Guest Lecture
Hard Problems and Exact Solutions

Matthew England
Coventry University Research Centre for
Computational Science and Mathematical Modelling

December 2024

Speaker supported by EPSRC grant EP/T015748/1.

Introductions

Matthew England: Associate Professor in Computer Science at Coventry University.

I am currently the co-Director of Coventry University's Research Centre for Computational Science and Mathematical Modelling (CSMM).

My job involves supervising research projects on the development, analysis and application of algorithms. I also teach on module 6008CEM (third year CS option).

https://matthewengland.coventry.domains/

CSMM Research Groups

We have research groups in:

- Symbolic algebra and logic (inc. applications to systems biology and algorithm optimisation with machine learning).
- Computational neuroscience (machine learning and signal processing development applied to EEG for disease detection).
- Human centred reinforcement learning (HVAC in buildings / cars; risk detection to first responders via uniform sensors).
- Computer vision technology (object detection in self-driving cars; vision for textiles to support greater accessibility).
- Data science applied to humanitarian energy systems; to understand and support healthier and more sustainable energy.

`www.coventry.ac.uk/research/areas-of-research/computational-science-mathematical-modelling/`

Outline of Guest Lecture

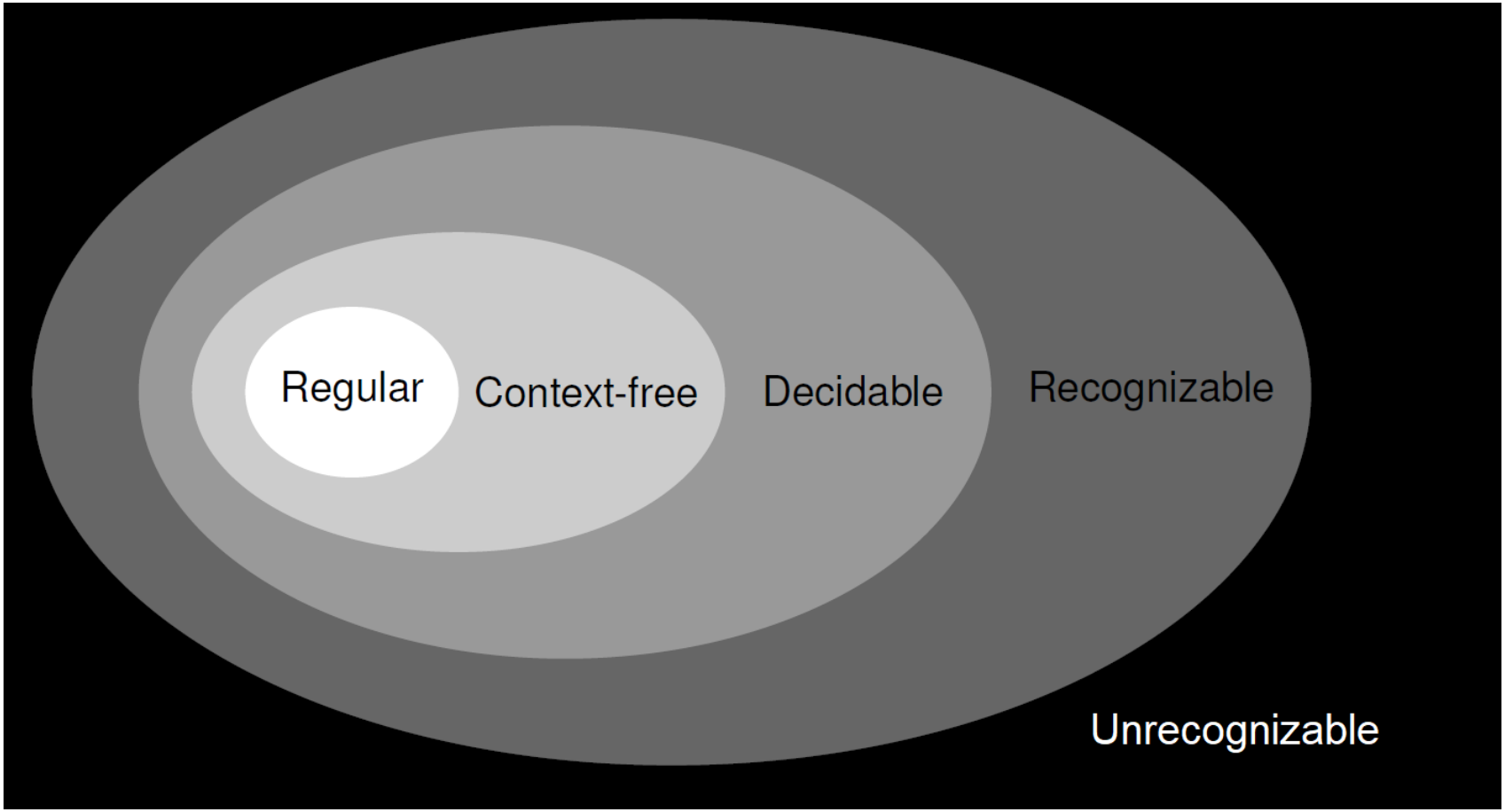
1. Recap of Background Material
   - Hard Problems (5002CEM/5000CMD)
   - Approaches to Hard problems (5003CEM/5002CMD)
2. Detour into some Alternatives
   - Exact Solutions
   - Real Quantifier Elimination

Note: "Detour" means non-examinable!

Outline

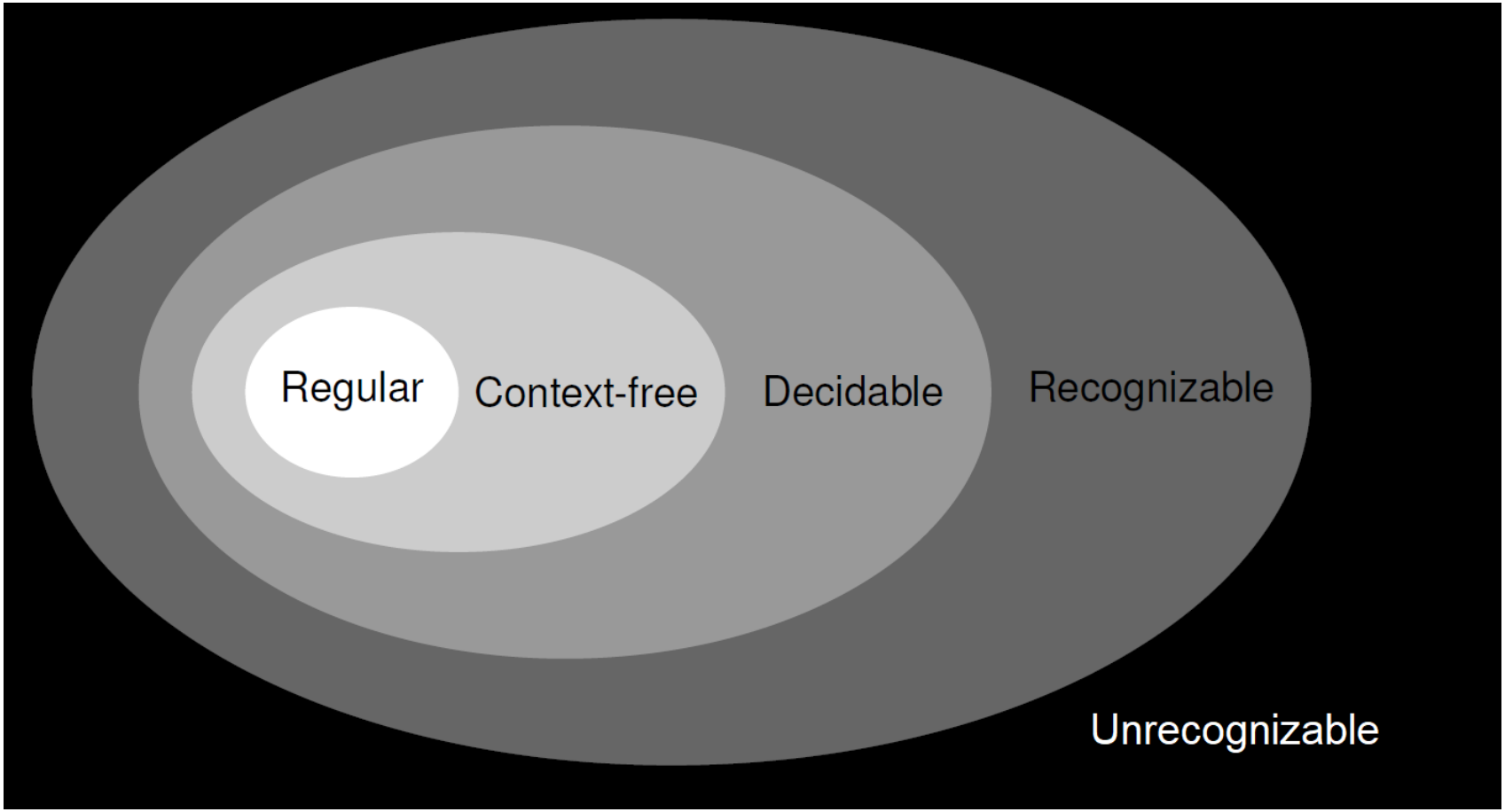
Computability

Computability theory is concerned with the question of what can be computed. Study through the lens of language recognition. We saw that problems (languages) could be categorised according to which models of computation could recognise them.



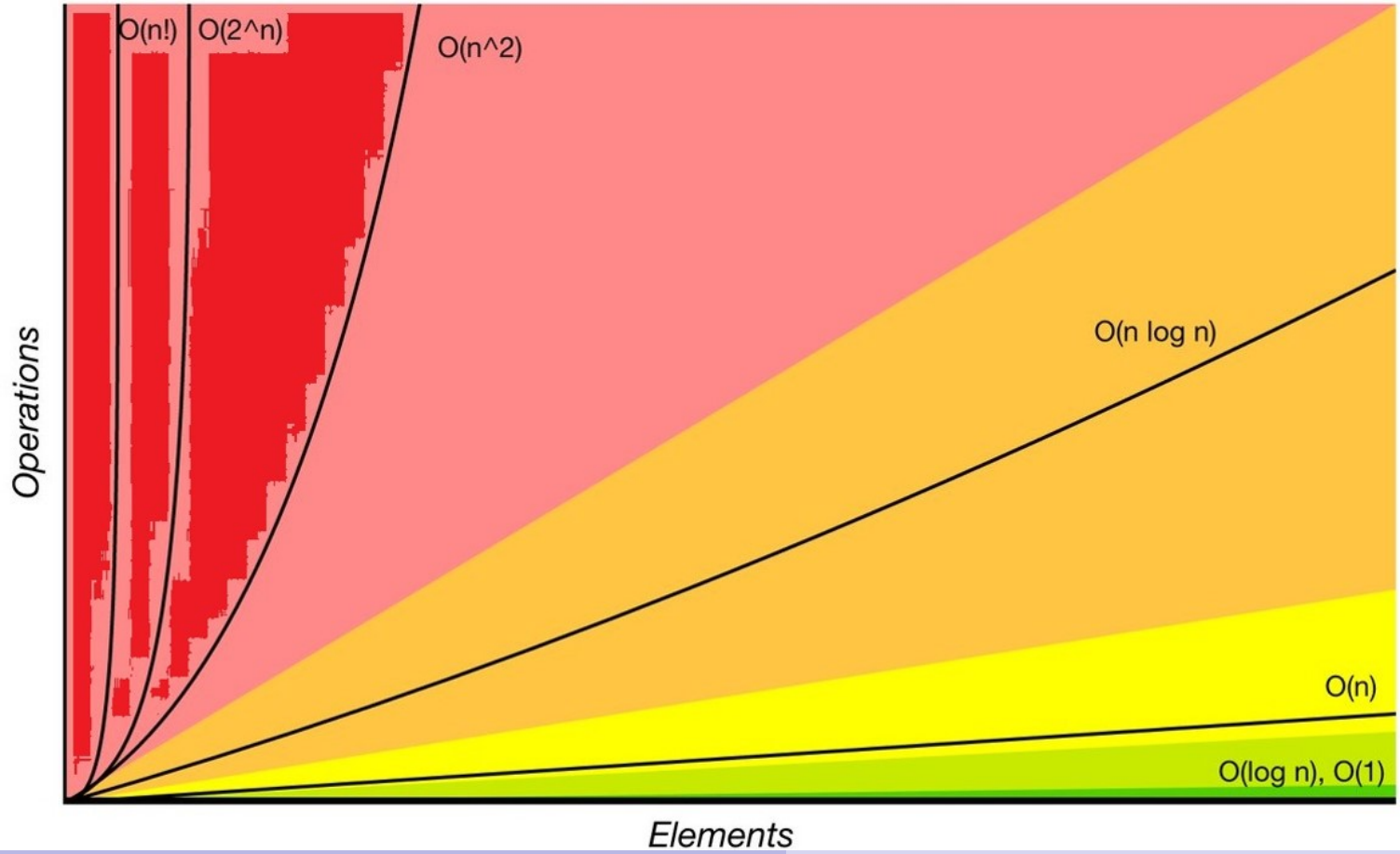
Computability

- Regular Languages recognised with DFAs, NFAs, RegEx.
- Context Free Languages recognised with PDAs, CFGs.
- Turning Machine decide some language but only recognise others.
- Some things are unrecognisable. I.e. some problems unsolvable!



Complexity of Algorithms

In first year you met the idea of the **Complexity of Algorithms**. E.g. Worst case complexity can be measured with Big-$\mathcal{O}$ theory. Different algorithms to solve the same problem can have different complexity: aim for the algorithm with least complexity.



Complexity of Problems

Computability theory led us to the **Complexity of Problems**.

