LEAFLET

50 YEARS OF THEORETICAL COMPUTER SCIENCE

Since the birth of ICALP and the EATCS

Foreword

To highlight the 50th anniversary of the ICALP conference and of the creation of EATCS, IRIF (Institut de Recherche en Informatique Fondamentale) has set up an exhibition on *50 Years of Theoretical Computer Science: Since the birth of ICALP and the EATCS*.

When first held in July 1972 at IRIA Rocquencourt near Paris, ICALP was the first conference for the newly drafted European Association for Theoretical Computer Science (EATCS). Beyond the start of a new academic venue, this was in many ways a defining moment for theoretical computer science. Through a historical tour and an overview of some key topics, the exhibition *50 Years of Theoretical Computer Science: Since the birth of ICALP and the EATCS* offers a dive into this field, often way too little known.

The exhibition was first presented at ICALP'22 on July 6–8, 2022 at Université Paris Cité and then continued its tour to the Mathématiques Informatique Recherche (MIR) library from December 8, 2022 to February 27, 2023.

Designed simultaneously with the exhibition, this leaflet aims to bring the discipline and its history into the hands of all experts, curious and afficionados of theoretical computer science. Close to twenty French and international scientific contributors took part in this project, striving to share the history of the discipline and to reflect the diversity and richness of its themes. The “Further reading” sections invite the most curious to dive deeper into specific topics.

Enjoy your reading!

ISBN 9-782958-699703

50 years of Theoretical Computer Science

Since the birth of ICALP and the EATCS

HIGHLIGHTS

Table of contents

HISTORY

HIST

ORY

Brussels, 1972

Where, when and why EATCS and ICALP started

Informatics or computer science was seen by other disciplines and by many politicians as simply a technology to support other enterprises. It was already clear however that to improve the correctness and efficiency of large-scale programs, theoretical studies were needed to investigate the principles and properties of computing. At the time, such work in Europe tended to be local and national. New funding for inter-European collaboration would be required.

Left page: Marcel-Paul Schützenberger at the first ICALP conference, July 3-7 1972

In those years

> ***"There was a very special spirit in the air; we knew that we were witnessing the birth of a new scientific discipline centered on the computer*** – (R. Karp)

> ***"There was absolutely no appreciation of the work on the issues of computing. Mathematicians did not recognize the emerging new field*** – (M. Rabin)

Rapport préliminaire sur l'Informatique Théorique

(M. Nivat, L. Nolin, M.-P. Schützenberger, 1971)

▪ This report outlines the main pillars of the new science and, for each pillar, describes the research subject addressed, with reference to a few specific authors:

— Algorithms, with specific reference to arithmetic operations (Winograd), sorting (Knuth, Floyd), graph algorithms (Rabin);

— Automata and formal languages, with reference to equations on the free monoid (Lentin), codes, finite automata and regular languages (Kleene, Krohn & Rhodes), push-down automata and context-free languages (Schützenberger), tree automata;

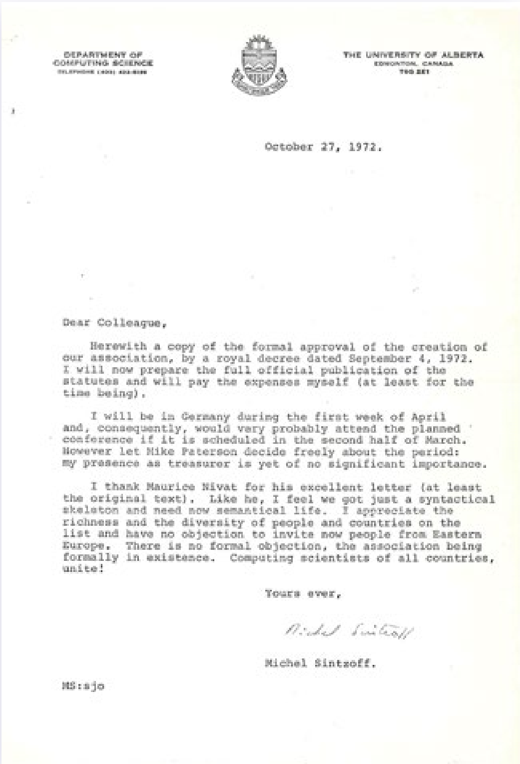
— Formal semantics of programming languages, where with experience from the syntactic and semantic definition of Algol 68, the need to provide precise formulations of the semantics of programming languages is discussed, based on the early works on axiomatic semantics (Floyd), operational semantics (McCarthy), approaches to semantics based on lambda-calculus (Scott) and combinatory logic (Nolin), and the theory of program schemes (Ianov, Luckham, Park & Paterson, and Strong).

The report underlines the theory of operating systems, of parallel concurrent and cooperating processes, and of the corresponding computation models (Dijkstra, Naur, Wirth) expected to play an important role in the future.

Foundation of the EATCS

▪ At the Berlaymont building of the EU Commission in Brussels, on January 27-28, 1972, there is a meeting chaired by Alfonso Caracciolo.

Participants: M. Nivat, L. Nolin, M. Gross (F), H. Langmaack, K. H. Böhling (D), I. Verbeek, J. de Bakker (NL), M. Paterson (UK), M. Sintzoff (B), C. Böhm, U. Montanari, G. Ausiello (I).
After presenting the report of M. Nivat, L. Nolin and M.-P. Schützenberger, they approve the proposal prepared by Maurice Nivat on cooperation among European universities, which leads in September to the creation of the European Association for Theoretical Computer Science (EATCS).



DEPARTMENT OF COMPUTING SCIENCE

THE UNIVERSITY OF ALBERTA
EDMONTON, CANADA

October 27, 1972.

Dear Colleague,

Herewith a copy of the formal approval of the creation of our association, by a royal decree dated September 4, 1972. I will now prepare the full official publication of the statutes and will pay the expenses myself (at least for the time being).

I will be in Germany during the first week of April and, consequently, would very probably attend the planned conference if it is scheduled in the second half of March. However let Mike Paterson decide freely about the period: my presence as treasurer is yet of no significant importance.

I thank Maurice Nivat for his excellent letter (at least the original text). Like he, I feel we got just a syntactical skeleton and need now semantical life. I appreciate the richness and the diversity of people and countries on the list and have no objection to invite now people from Eastern Europe. There is no formal objection, the association being formally in existence. Computing scientists of all countries, unite!

Yours ever,

Michel Sintzoff.

MS:sjo

Maurice Gross and Maurice Nivat at the first ICALP

First ICALP

On July 3-7, 1972, at IRIA (Rocquencourt, Paris) the first ICALP takes place. The Program Committee of C. Böhm, S. Eilenberg, P. Fisher, S. Ginzburg, G. Hotz, M. Nivat, L. Nolin, D. Park, M. Rabin, A. Salomaa, and A. van Wijngaarden is chaired by M.-P. Schützenberger.
The program includes 45 accepted papers (29 in English, 14 in French, 2 in German) on automata theory, theory of programming, theory of formal languages, and complexity of algorithms.

APPEL AUX COMMUNICATIONS

Théorie des Automates
des Langages
et de la Programmation
3-7 juillet 1972

IRIA

Programme of the first colloquium

First Bulletin

On December 1973, Maurice Nivat prepares the first Bulletin of EATCS at IRIA, Rocquencourt. The bulletin includes the minutes of the first general assembly and council meeting; reports on the second MFCS; and provides activity reports of the Mathematisch Centrum, Amsterdam, the Technological University, Delft, the Technological University, Twente, the Istituto di Scienza dell'Informazione, Università di Torino and the Institut de Programmation, Université Paris VI.

EATCS Awards

Awarded annually since 2000, this honours scientists from the community of Theoretical Computer Science in acknowledgment of their extensive and widely recognized contributions over a lifelong scientific career.

Richard Karp (2000), **Corrado Böhm** (2001), **Maurice Nivat** (2002), **Grzegorz Rozenberg** (2003), **Arto Salomaa** (2004), **Robin Milner** (2005), **Mike Paterson** (2006), **Dana S. Scott** (2007), **Leslie G. Valiant** (2008), **Gérard Huet** (2009), **Kurt Mehlhorn** (2010), **Boris (Boaz) Trakhtenbrot** (2011), **Moshe Y. Vardi** (2012), **Martin Dyer** (2013), **Gordon Plotkin**(2014), **Christos Papadimitriou** (2015), **Dexter Kozen** (2016), **Éva Tardos** (2017), **Noam Nisan** (2018), **Thomas Henzinger** (2019), **Mihalis Yannakakis** (2020), **Toniann (Toni) Pitassi** (2021), **Patrick Cousot** (2022)

Further reading

U. Brauer & W. Brauer. "Silver Jubilee of EATCS." *Bulletin of the EATCS* **62**:3–23, 1997.

G. Ausiello. *The Making of a New Science.* Springer, 2018.

G. Ausiello. "EATCS Golden Jubilee: How EATCS was born 50 years ago and why it is still alive and well." *Bulletin of the EATCS* **137**, 2022.

1937–2017

Maurice Nivat

A founding father of Theoretical Computer Science

As a mathematician he applied rigorous algebraic approaches to numerous domains, from formal languages to program semantics, from concurrent processes to discrete geometry.
As a scientific leader he undertook with incredible energy the mission of promoting study and research in the theory of computing.

■ Early years

■ 1937
Born in Clermont-Ferrand, France

Maurice (right) with siblings and Grandma

■ 1956
Enters *École Normale Supérieure*; his broad mindedness and originality flourish and he is the leader of a group of merry fellows which calls itself "Praesidium du Bordel Suprême"; he gets married and has his first son while still at ENS

■ 1959
Begins work at Institut Blaise Pascal and gets acquainted with computers and programming languages

■ 1969
Becomes professor at Université de Paris

Maurice, 20 years old

■ Founding the EATCS

Nivat (left) with Schützenberger at ICALP'72

■ 1971
With Louis Nolin and Marcel-Paul Schützenberger, presents a "charter" of theoretical computer science, called *Rapport préliminaire sur l'Informatique Théorique*; proposes to establish a collaboration with the main European universities and research centers

■ 1972
- Organises the first International Colloquium on Automata, Languages and Programming (ICALP)
- Organises with Alfonso Caracciolo the Brussels meeting where the creation of the EATCS is approved

■ 1973
Elected President of EATCS and edits *the first Bulletin of the EATCS*; founds the journal *Theoretical Computer Science*

Fostering French TCS

1973
Initiates the yearly *École de Printemps d'Informatique Théorique* bringing together younger researchers in pleasant historical places throughout France to learn about a topic in Theoretical Computer Science

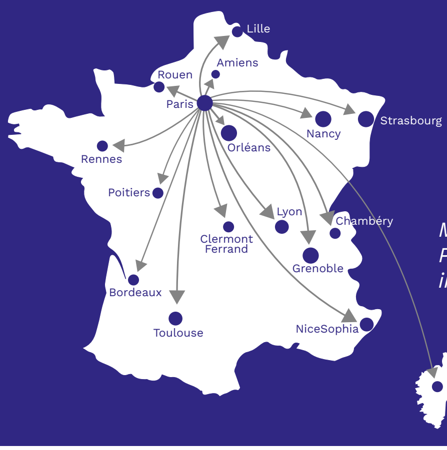
1975
Founds the *Laboratoire d'Informatique Théorique et Programmation* (LITP), of which the *Institut de Recherche en Informatique Fondamentale* (IRIF) that organises ICALP'22 is a descendant

1992
Founds the *Association Française d'Informatique Théorique*, the French arm of EATCS

Decoration by Minister of Research Hubert Curien, 2002

Scientific Legacy

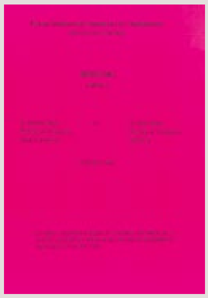
Maurice Nivat worked on many subjects: transductions, language theory, algebraic semantics, semantics of concurrency, infinite words, tilings... In each one, he had seminal ideas, and was able to direct his numerous students to the best-suited domains.



Maurice's former PhD students in France

Computer Science in Education

Throughout his career, Maurice fought for the introduction of computer science in education. His wish was finally fulfilled in 2021: an Agrégation d'Informatique (i.e., a contest to select Computer Science professors for high schools) was created.



1983 report on computer science education

Further reading

P.-L. Curien. "Une brève biographie scientifique de Maurice Nivat." Theoretical Computer Science **281**(1–2):3–23, 2002.

