Non-determinism

Peter Hajnal

Bolyai Institute, SzTE, Szeged

2023 fall

Computation vs. Reasoning

We defined the complexity of a computation. For example, a decision problem L belonged to the class $\mathcal{P}$ if there existed a Turing machine that could decide it with a guarantee that, for any input ω, the output/decision occurs within a polynomial number of steps dependent on $|\omega|$.

However, sometimes we don't want to perform the entire computation. We suffice with the machine somehow demonstrating/showing/proving that $\omega \in L$ (assuming it is the case).

The concept of computability discussed so far was such that for a known input, it was clear what configuration sequence the machine followed.

The human brain is not like that (we think). Thinking/reasoning does not work this way.

Determinism vs. Non-determinism

To the original computability, we add an adjective: the defined Turing machine is deterministic.

There are also non-deterministic machines. Below, we provide two alternative definitions for non-deterministic Turing machines.

Non-determinism: Version I

Similar to deterministic Turing machines, there are tapes, heads, states, etc. Here, we describe only the single tape version.

Definition: Non-deterministic TM

The transition function for a non-deterministic TM:

$$\delta\colon \Sigma \times \Gamma \times S \to \mathcal{P}(\{\leftarrow, ., \rightarrow\} \times \Gamma \times \{\leftarrow, ., \rightarrow\} \times S) \setminus \{\emptyset\}.$$

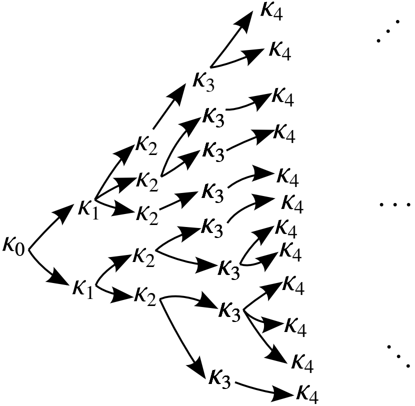
Non-determinism: Version I

That is, for a given configuration, not a single update rule is given, but a set of update rules.

For a given configuration, it is not necessary to specify a single succeeding configuration. Instead, a set of possible succeeding configurations is provided (each element of the set described by the transition function represents a possible succeeding configuration).

Non-determinism: Version I

Thus, the run for the input ω is not determined (in other words, non-deterministic), i.e., from the initial configuration $\kappa_0(\omega)$, multiple possible configurations can be reached. Thus, a tree rooted at $\kappa_0(\omega)$ describes the possible runs of the machine.



Non-determinism: Version I

To understand non-determinism, it is crucial to clarify when a machine computes a language.

Definition

For a non-deterministic machine to accept a language, every run must halt on every input.

The input ω is accepted by the non-deterministic Turing machine T if there exists a run leading to an ACCEPT state.

That is, rejecting ω is equivalent to all runs on ω leading to a REJECT state.

Non-determinism: Version II

Definition

In this case, we have an additional tape to the input and work tapes, called the witness/proof tape. We assume that its alphabet is Σ.

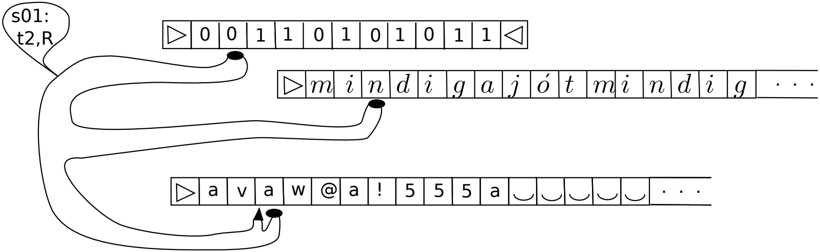
This tape is read-only, and the head can only move to the right.

The transition function is defined the same way as in the deterministic case, and the run is deterministic. That is, ω and τ (the content of the witness tape) uniquely determine a configuration sequence:

$$\kappa_0 = \kappa_0(\omega, \tau) \to \kappa_1 \to \kappa_2 \to \dots .$$

Non-determinism: Version II in Picture

The following diagram is a *photograph* of a configuration of a non-deterministic machine (II).


s01:
t2,R
▷ 0 0 1 1 0 1 0 1 0 1 1 ◁
▷ m i n d i g a j ó t m i n d i g · · ·
▷ a v a w @ a ! 5 5 5 a ␣ ␣ ␣ ␣ ␣ · · ·

Non-determinism: Version II

The problem arises again when we say that a non-deterministic machine accepts an L language. For this, it is necessary that the machine halts on any ω input with arbitrary witness tape content. The input ω is accepted by a non-deterministic Turing machine if there exists a witness tape content τ for which the run reaches an ACCEPT state.

During a witness tape τ, we may reach a REJECT state on the ω input. This does not necessarily mean that the input is incorrect. If $\omega \in L$, it means that τ is a bad choice/unconvincing witness. That's why in the non-deterministic case, we often give the name NOT-CONVINCED to the REJECT state. Rejection occurs when every τ witness tape content leads to the NOT-CONVINCED state.

Non-determinism: Relationship between the Two Versions

There is a significant difference between the two versions: in the first one, non-determinism is *scattered* during the run, and the final state is not determined until just before the last step.