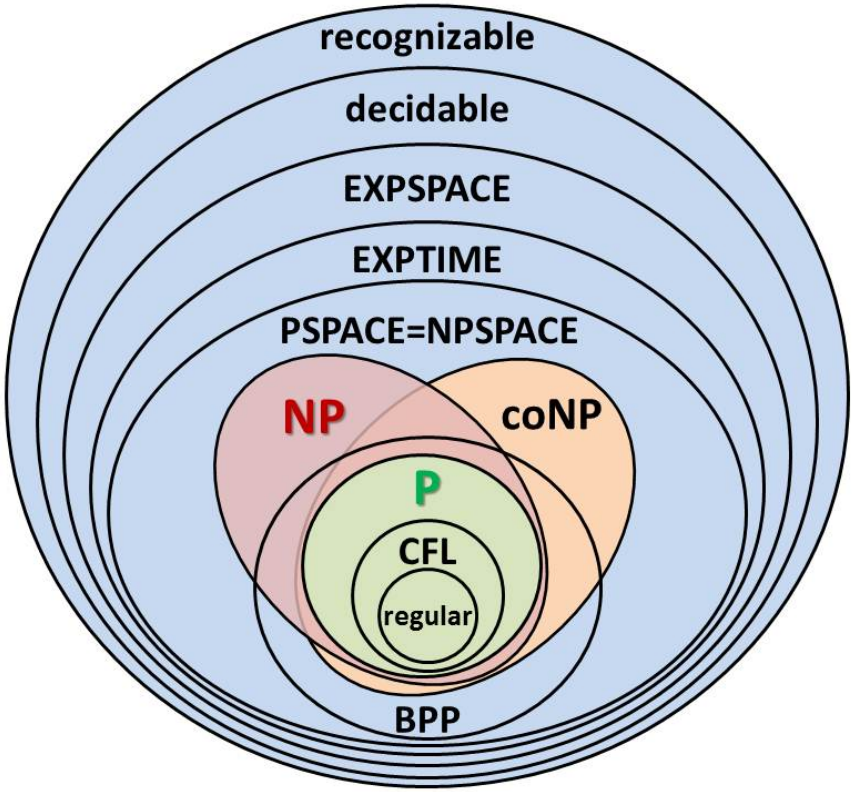
Further categorise the Turing Decidable problems according to complexity of fastest algorithm to solve them (on TM).

In particular:

P: Can be solved in **polynomial time**. I.e. $\mathcal{O}(n^c)$.

EXP: Can be solved in **exponential time**. I.e. $\mathcal{O}(c^n)$.

Here c is constant and n represents input size.



NP problems

You have also studied the class of **NP problems**. Who remembers what NP stands for?

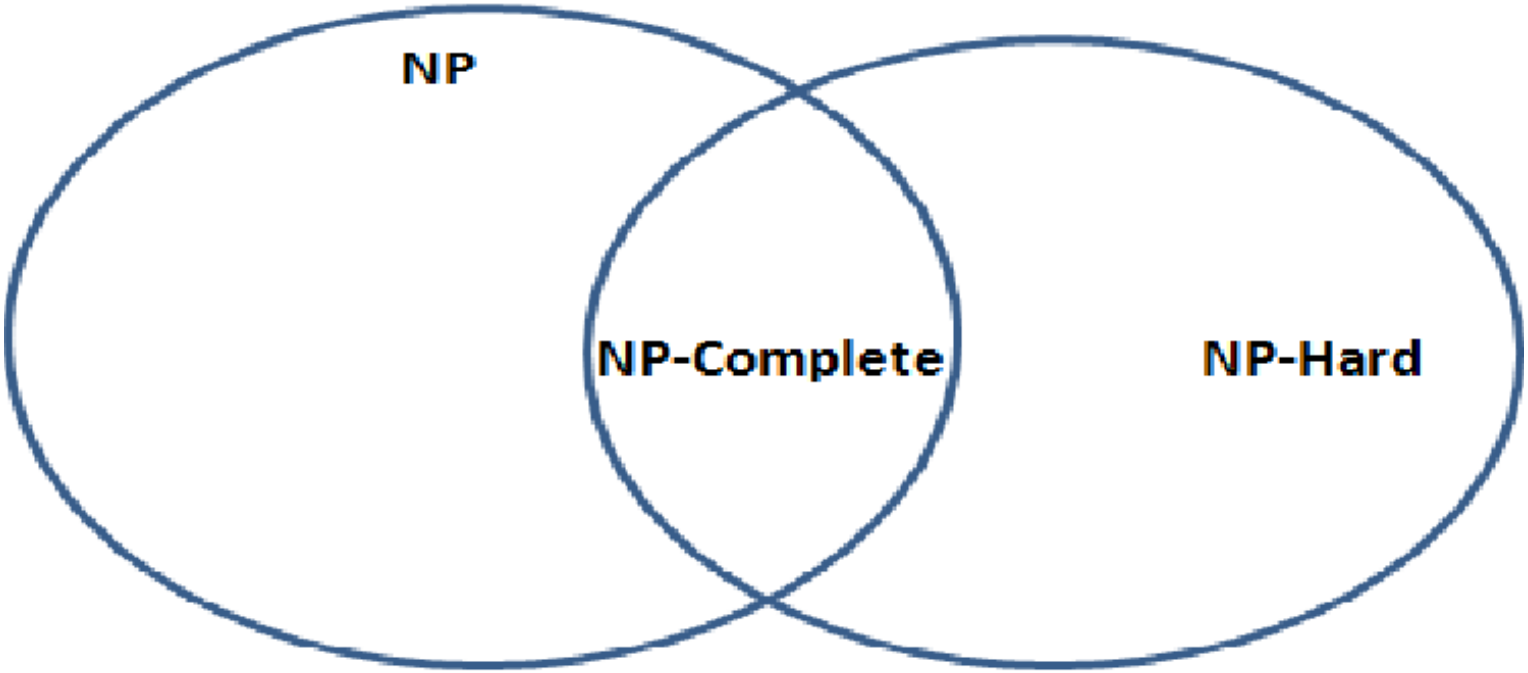
NP problems

You have also studied the class of **NP problems**: non-deterministic polynomial time problems. Technically the problems that can be solved in polynomial time on a non-deterministic Turing Machine. But equivalent to saying the decision problems whose solutions can be verified in in polynomial time.

A classic example is the **Boolean SAT Problem**. Here we are given a formula in propositional logic: i.e. built from Boolean variables, the logical operators $\wedge$ (and), $\vee$ (or), and $\neg$ (not).
The task is to determine if it is **satisfiable**: can we assign Boolean values to the variables to make the formula true?
With n variables there are 2^n (exponential) possible assignments to try. But given a solution it is trivial to verify the correctness.
Hence the SAT problem is in the NP problem class.

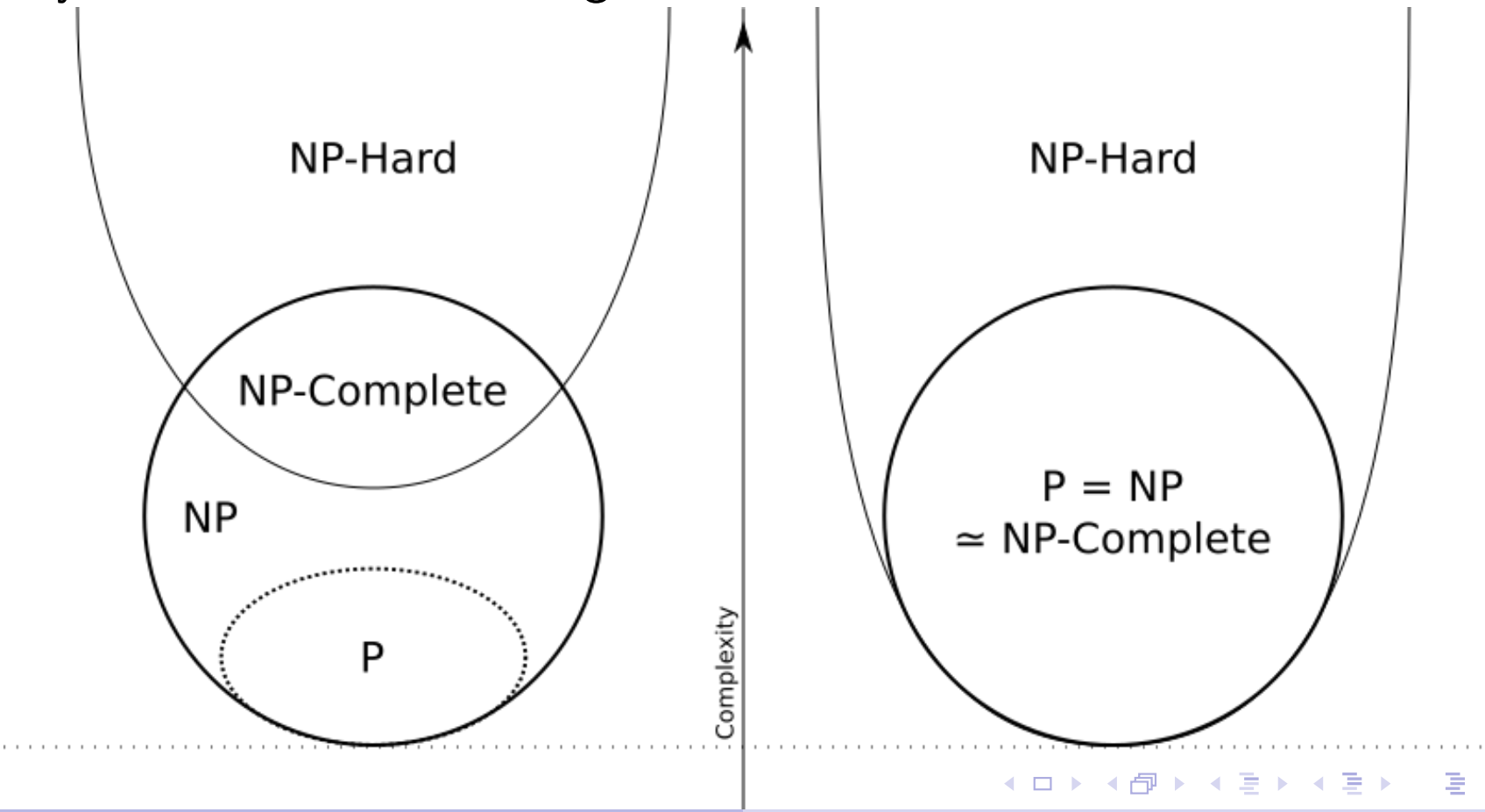
NP-hard and NP-complete

A problem is **NP-hard** if any problem in NP can be transformed into it in polynomial time (even if that problem is not in NP itself).
A problem is **NP-complete** if it is both NP and NP-hard.
E.g. The SAT problem is NP-complete. So if we ever found a polynomial time algorithm to solve the SAT problem then we would have access to polynomial time algorithms for **all** the NP problems.



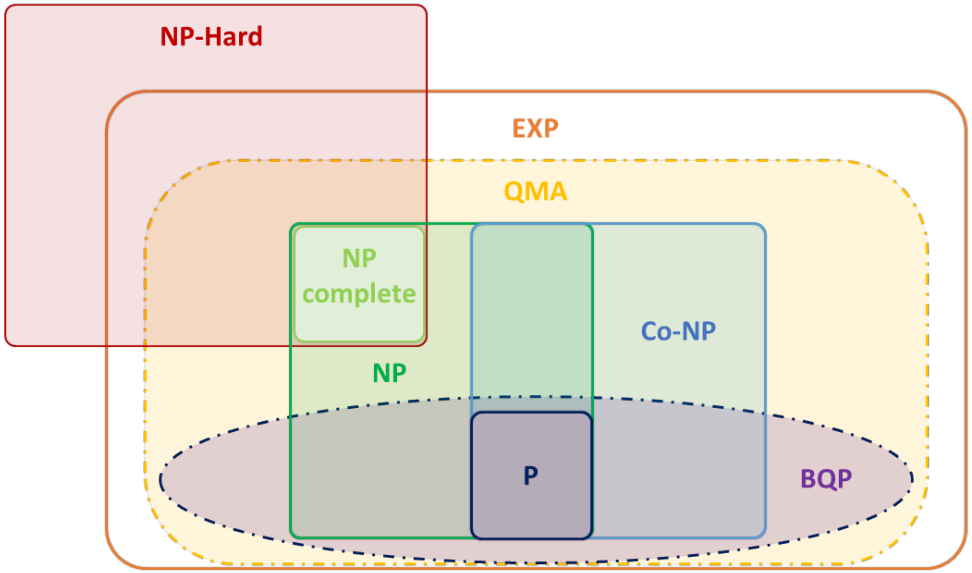
P vs NP

The greatest open problem in computer science is whether $P = NP$? At the moment, there are no polynomial time algorithms discovered for any NP-complete problem, but no one has proved beyond doubt that such algorithms cannot exist.



NP vs EXP

EXP is a strictly larger class than NP. I.e., all NP problems are in EXP, but EXP also has problems we know are not in NP (we know they cannot even have their solutions verified in polynomial time).



There are *many* other complexity classes out there, defined to help understand what is possible. But many real world problems are NP or EXP, so how to deal with these?

Outline

Parallel Computation

One way to tackle a hard problem is to provide a computer with more resources. E.g. parallel computing where we run code on multiple CPUs / machines at once. If we have two computers it will take half the time, right?

No. There will always be overhead in communication between the cores that make the absolute speed-up impossible.

More importantly, to apply parallel computing you need to design algorithms to make use of it: this is often a challenging task. Some algorithms are much easier to parallelize than others!

You can choose to learn about this next year in optional third year BSc CS module on *Parallel and Distributed Programming*.

Parallel Computation

One way to tackle a hard problem is to provide a computer with more resources. E.g. parallel computing where we run code on multiple CPUs / machines at once. If we have two computers it will take half the time, right?

No. There will always be overhead in communication between the cores that make the absolute speed-up impossible.

More importantly, to apply parallel computing you need to design algorithms to make use of it: this is often a challenging task. Some algorithms are much easier to parallelize than others!

You can choose to learn about this next year in optional third year BSc CS module on *Parallel and Distributed Programming*.

Parallel Computation

One way to tackle a hard problem is to provide a computer with more resources. E.g. parallel computing where we run code on multiple CPUs / machines at once. If we have two computers it will take half the time, right?

No. There will always be overhead in communication between the cores that make the absolute speed-up impossible.

More importantly, to apply parallel computing you need to design algorithms to make use of it: this is often a challenging task. Some algorithms are much easier to parallelize than others!

You can choose to learn about this next year in optional third year BSc CS module on *Parallel and Distributed Programming*.

Parallel Computation Limits

Parallel computation will still be limited on a high complexity algorithm. Always better to reduce the complexity of algorithm!

Big-O	Name	n=2	n=10	n=100
$O(1)$	Constant	1	1	1
$O(\log(n))$	Logarithmic	0.6	3	30
$O(n)$	Linear	2	10	100
$O(n\log(n))$	LogLinear	1.2	30	3000
$O(n^2)$	Quadratic	4	100	10,000
$O(n^3)$	Cubic	8	1,000	1,000,000
$O(n^4)$	Quartic	16	10,000	100,000,000
$\vdots$	$\vdots$			
$O(2^n)$	Exponential	4	1024	$1.27 * 10^{30}$

The quickest exponential algorithm doubles its computation requirements for each increase in input size: there will be a limit to how often you can double your CPUs!

Approximate Solutions

The usual approach to solving an NP or EXP problem is not so solve it, but to *approximate* a solution to it. I.e. to find an answer which is probably not perfect, but is good enough.

E.g. Your SatNav / Google Maps has to solve an NP problem when determining the shortest route from A to B. Finding the absolute shortest route would require checking exponentially many possibilities and thus take far too much computation and time.

