Logics and the Limits of Computation

CLAWS - *Computational Logic and Applications Winter School*

Guangdong Technion Israel Institute of Technology

Santiago Figueira

University of Buenos Aires & CONICET
Visiting Professor at GTIIT

October 18, 2025

Introduction

The limits of mechanization in mathematics

- Throughout history, mathematics has relied on the notion of a procedure to solve problems.
- But what exactly do we mean by a method that can always be carried out?
- Examples:
  - Long division algorithm
  - Euclid's algorithm for the gcd
  - Proof procedures in logic
- This motivates the need for a rigorous definition of effective method.
- Among the effective methods, which are the efficient ones?

In this talk I will cover...

- Part I - Computability: Which problems can be solved by an effective method?
- Part II - Complexity: Among the computable problems, which can be solved *efficiently* in terms of time or space?

Part I: Computability

- What does it mean for a problem to have an effective solution?
- Which problems can be solved by an effective method?

Origins of Computability

Origins of Computability

- Early 20th century: Hilbert posed the question of whether mathematics could be made completely mechanical.
- His vision: a formal system where every mathematical truth could, in principle, be decided by a finite procedure.
- This culminated in the famous Entscheidungsproblem (the decision problem):

  Is there an effective method to decide whether a logical first-order formula is universally valid?

- But, what is an effective method?

Origins of Computability

- Early 20th century: Hilbert posed the question of whether mathematics could be made completely mechanical.
- His vision: a formal system where every mathematical truth could, in principle, be decided by a finite procedure.
- This culminated in the famous Entscheidungsproblem (the decision problem):

  Is there an effective method to decide whether a logical first-order formula is universally valid?

- But, what is an effective method?
- Modern answer: a function programmable in Python (or replace Python by your favorite programming language, as they turn out to be equivalent).[1]

[1] With two provisos: (1) the time needed for the computation is not a problem, and (2) we need to assume unbounded memory.

Origins of Computability

- Early 20th century: Hilbert posed the question of whether mathematics could be made completely mechanical.
- His vision: a formal system where every mathematical truth could, in principle, be decided by a finite procedure.
- This culminated in the famous Entscheidungsproblem (the decision problem):

  Is there an effective method to decide whether a logical first-order formula is universally valid?

- But, what is an effective method?
- Modern answer: a function programmable in Python (or replace Python by your favorite programming language, as they turn out to be equivalent).[1]
- But in 1900 there was no such thing as a programming language!

[1] With two provisos: (1) the time needed for the computation is not a problem, and (2) we need to assume unbounded memory.

Church, Turing, and the birth of Computability

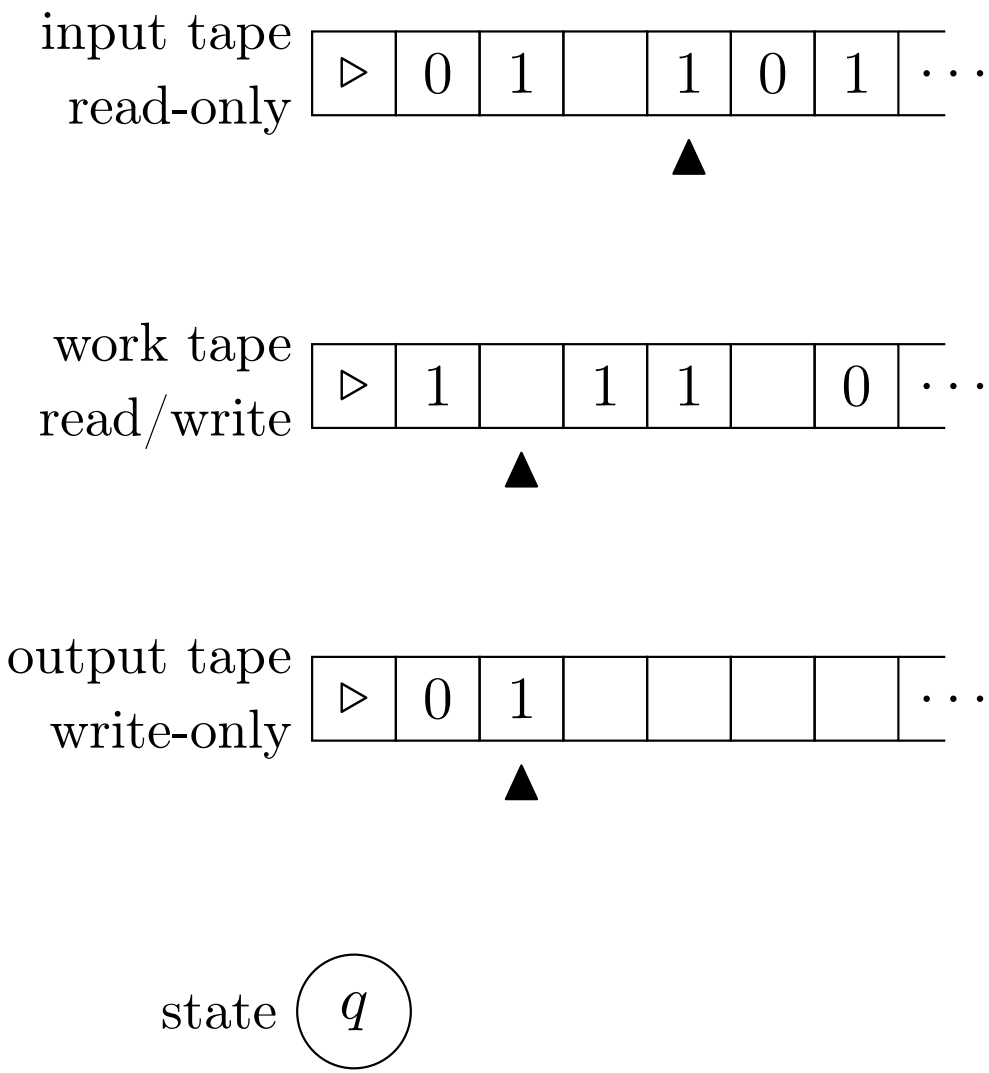
- 1930s: Attempts to formalize the vague notion of an effective procedure.
- **Alonzo Church**: λ-calculus.
- **Alan Turing**: Turing machines.
- Both models turned out to be equivalent $\Rightarrow$ Church–Turing Thesis.
- Consequence: the Entscheidungsproblem is unsolvable: there is no algorithm to decide validity in first-order logic.
- This marks the birth of Computability Theory.

Guiding question

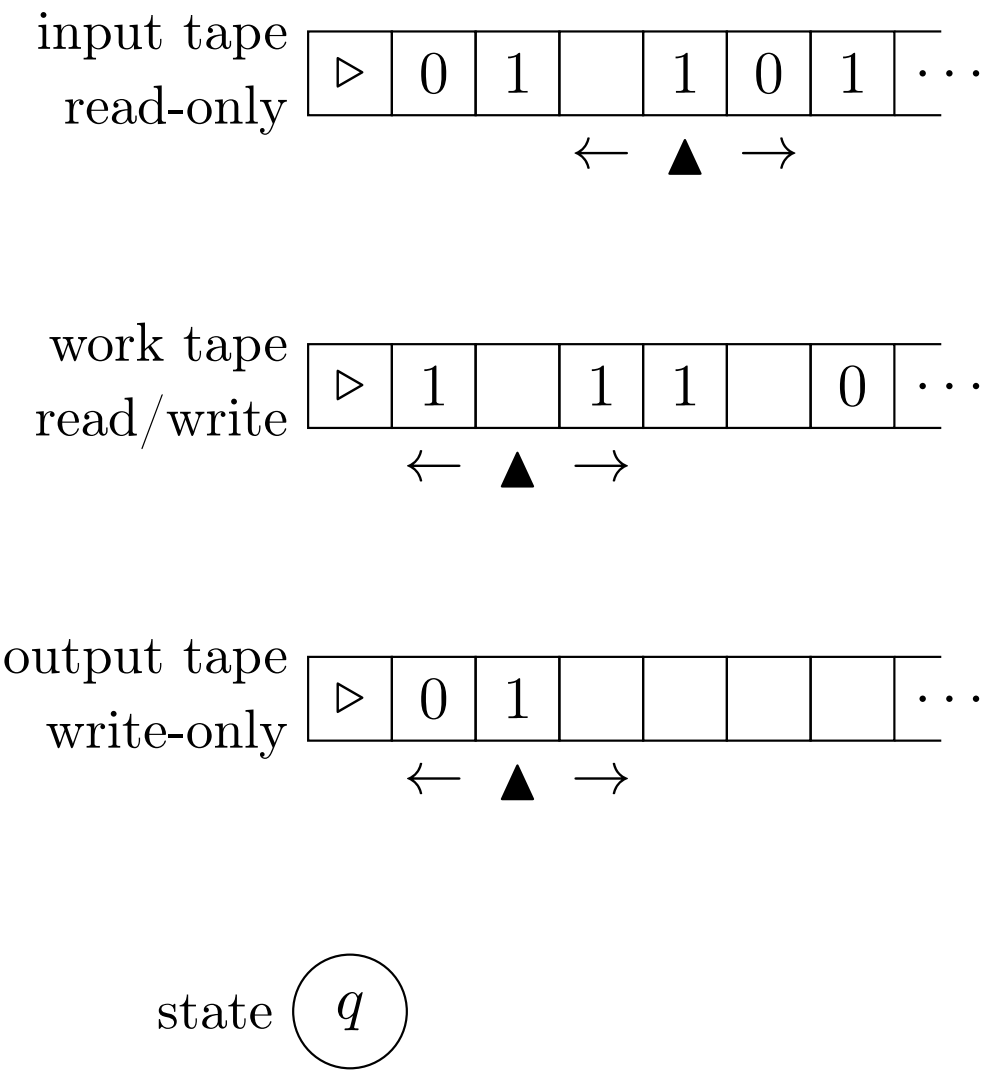
What does it mean for a problem to have an effective solution?