In the second version, non-determinism occurs with the choice of τ. That is, we fix our possible decisions, alternatives, beforehand. After that the run is deterministic.

Theorem

L is decidable by I-non-deterministic TM if and only if it is decidable by II-non-deterministic TM.

We won't prove the theorem here, but an interested student can easily do so.

Break

Language Classes Based on Non-deterministic Computation

We can base our definitions on either version of non-determinism (so that we arrive at the same language classes). We follow the second (witness tape) perspective. Then $\mathit{TIME}(\omega, \tau; T)$ and $\mathit{SPACE}(\omega, \tau; T)$ can be defined by copying the deterministic case.

Definition: $\mathit{NTIME}(\omega; T)$

If $\omega \in L$ then $\mathit{NTIME}(\omega; T)$ is defined as

$$\min\{\mathit{TIME}(\omega, \tau; T) : \text{where } \tau \text{ is such that } T \text{ reaches the ACCEPT state on } \omega\}.$$

If $\omega \notin L$ then

$$\mathit{NTIME}(\omega; T) = \min\{\mathit{TIME}(\omega, \tau; T) : \text{where } \tau \in \Sigma^*\}.$$

In other words, among accepting runs, the *most brilliant* witness tape content determines the time limit.

Language Classes Based on Non-deterministic Computation

Definition

If $\omega \in L$ then $NSPACE(\omega; T)$ is defined as

$$\min\{SPACE(\omega,\tau;T) : \text{where } \tau \text{ is such that } T \text{ reaches the ACCEPT state on } \omega\}.$$

If $\omega \notin L$ then

$$NSPACE(\omega; T) = \min\{TIME(\omega,\tau;T) : \text{where } \tau \in \Sigma^*\}.$$

Most Common Classes

Definition

$$\mathcal{NP} = \{L : \text{there exists a non-deterministic Turing machine } T \text{ that accepts } L, \text{ and there exists } i \in \mathbb{N} \text{ such that for every } \omega,\ NTIME(\omega; T) \leq |\omega|^i + i.\}$$

$$\mathcal{NEXP} = \{L : \text{there exists a non-deterministic Turing machine } T \text{ that accepts } L, \text{ and there exists } i \in \mathbb{N} \text{ such that for every } \omega,\ NTIME(\omega; T) \leq 2^{|\omega|^i + i}.\}$$

The Most Common Classes (Continued)

Definition

$$\mathcal{NL} = \{L : \text{there exists a non-deterministic Turing machine } T \text{ that accepts } L\text{, and there exists } i \in \mathbb{N} \text{ such that for every } \omega,\ \mathit{NSPACE}(\omega; T) \le i \log(|\omega| + 1).\}$$

$$\mathcal{NPSPACE} = \{L : \text{there exists a non-deterministic Turing machine } T \text{ that accepts } L\text{, and there exists } i \in \mathbb{N} \text{ such that for every } \omega,\ \mathit{NSPACE}(\omega; T) \le |\omega|^i + i.\}$$

Most Common Classes (Continued)

Definition

$$\mathcal{NEXPSPACE} = \{L : \text{there exists a non-deterministic Turing machine } T \text{ that accepts } L\text{, and there exists } i \in \mathbb{N} \text{ such that for every } \omega,\ NSPACE(\omega; T) \leq 2^{|\omega|^i + i}.\}$$

Again, note that defined classes are robust: if we slightly change the definition of the Turing machine, the corresponding classes remain the same.

We could have introduced the above definitions based on the first version of non-determinism as well.

Complementation and Determinism

Observation

Deterministic classes are closed under complementation.

Example

If $L \in_T \mathcal{P}$, then $\overline{L} = \Sigma^* - L$ also belongs to $\mathcal{P}$.

To prove this, let $\widetilde{T}$ be the Turing machine obtained from T with the following simple modification: we change the transition function so that if T reaches the ACCEPT state, then $\widetilde{T}$ (keeping everything else the same) enters the REJECT state, and vice versa.

With this, $\widetilde{T}$ exactly accepts the inputs rejected by T. That is, the computed language is the complement language.

The complexities of T and $\widetilde{T}$ are the same.

Complementation and Non-determinism

The above observation is far from obvious in the non-deterministic case, if true at all.

The following definitions are justified.

Definition

$$
\begin{aligned}
co\,\mathcal{NP} =& \{\overline{L} : L \in \mathcal{NP}\},\\
co\,\mathcal{NEXP} =& \{\overline{L} : L \in \mathcal{NEXP}\},\\
co\,\mathcal{NL} =& \{\overline{L} : L \in \mathcal{NL}\},\\
co\,\mathcal{NPSPACE} =& \{\overline{L} : L \in \mathcal{NPSPACE}\},\\
co\,\mathcal{NEXPSPACE} =& \{\overline{L} : L \in \mathcal{NEXPSPACE}\},
\end{aligned}
$$

Obvious Inclusions

More time, more languages. More space, more languages.

The power of non-determinism is more languages.

These statements are obvious from the definitions (if *more* means at least as many).

It is also natural that limited time implies limited space usage.