Instead, these systems provide a route which is not guaranteed to be the absolute shortest, but it will be fairly short, and importantly, will be provided to you quickly!

Approximate Solutions

The usual approach to solving an NP or EXP problem is not so solve it, but to *approximate* a solution to it. I.e. to find an answer which is probably not perfect, but is good enough.

E.g. Your SatNav / Google Maps has to solve an NP problem when determining the shortest route from A to B. Finding the absolute shortest route would require checking exponentially many possibilities and thus take far too much computation and time.

Instead, these systems provide a route which is not guaranteed to be the absolute shortest, but it will be fairly short, and importantly, will be provided to you quickly!

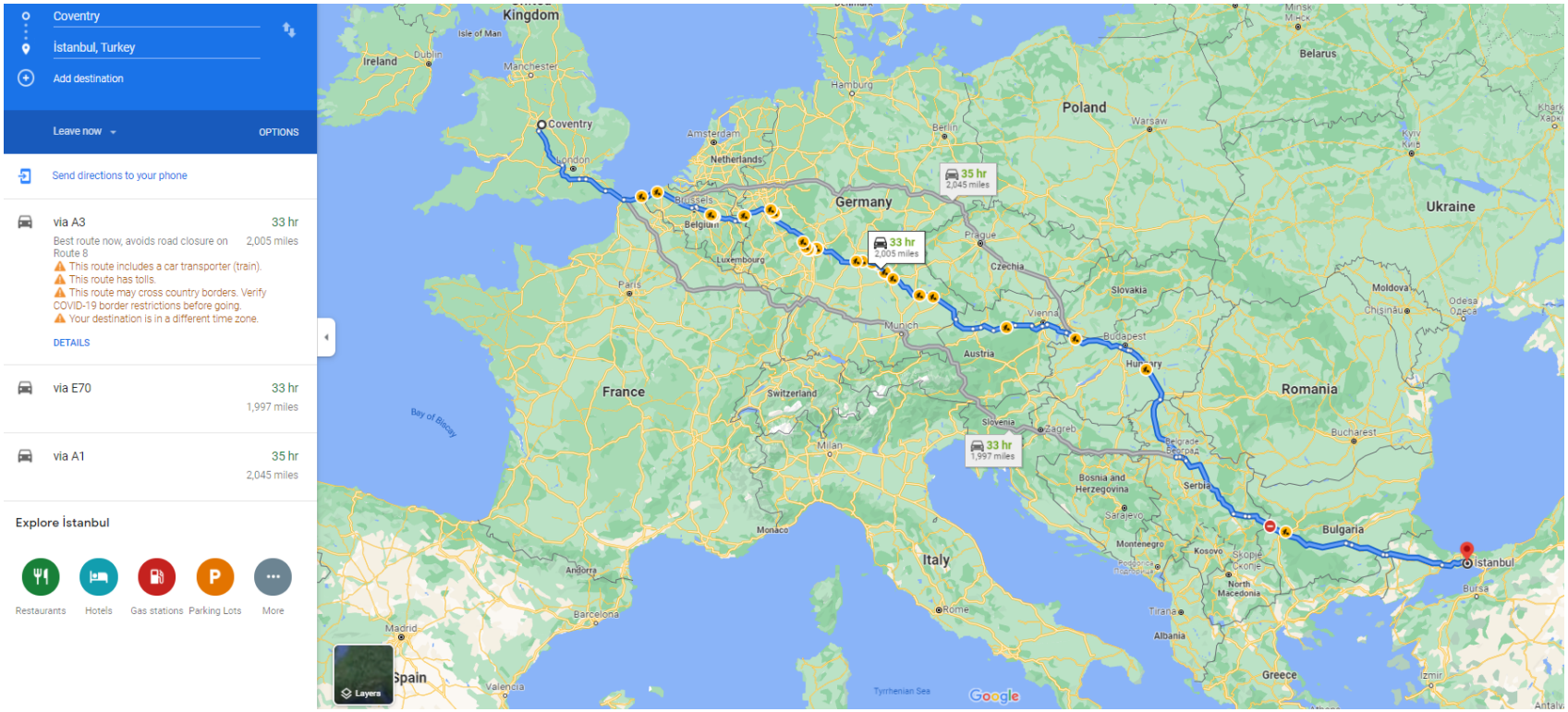
Approximate Solutions

The usual approach to solving an NP or EXP problem is not so solve it, but to *approximate* a solution to it. I.e. to find an answer which is probably not perfect, but is good enough.

E.g. Your SatNav / Google Maps has to solve an NP problem when determining the shortest route from A to B. Finding the absolute shortest route would require checking exponentially many possibilities and thus take far too much computation and time.

Instead, these systems provide a route which is not guaranteed to be the absolute shortest, but it will be fairly short, and importantly, will be provided to you quickly!

Coventry to Istanbul: 2s result from Google Maps



Simple Heuristics

A **heuristic algorithm** is one that solves a problem by trading accuracy for speed. Such algorithms can be very simple.

A **greedy heuristic** solves the problem by making a series of local decisions based on a simple criteria.

E.g. for the SatNav problem we could write a simple greedy heuristic: each time we are at a crossroads take road from current location that is closest in compass direction to our destination.

In this example we had better supplement it by not picking roads that are dead ends and noticing when we are in loops!

Simple Heuristics

A **heuristic algorithm** is one that solves a problem by trading accuracy for speed. Such algorithms can be very simple.

A **greedy heuristic** solves the problem by making a series of local decisions based on a simple criteria.

E.g. for the SatNav problem we could write a simple greedy heuristic: each time we are at a crossroads take road from current location that is closest in compass direction to our destination.

In this example we had better supplement it by not picking roads that are dead ends and noticing when we are in loops!

Watch your Language!

Heuristics can be a lot more complicated than the greedy example.

Please note the difference between a *complicated* algorithms and algorithms with high *complexity*!

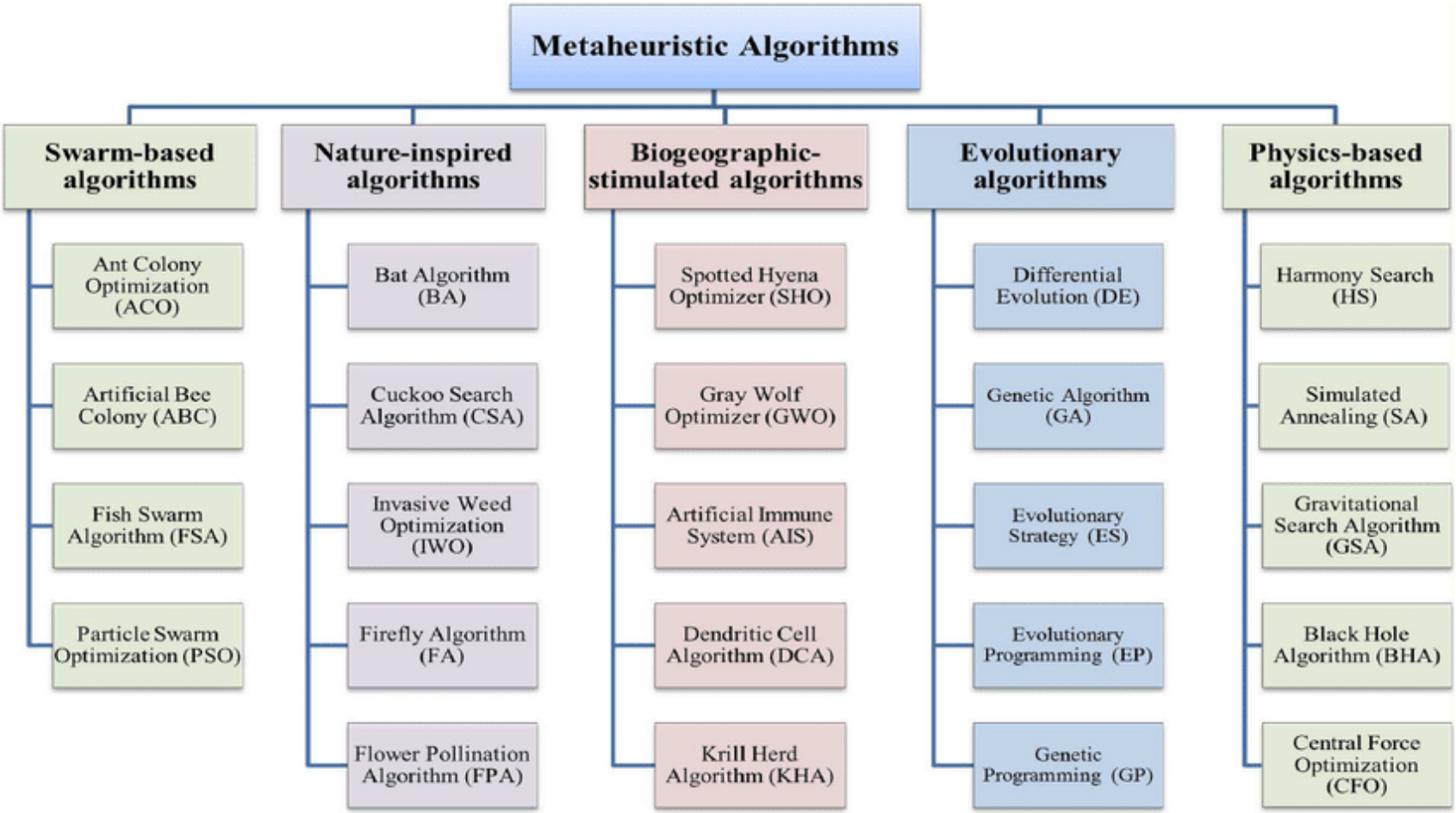
E.g. the brute force algorithm for the SatNav problem of *measure every possible route and pick the shortest* is not complicated to write down but would have very high complexity (exponential)! The actual algorithm used by GoogleMaps will have a lot more code, but will run with much lower (polynomial) complexity.

Metaheuristics

Metaheuristics are strategies for building sophisticated heuristic algorithms. There are a wide variety including those inspired:

- by how nature solves problems (ant colony optimisation, particle swarm optimisation);
- by natural selection (evolutionary algorithms, genetic algorithms);
- by the physics of cooling metal (simulated annealing).

You cannot just apply a meta-heuristic. You need to specialise the strategy to your particular problem. This can sometimes be easy and sometimes hard. The quality of the end results can be down to both the meta-heuristic strategy choice but also the specialisation.


Metaheuristic Algorithms
Swarm-based algorithms
Ant Colony Optimization (ACO)
Artificial Bee Colony (ABC)
Fish Swarm Algorithm (FSA)
Particle Swarm Optimization (PSO)
Nature-inspired algorithms
Bat Algorithm (BA)
Cuckoo Search Algorithm (CSA)
Invasive Weed Optimization (IWO)
Firefly Algorithm (FA)
Flower Pollination Algorithm (FPA)
Biogeographic-stimulated algorithms
Spotted Hyena Optimizer (SHO)
Gray Wolf Optimizer (GWO)
Artificial Immune System (AIS)
Dendritic Cell Algorithm (DCA)
Krill Herd Algorithm (KHA)
Evolutionary algorithms
Differential Evolution (DE)
Genetic Algorithm (GA)
Evolutionary Strategy (ES)
Evolutionary Programming (EP)
Genetic Programming (GP)
Physics-based algorithms
Harmony Search (HS)
Simulated Annealing (SA)
Gravitational Search Algorithm (GSA)
Black Hole Algorithm (BHA)
Central Force Optimization (CFO)

What about Machine Learning? (Slide 17/36)

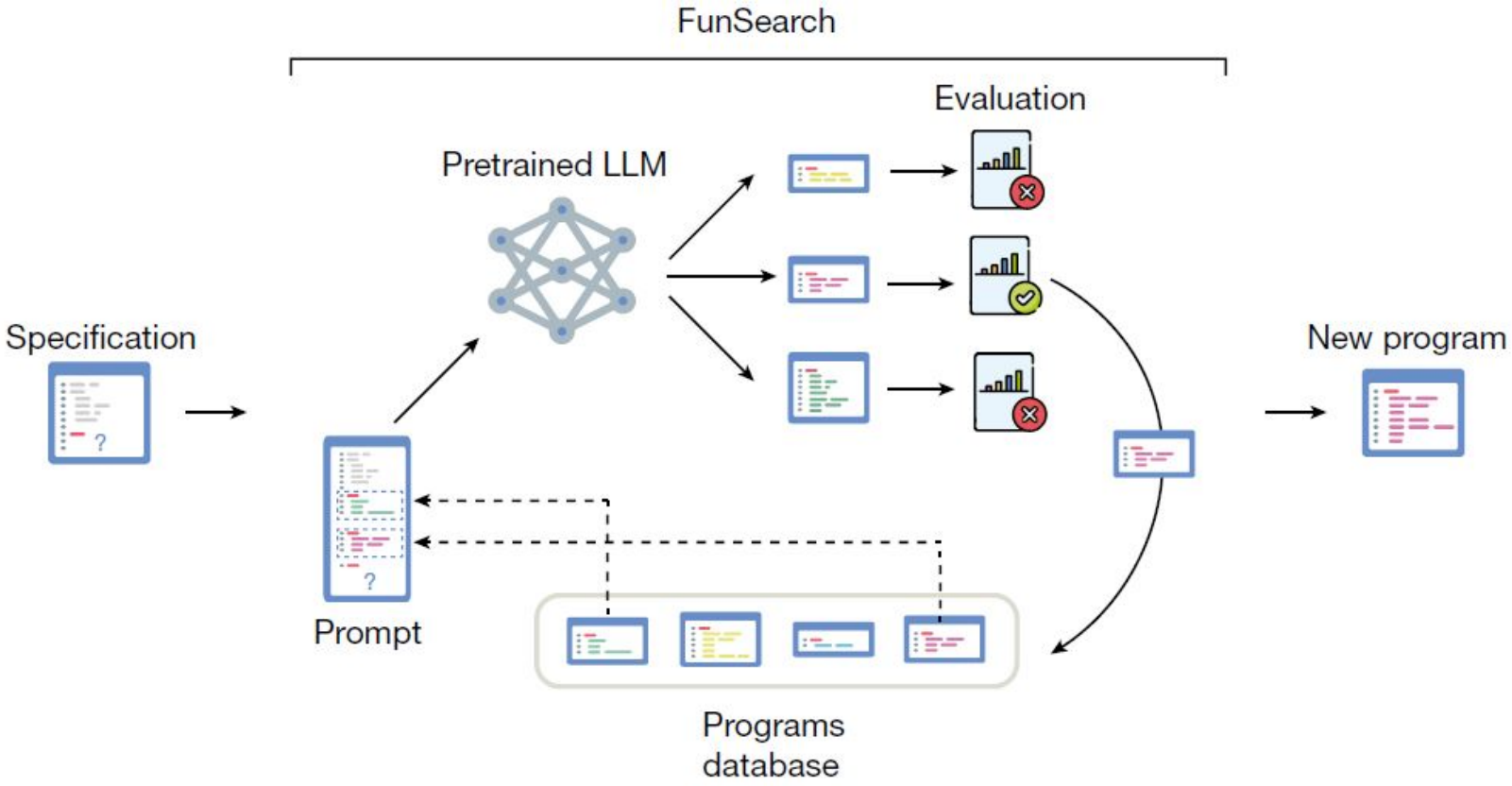
A Machine Learning (ML) model to solve an NP or EXP problem can be thought of as a heuristic algorithm to the problem (since ML is never guaranteed to be 100% accurate).

