

The NP-hard days of our lives (reloaded)

Armando Tacchella

www.star.dist.unige.it

www.star-lab.it

NED
THE MATRIX: RELOADED
VFX SHOT

WWW.THEMATRIX.COM

Overview

- Part I: Introduction to complexity theory
 - Problems, algorithms, programs and computers
 - Models of computation
 - Computability and (in)tractability
- Part II: A case study on intractability
 - Propositional satisfiability (SAT)
 - State of the art: desperately seeking the silver bullet
 - Hi-NRG: making SAT tractable in practice?

MORPHEUS ON THE FREEWAY
THE MATRIX RELOADED
VFX SHOT

WWW.THEMATRIX.COM



Part I: Introduction to complexity theory



A computer scientist's life

We are given **PROBLEMS** that we solve by writing **ALGORITHMS** that we implement in **PROGRAMS** which run on **COMPUTERS**.

Why don't we get a **REAL** life?

Interesting question! We will not answer it in this seminar. The answer is left as an easy exercise to the attendees. 😊



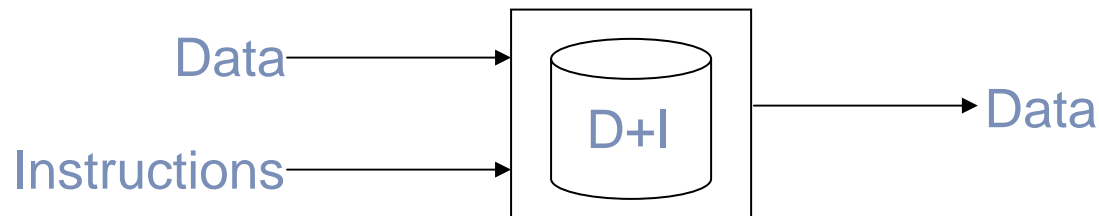
What is a COMPUTER?

“A device that computes, especially a programmable electronic machine that performs high-speed mathematical or logical operations or that assembles, stores, correlates, or otherwise processes information.”

The American Heritage® Dictionary of the English Language, Fourth Edition

“A general purpose device that can store and process coded information (data) and whose operation can be controlled using specific coded directives (instructions).”

Armando Tacchella, 2005



What is a PROGRAM?

“A set of coded instructions that enables a machine, especially a computer, to perform a desired sequence of operations.”

The American Heritage® Dictionary of the English Language, Fourth Edition

Ok, technically this is an **imperative** view of programming.

Alternative definitions for programs:

- transformations from input to output data (**functional** view)
- sets of interacting objects (**object-oriented** view)
- descriptions in terms of facts and rules (**logic-based** view)

Programming Language = instructions + syntax + semantics



What is an ALGORITHM?

A **step-by-step** problem-solving **procedure**, especially an **established, recursive computational procedure** for solving a problem in a finite number of steps.

The American Heritage® Dictionary of the English Language, Fourth Edition

Example: Euclid's algorithm

Input: two positive integers a, b s.t. $a \geq b \geq 0$

Output: the GCD of a and b

1. if b is equal to 0 STOP; a contains the GCD
2. save the value of b into a temporary t , calculate the new value of b as $a \bmod b$, and let the new value of a be t
3. repeat from step 1



ALGORITHMS vs. PROGRAMS

Algorithms

- are machine-independent
- do not require a programming language for their formulation
- provide an abstract view of computation

Programs

- are machine-dependent
- are formulated in terms of some programming language
- provide a concrete view of computation

We require that **ALGORITHMS = PROGRAMS**

The programming language is an implicit description of the set of possible algorithms that we can reason about.



What is a PROBLEM?

A question to be considered, solved, or answered.

The American Heritage® Dictionary of the English Language, Fourth Edition

A question to be answered, usually possessing several parameters, or free variables, whose values are left unspecified.

A problem is described by giving:

- (1) a description of its parameters
- (2) a statement of what properties the solution is required to satisfy

An instance is obtained by specifying values for the problem parameters

M.R. Garey and D.S. Johnson Computers and Intractability, 1979

Example: Sorting sequences

- (1) A sequence a of N integers, where $a[i]$ denotes the i -th element
- (2) A permutation s.t. $a[i] \leq a[i+1]$ for all $1 \leq i < N$



Fundamental aspects

COMPUTABILITY

Given a problem, can we write a program to solve it?

COMPLEXITY

How much of a given resource does a program require?

Resources in computer science (maybe also in real life? 😊)

Time (how many atomic **computation steps**)

Space (how many atomic **elements of storage**)

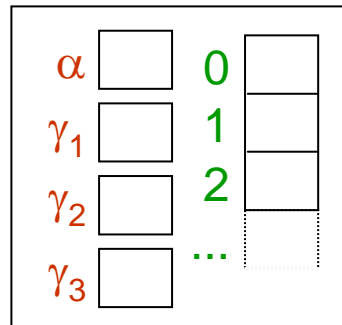
The answers to computability and complexity issues depend on the **COMPUTER!**



The computer: theory & practice



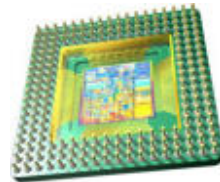
theory



Random Access
Machine



practice



CPU

Salient features of RAMs

- 4 registers:
 - accumulator α
 - index registers $\gamma_1, \gamma_2, \gamma_3$
- Infinite set of storage locations numbered 0, 1, 2, ...
- Registers and storage locations hold arbitrary precision integers

Plausible model: just assume you have enough memory and math precision in your CPU!

Computing with RAMs: architecture

Load and store

$\langle \text{reg} \rangle \leftarrow \langle \text{op} \rangle$

$\alpha \leftarrow \langle \text{mop} \rangle$

$\langle \text{op} \rangle \leftarrow \langle \text{reg} \rangle$

$\langle \text{mop} \rangle \leftarrow \alpha$

Jump

goto k

if $\langle \text{reg} \rangle \pi \langle \text{op} \rangle$ goto k

$\pi \in \{=, \neq, \geq, <, \leq, >\}$

Arithmetic instructions

$\alpha \leftarrow \alpha \bullet \langle \text{mop} \rangle$

$\bullet \in \{+, -, *, \text{div}, \text{mod}\}$

Index register

$\gamma_j \leftarrow \gamma_j \pm i$

$1 \leq j \leq 3, i \in 0, 1, 2, \dots$

Notation

$\langle \text{reg} \rangle$ is either α or $\gamma_j, 1 \leq j \leq 3$

$\rho(i)$ is the content of location i

$\langle \text{op} \rangle$ is an operand: integer, $\rho(i)$ or $\langle \text{reg} \rangle$

$\langle \text{mop} \rangle$ is a modified operand $\rho(i + \gamma_j)$

RAM program

A sequence of instructions numbered $0, 1, 2, \dots$. Instruction 0 is executed first, then instruction $i+1$ for all $i \geq 0$, unless:

- instr. is a goto, or
- instr. i is an if and $\langle \text{reg} \rangle \pi \langle \text{op} \rangle$ is true in which case the execution continues from instruction k



Computing with RAMS: example

RAM program to compute 2^n
 n is initially stored in location 0
 The result is stored in location 1

```

0:  $\gamma_1 \leftarrow \rho(0)$ 
1:  $\alpha \leftarrow 1$ 
2: if  $\gamma_1 = 0$  goto 6
3:  $\alpha \leftarrow \alpha * 2$ 
4:  $\gamma_1 \leftarrow \gamma_1 - 1$ 
5: goto 2
6:  $\rho(1) \leftarrow \alpha$ 
  
```

This program demonstrates that on RAMs:

- we can compute 2^n , and
- such computation is feasible in $k_1 n + k_2$ atomic steps,
- using k_3 locations of storage (besides the input locations).



Models of computation

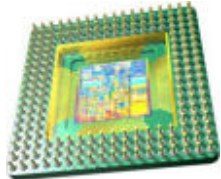


theory

RAM



practice



CPU

Computational Power

A machine M is more powerful than another M' if M can compute something that M' cannot

Are RAMs more powerful than CPUs? No!

Infinite memory
Arbitrary precision

The extensive use of these capabilities implies infinitely long computations

Are CPUs more powerful than RAMs? No!

State of the art
HW technology

Technological advances bring more speed but cannot increase the power of the CPU

RAMs are an effective model of computation

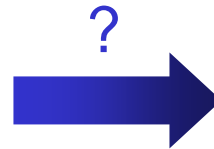


Seeking the essence of computation



theory

RAM



Is there a **model of computation** which:

- is **as powerful as RAMs**, and
- has a **more succinct definition**?



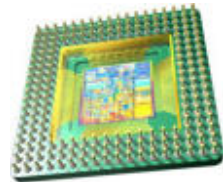
YES,
The Turing Machine!

Alan Mathison Turing

A.M. Turing's paper describing the **model of a programmable computer**, was published about 10 years before the first **practical Turing-complete computers** were built (Mark I and EDSAC, 1949)



practice



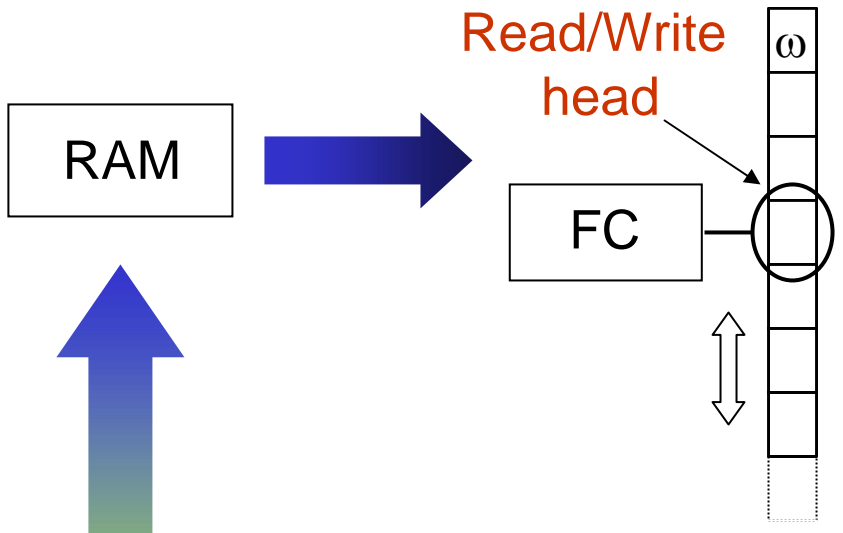
CPU



The Turing Machine



theory



I/O Tape

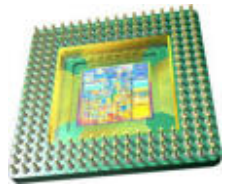
- Infinite number of cells
- Cells contain symbols out of fixed alphabet + 'β' (blank) and 'ω' (EOT)
- Moves: left, right, none

Finite Control (FC)

- Set of states + initial and halting states
- Transition function: given the current state and the symbol scanned by the R/W head, computes the next state, the symbol to be written on the I/O tape and the tape move.



practice



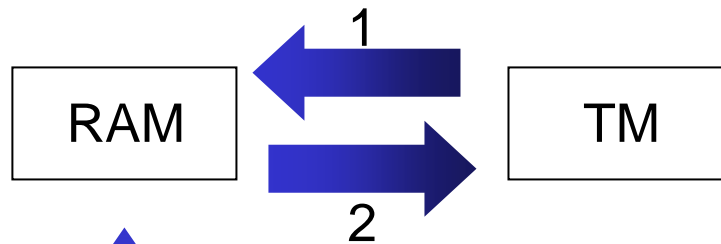
CPU



Turing Machine's muscles



theory



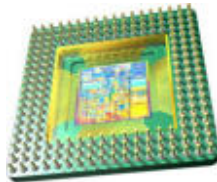
What is the relationship between TMs and RAMs?

(1) is straightforward: given some finite control we can write a RAM program that simulates it, and use the memory to simulate the I/O tape.

(2) is not straightforward, but it is true! If a RAM machine solves a problem in $f(n)$ steps, where n is the “size” of the problem, then there is a TM that solves it using up to $\text{poly}(f(n))$ transition steps.



practice

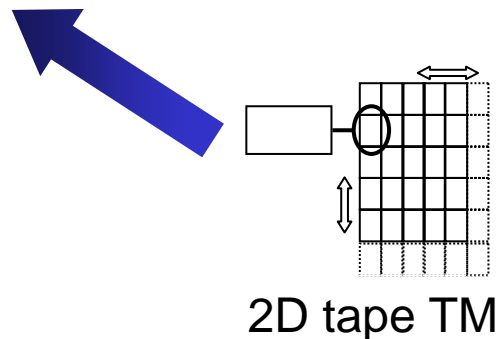
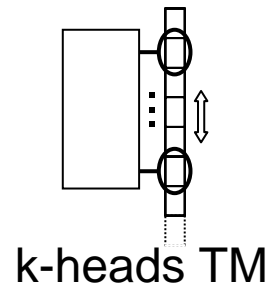
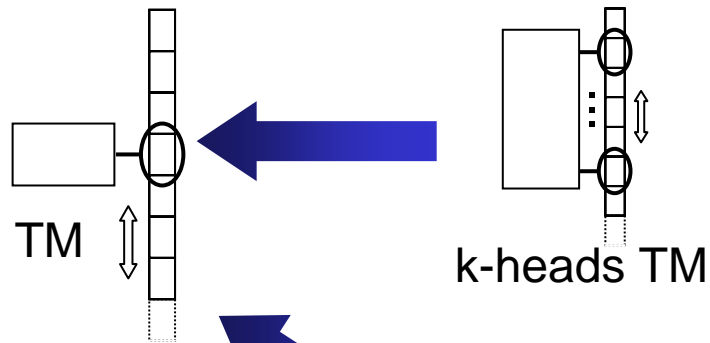
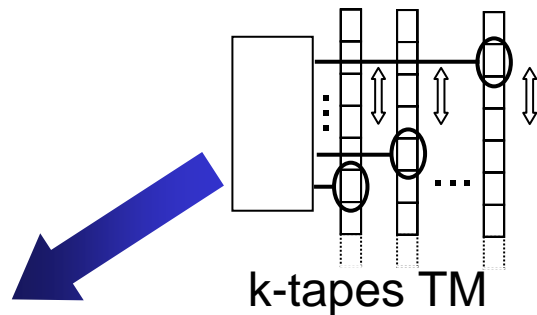


CPU

TMs are an effective model of computation



The Church-Turing Thesis (CTT)



Fact: the (basic) TM can simulate all its “realistic” extensions within a polynomial

Fact: TMs are equivalent to other models of computation discovered independently

CTT in A.M. Turing’s words: “Every function which would be regarded as computable can be computed by a Turing machine.”

CTT according to [Papadimitriou '93]
 “Any reasonable attempt to model computer algorithms and their performance is bound to end up with a model of computation and associated time cost that is equivalent to TMs within a polynomial.”



The geography of computability

“Hic sunt leones”
(Here there are lions)



Halting problem: does a given TM halt, i.e., terminate properly, on any input?



YOU ARE
HERE

GCD, sorting,
exponential, ...