Based on these, the following inclusions are obvious for the existing language classes:

$$\begin{array}{ccccccccc}
\mathcal{NL} & & \mathcal{NP} & \subseteq & \mathcal{NPSPACE} & & \mathcal{NEXP} & \subseteq & \mathcal{NEXPSPACE} \\
\rotatebox{90}{$\subseteq$} & & \rotatebox{90}{$\subseteq$} & & \rotatebox{90}{$\subseteq$} & & \rotatebox{90}{$\subseteq$} & & \rotatebox{90}{$\subseteq$} \\
\mathcal{L} & \subseteq & \mathcal{P} & \subseteq & \mathcal{PSPACE} & \subseteq & \mathcal{EXP} & \subseteq & \mathcal{EXPSPACE}
\end{array}$$

Technical Definitions

Definition

A function $t(n) : \mathbb{N} \to \mathbb{N}$ is called a nice time function if there is a Turing machine such that for every n-length input, it runs exactly for $t(n)$ time.

Definition

A function $s(n) : \mathbb{N} \to \mathbb{N}$ is called a nice space function if there is a Turing machine such that for every n-length input, it halts and touches exactly $s(n)$ cells on the work tape.

The above are technical conditions. However, all functions used so far are nice. For example, in the case of time, polynomial functions, 2^n, or in the case of space, $\lceil \log_2 n \rceil$, are also nice.

Example

Example

Assume $t(n)$ is a nice time function. Let T be an arbitrary Turing machine. Let ω be an arbitrary n-length input.

Copy ω to a work tape, then move the heads/hands back to the left ($2n$ time).

From here, simulate the run of T and with the help of the work tape simulate a machine W that exhibits the niceness of $t(n)$. The ACCEPT/REJECT state of T halts our machine.

We call the halting state of W the BUZZ state. We think of W as a clock. The BUZZ state stops the entire simulation.

So our new machine will definitely halt in $2n + t(n)$ time ($t(n) + 2n$ and $t(n)$ are of the same order of magnitude). It performs the computations of T if they fit into $t(n)$ time.

Example

Example

Assume $s(n)$ is a nice space function. Let T be an arbitrary Turing machine. Let ω be an arbitrary n-length input.

First, simulate the machine W exhibiting the niceness of $s(n)$. Then, overwrite the used cells with an empty symbol and put a SO-FAR character behind them.

After that, start the simulation of the T machine. If we read the SO-FAR character, we stop with a LOT-OF-MEMORY state.

The ACCEPT/REJECT state of T halts our machine. So our new machine will use $1 + s(n)$ space ($s(n)$ and $s(n) + 1$ are of the same order of magnitude).

The new machine performs the computations of T if they fit into $s(n)$ space. Furthermore ther will be two halting congiguration on any input (LOT-OF-MEMORY$\equiv$REJECT$\equiv$NON-CONVINCED).

Example

Example

Let T be a non-deterministic machine with a time complexity of $t(n)$ that computes the language L. It can have many runs of which we do not know anything about the time.

Based on the example above, IF $t(n)$ is nice, THEN it can be assumed that our machine stops in $t(n) \approx t(n) + 2n$ steps for every run. The halted runs do not change the accepted language.

Due to the time complexity condition, if $\omega \in L$, there will be a run leading to the ACCEPT state that terminates before the BUZZ state. The simulating machine *detects* this.

Example

Example

Let $s(n)$ be a nice space function. Let T be an $s(n)$ space-bounded machine. Then, we can achieve that our machine has exactly two different halting configurations:

Run T. However, at the end of the computation, *keep to ourselves* the result and before announcing the ACCEPT/REJECT states, erase the work tape in $s(n)$ length (due to niceness, this can be easily done).

Move every head/hand to the left. After that, reach the halting state corresponding to the computed result.

We have two different halting configurations (*photo* of the machine), and of course, our machine computes the same thing as the original.

Remark

Note that we can also use the nice time function (let W be the Turing machine proving this) to mark cells:

During the simulation of W, all other work tapes with heads/hands continuously move to the right until the BUZZ state. Then, beyond the tapes used by W, exactly $t(n)$ cells are marked

on the other tapes.

Break

Goal

Our goal is to understand the following inclusion chain:

$$\mathcal{L} \subset \mathcal{NL} \overset{(2)}{\subset} \mathcal{P} \subset \mathcal{NP} \overset{(1)}{\subset} \mathcal{PSPACE} \subset \mathcal{NPSPACE} \overset{(2)}{\subset} \mathcal{EXP} \subset \mathcal{NEXP}.$$

We numbered the still unproven inclusions.

Below, we will prove these. However, our goal is not to provide the shortest justification, but to summarize the results and introduce the methods for later use.

Warm-Up

Triviality A

$\mathcal{TIME}(t(n)) \subset \mathcal{SPACE}(t(n))$.

Indeed, the time constraint limits how far the work tape head/hand can move.

Triviality B

(i) $\mathcal{TIME}(t(n)) \subset \mathcal{NTIME}(t(n))$.

(ii) $\mathcal{SPACE}(s(n)) \subset \mathcal{NSPACE}(s(n))$.

Indeed, determinism can be seen as a special case of non-determinism.

Observation

Observation

$\mathcal{NTIME}(t(n)) \subset \mathcal{SPACE}(t(n))$, where $t(n)$ is a nice time function.

