Model Checking and Games

Part VII - Satisfiability Solving

Rüdiger Ehlers, Clausthal University of Technology

September 2019

Model checking so far

Scalability evolution

- Basic explicit-state model checking: Expressive logic (LTL), but we can handle only models of a certain maximum size
- Symbolic model checking: Use of BDDs allows to tackle larger state spaces.
- ...but there are many applications where this is still not enough.

Next step

Let us simplify the model checking problem a bit! We want to only know if a certain *bad state* is reachable. Can we analyze (reactive) systems with huge state spaces in this way?

Bounded Model Checking

Further simplification

Let us assume that for some value of n, we only want to know if a *bad state* can be reached in *exactly* n steps.

This is called *bounded model checking*

Effect of this simplification

We can now model the verification problem for some transition system $\mathcal{T} = (S, \mathrm{AP}, T, S_0)$ and some set of bad states B as a search problem for some path $\pi_0, \pi_1, \ldots, \pi_n$ with $\pi_0 \in S_0$, $\pi_n \in B$, and for all $0 \leq i < n$, we have $(\pi_i, \pi_{i+1}) \in T$.

SAT-based search

SAT-bases search

We will look at *SAT-solving* as one particular search approach that can deal with **huge** Kripke stuctures/transition systems in model checking.

SAT solvers are general tools for finding satisfiable assignments to Boolean functions.

Bounded model checking is only *one* particular application.

Agenda

- What is satisfiability solving?
- How can it be used for bounded model checking?
- How does it work?

The satisfiability problem

Definition

Let $\mathcal{V}$ be a set of boolean variables and ψ be a boolean formula over $\mathcal{V}$. The satisfiability problem for ψ is to decide whether there exists an assignment to the variables in $\mathcal{V}$ such that substituting the variables in ψ by their values in the assignment results in a boolean formula that is equivalent to **tt**.

Example

$$\psi(x, y, z) = (x \vee y \vee \neg z) \wedge (x \vee (y \wedge z)) \wedge (\neg x \vee \neg z)$$

Another example

Example

Consider the following boolean formula over the variables $\{x, y, z\}$:

$$\psi'(x, y, z) = \neg y \wedge (\neg x \vee z) \wedge (x \vee y) \wedge (\neg z \vee y)$$

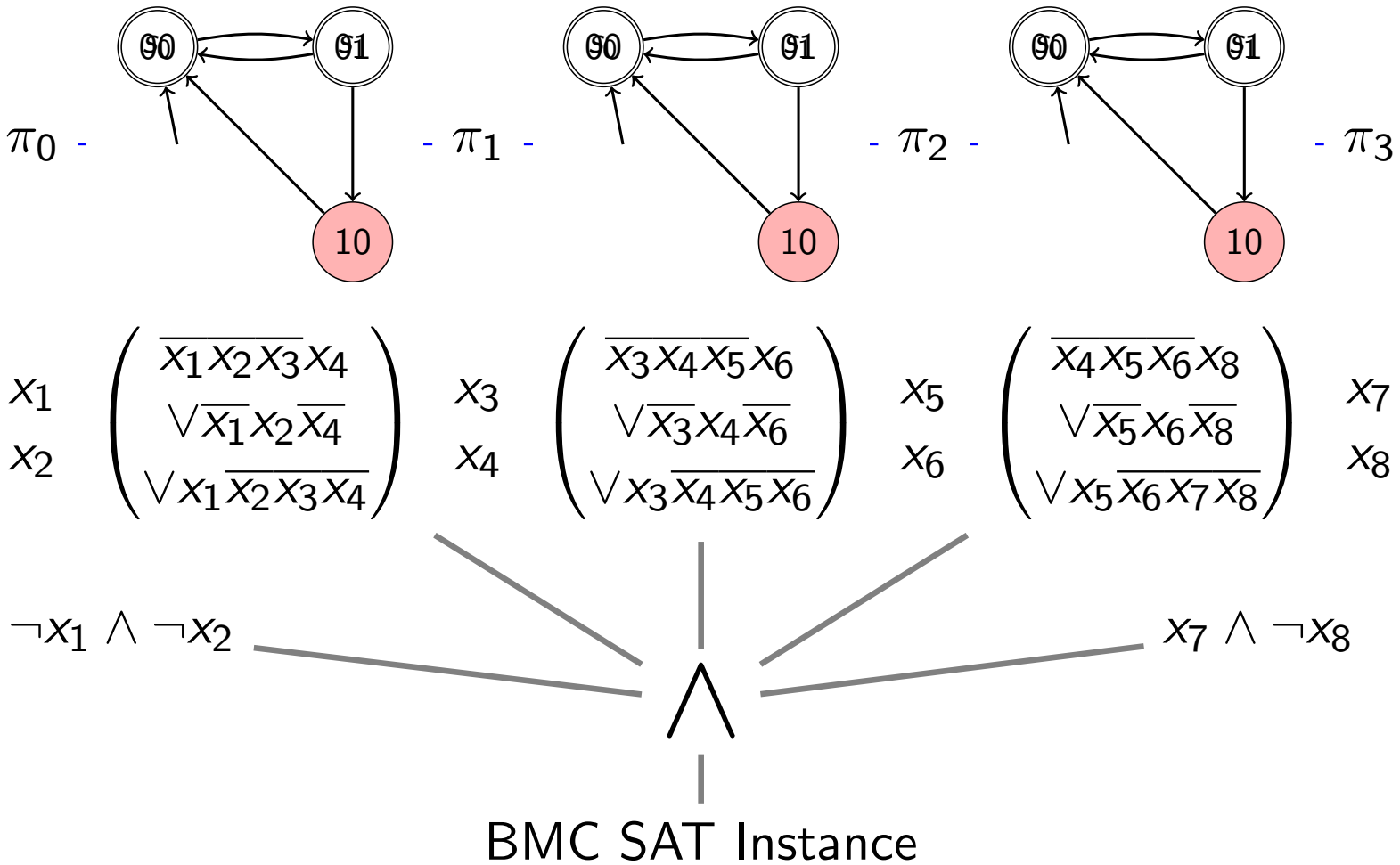
The formula is *unsatisfiable*. For any assignment to the variables x, y, and z, we have $\psi'(x, y, z) = \mathbf{ff}$. This can be proven, for example, by constructing a *truth table* for ψ' and checking if every cell in the $\psi'(x, y, z)$ column of the resulting table contains a **ff** value.

SAT solvers

A **SAT solver** is a tool that takes as input a boolean formula and checks whether it is satisfiable or not. In case of a positive answer, a SAT solver also computes a satisfying assignment.

The boolean formula used as input is also called a *SAT instance* in the literature.

SAT solving for bounded model checking



BMC as a SAT instance

What did we just do?

- Given a transition system $\mathcal{T} = (S, \mathrm{AP}, T, S_0)$ and some set of bad states B, we want to find a trace $\pi_0, \pi_1, \ldots, \pi_n$ with $\pi_0 \in S_0$, $\pi_n \in B$, and for all $0 \leq i < n$, we have $(\pi_i, \pi_{i+1}) \in T$.
- We encoded the problem into a SAT instance, where we have $(n+1)$ copies of Boolean variables for the state along the trace
- We replicated the encoding of the translation relation for all steps of the trace
- We added additional constraints for the initial and error states (at the beginning and the end of the trace)
- The resulting SAT encoding can be then be fed to a SAT solver.

CNF

SAT solvers

Most SAT solvers require the input to be in conjunctive normal form (CNF).

Definition

A boolean formula ψ over a set of variables $\mathcal{V} = \{v_1, \ldots, v_n\}$ is in *conjunctive normal form* (CNF) if it is of the form

$$\psi(v_1, \ldots, v_n) = \bigwedge_{i \in \{1,\ldots,m\}} c_i,$$

where each c_i is a *clause* over $\mathcal{V}$, i.e., a disjunction of *literals*. A literal is a variable or its negation.

CNF or not?

Some cases

$$\psi(x, y, z) = (x \vee y \vee \neg z) \wedge (x \vee (y \wedge z)) \wedge (\neg x \vee \neg z)$$

$\rightarrow$ No!

$$\psi'(x, y, z) = \neg y \wedge (\neg x \vee z) \wedge (x \vee y) \wedge (\neg z \vee y)$$

$\rightarrow$ Yes!