(Turing-)Computable
problems

PROBLEMS



Dealing with complexity (I)

From the **complexity viewpoint** a problem is either:

TRACTABLE if it can be solved using reasonable amounts of time AND space, or

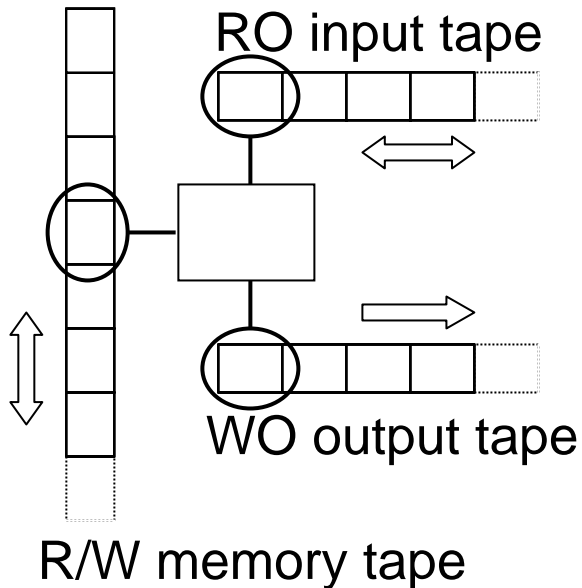
INTRACTABLE otherwise.

What is **reasonable**? Reasonable usually means that the **quantity of resource consumed** is at most **a polynomial function** on the “size” of **any problem instance**

What is **size**? Let's **use TMs** to formalize the concept!



TMs and complexity



Notation:

- x is the **input string** (the content of the input tape)
- $|x|=n$ is the **length of the input string**
- M is a **TM with input and output**
- **Time** is measured by the **number of transitions** to reach the **halting state** from the initial state
- **Space** is measured by the **number of “dirty” cells** in the **memory tape** when the machine **halts**

M operates within **time** $f(n)$ if, for any input string x , the **time required** by M on x is **at most** $f(|x|)$

TIME($f(n)$)

M operates within **space** $f(n)$ if, for any input string x , the **space required** by M on x is **at most** $f(|x|)$

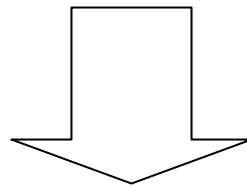
SPACE($f(n)$)



Formalizing (in)tractability

Let **PTIME** be the class of problems that can be solved by some TM that operates within $\text{TIME}(\text{poly}(n))$

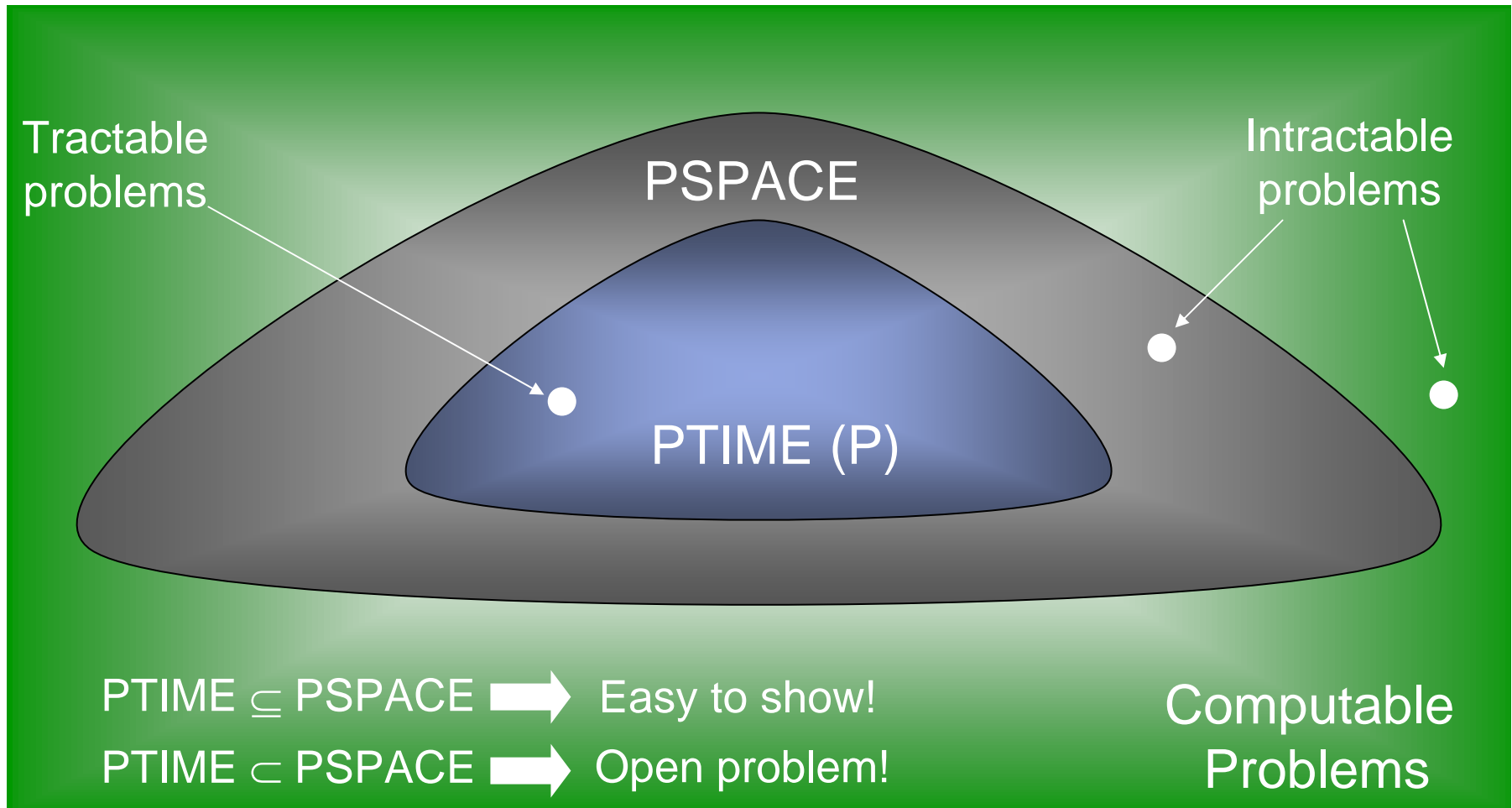
Let **PSPACE** be the class of problems that can be solved by some TM that operates within $\text{SPACE}(\text{poly}(n))$



A problem is tractable iff it is both in PTIME and in PSPACE, and untractable otherwise



The geography of complexity

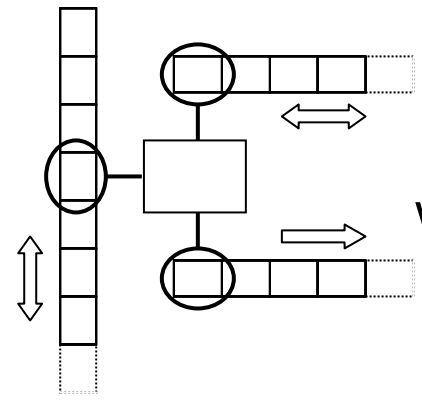


Complexity theory... in practice!



theory

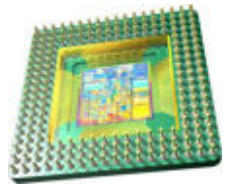
RAM



Deterministic TM with input and output



practice



CPU



If a problem is in P , then we are guaranteed to be able to write a program to solve it (in assembly, C, Java, Perl, your favorite language) that runs in polynomial time (and space) w.r.t the input "size"!



(Classic) Complexity theory: limits

If a problem is in P then it is guaranteed to admit a polynomial time (and space) algorithm to solve it

BUT

Proving that a program is (is not) in P **may not be easy!**

AND

Polynomial time can be **too coarse grained** as a **classification for practical purposes** (e.g., sorting with bubblesort or heapsort)

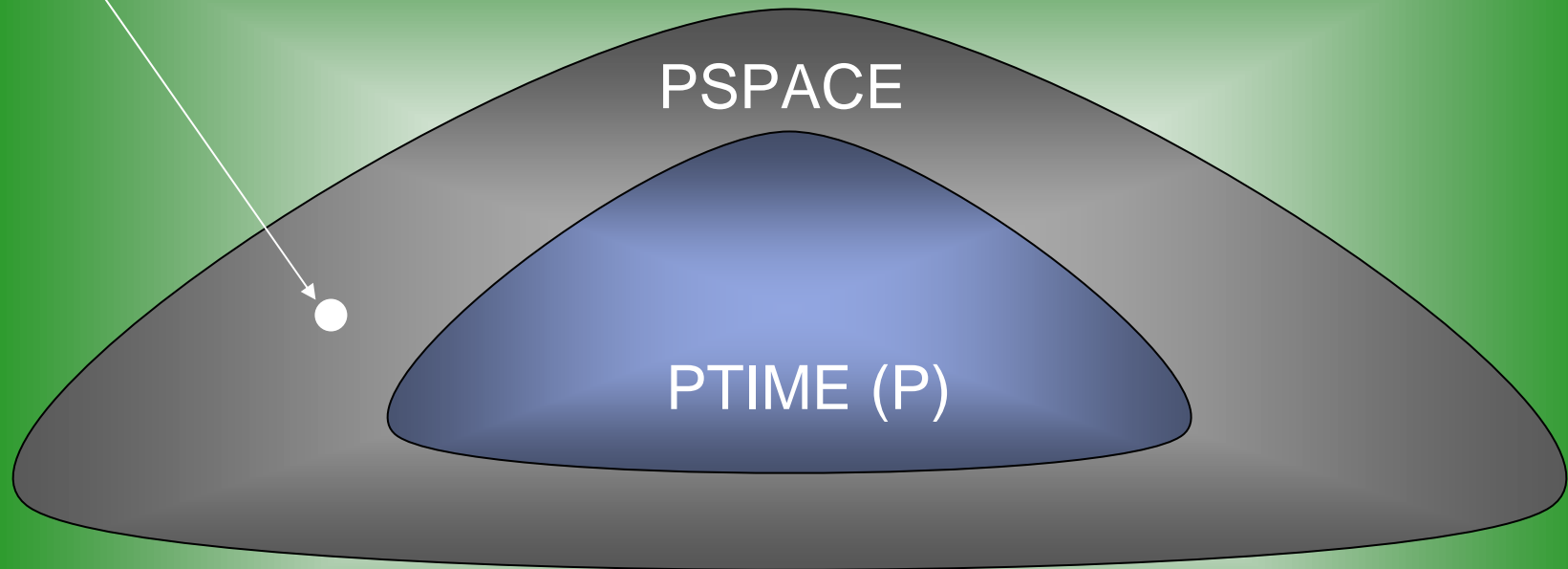
There's **more to it**:

- Finer grained complexity classes
- Parametric complexity theory
- Algorithmic theory and probabilistic complexity



Back to intractability...

These problems may require super-polynomial time,
but still their space requirements are reasonable



We are (implicitly) assuming that
 $P \subset \text{PSPACE}$, i.e., $\text{PSPACE} - P \neq \emptyset$

Computable
Problems

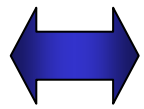


Modelling intractability

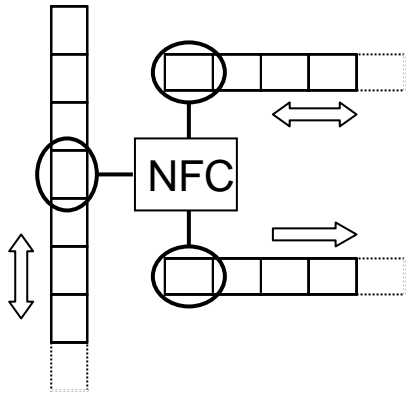
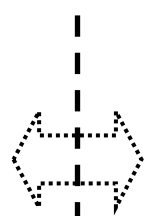


theory

RAM



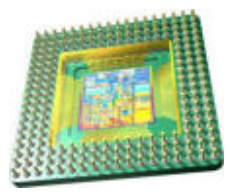
DTM



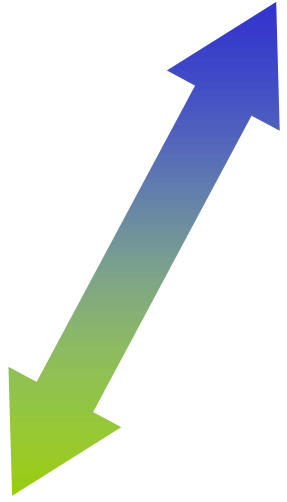
A different kind of computer?



practice



CPU



Nondeterministic TM (NTM)
 Memory and IO work the same as in DTM, but the finite control is **nondeterministic**: given the **current state** and the **symbols scanned by the heads**, there is more than one possible combination of state and tape moves from which the machine can choose **nondeterministically** the next configuration.



The power of non-determinism

Are nondeterministic TMs more powerful than TMs?

No! Any problem solvable in time $f(n)$ on a NTM can be solved on a DTM in time $k^{f(n)}$ with $k > 1$

Since recasting a problem from NTM to DTM involves an exponential blowup in time, NTMs are a way to characterize (reasonably the “easiest” class of) intractable problems