Let $L \in_T \mathcal{NTIME}(t(n))$. That is, T is a witness-tape deterministic Turing machine. In other words, T accepts the language L and for every input ω its time complexity is $t(|\omega|)$.

To prove the statement, we construct a deterministic Turing machine $\widetilde{T}$ based on T, which accepts the same language and has a space limit of $t(n)$.

For this, we keep the working tapes needed for the description of T and add one that plays the role of the witness tape and another that plays the role of a clock ($t(n)$ is a nice time function).

Of course, the new machine does not possess the *genius*/guessing property of non-deterministic machines.

The Plan

To describe the operation of $\widetilde{T}$, we outline how one of its runs looks like. From this, the transition function (formal description) can be read.

We assume that the length of our input is n.

$\widetilde{T}$: Initialization Phase

On the work tape playing the role of the witness tape, we mark $t(n)$ cells, which are closed with a special delimiter from Γ. This is a character used only for this purpose. When reading this character, we know that within the space limit, we cannot move to the right.

Since $t(n)$ is a nice time function, we can take a clock that *ticks* after $t(n)$ steps (and, of course, can be wound up again).

With the help of this clock, we can easily mark the area of the tape: we move to the right until the clock ticks.

On the work tape playing the role of the witness tape, we write the first possible $t(n)$ characters, which can be the beginning of a witness.
// We don't need more characters because a time-limited machine cannot read more.

$\widetilde{T}$: Simulation Phase

On the tapes corresponding to the working tapes of the T Turing machine, we simulate the run of T for the first witness for $t(n)$ time. The simulation either ends in the ACCEPT or the NON-AGREE state, or the time runs out/we run out of $t(n)$ time. Even the latter is considered as rejecting the tested witness (NON-AGREE state).

If the simulation reaches the ACCEPT state, we also accept the input, and $\widetilde{T}$ stops. If it reaches the NON-AGREE state, then on the tape playing the role of the witness tape, we overwrite its content with the next possible $t(n)$-length witness beginning. We clear the contents of the other tapes. We repeat the Simulation phase.

If the next witness start cannot be generated because we have tested all possible witness beginnings (the witnesses are exhausted), then we stop with the DISAGREE state.

$\widetilde{T}$: Properties

Now, the following two statements easily follow from the previous ones:

(1) $\widetilde{T}$ computes the language L,

(2) The space requirement of $\widetilde{T}$ is at most $t(n)$.

With this observation, we have established the result.

Specifically, the containment marked with (1) is implied.

Observation

The discussion of non-determinism previously showed that

$$\mathcal{SPACE}(s(n)) \subseteq \cup_{c\in\mathbb{N}} \mathcal{TIME}(c^{s(n)+\log(n+1)}).$$

Now we extend this to the non-deterministic case.

The observation could be proven relatively quickly. However, we choose a slower path. Our method will be crucial later. The reasoning of the proof will be important later.

Observation

$\mathcal{NSPACE}(s(n)) \subseteq \bigcup_{c\in\mathbb{N}} \mathcal{TIME}(c^{s(n)+\log(n+1)})$, where $s(n)$ is a nice space function.

Preparations

Let $L \in_T \mathcal{NSPACE}(s(n))$. That is, T is a non-deterministic Turing machine in the first sense, meaning the transition function is non-deterministic, the run can *branch*. T computes L and its space requirement is $s(n)$.

We can assume about T that upon halting, the input and work head are positioned at the beginning of the tape, and the first $s(n)$ characters of the work tape are empty (the machine wipes out its workspace). This way, two configurations can occur at the end of the run. Specifically, in the case of an accepting run, we know what the last configuration is.

Reduced Configuration

A *reduced configuration* for an I-non-deterministic Turing machine with space requirement $s(n)$ on input ω contains the following components:

(1) position of the input and work head,

(2) the first $s(n)$ characters of the work tape,

(3) the state of the machine.

Essentially, we have covered only the input tape content and the guaranteed unread/untouched portion of the work tape in the (full) configuration.

Let V be the set of reduced configurations. Then

$$|V| \leq \alpha_T \cdot (n+1) \cdot \beta_T^{s(n)}.$$

From a configuration κ, we can easily obtain the corresponding reduced configuration $\rho = \mathrm{red}(\kappa)$. If the ω input is known, then conversely, it is also true: ω and the reduced configuration ρ determine the full configuration $\kappa = \mathrm{conf}(\omega, \rho)$.

The Graph

Definition

Let T be a I-non-deterministic Turing machine and ω an input. Then $\overrightarrow{G}_{\omega,T}$ is the graph of reduced configurations associated with (T,ω). This is a directed graph where the set of vertices is the above V set, and $\overrightarrow{uv}$ exists if and only if the transition function allows the configuration $\mathrm{konf}(\omega, v)$ after the configuration $\mathrm{konf}(\omega, u)$.

Let V have a special element $v_0 = \mathrm{red}(\kappa_0(\omega))$ as the starting reduced configuration.

Let v_1 be the reduced configuration corresponding to an accepting halt.

Note that for a deterministic machine, the above concepts can also be introduced. In this case, every vertex in the defined directed graph would have outdegree 1.