Turing machines

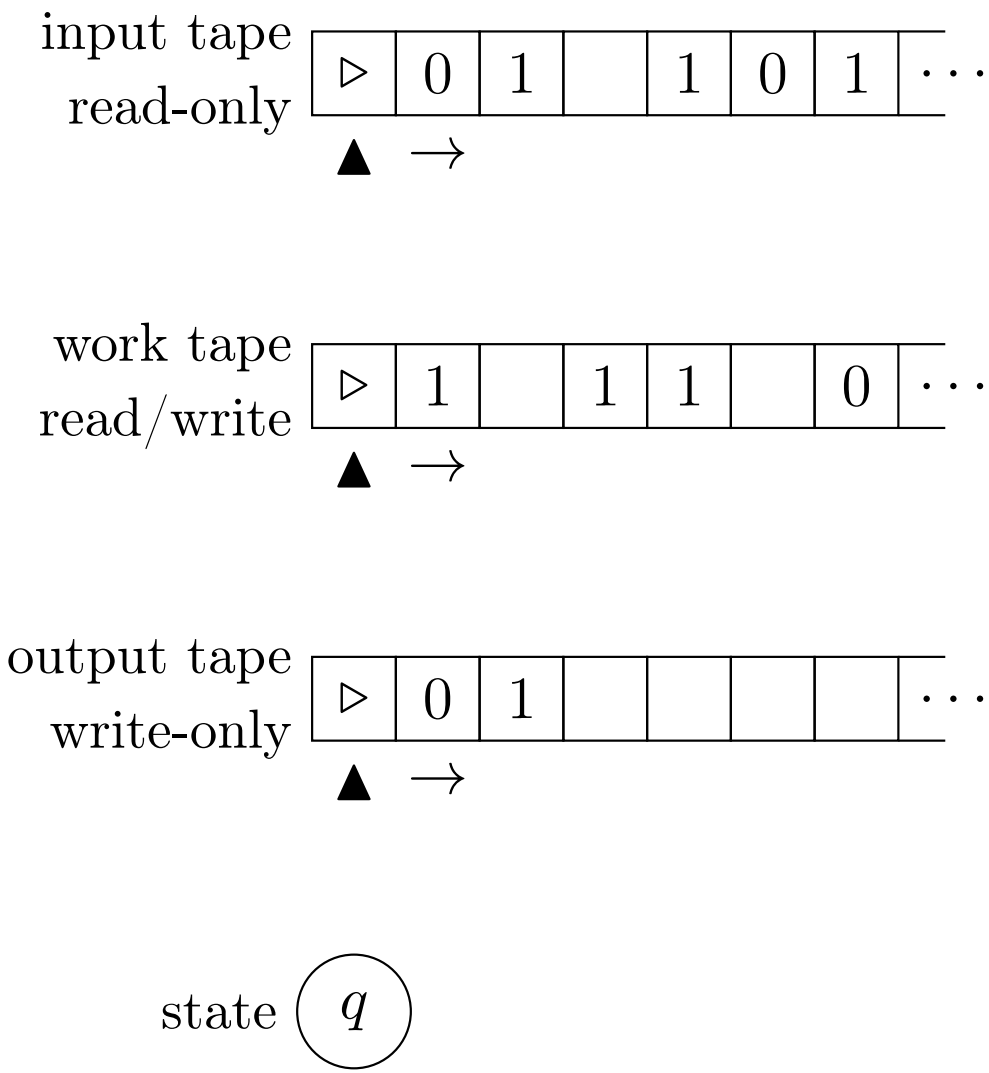
Turing Machine



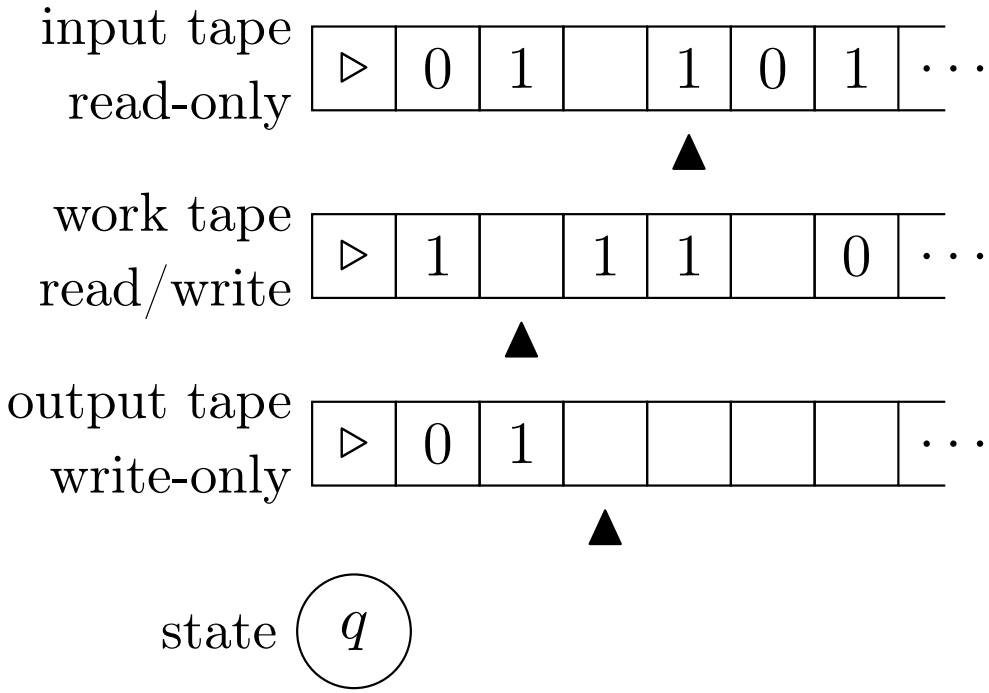
Turing Machine



Turing Machine



Turing Machine



If

state $==$ q

input tape $==$ 1

work tape $==$ $\square$

then

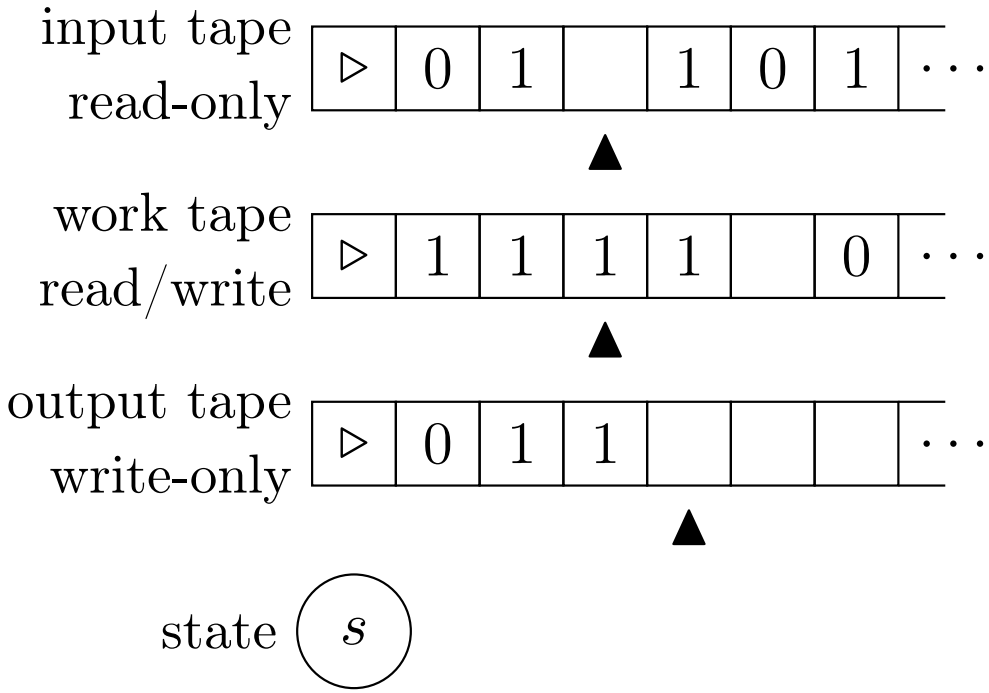
on input tape: move head L

on work tape: write 1 *and move head* R

on output tape: write 1 *(and move* R*)*

state $=$ s

Turing Machine



If

state $==$ q

input tape $==$ 1

work tape $==$ $\square$

then

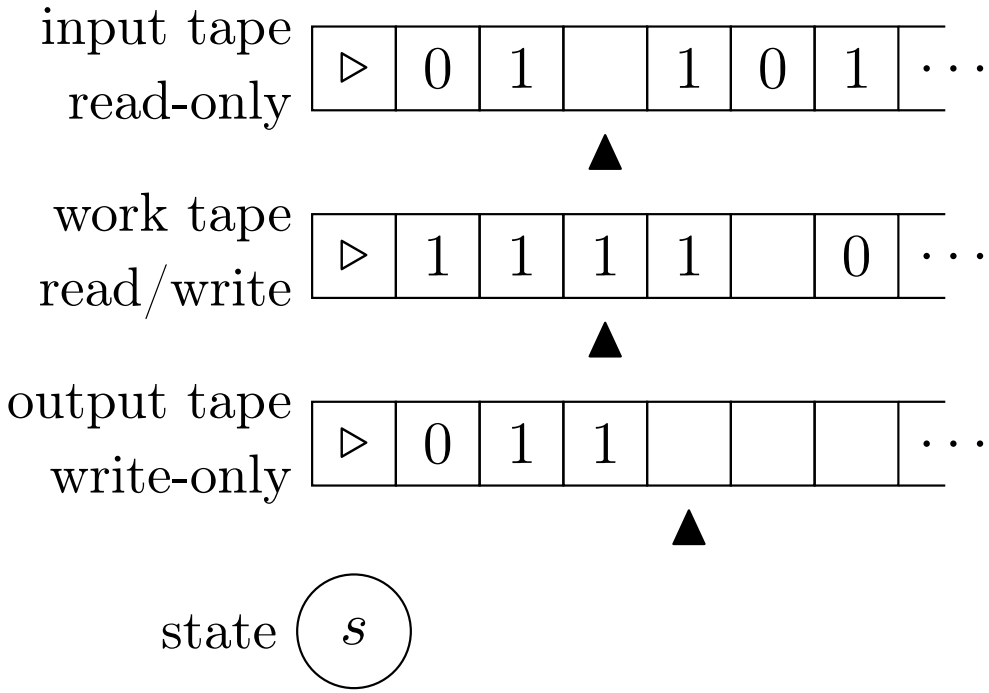
on input tape: move head L

on work tape: write 1 *and move head* R

on output tape: write 1 *(and move* R*)*

state $=$ s

Turing Machine



If

state $==$ s

input tape $==$ $\square$

work tape $==$ 1

then

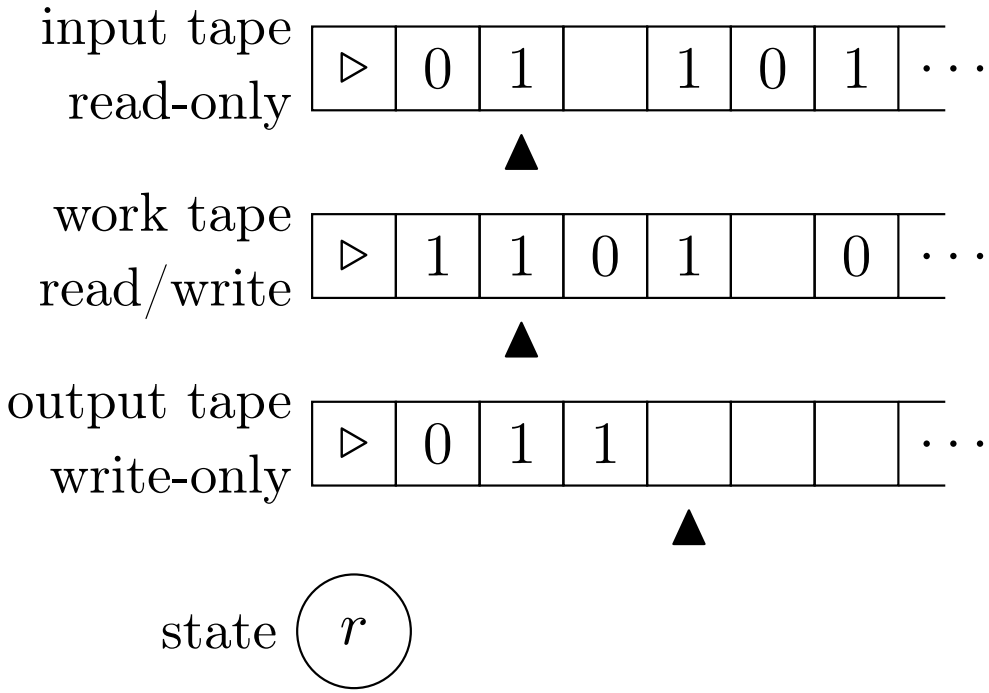
on input tape: move head L

on work tape: write 0 *and move head* L

on output tape: do nothing

state $=$ r

Turing Machine



If

state $==$ s

input tape $==$ $\square$

work tape $==$ 1

then

on input tape: move head L

on work tape: write 0 *and move head* L

on output tape: do nothing

state $=$ r

Turing Machine - formal definition

Definition

A **Turing machine** (or simply **machine**) is a triple (Σ, Q, δ), where

- Q: the finite set of **states**
  - $q_0 \in Q$: initial state
  - $q_f \in Q$: final state
- $\Sigma = \{0, 1, \triangleright, \square\}$ is the **alphabet** (it could have more symbols, but always finite)
- $\delta : Q \times \Sigma \times \Sigma \to \{L, R, S\} \times \Sigma \times \{L, R, S\} \times (\Sigma \cup \{S\}) \times Q$ is the **transition function**
  - L: left
  - R: right
  - S: stay

The transition function

INSTRUCTION

if $state == q \in Q$
$input\ tape == a \in \Sigma$
$work\ tape == b \in \Sigma$
then
on input tape: *move head* $L/R/S$
on work tape: *write* $c \in \Sigma$ *and move head* $L/R/S$
on output tape: *write* $d \in \Sigma$ *(and move to* R*) / do nothing*
$state = r \in Q$

$$\delta : Q \times \Sigma \times \Sigma \to \{L, R, S\} \times \Sigma \times \{L, R, S\} \times (\Sigma \cup \{S\}) \times Q$$

The transition function

INSTRUCTION

if $state == q \in Q$
$input\ tape == a \in \Sigma$
$work\ tape == b \in \Sigma$
then
on input tape: *move head* $L/R/S$
on work tape: *write* $c \in \Sigma$ *and move head* $L/R/S$
on output tape: *write* $d \in \Sigma$ *(and move to* R*) / do nothing*
$state = r \in Q$

$$\delta : Q \times \Sigma \times \Sigma \to \{L, R, S\} \times \Sigma \times \{L, R, S\} \times (\Sigma \cup \{S\}) \times Q$$

The transition function

INSTRUCTION

if $state == q \in Q$
$input\ tape == a \in \Sigma$
$work\ tape == b \in \Sigma$
then
on input tape: *move head* $L/R/S$
on work tape: *write* $c \in \Sigma$ *and move head* $L/R/S$
on output tape: *write* $d \in \Sigma$ *(and move to* R*) / do nothing*
$state = r \in Q$

$$\delta : Q \times \Sigma \times \Sigma \to \{L, R, S\} \times \Sigma \times \{L, R, S\} \times (\Sigma \cup \{S\}) \times Q$$

The transition function

INSTRUCTION

if $state == q \in Q$
$input\ tape == a \in \Sigma$
$work\ tape == b \in \Sigma$
then
on input tape: *move head* $L/R/S$
on work tape: *write* $c \in \Sigma$ *and move head* $L/R/S$
on output tape: *write* $d \in \Sigma$ *(and move to* R*) / do nothing*
$state = r \in Q$

$$\delta : Q \times \Sigma \times \Sigma \to \{L, R, S\} \times \Sigma \times \{L, R, S\} \times (\Sigma \cup \{S\}) \times Q$$

The transition function

INSTRUCTION

if $state == q \in Q$
$input\ tape == a \in \Sigma$
$work\ tape == b \in \Sigma$
then
on input tape: *move head* $L/R/S$
on work tape: *write* $c \in \Sigma$ *and move head* $L/R/S$
on output tape: *write* $d \in \Sigma$ *(and move to* R*) / do nothing*
$state = r \in Q$

$$\delta : Q \times \Sigma \times \Sigma \to \{L, R, S\} \times \Sigma \times \{L, R, S\} \times (\Sigma \cup \{S\}) \times Q$$

The transition function

INSTRUCTION

if $state == q \in Q$
$input\ tape == a \in \Sigma$
$work\ tape == b \in \Sigma$
then
on input tape: *move head* $L/R/S$
on work tape: *write* $c \in \Sigma$ *and move head* $L/R/S$
on output tape: *write* $d \in \Sigma$ *(and move to* R*) / do nothing*
$state = r \in Q$

$$\delta : Q \times \Sigma \times \Sigma \to \{L, R, S\} \times \Sigma \times \{L, R, S\} \times (\Sigma \cup \{S\}) \times Q$$

The transition function

INSTRUCTION

if $state == q \in Q$
$input\ tape == a \in \Sigma$
$work\ tape == b \in \Sigma$
then
on input tape: *move head* $L/R/S$
on work tape: *write* $c \in \Sigma$ *and move head* $L/R/S$
on output tape: *write* $d \in \Sigma$ *(and move to* R*)* / *do nothing*
$state = r \in Q$

$$\delta : Q \times \Sigma \times \Sigma \to \{L, R, S\} \times \Sigma \times \{L, R, S\} \times (\Sigma \cup \{S\}) \times Q$$

The transition function

INSTRUCTION

if $state == q \in Q$
$input\ tape == a \in \Sigma$
$work\ tape == b \in \Sigma$
then
on input tape: *move head* $L/R/S$
on work tape: *write* $c \in \Sigma$ *and move head* $L/R/S$
on output tape: *write* $d \in \Sigma$ *(and move to* R*)* / *do nothing*
$state = r \in Q$

$$\delta : Q \times \Sigma \times \Sigma \to \{L, R, S\} \times \Sigma \times \{L, R, S\} \times (\Sigma \cup \{S\}) \times Q$$

The transition function

INSTRUCTION

if $state == q \in Q$
$input\ tape == a \in \Sigma$
$work\ tape == b \in \Sigma$
then
on input tape: *move head* $L/R/S$
on work tape: *write* $c \in \Sigma$ *and move head* $L/R/S$
on output tape: *write* $d \in \Sigma$ *(and move to* R*) / do nothing*
$state = r \in Q$

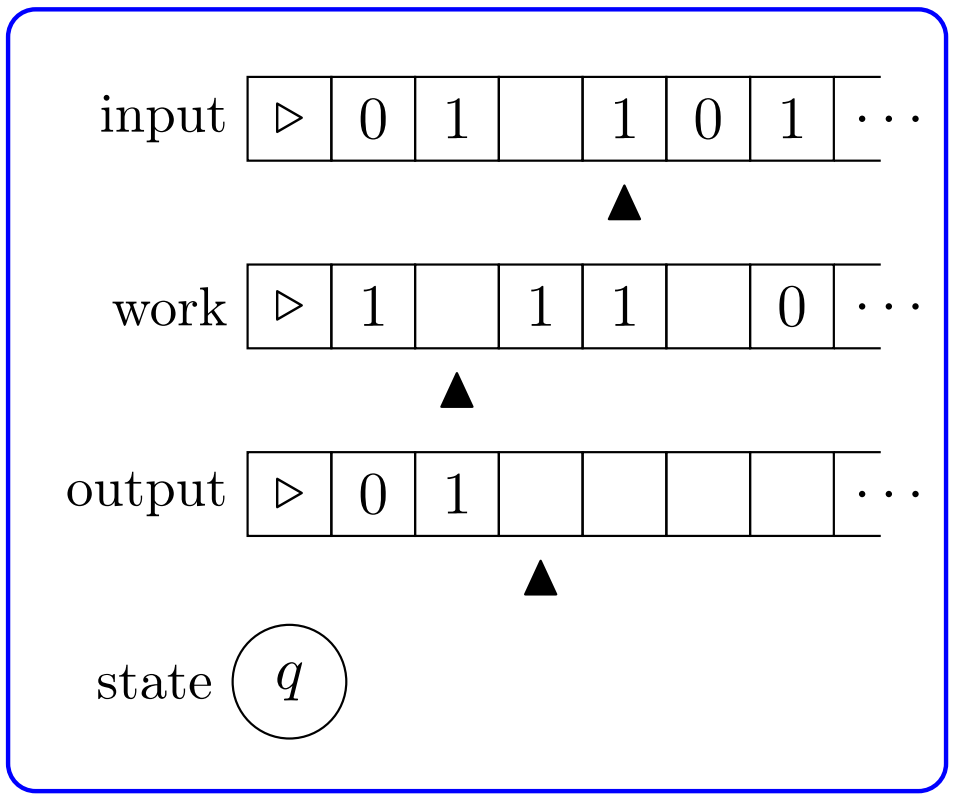
$$\delta : Q \times \Sigma \times \Sigma \to \{L, R, S\} \times \Sigma \times \{L, R, S\} \times (\Sigma \cup \{S\}) \times Q$$

There are restrictions on δ (prevent the head from going past the beginning of the tapes or erasing the $\triangleright$):

Configurations and computations

Configuration

A **configuration** is the snapshot of the machine at a given moment, for example