Translation to CNF

$$\psi(x, y, z) = (x \vee y \vee \neg z) \wedge (x \vee (y \wedge z)) \wedge (\neg x \vee \neg z)$$

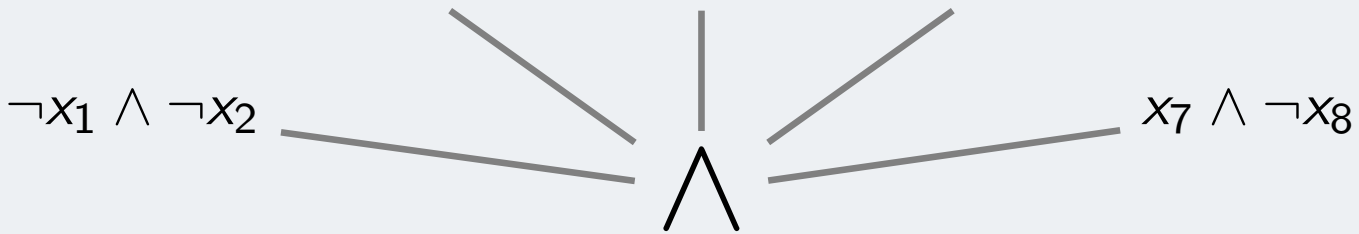
$$\Downarrow$$

$$\psi(x, y, z) = (x \vee y \vee \neg z) \wedge (x \vee y) \wedge (x \vee z) \wedge (\neg x \vee \neg z)$$

Translation from CNF of our example

Original encoding

$$\begin{pmatrix} \overline{x_1 x_2 x_3} x_4 \\ \vee \overline{x_1} x_2 \overline{x_4} \\ \vee x_1 \overline{x_2 x_3 x_4} \end{pmatrix} \quad \begin{pmatrix} \overline{x_3 x_4 x_5} x_6 \\ \vee \overline{x_3} x_4 \overline{x_6} \\ \vee x_3 \overline{x_4 x_5 x_6} \end{pmatrix} \quad \begin{pmatrix} \overline{x_4 x_5 x_6} x_8 \\ \vee \overline{x_5} x_6 \overline{x_8} \\ \vee x_5 \overline{x_6 x_7 x_8} \end{pmatrix}$$



CNF Encoding

$$\begin{aligned} & \neg x_1 \wedge \neg x_2 \wedge x_7 \wedge \neg x_8 \\ \wedge\ & (\neg x_1 \vee \neg x_2) \wedge (\neg x_3 \vee \neg x_4) \wedge (\neg x_3 \vee x_2) \vee (\neg x_1 \vee \neg x_4) \vee (\neg x_2 \vee \neg x_4) \\ \wedge\ & (x_1 \vee x_2 \vee x_3 \vee x_4) \\ \wedge\ & \ldots \end{aligned}$$

Translation from CNF of our example

CNF Encoding

$$
\begin{aligned}
&\neg x_1 \wedge \neg x_2 \wedge x_7 \wedge \neg x_8 \\
&\wedge (\neg x_1 \vee \neg x_2) \wedge (\neg x_3 \vee \neg x_4) \wedge (\neg x_3 \vee x_2) \vee (\neg x_1 \vee \neg x_4) \vee (\neg x_2 \vee \neg x_4) \\
&\wedge (x_1 \vee x_2 \vee x_3 \vee x_4) \\
&\wedge (\neg x_5 \vee \neg x_4) \wedge (\neg x_5 \vee \neg x_6) \wedge (\neg x_5 \vee x_4) \vee (\neg x_3 \vee \neg x_6) \vee (\neg x_4 \vee \neg x_6) \\
&\wedge (x_3 \vee x_4 \vee x_5 \vee x_6) \\
&\wedge (\neg x_5 \vee \neg x_6) \wedge (\neg x_7 \vee \neg x_8) \wedge (\neg x_7 \vee x_6) \vee (\neg x_5 \vee \neg x_8) \vee (\neg x_6 \vee \neg x_8) \\
&\wedge (x_5 \vee x_6 \vee x_7 \vee x_8)
\end{aligned}
$$

Input encoding for modern SAT solvers

Solver input file

- Most modern SAT solvers take input files in the *DIMACS* input format, which is a text format.
- They start with a line of the form `p cnf <nofVars> <nofClauses>`.
- The first line is followed by the *clauses*, which are a sequence of numbers, separated by a space, and terminated with a 0. Positive numbers represent positive literals, negative ones represent negative literals.

Example

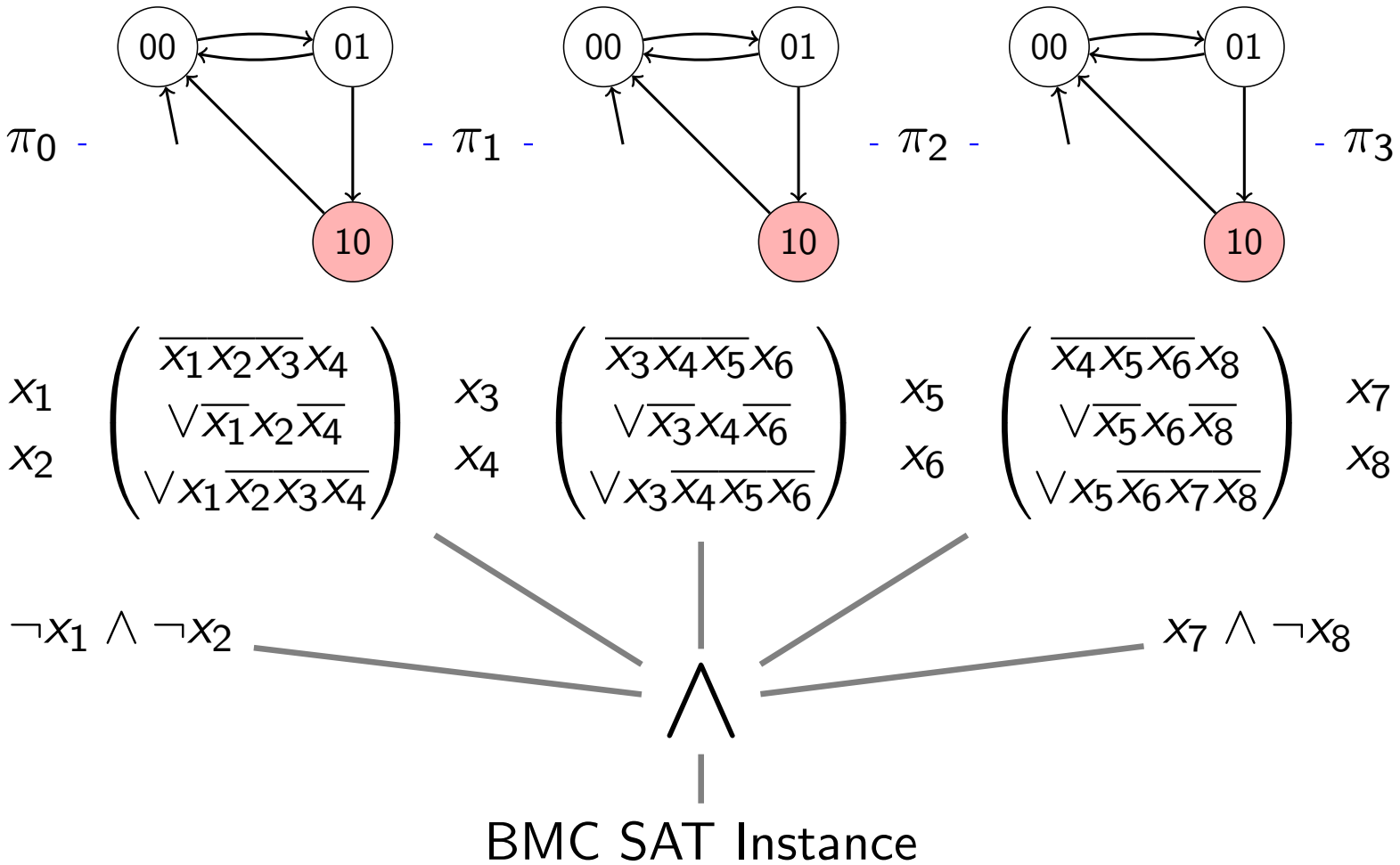
The formula $(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2)$ can be encoded as follows:

```
p cnf 2 2
1 -2 0
-1 2 0
```

Encoding of our bounded model checking problem

```
p cnf 8 22
-1 0
-2 0
7 0
-8 0
-1 -2 0
-3 -4 0
-3 2 0
-1 -4 0
-2 -4 0
1 2 3 4 0
-3 -4 0
-5 -6 0
-5 4 0
-3 -6 0
-4 -6 0
3 4 5 6 0
```

