verified in the same way.

We have: $L(A) \subseteq L(B) \quad \Leftrightarrow \quad L(A) \cap (\Sigma^* \setminus L(B)) = \emptyset$

Equivalence of regular expressions is **PSPACE**-complete

Let $A = (Q_A, \Sigma, \delta_A, q_{0,A}, F_A)$ and $B = (Q_B, \Sigma, \delta_B, q_{0,B}, F_B)$.

The power set construction yields the following automaton for $\Sigma^* \setminus L(B)$:

$$B' = (2^{Q_B}, \Sigma, \delta'_B, \{q_{0,B}\}, \{P \subseteq Q_B \mid P \cap F_B = \emptyset\})$$

where for all $a \in \Sigma$, $P, R \subseteq Q_B$ we have:

$$(P, a, R) \in \delta'_B \iff R = \{q \in Q_B \mid \exists p \in P : (p, a, q) \in \delta_B\}.$$

We then obtain an automaton C for $L(A) \cap (\Sigma^* \setminus L(B)) = L(A) \cap L(B')$:

$$C = (Q_A \times 2^{Q_B}, \Sigma, \delta_C, (q_{0,A}, \{q_{0,B}\}), F_A \times \{P \subseteq Q_B \mid P \cap F_B = \emptyset\})$$

where for all $a \in \Sigma$, $p, r \in Q_A$, $P, R \subseteq Q_B$ we have:

$$((p, P), a, (r, R)) \in \delta_C \iff (p, a, r) \in \delta_A \land (P, a, R) \in \delta'_B$$

Equivalence of regular expressions is **PSPACE**-complete

We have to check in polynomial space whether $L(C) \neq \emptyset$.

Caution: the automaton C (as well as B') cannot be explicitly constructed; it does not fit into polynomial space!

Define the following directed graph

$$G = (Q_A \times 2^{Q_B}, \{((p,P),(r,R)) \mid \exists a \in \Sigma : ((p,P),a,(r,R)) \in \delta_C\}).$$

We have: $L(C) \neq \emptyset$ if and only if in the graph G there is a path from $(q_{0,A}, \{q_{0,B}\})$ to a state from $F_A \times \{P \subseteq Q_B \mid P \cap F_B = \emptyset\}$.

The latter can be checked nondeterministically in polynomial space:

- Guess a state $(p,P) \in F_A \times \{P \subseteq Q_B \mid P \cap F_B = \emptyset\}$ (can be stored in polynomial space).
- Guess a path from $(q_{0,A}, \{q_{0,B}\})$ to (p,P). Thereby we only have to store the current vertex from G, which fits into polynomial space.

Equivalence of regular expressions is **PSPACE**-complete

(2) RegUniv(Σ) is **PSPACE**-hard.

Let $L \in$ **PSPACE** and $L(M) = L$ for a $p(n)$-space bounded deterministic Turing machine $M = (Q, \Sigma', \Gamma, \delta, q_0, q_J, q_N, \Box)$,
$p(n) > n$ a polynomial.

Let $\Omega = (Q \times \Gamma) \cup \Gamma$.

Let $w = w_1 \cdots w_n \in \Sigma^*$ an input for M with $|w| = n \geq 1$.

Configurations of M are identified with words from the language
$\text{Conf} = \{\Box u(q,a)v\Box \mid (q,a) \in Q \times \Gamma,\ uv \in \Gamma^{2p(n)}\} \subseteq \Omega^{2p(n)+3}$.

There is a function $\Delta : \Omega^3 \to \Omega$ such that for all $\alpha, \alpha' \in \Box\Omega^*\Box$ with $|\alpha| = |\alpha'|$ we have:

$$\alpha, \alpha' \in \text{Conf and } \alpha \vdash_M \alpha'$$
$$\Longleftrightarrow$$
$$\alpha \in \text{Conf and } \forall i \in \{-p(n), \dots, p(n)\} : \Delta(\alpha[i-1], \alpha[i], \alpha[i+1]) = \alpha'[i].$$

Equivalence of regular expressions is **PSPACE**-complete

The initial configuration is $\alpha_0 := \Box^{p(n)+1}(q_0, w_1)w_2 \cdots w_n \Box^{p(n)-n+2}$.

An accepting computation (if it exists)

$$\alpha_0 \vdash_M \alpha_1 \vdash_M \alpha_2 \vdash_M \cdots \vdash_M \alpha_l \in \mathrm{Accept}_M$$

of M on input w is encoded by the word $\alpha_0\alpha_1\alpha_2\cdots\alpha_l \in \Omega^*$.

We construct from w a regular expression $\beta(w)$ (with a logspace transducer) such tat $L(\beta(w))$ is the set of all words over the alphabet Ω, which do not describe an accepting computation of M on input w.

Hence: $w \notin L(M)$ if and only if $L(\beta(w)) = \Omega^*$.

For $C \subseteq \Omega$ we identify the set C with the regular expression $\bigcup_{a \in C} a$.