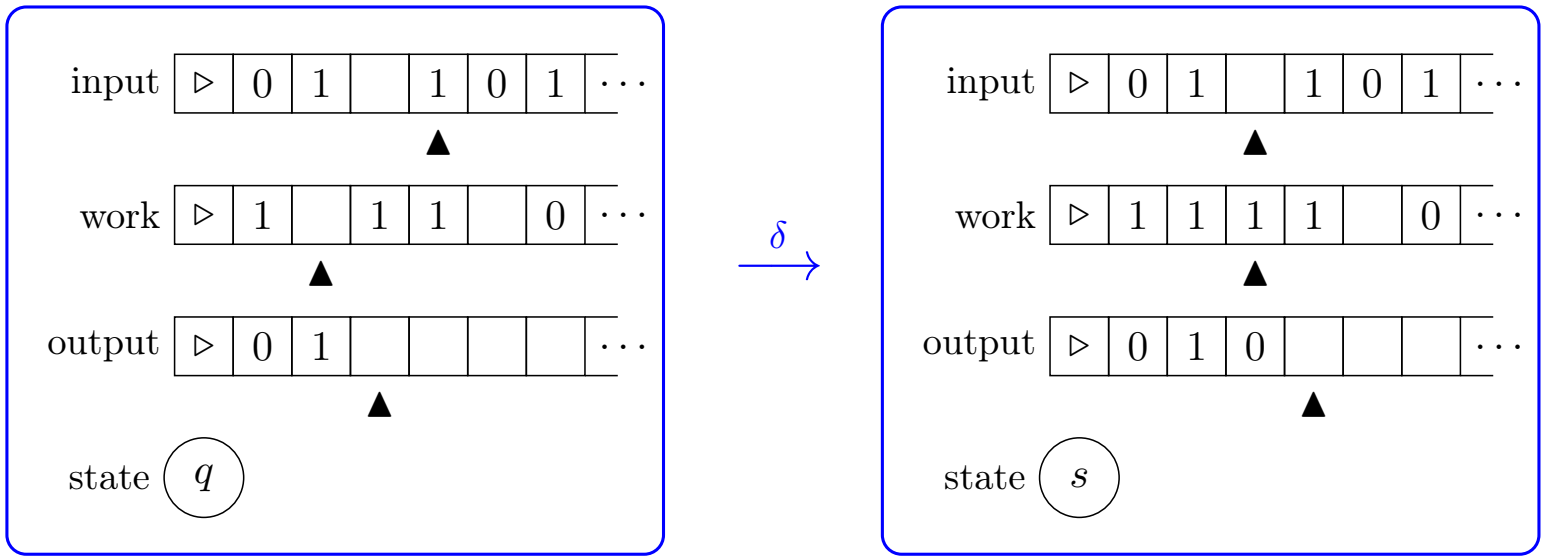


It has all the information about:

- contents of all tapes
- position of each head
- state

Evolution of a computation

The transition function δ makes the computation **evolve in one step** from a given configuration C:

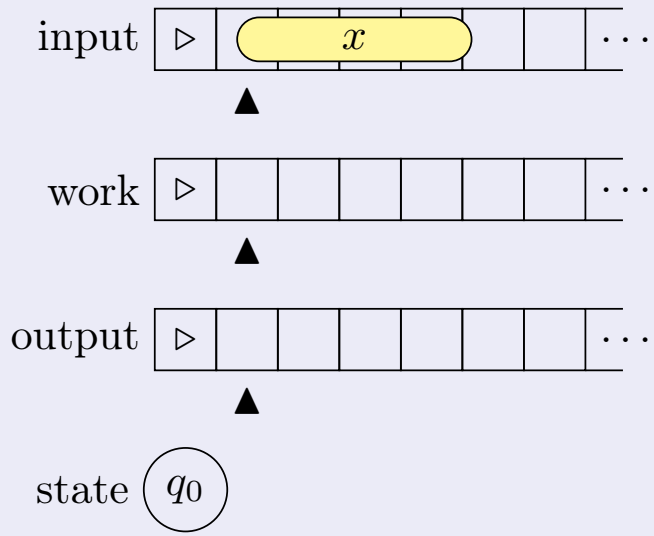


$$\delta(q, 1, \square) = (L, 1, R, 0, s)$$

If
- *state* $== q$
- *input tape* $== 1$
- *work tape* $== \square$

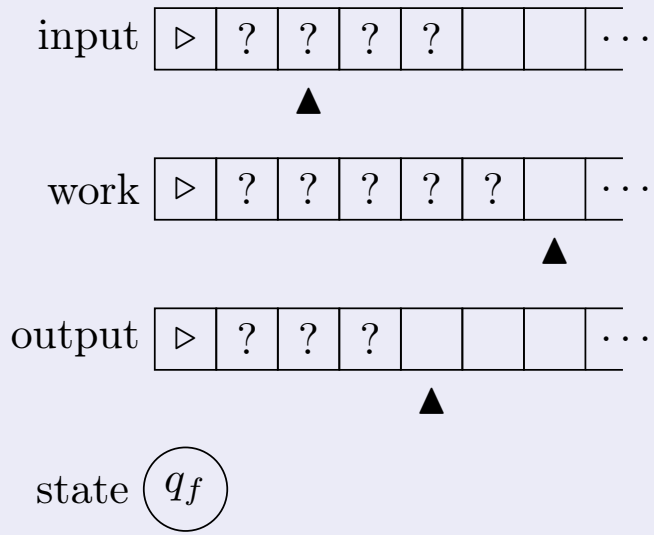
then
- *on input tape: move head* L
- *on work tape: write* 1 *and move head* R
- *on output tape: write* 1 *(and move* R*)*
- *state* $= s$

Initial configuration of M for $x \in \{0,1\}^*$



- input tape contains x and the rest blank
- work tape blank
- output tape blank
- all heads on the second cell
- the state is q_0 (initial state)

Final configuration of M



- the state is q_f (it freezes, it halts)
- anything may be written on work or output tapes
- any head position
- we say that a machine in a final configuration **halts** or **stops**

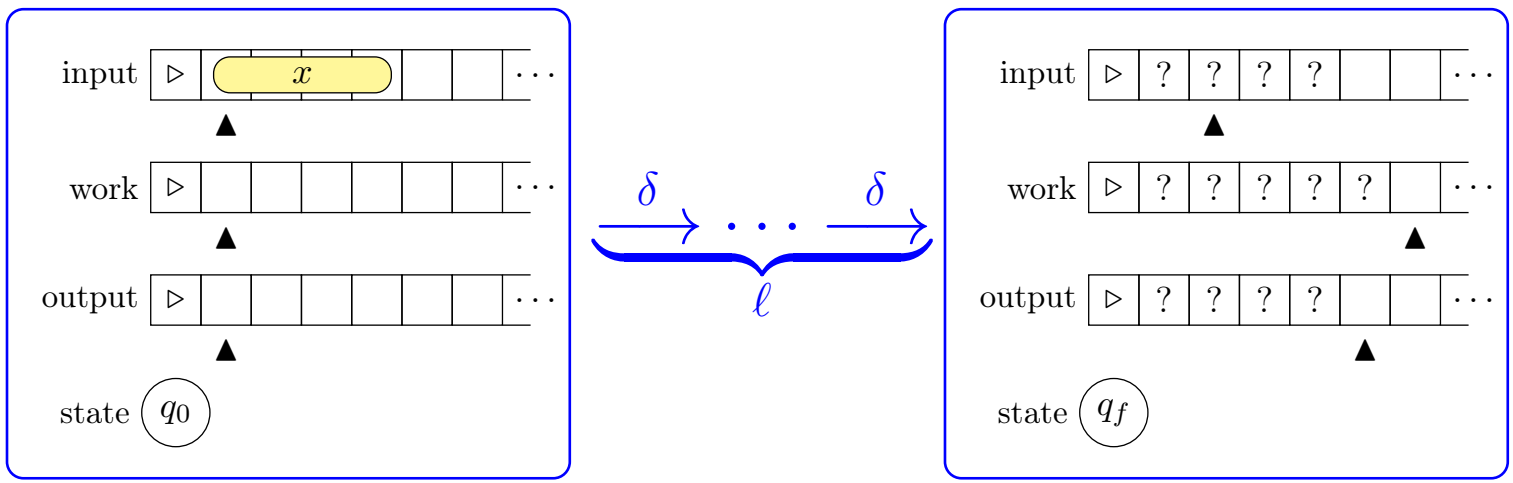
Computation

Definition

A **computation** of $M = (\Sigma, Q, \delta)$ starting from $x \in \{0,1\}^*$ is a sequence $C_0, \ldots, C_\ell$ of configurations such that

- C_0 is initial for x
- C_ℓ is final and
- C_{i+1} is the evolution of C_i in one step.

We call ℓ the **length** of the computation. We say that M with input x **halts** if there is a computation of M starting from x.

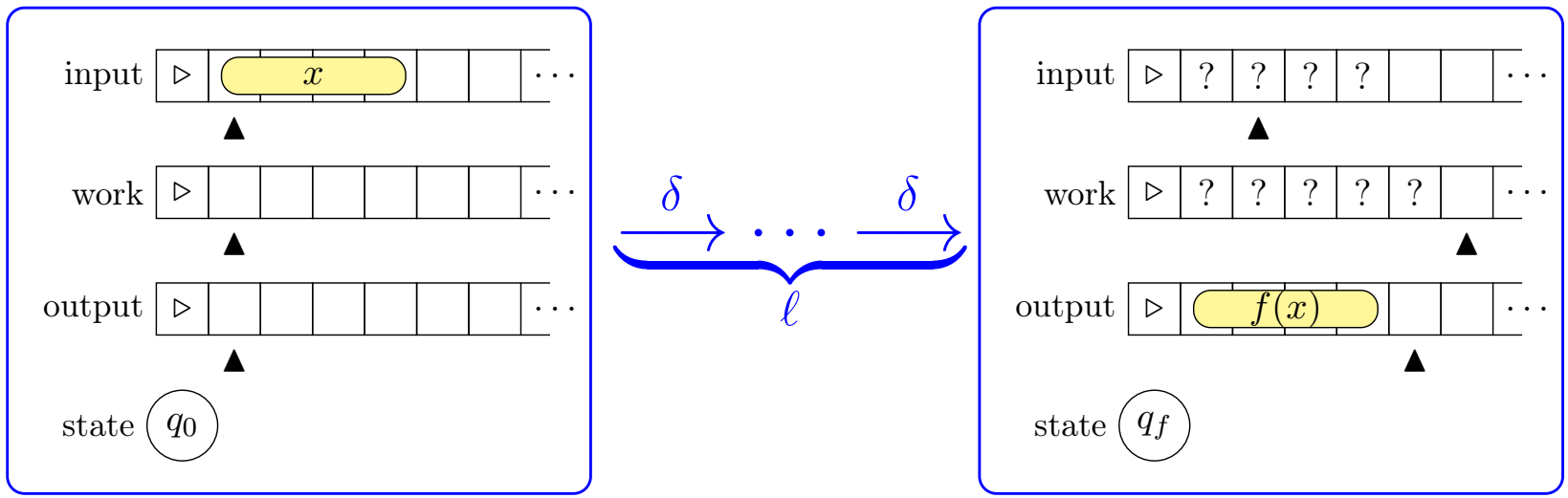


Computable functions

Computable functions

Definition

Let $f : \{0,1\}^* \to \{0,1\}^*$ and $M = (\Sigma, Q, \delta)$ be a machine. We say that M **computes** f if for every $x \in \{0,1\}^*$ there exists a computation $C_0, \ldots, C_\ell$ of M starting from x and in C_ℓ the output tape contains $f(x)$ followed by blanks.
In this case, we write $\boldsymbol{M(x)}$ for $f(x)$. We say that f is **computable** if there exists a machine that computes it.



'Computability' is an absolute notion

With this concept one has for the first time succeeded in giving an absolute definition of an interesting epistemological notion, i.e. one not depending on the formalism chosen. In all other cases treated previously, such as demonstrability or definability, one has been able to define them only relative to a given language, and for each individual language it is clear that the one thus obtained is not the one looked for. For the concept of computability however, although it is merely a special kind of demonstrability or definability, the situation is different.

Gödel, 1946

Church's Thesis

There are many models of computation, i.e. many ways to formalize the notion of an algorithm.

- Turing machines,
- λ-calculus,
- recursive functions,
- Post systems,
- register machines,
- programming languages like Python, C, Java, etc.[2]

[2] But recall the provisos about time and memory

Church's Thesis

There are many models of computation, i.e. many ways to formalize the notion of an algorithm.

- Turing machines,
- λ-calculus,
- recursive functions,
- Post systems,
- register machines,
- programming languages like Python, C, Java, etc.[2]

Church's Thesis.

Any algorithm for computing on strings (or natural numbers) can be carried out by a Turing machine.

[2] But recall the provisos about time and memory

Why Turing machines?

- All these models turn out to be equivalent in terms of computability.
- Turing machines are not the most practical model of computation. It's like programming in assembly language.
  - In practice, we use pseudocode to describe algorithms, and we assume that they can be implemented in a Turing machine.

Why Turing machines?

- All these models turn out to be equivalent in terms of computability.
- Turing machines are not the most practical model of computation. It's like programming in assembly language.
  - In practice, we use pseudocode to describe algorithms, and we assume that they can be implemented in a Turing machine.
- For Computability theory, it does not matter which model of computation we use, as they are all equivalent in terms of computability.

Why Turing machines?

- All these models turn out to be equivalent in terms of computability.
- Turing machines are not the most practical model of computation. It's like programming in assembly language.
  - In practice, we use pseudocode to describe algorithms, and we assume that they can be implemented in a Turing machine.
- For Computability theory, it does not matter which model of computation we use, as they are all equivalent in terms of computability.
- For Complexity theory, the choice of the model of computation does matter and Turing machines gives us a precise model to analyze time and space.

All functions are computable?

- Which functions $\{0,1\}^* \to \{0,1\}^*$ are computable?

All functions are computable?

- Which functions $\{0,1\}^* \to \{0,1\}^*$ are computable?
  - Probably all functions that you can think of! (unless you've taken a course in computability theory).

All functions are computable?

- Which functions $\{0,1\}^* \to \{0,1\}^*$ are computable?
  - Probably all functions that you can think of! (unless you've taken a course in computability theory).
- Are all functions $\{0,1\}^* \to \{0,1\}^*$ computable?

All functions are computable?

- Which functions $\{0,1\}^* \to \{0,1\}^*$ are computable?
  - Probably all functions that you can think of! (unless you've taken a course in computability theory).
- Are all functions $\{0,1\}^* \to \{0,1\}^*$ computable?
  - No, because there are only countably many Turing machines and hence countably many computable functions
  - However, there are uncountably many functions $\{0,1\}^* \to \{0,1\}^*$.

All functions are computable?

- Which functions $\{0,1\}^* \to \{0,1\}^*$ are computable?
  - Probably all functions that you can think of! (unless you've taken a course in computability theory).
- Are all functions $\{0,1\}^* \to \{0,1\}^*$ computable?
  - No, because there are only countably many Turing machines and hence countably many computable functions
  - However, there are uncountably many functions $\{0,1\}^* \to \{0,1\}^*$.
- So there are functions $\{0,1\}^* \to \{0,1\}^*$ that are not computable. In fact, uncountably many of them!

All functions are computable?

- Which functions $\{0,1\}^* \to \{0,1\}^*$ are computable?
  - Probably all functions that you can think of! (unless you've taken a course in computability theory).
- Are all functions $\{0,1\}^* \to \{0,1\}^*$ computable?
  - No, because there are only countably many Turing machines and hence countably many computable functions
  - However, there are uncountably many functions $\{0,1\}^* \to \{0,1\}^*$.
- So there are functions $\{0,1\}^* \to \{0,1\}^*$ that are not computable. In fact, uncountably many of them!
- But what concrete example do we have?

