How to prove **P** = **NP**?

Ralph Sarkis

SUMM - 2019

11 Janvier 2019

Introduction

What are we talking about ?

Introduction

What are we talking about ?

- Most famous open problem in theoretical computer science.
- Roughly translates to : If the solution to a problem can be verified easily, then can it be found easily ?
- Has way too many theoretical and practical applications.

Outline

Outline

- Formal definitions
  Computational models, Turing machines and complexity classes.

Outline

- Formal definitions
  Computational models, Turing machines and complexity classes.
- Upper bounds : algorithms and reductions.
  NP-hardness, Cook-Levin and examples.

Outline

- Formal definitions
  Computational models, Turing machines and complexity classes.
- Upper bounds : algorithms and reductions.
  NP-hardness, Cook-Levin and examples.
- Lower bounds : three barriers.
  Relativizing proofs, natural proofs and algebrizing proofs.

Origins of the question

Origins of the question

Hilbert's 10th problem

Given a polynomial with integer coefficients in several variables :
Find an algorithm that will determine whether it has integer roots or not.

Origins of the question

Hilbert's 10th problem

Given a polynomial with integer coefficients in several variables : Find an algorithm that will determine whether it has integer roots or not.

Emil Leon Post thought this was unsolvable and developed computability theory.

Models of computation

Rigorous definition of an algorithm, often with notions of resources and efficiency.

Models of computation

Rigorous definition of an algorithm, often with notions of resources and efficiency.

Examples

Turing machines, λ-calculus, finite state machines, RAM-model (modern computers), etc...

Turing machine

Defined as a 5-tuple $(Q, \Sigma, \Gamma, \delta, q_0)$ where

1. Q is a set of states with distinct accept and reject states,
2. Σ (a finite set of symbols) is the input alphabet,
3. $\Gamma \supseteq \Sigma$ (a finite set of symbols) is the tape alphabet,
4. $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the transition function, and
5. q_0 is the initial state.

Turing machine

Running a TM

Turing machine

Running a TM

1. Start with input $w \in \Sigma^*$ on tape, head on first symbol and at state q_0.

Turing machine

Running a TM

1. Start with input $w \in \Sigma^*$ on tape, head on first symbol and at state q_0.
2. At each step, follow the transition rule to change state, write on tape and move left or right.

Turing machine

Running a TM

1. Start with input $w \in \Sigma^*$ on tape, head on first symbol and at state q_0.
2. At each step, follow the transition rule to change state, write on tape and move left or right.
3. Run previous step until accept or reject state is reached.

Turing machine

Running a TM

1. Start with input $w \in \Sigma^*$ on tape, head on first symbol and at state q_0.
2. At each step, follow the transition rule to change state, write on tape and move left or right.
3. Run previous step until accept or reject state is reached.

A TM computes a function $f : \Sigma^* \to \Sigma^*$ if for any input w, it halts with $f(w)$ written on the tape.

Resources of a TM

Time

We say that M runs in time $T(n)$ if for any input of size n, the number of steps that M needs before halting is at most $T(n)$.

Resources of a TM

Time

We say that M runs in time $T(n)$ if for any input of size n, the number of steps that M needs before halting is at most $T(n)$.

Space

We say that M runs in space $S(n)$ if for any input of size n, the maximum number of cells of the tape used is at most $S(n)$.

Complexity classes

A complexity class is a set of functions $f : \Sigma^* \to \Sigma^*$ with "similar" needs in terms of resources.

Complexity classes

A complexity class is a set of functions $f : \Sigma^* \to \Sigma^*$ with "similar" needs in terms of resources.

Computable

A function f is computable if there exists a TM that computes f (no restriction on the resources).

Complexity classes

A complexity class is a set of functions $f : \Sigma^* \to \Sigma^*$ with "similar" needs in terms of resources.

Computable

A function f is computable if there exists a TM that computes f (no restriction on the resources).

P

A function f is in **P** if there exists a TM that computes f in time $T(n)$ for some polynomial T.

Complexity classes

A complexity class is a set of functions $f : \Sigma^* \to \Sigma^*$ with "similar" needs in terms of resources.

Computable

A function f is computable if there exists a TM that computes f (no restriction on the resources).

P

A function f is in **P** if there exists a TM that computes f in time $T(n)$ for some polynomial T.