Observation

Observation

$\omega \in L$ holds if and only if there exists a directed path $v_0 v_1$ in $\overrightarrow{G}_{\omega,T}$.

Indeed: $\omega \in L$ is equivalent to the existence of an accepting run on ω. These runs can be paired with the directed paths starting from v_0. A run is accepting if the corresponding path leads to v_1.

The graph defined above can be calculated efficiently.

Lemma

Lemma

There exists a deterministic machine $T_1(T)$ that, on input ω, computes the code of the triple $(\overrightarrow{G}_{\omega,T}, v_0, v_1)$.

Furthermore, the deterministic space requirement of $T_1(T)$ is

$$\alpha_T(s(n) + \log(n+1)).$$

The essence of the following statement is that for the computation of the graph behind T, ω, a very small amount of space on the work tape is needed. There is no *savings* at this point. Its significance will be revealed later.

The given space is sufficient to encode a constant number of reduced configuration codes on the work tape.

Proof of the Lemma

• For us, two spaces are needed for two reduced configurations. At the beginning of the run, we designate two blocks at the beginning of the work tape, each serving to store a reduced configuration ($s(n)$ nice space function).

• In the first block, we enumerate all possible code words. These labels encode the vertices of our graph. For each label, we decide whether it is the code of a reduced configuration. (After fixing a natural encoding rule, this task can be easily solved.)

• If not, we move on to the next label. If yes, we copy it to the output tape, followed by a ':'. After that, we write down the sequence of out-neighbors.

Proof of the Lemma (Continued)

• In the second block of the work tape, we also begin the enumeration of reduced configurations. If the codes of x and y reduced configurations appear on the work tape, we decide whether there is an edge from x to y. The input tape contains ω, and the T Turing machine — with a finite description — is known to us. The implementation of this subtask is again simple.

• If we get an edge, we copy y to the output tape. If we don't get an edge, we immediately move on to the next y search.

• If y is exhausted, we move on to the next x search. If the x's are also exhausted, then we have computed the code of $\overrightarrow{G}_{\omega,T}$.

Proof of the Lemma (Continued)

• Writing the codes of v_0 and v_1 onto the output tape is also easily achievable.

• We did not detail the solution of the subproblems. During their implementation, we do not need to exceed the given space constraint.

• The Lemma follows.

The Long-Awaited Proof

- $L \in_T \mathcal{NSPACE}(s(n))$. Run $T_1(T)$ on ω and write down the code of $\overrightarrow{G}_{\omega,T}, v_0, v_1$ on an additional work tape.

- We previously estimated the space requirement of this deterministic procedure, but what we need now is an estimate of its runtime. Its runtime is at most $2^{\beta_T(s(n)+\log(n+1))}$. The length of the calculated code is also at most $2^{\beta_T(s(n)+\log(n+1))}$.

- Decide whether there is a directed path $v_0 v_1$ in $\overrightarrow{G}_{\omega,T}$.

- There are several solutions to this. For example, breadth-first search.

- The algorithm can be realized on a Turing machine (T_2). Its runtime (without any special ideas) is polynomial in the length of the input.

- $T_1(T)$ and T_2 together precisely decide the L language, and their time complexity is $2^{\gamma_T(s(n)+\log(n+1))}$. This completes the proof.

Our Current Knowledge

We can supplement the proven inclusions with the classes of complements of the non-deterministic classes:

$$\begin{tikzcd}
 & \mathcal{NL} \arrow[dr, phantom, sloped, "\subseteq"] & & \mathcal{NP} \arrow[dr, phantom, sloped, "\subseteq"] & \\
\mathcal{L} \arrow[ur, phantom, sloped, "\subseteq"] \arrow[dr, phantom, sloped, "\subseteq"] & & \mathcal{P} \arrow[ur, phantom, sloped, "\subseteq"] \arrow[dr, phantom, sloped, "\subseteq"] & & \mathcal{PSPACE} \\
 & co\,\mathcal{NL} \arrow[ur, phantom, sloped, "\subseteq"] & & co\,\mathcal{NP} \arrow[ur, phantom, sloped, "\subseteq"] &
\end{tikzcd}$$

$$\begin{tikzcd}
 & \mathcal{NSPACE} \arrow[dr, phantom, sloped, "\subseteq"] & \\
\mathcal{PSPACE} \arrow[ur, phantom, sloped, "\subseteq"] \arrow[dr, phantom, sloped, "\subseteq"] & & \mathcal{EXP} \\
 & co\,\mathcal{NSPACE} \arrow[ur, phantom, sloped, "\subseteq"] &
\end{tikzcd}$$

Further Coincidences

$$\mathcal{NL} = \text{co}\mathcal{NL},$$

$$\mathcal{PSPACE} = \mathcal{NPSPACE} = \text{co}\mathcal{NPSPACE}.$$

We will discuss these relationships later if time allows.

Further Genuine Inclusions

It is also true that with a substantial increase in time or space constraints, we obtain larger classes:

$$\mathcal{L} \subsetneq \mathcal{PSPACE},$$

$$\mathcal{P} \subsetneq \mathcal{EXP}.$$

However, more than this is not known.

The question of whether „The inclusion $\mathcal{P} \subseteq \mathcal{NP}$ is proper, or an equality holds?" is considered a central problem in 21st-century mathematics.