NPTIME (NP) is the class of problems solvable in polynomial time on a NTM

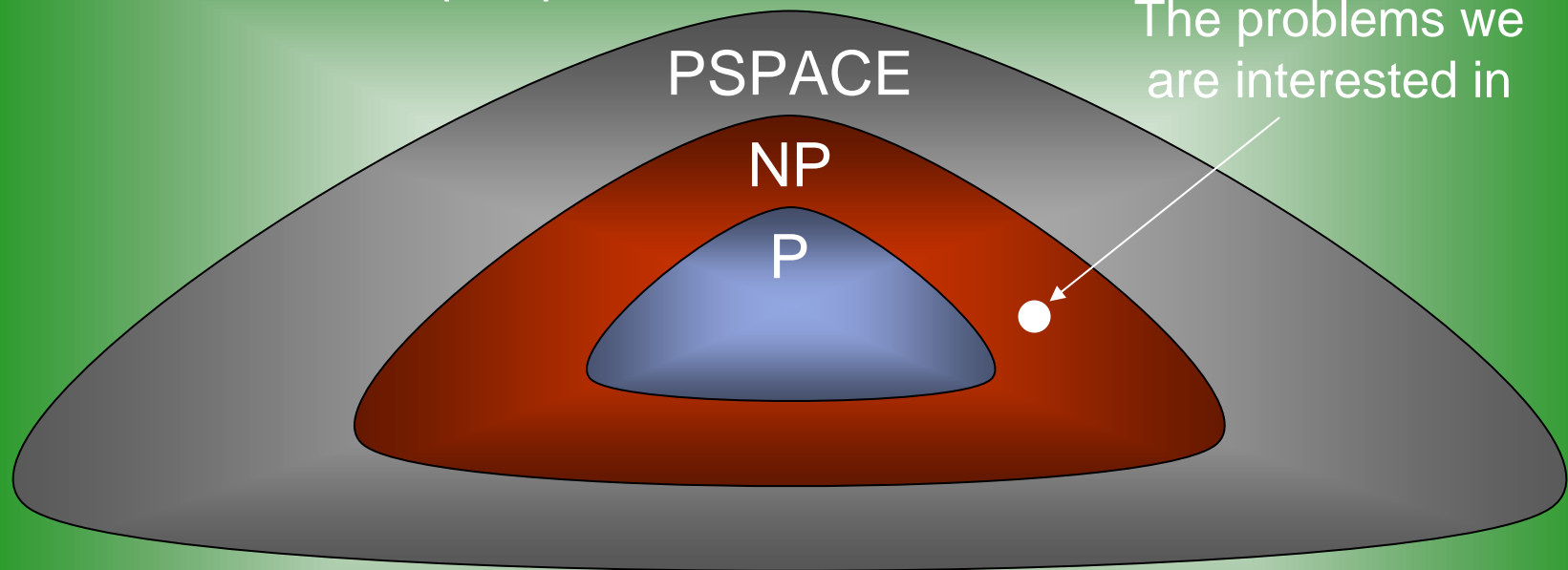


The complexity class NP

$P \subseteq NP \rightarrow$ Easy to show

$P \subset NP \rightarrow$ Famous open problem $P \neq NP$ 😊

The problems we
are interested in



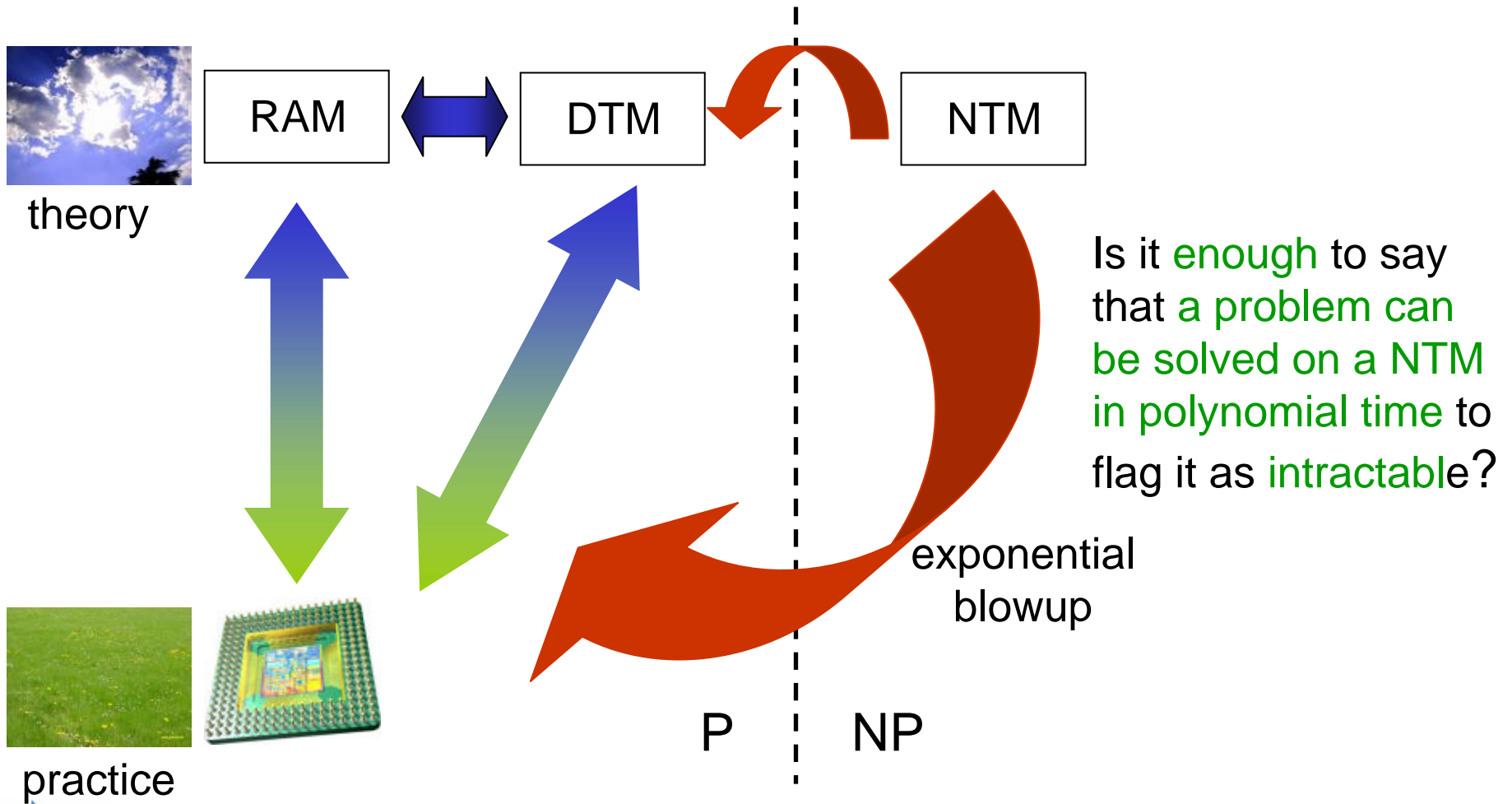
$NP \subseteq PSPACE \rightarrow$ Less obvious, but true!

$NP \subset PSPACE \rightarrow$ Open problem

Computable
Problems



Intractability: from theory to practice



The Theory of NP-completeness (I)

- P is in NP since the simplest NTM is the DTM ! 😊
- Therefore saying that a **problem is in NP** is **not sufficient to flag it as intractable**: it could still be in P !
- Being in $P-NP$ would be a **candidate characterization** for (the simplest class of) **intractable problems**
- For what we know, **$P-NP$ might be empty!**
- We need a **class of problems** that could be a **candidate for $P-NP$** , if we knew $P \neq NP$ for sure
- **NP -complete problems** are such a class



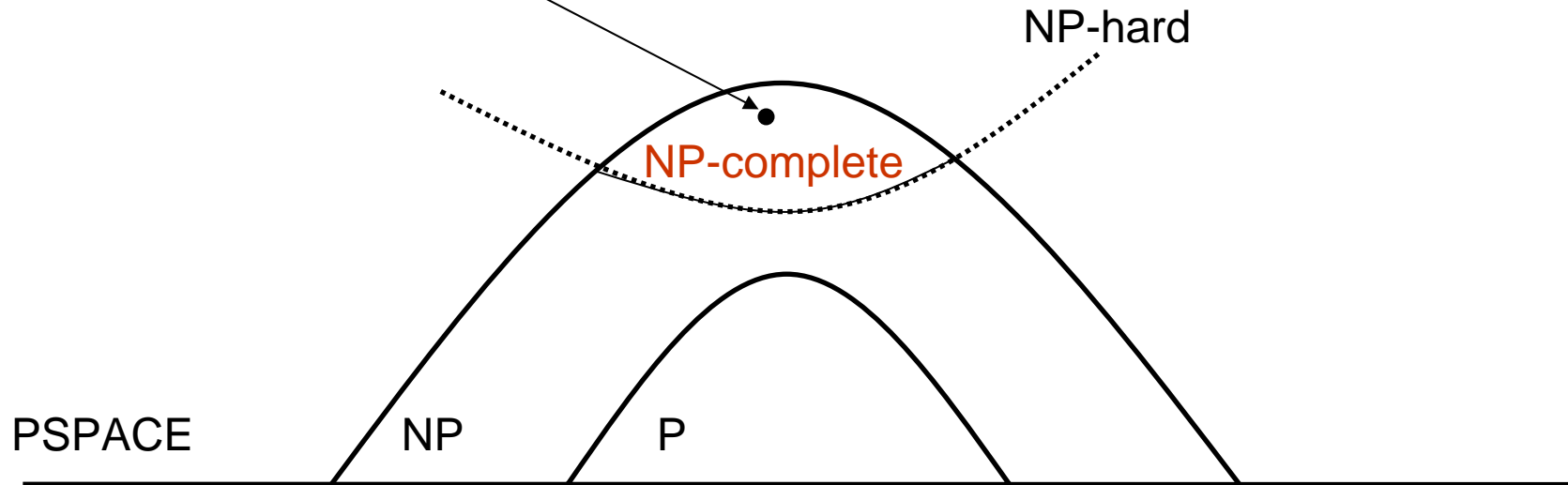
The Theory of NP-completeness (II)

- A problem p is NP-complete iff:
 - $p \in \text{NP}$ (membership – upper bound)
 - the translation from any other problem in the class NP to the problem p is computable in polynomial time by a DTM (hardness – lower bound)
- NP-complete problems form an equivalence class modulo polytime translations
- If we could prove that at least one NP-complete problem can (cannot) be solved in polynomial time on a DTM then we would have proved that $P = \text{NP}$ ($P \neq \text{NP}$)



NP-completeness at a glance

Are these **interesting problems** or just nice exercises for the theorists?



The NP-hard days of our lives

NP-complete problems are a fact of everyday's life:

- **Network Design**: constrained spanning trees, network partitioning, constrained routing, constrained flows
- **Storage and Retrieval**: dynamic storage allocation, multiple copy file allocation, Boyce-Codd normal form violation, sparse matrix compression
- **Sequencing and Scheduling**: constrained sequencing, multiprocessor scheduling, job-shop scheduling, staff scheduling, production planning
- **Program optimization**: register sufficiency, feasible register assignment, register sufficiency for loops

M.R. Garey and D.S. Johnson Computers and Intractability, 1979

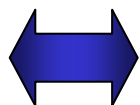


Aren't we smart enough?

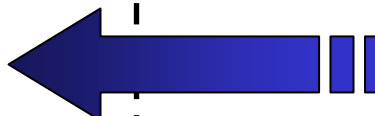


theory

RAM



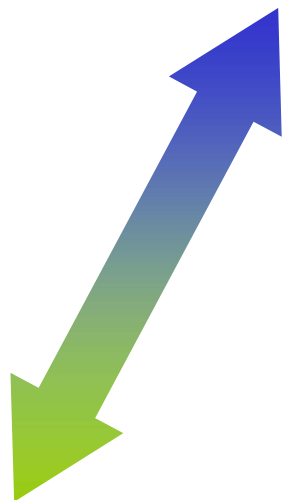
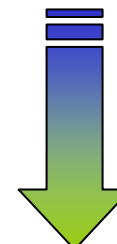
DTM



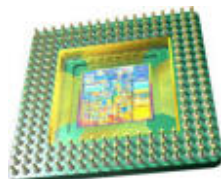
NTM

new algorithms?

new hardware?



practice



The **P vs. NP** issue has to do with our ability to devise **new hardware** or **efficient algorithms** for the hardware that we have

P | NP



Looking for new hardware (I)

Quantum computing (source <http://www.qubit.org/>)

- Arisen from an idea of **Richard Feynman** that **quantum mechanics** might lead to a **qualitatively new kind of computer** that is **far more powerful** than any classical device
- A **new mode of information processing** that harnesses physical phenomena unique to quantum mechanics (esp. interference)
- **Bits are 0,1 or a superposition thereof** (with known probability)
- These are the most important applications currently known:
 - **Cryptography**, perfectly secure communication.
 - **Searching**, especially algorithmic searching (Grover's algorithm).
 - **Factorising large numbers** very rapidly (Shor's algorithm).
- **Obstacles**: error correction, decoherence, and hardware
- In **2001** two **IBM scientists** built a **7 qubit quantum computer** that ran **Shor's algorithm** to factorise the number **15...**



Looking for new hardware (II)

Molecular computing (source: web surfing)

- Use DNA molecules plus enzymes as massively parallel computing devices
- In November of 1994, Leonard Adleman found a way to solve the "Hamiltonian path problem" a known NP-complete problem
- Thus far, Adleman has only tested his DNA model with instances comprised of 6 vertices...
- University of Wisconsin created a molecular computer "chip", a layer of gold over a glass plate where:
 - strands of DNA represent solutions to a computational problem with 16 possible answers, and then
 - enzymes were applied to the gold slide to strip out all the DNA with the incorrect answers and, and thus, solving the calculation
- A DNA computer the size of a penny, if practical, could hold up to 10 terabytes of data!



Looking for new algorithms (I)

DREAM

Using current hardware technology make the P vs. NP question irrelevant in practice, at least for specific subclasses of NP-complete problems

Aren't we talking about an equivalence class?
If you get one, then you should get them all!

Yes, that's what the definition says, but

- proving that a problem is NP-complete requires a translation to and from some known NP-complete problem, and
- not all the problems are easily translatable into each other!

This is a known limit of the classic complexity theory!

In practice, the ability to solve a given NP-complete problem efficiently (a problem, not a single instance!) is relevant only up the problems that are know to be easily translatable to the one conquered



Looking for new algorithms (II)

A plausible research agenda

1. Find a **significant NP-complete problem**
2. Throw the **kitchen sink** at it and **analyze the results**
3. Build a **new system** that exploits a combination of:
 - **fast and cheap computers** (clusters, maybe grids)
 - **sophisticated algorithms**: machine learning, program synthesis, data mining, ... the works!
 - **high-performance software**
 - **evolving domain knowledge**
4. **Deploy the system on the web**, let the users supply problems and watch the system **evolve and adapt** to solve hard instances

IBM defines (3) as “Deep Computing”



Main points of Part I

➤ Computation

- Choice of the computer defines computability and complexity
- The TM is our choice, maybe the only possible choice (CTT)

➤ (In)Tractability

- tractable means computable by a DTM in poly time and space
- intractable means to be outside the PTIME (P) bounds

➤ P vs. NP

- NPTIME (NP) is the class of problems computable on NTMs in poly time -> they require exponential time on DTMs
- NP-complete is the candidate class for NP-P, if $P \neq NP$
- Solving NP-complete problems efficiently is an interesting, challenging and still open research task



Part II: A case study on intractability



Boolean logic: syntax

Given a set of variables $X = \{x_1, \dots, x_n\}$, a sentence is either :

- a variable $x \in X$
- $\neg\alpha$ where α is a sentence
- $(\alpha \circ \beta)$ where α, β are sentences and $\circ \in \{\wedge, \vee, \equiv, \oplus\}$

Examples

$$(x_1 \wedge \neg x_1)$$

Absurdity

$$(\neg(x_1 \vee x_2) \equiv (\neg x_1 \wedge \neg x_2))$$

De Morgan's law

$$((x_1 \oplus x_3) \oplus (x_2 \wedge x_4))$$

MSB sum in a two-bit adder

$$(x_1x_2 + x_3x_4)$$

We use also the term **formula** to denote sentences



Boolean logic: semantics

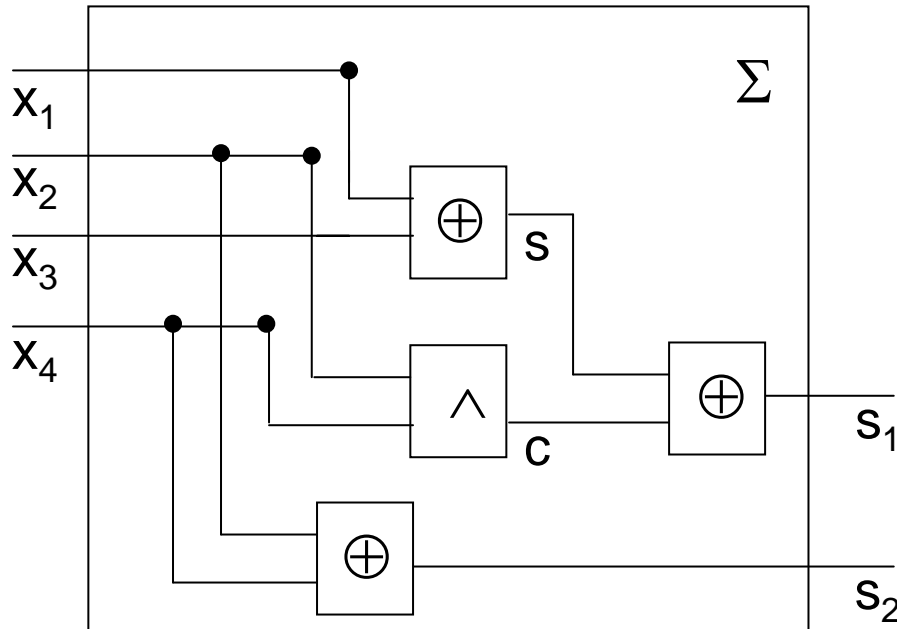
- The variables $x \in X$ are assigned a value in $\{0, 1\}$ by a (total) interpretation function $I : X \rightarrow \{0, 1\}$
- I is extended to a sentence α (denoted α^I) as follows :

$(\neg\alpha)^I$	α^I	$(\alpha \oplus \beta)^I$	$(\alpha \equiv \beta)^I$	$(\alpha \wedge \beta)^I$	$(\alpha \vee \beta)^I$	α^I	β^I
1	0	0	1	0	0	0	0
0	1	1	0	0	1	0	1
		1	0	0	1	1	0
		0	1	1	1	1	1
NOT		XOR	IFF	AND	OR		

- If α is a variable then $\alpha^I = I(\alpha)$



Formulas and circuits



$$s_1 = \overbrace{((x_1 \oplus x_3))}^s \oplus \overbrace{(x_2 \wedge x_4)}^c$$

$$s_2 = (x_2 \oplus x_4)$$

input signal		variable
gate		operator
inverter	↔	not
intermediate signal		subformula



Satisfiability & Tautology

Problem

SAT

Given a sentence α over the set of variables X does an interpretation $I : X \rightarrow \{0, 1\}$ such that $\alpha^I = 1$ exist?