However, the size of the exponential solution space that these problems have makes it challenging to train such an ML model.

Recently, ML has found success in discovering / designing / improving algorithms for hard problems. E.g.

Romera-Paredes *et al.*
Mathematical discoveries from program search with large language models.
Nature, 625, pp. 468−−475, 2023.
https://doi.org/10.1038/s41586-023-06924-6

Starts with a database of heuristic methods for the problems and evolves them by feeding two at a time as a prompt to an LLM (Codey) which is asked to merge them; with the outputs evaluated for fitness and the best added into the database.


FunSearch
Evaluation
Pretrained LLM
Specification
New program
Prompt
Programs
database

Outline

Reason for Exact Solution: Accuracy

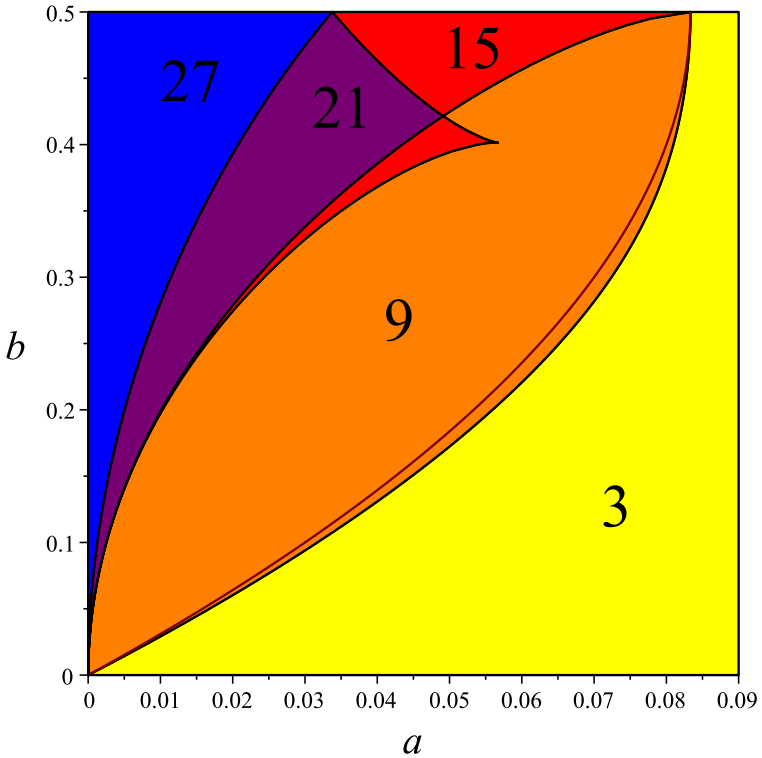
There is still a case for finding exact solutions.

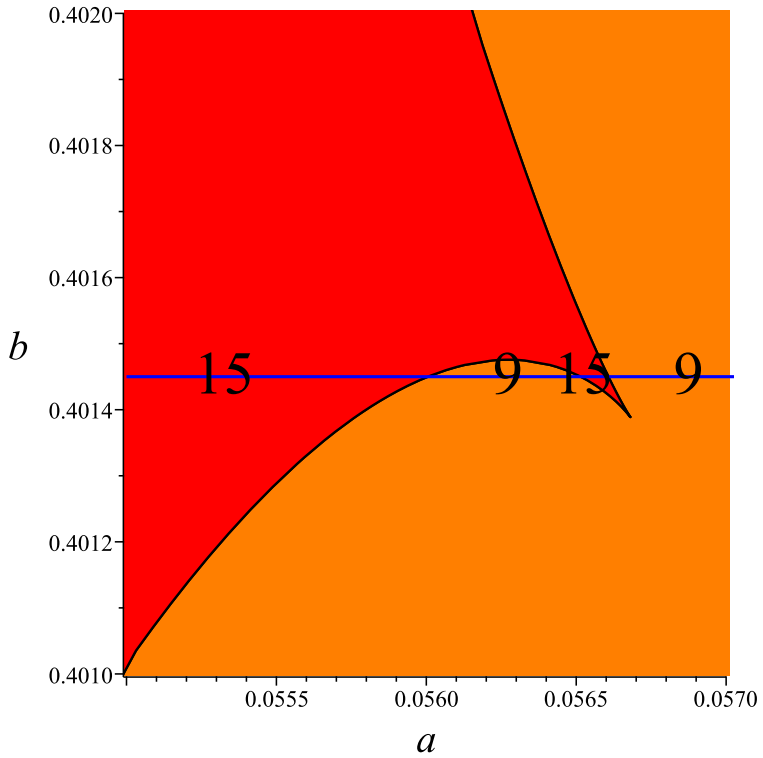
For example, when accuracy is of paramount importance. E.g. safety critical systems; formal verification procedures.

Reason for Exact Solution: Understanding (Slide 19/36)

There is still a case for finding exact solutions.

Exact solutions can offer *fundamental insight* that can be missing from approximate ones. We may want to understand not just what the solution is but what it means; what properties it has.





Back to the SAT Problem

Recall again the **Boolean SAT Problem**: given a formula in propositional logic: i.e. one built up from Boolean variables, the logical operators $\wedge$ (and), $\vee$ (or), and $\neg$ (not); determine if it is **satisfiable** (can we assign Boolean values to the variables to make the formula true?)

Trying all possible assignments is clearly hopeless. But rather than reaching for approximate solutions there are smarter ways to solve it exactly. These smarter ways still have worst case exponential complexity (if they didn't they would prove P=NP). But they significant;y improve the average case complexity!

SAT Solvers

SAT-Solvers are software dedicated for finding an (exact) solution to the SAT-Problem (and thus those that can be reduced to it).

SAT-solvers today can routinely solve problem instances with tens of thousands of variables over formulae involving millions of symbols. They are hence used every day in industry (circuit design, software verification, scheduling, planning, etc.)

This does not contradict the complexity mathematics. The algorithms in SAT-solvers are of exponential complexity. However, the Big-$\mathcal{O}$ concerns the worst case. These cases exist – and here SAT solvers fail – however, it appears they are rare both statistically and in practice.

I.e. the worst case may be exponential, but the average case is not.

SAT Solvers

SAT-Solvers are software dedicated for finding an (exact) solution to the SAT-Problem (and thus those that can be reduced to it).

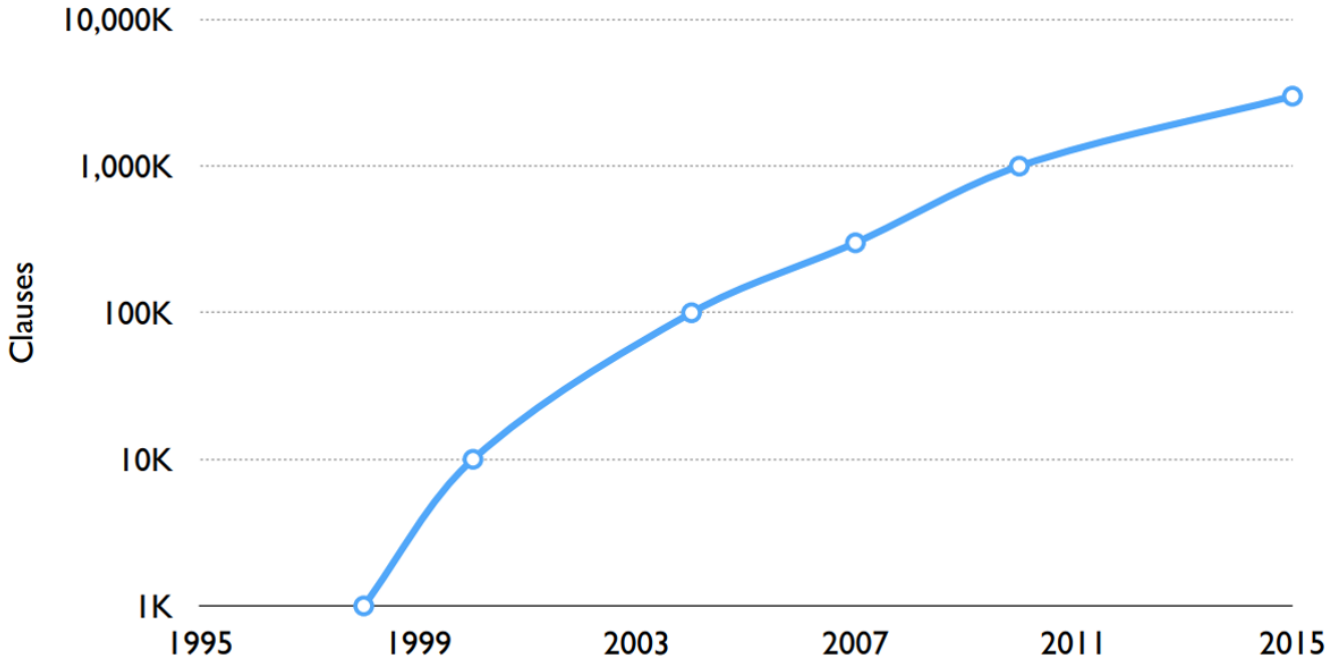
SAT-solvers today can routinely solve problem instances with tens of thousands of variables over formulae involving millions of symbols. They are hence used every day in industry (circuit design, software verification, scheduling, planning, etc.)

This does not contradict the complexity mathematics. The algorithms in SAT-solvers are of exponential complexity. However, the Big-$\mathcal{O}$ concerns the worst case. These cases exist – and here SAT solvers fail – however, it appears they are rare both statistically and in practice.

I.e. the worst case may be exponential, but the average case is not.

SAT Solver Algorithms

The advance of SAT solvers followed a breakthrough in the 1990s allowing intelligent search through the possible assignments.



Based on a slide from Vijay Ganesh

You can study that algorithm if you choose optional BSc CS module *Advanced Programming Paradigms*.

My Research

Exact algorithms to solve problems involving non-linear polynomials:

- their invention and development;
- their analysis (correctness, complexity, output properties);
- their integration with computational logic solvers;
- their application (at the moment particularly in systems biology and economics); and
- their optimisation with data science and machine learning.

Symbolic Computation: computation that manipulates algebraic objects exactly (e.g. polynomial arithmetic, exact numerics using fractions and algebraic numbers).

Finish today one such example: quantifier elimination.

Outline

Quantifier Elimination Problem

We consider logical formula whose atoms are not Boolean variables, but instead statements about the signs of polynomials with integer coefficients: i.e. $f \, \sigma \, 0$ where f is a polynomial and $\sigma \in \{=, <, >\}$ (and by logical combination also $\{\leq, \geq, \neq\}$).

The formula may be preceded by quantification statements on variables: either $\forall$ (for all) or $\exists$ (there exists).

Quantifier Elimination (QE) means to take such a formula and produce one that is equivalent and does not use quantifiers.

Tarski proved QE should be possible in the 1940s. The first algorithm was implemented in the 1970s. But it remains computationally challenging.

Example: Fully quantified formulae examples (Slide 25/36)

Input: $\exists x, x^2 + 1 > 0$

Output: `True`

Input: $\forall x, x^2 + 1 > 0$

Output: `True`

Input: $\exists x, x^2 + 3x + 1 > 0$

Output: `True`

Input: $\forall x, x^2 + 3x + 1 > 0$

Output: `False`

Example: Fully quantified formulae examples (Slide 25/36)

Input: $\exists x, x^2 + 1 > 0$

Output: `True`

Input: $\forall x, x^2 + 1 > 0$

Output: `True`

Input: $\exists x, x^2 + 3x + 1 > 0$

Output: `True`

Input: $\forall x, x^2 + 3x + 1 > 0$

Output: `False`

Example: Fully quantified formulae examples (Slide 26/36)