Interpreting the output of the SAT solver



Summary of practical use of SAT solvers

Main use for us

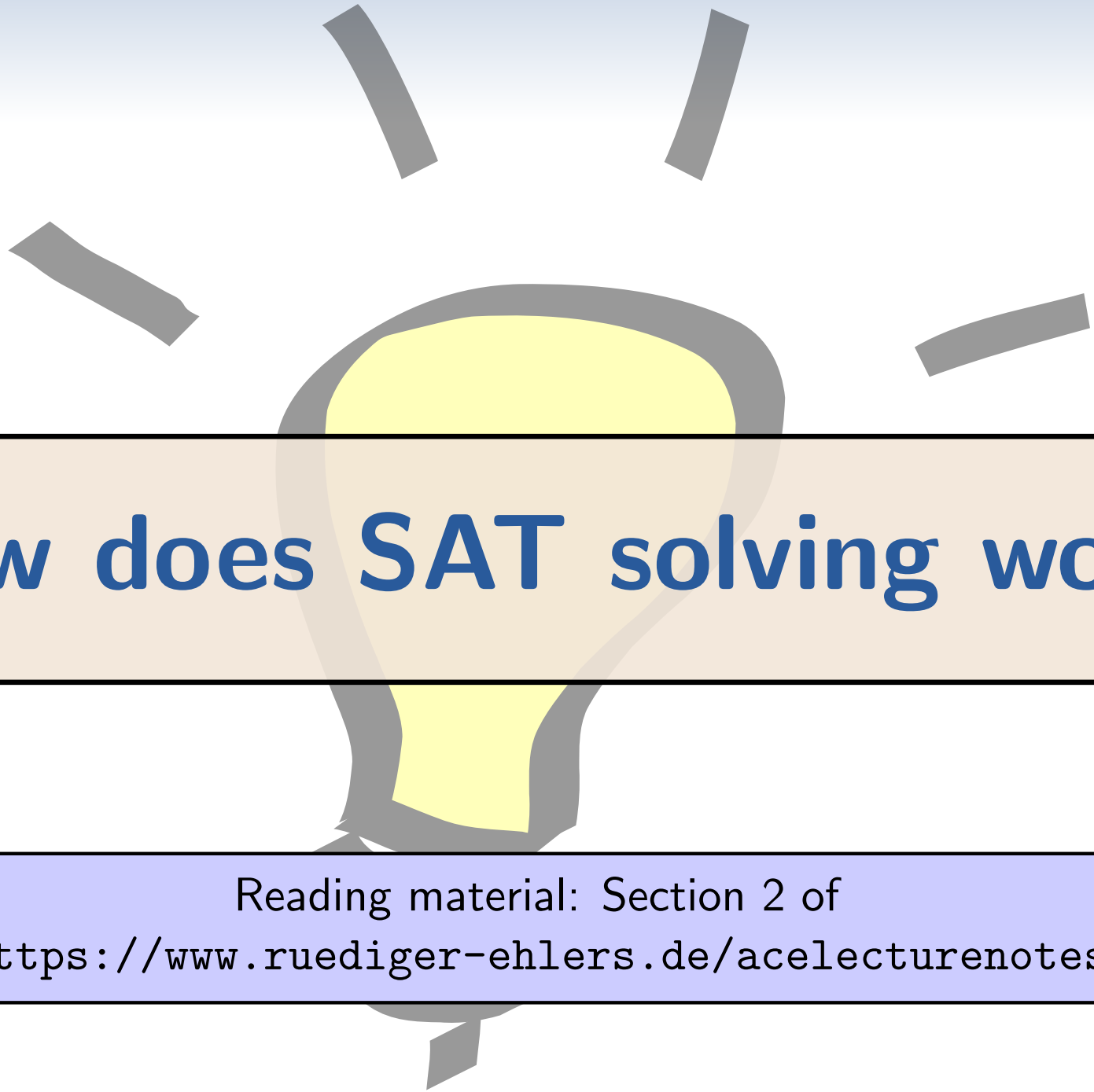
- Bounded model checking!
- (there are other applications as well).

Requirement for practical use

SAT solvers need to be *fast* for them to be useful.

Despite the NP-completeness of the problem that they solve, they perform quite well in many applications.

They do so by employing a bunch of *heuristics*.



How does SAT solving work?

Reading material: Section 2 of
https://www.ruediger-ehlers.de/acelecturenotes

A first example: Sudoku

1				7				
		9	3		8	2		
		2			9		8	
	8					4	5	2
			6					
		1					3	
	4			1		3	7	
		6	7			9		
	1							

Why does Sudoku solving work in practice?

Answer: Powerful combination of *inference* techniques

- Saturating the solution with *safe* choices
- Making a *guess* whenever no safe choices can be made.

Corollary

...there is no need to be afraid of NP-complete problems!

The generalization of these ideas to SAT solving...

$\rightarrow$ ... makes *satisfiability solvers* work.

The DPLL Procedure

Example (in CNF)

Let us consider the following boolean formula over the variables $\mathcal{V} = \{a, b, c, x, y, z\}$:

$$\psi(x,y,z,a,b,c) = \underbrace{x}_{c_1} \wedge \underbrace{(\neg x \vee y \vee z)}_{c_2} \wedge \underbrace{(\neg y \vee z)}_{c_3} \wedge \underbrace{(y \vee a)}_{c_4}$$
$$\wedge \underbrace{(y \vee b)}_{c_5} \wedge \underbrace{(\neg a \vee \neg b)}_{c_6} \wedge \underbrace{(\neg y \vee a \vee b)}_{c_7}$$
$$\wedge \underbrace{(\neg a \vee b \vee c)}_{c_8} \wedge \underbrace{(a \vee b \vee \neg c)}_{c_9}$$
$$\wedge \underbrace{(a \vee \neg b \vee c)}_{c_{10}} \wedge \underbrace{(a \vee \neg b \vee \neg c)}_{c_{11}}$$

A first approach to SAT solving

First try

Idea: Let us build a *truth table!*

Problem: It has 64 entries!

Another approach

New idea: Let us gradually build a satisfying assignment and check after every *choice* if the *partial assignment* already violates a clause

Preparation: We fix an order of the variables: x, y, z, a, b, c

Our running example (again)

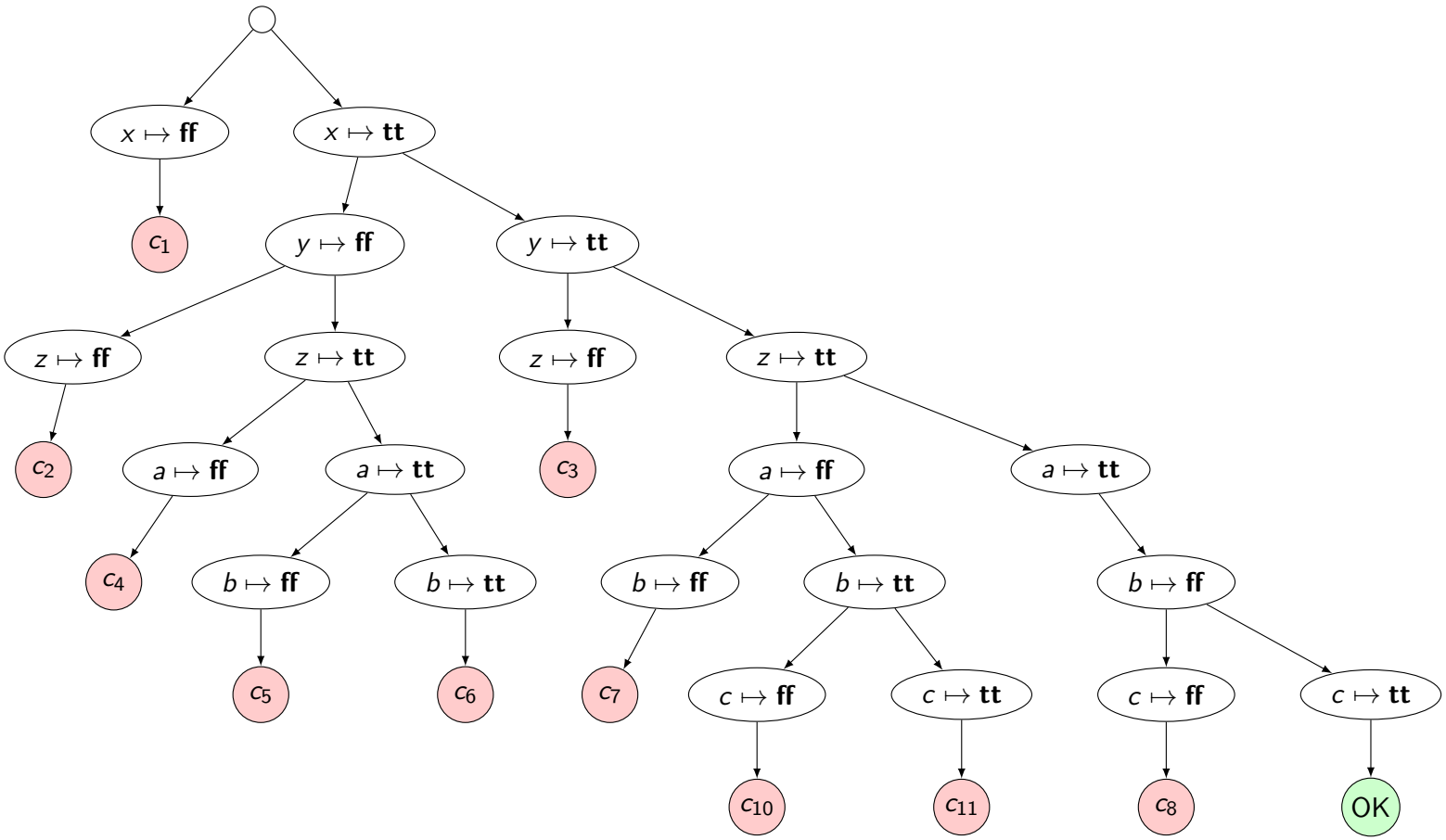
$$\psi(x,y,z,a,b,c) = \underbrace{x}_{c_1} \wedge \underbrace{(\neg x \vee y \vee z)}_{c_2} \wedge \underbrace{(\neg y \vee z)}_{c_3} \wedge \underbrace{(y \vee a)}_{c_4}$$
$$\wedge \underbrace{(y \vee b)}_{c_5} \wedge \underbrace{(\neg a \vee \neg b)}_{c_6} \wedge \underbrace{(\neg y \vee a \vee b)}_{c_7}$$
$$\wedge \underbrace{(\neg a \vee b \vee c)}_{c_8} \wedge \underbrace{(a \vee b \vee \neg c)}_{c_9}$$
$$\wedge \underbrace{(a \vee \neg b \vee c)}_{c_{10}} \wedge \underbrace{(a \vee \neg b \vee \neg c)}_{c_{11}}$$