If so, α is satisfiable, and unsatisfiable otherwise.

Problem

TAUT

Given a sentence α over the set of variables X are all the interpretations $I : X \rightarrow \{0, 1\}$ such that $\alpha^I = 1$?

If so, α is a tautology ("a sentence that justifies itself")

Relationship between SAT and TAUT

If $\neg\alpha$ is satisfiable, then α is not a tautology

If α is unsatisfiable, then $\neg\alpha$ is a tautology

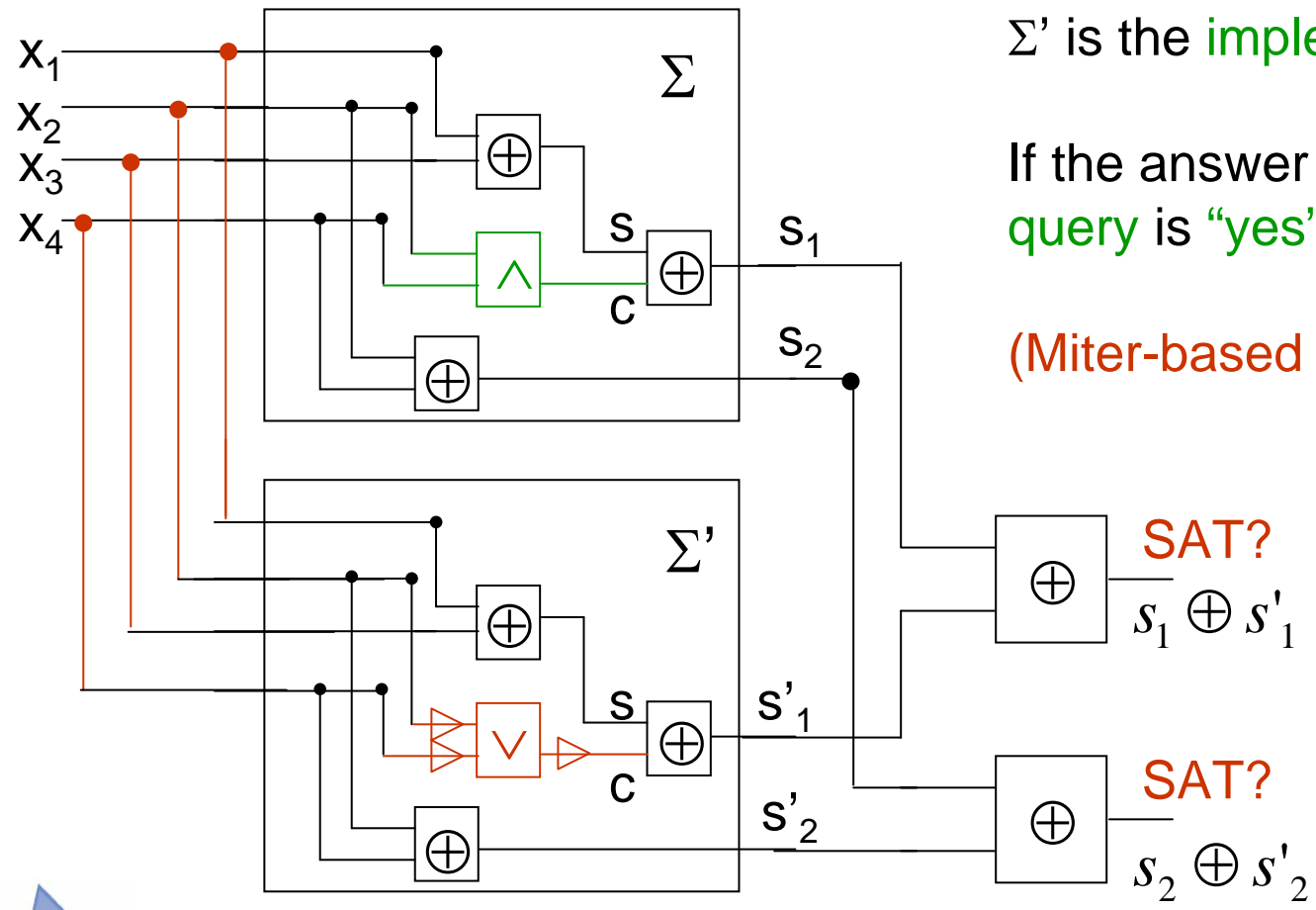


Importance of SAT and TAUT

- **Boolean logic** represents the **simplest logical framework**, yet it has **practical relevance** in many fields where Automated Reasoning is an enabling technology:
 - **Formal Verification**
 - **Knowledge representation and reasoning**
 - **Planning and scheduling**
- **Using Boolean logic** in applicative problems involves:
 - **A formal model** of the application domain
 - An **encoding** of (part of) the model into boolean logic
 - **Queries**, in the form of **SAT or TAUT problems**
 - The results of the queries concur to provide the solution(s) to the original problem(s)



Example: Equivalence Verification



Σ is the **abstract model** and Σ' is the **implementation**

If the answer to **at least one SAT query** is "yes", then Σ' has a bug!

(Miter-based FEV of circuits)

SAT? $s_1 \oplus s'_1$

Same as TAUT of:

$(s_1 \equiv s'_1) \wedge$

$(s_2 \equiv s'_2)$

SAT? $s_2 \oplus s'_2$



The complexity of SAT

- SAT is the **first problem shown to be NP-complete** (by S. Cook in 1971, using a reduction from NTM to SAT)
- Indeed, **n variables** imply **2^n possible interpretations** from which **at least one** satisfying the formula **must be guessed**: **easy to check – hard to guess** situation!
- There are **cases** in which a **polynomial time algorithm** exists: no guessing is necessary.
- A **SAT solver** (or simply, solver) is a **program that solves SAT** by finding a satisfying interpretation **automatically**



Mechanizing SAT (I)

- Formulas must be in Conjunctive Normal Form (CNF)
- A formula in CNF is:
 - a n -ary conjunction of clauses
 - each clause is a m -ary disjunction of literals
 - each literal is either a variable or the negation thereof
- Any formula α can be translated using up to a linear time in the size of α into a CNF α' such that α' is satisfiable iff α is satisfiable
- We call the above property “equisatisfiability”
- α' is not equivalent to α because the translation adds new variables to represent subformulas of α



Mechanizing SAT (II)

$$s_1 = ((x_1 \oplus x_3) \oplus (x_2 \wedge x_4))$$

Boolean sentence



$(s \vee c) \wedge$
$(\neg s \vee \neg c) \wedge$
$(\neg s \vee x_1 \vee x_3) \wedge$
$(\neg s \vee \neg x_1 \vee \neg x_3) \wedge$
$(s \vee \neg x_1 \vee x_3) \wedge$
$(s \vee x_1 \vee \neg x_3) \wedge$
$(c \vee \neg x_2 \vee \neg x_4) \wedge$
$(\neg c \vee x_2) \wedge$
$(\neg c \vee x_4)$

CNF translation



$$\begin{aligned} & \{ \{s, c\}, \\ & \{ \bar{s}, \bar{c} \}, \\ & \{ \bar{s}, x_1, x_3 \}, \\ & \{ \bar{s}, \bar{x}_1, \bar{x}_3 \}, \\ & \{ s, \bar{x}_1, x_3 \}, \\ & \{ s, x_1, \bar{x}_3 \}, \\ & \{ c, \bar{x}_2, \bar{x}_4 \}, \\ & \{ \bar{c}, x_2 \}, \\ & \{ \bar{c}, x_4 \} \end{aligned}$$

Set of sets of literals



Mechanizing SAT (III)

$\{\{s, c\},$
 $\{\bar{s}, \bar{c}\},$
 $\{\bar{s}, x_1, x_3\},$
 $\{\bar{s}, \bar{x}_1, \bar{x}_3\},$
 $\{s, \bar{x}_1, x_3\},$
 $\{s, x_1, \bar{x}_3\},$
 $\{c, \bar{x}_2, \bar{x}_4\},$
 $\{\bar{c}, x_2\},$
 $\{\bar{c}, x_4\}\}$

- Each **clause** is a **constraint**, similar to a constraint in **linear algebraic systems**
 - If an interpretation **violates a single constraint** then **it does not satisfy the formula** (e.g. any interpretation where $s=0$ and $c=0$)
 - If an interpretation **satisfies all the constraints**, then **it satisfies the formula** (e.g., all the interpretations where the MSB of the sum is 1)
- We start by looking at **solvers** that are
 - **sound**: if the solver produces an interpretation, then the formula is satisfiable under it
 - **complete**: if the solver does not produce an interpretation, then the formula is unsatisfiable



Solver 0: Brute Force

Input : A CNF formula α over a set of variables X ($|X| = n$)

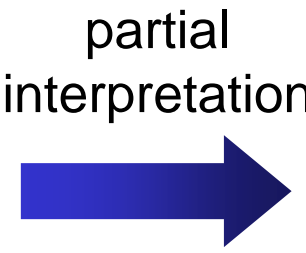
- 1 Let $X = I$ (shorthand for $I(x_i) = v_i$ for all $x_i \in X$ and $v_i \in \{0,1\}$)
- 2 Check if all the clauses in α are satisfied by I
- 3 If so, $\alpha^I = 1$ and α is satisfiable; STOP.
- 4 Otherwise, get a new I by flipping one of the v_i 's; go to step 2
- 5 If there is no new I to try, then α is unsatisfiable; STOP.

Solver 0 is sound and complete : it takes $O(2^n)$ time



Interpretation vs. partial interpretation

$\{s, c\},$		$\{s, c\},$	$\{s, c\},$	$\{s, c\},$
$\{\bar{s}, \bar{c}\},$		$\{\bar{s}, \bar{c}\},$	$\{\bar{s}, \bar{c}\},$	$\{\bar{s}, \bar{c}\},$
$\{\bar{s}, x_1, x_3\},$		$\{\bar{s}, x_1, x_3\},$	$\{\bar{s}, x_1, x_3\},$	$\{\bar{s}, x_1, x_3\},$
$\{\bar{s}, \bar{x}_1, \bar{x}_3\},$ interpretation		$\{\bar{s}, \bar{x}_1, \bar{x}_3\},$	$\{\bar{s}, \bar{x}_1, \bar{x}_3\},$	$\{\bar{s}, \bar{x}_1, \bar{x}_3\},$
$\{s, \bar{x}_1, x_3\},$		$\{s, \bar{x}_1, x_3\},$	$\{s, \bar{x}_1, x_3\},$	$\{s, \bar{x}_1, x_3\},$
$\{s, x_1, \bar{x}_3\},$		$\{s, x_1, \bar{x}_3\},$	$\{s, x_1, \bar{x}_3\},$	$\{s, x_1, \bar{x}_3\},$
$\{c, \bar{x}_2, \bar{x}_4\},$ $x_i=0 \ i \in \{1, \dots, 4\},$ $s=0$ and $c=0$		$\{c, \bar{x}_2, \bar{x}_4\},$	$\{c, \bar{x}_2, \bar{x}_4\},$	$\{c, \bar{x}_2, \bar{x}_4\},$
$\{\bar{c}, x_2\},$		$\{\bar{c}, x_2\},$	$\{\bar{c}, x_2\},$	$\{\bar{c}, x_2\},$
$\{\bar{c}, x_4\}$		$\{\bar{c}, x_4\}$	$\{\bar{c}, x_4\}$	$\{\bar{c}, x_4\}$

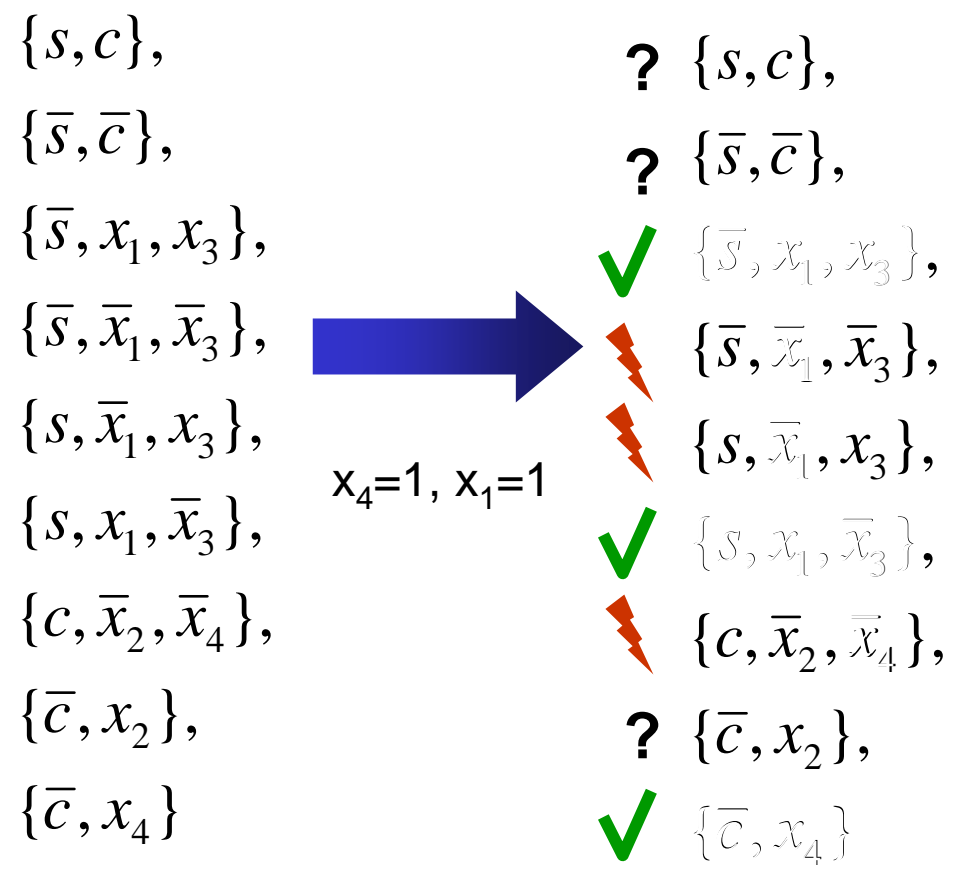


$x_i=0 \ i \in \{1, \dots, 4\},$
 $s=0$ and $c=0$

$x_4=1, x_1=1$



Unleashing partial interpretations



Any partial interpretation:

- satisfies (✓) some clauses
- simplifies (⚡) some literals

The satisfied clauses and the simplified literals can be disregarded when trying to extend the partial interpretation to a total (satisfying) one.

When all the literals in a clause are simplified, the clause is violated, and all the extensions of the partial interpretation are bound to be unsatisfying ones!