I. Bellin. "Maurice Nivat, Une vision à long terme de la recherche en informatique." *Interstices*, 2008.

algorithm
graph
problem
complexity
language
approximation
time
bound
abstract
automata
optimal
polynomial
extended
tree
system
network
finite
model
computation
lower
power
theory
logic
program
application
set
matching
regular
complete
random
bounded
game
theorem
path
process
equivalence
function
communication
programming
data
machine
parallel
information
quantum
semantic
approximate
characterization
linear
weighted
grammar
probabilistic
deterministic
constraint
scheduling
improved
testing
simple
order
structure
distance
space
boolean
computing
context
scheme
analysis
property
calculu
string
verification
pushdown
knowledge
two
one
algebra
markov
sampling
branching
rewriting
dynamic
search
recursive
stochastic
proof
free
type
net
matrix
number
counting
randomized
sparse
learning
sublinear
new
first
succinct
concurrent
cost
method
cycle
local
result
tight
circuit
planar
protocol

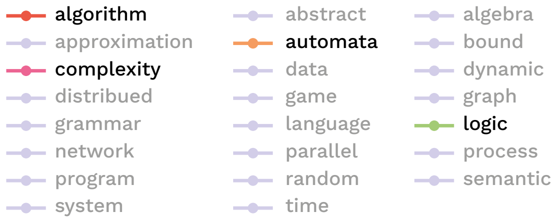
ICALP Through Time

Mining publications data

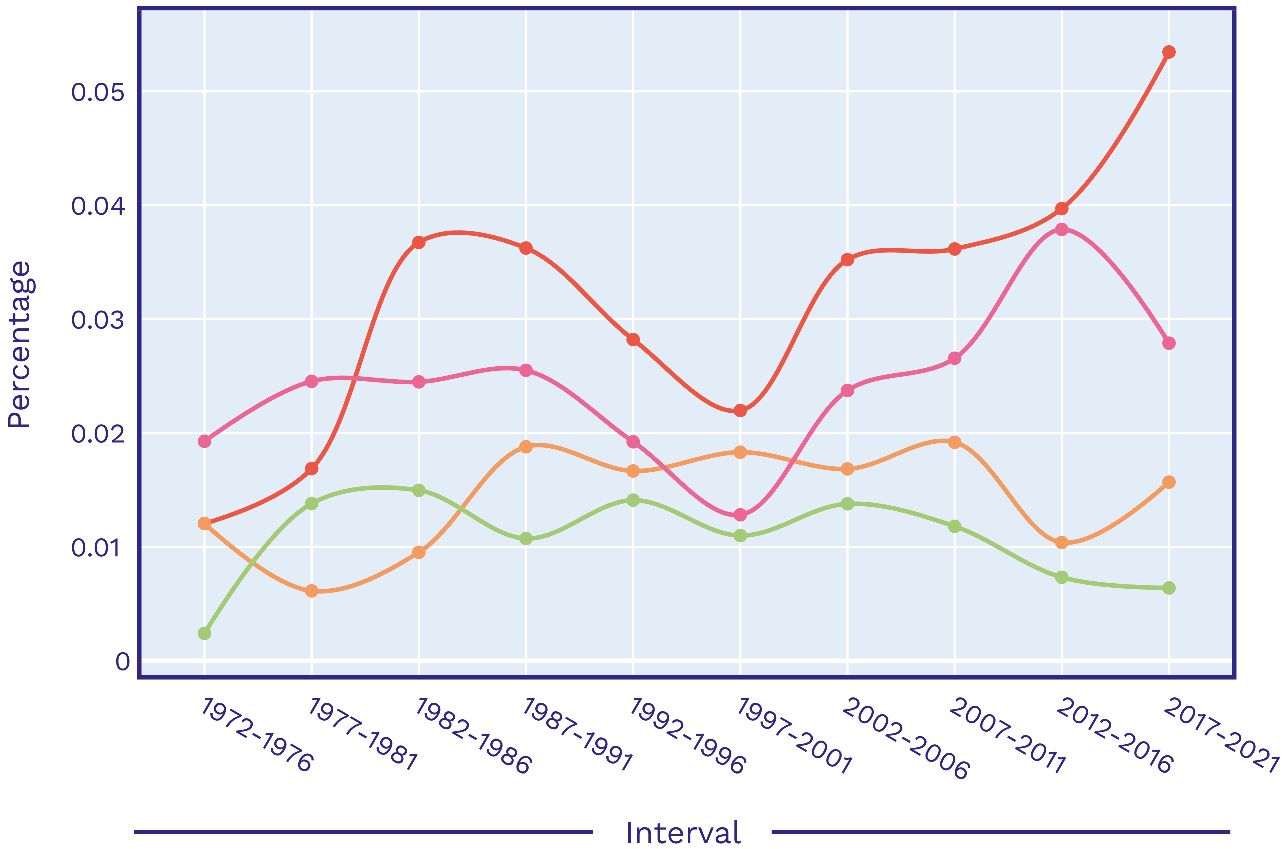
In the 50 years since its inception, the ICALP conference has evolved in pace with the scientific advances and the growth and maturation of the Theoretical Computer Science community. This poster, based on an analysis of DBLP data, provides a bird's eye view of that evolution.

ICALP Topics

Throughout its fifty-year history, ICALP has provided a broad coverage of topics in Theoretical Computer Science. How has the relevance of research topics within the theoretical-computer-science community changed since 1972?

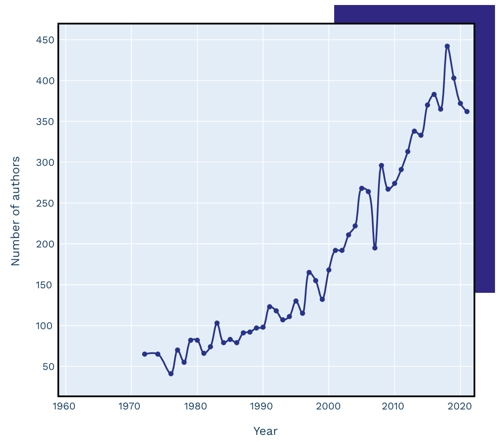


Percentage of ICALP papers whose titles mention the word algorithm, complexity, automata, or logic.

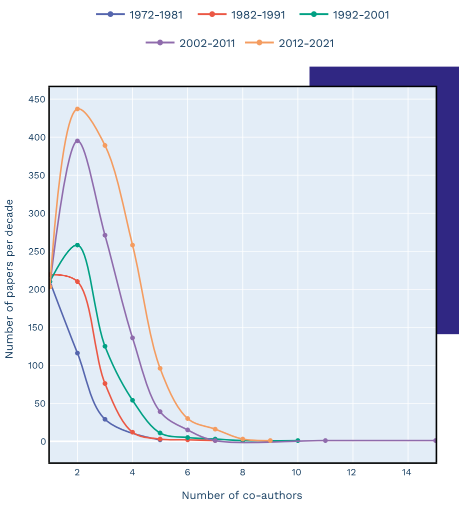


■ ICALP Authorship

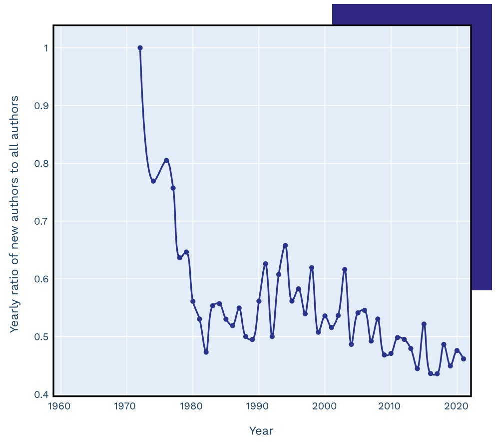
Like other major conferences in Theoretical Computer Science, the authorship at ICALP tends to stabilise over time.



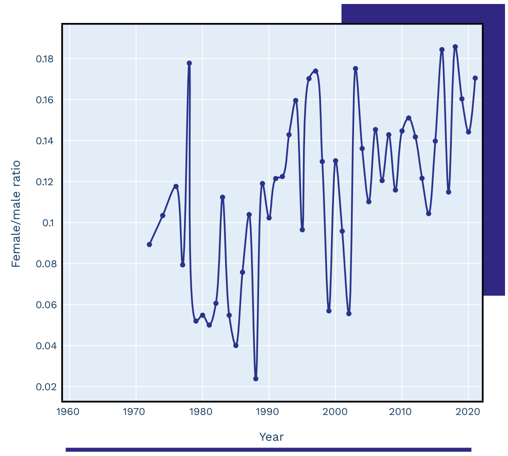
Number of authors per year, corresponding to a similar evolution in the number of accepted papers.



Number of papers with each co-authorship size, per decade. Papers with two, three, and even four authors have gradually become more common than single-author papers.



Percentage of new authors per year. Every year, approximately half the authors at TCS conferences are newcomers to the conference.

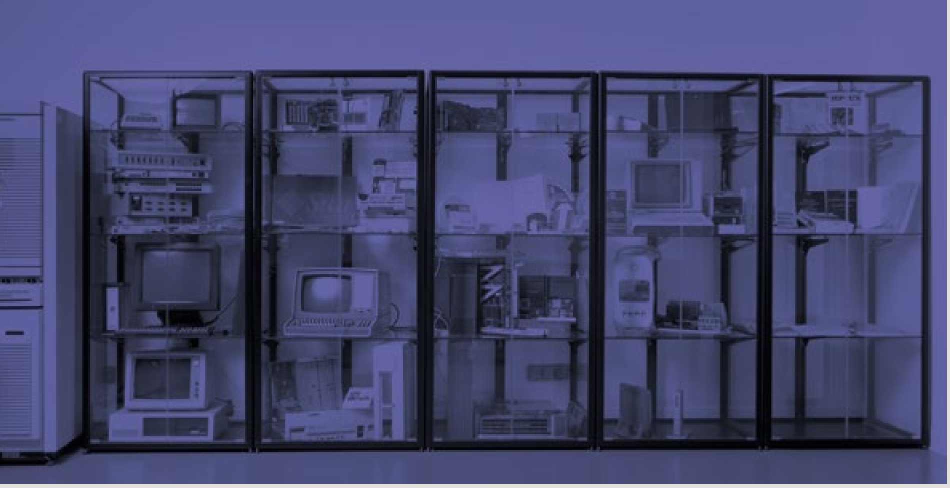


Ratio of women over men among authors. Still below 0.2 for almost all TCS conferences in 2021.

Further reading

P. Crescenzi. "Celebrating 50 years of ICALP: A data and graph mining analysis."
https://slides.com/piluc/icalp-50?token=fl3BBJ8j

Link to the full data analysis




HIGHL


odd-girth
SPC(5)

IGHTS

dynamic
Distributed Hash Tables
The boosting algorithm
in machine learning
(due to Schapire)

Ford
Fulkerson
Max-Flow

Weighted Fair Queueing Balanced Trees
Gaussian
Metropolis Algorithm Matching algorithms
Elimination
Ford-Fulkerson Max-Flow, Edmonds-Karp
Balanced Trees
Sorting, Quicksort
quantum
computing
FFT LLL Page rank
Linear Programming
Dynamic Programming
Interior Point, Simplex online matching Dijkstra
Cole Vishkin algorithm (mostly theoretically) Heaps Signal Processing
Simulated Annealing online algorithms Balanced Trees Edmonds-Karp
Counting algorithm with Hyperloglog
algorithm longest common subsequence online
Shor's factoring algorithm programming Gaussian
Elimination
ranking Randomized Incremental Construction in machine learning
Brozozowski DFA minimisation algorithm Distributed Hash Tables
belief propagation Lamport Clocks (again not sure if it counts) Matching algorithms
All APSP allgorithms The Berlekamp-Welch error correction
algorithm, error correction rsa approximate TSP
quantum computing genetic algorithms Lovasz lattice fides for
Matching algorithms Signal Processing and Communication: Christo
bipartite FFT and Viterbi (special case of dynamic
general programming)
deep learning codes
trees
Greedy
rsa

fides for
approximate TSP
Christo codes
Gaussian
Elimination
ant colony
systems
genetic algorithms Lempel-Ziv
matrix multiplication
Distributed Hash Tables
codes algorithms, e.g.
Euclidean Algorithm
Gale-Shapley
Error-Correcting Codes
Approximation

Merkle
Trees algorithm
in Reed-Solomon
Paxos
Dynamic
Programming
fides for
approximate TSP
codes Lovasz
Depth-First-Search
Balanced Trees
Sorting, Quicksort
Page rank
Hashing
Dijkstra
LLL
Heaps
FFT