Break

IDEAL-ELEMENT-TEST

IDEAL-ELEMENT-TEST

Input: a finitely generated ideal in the polynomial ring $\mathbb{Q}[x_1, x_2, \ldots, x_n]$ and a polynomial p. The ideal is given by generators $g_1, g_2, \ldots, g_N$. The question is whether p belongs to the ideal.

It is easy to describe the ideal: polynomials of the form $\alpha_1 g_1 + \alpha_2 g_2 + \ldots + \alpha_N g_N$, where α_i are polynomials as well. To provide an efficient non-deterministic algorithm based on this, we would need an estimate for the coefficients proving belonging to the ideal (their degrees and coefficients). This is not simple.

Using Gröbner bases theory, we can devise an algorithm with complexity $\mathcal{EXPSPACE}$. That is,

$$\text{IDEAL-ELEMENT-TEST} \in \mathcal{EXPSPACE}.$$

IDEAL-COMPLETENESS

IDEAL-COMPLETENESS

Input: a finitely generated ideal in the polynomial ring $\mathbb{Q}[x_1, x_2, \ldots, x_n]$. The ideal is given by generators $g_1, g_2, \ldots, g_N$. The question is whether the ideal is the entire ring, i.e., whether 1 belongs to the ideal.

This is obviously a special case of the previous problem. Its complexity is at most the same as that of the previous question.

Using Gröbner bases theory, we can devise an algorithm with complexity $\mathcal{PSPACE}$. That is,

$$\text{IDEAL-COMPLETENESS} \in \mathcal{PSPACE}.$$

SLIDING-BLOCK-PUZZLE

SLIDING-BLOCK-PUZZLE

Input: an $n \times m$ grid with non-overlapping rectangles placed on it (as the base path). The rectangles do not cover the entire grid, allowing them to be slid around (parallel to the sides of the base grid, respecting non-overlapping). We need to decide whether, from the initial configuration, we can reach a target configuration by sliding the rectangles. That is, is one of the target configurations reachable (for example, can we move one of the rectangles to a specified position)?

HAMILTON

HAMILTON

HAMILTON problem's input is a graph. We need to decide if it contains a Hamiltonian cycle.

Let's describe a non-deterministic Turing machine: We expect the witness tape to contain a list of vertices. We must check that consecutive vertices are connected in the graph, and the first and last vertices are also connected. We also need to check the „promise" that every vertex is listed exactly once.

Our tests can certainly be carried out in polynomial time. If all of these tests pass, then the witness proves that there is a Hamiltonian cycle in the input graph. On the other hand, for every graph with a Hamiltonian cycle, a witness can be found. Thus, we have that

$$\text{HAMILTON} \in \mathcal{NP}.$$

HAMILTON (continued)

For $co\mathcal{NP}$ membership, we would need to efficiently justify the absence of a Hamiltonian cycle.

It's easy to design a polynomial-time Turing machine that tests if the content of the witness tape is a set U of vertices and if the number of components of $G - U$ is greater than the number of elements in U.

If yes, then we can be sure that our graph does not have a Hamiltonian cycle. Indeed, after removing U, the remaining edges of the Hamiltonian cycle would guarantee that we don't have more components than $|U|$.

However, the machine described above does NOT prove the $co\mathcal{NP}$ membership of the HAMILTON language. It is not true that the absence of a Hamiltonian cycle can be reliably proven in this way. The Petersen graph, for example, does not have a Hamiltonian cycle. The above machine would not accept it.

LP-TESTING

LP-TESTING problem

Input: a matrix $A \in \mathbb{Q}^{m\times n}$ and a column vector $b \in \mathbb{Q}^{m\times 1}$.

We need to decide whether the $Ax = b$ system of equations $(x = (x_1, x_2, \ldots, x_n)^T)$ has a non-negative solution.

In fact, we can work with integers as well. We can multiply our equations by the least common multiple of the denominators in the input, effectively clearing the fractions. The size of the original input system's coefficient description can be estimated as a polynomial (square) of the original input size.

LP-TESTING problem (continued)

The $\mathcal{NP}$ membership seems simple. We just need to write a solution on the witness tape. The machine only checks this.

The problem is that the check is only polynomial in the size of the witness numbers (instead of the input numbers).

That is, we must be careful not to have a witness significantly longer than the input size. Such witnesses exist. The reasoning for this is not presented here.

Theorem

$$\text{LP-TESTING} \in \mathcal{NP}.$$

LP-TESTING problem (continued)

A method is described to establish the insolubility of an integer system in non-negative numbers.

The multiple of our equations and their sum is a consequence of the original system.

If we make this inference in a way that every coefficient in the combined linear expression is non-negative, while a negative number appears on the right side, it will be very transparent that the deduced system has no non-negative solution. Thus, the original system cannot have one either.

As with the previously unexplained reasoning, it can be shown that among the conclusions of the proof, there is also one that can be handled with combinable coefficients.

LP-TESTING problem (continued)

So, it can be read from the witness tape and tested in polynomial time. The strategy described above leads to an $\mathcal{NP}$ algorithm if it is true that if an input system is unsolvable, such a proof can be found for it. This is true, and this is the well-known Farkas' Lemma.