Solver 1: Heuristic Search

Input : A CNF formula α over a set of variables X ($|X| = n$)

Let I be a partial interpretation (initially $I(x) = ?$ for all $x \in X$)

- ① If I satisfies all the clauses in α then STOP
- ② If I violates some clause in α then BACKTRACK
- ③ Choose a variable \tilde{x} in α such that $I(\tilde{x}) = ?$ and a value v

Let I' be the function
$$I'(x) = \begin{cases} I(x) & \text{if } x \neq \tilde{x} \\ v & \text{if } x = \tilde{x} \end{cases}$$

Finally, let $I(x) = I'(x)$ and go back to step 1

Solver 1 is sound and complete : it takes $O(2^n)$ time



Search & Backtracking

- Heuristic search starts with an “empty” interpretation and tries to extend it to a satisfying one
- Unless the heuristic (step 3) always guesses the right 0,1 value the search will hit a dead end, i.e., a partial interpretation that violates some clauses and should not be extended any further
- **BACKTRACK** does the following:
 - Consider the last heuristic choice, flip the value chosen, recompute I, and let the search start again (from step 1)
 - If there are no more heuristic choices to be reverted, then the formula is unsatisfiable



Solver 1: Example

$\{\{\bar{x}_1, \bar{y}, x_2\}, \{x_1, \bar{y}, \bar{x}_3\}, \{\bar{y}, \bar{x}_2\}, \{y, x_2, \bar{x}_3\}, \{x_2, x_3\}\}$

$x_1 = 0$ | $x_1 = 1$

$\{\{\bar{y}, \bar{x}_3\}, \{\bar{y}, \bar{x}_2\},$

$\{\{\bar{y}, x_2\}, \{\bar{y}, \bar{x}_2\},$

$\{y, x_2, \bar{x}_3\}, \{x_2, x_3\}\}$

$\{y, x_2, \bar{x}_3\}, \{x_2, x_3\}\}$

$y = 1$ | $y = 0$

$\{\{\bar{x}_3\}, \{\bar{x}_2\},$

$\{\{x_2, \bar{x}_3\},$

$\{x_2, x_3\}\}$

$\{x_2, x_3\}\}$

$x_2 = 0$
 $x_3 = 0$

$x_2 = 1$

$\{\{\}\}$

$\{\{\}\}$

X conflict
(no-good)

✓ solution
(good)

$y = 0$ | $y = 1$

$\{\{x_2, \bar{x}_3\},$

$\{\{x_2\}, \{\bar{x}_2\},$

$\{x_2, x_3\}\}$

$\{x_2, x_3\}\}$

$x_2 = 1$

$x_2 = 0$

$\{\{\}\}$

$\{\{\}\}$

✓

X



Heuristics and NP-completeness

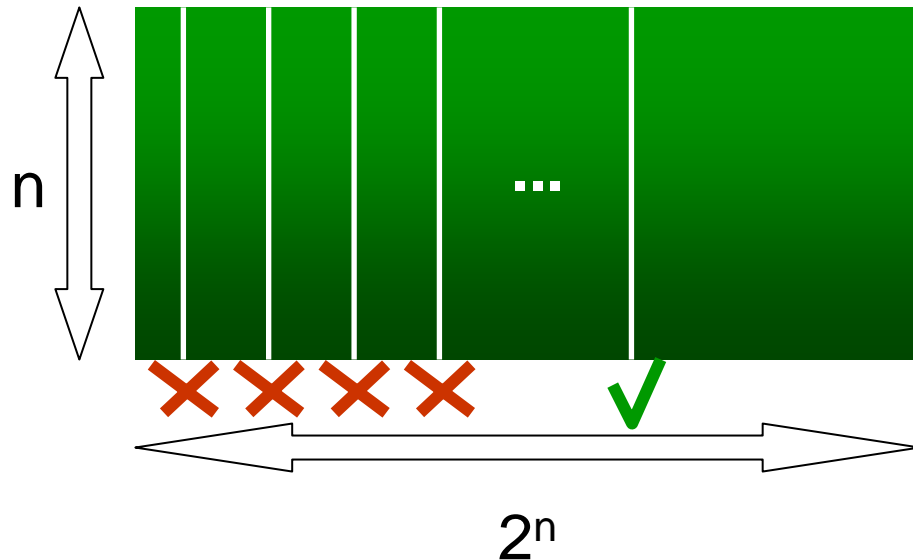
- What kind of **heuristics** can we think of?
 - **Oracle**: guesses the right assignment in $O(n)$ (an NTM!)
 - **Perfect heuristic**: finds the solution in $O(\text{poly}(n))$
 - **Real heuristic**: **ordering** of the variables, either
 - **static**, fixed before the search begins, or
 - **dynamic**, (re)computed during the search
- How **good** can a heuristic be?
 - Computing an **optimal real heuristic** is **NP-hard**!
 - **Other realistic strategies** are bound to be suboptimal
 - Think of a **game** where **for any fixed strategy** that you may deploy, your **opponent is going to outsmart you!**

Complexity of solver 1 is the same as solver 0!

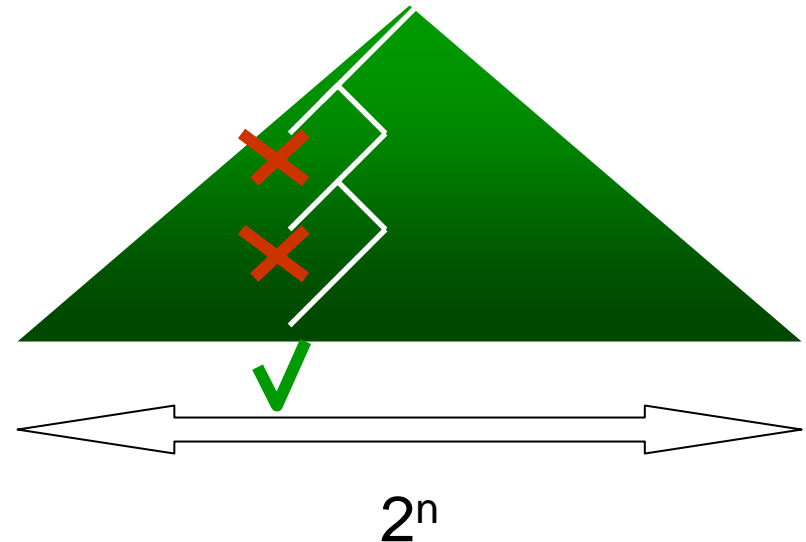


Brutality vs. Heuristics

Solver 0



Solver 1



“Considerate la vostra semenza:
fatti non foste a viver come bruti
ma per seguir virtute e canoscenza.”

Dante Alighieri – Divina Commedia - Inferno



Solver 2: Search + Deduction(FW)

- Use Solver 1 as a basis, but apply polytime procedures to find forced assignments, e.g.:
 - **Unit propagation**: if all the literals in a clause but one are assigned to 0, then the last one must be assigned to 1
 - **Monotone literals**: if a literal appears throughout the formula with only positive (negative) polarity, then it can be assigned to 1 (0)
- These kind of procedures, a.k.a. lookahead techniques, try to prune bad assignment choices up front
- They are applied as preprocessing steps, or, more frequently, before each branch in the search

Solver 2 is sound and complete : it takes $O(2^n)$ time



Solver 2: Example

$\{\{\bar{x}_1, \bar{y}, x_2\}, \{x_1, \bar{y}, \bar{x}_3\}, \{\bar{y}, \bar{x}_2\}, \{y, x_2, \bar{x}_3\}, \{x_2, x_3\}\}$

$x_1 = 0$ | $x_1 = 1$

$\{\{\bar{y}, \bar{x}_3\}, \{\bar{y}, \bar{x}_2\},$

$\{\{\bar{y}, x_2\}, \{\bar{y}, \bar{x}_2\},$

$\{y, x_2, \bar{x}_3\}, \{x_2, x_3\}\}$

$\{y, x_2, \bar{x}_3\}, \{x_2, x_3\}\}$

$y = 1$ | $y = 0$

$\{\{\bar{x}_3\}, \{\bar{x}_2\},$

$\{\{x_2, \bar{x}_3\},$

$\{x_2, x_3\}\}$

$\{x_2, x_3\}\}$

$x_2 = 0$
 $x_3 = 0$

$x_2 = 1$

$y = 0$ | $y = 1$

$\{\{x_2, \bar{x}_3\},$

$\{\{x_2\}, \{\bar{x}_2\},$

$\{x_2, x_3\}\}$

$\{x_2, x_3\}\}$

$x_2 = 1$

$x_2 = 0$

$\{\{\}\}$

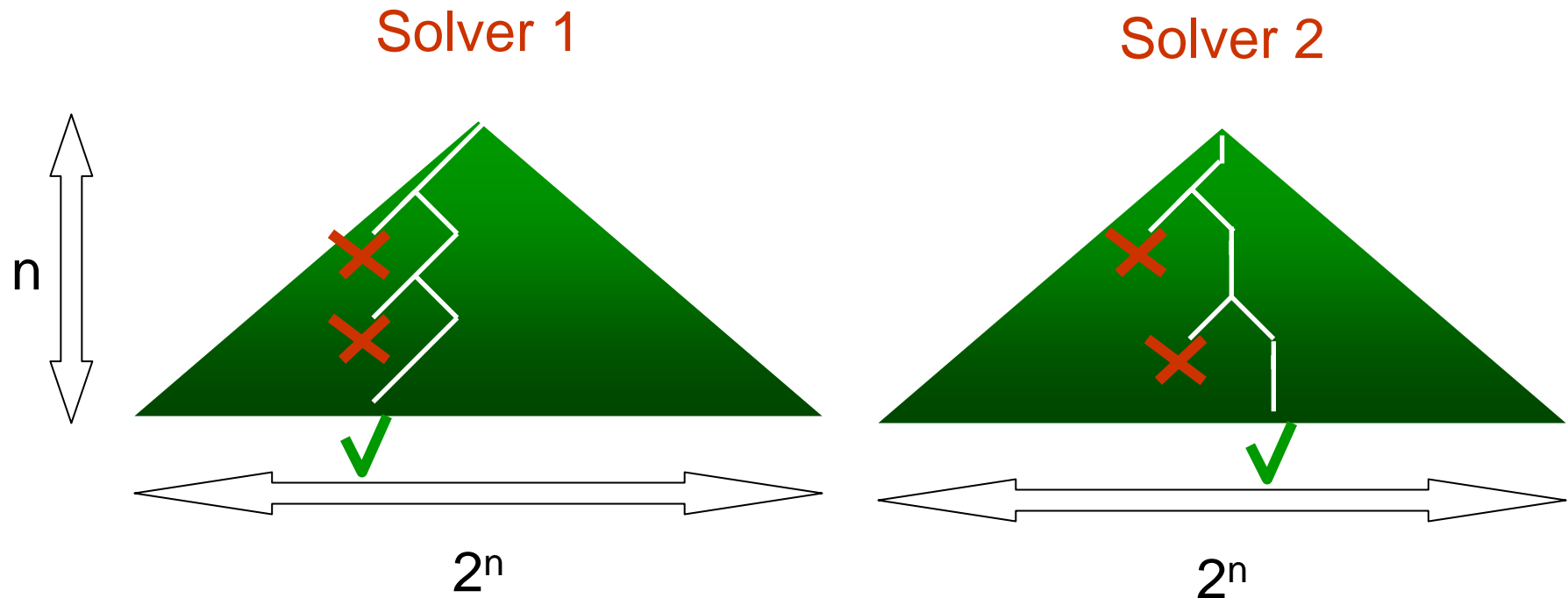
$\{\{\}\}$

$\{\{\}\}$

$\{\{\}\}$



Effect of deduction techniques



Using deduction techniques **reduces the number of choices** that have to be backtracked over in case of failure and, **in some cases**, can **determine the (un)satisfiability of the formula** without performing any search.



Solver 3: Search + Deduction(FW and BW)

- In Solver 2 no reasoning is done on the conflicts
- Once the solver hits a conflict, we can
 - apply polytime procedures to deduce what choices are effectively involved in making the conflict arise
 - use the deduced facts to avoid backtracking over irrelevant choices, thus pruning (backwardly) the search space
- Several strategies are possible (a.k.a. lookback)
 - skip the choices that did not cause the conflict (backjumping)
 - avoid repeating the same mistakes at future points (learning)
 - rearrange the search tree as to optimize its size w.r.t the new findings (dynamic backtracking)

Solver 3 is sound and complete : it takes $O(2^n)$ time



State of the art in SAT

- Solver 3 is the basis for current SotA complete solvers
- Important facts:
 - No implemented solver is claiming polynomial worst case time complexity in the general case
 - Solver 3 has a provable exponential lower bound, i.e, there exist a class of formulas s.t. Solver 3 is bound to run in exponential time when trying to find a satisfying interpretation
- Are these important limitations? Yes and no!
 - Yes, no implementation of solver 3 is ever going to have a reasonable running time in the worst case
 - No, implementations of solver 3 can deal with instances having 1000s of variables and 100.000s of clauses!



Can we do any better?

- **More powerful deduction systems**
 - Resolution
 - Extended resolution
- **Incomplete/approximate methods**
 - Incomplete solvers: use some stochastic local search method
 - Approximate solvers: relax to tractable domains
- **Structure-based engines**
 - Polynomial SAT problems: 2SAT, XOR-SAT and Horn-SAT
 - Graph-theoretical analysis
- **Other methods**
 - Binary decision diagrams
 - SAT in hardware
 - ...



Resolution (I)

Let α be a formula in CNF, and let $\{x, P\}$ and $\{\bar{x}, Q\}$ be two clauses such that P and Q do not contain any l such that l is in P and \bar{l} is in Q

Resolution rule : all the interpretations that satisfy α satisfy also the clause $\{P, Q\}$, i.e., $\{P, Q\}$ can be added to α without changing its satisfiability status.

1. $\{x, y, z\}$

2. $\{\bar{x}, y, w\}$



3. $\{y, z, w\}$

Example:

In any interpretation I , the variable x is either true or false:

- if x is true, either y or w must be true for I to satisfy α ;
- if x is false, either y or z must be true for I to satisfy α ;

Therefore, any interpretation that satisfies α must also satisfy either y, z , or w .



Resolution (II)

A resolution **proof** is a **sequence of applications** of the **resolution rule** to the clauses in the “**database**” (original formula + inferred clauses)

Proof by saturation

Given a CNF α , **apply all the possible resolutions**, each time adding the result to α , until, e.g., **no further new clauses can be generated**; conclude that α is **satisfiable**.

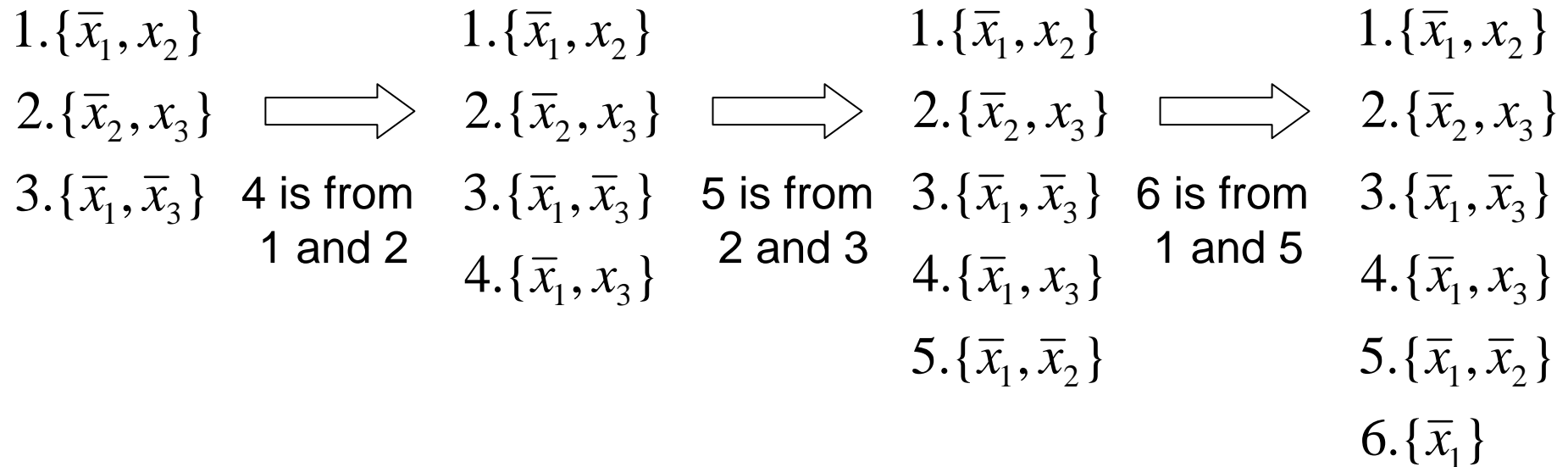
Proof by refutation

Given a CNF α , **apply all the possible resolutions**, each time adding the result to α , until **an empty clause is generated**; conclude that α is **unsatisfiable**.