Algorithms that Shaped the World

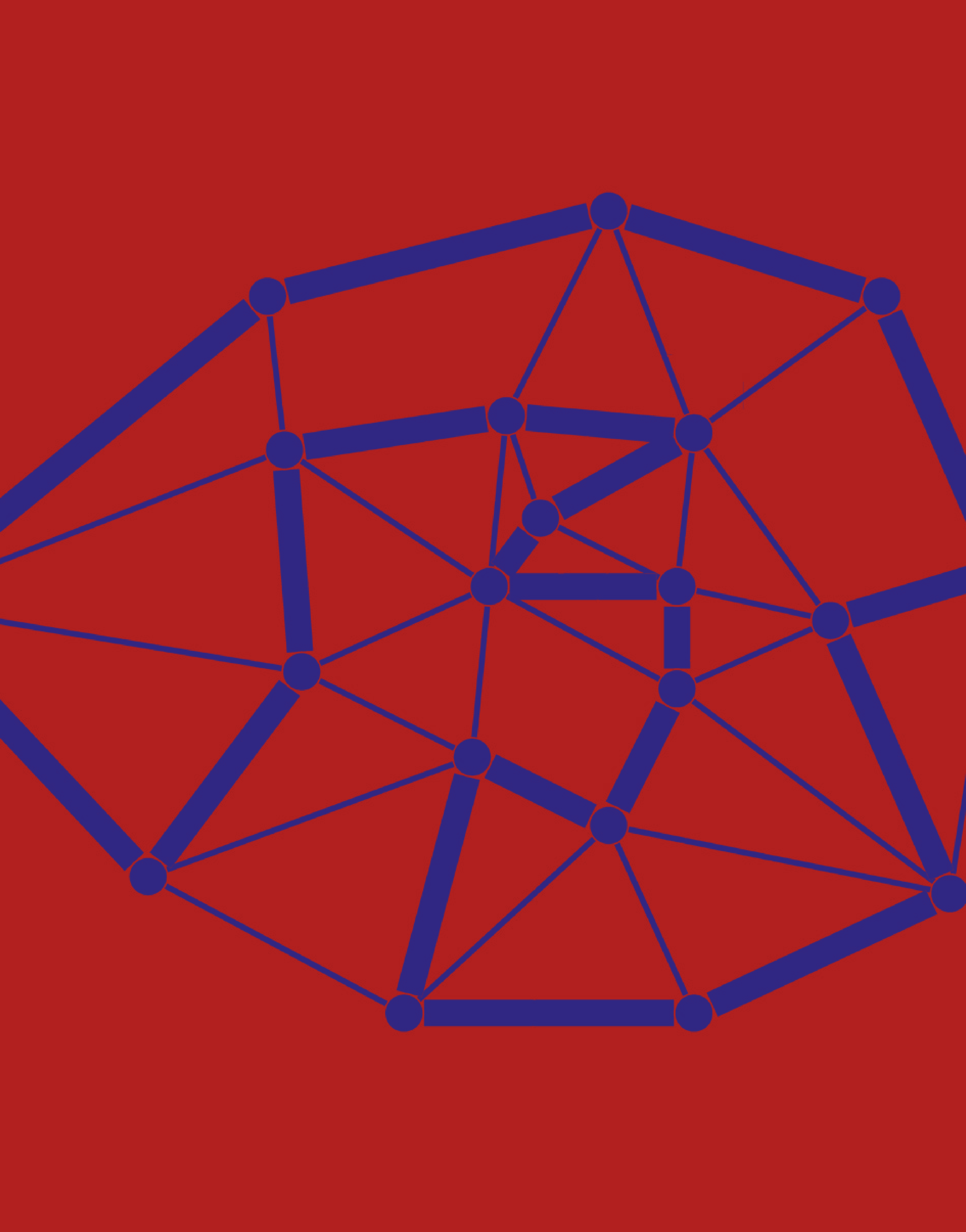
Algorithms are the hearts of computing systems. They are usually not visible to the user, but they keep the systems going and provide functionality and speed. Without algorithms there would be no systems. Not surprisingly, every computer scientist is taught algorithms. The design and analysis of algorithms is a subject of intellectual depth and beauty with wide-ranging impact on the real world.

Further reading

K. Mehlhorn. *Data Structures and Algorithms,* volumes 1, 2, and 3. EATCS Monographs on Theoretical Computer Science, Springer, 1984.

K. Mehlhorn & S. Näher. *LEDA: A Platform for Combinatorial and Geometric Computing.* Cambridge University Press, 1999.

T. H. Cormen, C. E. Leiserson, R. L. Rivest, & C. Stein. *Introduction to Algorithms,* 3rd Edition. MIT Press, 2009.



Computational Complexity

Classifying problems by hardness

In the 1930's, Church, Turing and others proposed the "right" notion of algorithm and studied what is recursive, i.e., what can be solved at all by computers. Later, with the first computers, the efficiency of algorithms became crucial. Computational complexity was born.

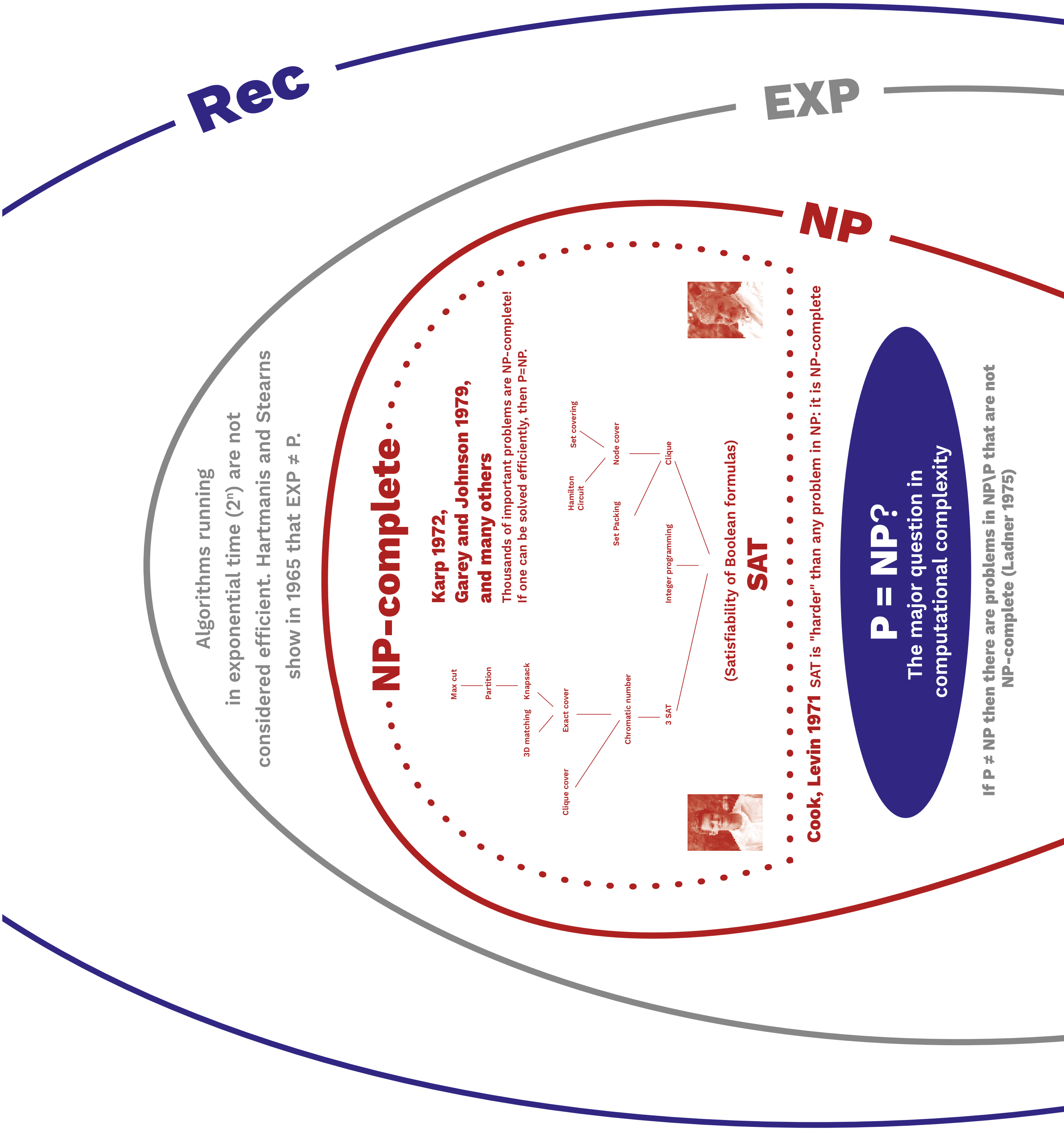
Further reading

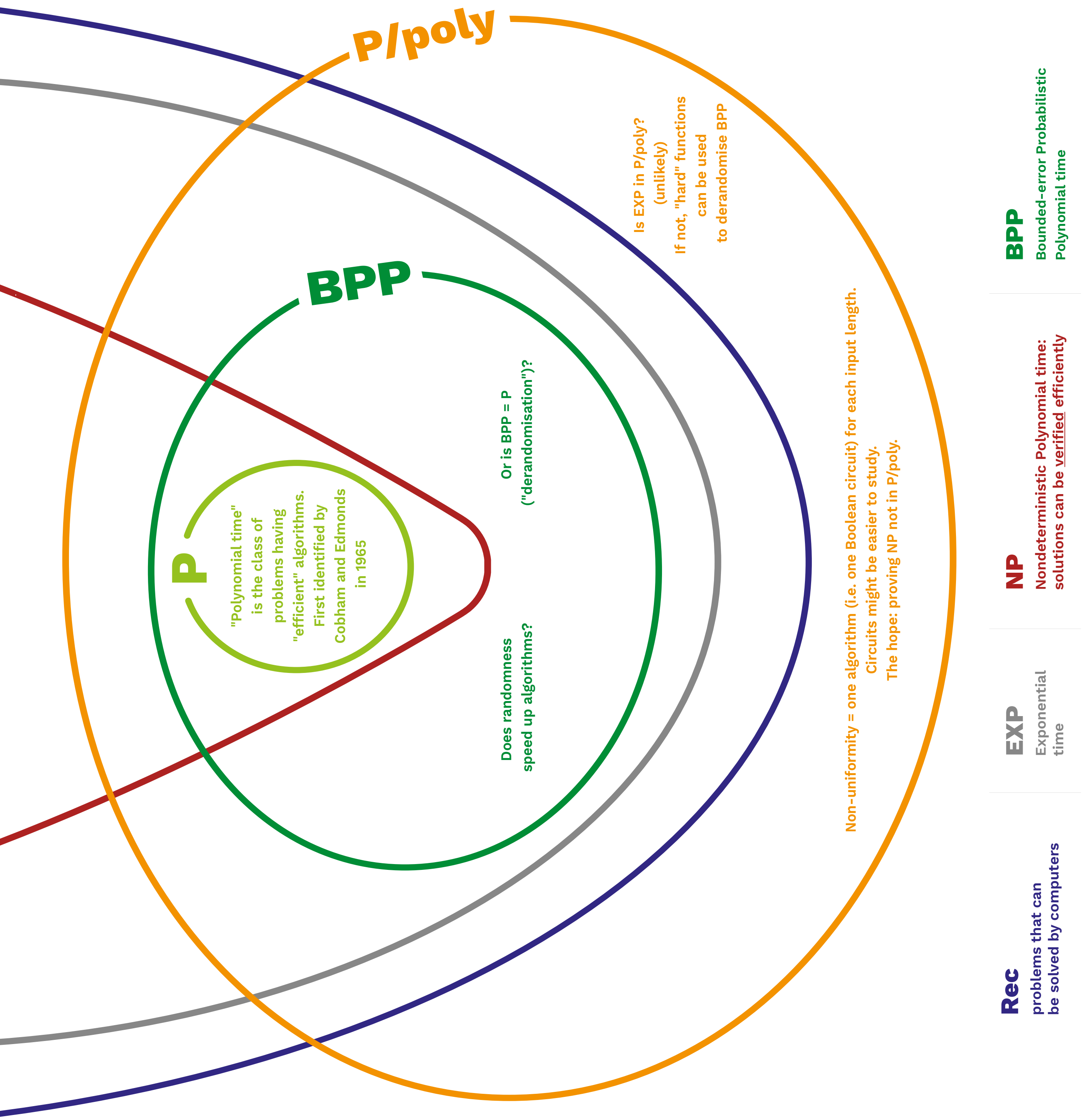
S. A. Cook. "The complexity of theorem proving procedures." *Proceedings of STOC'71.* ACM, 1971.