Therefore,

$$\text{LP-TESTING} \in co\mathcal{NP}.$$

LP-TESTING problem (continued)

Currently, there are several algorithms for linear programming that run in polynomial time. That is,

$$\text{LP-TESTING} \in \mathcal{P}.$$

- Note that LP-TESTING is one decision version of the linear programming optimization problem.
- Its $\mathcal{NP}$ membership is based on classic estimates.
- Its $co\mathcal{NP}$ membership is based on Farkas' Lemma, published by Gyula Farkas in 1902.
- The celebrated simplex algorithm for LP optimization was published in 1947 (Dantzig).
- In 1972, Klee and Minty proved that the algorithm is not polynomial (in fact, it is exponential).
- The first polynomial algorithm was given by Kachian in 1979.

PRIME-TESTING

PRIME-TESTING

The input to the problem is a positive integer *n* (encoded, for example, in base 10). We need to decide whether it is a prime number.

In the context of PRIME-TESTING, the simple task is to prove non-primality. For this, we only need to produce a proper divisor as a witness. It is easy to verify divisibility (and truth) of the witness.

This implies that

$$\text{PRIME-TESTING} \in \text{co}\mathcal{NP}.$$

PRIME-TESTING (continued)

Pratt's proof scheme (1975) shows that

$$\text{PRIME-TESTING} \in \mathcal{NP}.$$

The Agrawal-Kayal-Saxena primality test leads to the following theorem:

$$\text{PRIME-TESTING} \in \mathcal{P}.$$

PERFECT-MATCHING-TESTING

PERFECT-MATCHING-TESTING

The input is a simple graph. We need to decide whether the input contains a perfect matching.

We do not discuss the encoding of the input. However, we note that in the above sense, v, the number of vertices, can also be considered the size of the input (instead of the length of the code).

PERFECT-MATCHING-TESTING (continued)

First, we describe a nondeterministic algorithm. We use the second interpretation of nondeterminism, deciding acceptance with the help of the content of a witness tape. The content of the witness tape will be a set M of pairs of vertices.

The machine T tests whether the pairs of vertices are connected by an edge, and each vertex appears in exactly one pair. If the answer is yes for both tests, then we reach the ACCEPT state. If the witness fails any of the tests, the machine transitions to the REJECT state.

For the existence of a perfect matching, it is easy to provide a proving witness. If there is no perfect matching, then each witness will fail.

The tests can be easily performed in polynomial time. Thus, we have that

$$\text{PERFECT-MATCHING-TESTING} \in \mathcal{NP}.$$

PERFECT-MATCHING-TESTING (continued)

With knowledge of Tutte's theorem, we have a simple task if we want to prove the non-existence of a perfect matching.

Let the content of the witness tape be a set T. For $\omega \equiv G$ graph and $\tau \equiv T$ set, the machine determines the components of $G - T$, counts the components with odd size (number of vertices), and compares this number with $|T|$. If $|T|$ is smaller than the number of odd-size components, the machine transitions to the ACCEPT state (we define the machine for the complement language; acceptance means that the complement language is an element, i.e., there is no perfect matching). Otherwise, the machine transitions to the REJECT state.

The proof of the polynomial realizability of the machine is left to the reader.

PERFECT-MATCHING-TESTING (continued)

The correctness of the algorithm (if there is no maximum matching in G, then a suitable witness T proves it) is just Tutte's theorem.

Thus, we obtain the following

$$\text{PERFECT-MATCHING-TESTING} \in \text{co}\mathcal{NP}.$$

PERFECT-MATCHING-TESTING (continued)

The implementation of the Edmonds algorithm on a Turing machine is a polynomial algorithm.

This (the quite complex) algorithm leads to a stronger statement than the previous two results:

$$\text{PERFECT-MATCHING-TESTING} \in \mathcal{P}.$$

DIRECTED-REACHABILITY

DIRECTED-REACHABILITY

Given a simple directed graph $\overrightarrow{G}$ and two vertices s and t, decide if there is a directed st walk in $\overrightarrow{G}$.

Naturally, we need to encode the input $(\overrightarrow{G}, s, t)$. DIRECTED-REACHABILITY is the set of codes for which the graph component contains an st walk.

The breadth-first search algorithm implies that

$$\text{DIRECTED-REACHABILITY} \in \mathcal{P}.$$

DIRECTED-REACHABILITY (continued)

The algorithm we provide is a nondeterministic algorithm.

We can think of it as an algorithm for a *wandering* walker in the $\overrightarrow{G}$ graph. During the walk, we check if we are at t and count how many steps we have taken so far.

If we reach t, we halt with the ACCEPT state.

If we don't reach t, we check if we have taken v steps. If yes, we transition to the REJECT/NON-CONVVINCED state.

If we haven't taken so many steps, we nondeterministically choose a vertex. We check if we can move from the previous vertex to this one via an edge. If not, we again transition to the REJECT state. If yes, we erase the previous vertex (!).

DIRECTED-REACHABILITY

Of course, we keep the place of the erased vertex for the later part of the walk. This ensures that there are at most two vertices on the tape at any moment during the run and a counter with a value of at most v. The required tape space is $\mathcal{O}(\log v)$.