Resolution can generate up to an exponential number of clauses!



Resolution: proof by saturation

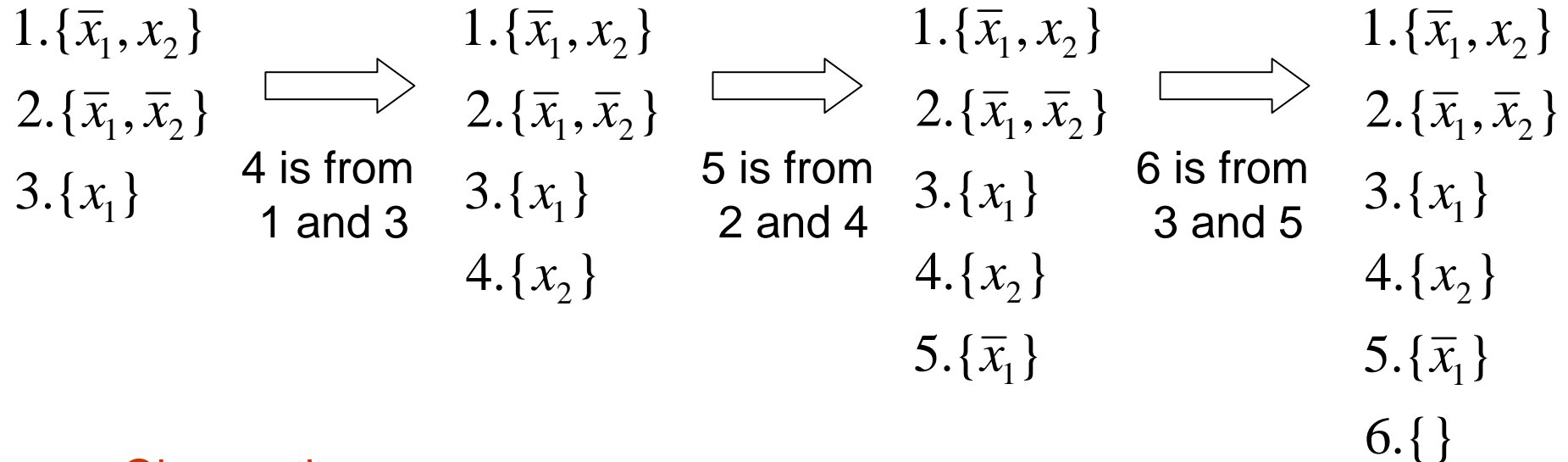


Observations:

- In the end, no new clauses can be generated by applying the resolution rule: resolution has saturated, i.e., the formula is satisfiable
- Among the newly generated clauses, 6 says that x_1 must be false for the whole formula to hold true!



Resolution: proof by refutation



Observations:

- 6 is an empty clause: it comes from adding x_1 and its negation to the formula at the same time
- Since both x_1 and its negation are implied by the formula then the formula is a contradiction (the negation of a tautology)



Power and resolution

p-simulation

A deduction system D p-simulates another D' provided that there is a polynomial time procedure to convert any proof in D' to a proof in D .

Power of deduction systems

Given two deduction systems D, D' :

- they are **equally powerful** if D can p-simulate D' , and D' can p-simulate D .
- **D is more powerful than D'** , if D can p-simulate D' , but not the contrary
- they are incomparable if none of the two can p-simulate the other

Resolution is more powerful than Solver 0,1,2
and it can p-simulate Solver 3 as well

Where's the catch? Why aren't we using resolution in SotA solvers?
(Btw, Resolution has an exponential lower bound as well!)



Extended resolution

- As resolution, plus the possibility to introduce new variables in the form of definitions
- A proof is a sequence where each step is:
 - an application of the resolution rule, or
 - a definition $w \equiv I_1 \wedge I_2$, where w is a new (extended) variable, I_1 and I_2 are two literals, possibly already extended
- Some interesting facts:
 - Extended resolution is more powerful than resolution
 - There is no known deduction system more powerful than extended resolution
 - There are no known exponential lower bounds for extended resolution: another view of the P vs. NP problem!



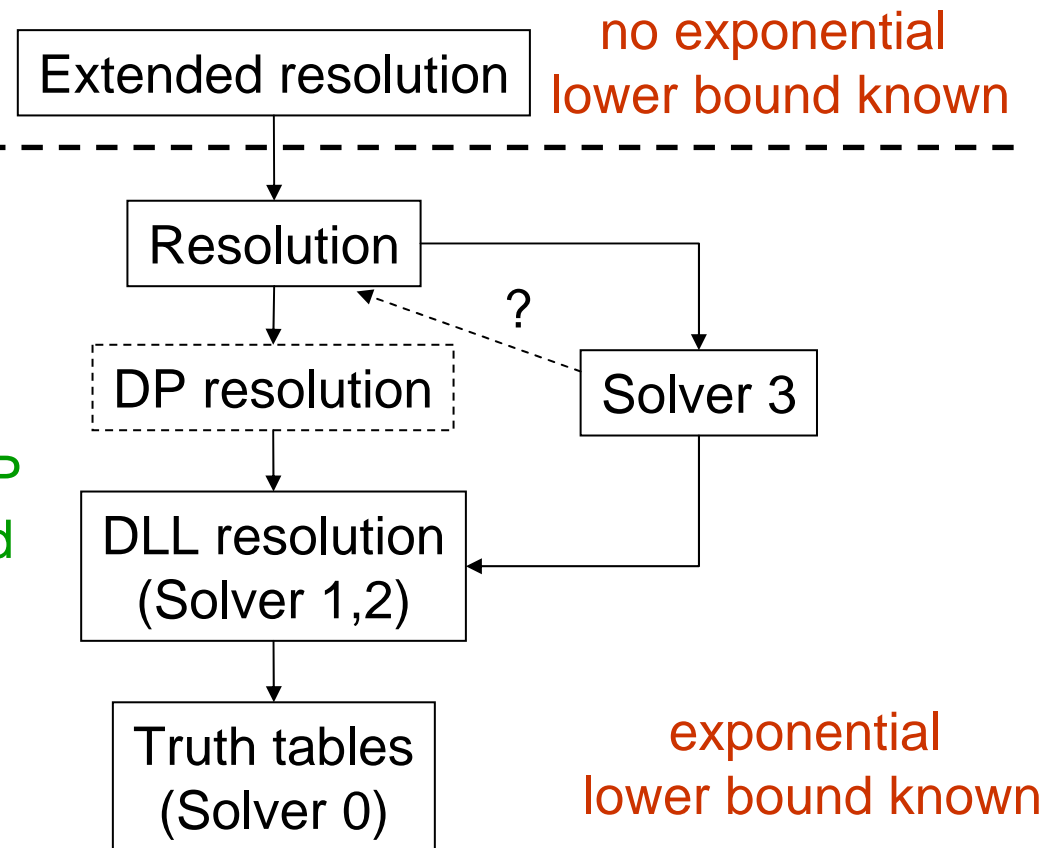
The power of (extended) resolution

P vs. NP
threshold

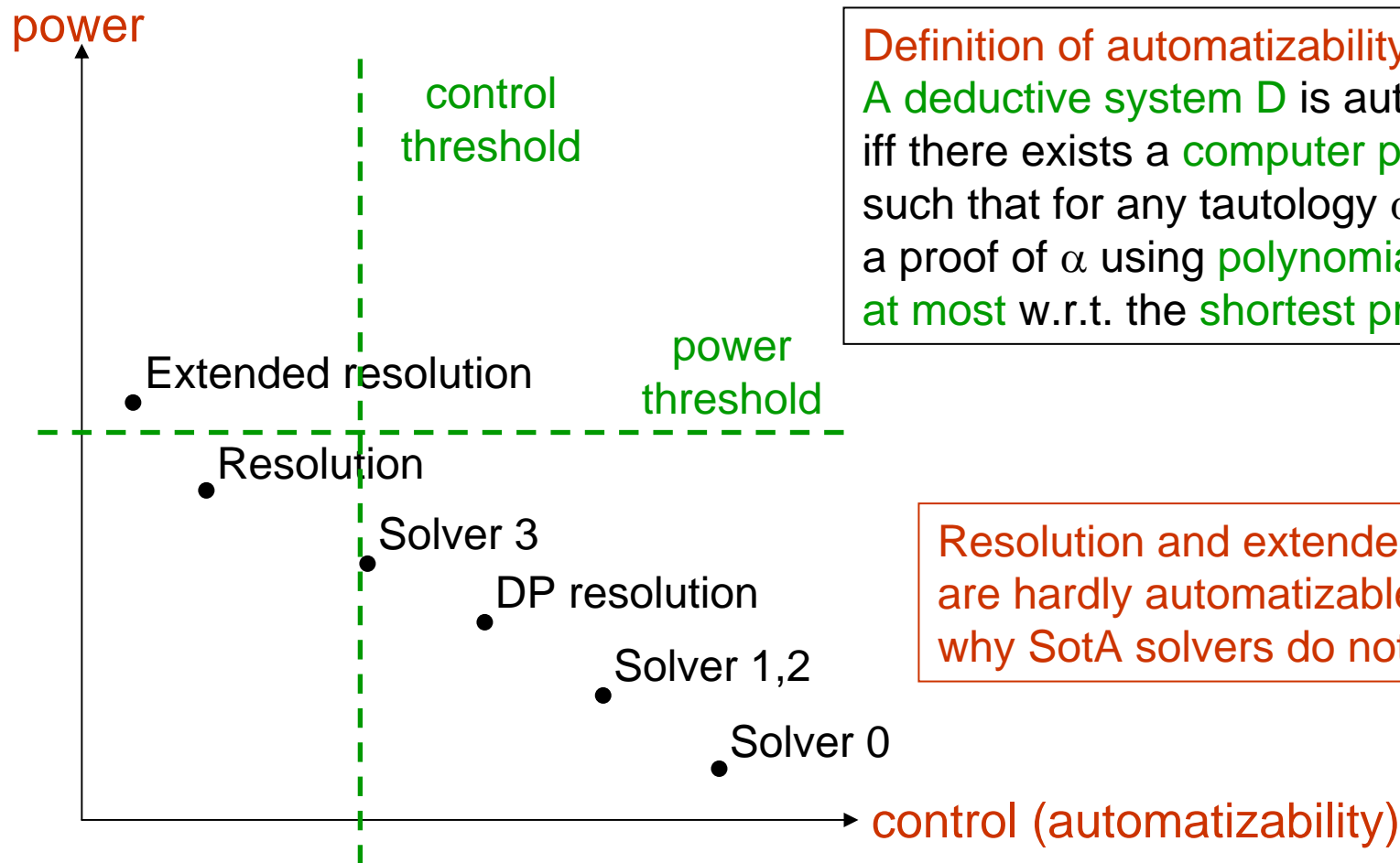
DP is an algorithm devised
by M. Davis and H. Putnam

DLL is an “improved” version of DP
by Davis, Logemann and Loveland
(DLL memory requirements are
guaranteed to be tighter than DP)

“ \longrightarrow ” means “p-simulates”



Power vs. control



Definition of automatizability

A deductive system D is automatizable iff there exists a computer program P_D such that for any tautology α , P_D produces a proof of α using polynomially more steps at most w.r.t. the shortest proof in D of α

Resolution and extended resolution are hardly automatizable: that is why SotA solvers do not use them!



Incomplete solvers

➤ Incomplete solvers

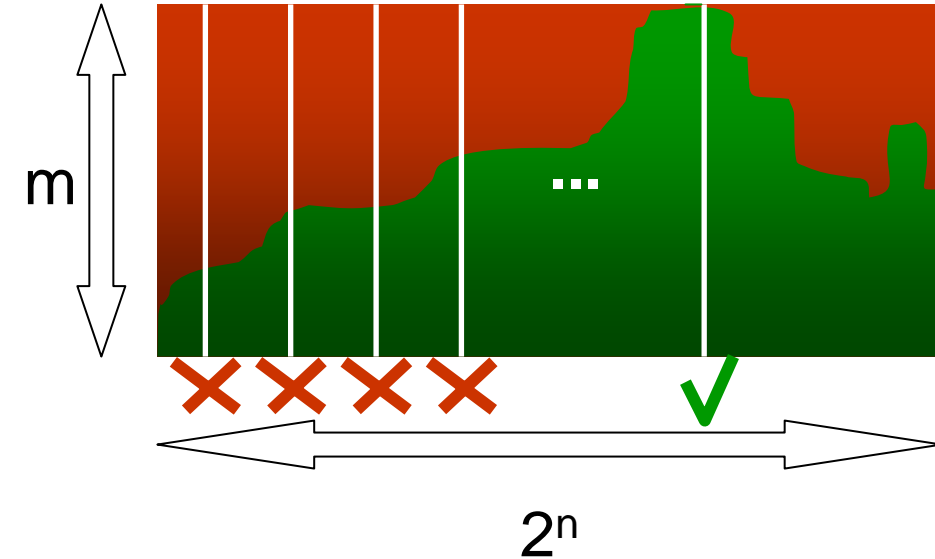
- start with a **random assignment** to all the variables
- if all the clauses are satisfied, STOP; otherwise, obtain a new assignment (e.g., by **flipping one or more values**) and try again for a **fixed number** of steps
- several variants: **greedy algorithms**, **random walk**, **tabu search**, **genetic algorithms**, **simulated annealing**, etc.

➤ **State-of-the-art incomplete solvers** can deal with **satisfiable formulas** that are **outside the capabilities** of **complete solvers**

➤ **They cannot prove unsatisfiability**, i.e., no conclusive answer can be extracted when they give up



Incomplete solvers at work



n is the number of variables
m is the number of clauses
■ satisfied clauses
■ violated clauses



Structure-based engines

- The idea is to **leverage knowledge** about the formula
 - A **polynomial time algorithm** is available when
 - all the clauses have at most 2 literals (**2SAT**)
 - all the clauses have at most one positive literal (**HornSAT**)
 - the formula is made up of XOR clauses only (**XorSAT**)
 - **More efficient algorithms** can be devised when:
 - independent and dependent variables are known
 - decoupled or loosely-coupled subformulas are known
 - detailed domain knowledge is available
 - If we know nothing, then we can **try to extract structural infos**
- What is **structure**?
 - **What kind of information is relevant?**
 - How can we **extract and leverage** such information?