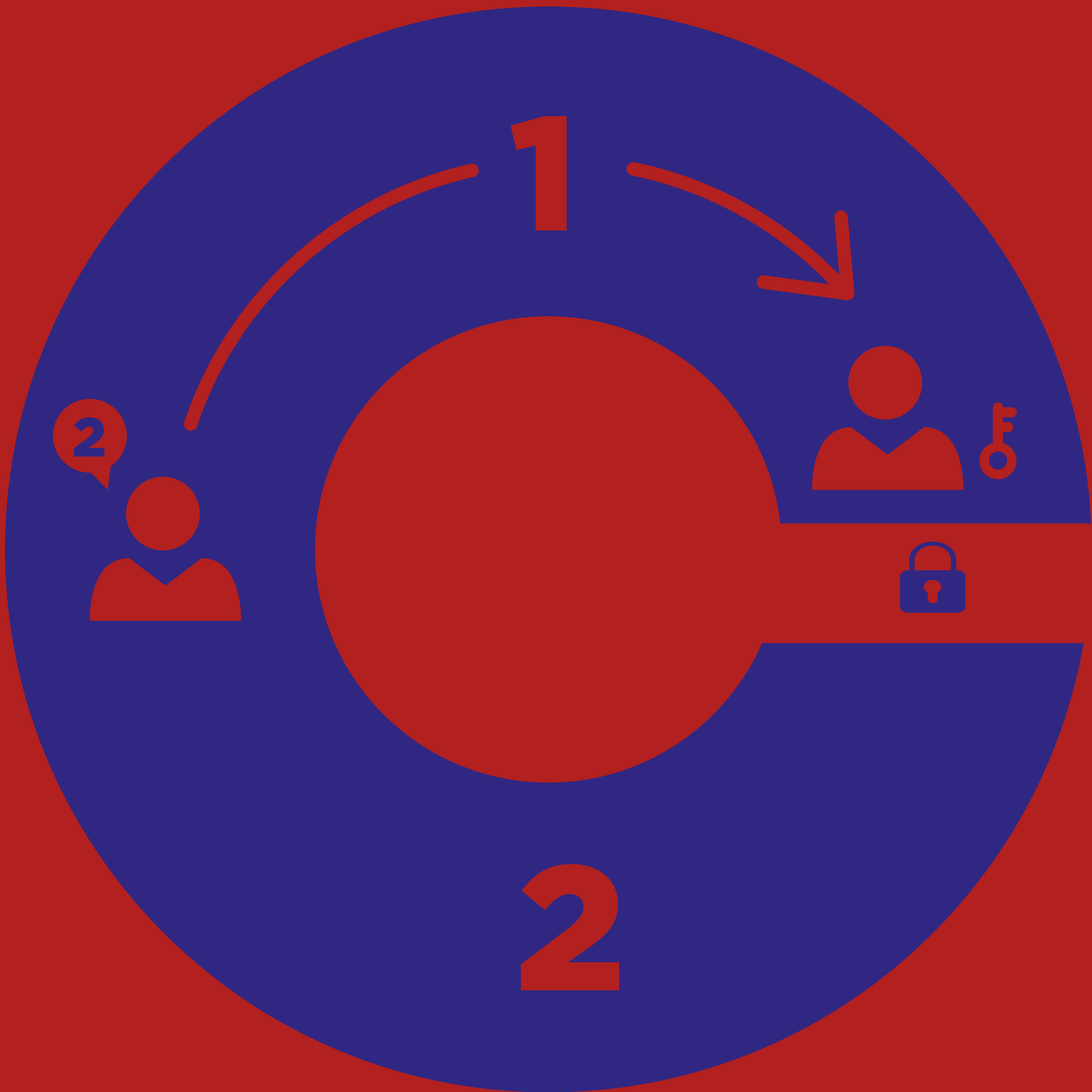
L. A. Levin. "Universal search problems." *Problems of Information Transmission,* **9**(3):265–266, 1973.

R. M. Karp. "Reducibility Among Combinatorial Problems." *Complexity of Computer Computations.* Springer, 1972.

S. Arora & B. Barak. *Computational Complexity. A Modern Approach.* Cambridge University Press, 2009.


Rec
EXP
NP
Algorithms running in exponential time (2^n) are not considered efficient. Hartmanis and Stearns show in 1965 that EXP ≠ P.
NP-complete
Karp 1972, Garey and Johnson 1979, and many others
Thousands of important problems are NP-complete! If one can be solved efficiently, then P=NP.
Max cut
Partition
Knapsack
3D matching
Exact cover
Clique cover
Chromatic number
3 SAT
Integer programming
Set covering
Hamilton Circuit
Node cover
Set Packing
Clique
(Satisfiability of Boolean formulas)
SAT
Cook, Levin 1971 SAT is "harder" than any problem in NP: it is NP-complete
P = NP?
The major question in computational complexity
If P ≠ NP then there are problems in NP\P that are not NP-complete (Ladner 1975)


P/poly
Is EXP in P/poly?
(unlikely)
If not, "hard" functions
can be used
to derandomise BPP
BPP
Or is BPP = P
("derandomisation")?
Does randomness
speed up algorithms?
P
"Polynomial time"
is the class of
problems having
"efficient" algorithms.
First identified by
Cobham and Edmonds
in 1965
Non-uniformity = one algorithm (i.e. one Boolean circuit) for each input length.
Circuits might be easier to study.
The hope: proving NP not in P/poly.
BPP
Bounded-error Probabilistic
Polynomial time
NP
Nondeterministic Polynomial time:
solutions can be verified efficiently
EXP
Exponential
time
Rec
problems that can
be solved by computers


1
2

Zero-Knowledge Proofs

Showing that a problem has a solution without revealing it

Is it possible to demonstrate that we know how to prove a theorem, but without disclosing the proof? Surprisingly, the answer turns out to be "yes." This result, discovered in the 80's, had a profound impact on our understanding of privacy, and opened the floodgates of a myriad of applications in cryptography and computer security.

The Origin

1985
Goldwasser, Micali, and Rackoff introduced the notion of zero-knowledge proofs: proofs that yield no information beyond the validity of the statement.

1986
Goldreich, Micali, and Wigderson, showed the wide applicability of this concept: they demonstrated that, under widely believed assumptions, any theorem whose proof can be verified efficiently also admits a zero-knowledge proof.

An example

Imagine a network of radio towers that can emit at three different frequencies. To avoid interference, we want that two nearby towers always emit at a different frequency. In general, determining whether this task can be achieved is a hard combinatorial problem. Dodgy is an agency that claims to have a solution (a setting of the frequencies), wishes to sell it to an operator and will only reveal its frequency setting after it has been paid. The operator, Towergrid, is suspicious and wants to be convinced that Dodgy really knows a solution before paying.

The paper of Goldreich, Micali, and Wigderson gives a nice solution to the above conundrum.

1 - Dodgy chooses random names for the frequencies, e.g., A,B, and C.

2 - Dodgy puts the name of the chosen frequency for each tower in a “cryptographic box.”

3 - Towergrid then asks Dodgy to open two randomly chosen boxes for nearby towers.

4 - Towergrid checks whether the letters are indeed different.

After enough repetitions of steps 1 to 4, any cheater is guaranteed to be caught (with whatever probability of error Towergrid likes to achieve), but the solution is never revealed.

This radio-tower problem described above is well-known to be “NP-complete.” In essence, this means that by finding a zero-knowledge proof for this problem, Goldreich, Micali, and Wigderson have in fact found a zero-knowledge proof for all problems with efficiently-verifiable proofs!

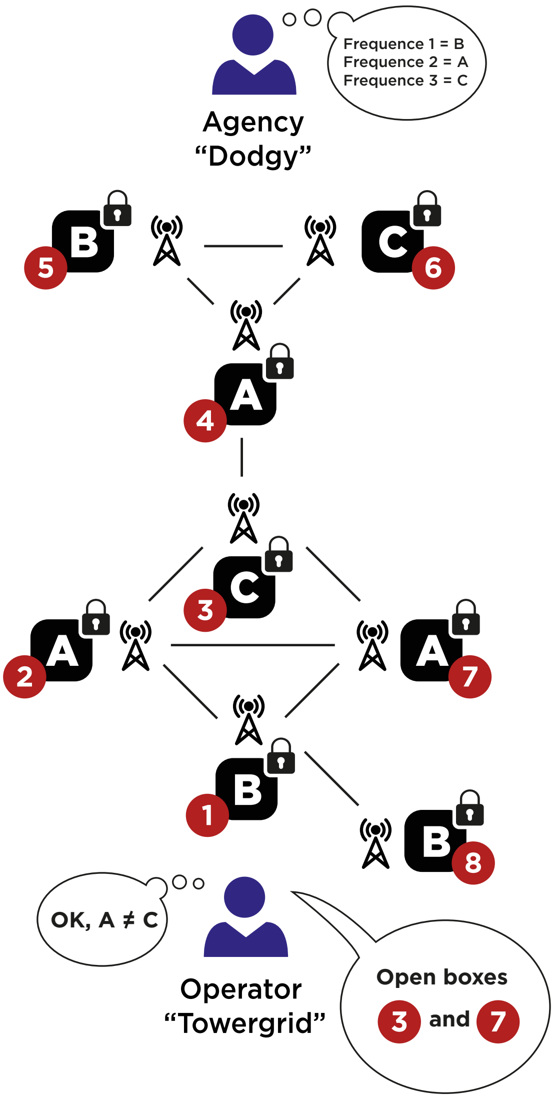
Further reading

S. Goldwasser, S. Micali, & C. Rackoff. “The knowledge complexity of interactive proof systems.” *Proceedings of STOC’85.* ACM, 1985.

O. Goldreich. *Foundations of Cryptography,* Volume 1. Cambridge University Press, 2008.

M. Green. “Zero Knowledge Proofs: An illustrated primer.” *A Few Thoughts on Cryptographic Engineering,* 2014.

The radio-tower problem

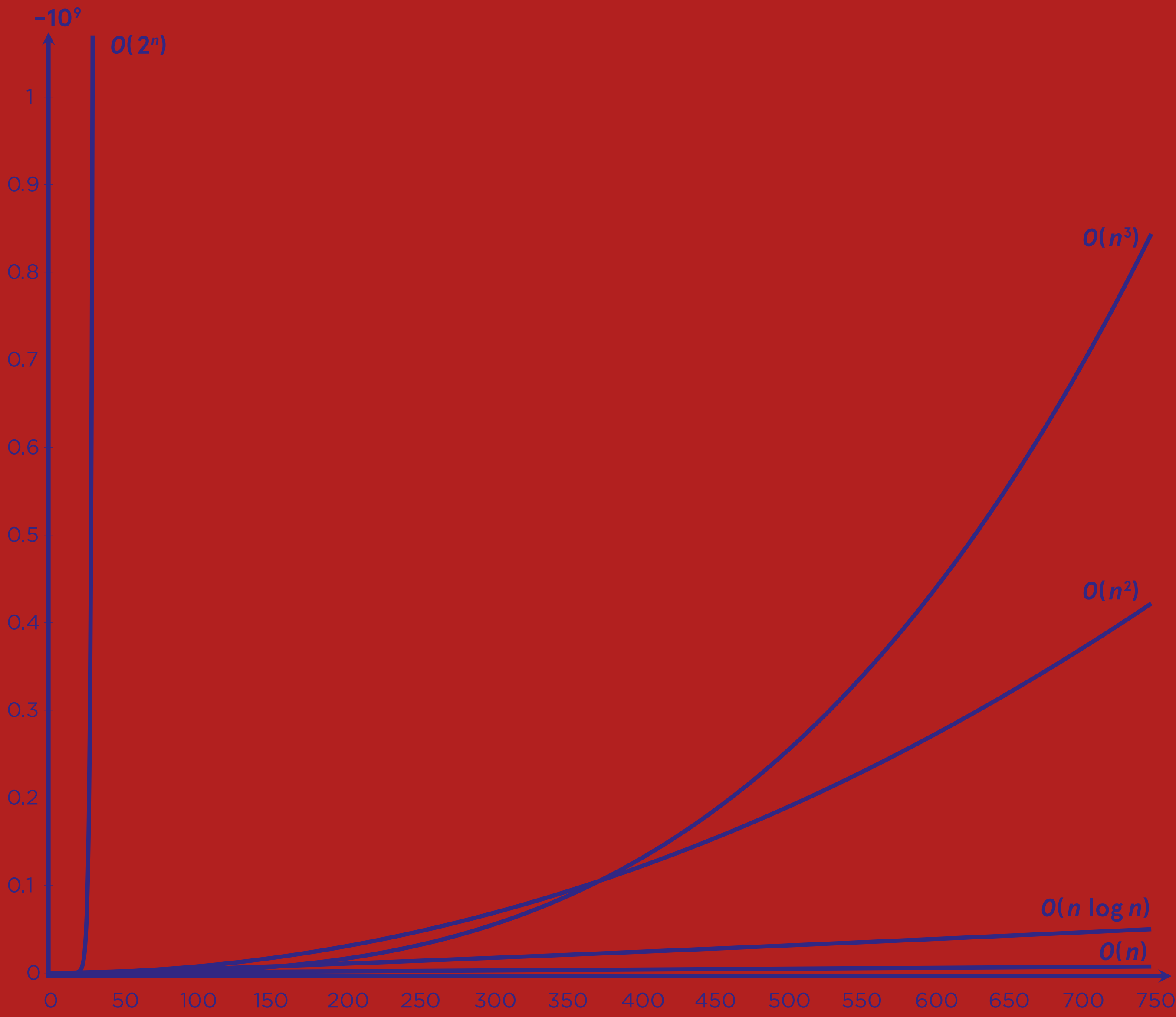


Impact

37 years laters, zero-knowledge proofs have revolutionised cryptography.

They enable powerful authentication and verification mechanisms: any user can demonstrate possession of an appropriate credential, or execution of an appropriate procedure, without revealing any of the private information (personal data, passwords, cryptographic keys) used in the process.

They are a core component in blockchain or in electronic voting, and are routinely used by banks and companies in the finance sector.



~10⁹
O(2ⁿ)
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
O(n³)
O(n²)
O(n log n)
O(n)
0
50
100
150
200
250
300
350
400
450
500
550
600
650
700
750

Fine-Grained Complexity

A way to prove exact time bounds

For any computational problem, the two most important factors for designing an algorithm are its efficiency and optimality. However, one of the major challenges in complexity theory has been the inability to prove unconditional time lower bounds. Nevertheless, we would like to provide evidence that say a problem A with a running time $T(n)$ that has not been improved in decades, also requires $T(n)^{1-o(1)}$ time, thus explaining the lack of progress on the problem. Unfortunately, such unconditional time lower bounds seem very difficult to obtain. Towards that, the area of fine-grained complexity has been developed.