Coding Turing machines as words

- Turing machines are finite objects, so they can be effectively encoded as words in $\{0,1\}^*$.
  - If you prefer, think of Python programs instead of Turing machines; any Python program can be encoded as a word in $\{0,1\}^*$ in ASCII.

Coding Turing machines as words

- Turing machines are finite objects, so they can be effectively encoded as words in $\{0,1\}^*$.
  - If you prefer, think of Python programs instead of Turing machines; any Python program can be encoded as a word in $\{0,1\}^*$ in ASCII.
- $\langle M \rangle$ is a word in $\{0,1\}^*$ that encodes the machine M.
- So if $x \in \{0,1\}^*$, we can talk about 'the machine x' to refer to the machine encoded by x, namely the only machine M such that $\langle M \rangle = x$.

Coding Turing machines as words

- Turing machines are finite objects, so they can be effectively encoded as words in $\{0,1\}^*$.
  - If you prefer, think of Python programs instead of Turing machines; any Python program can be encoded as a word in $\{0,1\}^*$ in ASCII.
- $\langle M \rangle$ is a word in $\{0,1\}^*$ that encodes the machine M.
- So if $x \in \{0,1\}^*$, we can talk about 'the machine x' to refer to the machine encoded by x, namely the only machine M such that $\langle M \rangle = x$.
- We fix a bijection between the set of all Turing machines and the set of $\{0,1\}^*$.
  - Don't worry about the word representing an invalid machine; we can just say that it is a machine that never halts.

Coding Turing machines as words

- Turing machines are finite objects, so they can be effectively encoded as words in $\{0,1\}^*$.
  - If you prefer, think of Python programs instead of Turing machines; any Python program can be encoded as a word in $\{0,1\}^*$ in ASCII.
- $\langle M \rangle$ is a word in $\{0,1\}^*$ that encodes the machine M.
- So if $x \in \{0,1\}^*$, we can talk about 'the machine x' to refer to the machine encoded by x, namely the only machine M such that $\langle M \rangle = x$.
- We fix a bijection between the set of all Turing machines and the set of $\{0,1\}^*$.
  - Don't worry about the word representing an invalid machine; we can just say that it is a machine that never halts.
- So:
  - machines are codified as strings
  - machines take a sting as input
  - so a machine can interpret its input as a machine

The Halting Problem

We define $\mathsf{HALT} : \{0,1\}^* \to \{0,1\}$ as

$$\mathsf{HALT}(x) = \begin{cases} 1 & \text{if machine } x \text{ with input } x \text{ halts} \\ 0 & \text{if not} \end{cases}$$

The Halting Problem

We define $\mathsf{HALT} : \{0,1\}^* \to \{0,1\}$ as

$$\mathsf{HALT}(x) = \begin{cases} 1 & \text{if machine } x \text{ with input } x \text{ halts} \\ 0 & \text{if not} \end{cases}$$

Suppose a machine M that computes the following:

input x (but irrelevant in this case)
$y := 4$
While there are primes $p_1, p_2 \leq y$ such that $p_1 + p_2 = y$
    $y := y + 2$

The Halting Problem

We define $\mathsf{HALT} : \{0,1\}^* \to \{0,1\}$ as

$$\mathsf{HALT}(x) = \begin{cases} 1 & \text{if machine } x \text{ with input } x \text{ halts} \\ 0 & \text{if not} \end{cases}$$

Suppose a machine M that computes the following:

input x (but irrelevant in this case)
$y := 4$
While there are primes $p_1, p_2 \leq y$ such that $p_1 + p_2 = y$
    $y := y + 2$

Exercise

Let $e = \langle M \rangle$. What is $\mathsf{HALT}(e)$?

The Halting Problem

We define $\mathsf{HALT} : \{0,1\}^* \to \{0,1\}$ as

$$\mathsf{HALT}(x) = \begin{cases} 1 & \text{if machine } x \text{ with input } x \text{ halts} \\ 0 & \text{if not} \end{cases}$$

Suppose a machine M that computes the following:

input x (but irrelevant in this case)
$y := 4$
While there are primes $p_1, p_2 \leq y$ such that $p_1 + p_2 = y$
    $y := y + 2$

Exercise

Let $e = \langle M \rangle$. What is $\mathsf{HALT}(e)$?

If you find the answer:

- write it in LaTex
- send it to me by email and wait for my instructions
- don't tell anyone else

The Halting Problem

We define $\mathsf{HALT} : \{0,1\}^* \to \{0,1\}$ as

$$\mathsf{HALT}(x) = \begin{cases} 1 & \text{if machine } x \text{ with input } x \text{ halts} \\ 0 & \text{if not} \end{cases}$$

Suppose a machine M that computes the following:

input x (but irrelevant in this case)
$y := 4$
While there are primes $p_1, p_2 \leq y$ such that $p_1 + p_2 = y$
    $y := y + 2$

Exercise

Let $e = \langle M \rangle$. What is $\mathsf{HALT}(e)$?

If you find the answer:

- write it in LaTex
- send it to me by email and wait for my instructions
- don't tell anyone else

$\mathsf{HALT}(e) = 1$ iff the Goldbach Conjecture (1749) is false

An unsolvable problem

Theorem (Turing 1936)

HALT is not computable.

An unsolvable problem

Theorem (Turing 1936)

HALT is not computable.

Proof.

Suppose HALT is computable. Define a machine M doing this:

> *input* x
> *if* $\mathsf{HALT}(x) = 1$ *then loop forever*
> *else terminate (output is irrelevant)*

$$M(x) \text{ halts} \iff \mathsf{HALT}(x) = 0.$$

Let $e = \langle M \rangle$.

$$\begin{aligned} M(e) \text{ halts} \quad & \text{iff } \mathsf{HALT}(e) = 0 \\ & \text{iff machine } e \text{ with input } e \text{ does not halt} \\ & \text{iff } M(e) \text{ does not halt} \end{aligned}$$

Contradiction. □

Decision problems

We restrict ourselves to **decision problems**, which are formalized as Boolean functions:

$$f : \{0,1\}^* \to \{0,1\}$$

Each Boolean function f represents the language (set of words in $\{0,1\}^*$)

$$\mathcal{L}(f) = \{x \mid f(x) = 1\} \subseteq \{0,1\}^*$$

And conversely: any language $\mathcal{L} \subseteq \{0,1\}^*$ can be represented by the Boolean function

$$\chi_{\mathcal{L}}(x) = \begin{cases} 1 & \text{if } x \in \mathcal{L} \\ 0 & \text{otherwise} \end{cases}$$

decision problems = Boolean functions = languages

Examples of computable decision problems

Example

The problem of deciding whether a number x is even

$$f : \{0,1\}^* \to \{0,1\}$$

$$f(\langle x \rangle) = \begin{cases} 1 & \text{if } x \text{ is even} \\ 0 & \text{if not} \end{cases}$$

or equivalently with

$$\mathcal{L} = \{\langle x \rangle \mid x \text{ is even}\}$$

Examples of computable decision problems

Example

The problem of deciding whether a number x is even

$$f : \{0,1\}^* \to \{0,1\}$$

$$f(\langle x \rangle) = \begin{cases} 1 & \text{if } x \text{ is even} \\ 0 & \text{if not} \end{cases}$$

or equivalently with

$$\mathcal{L} = \{\langle x \rangle \mid x \text{ is even}\}$$

Example

The problem of deciding whether a graph G has a k-coloring

$$f : \{0,1\}^* \to \{0,1\}$$

$$f(\langle G, k \rangle) = \begin{cases} 1 & \text{if } G \text{ has a } k\text{-coloring} \\ 0 & \text{if not} \end{cases}$$

or equivalently with

$$\mathcal{L} = \{\langle G, k \rangle \mid G \text{ has a } k\text{-coloring}\}$$

More examples of computable problems (connected to logic)

Boolean satisfiability

Deciding if a propositional formula is satisfiable.

First-order model checking

Deciding if a first-order formula is true in a given finite structure.

Presburger arithmetic

Deciding if a first-order formula is true in the standard model of arithmetic with addition $\langle \mathbb{N}, +, 0, 1 \rangle$ (Presburger).

Real closed fields

Deciding if a first-order formula is true in the standard model of real numbers $\langle \mathbb{R}, +, \cdot, 0, 1, < \rangle$ (Tarski–Seidenberg).

Examples of non-computable problems (connected to logic)

Entscheidungsproblem

Deciding if a first-order formula is universally valid. Or its dual problem: deciding if a first-order formula is satisfiable (posed by Hilbert and answered by Church and Turing).

Examples of non-computable problems (connected to logic)

Entscheidungsproblem

Deciding if a first-order formula is universally valid. Or its dual problem: deciding if a first-order formula is satisfiable (posed by Hilbert and answered by Church and Turing).

Finite satisfiability

Deciding if a first-order formula has a finite model (Trakhtenbrot).

Examples of non-computable problems (connected to logic)

Entscheidungsproblem

Deciding if a first-order formula is universally valid. Or its dual problem: deciding if a first-order formula is satisfiable (posed by Hilbert and answered by Church and Turing).

Finite satisfiability

Deciding if a first-order formula has a finite model (Trakhtenbrot).

Arithmetic truth

Deciding if a first-order formula is true in the standard model of arithmetic $\langle \mathbb{N}, +, \cdot, 0, 1 \rangle$ (Tarski)

Examples of non-computable problems (connected to logic)

Entscheidungsproblem

Deciding if a first-order formula is universally valid. Or its dual problem: deciding if a first-order formula is satisfiable (posed by Hilbert and answered by Church and Turing).

Finite satisfiability

Deciding if a first-order formula has a finite model (Trakhtenbrot).

Arithmetic truth

Deciding if a first-order formula is true in the standard model of arithmetic $\langle \mathbb{N}, +, \cdot, 0, 1 \rangle$ (Tarski)

Hilbert's tenth problem

Deciding if a Diophantine equation has an integer solution (Matiyasevich–Robinson–Davis–Putnam)

Part II: Complexity

- Which problems can be solved efficiently?
- What does 'efficiently' mean?

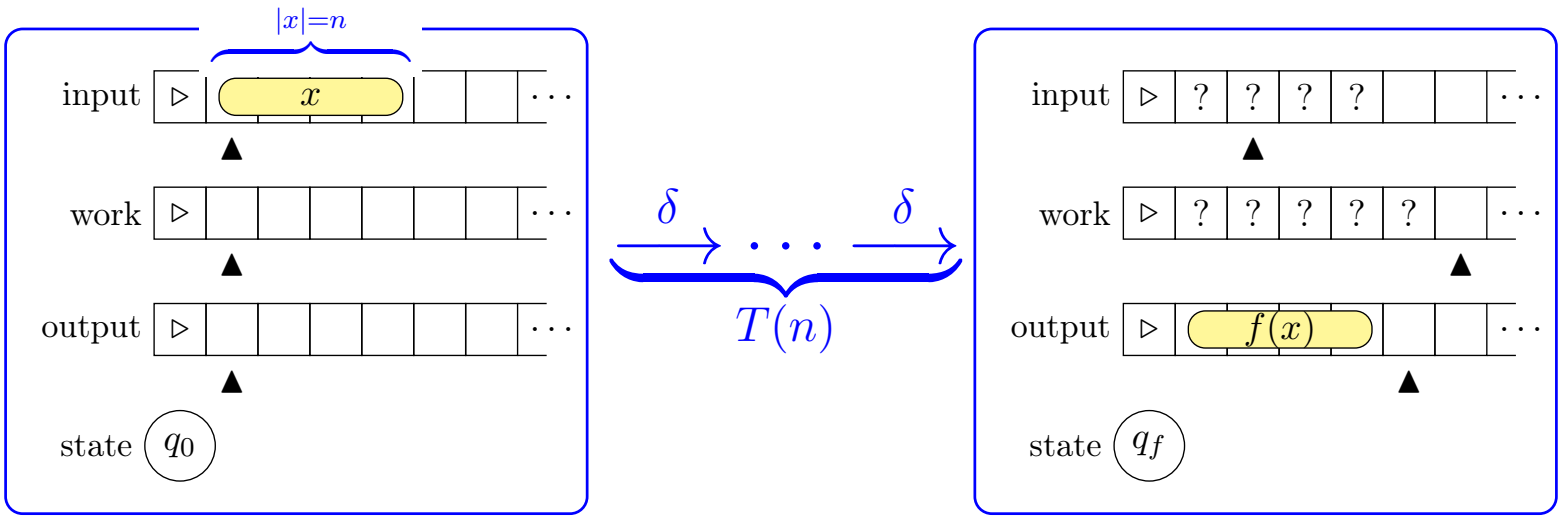
Computation time

Computation time

Definition

Let $f : \{0,1\}^* \to \{0,1\}^*$, $T : \mathbb{N} \to \mathbb{N}$ and $M = (\Sigma, Q, \delta)$ be a Turing machine.

- $\boldsymbol{M}$ **runs in time** $\boldsymbol{T(n)}$ if for every $x \in \{0,1\}^*$ there is a computation of M starting from x of length $\leq T(|x|)$ (in particular, M never loops forever).
- $\boldsymbol{M}$ **computes** $\boldsymbol{f}$ **in time** $\boldsymbol{T(n)}$ if M runs in time $T(n)$ and M computes f.



Polynomial time computation

Definition

M **runs in polynomial time** if there exists a polynomial p such that M runs in time p.

The classes **P** and **NP**

Languages decided by Turing machines

Let $\mathcal{L} \subseteq \{0,1\}^*$ be a language (or problem).

Definition

A Turing machine M **decides** $\mathcal{L}$ [in time $T(n)$] if M computes $\chi_{\mathcal{L}}$ [in time $T(n)$].
We say that M

- **accepts** x when $M(x) = 1$;
- **rejects** x when $M(x) = 0$.

Intuition of **P** and **NP**

- **P** is the class of problems that can be solved in polynomial time by a Turing machine.
  - These are the problems considered feasible or efficiently solvable.

Intuition of **P** and **NP**

- **P** is the class of problems that can be solved in polynomial time by a Turing machine.
  - These are the problems considered feasible or efficiently solvable.
- **NP** is the class of problems for which a solution can be verified in polynomial time by a Turing machine.
  - So the solution can be efficiently verified.

Intuition of **P** and **NP**

- **P** is the class of problems that can be solved in polynomial time by a Turing machine.
  - These are the problems considered feasible or efficiently solvable.
- **NP** is the class of problems for which a solution can be verified in polynomial time by a Turing machine.
  - So the solution can be efficiently verified.
- $\mathbf{P} \subseteq \mathbf{NP}$ because if a problem can be solved in polynomial time, then a solution can be verified in polynomial time (just solve it and check that the solution is correct).
- It is an open problem whether $\mathbf{P} = \mathbf{NP}$ or $\mathbf{P} \neq \mathbf{NP}$.
- Most computer scientists believe that $\mathbf{P} \neq \mathbf{NP}$.

The formal definition of **P** and **NP**

Complexity class: P

P is the class of languages $\mathcal{L}$ such that there exists a Turing machine M running in polynomial time such that M decides $\mathcal{L}$, i.e. for every x:

$$x \in \mathcal{L} \qquad \text{iff} \qquad M(x) = 1$$

The formal definition of **P** and **NP**

Complexity class: **P**

P is the class of languages $\mathcal{L}$ such that there exists a Turing machine M running in polynomial time such that M decides $\mathcal{L}$, i.e. for every x:

$$x \in \mathcal{L} \qquad \text{iff} \qquad M(x) = 1$$

Complexity class: **NP**

NP is the class of languages $\mathcal{L}$ such that there exists a polynomial p and a Turing machine M running in polynomial time such that for every x:

$$x \in \mathcal{L} \qquad \text{iff} \qquad \text{there exists } u \in \{0,1\}^{p(|x|)} \text{ s.t. } M(xu) = 1$$

M is called the **verifier for** $\mathcal{L}$; u is called a **certificate for** x.