Input: $\exists x, x^2 + 1 > 0$

Output: `True`

Input: $\forall x, x^2 + 1 > 0$

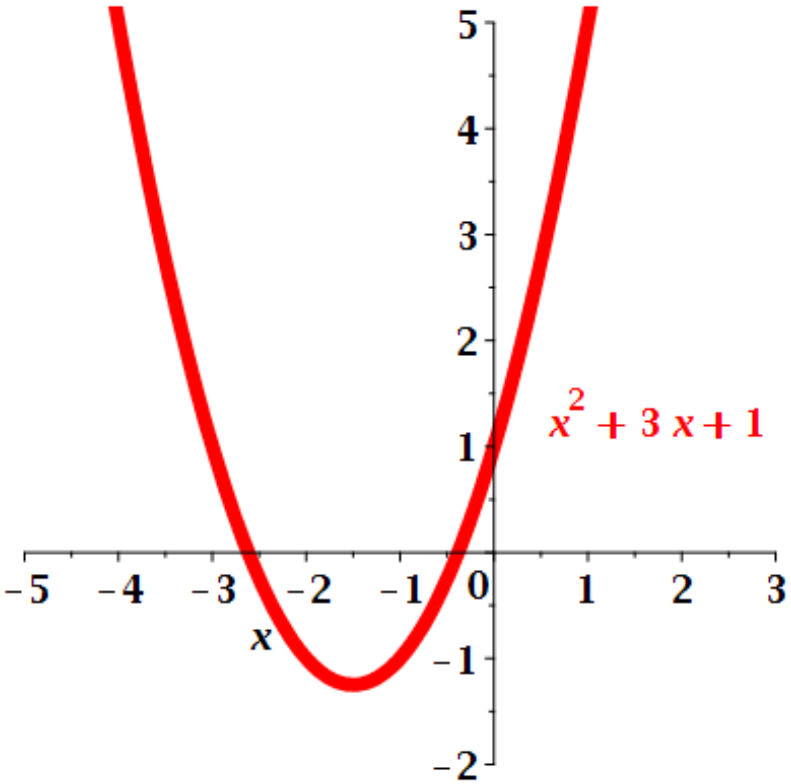
Output: `True`

Input: $\exists x, x^2 + 3x + 1 > 0$

Output: `True`

Input: $\forall x, x^2 + 3x + 1 > 0$

Output: `False`



Partially quantified formulae example

So we have that

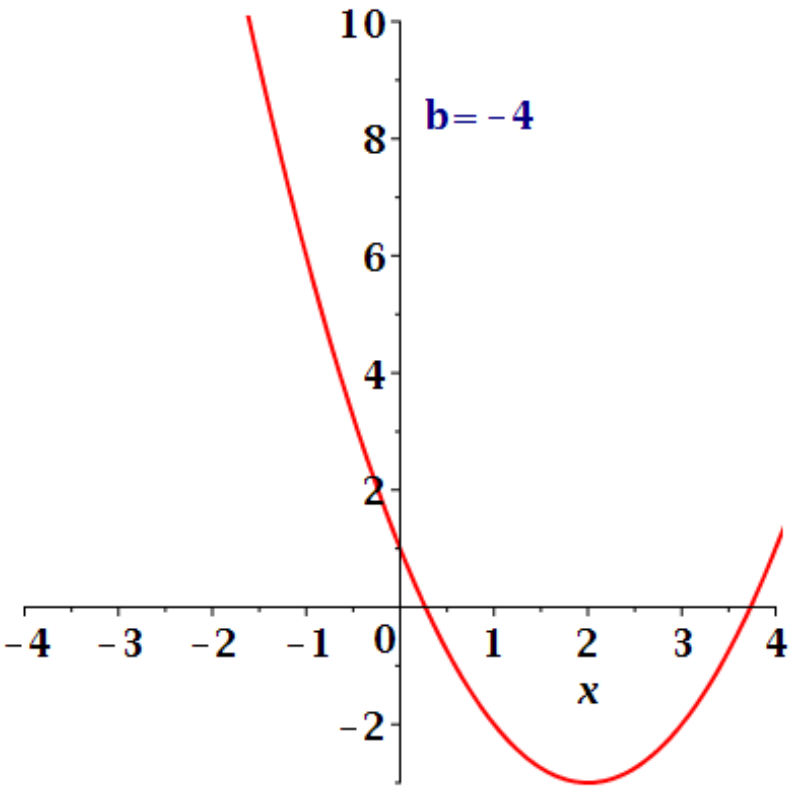
$$\forall x, x^2 + 1 > 0 \text{ is } \texttt{True},$$
$$\forall x, x^2 + 3x + 1 > 0 \text{ is } \texttt{False}.$$

What should we expect for this?

Input: $\forall x, x^2 + bx + 1 > 0$

Output: $(-2 < b) \wedge (b < 2)$

The answer must depend on b.



Partially quantified formulae example

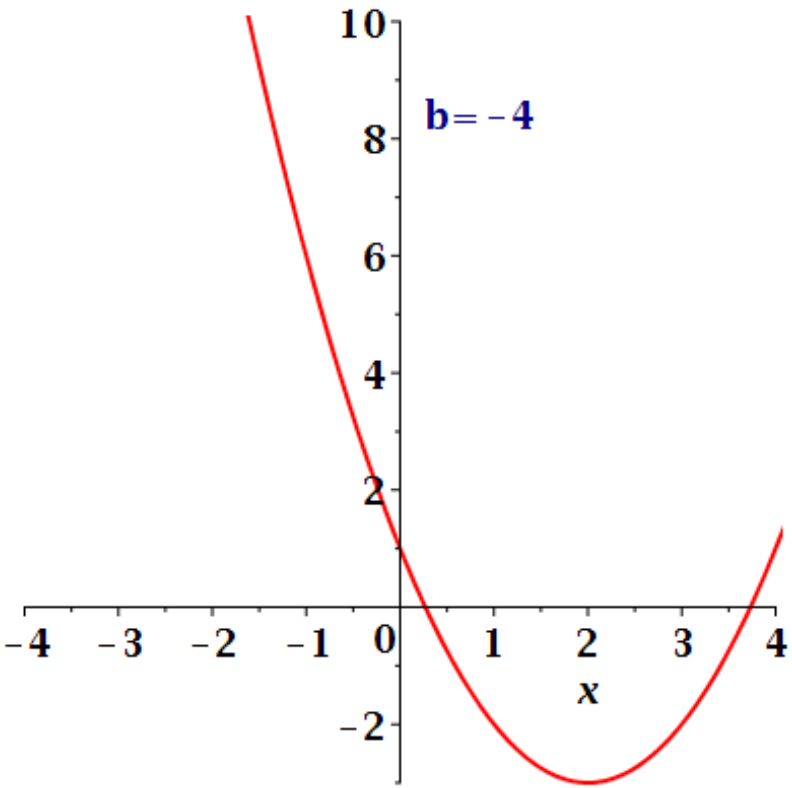
So we have that

$$\forall x, x^2 + 1 > 0 \text{ is } \texttt{True},$$
$$\forall x, x^2 + 3x + 1 > 0 \text{ is } \texttt{False}.$$

What should we expect for this?

$$\text{Input: } \forall x, x^2 + bx + 1 > 0$$
$$\text{Output: } (-2 < b) \wedge (b < 2)$$

The answer must depend on b.



Partially quantified formulae example

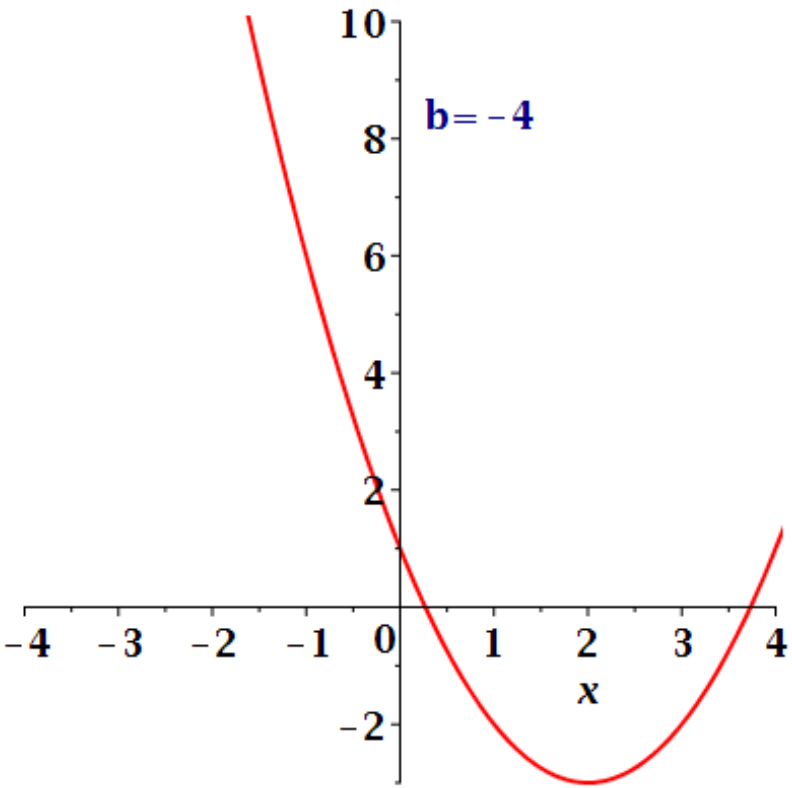
So we have that

$$\forall x, x^2 + 1 > 0 \text{ is } \texttt{True},$$
$$\forall x, x^2 + 3x + 1 > 0 \text{ is } \texttt{False}.$$

What should we expect for this?

$$\text{Input: } \forall x, x^2 + bx + 1 > 0$$
$$\text{Output: } (-2 < b) \wedge (b < 2)$$

The answer must depend on b.



Partially quantified Tarski formulae

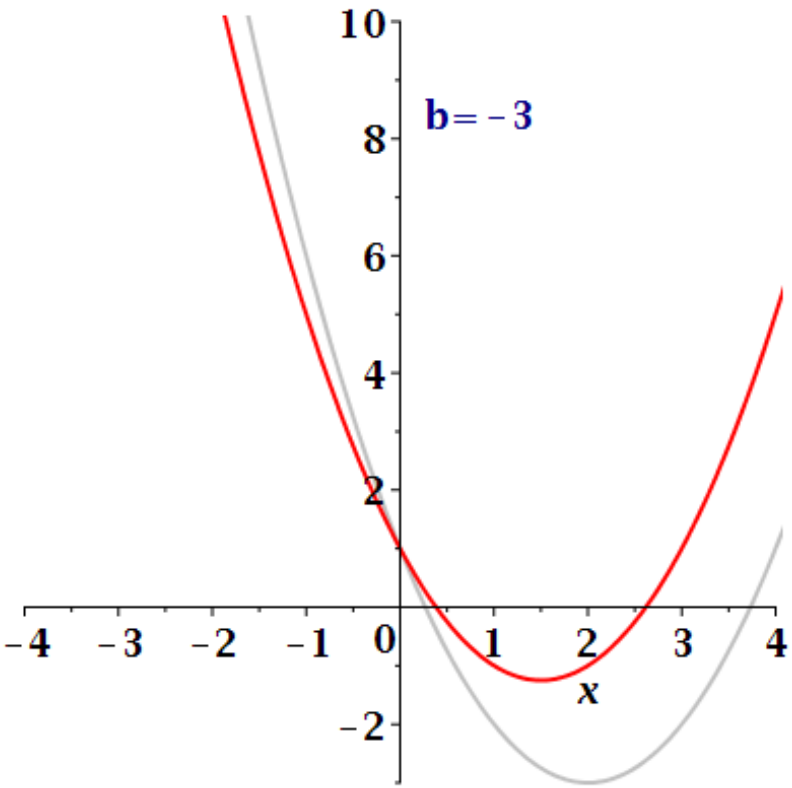
So we have that

$$\forall x, x^2 + 1 > 0 \text{ is } \texttt{True} \ ,$$
$$\forall x, x^2 + 3x + 1 > 0 \text{ is } \texttt{False} \ .$$

What should we expect for this?

$$\text{Input: } \forall x, x^2 + bx + 1 > 0$$
$$\text{Output: } (-2 < b) \wedge (b < 2)$$

The answer must depend on b.



Partially quantified Tarski formulae

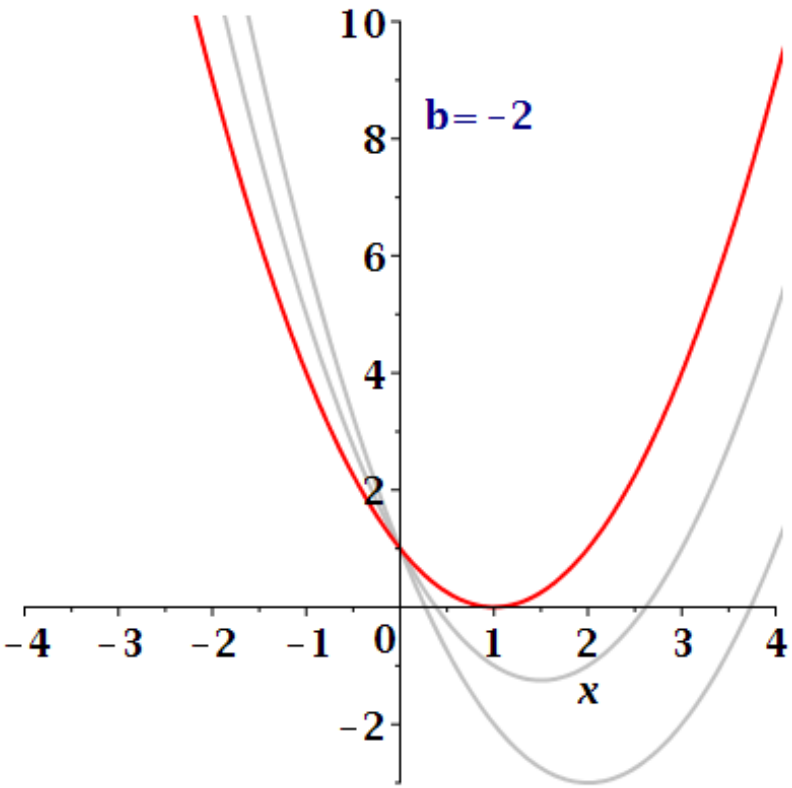
So we have that

$$\forall x, x^2 + 1 > 0 \text{ is } \texttt{True} \ ,$$
$$\forall x, x^2 + 3x + 1 > 0 \text{ is } \texttt{False} \ .$$

What should we expect for this?

$$\text{Input: } \forall x, x^2 + bx + 1 > 0$$
$$\text{Output: } (-2 < b) \wedge (b < 2)$$

The answer must depend on b.



Partially quantified Tarski formulae

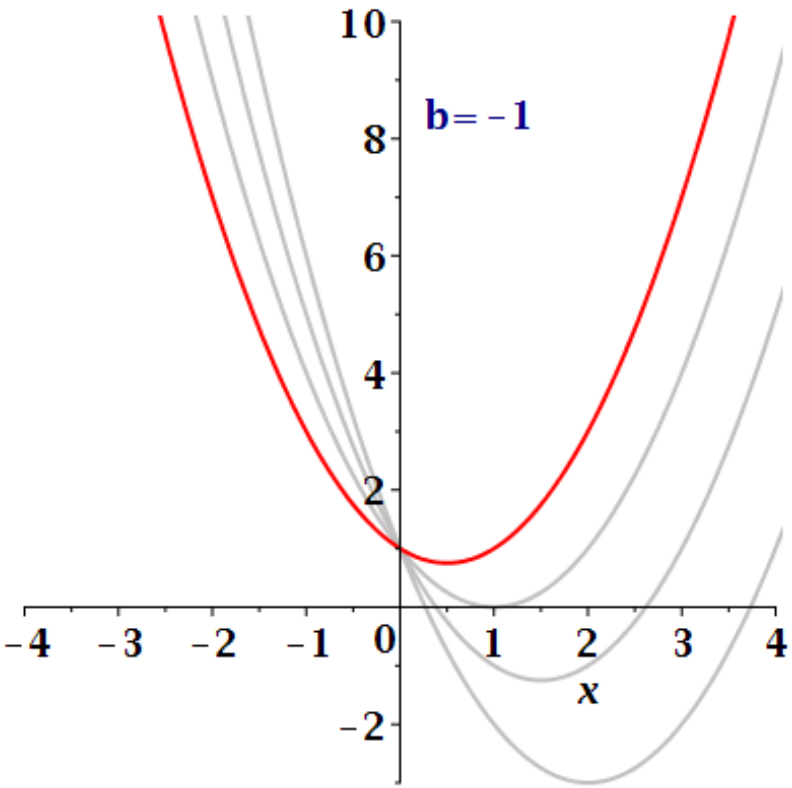
So we have that

$$\forall x, x^2 + 1 > 0 \text{ is } \texttt{True} \ ,$$
$$\forall x, x^2 + 3x + 1 > 0 \text{ is } \texttt{False} \ .$$

What should we expect for this?

$$\text{Input: } \forall x, x^2 + bx + 1 > 0$$
$$\text{Output: } (-2 < b) \wedge (b < 2)$$

The answer must depend on b.