What is Fine-Grained complexity theory?

Fine-Grained complexity theory is based on fine-grained reductions that focus on the exact running times for computational problems. The techniques mimic the idea of proving NP-hardness for problems, except that in the latter case we don't care about the exact hardness. Over decades, using fine-grained reductions, many meaningful relationships between problems in the classical setting have been made. More recently, similar connections gave been explored in the quantum setting as well.

The Approach

The approach is:

1 — To select a key problem X that for some function T, is conjectured to not be solvable by any $O(T(n)^{1-\varepsilon})$ time algorithm for $\varepsilon > 0$, and

2 — To reduce X in a fine-grained way to many important problems, thus giving (mostly) tight conditional time lower bounds for them.

Some of the key problems for example are the CNF-SAT problem, the 3-SUM problem, and the All Pairs Shortest Paths Problem (APSP).

GLOSSARY

CNF-SAT
Given a Boolean formula in its conjunctive normal form on n variables, is there an assignment to these variables such that the formula evaluates to true?

3-SUM
Given a list of n integers, is there a triple a, b, c in the list such that a + b + c = 0?

All Pairs Shortest Path
Given a graph of n nodes with weighted edges, output the shortest path between all the pairs of nodes in the graph.

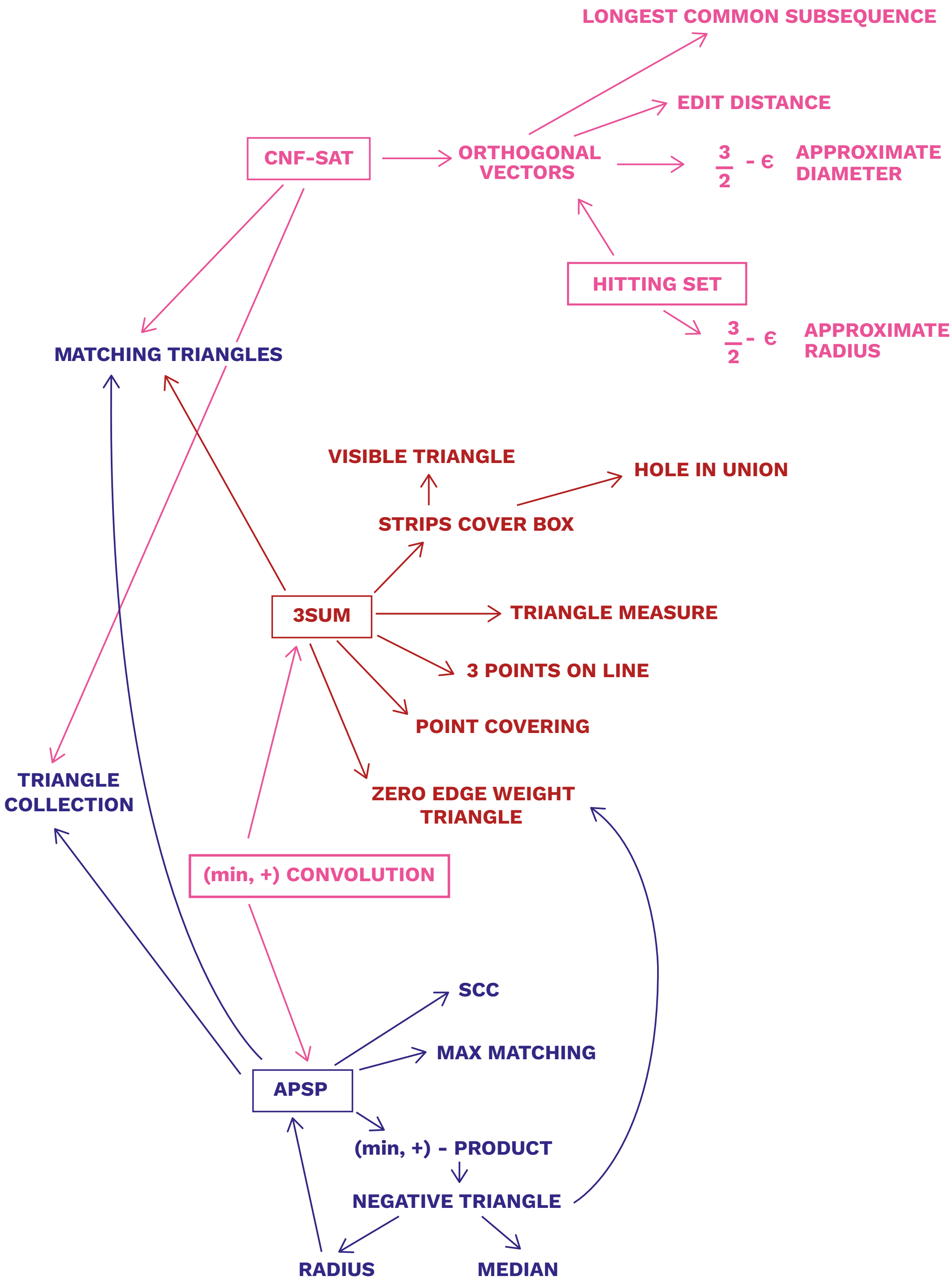
Further reading

V. V. Williams. "Hardness of easy problems: Basing hardness on popular conjectures such as the Strong Exponential Time Hypothesis." *Proceedings of IPEC'15.* LIPIcs **43**, LZI, 2015.

V. V. Williams & R. R. Williams. "Subcubic equivalences between path, matrix, and triangle problems." *Journal of the ACM* **65**(5):27, 2018.

M. Patrascu. "Towards polynomial lower bounds for dynamic problems." *Proceedings of STOC'10.* ACM, 2010

■ Some key problems and their Fined-Grained reductions



Logic and Computational Complexity

A Perfect match

The unity of logic and computation has manifested itself in the development of computability theory from the 1930s onward and the development of computational complexity from the 1960s onward. Computability theory delineates the boundary between decidability and undecidability. Computational complexity delineates the boundary between tractability and intractability. Logic provides prototypical complete problems for complexity classes and led to descriptive complexity, a framework for characterising complexity classes using logical resources.

Complete problems

1936
Church-Turing Theorem
First-Order Validity is computably enumerable (c.e.)-complete.

1949
Trakhtenbrot's Theorem
First-Order Finite Satisfiability is computably enumerable (c.e)-complete.

1971
Cook-Levin Theorem
SAT is NP-complete.

Descriptive complexity

1974
Fagin's Theorem
NP = ESO. In words, a decision problem Q is in NP if and only if Q is expressible in existential second-order logic ESO.

"machine-free characterisation of NP with no mention of polynomial"

Example: SAT is definable by the ESO-formula

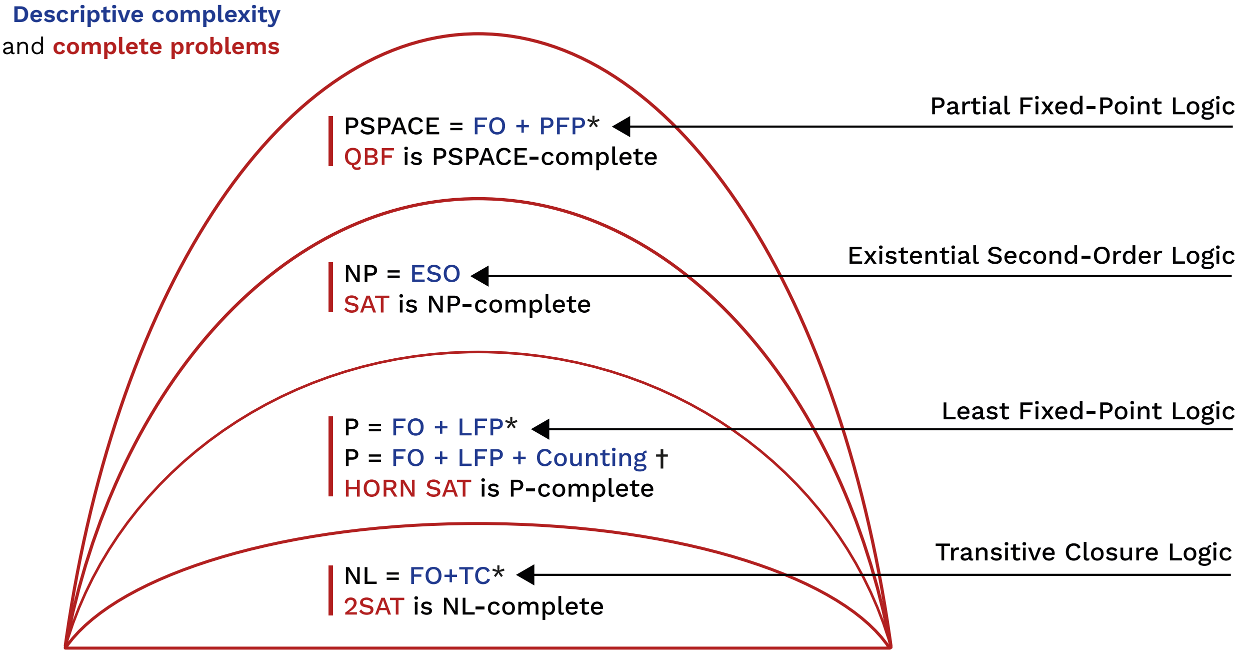
$$\exists S\,\forall c\,\exists v\big((P(c,v)\wedge S(v))\vee(N(c,v)\wedge\neg S(v))\big)$$

1982
Immerman-Vardi Theorem
P = FO+LFP on classes of ordered finite structures.

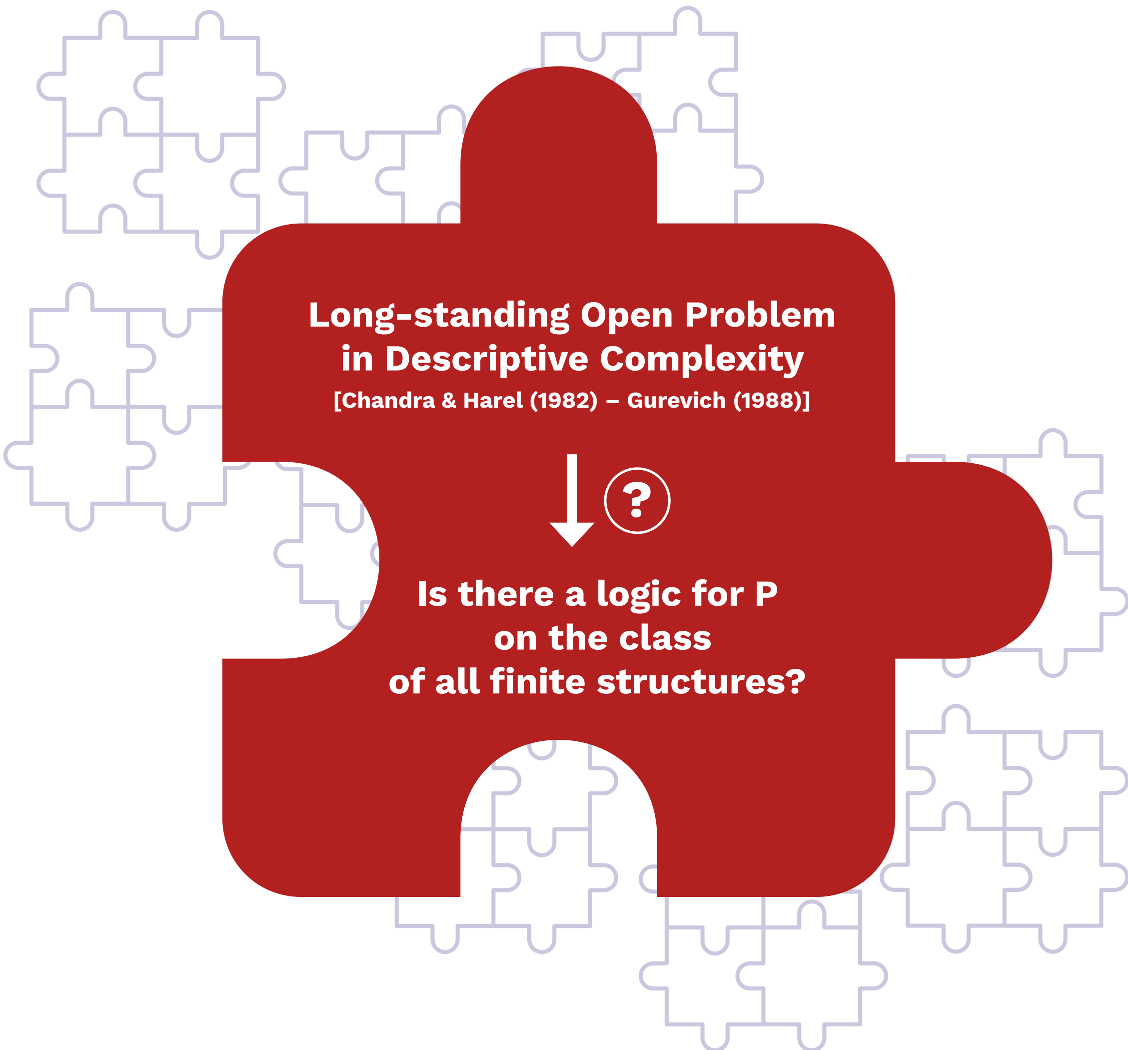
2010
Grohe's Theorem
If **C** is a class of graphs with at least one excluded minor, then on **C**

P = FO + LFP + Counting.