Our first algorithm

Algorithm 1 A first, very simple, SAT solving algorithm

```
1:  function SEARCH(𝒱,ψ,{v_1 ↦ b_1,...,v_n ↦ b_n})
2:      if some clause in ψ is falsified by {v_1 ↦ b_1,...,v_n ↦ b_n} then
3:          return ∅
4:      end if
5:      if n = |𝒱| then
6:          return {v_1 ↦ b_1,...,v_n ↦ b_n}
7:      end if
8:      A ← SEARCH(𝒱,ψ,{v_1 ↦ b_1,...,v_n ↦ b_n, v_{n+1} ↦ false})
9:      if A ≠ ∅ then return A
10:     A' ← SEARCH(𝒱,ψ,{v_1 ↦ b_1,...,v_n ↦ b_n, v_{n+1} ↦ true})
11:     if A' ≠ ∅ then return A'
12:     return ∅
13: end function
```

The run of Algorithm 1 on our running example

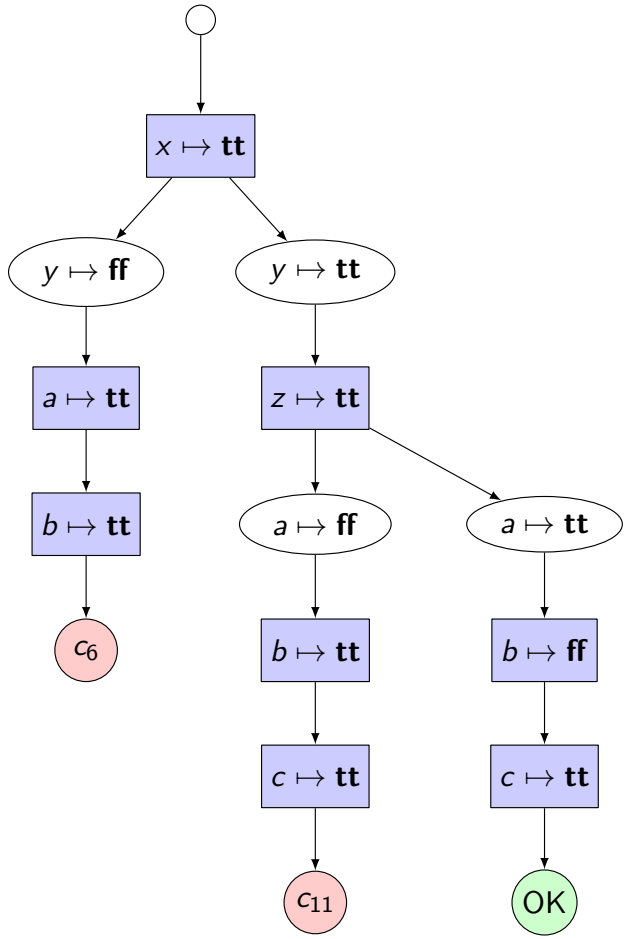


$$\psi(x,y,z,a,b,c) = \underbrace{x}_{c_1} \wedge \underbrace{(\neg x \vee y \vee z)}_{c_2} \wedge \underbrace{(\neg y \vee z)}_{c_3} \wedge \underbrace{(y \vee a)}_{c_4} \wedge \underbrace{(y \vee b)}_{c_5} \wedge \underbrace{(\neg a \vee \neg b)}_{c_6}$$

$$\wedge \underbrace{(\neg y \vee a \vee b)}_{c_7} \wedge \underbrace{(\neg a \vee b \vee c)}_{c_8} \wedge \underbrace{(a \vee b \vee \neg c)}_{c_9} \wedge \underbrace{(a \vee \neg b \vee c)}_{c_{10}} \wedge \underbrace{(a \vee \neg b \vee \neg c)}_{c_{11}}$$

Unit propagation

$$\psi(x,y,z,a,b,c) = \underbrace{x}_{c_1} \wedge \underbrace{(\neg x \vee y \vee z)}_{c_2} \wedge \underbrace{(\neg y \vee z)}_{c_3}$$
$$\wedge \underbrace{(y \vee a)}_{c_4} \wedge \underbrace{(y \vee b)}_{c_5} \wedge \underbrace{(\neg a \vee \neg b)}_{c_6}$$
$$\wedge \underbrace{(\neg y \vee a \vee b)}_{c_7} \wedge \underbrace{(\neg a \vee b \vee c)}_{c_8}$$
$$\wedge \underbrace{(a \vee b \vee \neg c)}_{c_9} \wedge \underbrace{(a \vee \neg b \vee c)}_{c_{10}}$$
$$\wedge \underbrace{(a \vee \neg b \vee \neg c)}_{c_{11}}$$



(BB)

Pure literal rule

Example

$\psi(a, b, c, d, e) = (\neg a \vee b) \wedge (\neg b \vee c) \wedge (\neg c \vee d) \wedge (\neg c \vee e) \wedge (\neg d \vee \neg e)$

Note: Unit propagation cannot be applied.

Solution

Solution with only applying the pure literal rule:

$$\{a \mapsto \mathbf{ff}, b \mapsto \mathbf{ff}, c \mapsto \mathbf{ff}, d \mapsto \mathbf{ff}\}$$

The value of e does not matter.

The complete DPLL algorithm

Algorithm 2 The complete DPLL algorithm with *unit propagation* and with the *pure literal rule*. The parameter $\mathcal{V}$ represents the set of variables, ψ is the CNF formula, and $A : \mathcal{V} \rightharpoonup \mathbb{B}$ is the current partial assignment.

```
1:  function SEARCH(𝒱,ψ,A)
2:      for all assignments v_k = b_k implied by (ψ, A) by unit propagation do
3:          A := A ∪ {v_k ↦ b_k}
4:      end for
5:      for all assignments v_k = b_k implied by (ψ, A) by pure literal elimination do
6:          A := A ∪ {v_k ↦ b_k}
7:      end for
8:      if some clause in ψ is falsified by A then
9:          return ∅
10:     end if
11:     if |A| = |𝒱| then
12:         return A
13:     end if
14:     Pick a variable v that is not yet in the domain of A
15:     A' ← SEARCH(𝒱,ψ,A ∪ {v ↦ false})
16:     if A' ≠ ∅ then return A'
17:     A' ← SEARCH(𝒱,ψ,A ∪ {v ↦ true})
18:     if A' ≠ ∅ then return A'
19:     return ∅
20: end function
```

DPLL SAT Solving

The Davis–Putnam–Logemann–Loveland (DPLL) procedure

- Backtracking
- Unit propagation
- Pure literal elimination

First appearance: 1962

Properties

- DPLL is often faster than building a truth table
- **DPLL is still comparably slow**

Conflict-Driven clause learning (CDCL)

Running example

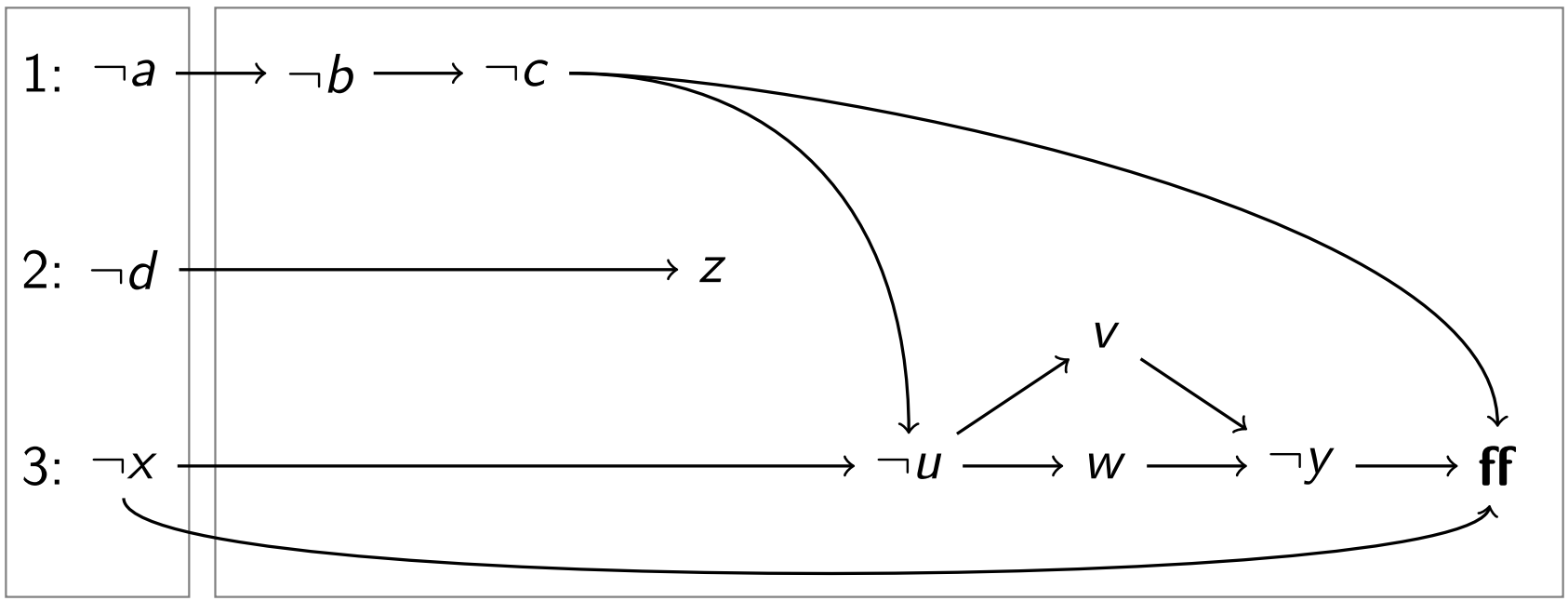
$$\begin{aligned}\varphi(a,b,c,d,x,y,z,u,v,w) = {} & (a \vee \neg b) && \wedge (\neg a \vee d)\\ & \wedge (\neg a \vee y) && \wedge (b \vee \neg c)\\ & \wedge (d \vee z) && \wedge (\neg d \vee \neg z)\\ & \wedge (c \vee x \vee \neg u) && \wedge (u \vee v)\\ & \wedge (u \vee w) && \wedge (\neg v \vee \neg w \vee \neg y)\\ & \wedge (c \vee x \vee y) && \wedge (c \vee \neg x \vee \neg y)\\ & \wedge (c \vee \neg x \vee y)\end{aligned}$$