Partially quantified Tarski formulae

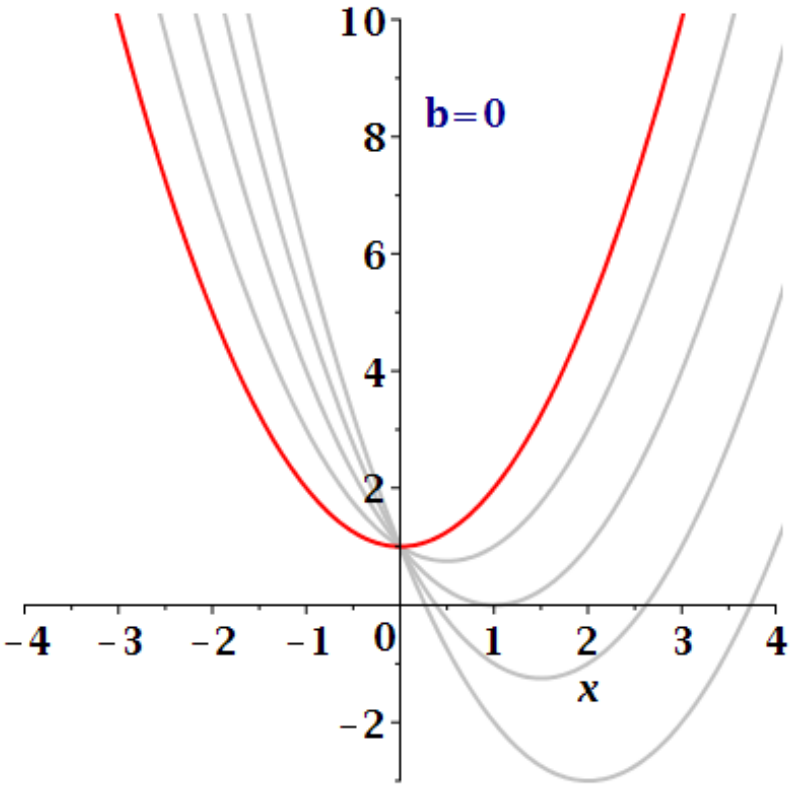
So we have that

$$\forall x, x^2 + 1 > 0 \text{ is } \texttt{True} \ ,$$
$$\forall x, x^2 + 3x + 1 > 0 \text{ is } \texttt{False} \ .$$

What should we expect for this?

$$\text{Input: } \forall x, x^2 + bx + 1 > 0$$
$$\text{Output: } (-2 < b) \wedge (b < 2)$$

The answer must depend on b.



Partially quantified Tarski formulae

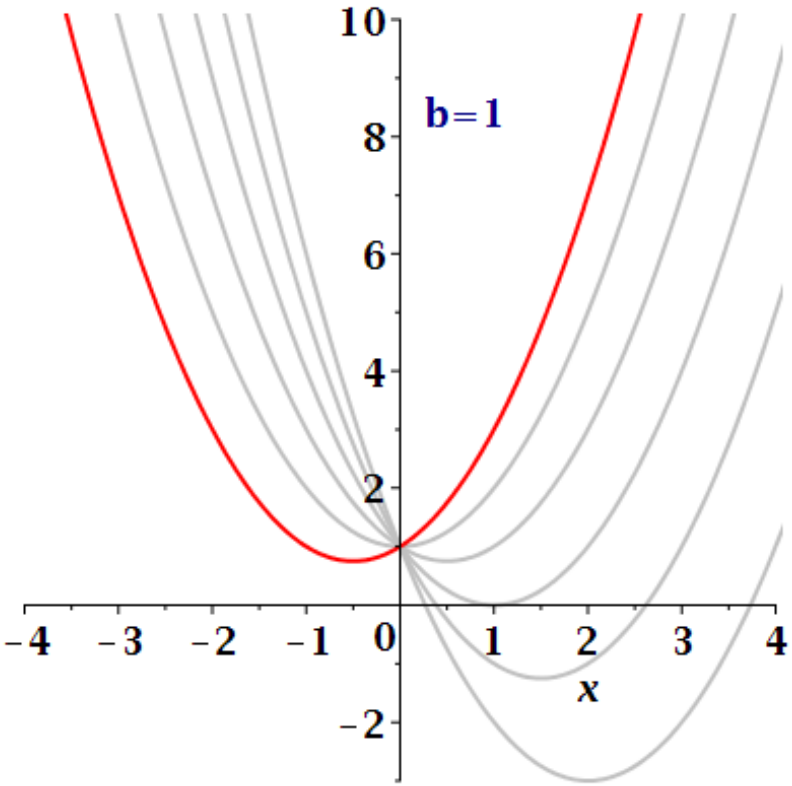
So we have that

$$\forall x, x^2 + 1 > 0 \text{ is } \texttt{True} \ ,$$
$$\forall x, x^2 + 3x + 1 > 0 \text{ is } \texttt{False} \ .$$

What should we expect for this?

$$\text{Input: } \forall x, x^2 + bx + 1 > 0$$
$$\text{Output: } (-2 < b) \wedge (b < 2)$$

The answer must depend on b.



Partially quantified Tarski formulae

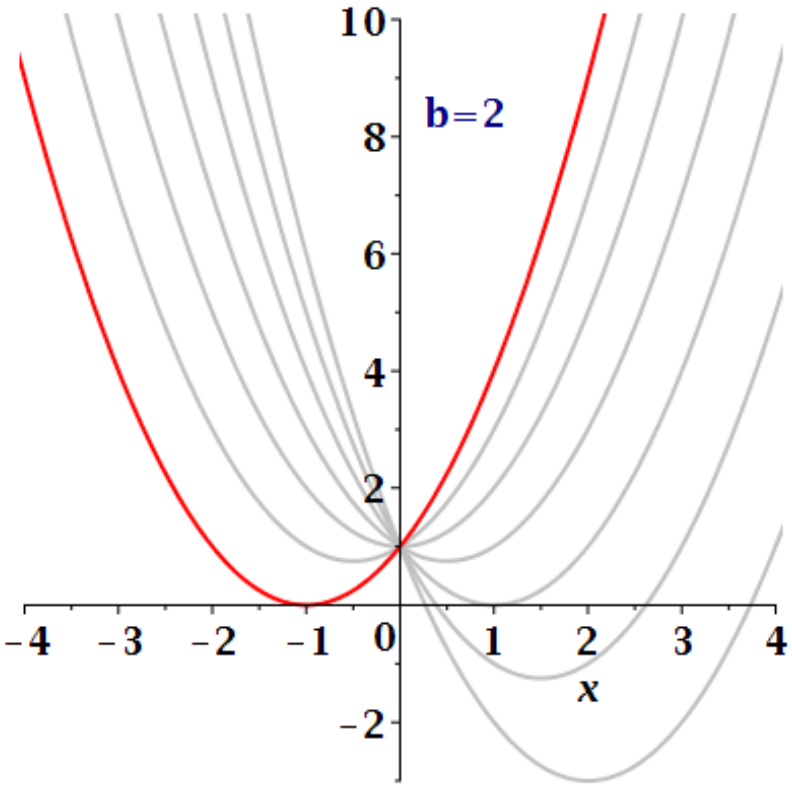
So we have that

$$\forall x, x^2 + 1 > 0 \text{ is } \texttt{True} \ ,$$
$$\forall x, x^2 + 3x + 1 > 0 \text{ is } \texttt{False} \ .$$

What should we expect for this?

$$\text{Input: } \forall x, x^2 + bx + 1 > 0$$
$$\text{Output: } (-2 < b) \wedge (b < 2)$$

The answer must depend on b.



Partially quantified Tarski formulae

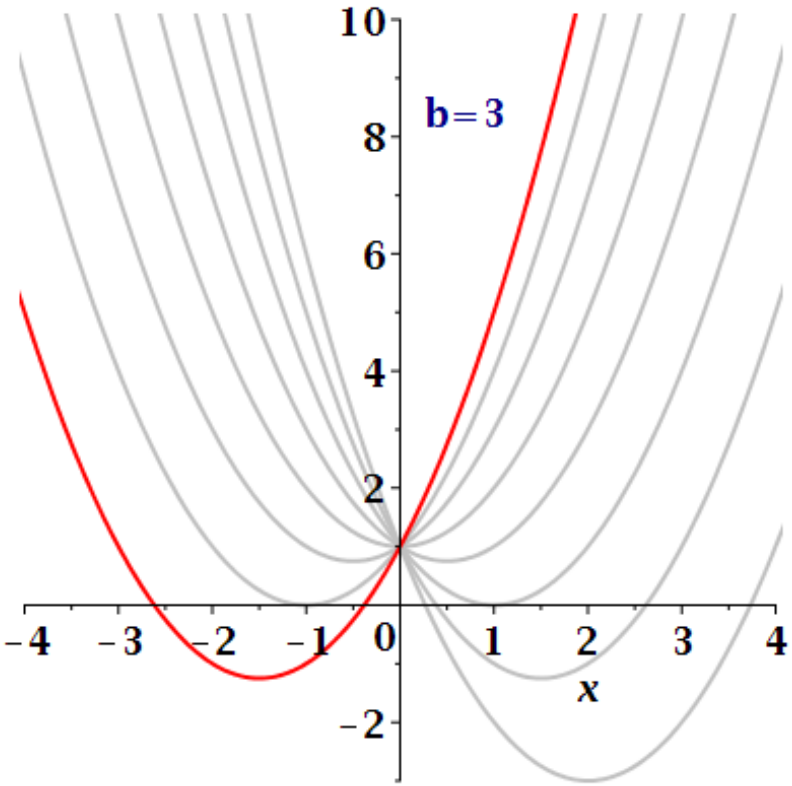
So we have that

$$\forall x, x^2 + 1 > 0 \text{ is } \texttt{True} \ ,$$
$$\forall x, x^2 + 3x + 1 > 0 \text{ is } \texttt{False} \ .$$

What should we expect for this?

$$\text{Input: } \forall x, x^2 + bx + 1 > 0$$
$$\text{Output: } (-2 < b) \wedge (b < 2)$$

The answer must depend on b.



Partially quantified Tarski formulae

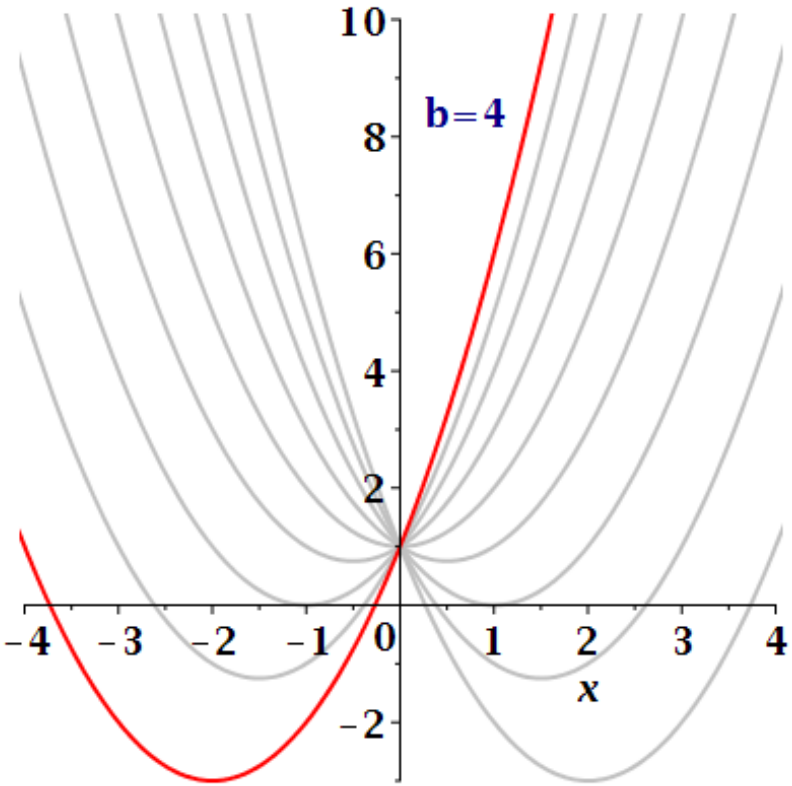
So we have that

$$\forall x, x^2 + 1 > 0 \text{ is } \texttt{True} \ ,$$
$$\forall x, x^2 + 3x + 1 > 0 \text{ is } \texttt{False} \ .$$

What should we expect for this?

$$\text{Input: } \forall x, x^2 + bx + 1 > 0$$
$$\text{Output: } (-2 < b) \wedge (b < 2)$$

The answer must depend on b.



Partially quantified Tarski formulae

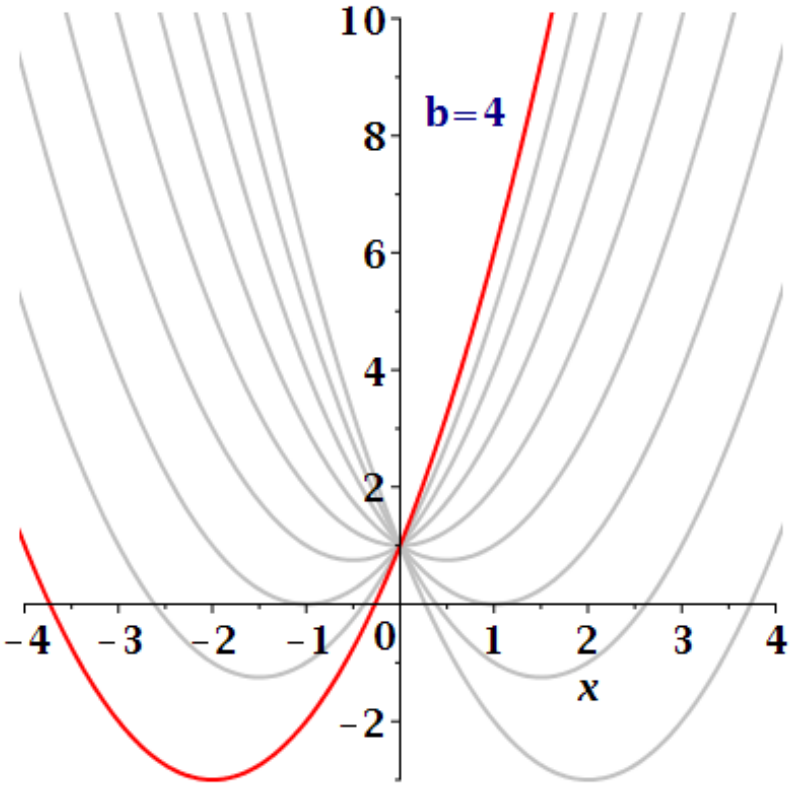
So we have that

$$\forall x, x^2 + 1 > 0 \text{ is } \texttt{True} \ ,$$
$$\forall x, x^2 + 3x + 1 > 0 \text{ is } \texttt{False} \ .$$

What should we expect for this?

$$\text{Input: } \forall x, x^2 + bx + 1 > 0$$
$$\text{Output: } (-2 < b) \wedge (b < 2)$$

The answer must depend on b.