Key Property: Linear order definable in FO + LFP + Counting on C.



*on classes of *ordered* finite structures
† on classes of finite structures *excluding at least one minor*

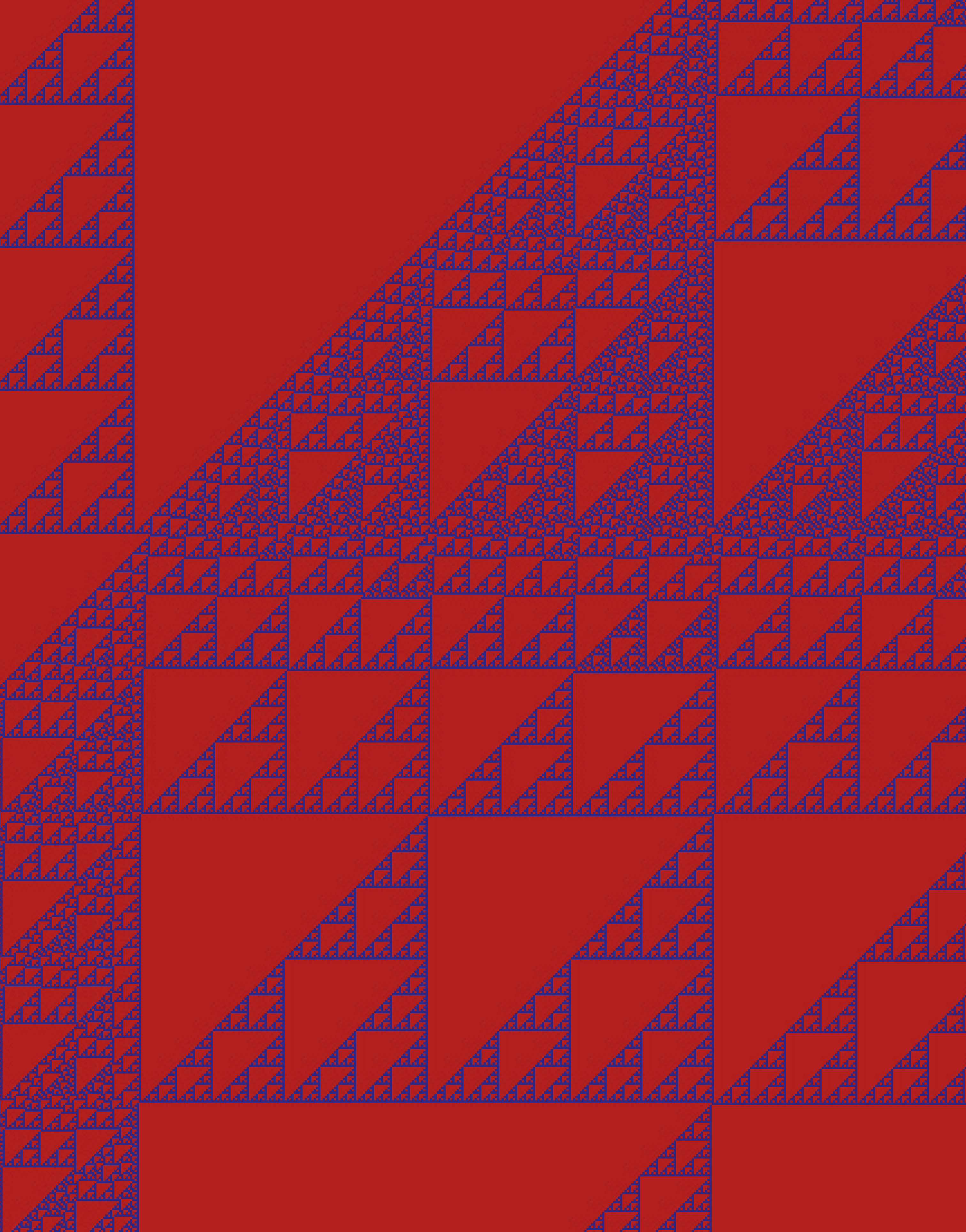


Further reading

R. Fagin. "Generalized first-order spectra and polynomial-time recognizable sets." *Complexity of Computation* **7**:43–73. SIAM-AMS, 1974

N. Immerman. *Descriptive Complexity.* Texts in Computer Science, Springer, 1999.

E. Grädel, P. G. Kolaitis, L. Libkin, M. Marx, J. Spencer, M. Y. Vardi, Y. Venema, & S. Weinstein. *Finite Model Theory and its Applications.* Texts in Theoretical Computer Science, An EATCS Series, Springer, 2007.



Automata Theory

Abstract machines and their computational power

Automata Theory is one of the oldest research areas in Computer Science. Historically, it developed with the theory of formal languages, since automata were categorised by the classes of languages they can recognise. Today, automata-based formalisms are widely applied in modern computing. Indeed, every computing device has "automata inside!"

What is Automata Theory?

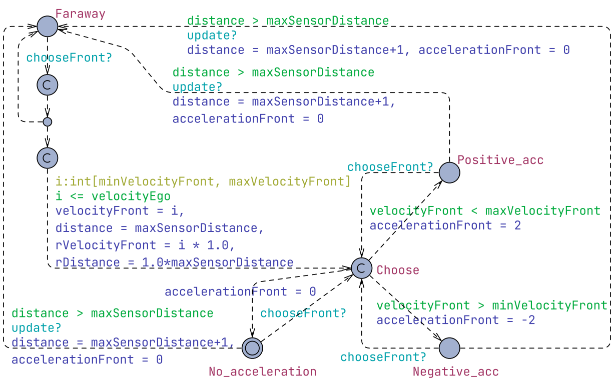
Automata Theory is a research area that is concerned with the study of abstract computing devices and of their computational power.
It emerged from A. Turing's study of the power of general-purpose computation and from S.C. Kleene's formalisation of an earlier proposal by McCulloch and Pitts that was motivated by the study of networks of neurons.

Connections with mathematics

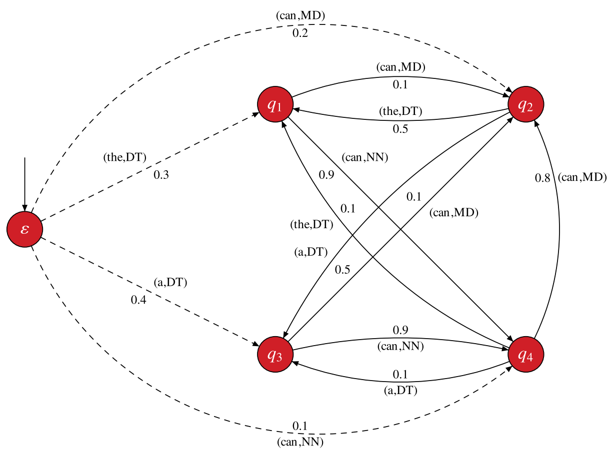
Automata Theory uses increasingly sophisticated mathematical techniques to study the power of abstract computational devices.
It has close connections with classic and novel fields of Mathematics such as group theory and the theory of algebraic structures, logic, (finite) model theory, number theory, (automatic) real function theory, symbolic dynamics, and topology.

Where is Automata Theory used in computer science?

The short answer is that automata are everywhere in Computer Science! Initially, their study was motivated by, and had immediate application in, fields such as computer design, compilation of programming languages, and search and pattern matching. Their use then spread across the whole field.



An automaton describing the behaviour of a car driving in front of an autonomous vehicle as a player in a stochastic priced timed game. The tool Uppaal Stratego can be used to synthesise winning strategies in such games.



A weighted word automaton for part-of-speech tagging in English.

Selected key Milestones in Automata Theory

1936 **A. Turing:** Turing machines

1943 **W. McCulloch, W. Pitts:**
Nerve nets as finite automata

1948 **J. von Neumann:**
The general and logical theory of automata

1951 **S.C. Kleene:**
Regular expressions, Kleene's Theorem

1955 **M.P. Schützenberger:**
Algebraic theory of automata: Syntactic semigroup and variable-length codes

1956 **E.F. Moore:**
Minimal automata

1957 **J. Myhill:**
Non-deterministic automata and determinisation.

1958 **A. Nerode:**
Nerode equivalence

J.R. Büchi, C.C. Elgot, B.A. Trakhtenbrot:
Finite automata and monadic second-order logic (MSO)

1959 **M.O. Rabin, D. Scott:**
Finite automata and their decision problems

1963 **N. Chomsky, M.P. Schützenberger:**
Context-free languages and pushdown automata

1965 **M.P. Schützenberger:**
Star-free expressions and group-free monoids

K. Krohn and J. Rhodes:
Decomposition of automata

1969 **M.O. Rabin:**
Automata on infinite trees and MSO

1982 **Y. Gurevich, L. Harrington:**
Trees, automata and games

W. Thomas:
Classifying regular events in symbolic logic

1988 **N. Immerman, R. Szelepszenyi:**
Complementation of linear bounded automata

K. Hashiguchi:
Solution of the restricted star-height problem

Further reading

J.-É. Pin (Ed.). *Handbook of automata theory.* Volumes I and II. European Mathematical Society, 2021.

H.Straubing. *Finite automata, formal logic, and circuit complexity.* Progress in Theoretical Computer Science. Birkhäuser, 1994.

Model Checking

Proving system correctness, automatically

One of the goals of computing as a whole is to develop computing systems that perform the tasks they were designed to do in a reliable manner. Model checking is an area of research in Theoretical Computer Science that has had huge impact on achieving that difficult goal.

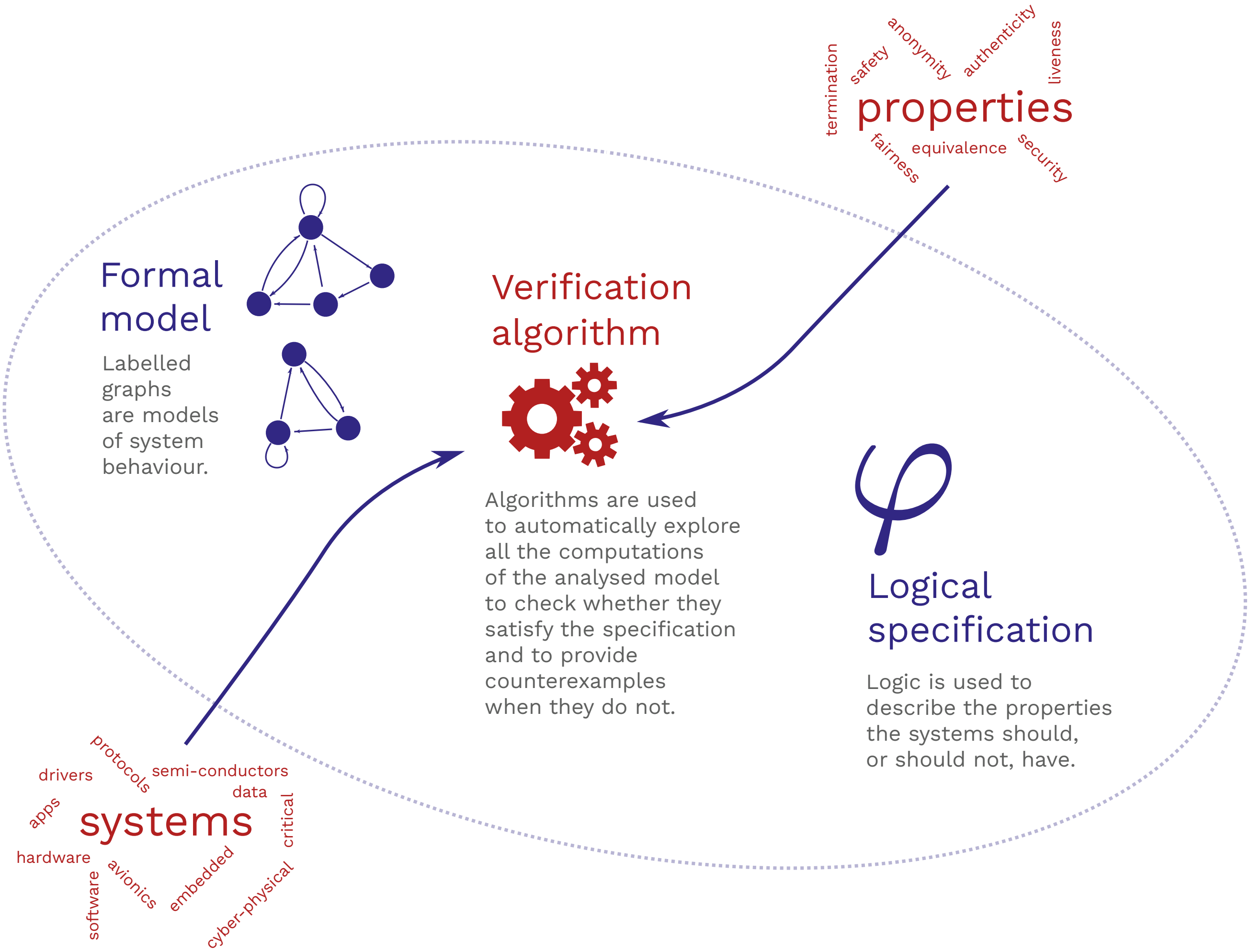
What is Model Checking?