Example of a problem in **NP**

A set X of vertices of a graph $G = (V, E)$ is **independent** if there are no $u, v \in X$ such that $(u, v) \in E$.

Problem: Independent set

$$\mathsf{INDSET} = \{\langle G, k\rangle \mid \begin{array}{l} G \text{ has an independent set of} \\ \geq k \text{ nodes} \end{array} \}$$

Example of a problem in **NP**

A set X of vertices of a graph $G = (V, E)$ is **independent** if there are no $u, v \in X$ such that $(u, v) \in E$.

Problem: Independent set

$$\mathsf{INDSET} = \{\langle G, k\rangle \mid \begin{array}{l} G \text{ has an independent set of} \\ \geq k \text{ nodes} \end{array}\}$$

- we assume $V = \{0, 1, \ldots, |V| - 1\}$
- certificate: list of k distinct vertices of V that form an independent set
- M on input $x = \langle G, k\rangle u$ checks that the certificate u is an independent set of G and has at least k vertices
- M runs in polynomial time.
- $x \in \mathsf{INDSET}$ iff there exists $u \in \{0, 1\}^{p(|x|)}$ such that $M(xu) = 1$.

Polynomial reducibility

Polynomial reducibility

Let $\mathcal{L}, \mathcal{L}' \subseteq \{0,1\}^*$ be languages (or problems).

Definition

$\mathcal{L}$ is **(Karp) polynomially reducible** to $\mathcal{L}'$, written $\mathcal{L} \leq_{\mathrm{p}} \mathcal{L}'$, if there exists a polynomial-time computable function

$$f : \{0,1\}^* \to \{0,1\}^*$$

such that for all $x \in \{0,1\}^*$

$$x \in \mathcal{L} \qquad \text{iff} \qquad f(x) \in \mathcal{L}'.$$

f is called a **polynomial reduction** from $\mathcal{L}$ to $\mathcal{L}'$.

Polynomial reducibility

Let $\mathcal{L}, \mathcal{L}' \subseteq \{0,1\}^*$ be languages (or problems).

Definition

$\mathcal{L}$ is **(Karp) polynomially reducible** to $\mathcal{L}'$, written $\mathcal{L} \leq_{\mathrm{p}} \mathcal{L}'$, if there exists a polynomial-time computable function

$$f : \{0,1\}^* \to \{0,1\}^*$$

such that for all $x \in \{0,1\}^*$

$$x \in \mathcal{L} \qquad \text{iff} \qquad f(x) \in \mathcal{L}'.$$

f is called a **polynomial reduction** from $\mathcal{L}$ to $\mathcal{L}'$.

Complexity class: NP-hard, NP-complete

$\mathcal{L}$ is **NP-hard** if $\mathcal{L}' \leq_{\mathrm{p}} \mathcal{L}$ for every $\mathcal{L}' \in \mathbf{NP}$.
$\mathcal{L}$ is **NP-complete** if $\mathcal{L} \in \mathbf{NP}$ and $\mathcal{L}$ is **NP-hard**.

Polynomial reducibility

Intuition

- $\mathcal{L} \leq_{\mathrm{p}} \mathcal{L}'$ means that $\mathcal{L}$ is not harder than $\mathcal{L}'$
- **NP-hard** problems are harder than any problem in **NP**
- **NP-complete** problems are the hardest problems in **NP**

Polynomial reducibility

Intuition

- $\mathcal{L} \leq_{\mathrm{p}} \mathcal{L}'$ means that $\mathcal{L}$ is not harder than $\mathcal{L}'$
- **NP-hard** problems are harder than any problem in **NP**
- **NP-complete** problems are the hardest problems in **NP**

Proposition

The relation $\leq_{\mathrm{p}}$ is transitive.

Polynomial reducibility

Intuition

- $\mathcal{L} \leq_{\mathrm{p}} \mathcal{L}'$ means that $\mathcal{L}$ is not harder than $\mathcal{L}'$
- **NP-hard** problems are harder than any problem in **NP**
- **NP-complete** problems are the hardest problems in **NP**

Proposition

The relation $\leq_{\mathrm{p}}$ is transitive.

Proposition

If $\mathcal{L} \leq_{\mathrm{p}} \mathcal{L}'$ and $\mathcal{L}' \in \mathbf{P}$, then $\mathcal{L} \in \mathbf{P}$.

Polynomial reducibility

Intuition

- $\mathcal{L} \leq_{\mathrm{p}} \mathcal{L}'$ means that $\mathcal{L}$ is not harder than $\mathcal{L}'$
- **NP-hard** problems are harder than any problem in **NP**
- **NP-complete** problems are the hardest problems in **NP**

Proposition

The relation $\leq_{\mathrm{p}}$ is transitive.

Proposition

If $\mathcal{L} \leq_{\mathrm{p}} \mathcal{L}'$ and $\mathcal{L}' \in \mathbf{P}$, then $\mathcal{L} \in \mathbf{P}$.

Theorem

If $\mathbf{NP\text{-}hard} \cap \mathbf{P} \neq \emptyset$, then $\mathbf{P} = \mathbf{NP}$.

Polynomial reducibility

Intuition

- $\mathcal{L} \leq_{\mathrm{p}} \mathcal{L}'$ means that $\mathcal{L}$ is not harder than $\mathcal{L}'$
- **NP-hard** problems are harder than any problem in **NP**
- **NP-complete** problems are the hardest problems in **NP**

Proposition

The relation $\leq_{\mathrm{p}}$ is transitive.

Proposition

If $\mathcal{L} \leq_{\mathrm{p}} \mathcal{L}'$ and $\mathcal{L}' \in \mathbf{P}$, then $\mathcal{L} \in \mathbf{P}$.

Theorem

If $\mathbf{NP\text{-}hard} \cap \mathbf{P} \neq \emptyset$, then $\mathbf{P} = \mathbf{NP}$.

Theorem

If $\mathcal{L} \in \mathbf{NP\text{-}complete}$, then $\mathcal{L} \in \mathbf{P}$ iff $\mathbf{P} = \mathbf{NP}$.

The problem SAT

A Boolean formula is in **conjunctive normal form (CNF)** if it is of the form

$$\underbrace{(a_{11} \vee \cdots \vee a_{1n_1})}_{\text{clause}} \wedge \underbrace{(a_{21} \vee \cdots \vee a_{2n_2})}_{\text{clause}} \wedge \cdots \wedge \underbrace{(a_{m1} \vee \cdots \vee a_{mn_m})}_{\text{clause}}$$

where a_{ij} is a **literal**, i.e. of the form p or $\neg p$ for some proposition p.

Example of a CNF formula

$$\varphi = (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4)$$

We represent a formula $\varphi \in \text{CNF}$ by a word $\langle \varphi \rangle$ in $\{0,1\}^*$.

The problem SAT

A Boolean formula is in **conjunctive normal form (CNF)** if it is of the form

$$\underbrace{(a_{11} \vee \cdots \vee a_{1n_1})}_{\text{clause}} \wedge \underbrace{(a_{21} \vee \cdots \vee a_{2n_2})}_{\text{clause}} \wedge \cdots \wedge \underbrace{(a_{m1} \vee \cdots \vee a_{mn_m})}_{\text{clause}}$$

where a_{ij} is a **literal**, i.e. of the form p or $\neg p$ for some proposition p.

Example of a CNF formula

$$\varphi = (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4)$$

We represent a formula $\varphi \in \text{CNF}$ by a word $\langle \varphi \rangle$ in $\{0,1\}^*$.

Problem: SAT (Boolean satisfaction in CNF)

$$\mathsf{SAT} = \{\langle \varphi \rangle : \varphi \in \text{CNF is satisfiable}\}$$

SAT is **NP**

Theorem

SAT $\in$ **NP**.

SAT is **NP**

Theorem

SAT $\in$ **NP**.

Proof idea.

Given a formula φ and the valuation u (certificate), we can verify in polynomial time whether $u \models \varphi$.
The valuation is at most as long as the formula, so it is a polynomial-size certificate. $\square$

Certificates for SAT

Example of a satisfiable formula

Let

$$\varphi = (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4)$$

$u = 0111$ is a certificate for φ because

$$0111 \models \varphi.$$

(valuation 0111 means $x_1 \mapsto 0, x_2 \mapsto 1, x_3 \mapsto 1, x_4 \mapsto 1$)
Verifying that 0111 satisfies each clause of φ can be done in polynomial time.

SAT is **NP-hard**

Theorem (Cook-Levin)

SAT $\in$ **NP-hard**.

Proof

Fix $\mathcal{L} \in \mathbf{NP}$ and let us show that $\mathcal{L} \leq_{\mathrm{p}} \mathsf{SAT}$.

SAT is **NP-hard**

Theorem (Cook-Levin)

SAT $\in$ **NP-hard**.

Proof

Fix $\mathcal{L} \in \mathbf{NP}$ and let us show that $\mathcal{L} \leq_{\mathrm{p}} \mathsf{SAT}$.
Since $\mathcal{L} \in \mathbf{NP}$, there is a polytime M and a polynomial p s.t.

$$x \in \mathcal{L} \qquad \text{iff} \qquad \exists u \in \{0,1\}^{p(|x|)} \; M(xu) = 1.$$

SAT is **NP-hard**

Theorem (Cook-Levin)

SAT $\in$ **NP-hard**.

Proof

Fix $\mathcal{L} \in \mathbf{NP}$ and let us show that $\mathcal{L} \leq_{\mathrm{p}} \mathsf{SAT}$.
Since $\mathcal{L} \in \mathbf{NP}$, there is a polytime M and a polynomial p s.t.

$$x \in \mathcal{L} \qquad \text{iff} \qquad \exists u \in \{0,1\}^{p(|x|)} \; M(xu) = 1.$$

We define $F_x : \{0,1\}^{p(|x|)} \to \{0,1\}$ as

$$F_x(u) = M(xu).$$

SAT is **NP-hard**

Theorem (Cook-Levin)

SAT $\in$ **NP-hard**.

Proof

Fix $\mathcal{L} \in \mathbf{NP}$ and let us show that $\mathcal{L} \leq_{\mathrm{p}} \mathsf{SAT}$.
Since $\mathcal{L} \in \mathbf{NP}$, there is a polytime M and a polynomial p s.t.

$$x \in \mathcal{L} \qquad \text{iff} \qquad \exists u \in \{0,1\}^{p(|x|)} \; M(xu) = 1.$$

We define $F_x : \{0,1\}^{p(|x|)} \to \{0,1\}$ as

$$F_x(u) = M(xu).$$

We can compute $\varphi_x(q_1, \ldots, q_{p(|x|)}) \in \mathrm{CNF}$ such that

$$u \models \varphi_x \qquad \text{iff} \qquad F_x(u) = 1 \qquad \text{iff} \qquad M(xu) = 1$$

SAT is **NP-hard**

Theorem (Cook-Levin)

SAT $\in$ **NP-hard**.

Proof

Fix $\mathcal{L} \in \mathbf{NP}$ and let us show that $\mathcal{L} \leq_{\mathrm{p}} \mathsf{SAT}$.
Since $\mathcal{L} \in \mathbf{NP}$, there is a polytime M and a polynomial p s.t.

$$x \in \mathcal{L} \qquad \text{iff} \qquad \exists u \in \{0,1\}^{p(|x|)}\ M(xu) = 1.$$

We define $F_x : \{0,1\}^{p(|x|)} \to \{0,1\}$ as

$$F_x(u) = M(xu).$$

We can compute $\varphi_x(q_1, \ldots, q_{p(|x|)}) \in \mathrm{CNF}$ such that

$$u \models \varphi_x \qquad \text{iff} \qquad F_x(u) = 1 \qquad \text{iff} \qquad M(xu) = 1$$

Then $x \in \mathcal{L}$ iff φ_x is satisfiable iff $\varphi_x \in \mathsf{SAT}$.

SAT is **NP-hard**

Theorem (Cook-Levin)

SAT $\in$ **NP-hard**.

Proof **incorrect!**

Fix $\mathcal{L} \in \mathbf{NP}$ and let us show that $\mathcal{L} \leq_{\mathrm{p}} \mathsf{SAT}$.
Since $\mathcal{L} \in \mathbf{NP}$, there is a polytime M and a polynomial p s.t.

$$x \in \mathcal{L} \qquad \text{iff} \qquad \exists u \in \{0,1\}^{p(|x|)}\ M(xu) = 1.$$

We define $F_x : \{0,1\}^{p(|x|)} \to \{0,1\}$ as

$$F_x(u) = M(xu).$$

We can compute $\varphi_x(q_1, \ldots, q_{p(|x|)}) \in \mathrm{CNF}$ such that

$$u \models \varphi_x \qquad \text{iff} \qquad F_x(u) = 1 \qquad \text{iff} \qquad M(xu) = 1$$

Then $x \in \mathcal{L}$ iff φ_x is satisfiable iff $\varphi_x \in \mathsf{SAT}$.
Problem: φ_x has exponential size: $O(p(|x|)2^{p(|x|)})$.

Proof idea of Cook-Levin Theorem (now correct!)

Fix $\mathcal{L} \in \mathbf{NP}$ and let us show that $\mathcal{L} \leq_{\mathrm{p}} \mathsf{SAT}$.
Since $\mathcal{L} \in \mathbf{NP}$, there is a machine M such that

- M runs in time $t(n)$, with t a polynomial
- there exists a polynomial p such that

$$x \in \mathcal{L} \qquad \text{iff} \qquad \exists u \in \{0,1\}^{p(|x|)} \; M(xu) = 1$$

Proof idea of Cook-Levin Theorem (now correct!)

Fix $\mathcal{L} \in \mathbf{NP}$ and let us show that $\mathcal{L} \leq_{\mathrm{p}} \mathsf{SAT}$.
Since $\mathcal{L} \in \mathbf{NP}$, there is a machine M such that

- M runs in time $t(n)$, with t a polynomial
- there exists a polynomial p such that

$$x \in \mathcal{L} \qquad \text{iff} \qquad \exists u \in \{0,1\}^{p(|x|)} \; M(xu) = 1$$

We work with a summarized version of a configuration, called **snapshot**, which contains only the information needed to compute the next snapshot.

- Fixing M, each snapshot can be encoded with constant k bits

The computation of M on input xu has length $m = t(\underbrace{|x| + p(|x|)}_{\text{length of } xu})$,

Proof idea of Cook-Levin Theorem (now correct!)

Fix $\mathcal{L} \in \mathbf{NP}$ and let us show that $\mathcal{L} \leq_{\mathrm{p}} \mathsf{SAT}$.
Since $\mathcal{L} \in \mathbf{NP}$, there is a machine M such that

- M runs in time $t(n)$, with t a polynomial
- there exists a polynomial p such that

$$x \in \mathcal{L} \qquad \text{iff} \qquad \exists u \in \{0,1\}^{p(|x|)} \; M(xu) = 1$$

We work with a summarized version of a configuration, called **snapshot**, which contains only the information needed to compute the next snapshot.

- Fixing M, each snapshot can be encoded with constant k bits

The computation of M on input xu has length $m = t(\underbrace{|x| + p(|x|)}_{\text{length of } xu})$,

M accepts xu iff there exists a sequence of snapshots where the first one corresponds to the initial configuration and the last one to an accepting configuration, and each snapshot follows from the previous one according to the transition function δ of M.