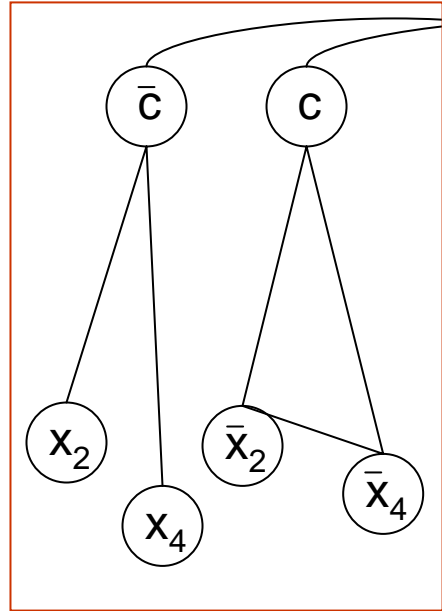


What is structure?

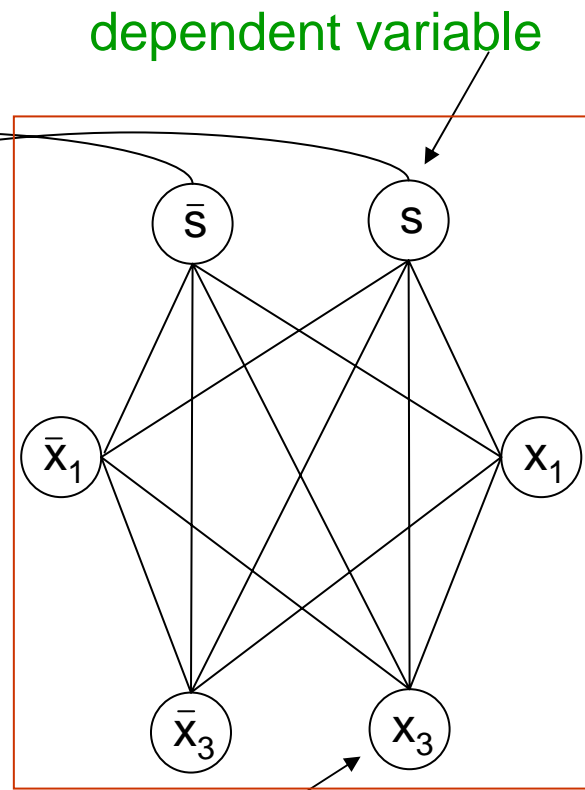
- $\{s, c\},$
- $\{\bar{s}, \bar{c}\},$
- $\{\bar{s}, x_1, x_3\},$
- $\{\bar{s}, \bar{x}_1, \bar{x}_3\},$
- $\{s, \bar{x}_1, x_3\},$
- $\{s, x_1, \bar{x}_3\},$
- $\{c, \bar{x}_2, \bar{x}_4\},$
- $\{\bar{c}, x_2\},$
- $\{\bar{c}, x_4\}$



literal \Rightarrow vertex
 k-clause \Rightarrow k-clique



LSB carry



MSB sum

dependent variable

independent variable

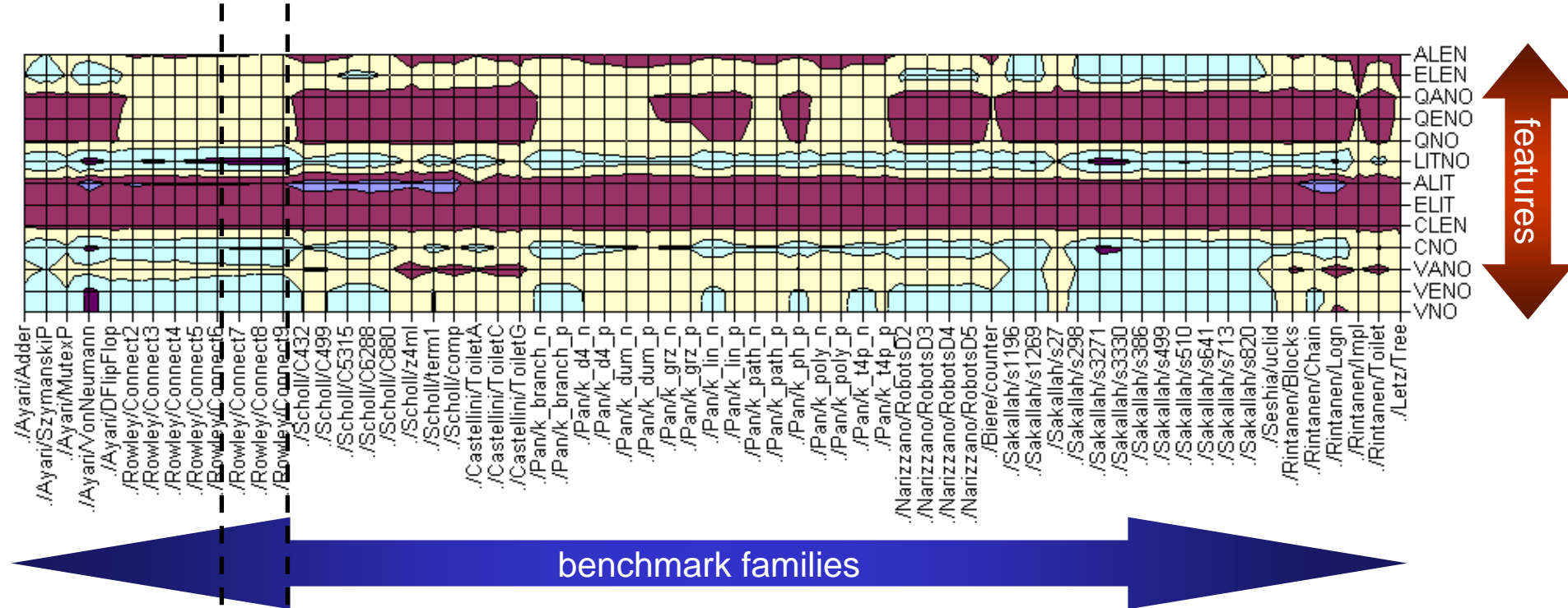


Measuring structure with features

- **Size features:** # of variables, clauses, literals, ...
- **Balance features:** # of 1-clauses, 2-clauses, ..., negated vs. positive literals, ...
- **Constraint-graphs features**
 - **Variable:** variable \approx node, edge \approx variables share clause
 - **Variable-clause:** bipartite, which variables in which clauses
 - **Conflict graph:** clause \approx node, edge \approx share a resolvent
- **Tractability features:** % of Horn cls, % of 2-clauses, ...
- **Metrics from relaxations:** e.g., run LP relaxation
- **Metrics from tentative runs:** e.g., run incomplete solvers



Features and classification



cluster

benchmark families

features

Target: classify SAT instances based on their features!

Why?

Use them as off-line selection policies (SATZILLA) or, use them for on-line automated learning & classification



Exploiting structure?

- Extracting features is **time consuming**
 - Not a serious issue for offline selections
 - Problematic for **online** usage
 - Which features are really relevant? (inductive learning)
 - Can we afford to compute the relevant features?
- Features are only **part of the picture**
 - Exploiting structures relates, e.g., to the ability of finding minimal “cuts” to partition the original problem (MINCUT)
 - MINCUT and similar structure-related queries are NP-complete!
 - Nice approximations exists, but still, they might require too much time to compute

A woodsman is trying to cut a tree using a blunt axe. Someone passes by and asks: “Why don’t you stop and sharpen your axe?”. The woodsman replies: “I can’t stop, I have to cut all these trees before sunset!”



Summing up...

- **Sound and complete** methods are either
 - easy to control, but provably not powerful enough, or
 - possibly powerful enough, but hard to control
- **Incomplete methods** are
 - (very) fast on satisfiable (or unsatisfiable) formulas
 - ... incomplete!
 - relaxations share similar disadvantages
- **Structure-based** methods
 - efficient in the case of offline selection
 - troublesome in the case of online selection
 - need for powerful structure analysis -> high complexity!
- **Other methods** not touched here: there's no free lunch!



Thesis

If there is a general purpose, practically efficient SAT solver then it is not in the form of a single algorithm (no silver bullet!)



Remembering Hercules and the Hydra...

Once the hydra emerged, Hercules seized it. [...] With his club, Hercules attacked the many heads of the hydra, but as soon as he smashed one

head, two more would burst forth in its place! [...] Hercules called on Iolaus to help him out [...]. Each time Hercules bashed one of the hydra's heads, Iolaus held a torch to the headless tendons of the neck. The flames prevented the growth of replacement heads, and finally, Hercules had the better of the beast.



Hi-NRG

High-performance Network of Reasoning engines

➤ Hi-NRG in 2004

- First (rough!) presentation of the idea
- A self-organizing network of cooperating reasoning engines with different capabilities

➤ Hi-NRG in 2005

- This presentation! 😊
- A set of hand-coded and/or automatically synthesized modules, each one devoted to fulfill a specific task and cooperating with the others. The system evolves according to adaptive patterns that are learned through experience in order to achieve complex reasoning tasks robustly and efficiently
- Emergent Reasoner



Hi-NRG idea borrows from...

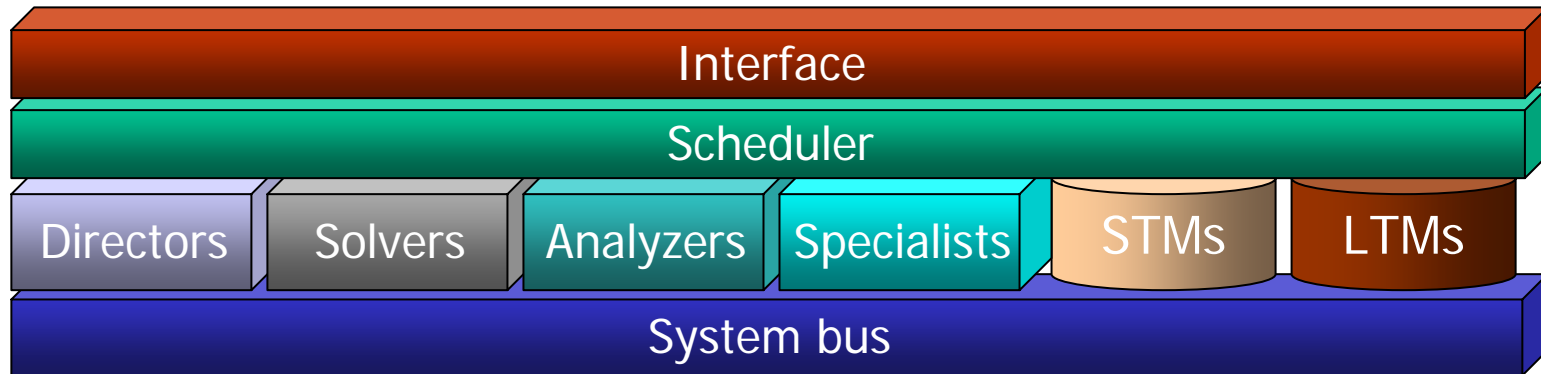
- Deep Computing (<http://www.research.ibm.com/dci>)
- Economics and Management (e.g., portfolios)
- Research in various CS fields
 - Automated Reasoning
 - Robotics
 - Machine learning
 - Cluster and GRID computing
 - Agent oriented software engineering



"The new methodology bases its decomposition of intelligence into individual behavior generating modules, whose coexistence and co-operation let more complex behaviors emerge." *R.A. Brooks*



Overview of the system



Interface: Receives the instances, provides user's and adminin's views

Scheduler: keeps track of the instances and decides those to be runned

System bus: provides distributed process control and communication facilities

Core engines: provide the (basic) reasoning functionalities

- **Directors:** learning modules, metasolvers, classifiers
- **Solvers:** interpretation/proof search engines
- **Analyzers:** structural information extractors
- **Specialists:** efficient special-purpose reasoners
- **Memory modules:** STMs (tactical) and LTMs (strategic)

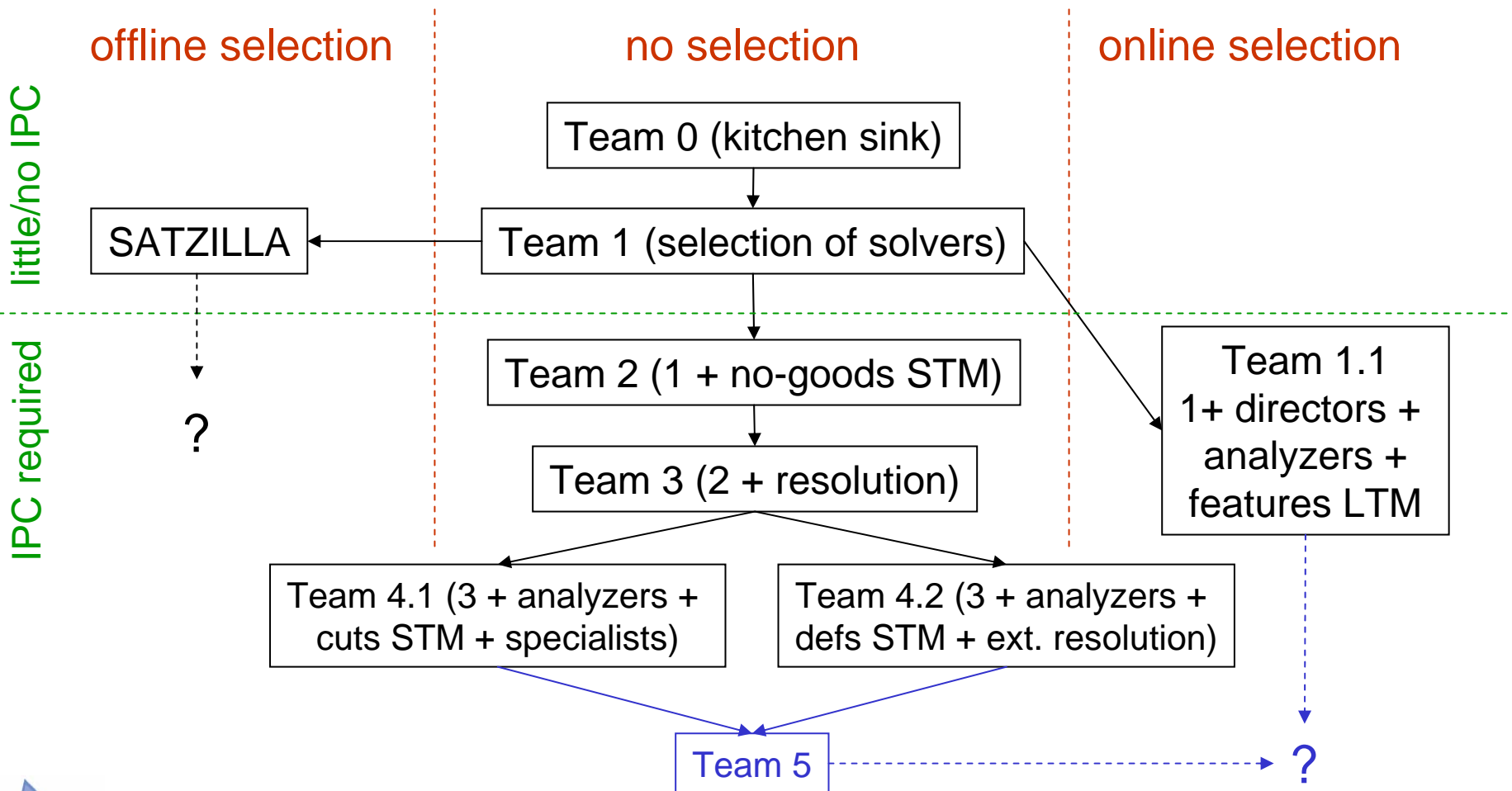


R&D roadmap (I)