Examples of systems

- critical systems
- security protocols
- data

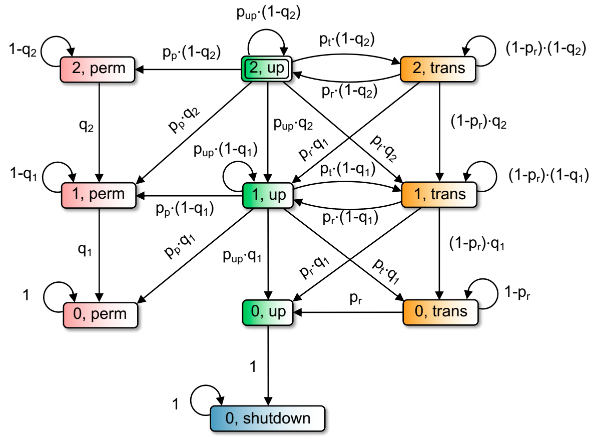
Examples of properties

- bugs
- termination
- privacy

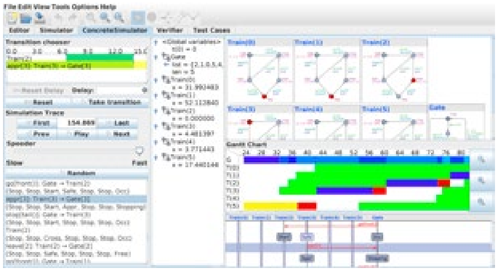


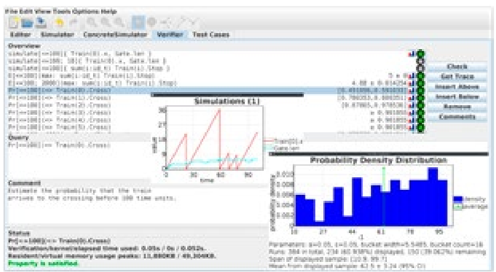
What does the checking?

Software tools carrying out this analysis are called model checkers and have been used to find and fix bugs in many mission-critical hardware and software systems, in program synthesis, and in optimal scheduling among many other applications. Examples of model checkers are Alloy Analyzer, BLAST, CADP, FDR2, HyTech, Java Pathfinder, mCRL2, NuSMV, Prism, SPIN, TLA+, and UPPAAL.



A discrete-time Markov Chain PRISM model of an embedded system comprising a processor which reads and processes data from two sensors.





The pictures above describe the application of the model checker UPPAAL to the classic "train-gate example" where six trains want to cross a one-track bridge and to do so safely. Each train has a specified arrival rate and can be stopped before some time threshold. When a train is stopped, it can start again. Eventually trains cross the bridge and go back to their safe state. In the second picture, the tool is used to estimate the probability that Train 0 will cross the bridge in less than 100 units of time.

Edmund Melson Clarke (left), E. Allen Emerson (centre) and Joseph Sifakis (right) received the 2007 A.M. Turing Award "for their role in developing Model-Checking into a highly effective verification technique that is widely adopted in the hardware and software industries." Those scientists introduced Model Checking as an algorithmic system verification technique in two path-breaking papers published in 1981 (Edmund M. Clarke, E. Allen Emerson) and 1982 (Jean-Pierre Queille; Joseph Sifakis).

Further reading

C. Baier & J.-P. Katoen. *Principles of Model Checking.* MIT Press, 2008.

E. M. Clarke, O. Grumberg, D. Kroening, D. Peled, & H. Veith. *Model Checking,* 2nd edition. MIT Press, 2016.

E. M. Clarke, E. A. Emerson, & J. Sifakis. "Model checking: algorithmic verification and debugging." *Communications of the ACM* **52**(11):74–84, 2009.

```
tteral) ~(cls:clause list) : clause list =
un (c:clause) ->
t.filter_map (fun (l':litteral) ->
 if l = l' then raise SAT
 else if l = -l' then None
 else Some l') c)
one) cls

(cls:clause list) : litteral =
 (fun (c:clause) -> List.length c = 1) cls)


 let sat ~(cls:clause list) : interpretation
   let rec aux (cls:clause list) (i:interpre
     if cls = [] then Some i
     else if List.mem [] cls then None
     else try
         let l = unit_litteral cls in
         aux (simplify l cls) (l::i)
       with Not_found ->
```

The Science of Programming

Languages & tools

Parsing of programming languages was based on the study of grammars, formal languages and automata. At ICALP'72, 30 out of 50 presentations dealt with formal languages and automata theory. In the 1970's, the theory of programming languages turned to the description of their semantics with algebra, denotational semantics, and mathematical logic. Since then, new conferences have appeared about logic in computer science, principles of programming languages, compilers, functional programming, types, static analysis, concurrency, automatic verification.

Programming Languages

The next 700 programming languages predicted by Peter Landin in 1965 are now nearly existing. Today languages are introduced with their semantics written in a more or less formal setting. Mathematical models have also influenced the design of new concepts (types, closures, objects, etc).

FROM THE 1930'S FOUNDATIONAL COMPUTATION MODELS...

Lambda-calculus

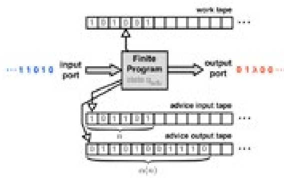
Church

$(\lambda x.M)N \longrightarrow M\{x := N\}$

CLOSURES →

← MONADS

Turing machines



Turing

...TO REAL-WORLD LANGUAGES

Functional languages

Procedural languages

WITH PROGRAMMING PARADIGMS

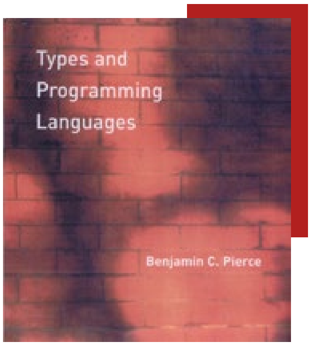
- Object-oriented
- Logic
- Scripting
- Descriptive
- Functional
- Procedural

Programming Tools

The first programming tools dealt with compiler construction or program profiling. Nowadays they include program verification, static analysis, and program testing. These new tools have followed theoretical progress in the semantics of programming languages, dependent high-order types, interactive proof-checkers, automatic provers, and abstract interpretation.

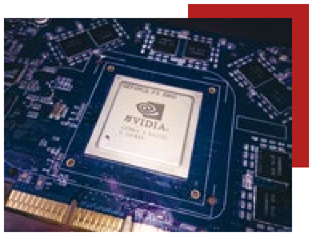
TYPES

POLYMORPHISM
OVERLOADING
GRADUAL TYPING
MODULES AND FUNCTORS



COMPILERS

VIRTUAL MACHINES
OPTIMIZED CODE
SPECIAL HARDWARE

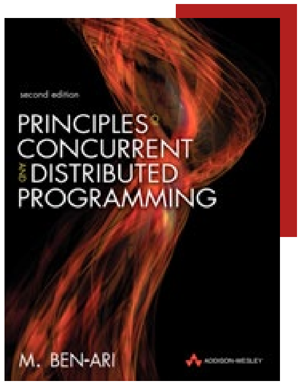


VERIFICATION

LOGIC FOR PROGRAMS
MACHINE-CHECKED PROOFS
STATIC ANALYSIS

CONCURRENCY

RACE-FREE
RESOURCE ALLOCATION
PROOFS



Further reading

P. J. Landin. “The next 700 programming languages.” *Communications of the ACM* **9**(3):157–166, 1966.

G. Winskel. *The Formal Semantics of Programming Languages: An Introduction.* MIT Press, 1993.

J. C. Mitchell. *Concepts in Programming Languages.* Cambridge University Press, 2003.

$$\vdash \Gamma,$$

$$y, x < z \wedge z < y \quad \text{(ax)} \qquad \vdash \Gamma, \neg$$

$$\vdash \Gamma, x < y \wedge (\neg x < z \vee \neg z < y), \neg x < y$$

$$\vdash \Gamma, x < y \wedge (\neg x < z \vee \neg z < y), \neg x < y \vee$$

$$x.\varphi_1(x), \exists y.\varphi'_1(x, y), x < y \wedge (\neg x < z \vee \neg z < y$$

$$\varphi_1(x), \exists y.\varphi'_1(x, y), \forall z.x < y \wedge (\neg x < z \vee \neg z <$$

$$\vdash \exists x.\varphi_1(x), \exists y.\varphi'_1(x, y), \exists z.\varphi'_2(x, y, z$$

$$\vdash \exists x.\varphi_1(x), \varphi_1(x), \forall y \exists z.\varphi'_2(x, y, z),$$

$$\vdash \exists x.\varphi_1(x), \varphi_1(x), \exists x.\varphi_2(x$$

$$\vdash \exists x.\varphi_1(x), \exists x.\varphi_2(x)$$

$$\vdash (\exists x.\varphi_1(x)) \vee (\exists x.\varphi_2(x))$$

Machine Checked Proofs

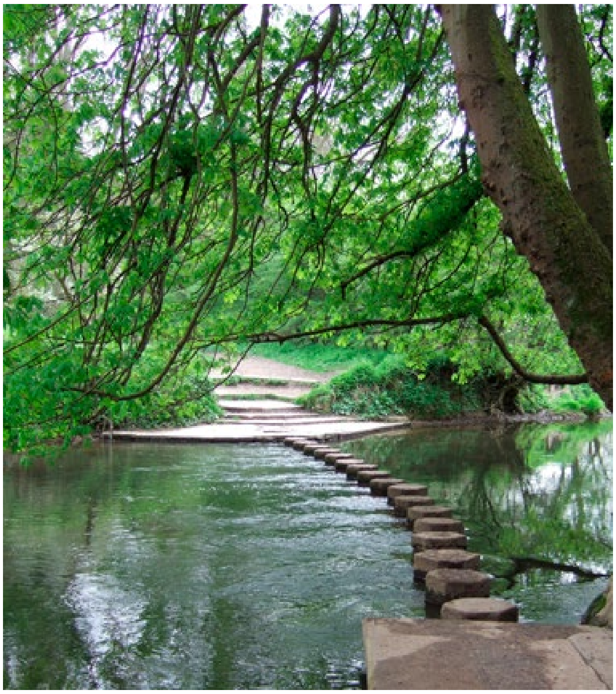
When computers improve mathematical rigour

Since the invention of the concept of proof in ancient Greece, mathematicians have always sought to write ever more rigourous proofs: identifying axioms precisely, defining every object used in the proof, avoiding the call to intuition, etc. Machine-checked proof is a new step in this never ending quest of rigour.
A machine-checked proof is written with such precision that a computer program can check its correctness.

The beginning

The two first proof-checkers were Automath (de Bruijn, 1967), and then LCF (Milner, 1972). Their goals were different: Automath was designed to check general mathematical proofs, LCF, more specifically, proofs of properties of programs.

For long, mathematics was the only science not to use instruments. The computer is becoming the telescope of mathematicians



Like the crossing of a river ford, a mathematical proof goes step by step

Today

The development of proof-checkers triggered the development of new theories, besides set theory, to express mathematics: each system innovates, introducing new features to express mathematical statements and proofs, just like each new programming language introduces new features to express programs.

Popular proof-checkers are ACL 2, Agda, Coq, HOL Light, HOL 4, Lean, Mizar, Nuprl, PVS, and many others. These proof-checkers are specific to one theory. Others, such as Beluga, Dedukti, Isabelle, Lambda-prolog, Twelf, and others are frameworks, where various theories can be defined.

They have in total more than 10,000 users.

Recent proofs

2000: four colour theorem (Gonthier et al.)
2008: correctness of the C compiler CompCert (Leroy et al.)
2009: correctness of the operating system seL4 (Klein et al.)
2012: Feit-Thompson theorem (Gonthier et al.)
2014: Kepler's conjecture (Hales et al.)
2014: UniMath a body of mathematics using univalent foundations (Voevodsky et al.)

Several of these projects aim at gathering a substantial body of mathematics, like Euclid's *Element* and Bourbaki's *Eléments de mathématiques* did.