QE Complexity

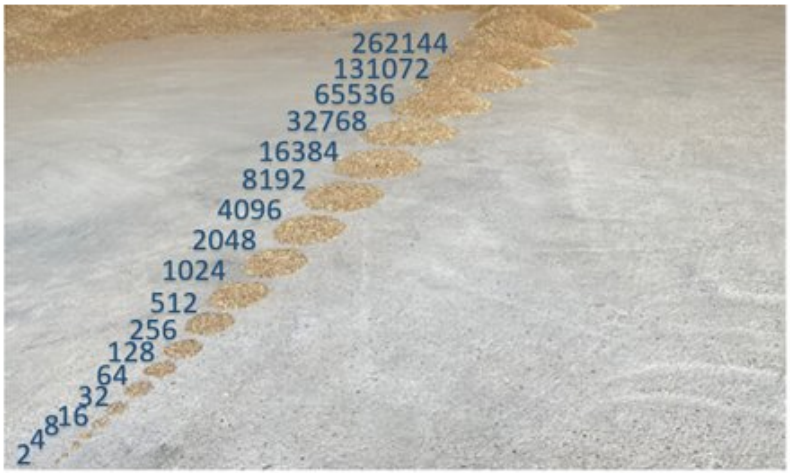
It is possible to write algorithms to perform QE. These algorithms are algebraic, i.e. they manipulate the polynomials used in the problem (they do not plot graphs). The answers produced are exact (no risk of floating point errors).

QE is always possible for formulae over the real numbers. But the problem is **proven** to be doubly exponential, i.e. $\mathcal{O}(2^{2^n})$ in the number of variables n used in the polynomials.

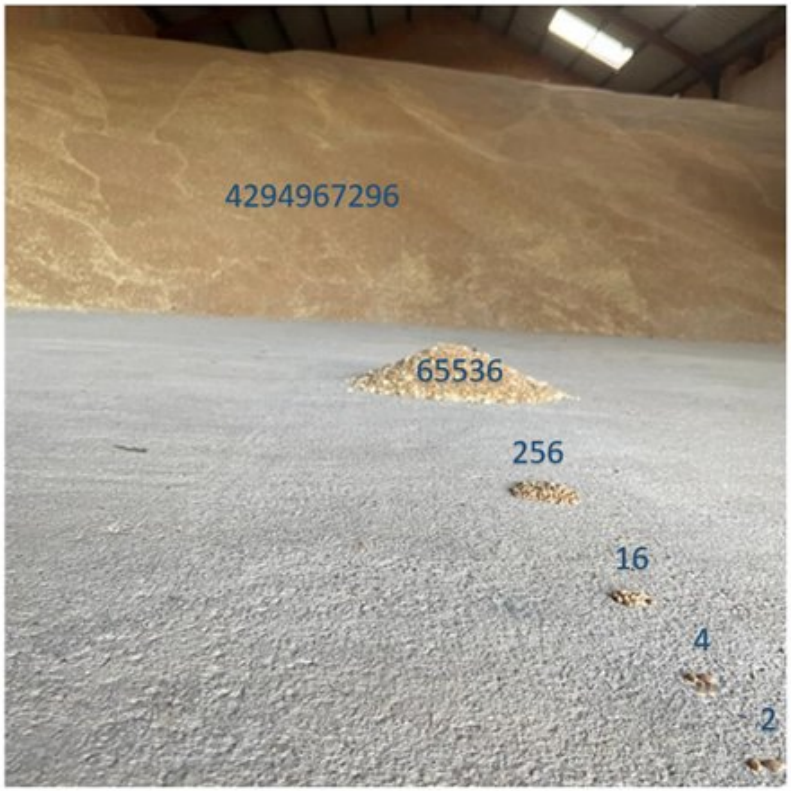
This is a statement on the problem: i.e. it is proven impossible to solve the problem with an algorithm of lower complexity class. The proof shows a QE problem where it takes doubly exponentially many symbols to write down the answer: so any algorithm that can solve it must be doubly exponential in printing the answer alone!

The Doubly Exponential Wall

Images by
Tereso del Rio



Exponential Growth



Doubly Exponential Growth

Any value in a doubly exponential algorithm? (Slide 30/36)

2^{2^4} is less than 66,000 while 2^{2^5} is over 4 billion! So are such algorithms not hopeless?

- Big-O doesn't kick in for a while so small n is fine.
- Lots of special cases where the Big-O doesn't apply.
- Many of the problems tackled are one-off calculations so we don't mind paying a big one-off computational price.
- Some of the problems are so important, that answers are worth waiting for: chip verification; safety critical system analysis; mathematical theorem proving etc.

The most used algorithm for QE involved **Cylindrical Algebraic Decomposition**: where real space is decomposed according to the zeros of the polynomials involved, in such a way that the cells align in cylinders with respect to a variable ordering.

Any value in a doubly exponential algorithm? (Slide 30/36)

2^{2^4} is less than 66,000 while 2^{2^5} is over 4 billion! So are such algorithms not hopeless?

- Big-O doesn't kick in for a while so small n is fine.
- Lots of special cases where the Big-O doesn't apply.
- Many of the problems tackled are one-off calculations so we don't mind paying a big one-off computational price.
- Some of the problems are so important, that answers are worth waiting for: chip verification; safety critical system analysis; mathematical theorem proving etc.

The most used algorithm for QE involved **Cylindrical Algebraic Decomposition**: where real space is decomposed according to the zeros of the polynomials involved, in such a way that the cells align in cylinders with respect to a variable ordering.

Any value in a doubly exponential algorithm? (Slide 30/36)

2^{2^4} is less than 66,000 while 2^{2^5} is over 4 billion! So are such algorithms not hopeless?

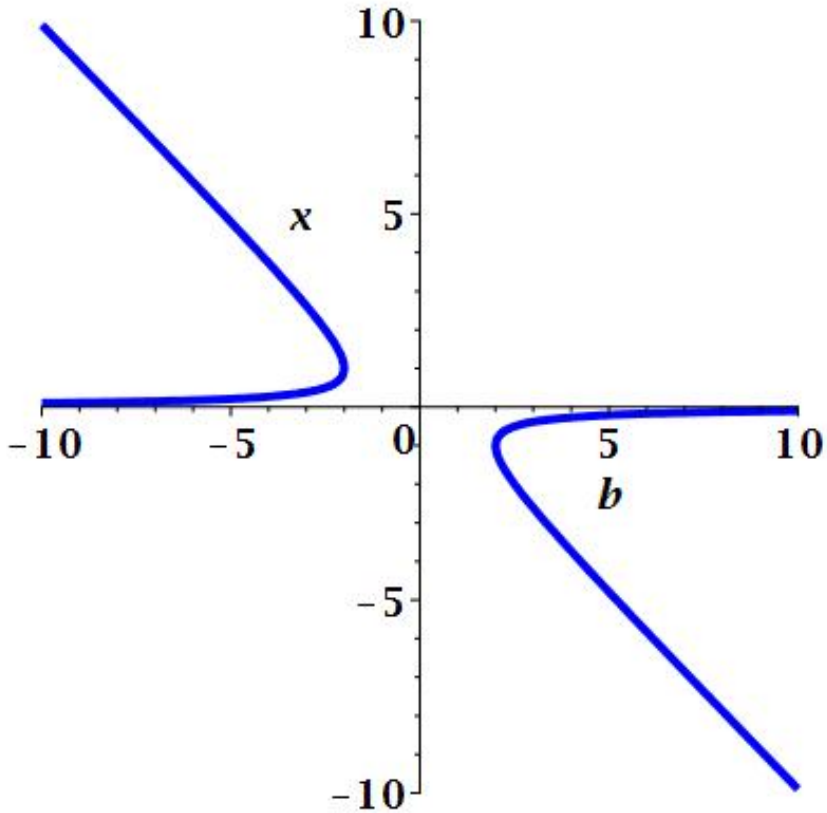
- Big-O doesn't kick in for a while so small n is fine.
- Lots of special cases where the Big-O doesn't apply.
- Many of the problems tackled are one-off calculations so we don't mind paying a big one-off computational price.
- Some of the problems are so important, that answers are worth waiting for: chip verification; safety critical system analysis; mathematical theorem proving etc.

The most used algorithm for QE involved **Cylindrical Algebraic Decomposition**: where real space is decomposed according to the zeros of the polynomials involved, in such a way that the cells align in cylinders with respect to a variable ordering.

QE via CAD Example

How to determine with CAD?

$$\exists x, x^2 + bx + 1 \leq 0$$



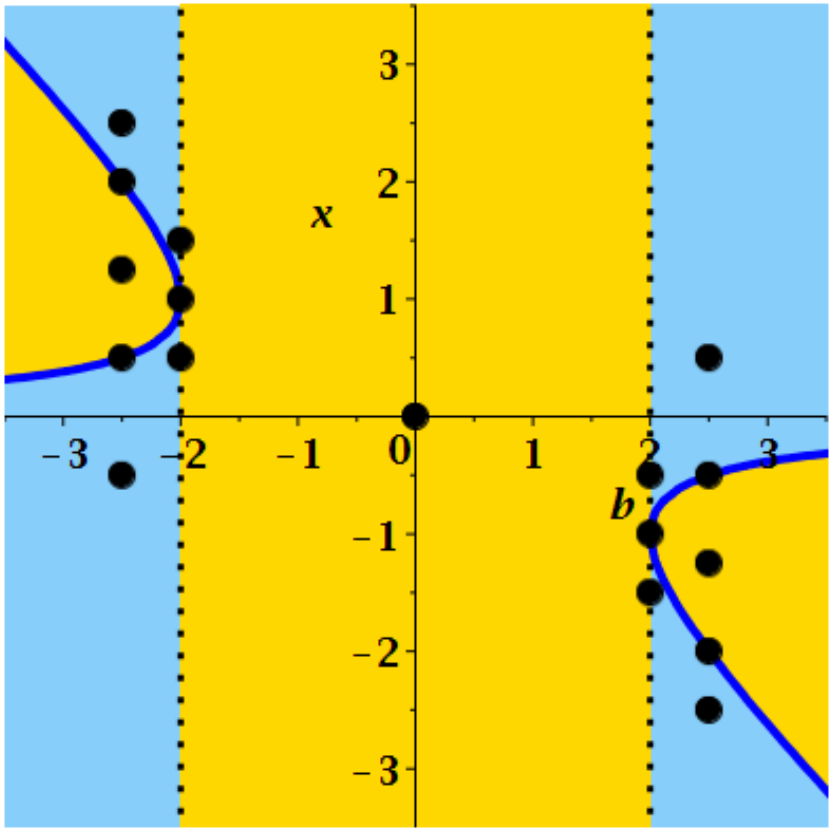
QE via CAD Example

How to determine with CAD?

$$\exists x, x^2 + bx + 1 \leq 0$$

To solve we:

Build a sign-invariant CAD for $f = x^2 + bx + 1$.



QE via CAD Example

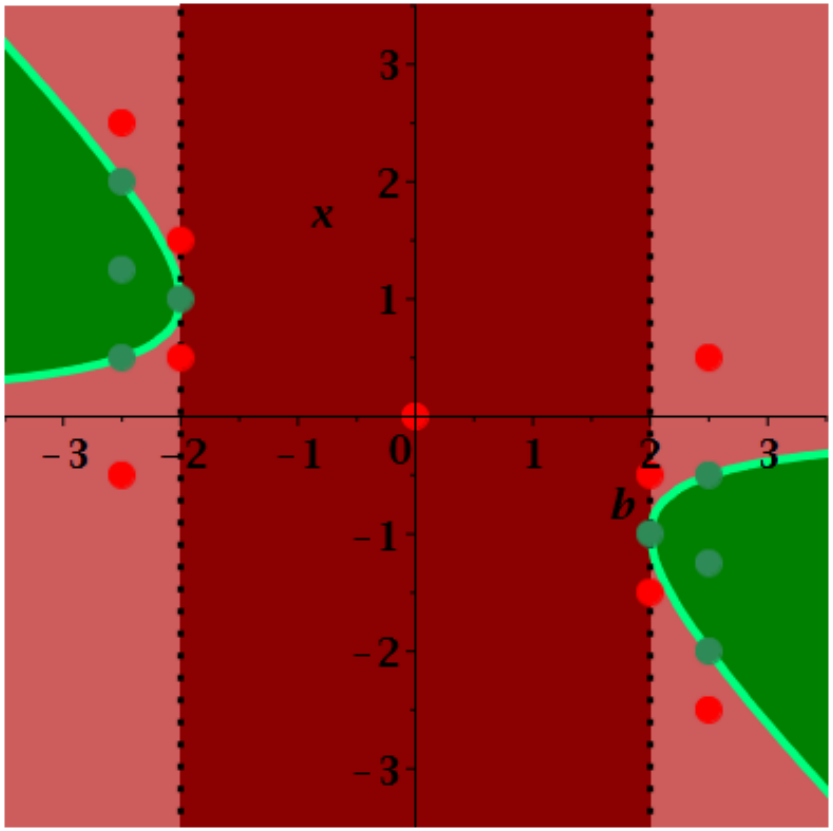
How to determine with CAD?

$$\exists x, x^2 + bx + 1 \leq 0$$

To solve we:

Build a sign-invariant CAD for $f = x^2 + bx + 1$.

Tag each cell true or false according to $f \leq 0$.