NP

A function f is in **NP** if there exists a TM that, given $w, x \in \Sigma^*$, can say whether $f(x) = w$ in time $T(n)$ for some polynomial T.

Complexity classes

Complexity zoo.

Examples

Multiplication

Given two numbers $a, b \in \mathbb{N}$, output $a \cdot b$.

Examples

Multiplication

Given two numbers $a, b \in \mathbb{N}$, output $a \cdot b$.

Factorization

Given a number $x \in \mathbb{N}$, output primes $p_1, \ldots, p_n$ such that $p_1 \cdots p_n = n$.

Examples

Multiplication

Given two numbers $a, b \in \mathbb{N}$, output $a \cdot b$.

Factorization

Given a number $x \in \mathbb{N}$, output primes $p_1, \ldots, p_n$ such that $p_1 \cdots p_n = n$.

Sudoku

Given a Sudoku board, output the solved board.

Examples

Multiplication

Given two numbers $a, b \in \mathbb{N}$, output $a \cdot b$.

Factorization

Given a number $x \in \mathbb{N}$, output primes $p_1, \ldots, p_n$ such that $p_1 \cdots p_n = n$.

Sudoku

Given a Sudoku board, output the solved board.

Bitcoin mining

Given the data for a block, output a nonce that satisfies the mining requirement.

Reductions

We say that a problem A reduces to a problem B if an algorithm that solves B can be used to define an algorithm that solves A.

Reductions

We say that a problem A reduces to a problem B if an algorithm that solves B can be used to define an algorithm that solves A.

We write $A \leq B$ because B is at least as hard as A.

Reductions

We say that a problem A reduces to a problem B if an algorithm that solves B can be used to define an algorithm that solves A.

We write $A \leq B$ because B is at least as hard as A.

When interested in polytime reductions, we write $A \leq_p B$.

Cook-Levin

Satisfiability (SAT)

Given a boolean formula ϕ in n variables, decide whether there are values of $x_1, \ldots, x_n$ such that $\phi(x_1, \ldots, x_n)$.

Cook-Levin

Satisfiability (SAT)

Given a boolean formula ϕ in n variables, decide whether there are values of $x_1, \ldots, x_n$ such that $\phi(x_1, \ldots, x_n)$.

Theorem

Any problem in **NP** has a polytime reduction to SAT.

Cook-Levin

Satisfiability (SAT)

Given a boolean formula ϕ in n variables, decide whether there are values of $x_1, \ldots, x_n$ such that $\phi(x_1, \ldots, x_n)$.

Theorem

Any problem in **NP** has a polytime reduction to SAT.

Thus, if there exists an polytime algorithm for SAT, then **P** = **NP**.

NP-hardness

If SAT reduces to a problem X, then X is also **NP**-hard, hence solving X in polytime implies **P** $=$ **NP**.

NP-hardness

If SAT reduces to a problem X, then X is also **NP**-hard, hence solving X in polytime implies **P** $=$ **NP**.

Examples

Traveling salesperson problem, solving an $n \times n \times n$ Rubik's cube optimally, finding a valid move in an $n \times n$ checkers board.

Diagonalization

Some functions are not computable

The number of TM is countable because each TM has a finite description $(Q, \Sigma, \Gamma, \delta, q_0)$.

Diagonalization

Some functions are not computable

The number of TM is countable because each TM has a finite description $(Q, \Sigma, \Gamma, \delta, q_0)$.

The number of languages is uncountable (cardinality of $\mathcal{P}(\Sigma^*)$).

Diagonalization

Some functions are not computable

The number of TM is countable because each TM has a finite description $(Q, \Sigma, \Gamma, \delta, q_0)$.

The number of languages is uncountable (cardinality of $\mathcal{P}(\Sigma^*)$).

Unfortunately, this kind of proof relativizes.

Oracles

An oracle is a black box that can solve a particular problem in a single step.

Oracles

An oracle is a black box that can solve a particular problem in a single step.

We write $\mathbf{P}^L$ to denote the set of languages that can be decided in polytime with a TM that has access to an oracle for L.

Oracles

An oracle is a black box that can solve a particular problem in a single step.

We write $\mathbf{P}^L$ to denote the set of languages that can be decided in polytime with a TM that has access to an oracle for L.

We say that a proof relativizes if it works whether or not oracles are allowed.