Ω^k denotes the regular expression $\underbrace{\Omega \cdot \Omega \cdots \Omega}_{k \text{ many}}$.

Equivalence of regular expressions is **PSPACE**-complete

We have $\beta(w) = \beta_1 \cup \beta_2 \cup \beta_3 \cup \beta_4 \cup \beta_5$, where the regular expressions β_i $(1 \le i \le 5)$ are defined as follows:

(a) All words that do not have the right length:

$$\beta_1 = \varepsilon \cup \bigcup_{i=1}^{2p(n)+2} \left(\Omega^{2p(n)+3}\right)^* \Omega^i$$

(b) All words that do not beginn with the initial configuraton $\alpha_0 = \Box^{p(n)+1}(q_0, w_1)w_2 \cdots w_n \Box^{p(n)-n+2}$:

$$\beta_2 = \bigcup_{i=-p(n)-1}^{p(n)+1} \Omega^{i+p(n)+1} \cdot (\Omega \setminus \{\alpha_0[i]\}) \cdot \Omega^*$$

(c) All words, where a block of length $2p(n)+3$ does not begin or end with $\Box$:

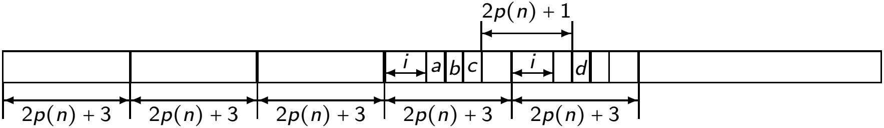
$$\beta_3 = \left(\Omega^{2p(n)+3}\right)^* (\Omega \setminus \{\Box\}) \cup \left(\Omega^{2p(n)+3}\right)^* \Omega^{2p(n)+2} (\Omega \setminus \{\Box\})$$

Equivalence of regular expressions is **PSPACE**-complete

(d) All words that "do not respect Δ somewhere":

$$\beta_4 = \bigcup_{i=0}^{2p(n)} (\Omega^{2p(n)+3})^* \Omega^i \left(\bigcup_{u \in \Omega^3} u \cdot \Omega^{2p(n)+1} \cdot (\Omega \setminus \{\Delta(u)\}) \right) \Omega^*$$

In the following picture we have $u = abc$ and $d \notin \Omega \setminus \{\Delta(u)\}$:



(e) All words that do not contain the accepting state q_J:

$$\beta_5 = (\Omega \setminus (\{q_J\} \times \Gamma))^*$$

Claim: $\Omega^* \setminus L(\beta(w)) = \bigcap_{i=1}^{5} (\Omega^* \setminus L(\beta_i))$ is the set of all words $\alpha_0 \alpha_1 \alpha_2 \cdots \alpha_l$ that describe an accepting computation of M on input w.

Equivalence of regular expressions is **PSPACE**-complete

That x belongs to $\bigcap_{i=1}^{5}(\Omega^* \setminus L(\beta_i))$ means:

- x has the form $\alpha_0\alpha_1\cdots\alpha_l$ with $|\alpha_i| = 2p(n) + 3$ for all $1 \le i \le l$ and α_0 is the initial configuration (due to β_1 and β_2).
- for all $1 \le i \le l$, we have $\alpha_i \in \Box\Omega^*\Box$ (due to β_3).
- for all $1 \le t \le l-1$ and all positions i with $|i| \le p(n)$ we have: $\alpha_{t+1}[i] = \Delta(\alpha_t[i-1], \alpha_t[i], \alpha_t[i+1])$ (due to β_4).

  Due to the equivalence from slide 161 (bottom) and the above points, this is equivalent to $\alpha_0 \vdash_M \alpha_1 \vdash_M \alpha_2 \vdash_M \cdots \vdash_M \alpha_l$.
- One of the configurations α_i must be accepting (due to β_5).

  This configuration must be α_l (since our Turing machines terminate when they reach the state q_J).

Together, these properties are equivalent to x being an accepting computation of M on input w.

Equivalence of regular expressions is **PSPACE**-complete

Finally, we encode the symbols from $\Omega = (Q \times \Gamma) \cup \Gamma$ by bit strings, in order to get **PSPACE**-hardness for every alphabet with at least two symbols.

Let us write Ω as $\Omega = \{a_1, a_2, \dots, a_k\}$.

We replace in the regular expression $\beta(w)$ every occurrence of the symbol a_i by $a^i b^{k-i}$.

Let $\beta'(w)$ be the resulting regular expression over the alphabet $\{a, b\}$.

In addition, we construct a regular expression β'' over $\{a, b\}$ such that

$$L(\beta'') = \{a, b\}^* \setminus \{a^i b^{k-i} \mid 1 \le i \le k\}^*.$$

We then have:

$$L(\beta'(w) \cup \beta'') = \{a, b\}^* \iff L(\beta(w)) = \Omega^* \iff w \notin L.$$

□