DPLL variable order

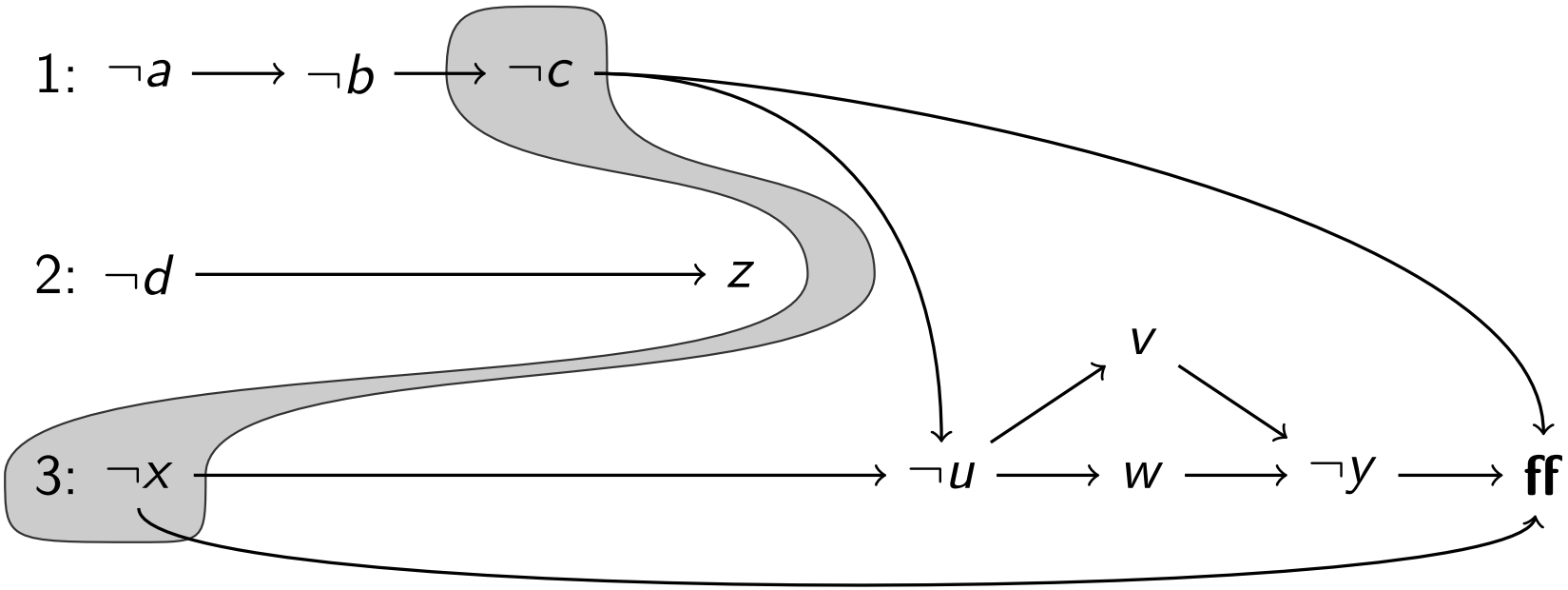
a,b,c,d,x,y,z,u,v,w

$\rightarrow$ Let's do DPLL on φ on the blackboard (copy)!

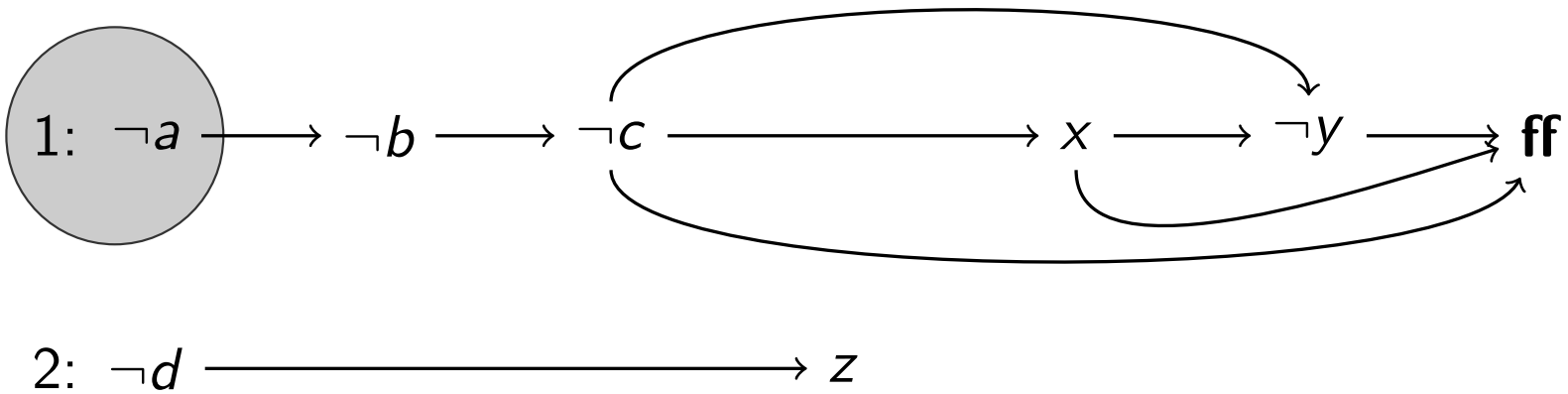
Implication graph



A cut through an implication graph



Another implication graph (and a cut through it)



DPLL+Clause learning = CDCL

Core techniques in CDCL solvers

- Backtracking
- Unit propagation
- Pure literal elimination
- Conflict-driven clause learning

Main publications for CDCL were made between 1997 and 2004.[a]

[a] Handbook of Satisfiability, page 119, first paragraph of Section 3.6.3.3

Which cut to choose?

Most SAT solvers...

...use *augmenting* clauses.

These contain exactly one literal for the highest decision level.[a]