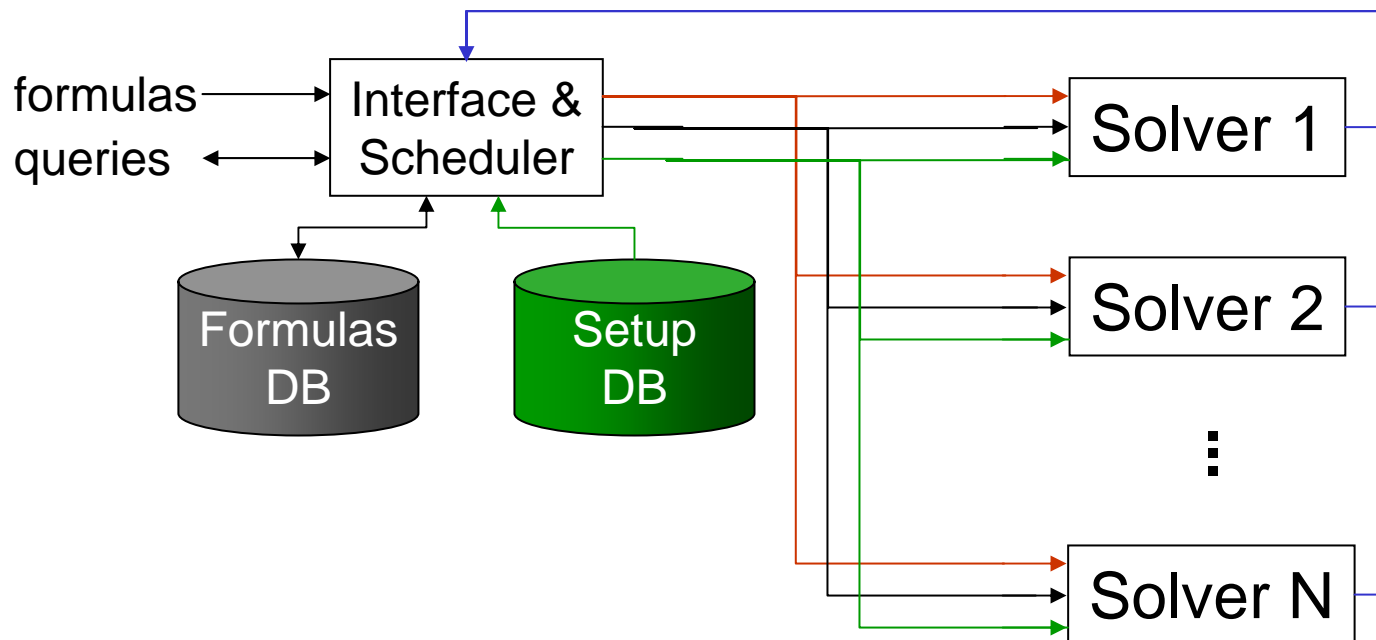
- **Baseline: the kitchen sink!**
 - Consider the **best state-of-the-art SAT solvers** (complete, incomplete, special purpose, ...)
 - Consider **publicly available SAT instances and models**;
 - Extract a **representative test set** (a few thousands instances)
 - Run **all the solvers in parallel** on each instance, keep the run time of the **best performer as a baseline**
- **Team-based development**
 - Approach Hi-NRG full fledged architecture **gradually**
 - Build **teams of core reasoners** with **increasing strength**: the simplest team must be better than the “kitchen sink” baseline
 - **Develop** interface, scheduler and system bus **facilities incrementally**



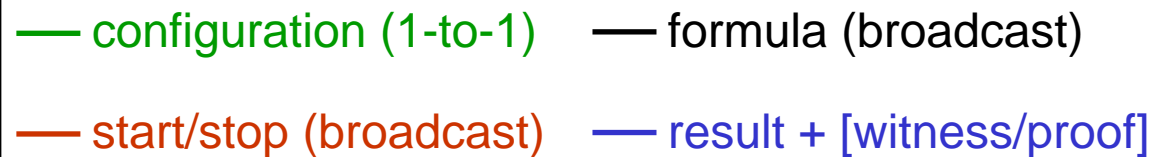
R&D Roadmap (II)



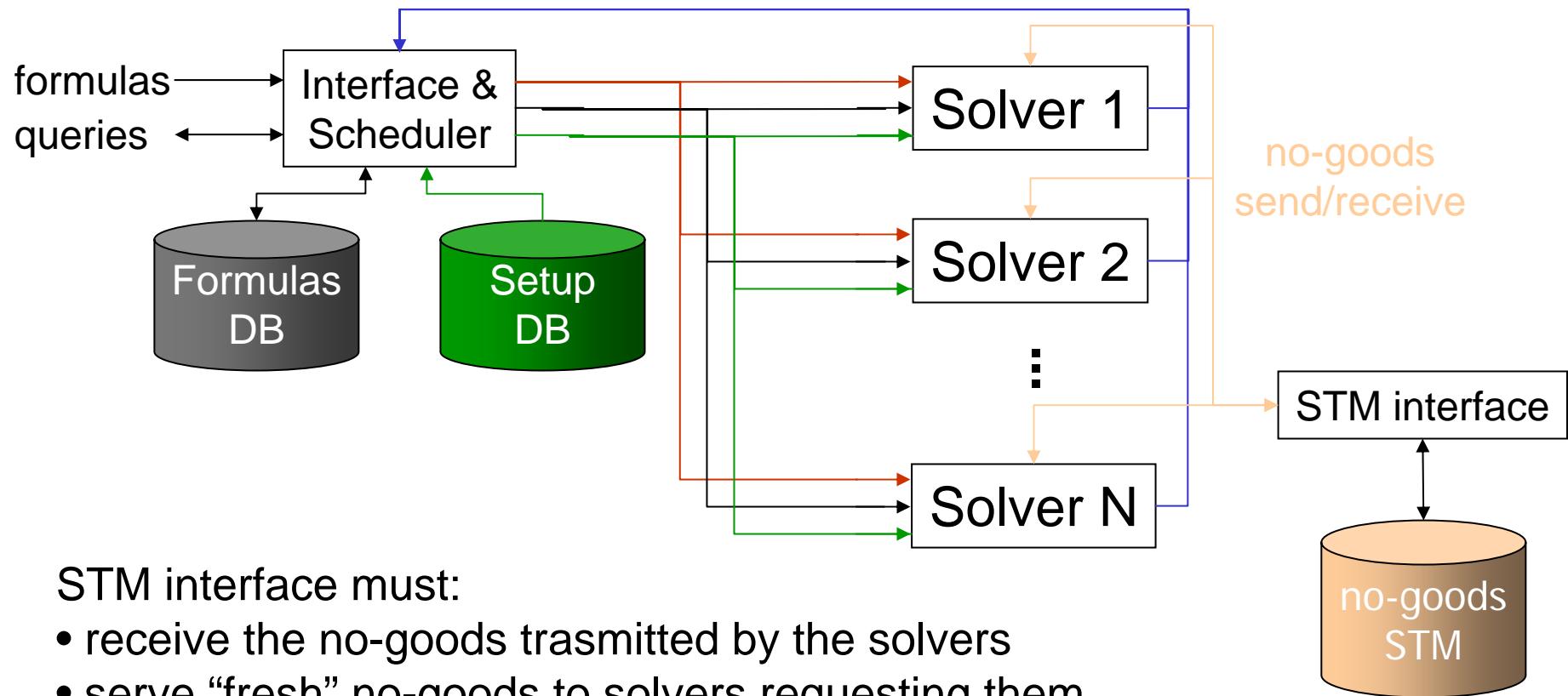
Team 0/1 structure



System bus
signals



Team 2 structure

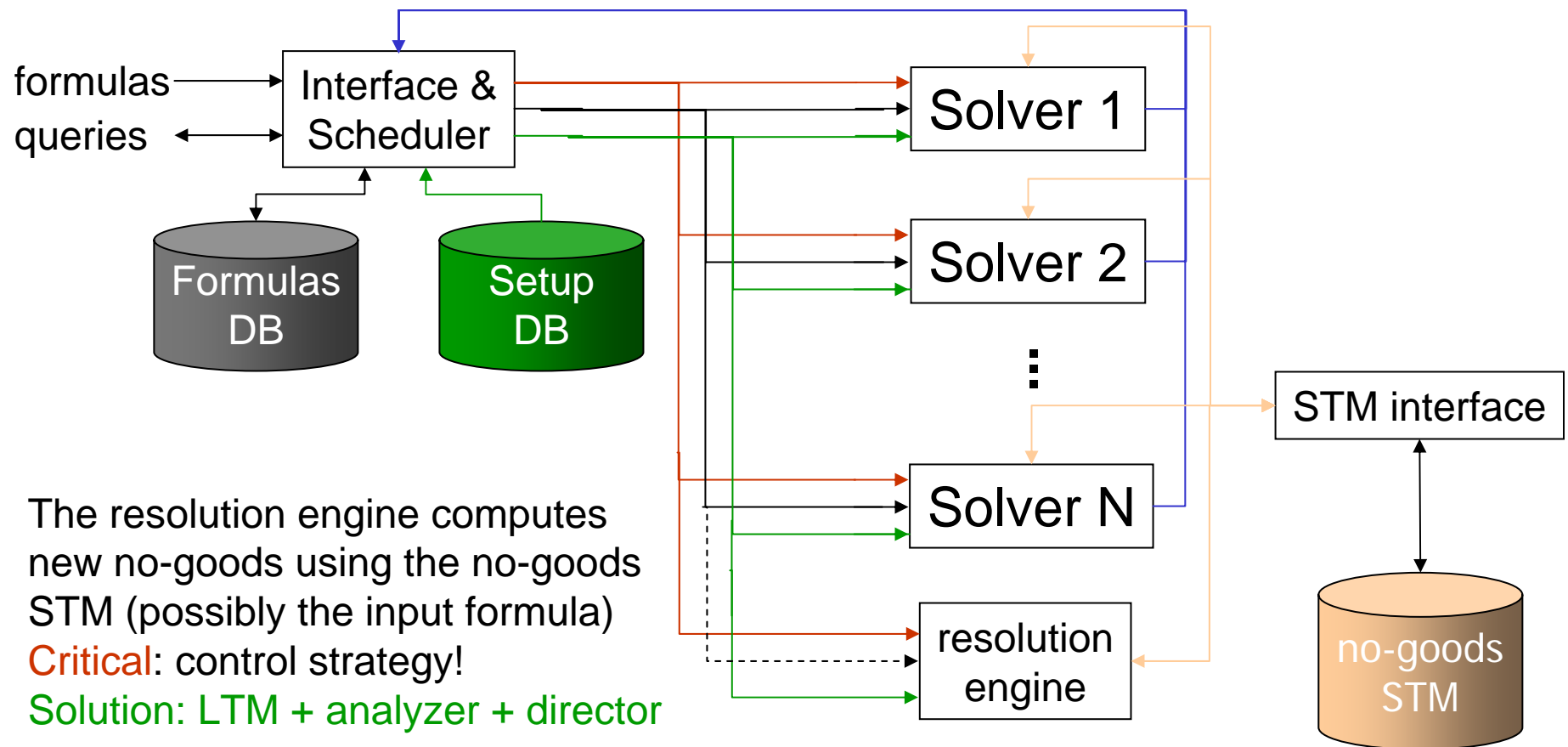


STM interface must:

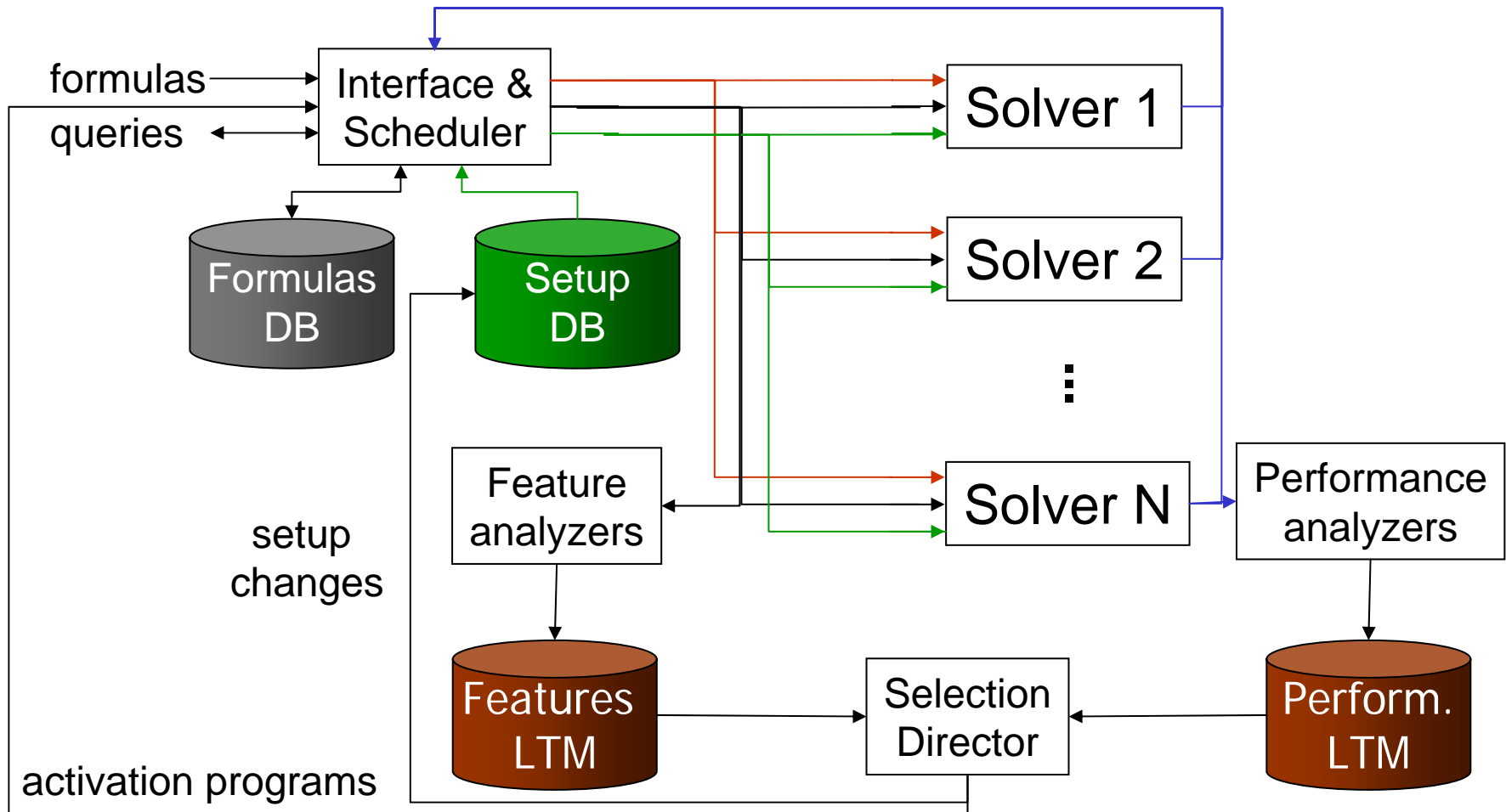
- receive the no-goods transmitted by the solvers
- serve “fresh” no-goods to solvers requesting them
- remove, or avoid storing redundant no-goods



Team 3 structure



Team 1.1 structure



Conclusions

- P vs. NP question is of central importance in CS
- NP-completeness theory characterizes NP-P (if $P \neq NP$)
- SAT is one of the prototypical NP-complete problems
- State of the art in SAT demonstrates that the silver bullet is unlikely
- Hi-NRG project: make the P vs. NP question irrelevant in practice
- Ongoing work
 - Team 0/1 system bus, solver selection, preliminary experiments
 - Team 2-3 functional specifications, system bus prototype
 - Team >3 early design stage