Thus, we have that

$$\text{DIRECTED-REACHABILITY} \in \mathcal{NL}.$$

DIRECTED-REACHABILITY (continued)

We present another deterministic algorithm, which uses space very economically:

$$\begin{aligned}\text{DIRECTED-REACHABILITY} &\in \bigcup_{\alpha\in\mathbb{N}} \mathcal{SPACE}(\alpha \log_2^2 n)\\ &= \mathcal{SPACE}(\log^2 n).\end{aligned}$$

This is proven by the following recursive algorithm.

DIRECTED-REACHABILITY (continued)

Directed-Reachability:
BOUNDED-DIRECTED-REACHABILITY($x, y, 2^\ell$)

// Given vertices x and y, it tests whether there is a walk between them of at most 2^ℓ steps. We can think of this walk as a *lazy* walk. At each step, we have two options: either move to a neighbor or stay put. In a lazy walk, we can assume that the length is exactly 2^ℓ.

If $\ell = 0$, then test if $x = y$ or $\overrightarrow{xy}$ is an edge. If the test succeeds, halt with the ACCEPT state, otherwise halt with the REJECT state.

If $\ell > 0$, then For each $k \in V$,
// k is the middle point of the lazy walk, meaning there is a lazy walk of $2^{\ell-1}$ steps from x to k and from k to y. We try all possibilities.

DIRECTED-REACHABILITY (continued)

Directed-Reachability: BOUNDED-DIRECTED-REACHABILITY($x, y, 2^\ell$) (continued)

```
(1) BOUNDED-DIRECTED-REACHABILITY($x, k, 2^{\ell-1}$)
    if NO, then next $k$ and back to (1)
    if NO and there is no next $k$ ($V$ exhausted), then REJECT.
    if YES, then
(2) BOUNDED-DIRECTED-REACHABILITY($k, y, 2^{\ell-1}$)
    if YES, then ACCEPT state and halt
    if NO, then next $k$ and back to (1)
    if NO, and there is no next $k$ ($V$ exhausted), then back to (1)
              to the NO branch.
```

The above algorithm, when run with $\ell = \lceil \log_2 |V(G)| \rceil$, solves reachability. The algorithm is implemented on a Savitch Turing machine in a space-efficient way.

DIRECTED-REACHABILITY (continued)

Savitch's Theorem

The recursive algorithm described above can be implemented on a Turing machine in such a way that at any configuration, at most ℓ blocks are used on the work tape, where each block has a length of at most $\mathcal{O}(\log|V|)$ (enough to store a finite number of vertices).

We provide only ideas for the exact implementation:

DIRECTED-REACHABILITY (Continuation)

- The structure of the recursion can be represented with a tree.

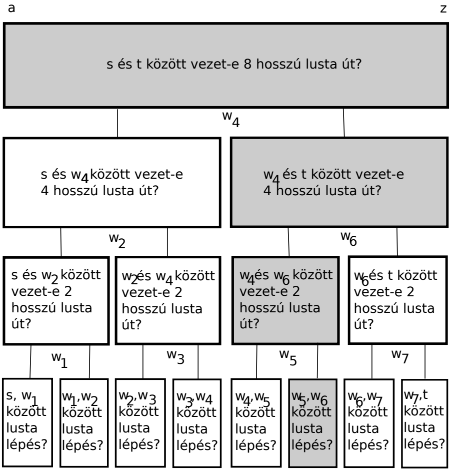
- The root of the tree is the DIRECTED-REACHABILITY problem, i.e., whether there is a lazy walk of length 2^ℓ.

- Every p=(u to v has a lazy walk of length 2^ℓ) problem breaks down into two subproblems.

- For a middle vertex w, $p_{left}(w)=$ (does a lazy walk of length $2^{\ell-1}$ lead from u to w), and $p_{right}(w)=$ (does a lazy walk of length $2^{\ell-1}$ lead from w to v) are the two subproblems of the p problem.

- The two tasks are *siblings* to each other.

DIRECTED-REACHABILITY (Continuation)


a
z
s és t között vezet-e 8 hosszú lusta út?
w_4
s és w_4 között vezet-e 4 hosszú lusta út?
w_4 és t között vezet-e 4 hosszú lusta út?
w_2
w_6
s és w_2 között vezet-e 2 hosszú lusta út?
w_2 és w_4 között vezet-e 2 hosszú lusta út?
w_4 és w_6 között vezet-e 2 hosszú lusta út?
w_6 és t között vezet-e 2 hosszú lusta út?
w_1
w_3
w_5
w_7
s, w_1 között lusta lépés?
w_1,w_2 között lusta lépés?
w_2,w_3 között lusta lépés?
w_3,w_4 között lusta lépés?
w_4,w_5 között lusta lépés?
w_5,w_6 között lusta lépés?
w_6,w_7 között lusta lépés?
w_7,t között lusta lépés?

The content of the work tape always represents a task (node in the tree) along with the path to the root. One example is highlighted with shading.

DIRECTED-REACHABILITY (Continuation)

The essence of the Savitch implementation is that the description of the path in the figure fits into the memory required by the theorem.

Furthermore, updating the path can be solved with a Turing machine.

The details of the complete implementation go beyond the scope of the course.

This is the End!

Thank you for your attention!