Relativizing proofs

P vs. **NP** has contradicting relativizations

Relativizing proofs

P vs. **NP** has contradicting relativizations

Baker, Gill and Solovay showed there are languages A, B such that

$$\mathbf{P}^A = \mathbf{NP}^A$$

$$\mathbf{P}^B \neq \mathbf{NP}^B$$

Relativizing proofs

P vs. **NP** has contradicting relativizations

Baker, Gill and Solovay showed there are languages A, B such that

$$\mathbf{P}^A = \mathbf{NP}^A$$

$$\mathbf{P}^B \neq \mathbf{NP}^B$$

Thus, no relativizing proofs can lead to a solution.

Circuits

Definition

A directed acyclic graphs where each vertex is either an input with in-degree 0 or a gate computing AND, OR, or NOT of its inputs. One vertex is the output of the circuit.

Circuits

Definition

A directed acyclic graphs where each vertex is either an input with in-degree 0 or a gate computing AND, OR, or NOT of its inputs. One vertex is the output of the circuit.

Resources

The size of a circuit is the number of edges and the depth of a circuit is the maximal length of path ending in the output gate.

Circuits

Definition

A directed acyclic graphs where each vertex is either an input with in-degree 0 or a gate computing AND, OR, or NOT of its inputs. One vertex is the output of the circuit.

Resources

The size of a circuit is the number of edges and the depth of a circuit is the maximal length of path ending in the output gate.

P/poly denotes the set of languages that can be decided by a circuit of polynomial size.

Natural proofs

$$\mathbf{NP} \not\subseteq \mathbf{P/poly} \implies \mathbf{P} \neq \mathbf{NP}$$

Natural proofs

$$\mathbf{NP} \not\subseteq \mathbf{P/poly} \implies \mathbf{P} \neq \mathbf{NP}$$

Strategy

Natural proofs

$$\mathbf{NP} \not\subseteq \mathbf{P/poly} \implies \mathbf{P} \neq \mathbf{NP}$$

Strategy

1. Find some property X of functions that SAT (or some other **NP** problem) satisfies.

Natural proofs

$$\mathbf{NP} \not\subseteq \mathbf{P/poly} \implies \mathbf{P} \neq \mathbf{NP}$$

Strategy

1. Find some property X of functions that SAT (or some other **NP** problem) satisfies.
2. Show that no function in **P/poly** has property X.

Natural proofs

We say this is a natural proof if :

Natural proofs

We say this is a natural proof if :

1. "Many" functions satisfy X. (Largeness)

Natural proofs

We say this is a natural proof if :

1. "Many" functions satisfy X. (Largeness)
2. Easy to verify if a function satisfies X. (Constructivity)

Natural proofs

We say this is a natural proof if :

1. "Many" functions satisfy X. (Largeness)
2. Easy to verify if a function satisfies X. (Constructivity)

Razbarov and Rudich showed that natural proof will inevitably fail.

Algebrization

Similar to relativization, but we allow low-degree extension of oracles in finite fields.

Reasons to believe $\mathbf{P} = \mathbf{NP}$

Reasons to believe $\mathbf{P} = \mathbf{NP}$

- All these barriers to prove $\mathbf{P} \neq \mathbf{NP}$.

Reasons to believe **P** = **NP**

- All these barriers to prove **P** $\neq$ **NP**.
- We might find an $O(n^d)$ algorithm with d huge.

Reasons to believe **P** = **NP**

- All these barriers to prove **P** $\neq$ **NP**.
- We might find an $O(n^d)$ algorithm with d huge.
- We could prove existence of such an algorithm.

Reasons to believe **P** = **NP**

- All these barriers to prove **P** $\neq$ **NP**.
- We might find an $O(n^d)$ algorithm with d huge.
- We could prove existence of such an algorithm.
- Upper bounders have been way more successful than lower bounders.

Reasons to believe $\mathbf{P} \neq \mathbf{NP}$

Reasons to believe $\mathbf{P} \neq \mathbf{NP}$

- No one has been able to solve **NP-complete** problems in polytime.

Reasons to believe $\mathbf{P} \neq \mathbf{NP}$

- No one has been able to solve **NP-complete** problems in polytime.
- If $\mathbf{P} = \mathbf{NP}$, then $\mathbf{P} = \mathbf{PH}$ and **PH** includes so many harder problems (e.g. : proving theorems).

Thank you !

Merci !