Given x we compute in polynomial time a Boolean formula φ_x with variables $\bar{i}$, $\bar{c}$, $\bar{s}_0, \ldots, \bar{s}_m$ such that:

- $\bar{i}$ encodes that the first part of the input is x
- $\bar{c}$ encodes the second part of the input, the certificate u
- $\bar{s}_0$ encodes the initial snapshot z_0 (depending on x and u)
- $\bar{s}_{i+1}$ encodes the snapshot that follows from the snapshot encoded by $\bar{s}_i$
- $\bar{s}_m$ encodes that the last snapshot and corresponds to an accepting configuration

φ_x is satisfiable iff there exists a certificate u and a sequence of snapshots of $M(xu)$ satisfying the following above conditions (namely, accepting x).
Therefore

$$x \in \mathcal{L} \qquad \text{iff} \qquad \varphi_x \in \mathsf{SAT}.$$

φ_x is computable in polytime from x (and hence has polysize). □

3SAT

A Boolean formula is in 3CNF if it is of the form

$$\underbrace{(a_{11} \vee a_{12} \vee a_{13})}_{\text{clause}} \wedge \underbrace{(a_{21} \vee a_{22} \vee a_{23})}_{\text{clause}} \wedge \cdots \wedge \underbrace{(a_{m1} \vee a_{m2} \vee a_{m3})}_{\text{clause}}$$

where a_{ij} is a **literal**.

3SAT

A Boolean formula is in 3CNF if it is of the form

$$\underbrace{(a_{11} \vee a_{12} \vee a_{13})}_{\text{clause}} \wedge \underbrace{(a_{21} \vee a_{22} \vee a_{23})}_{\text{clause}} \wedge \cdots \wedge \underbrace{(a_{m1} \vee a_{m2} \vee a_{m3})}_{\text{clause}}$$

where a_{ij} is a **literal**.

Problem: SAT (Boolean satisfaction in CNF)

$$\mathsf{SAT} = \{\langle \varphi \rangle : \varphi \in \text{CNF is satisfiable}\}$$

Problem: 3SAT (Boolean satisfiability in 3CNF)

$$\mathsf{3SAT} = \{\langle \varphi \rangle : \varphi \in \text{3CNF is satisfiable}\}$$

Proposition

$\mathsf{SAT} \leq_{\mathrm{p}} \mathsf{3SAT}$, so 3SAT is also **NP-complete**.

Example of an **NP-complete** problem

Example of an **NP-complete** problem

Proposition

INDSET $\in$ **NP-complete**.

Example of an **NP-complete** problem

Proposition

INDSET $\in$ **NP-complete**.

Idea of the Proof.

We already saw that INDSET is in **NP**.

Example of an **NP-complete** problem

Proposition

INDSET $\in$ **NP-complete**.

Idea of the Proof.

We already saw that INDSET is in **NP**.
To see that INDSET is **NP-hard**, we show that 3SAT $\leq_{\mathrm{p}}$ INDSET.
Given

$$\varphi = (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_4) \wedge (x_2 \vee x_3 \vee \neg x_4)$$

Example of an **NP-complete** problem

Proposition

INDSET $\in$ **NP-complete**.

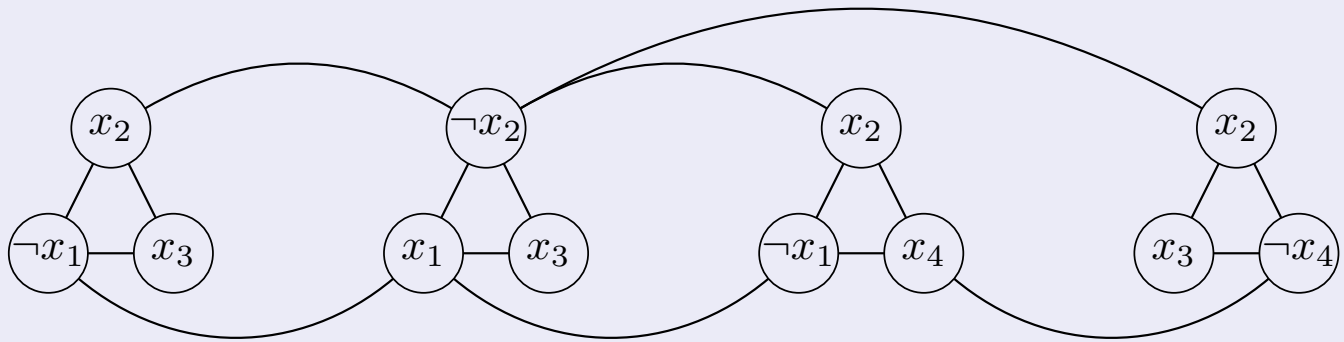
Idea of the Proof.

We already saw that INDSET is in **NP**.
To see that INDSET is **NP-hard**, we show that 3SAT $\leq_{\mathrm{p}}$ INDSET.
Given

$$\varphi = (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_4) \wedge (x_2 \vee x_3 \vee \neg x_4)$$

we compute in polytime the following graph:

$G_\varphi =$



Example of an **NP-complete** problem

Proposition

INDSET $\in$ **NP-complete**.

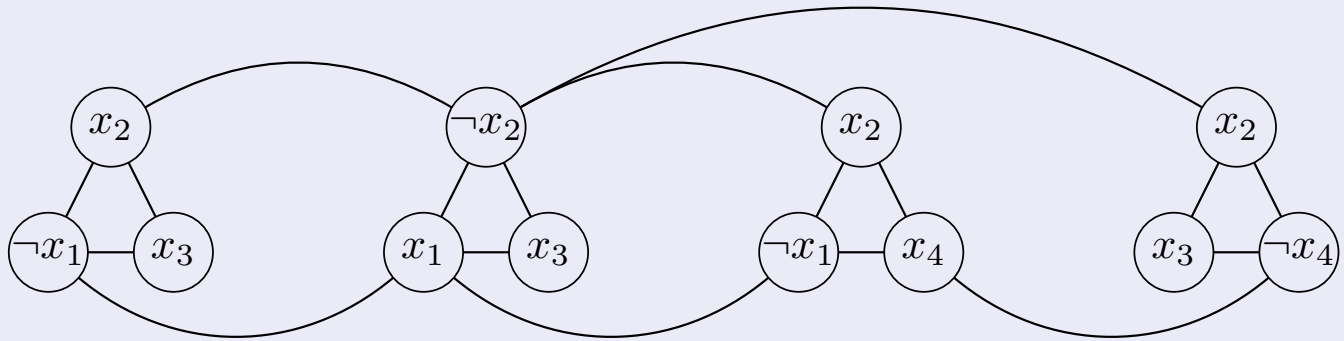
Idea of the Proof.

We already saw that INDSET is in **NP**.
To see that INDSET is **NP-hard**, we show that 3SAT $\leq_{\mathrm{p}}$ INDSET.
Given

$$\varphi = (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_4) \wedge (x_2 \vee x_3 \vee \neg x_4)$$

we compute in polytime the following graph:

$G_\varphi =$



Example of an **NP-complete** problem

Proposition

INDSET $\in$ **NP-complete**.

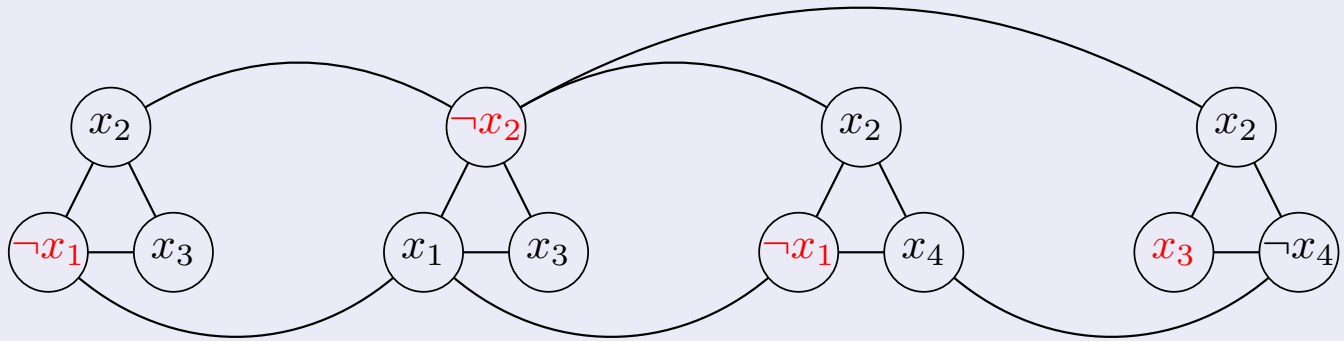
Idea of the Proof.

We already saw that INDSET is in **NP**.
To see that INDSET is **NP-hard**, we show that 3SAT $\leq_{\mathrm{p}}$ INDSET.
Given

$$\varphi = (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_4) \wedge (x_2 \vee x_3 \vee \neg x_4)$$

we compute in polytime the following graph:

$G_\varphi =$



Example of an **NP-complete** problem

Proposition

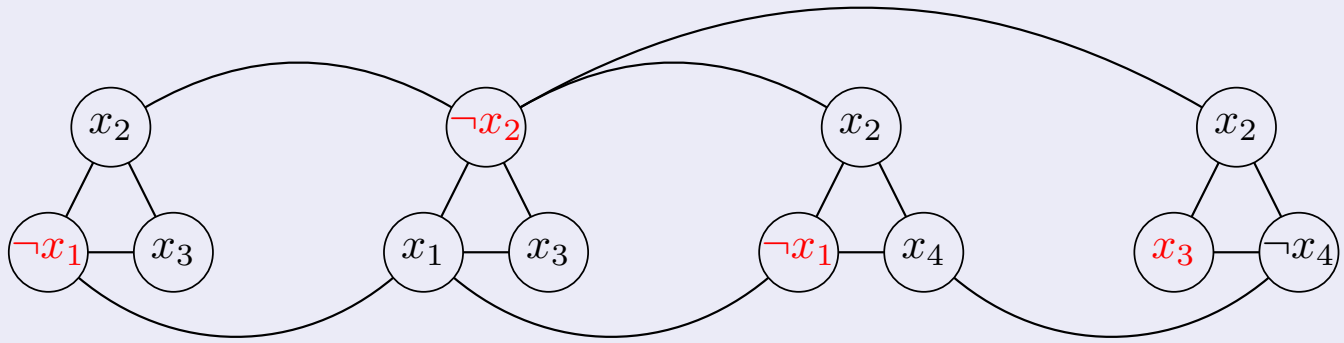
INDSET $\in$ **NP-complete**.

Idea of the Proof.

We already saw that INDSET is in **NP**.
To see that INDSET is **NP-hard**, we show that 3SAT $\leq_{\mathrm{p}}$ INDSET.
Given

$$\varphi = (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_4) \wedge (x_2 \vee x_3 \vee \neg x_4)$$

we compute in polytime the following graph:

$G_\varphi =$ 

$\varphi \in$ 3SAT iff G_φ has an independent set of size 4 iff $\langle G_\varphi, 4\rangle \in$ INDSET.

Examples of logic problems and their complexity

Boolean model checking

Given a valuation v and a Boolean formula φ, deciding if $v \models \varphi$ is in $\mathbf{P}$.

Examples of logic problems and their complexity

Boolean model checking

Given a valuation v and a Boolean formula φ, deciding if $v \models \varphi$ is in $\mathbf{P}$.

Modal model checking

Given a finite pointed Kripke structure K, v and a modal formula φ, deciding if $K, v \models \varphi$ is in $\mathbf{P}$.

Examples of logic problems and their complexity

Boolean model checking

Given a valuation v and a Boolean formula φ, deciding if $v \models \varphi$ is in $\mathbf{P}$.

Modal model checking

Given a finite pointed Kripke structure K, v and a modal formula φ, deciding if $K, v \models \varphi$ is in $\mathbf{P}$.

Existence of modal bisimulation

Given finite pointed Kripke structures K_1, v_1 and K_2, v_2, deciding if $K_1, v_1 \leftrightarrows K_2, v_2$ is in $\mathbf{P}$.

Examples of logic problems and their complexity

3SAT

Deciding if a 3CNF Boolean formula is satisfiable is **NP-complete**.

Examples of logic problems and their complexity

3SAT

Deciding if a 3CNF Boolean formula is satisfiable is **NP-complete**.

2SAT

Deciding if a 2CNF Boolean formula is satisfiable is in **P**.

Examples of logic problems and their complexity

3SAT

Deciding if a 3CNF Boolean formula is satisfiable is **NP-complete**.

2SAT

Deciding if a 2CNF Boolean formula is satisfiable is in **P**.

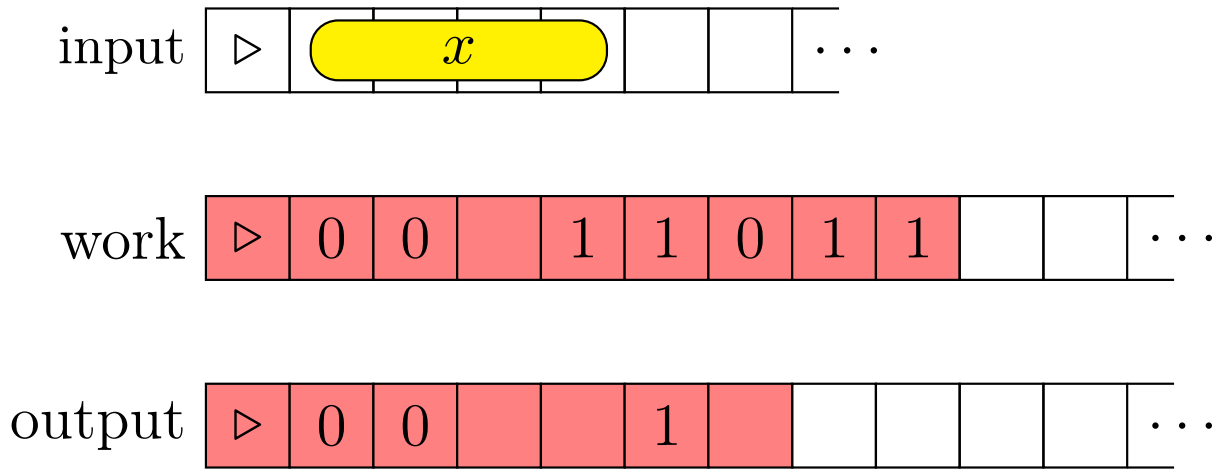
Model checking of positive existential first-order formulas

Given a finite structure A and a positive existential first-order sentence φ, deciding if $A \models \varphi$ is **NP-complete**.

- Positive: no negation
- Existential: only existential quantifiers

The class PSpace

Space used by a Turing machine



Definition

Given a Turing machine M and $x \in \{0,1\}^*$, the **space** used by M on input x is the number of cells ever visited by the head on the work and output tapes during the computation of M starting from x. Input tape cells are not counted.

PSpace

Complexity class: PSpace

PSpace is the class of languages $\mathcal{L}$ such that there exists a Turing machine M such that

- M decides $\mathcal{L}$
- M uses polynomial space

PSpace

Complexity class: PSpace

PSpace is the class of languages $\mathcal{L}$ such that there exists a Turing machine M such that

- M decides $\mathcal{L}$
- M uses polynomial space

Proposition

NP $\subseteq$ **PSpace**.

PSPACE

Complexity class: PSPACE

PSPACE is the class of languages $\mathcal{L}$ such that there exists a Turing machine M such that

- M decides $\mathcal{L}$
- M uses polynomial space

Proposition

$\mathbf{NP} \subseteq \mathbf{PSPACE}$.

Idea of the proof.

Suppose $\mathcal{L} \in \mathbf{NP}$. Then there exists a polytime M and a polynomial $p(n)$ such that for all $x \in \{0,1\}^*$:

$$x \in \mathcal{L} \qquad \text{iff} \qquad \text{there exists } u \in \{0,1\}^{p(|x|)} \text{ s.t. } M(xu) = 1$$

To decide if $x \in \mathcal{L}$ we can try all certificates of length $p(|x|)$, one by one, reusing the same space for each certificate. The time is exponential, but the space is polynomial. □

Quantified Boolean Formulas (QBF)

- A **quantified Boolean formula (QBF)** is a Boolean formula in which variables may be bound by quantifiers —existential ($\exists$) and universal ($\forall$)— instead of being all free.
- It's the propositional analogue of first-order logic with quantification over Boolean variables.