[a] Source: Handbook of Satisfiability, p. 119

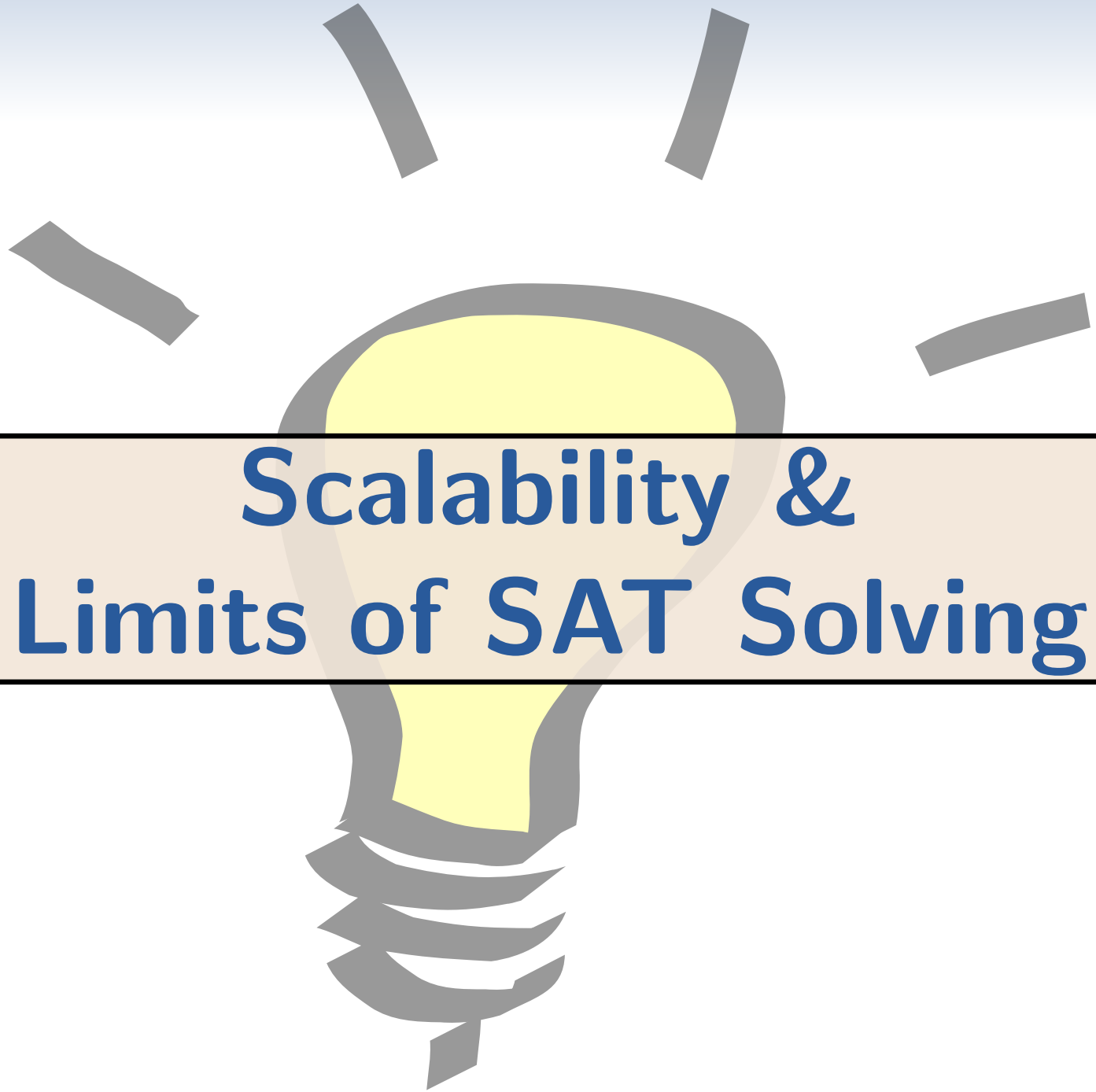
A property of Augmenting Clauses

They give rise to unit propagation directly after backtracking.

Augmenting techniques for CDCL/DPLL Solvers

Important examples

- Fast implementations
- Smart & efficient variable selection
- Random restarts and deletion of learned clauses



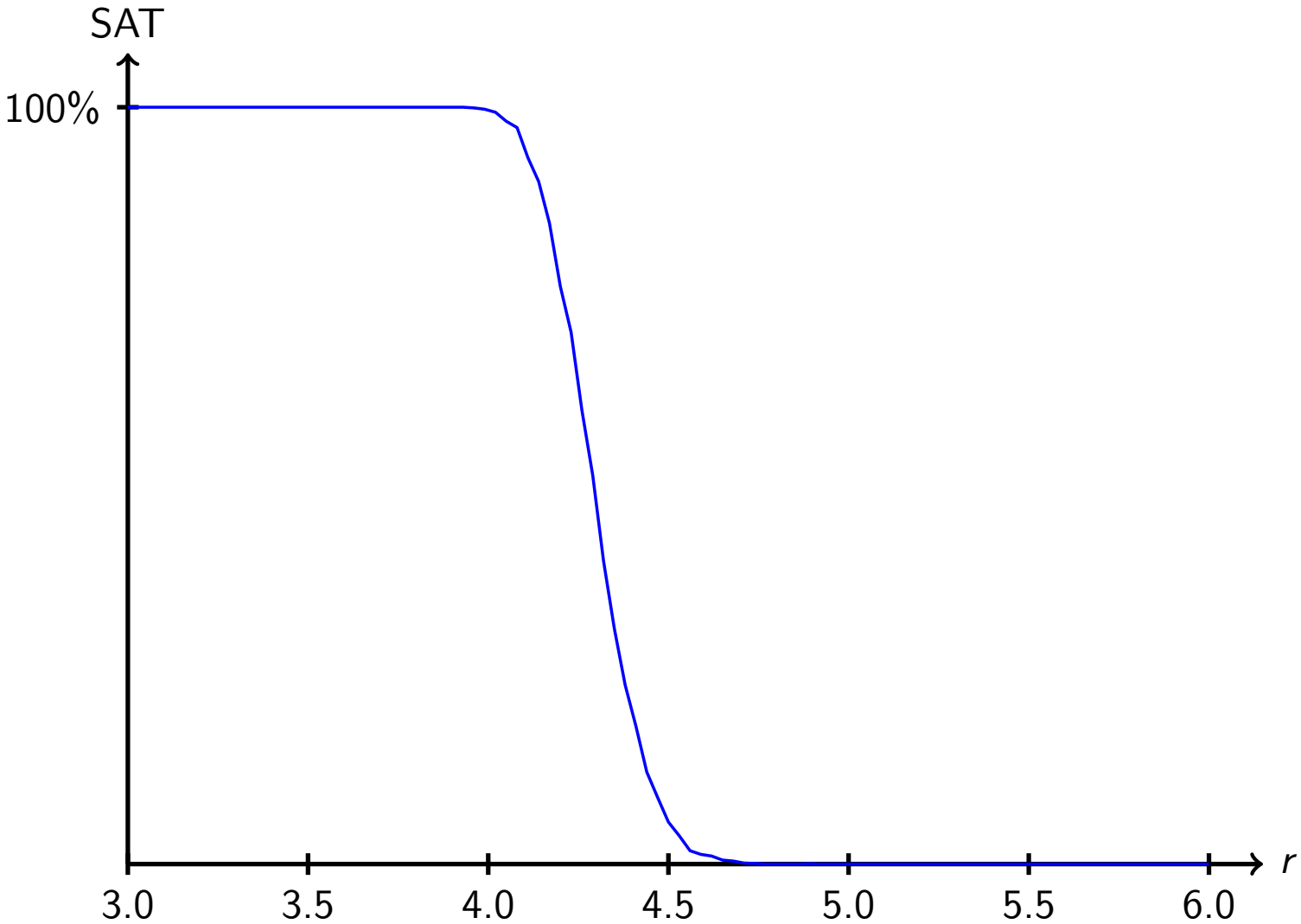
Scalability & Limits of SAT Solving

A quotation from Donald Knuth

Almost nothing is known about why the heuristics in modern solvers work as well as they do, or why they fail when they do. Most of the techniques that have turned out to be important were originally introduced for the wrong reasons!

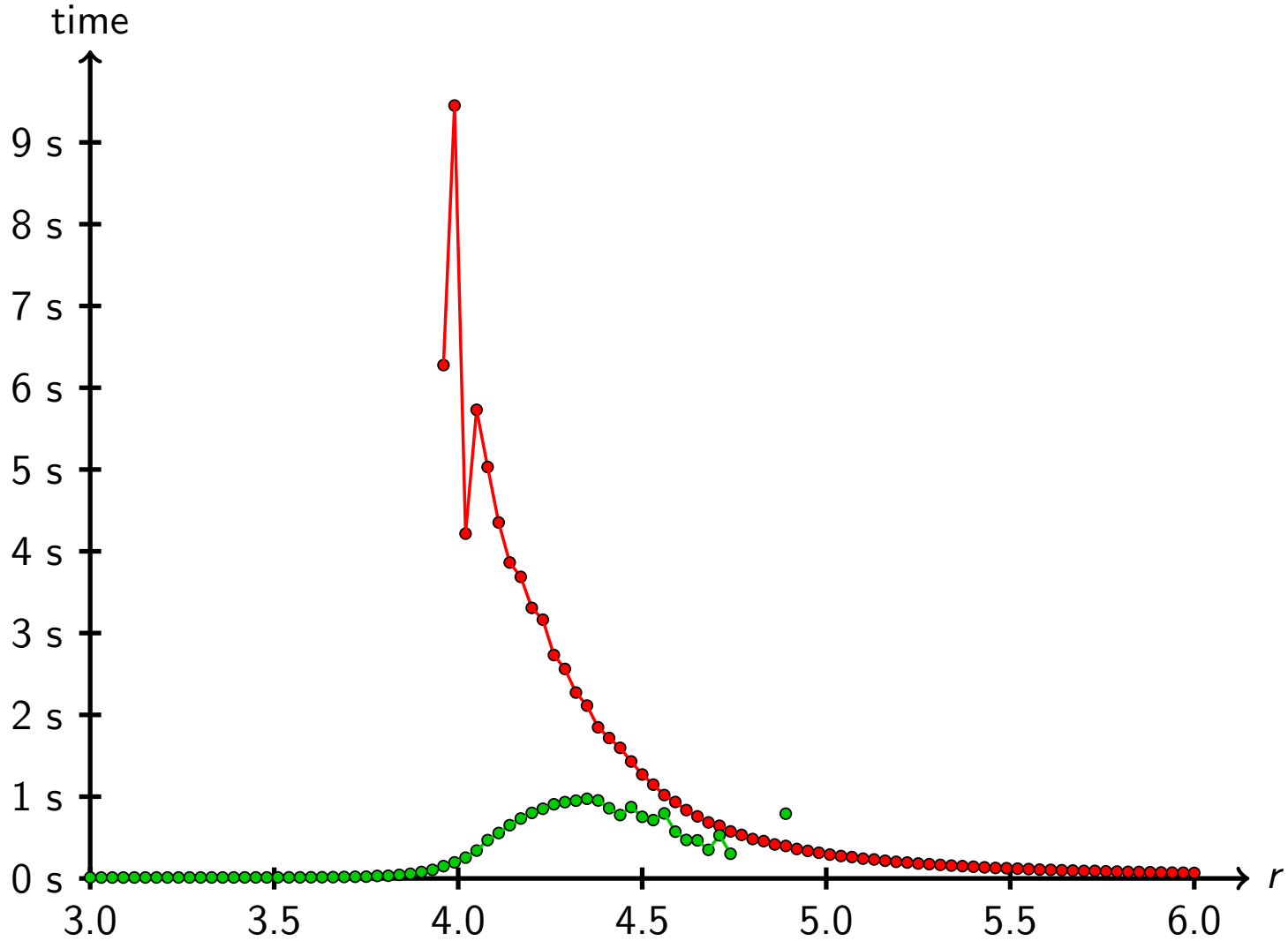
Donald Knuth in an interview, see lecture notes for citation information