QE via CAD Example

How to determine with CAD?

$$\exists x, x^2 + bx + 1 \leq 0$$

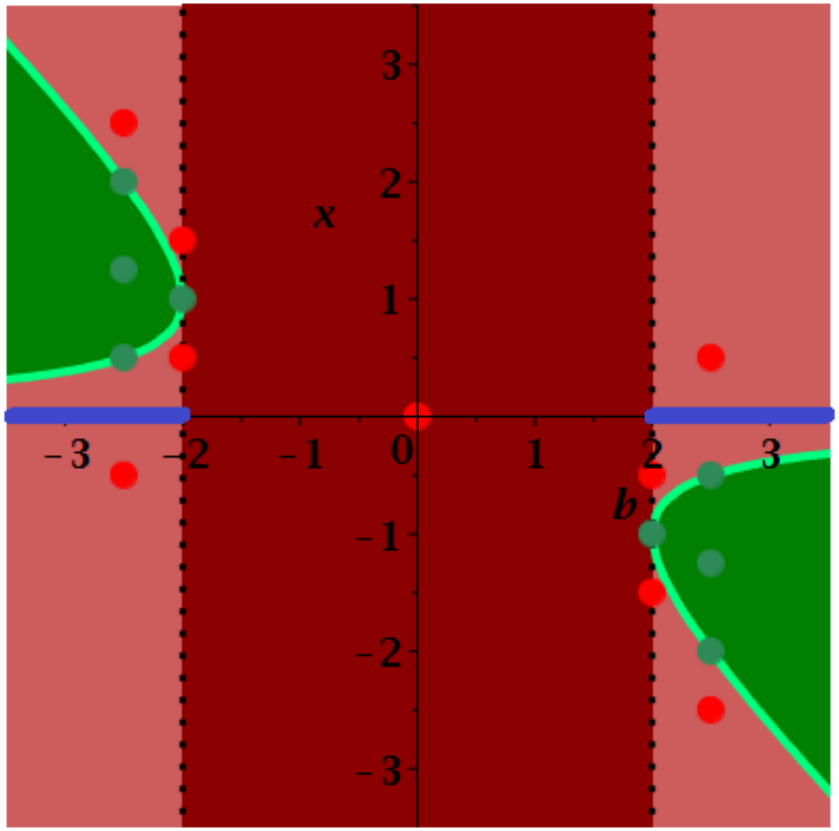
To solve we:

Build a sign-invariant CAD for $f = x^2 + bx + 1$.

Tag each cell true or false according to $f \leq 0$.

Take disjunction of projections of true cells:

$$b < -2 \vee b = -2 \vee b = 2 \vee b > 2$$



QE via CAD Example

How to determine with CAD?

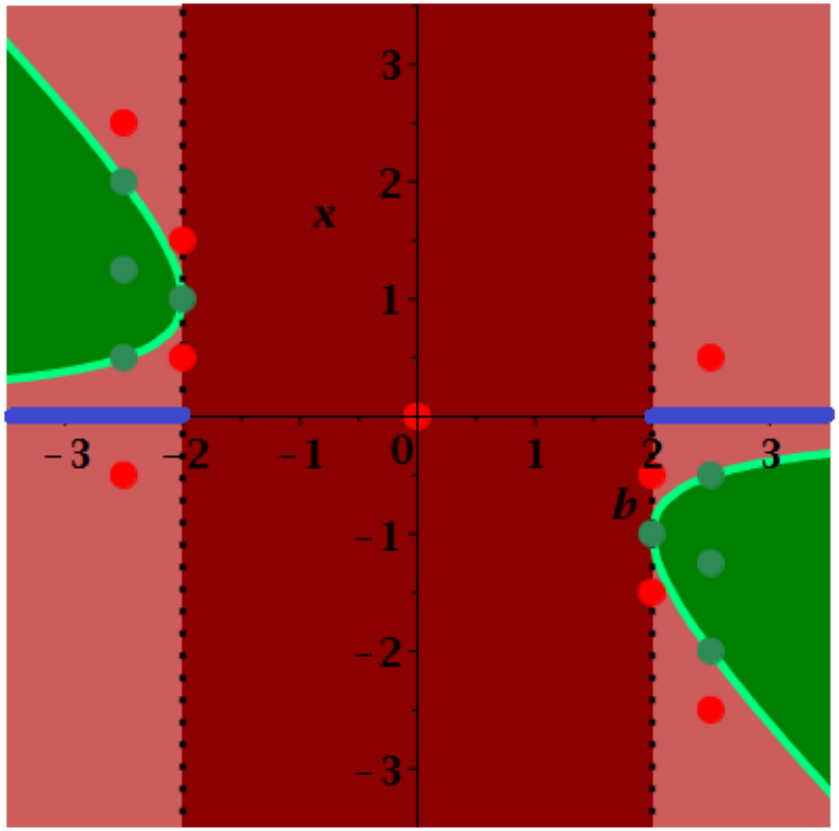
$$\exists x, x^2 + bx + 1 \leq 0$$

To solve we:

Build a sign-invariant CAD for $f = x^2 + bx + 1$.

Tag each cell true or false according to $f \leq 0$.

Take disjunction of projections of true cells:



$$\Longrightarrow$$

$$b \leq -2 \vee b \geq 2$$

Universal QE via CAD

In general we can perform Real QE on a decomposition as follows.

- Eliminate existential quantifiers by projecting the true cells, as in the previous example.
- Eliminate universal quantifiers by using the relation

$$\forall x P(x) = \neg \exists x \neg P(x)$$

and then proceeding with existential QE.

Recall our original example was $\forall x, x^2 + bx + 1 > 0$. The above leads us to study $\exists x, x^2 + bx + 1 \leq 0$ which from the previous slide we know has solution $b \leq -2 \vee b \geq +2$. The solution to our universally quantified problem is then the negation of this: $-2 < b \wedge b < +2$ (as we found numerically earlier).

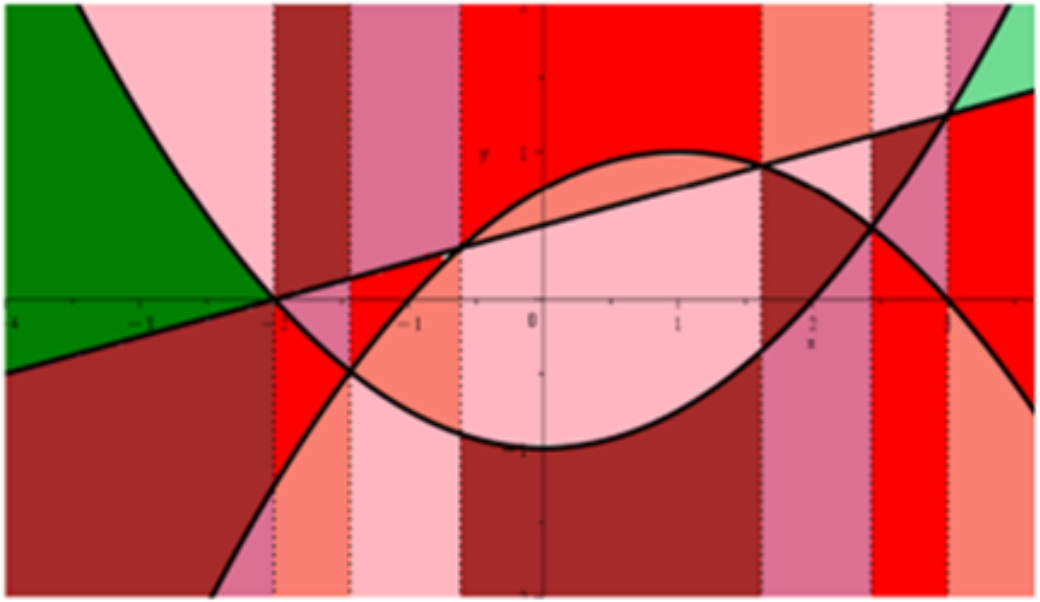
The DEWCAD Project

CU leads the EPSRC funded DEWCAD Project: *Pushing Back the Doubly-Exponential Wall of Cylindrical Algebraic Decomposition*.

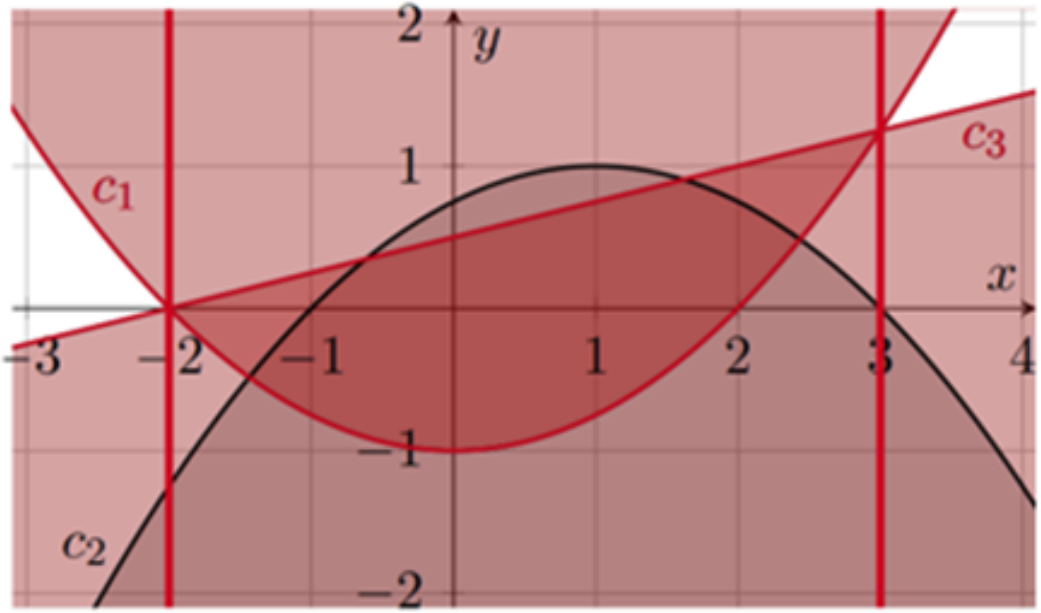
Since CAD solves the QE problem it must have doubly exponential complexity in the worst case. Better in the average case?
We cannot destroy the QE wall; but we can delay the doubly exponential behaviour to allow many more problems to be solved.
We do this by combing CAD with SAT-solvers; and redesigning the CAD algorithm using the ideas behind SAT solvers.

For more information, see this recent survey paper:

M. England.
Recent Developments in Real Quantifier Elimination and Cylindrical Algebraic Decomposition (Extended Abstract of Invited Talk).
Computer Algebra in Scientific Computing (Proc. CASC '24), pp. 1-10. (Lecture Notes in Computer Science, vol 14938). Springer, 2024.
https://doi.org/10.1007/978-3-031-69070-9_1



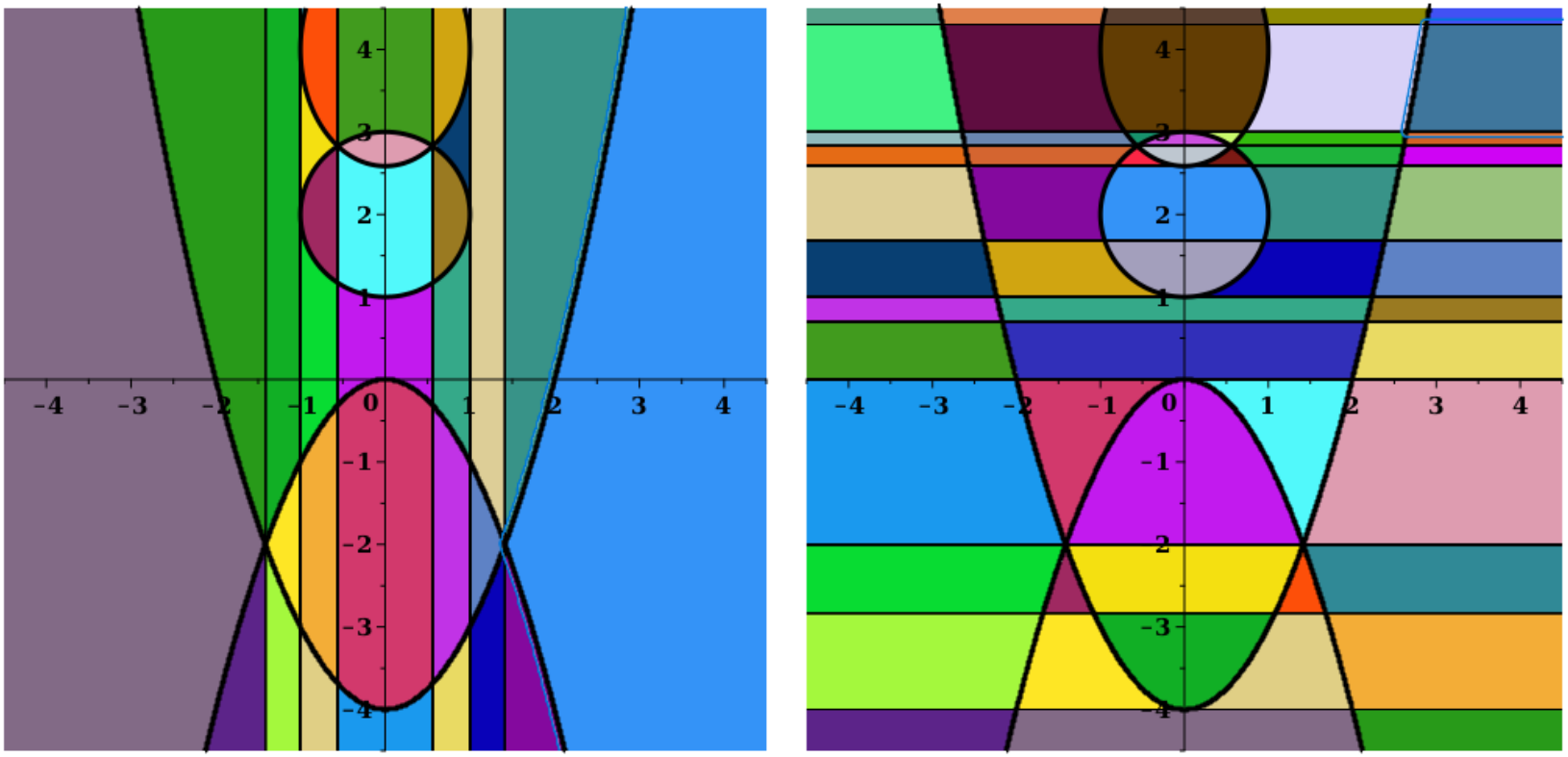
Cell construction in traditional CAD



Cell construction in conflict driven cylindrical coverings

Machine Learning for Quantifier Elimination (Slide 35/36)

QE via CAD works by (algebraically) decomposing space: different decompositions are possible (some much smaller than others).



Coventry University led an EPSRC Project: *Embedding Machine Learning within Quantifier Elimination Procedures* to use ML to guide the algorithm to a small decomposition.

PhD Project on Machine Learning for CAD (Slide 36/36)

Coventry University student Tereso del Río studied this further for his PhD, identifying an optimal ML pipeline for the task.

He also used explainable AI to explore how ML made the decision. The explanations allowed him to synthesis short human code which:

(a) performed almost as well as the ML on our dataset;
(b) but can now be used independently of ML technology; and
(c) allows better generalisation to other datasets than the ML.

L. Pickering, T. del Río Almajano, M. England and K. Cohen.
Explainable AI Insights for Symbolic Computation: A case study on selecting the variable ordering for cylindrical algebraic decomposition.
Journal of Symbolic Computation, **123**, Article Number 102276, 2024.

The End

Matthew.England@coventry.ac.uk

https://matthewengland.coventry.domains/