Example

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

is false: if x_1 is true, then x_3 must also be true, but x_3 is universally quantified.

The TQBF problem

Problem: TQBF (*True Quantified Boolean Formula*)

$$\mathsf{TQBF} = \{\langle \psi \rangle : \psi \text{ is a true QBF sentence}\}$$

The TQBF problem

Problem: TQBF (*True Quantified Boolean Formula*)

$$\mathsf{TQBF} = \{\langle\psi\rangle : \psi \text{ is a true QBF sentence}\}$$

Theorem

$\mathsf{TQBF} \in$ **PSPACE-complete**

Idea of the proof.

For showing that $\mathsf{TQBF} \in$ **PSPACE**, we can use a recursive algorithm that evaluates the formula by eliminating one quantifier at a time, reusing the same space for each recursive call.
The proof of hardness is more involved (see next). □

Configuration graph

Definition

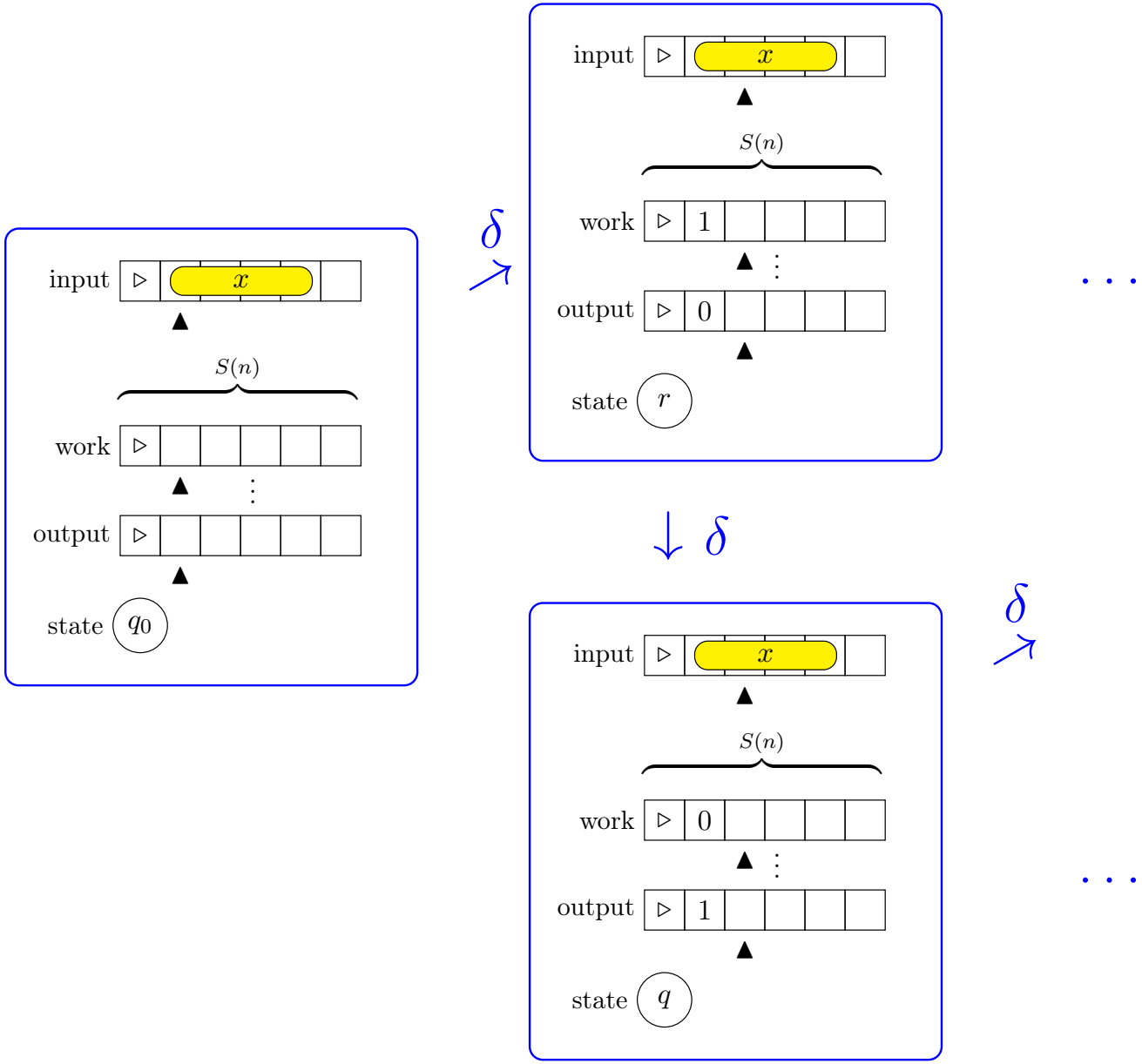
Let M be a Turing machine that uses space $S(n)$. The **configuration graph** of M on input x, denoted $G_{M,x}$, is a directed graph such that

- the set of vertices is the configurations using up to $S(|x|)$ cells on each work tape and on the output tape
- there is an edge from C to C' if C' is a one-step evolution of C according to M

Note

M accepts x if and only if there exists a path in $G_{M,x}$ from the initial configuration of M on input x to the accepting configuration (we may assume that there is only one).

Configuration graph



'One-step evolution' in CNF

Proposition

Let M be a Turing machine that uses space $S(n) \geq \log n$, and let c be a constant such that for every $x \in \{0,1\}^*$, any configuration C of a computation of M on input x can be represented with $|\langle C \rangle| = c \cdot S(|x|)$ bits.
Given x, we can compute in polynomial time in $S(n)$ a formula $\varphi_{M,x}(\bar{s}, \bar{t})$ in CNF with variables

$$\bar{s} = s_1, \ldots, s_{c \cdot S(|x|)} \qquad \text{and} \qquad \bar{t} = t_1, \ldots, t_{c \cdot S(|x|)}$$

such that

$\langle C \rangle \langle C' \rangle \models \varphi_{M,x}(\bar{s}, \bar{t})$ iff there is an edge from C to C' in $G_{M,x}$

Proof that TQBF $\in$ PSPACE-hard

- Let $\mathcal{L} \in \mathbf{PSPACE}$ and let M be a machine that decides $\mathcal{L}$ using space $S(n)$, where S is a polynomial.

Proof that TQBF $\in$ PSPACE-hard

- Let $\mathcal{L} \in \mathbf{PSPACE}$ and let M be a machine that decides $\mathcal{L}$ using space $S(n)$, where S is a polynomial.
- We consider $G_{M,x}$ and define QBF formulas $\psi_i(\bar{s}, \bar{t})$ with free variables $\bar{s}, \bar{t}$, which are tuples of dimension $c \cdot S(|x|)$, such that

$$\langle C \rangle \langle C' \rangle \models \psi_i(\bar{s}, \bar{t}) \quad \text{iff} \quad \begin{array}{l} \text{there is a directed path of} \\ \text{length} \leq 2^i \text{ in } G_{M,x} \text{ from } C \\ \text{to } C' \end{array}$$

Proof that TQBF $\in$ PSPACE-hard

- Let $\mathcal{L} \in \mathbf{PSPACE}$ and let M be a machine that decides $\mathcal{L}$ using space $S(n)$, where S is a polynomial.
- We consider $G_{M,x}$ and define QBF formulas $\psi_i(\bar{s}, \bar{t})$ with free variables $\bar{s}, \bar{t}$, which are tuples of dimension $c \cdot S(|x|)$, such that

$$\langle C \rangle \langle C' \rangle \models \psi_i(\bar{s}, \bar{t}) \quad \text{iff} \quad \begin{array}{l}\text{there is a directed path of} \\ \text{length} \leq 2^i \text{ in } G_{M,x} \text{ from } C \\ \text{to } C'\end{array}$$

- Let $n = |x|$. If there is a path from C to C' in $G_{M,x}$, then there is one of length $\leq 2^{c \cdot S(n)}$, because $G_{M,x}$ has at most $2^{c \cdot S(n)}$ vertices.
- $x \in \mathcal{L}$ iff $\overbrace{\langle C_0 \rangle \langle C_f \rangle \models \psi_{c \cdot S(n)}(\bar{s}, \bar{t})}^{\text{QBF}}$
- $\mathcal{L} \leq_{\text{p}} \mathsf{TQBF}$ as long as we can define $\psi_{c \cdot S(n)}$ in polynomial time in $n = |x|$.

Recall

$$\langle C\rangle\langle C'\rangle \models \psi_i(\bar{s},\bar{t}) \quad \text{iff} \quad \begin{array}{l}\text{there is a directed path of}\\ \text{length} \leq 2^i \text{ in } G_{M,x} \text{ from } C\\ \text{to } C'\end{array}$$

We define $\psi_i(\bar{s},\bar{t})$ recursively on i.

- $\psi_0(\bar{s},\bar{t})$ is defined as $\bar{s} = \bar{t} \vee \varphi_{M,x}(\bar{s},\bar{t})$. We compute ψ_0 in polynomial time in n.

Recall

$$\langle C\rangle\langle C'\rangle \models \psi_i(\bar{s},\bar{t}) \quad \text{iff} \quad \begin{array}{l}\text{there is a directed path of}\\ \text{length} \leq 2^i \text{ in } G_{M,x} \text{ from } C\\ \text{to } C'\end{array}$$

We define $\psi_i(\bar{s},\bar{t})$ recursively on i.

- $\psi_0(\bar{s},\bar{t})$ is defined as $\bar{s} = \bar{t} \vee \varphi_{M,x}(\bar{s},\bar{t})$. We compute ψ_0 in polynomial time in n.
- For $i > 0$ we could define

$$\psi_i(\bar{s},\bar{t}) = \exists \bar{r}\ \psi_{i-1}(\bar{s},\bar{r}) \wedge \psi_{i-1}(\bar{r},\bar{t})$$

Recall

$$\langle C\rangle\langle C'\rangle \models \psi_i(\bar{s},\bar{t}) \quad \text{iff} \quad \begin{array}{l}\text{there is a directed path of}\\ \text{length} \leq 2^i \text{ in } G_{M,x} \text{ from } C\\ \text{to } C'\end{array}$$

We define $\psi_i(\bar{s},\bar{t})$ recursively on i.

- $\psi_0(\bar{s},\bar{t})$ is defined as $\bar{s} = \bar{t} \vee \varphi_{M,x}(\bar{s},\bar{t})$. We compute ψ_0 in polynomial time in n.
- For $i > 0$ we could define

$$\psi_i(\bar{s},\bar{t}) = \exists \bar{r}\ \psi_{i-1}(\bar{s},\bar{r}) \wedge \psi_{i-1}(\bar{r},\bar{t})$$

but the size of ψ_i would be exponential in i, so it cannot be computed in polynomial time; this approach fails.

- For $i > 0$, instead we define:

$$\psi_i(\bar{s}, \bar{t}) = \exists \bar{r} \forall \bar{u}, \bar{v} \; ((\bar{u} = \bar{s} \wedge \bar{v} = \bar{r}) \vee (\bar{u} = \bar{r} \wedge \bar{v} = \bar{t})) \rightarrow \psi_{i-1}(\bar{u}, \bar{v})$$

$$\overbrace{\bar{s} \rightarrow \cdots\cdots\cdots \rightarrow \bar{r}}^{(\bar{u}=\bar{s}\wedge\bar{v}=\bar{r})\rightarrow\psi_{i-1}(\bar{u},\bar{v})} \underbrace{\rightarrow \cdots\cdots\cdots \rightarrow \bar{t}}_{(\bar{u}=\bar{r}\wedge\bar{v}=\bar{t})\rightarrow\psi_{i-1}(\bar{u},\bar{v})}$$

- For $i > 0$, instead we define:

$$\psi_i(\bar{s}, \bar{t}) = \exists \bar{r} \forall \bar{u}, \bar{v}$$
$$((\bar{u} = \bar{s} \wedge \bar{v} = \bar{r}) \vee (\bar{u} = \bar{r} \wedge \bar{v} = \bar{t})) \rightarrow \psi_{i-1}(\bar{u}, \bar{v})$$

$$\overbrace{\bar{s} \rightarrow \cdots\cdots\cdots \rightarrow \bar{r}}^{(\bar{u}=\bar{s}\wedge\bar{v}=\bar{r})\rightarrow\psi_{i-1}(\bar{u},\bar{v})} \underbrace{\rightarrow \cdots\cdots\cdots \rightarrow \bar{t}}_{(\bar{u}=\bar{r}\wedge\bar{v}=\bar{t})\rightarrow\psi_{i-1}(\bar{u},\bar{v})}$$

- To compute ψ_i we need i recursive calls down to the base case.

- For $i > 0$, instead we define:

$$\psi_i(\bar{s}, \bar{t}) = \exists \bar{r} \forall \bar{u}, \bar{v} \\ ((\bar{u} = \bar{s} \wedge \bar{v} = \bar{r}) \vee (\bar{u} = \bar{r} \wedge \bar{v} = \bar{t})) \rightarrow \psi_{i-1}(\bar{u}, \bar{v})$$

$$\overbrace{\bar{s} \rightarrow \cdots\cdots\cdots \rightarrow \bar{r}}^{(\bar{u}=\bar{s} \wedge \bar{v}=\bar{r}) \rightarrow \psi_{i-1}(\bar{u},\bar{v})} \rightarrow \underbrace{\cdots\cdots\cdots \rightarrow \bar{t}}_{(\bar{u}=\bar{r} \wedge \bar{v}=\bar{t}) \rightarrow \psi_{i-1}(\bar{u},\bar{v})}$$

- To compute ψ_i we need i recursive calls down to the base case.
- Thus we compute $\psi_{c \cdot S(n)}$ in polynomial time in $S(n)$, which is polynomial in n because S is also polynomial.
- $\psi_{c \cdot S(n)}$ can be transformed into a QBF in polytime.

Examples of logic problems and their complexity

Modal satisfiability

Deciding if a modal formula is satisfiable is **PSPACE-complete**.