Random 3SAT – Satisfiability



Source: $101 \cdot 2270$ runs of `picosat` (within 1490 minutes), $n = 250$

Random 3SAT – Satisfiability



Source: $101 \cdot 2270$ runs of `picosat` (within 1490 minutes), $n = 250$

Studying random 3SAT

Observation

The study of random 3SAT instances indicates that SAT solving seems to work well on instances whose satisfiability or unsatisfiability can be shown easily. Only for values of r where the satisfiability of an instance is relatively uncertain, the solver needs some more computation time.

Interpretation

This observation suggests that a similar effect may hold for SAT instances from practice.

A Conjecture

Satisfiability Threshold Conjecture

For all $k \in \mathbb{N}$ with $k \geq 3$, there exists some value $t_k \in \mathbb{R}_{>0}$ such that for all $r \in \mathbb{R}_{>0}$:

$$\lim_{n \to \infty} \mathrm{Prob}(\textit{SAT instance in } F_k(n, rn) \text{ is satisfiable}) = \begin{cases} 1 & \text{if } r < t_k \\ 0 & \text{if } r > t_k \end{cases}$$

A difficult problem for SAT

The pidgeon hole principle

Assume that we have n pidgeons that we want to place into m pidgeon holes such that in no pidgeon hole, there is more than one pidgeon.

It is relatively easy to see that this can only work if we have $m \geq n$.

SAT Encoding

Using $n \cdot m$ variables, we need $n + n \cdot (n-1) \cdot m$ clauses, with a mean number of $\frac{nm+2n(n-1)m}{n+n(n-1)m}$ literals per clause.

(Demo: Minisat computation time, Demo: Learned clauses)

Analyzing the Pidgeon Hole Principle SAT instances (1)

Question

Can we somehow prove that the DPLL algorithm always performs badly for the pidgeon hole principle problem?

Answer

Yes!

Analyzing the Pidgeon Hole Principle SAT instances (2)

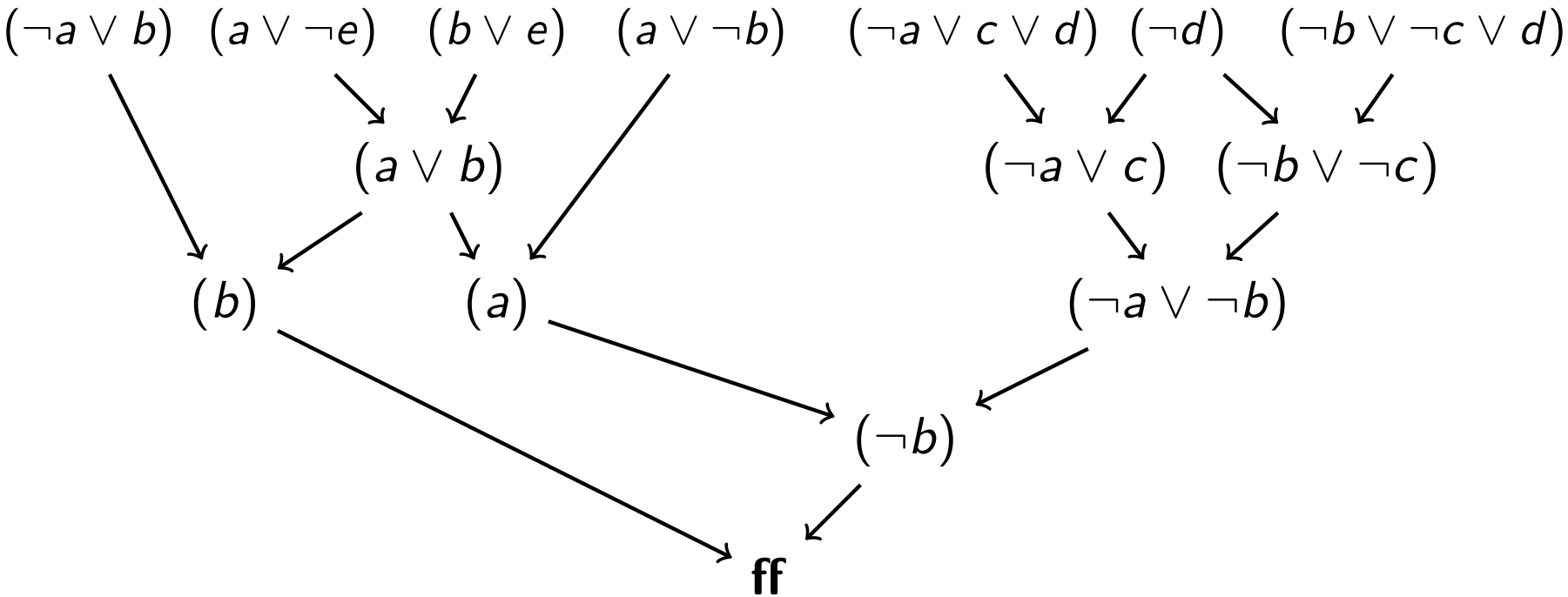
Approach

Every family of unsatisfiable SAT instances that only have exponential-sized *resolutions proofs* induce exponential-time CDCL algorithm performance, as the CDCL algorithm essentially performs a resolution proof (for unsatisfiable SAT instances).

Example for explaining resolution proofs on the black-board

$$(a \vee \neg e) \wedge (b \vee e) \wedge (\neg a \vee c \vee d) \wedge (\neg b \vee \neg c \vee d) \wedge (\neg d)$$
$$\wedge (\neg a \vee b) \wedge (a \vee \neg b)$$

Analyzing the Pidgeon Hole Principle SAT instances (3)



Analyzing the Pidgeon Hole Principle SAT instances (4)

Definition: Tree-like resolution proofs

This is a resolution proof that is tree-shaped. Every leaf contains a clause of the CNF, and the root is the **ff** clause.

A difficult problem

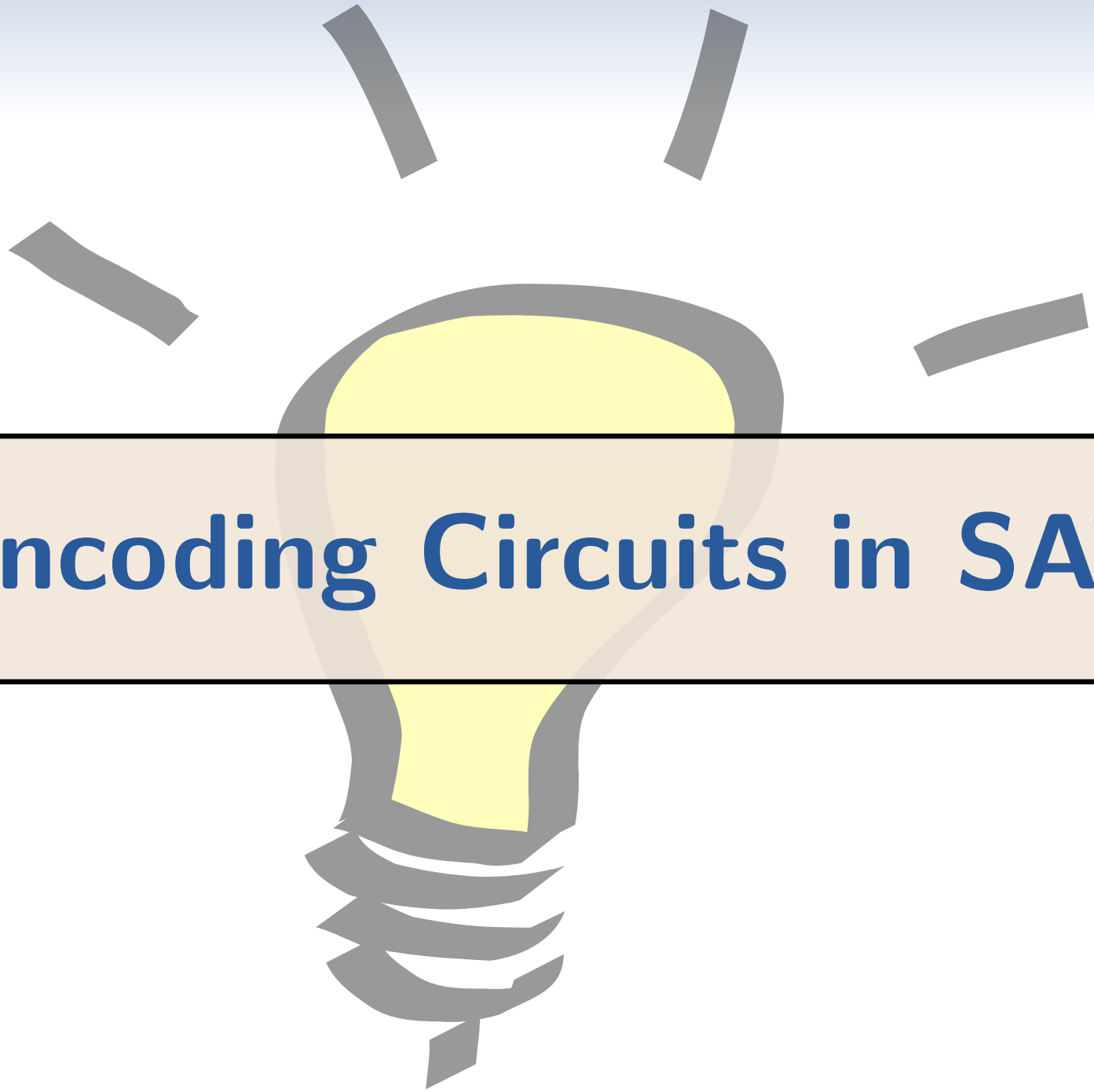
The pidgeon-hole principle problems require exponential-size resolution proofs.