Two proofs of Peirce's law, in COQ and in the natural deduction calculus

```
Goal ((P -> Q) -> P) -> P.
intro piqip.
assert (ponp: P \/ ~P).
exact (classic P).
destruct ponp as [p|np].
assumption.
apply piqip.
intro p.
destruct np.
assumption.
Qed.
```

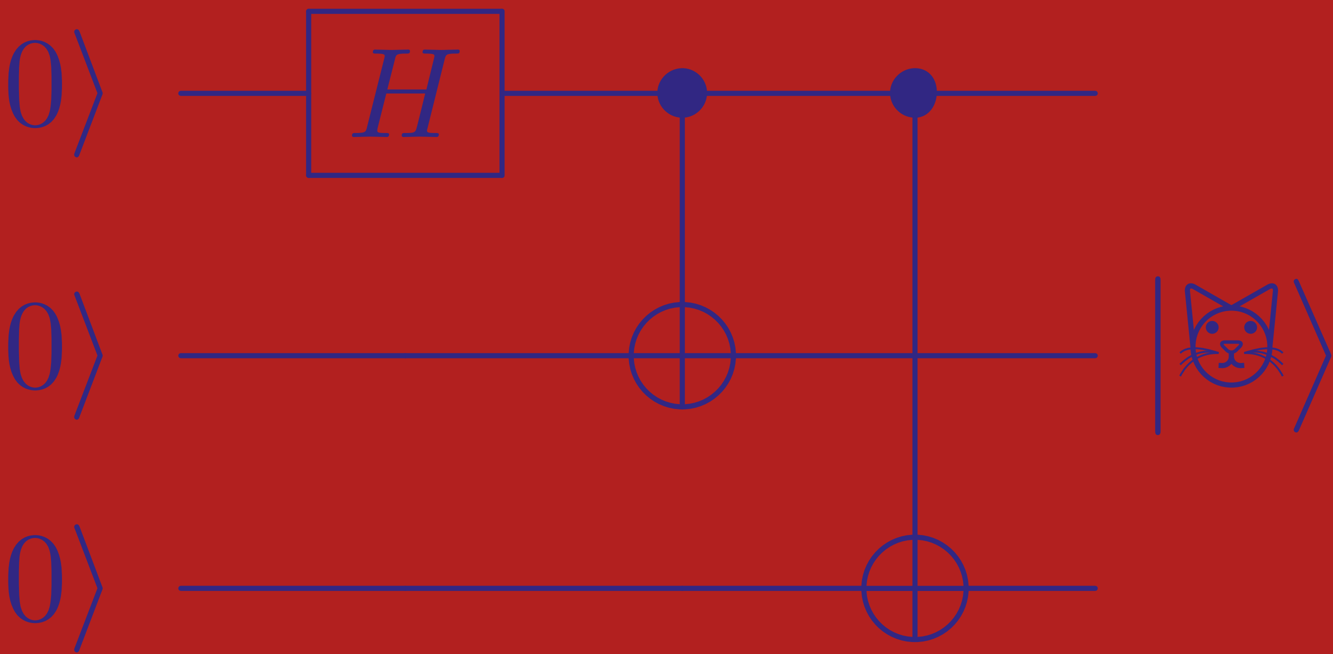
$$
\dfrac{
  \dfrac{}{\Gamma, \neg P \vdash \neg P}\,\text{(ax)}
  \quad
  \dfrac{
    \dfrac{}{\Gamma, \neg P \vdash (P \to Q) \to P}\,\text{(ax)}
    \quad
    \dfrac{
      \dfrac{
        \dfrac{
          \dfrac{}{\Gamma, \neg P, P, \neg Q \vdash \neg P}\,\text{(ax)}
          \quad
          \dfrac{}{\Gamma, \neg P, P, \neg Q \vdash P}\,\text{(ax)}
        }{\Gamma, \neg P, P, \neg Q \vdash \bot}\,(\neg\text{e})
      }{\Gamma, \neg P, P \vdash Q}\,(\bot_\text{c})
    }{\Gamma, \neg P \vdash P \to Q}\,(\to\text{i})
  }{\Gamma, \neg P \vdash P}\,(\to\text{e})
}{
  \dfrac{
    \dfrac{\Gamma, \neg P \vdash \bot}{\Gamma \vdash P}\,(\bot_\text{c})
  }{\vdash ((P \to Q) \to P) \to P}\,(\to\text{i})
}\,(\neg\text{e})
$$

Further reading

S. Owre and N. Shankar. "A Brief Overview of PVS." In *Proceedings of TPHOLs'08, Lecture Notes in Computer Science* **5170**:22–27, Springer, 2008.

Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. An EATCS Series, Springer, 2004.

G. Dowek. *Computation, Proof, Machine: Mathematics Enters a New Age.* Cambridge University Press, 2015.


0⟩
0⟩
0⟩
H

Quantum Computing

or, using Schrödinger's cat to solve problems faster

The development of quantum mechanics forced us to drastically rethink the definition of computation, leading to a new computational model called *quantum computing*. This model exploits quantum properties to solve some computational tasks more efficiently, and cryptographic tasks more securely, than classical computers.

Time Line

1905 › 35
Development of quantum mechanics

1970
Birth of quantum crypto with Wiesner quantum money scheme

1980 › 90
Theoretical conception of quantum computers

1990 › 2000
Conception of quantum algorithms, error correcting codes, quantum complexity theory

2000 › ...
Boom of quantum computing, first quantum devices

Subfields

Quantum algorithms. Solving computational tasks related to quantum mechanics (e.g., simulating molecular dynamics), as well as tasks unrelated to quantum mechanics (e.g., factorisation and search)

Quantum cryptography. Using quantum properties to achieve secure protocols for key exchange, money schemes,...

Quantum complexity theory. Fundamental connections between physics problems and quantum complexity classes

Quantum logic and programming languages. Developing and compiling applications on different physical architectures

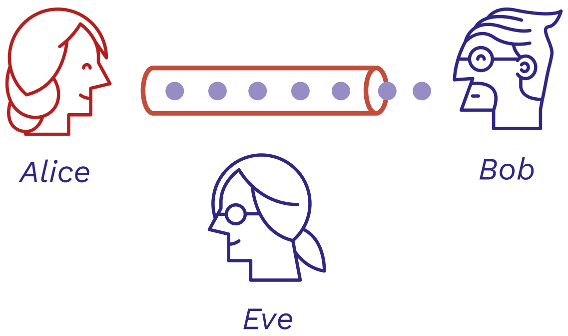
And more... Quantum information, quantum error correction, ...

From Theory to Practice

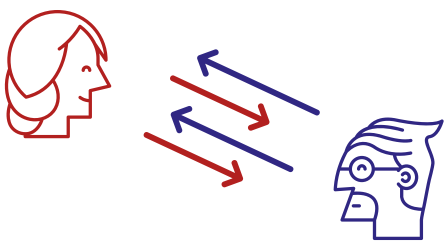
In parallel to developing the theory of quantum computation, there is a worldwide effort to actually build quantum computing devices and implement their applications. Some devices are already able to implement certain cryptographic protocols, and even made it to the public market. In contrast, the actual implementation of quantum algorithms is still in its infancy.
Recent years did bring an exciting first step called "quantum advantage:" a quantum device solving a computational task that cannot be efficiently solved by a classical computer.

Google's Sycamore quantum processor used for first quantum advantage experiment

PROTOCOL FOR SHARING SECRET KEY WITH PERFECT SECURITY



Alice sends qbits to Bob via untrusted channel



Using trusted classical channel, Alice and Bob check that Eve did not tamper with the state

Further reading

M. A. Nielsen & I. L. Chuang. *Quantum Computation and Quantum Information.* Cambridge University Press, 2010.

C. H. Bennett & G. Brassard. "Quantum cryptography: Public key distribution and coin tossing." *Theoretical Computer Science* **560**(1):7–11, 2014.

F. Arute & al., "Quantum supremacy using a programmable superconducting processor." *Nature* **574**:505–510, 2019.

Thank you

Project coordination

Sandrine Cadet
Communication officer, IRIF

Sylvain Schmitz
Professor, Université Paris Cité

Contributors

Luca Aceto - Reykjavik University, Gran Sasso Science Institute, L'Aquila
Susanne Albers - Technische Universität München
Simon Apers - CNRS
Giorgio Ausiello - Sapienza University of Rome
Alex Bredariol Grilo - LIP6, CNRS, Sorbonne Université
Ioannis Catzigiannakis - Sapienza University of Rome
Geoffroy Couteau - CNRS
Pierluigi Crescenzi - Gran Sasso Science Institute, L'Aquila
Gilles Dowek - Inria, ENS Paris-Saclay
Irène Guessarian - IRIF, Emerita at Sorbonne Université
Hugo Herbelin - Inria, IRIF
Ce Jin - MIT
Phokion G. Kolaitis - UC Santa Cruz, IBM Research
Jean-Jacques Levy - Inria, IRIF, CNRS
Kurt Mehlhorn - Max-Planck-Institut für Informatik
Mike Paterson - University of Warwick
Subhasree Patro - QuSoft, CWI Amsterdam
Sylvain Périfel - IRIF, Université Paris Cité
Jean-Eric Pin - IRIF, CNRS
Sylvain Schmitz - IRIF, Université Paris Cité

Images by the contributors plus

Inria/Photo Studio 9, Christian Morel/IRIF/CNRS Photothèque, Famille Nivat, Andreas Maletti, Forest Stearns, Wellcome Collection, Martyn Davies, Diego Torres Silvestre, Marta Kwiatkowska, Kim G. Larsen, Marius Mikucionis, Gethin Norman, Joseph Sifakis, AdobeStock, Noun project.

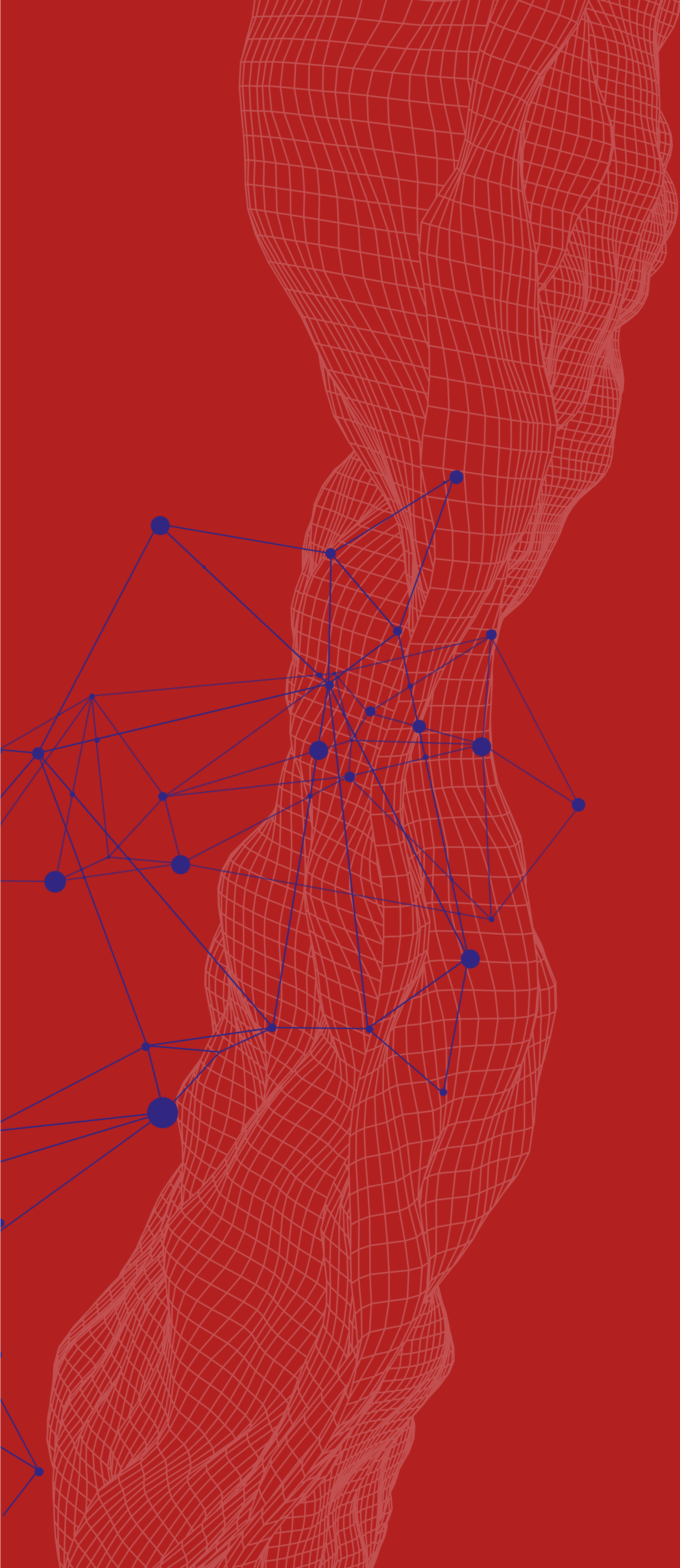
Layout

Faustine Brunet & Patrik Aveillan

Printing

Code&Clics, Miribel (Ain)

Partners of the project



0 euro TTC

9 782958 698703