Examples of logic problems and their complexity

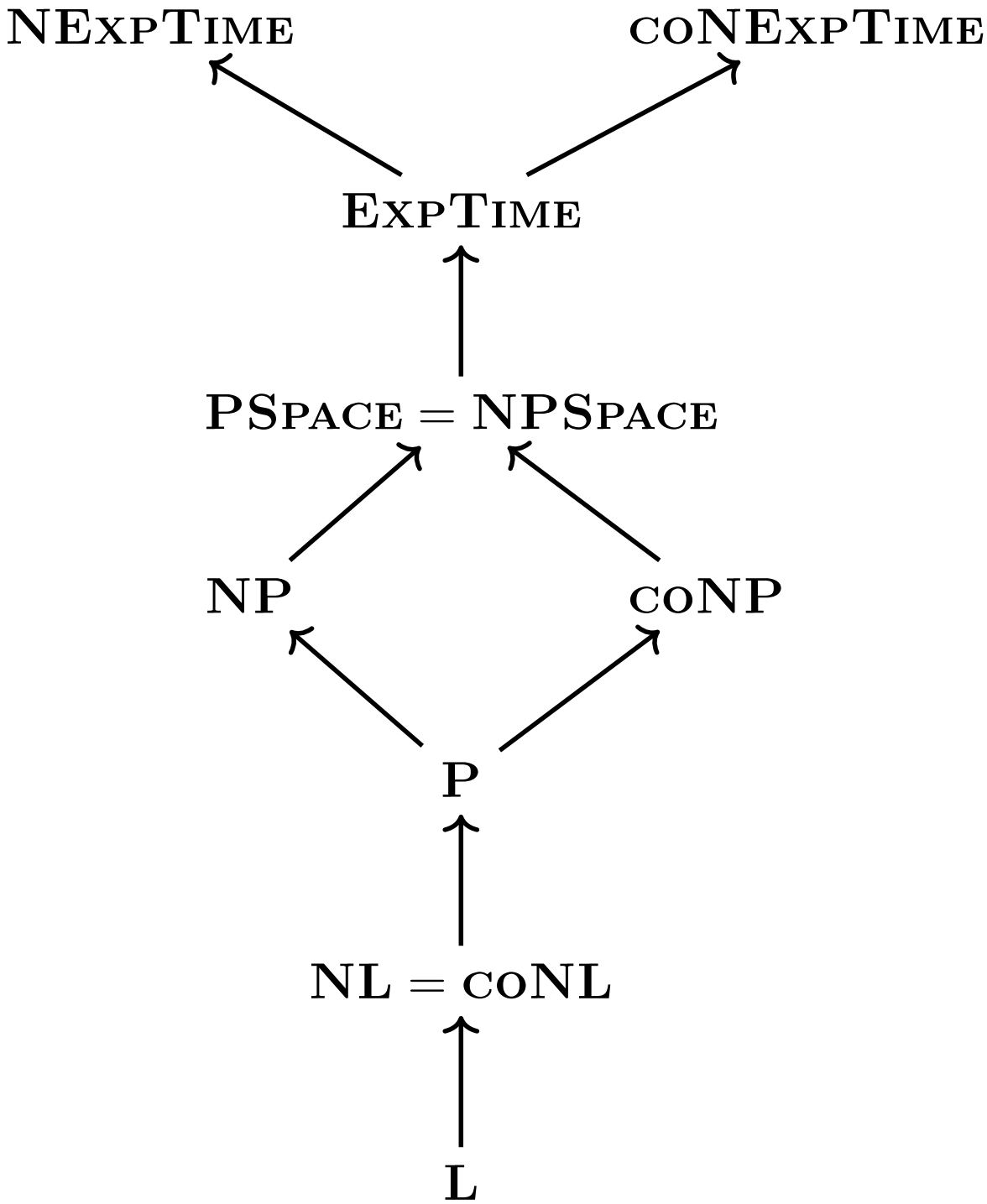
Modal satisfiability

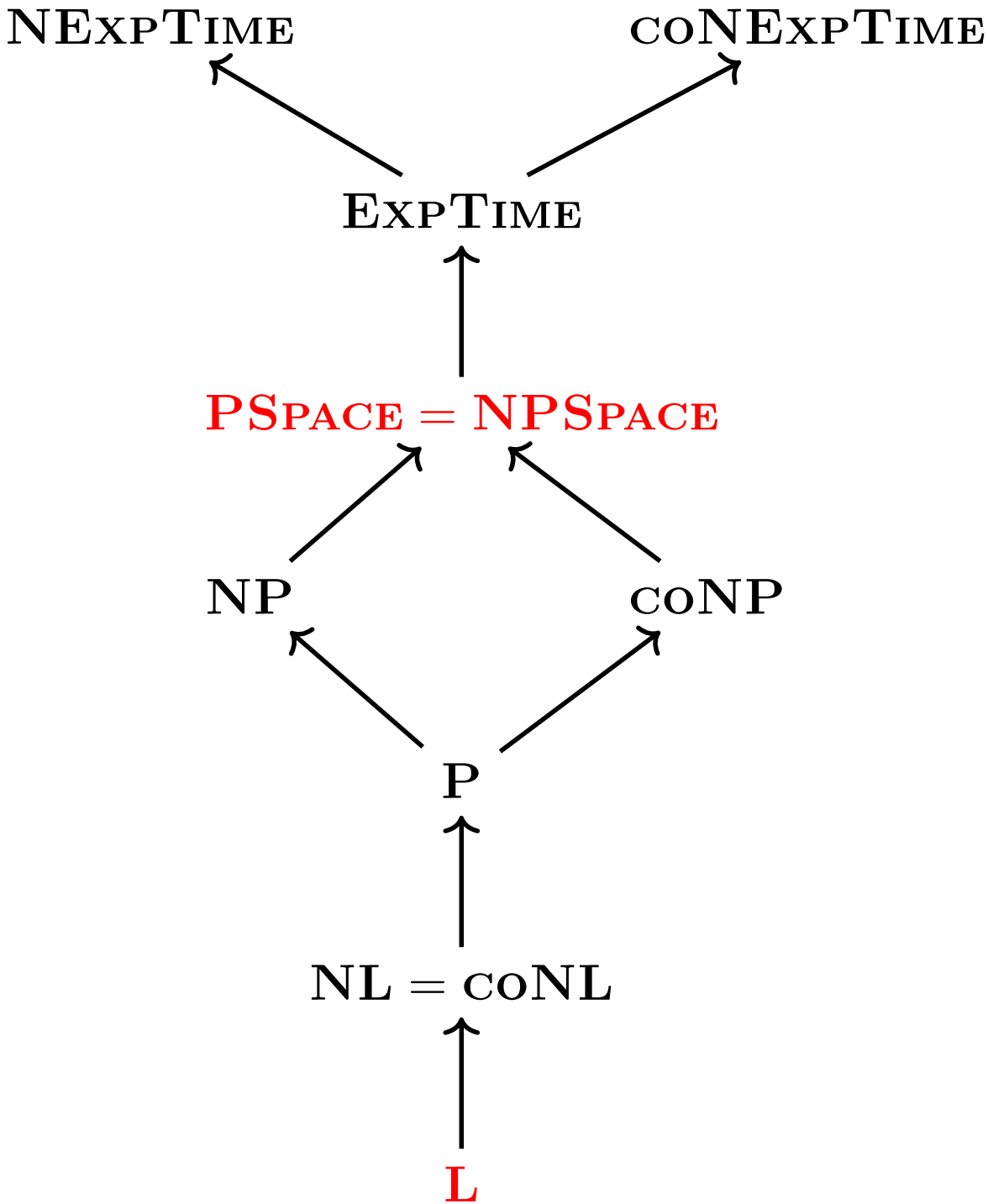
Deciding if a modal formula is satisfiable is **PSPACE-complete**.

First-order model checking

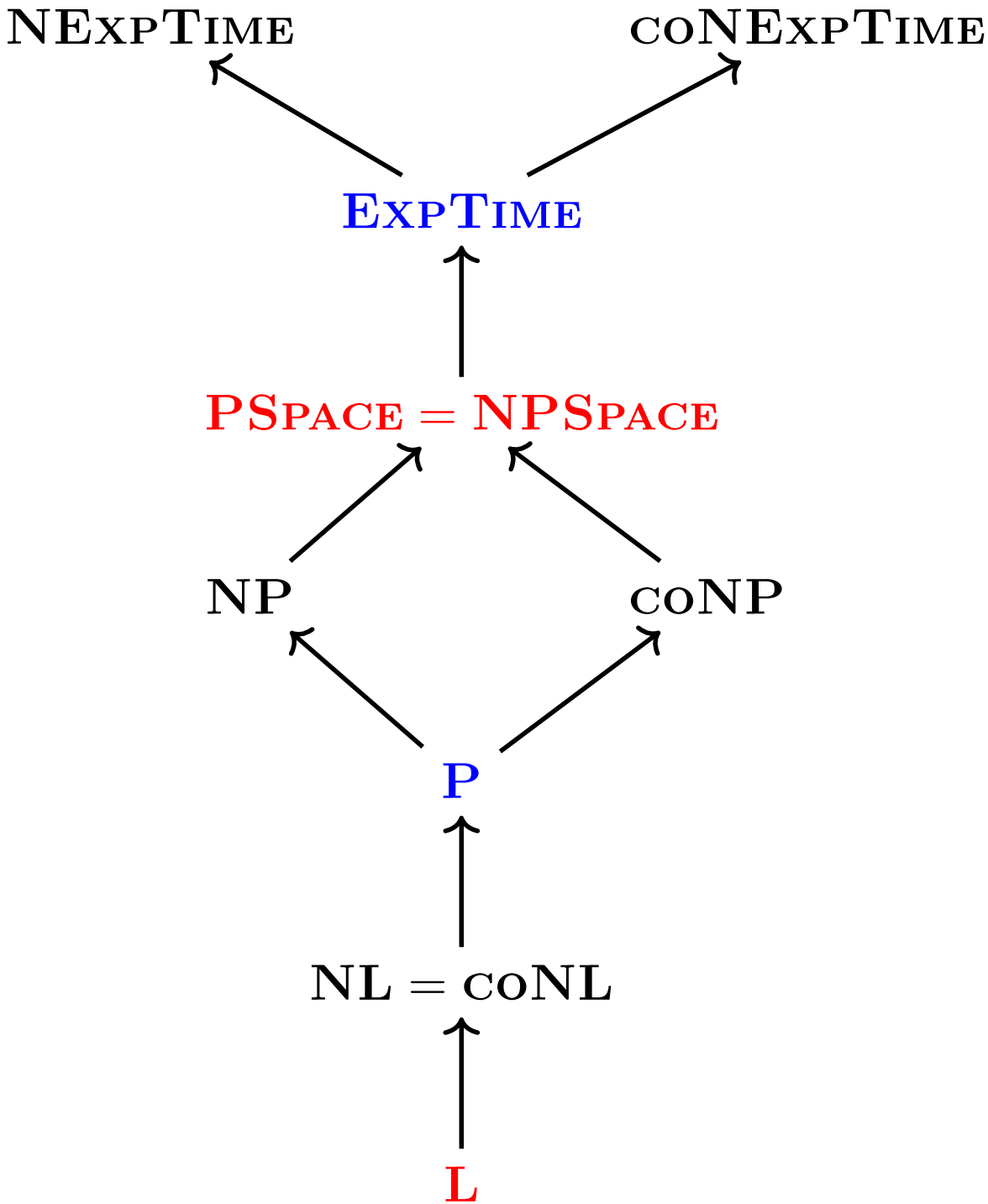
Given a finite structure A and a first-order sentence φ, deciding if $A \models \varphi$ is **PSPACE-complete**.

Beyond **PSpace**


NExpTime
coNExpTime
ExpTime
PSpace = NPSpace
NP
coNP
P
NL = coNL
L

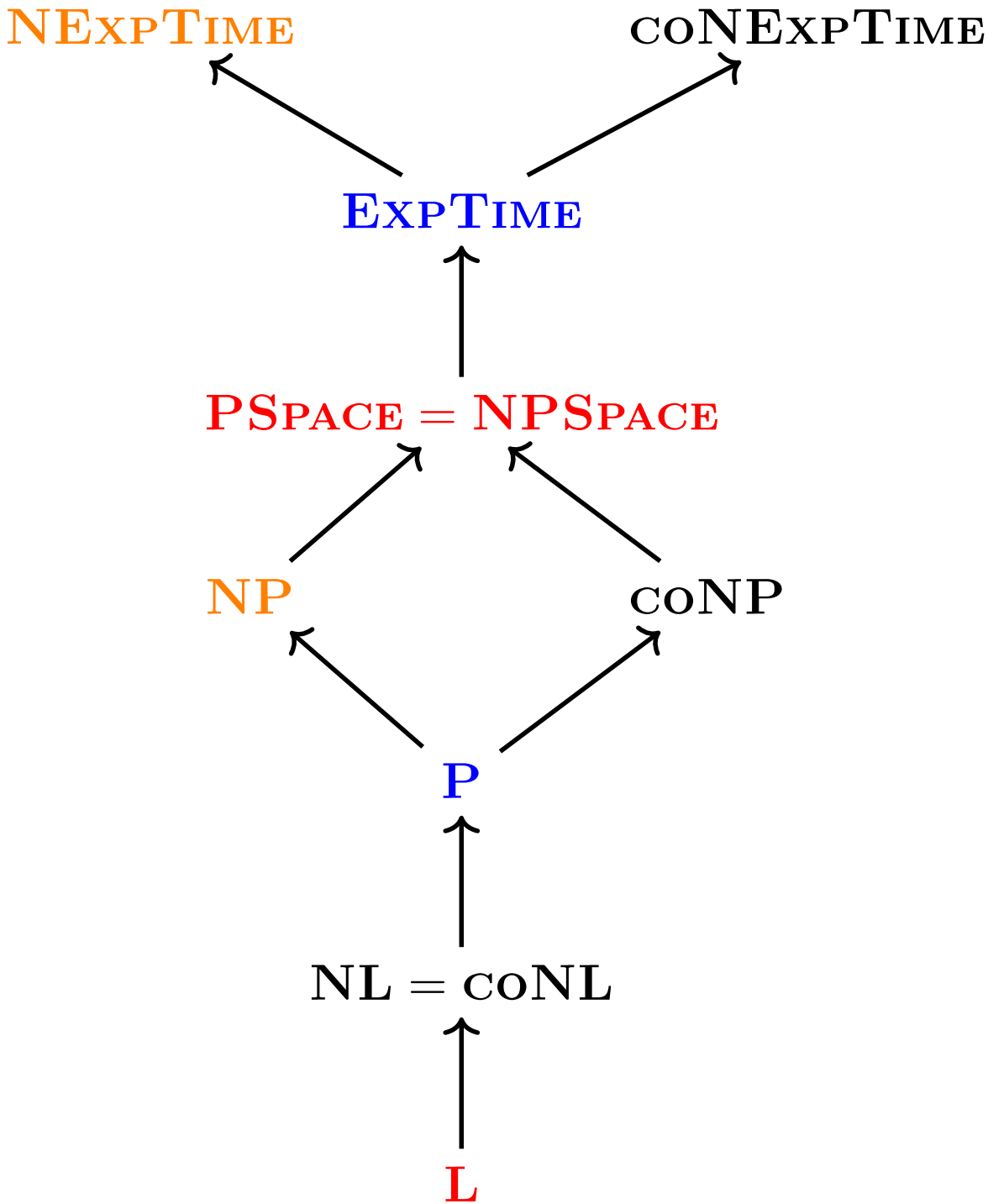


$\mathbf{L} \subsetneq \mathbf{PSpace}$



$\mathbf{L} \subsetneq \mathbf{PSpace}$

$\mathbf{P} \subsetneq \mathbf{ExpTime}$



$\mathbf{L} \subsetneq \mathbf{PSpace}$

$\mathbf{P} \subsetneq \mathbf{ExpTime}$

$\mathbf{NP} \subsetneq \mathbf{NExpTime}$

Example of logic problems and their complexity

Monadic Second Order Logic (MSO) extends FO by allowing quantification over sets of elements, not just over individual elements.

MSO model checking

Given a finite structure A and an MSO-sentence φ, decide if $A \models \varphi$ is

- **PSpace-complete** if A is restricted to finite words (relations for talking about positions and labels)
- **ExpTime-complete** if A is restricted to finite labeled trees (relations to talk about 'child' and labels)
- **Non-elementary** if A is an arbitrary finite graph.
  - Non-elementary means that the time is not bounded by a tower of exponentials:

$$2^{2^{\cdot^{\cdot^{2^n}}}}$$

Expressiveness vs. Complexity

- The richer the logical language, the more it can express—but the harder its computational problems become.
- Propositional logic: satisfiability is **NP-complete**, but its expressive power is very limited.
- First-order logic: highly expressive, but satisfiability is undecidable.
- Modal logic: offers a good balance between expressive power and computational behavior!

Expressiveness vs. Complexity

- The richer the logical language, the more it can express—but the harder its computational problems become.
- Propositional logic: satisfiability is **NP-complete**, but its expressive power is very limited.
- First-order logic: highly expressive, but satisfiability is undecidable.
- Modal logic: offers a good balance between expressive power and computational behavior!

Trade-off

There is a fundamental balance between expressive power and computational tractability.

Modal logic: where expressiveness meets efficiency.
Not too weak to be boring,
not too strong to be undecidable.

Thank you!