Source: Alasdair Urquhart. Hard examples for resolution. J. ACM, 34(1):209–219, 1987. doi: 10.1145/7531.8928. URL http://doi.acm.org/10.1145/7531.8928

Pidgeon hole principle – Conclusion

The pidgeon hole principle SAT instances show that the reasoning rules applied by SAT solvers do not always lead to high performance. In fact, regardless of how the solver deletes learned clauses or how variables to branch on are selected, exponential running time cannot be avoided by a CDCL-style solver if no additional measures are taken.

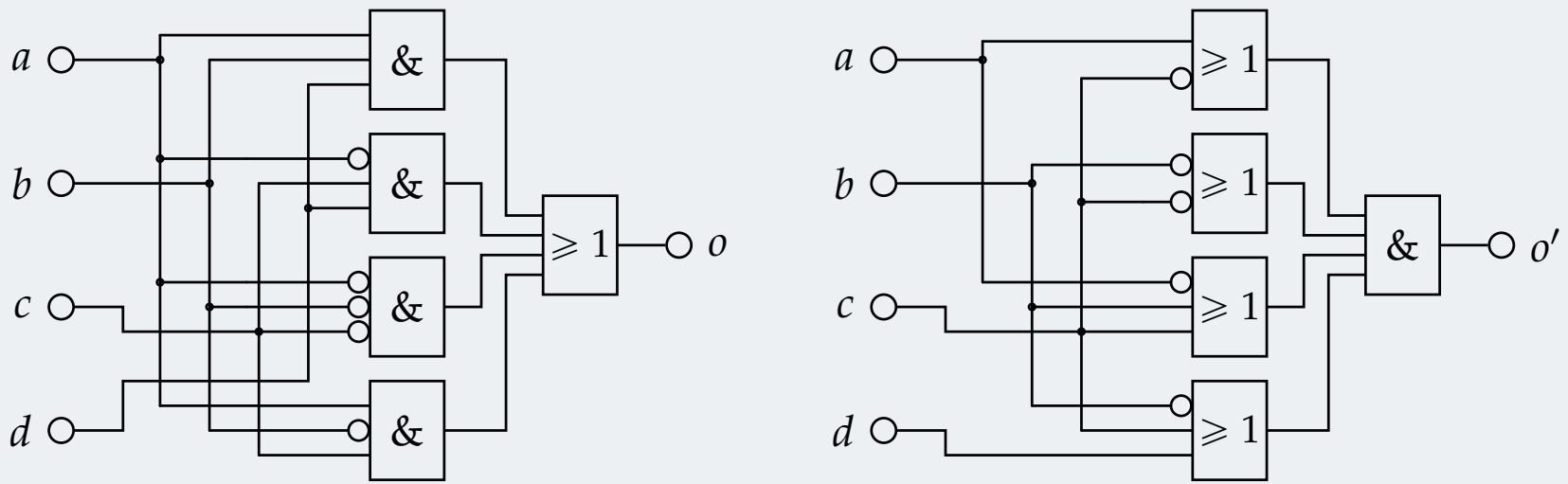
The study of pidgeon hole principle instances is relevant to the practice of SAT solving as many practical problems can include them as sub-problems when the current partial assignment by the solver is not a good one. In such a case, the solver can spend a lot of time on tackling an impossible-to-solve subproblem, which is one of the reasons why modern solver use random restarts.



Encoding Circuits in SAT

Circuit equivalence testing

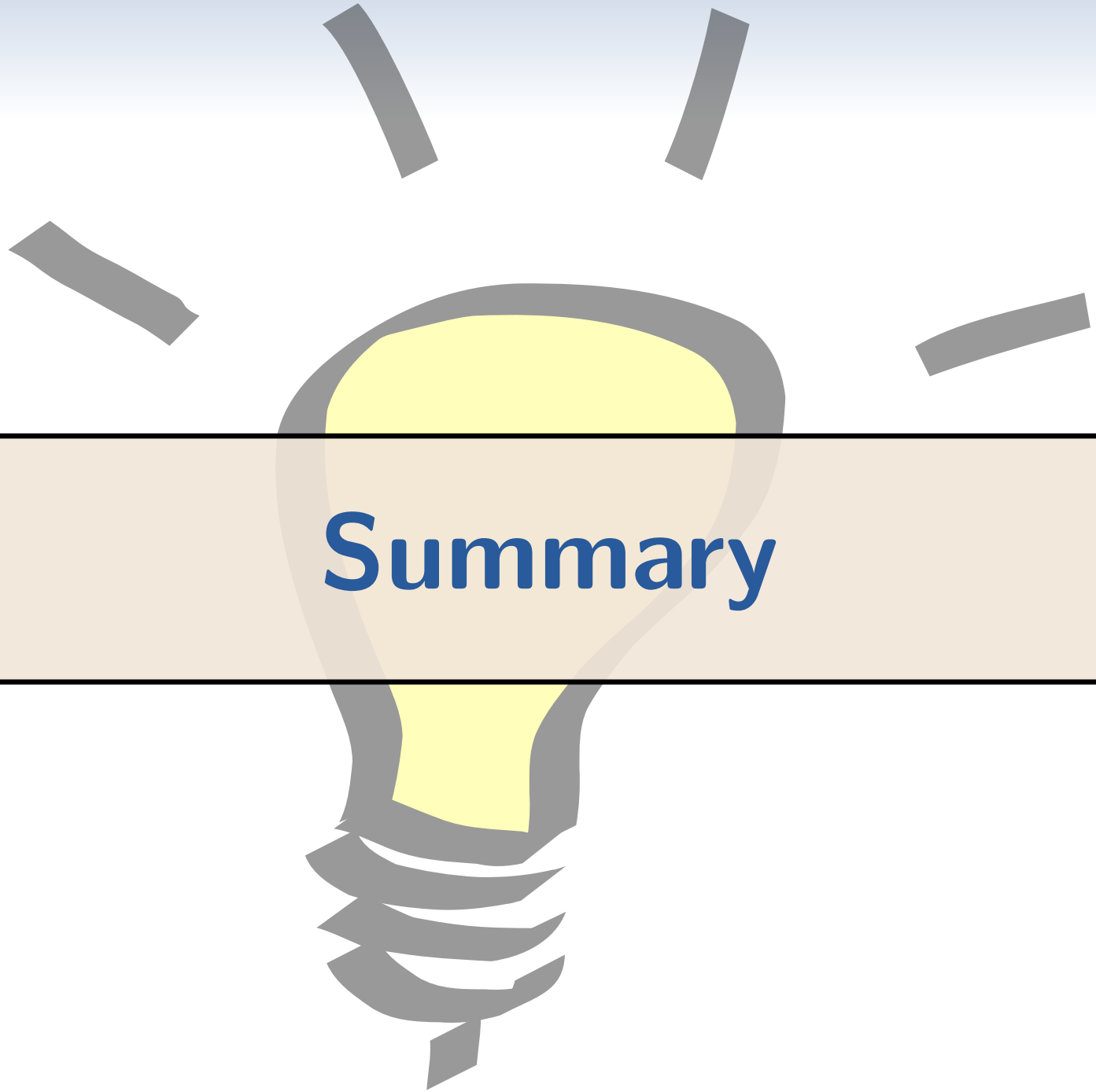
Circuit equivalence testing



Running a SAT solver...

...yields the assignment $\{a \mapsto \mathbf{tt}, b \mapsto \mathbf{tt}, c \mapsto \mathbf{tt}, d \mapsto \mathbf{tt}, g_1 \mapsto \mathbf{tt}, g_2 \mapsto \mathbf{ff}, g_3 \mapsto \mathbf{ff}, g_4 \mapsto \mathbf{ff}, o \mapsto \mathbf{tt}, g_1' \mapsto \mathbf{tt}, g_2' \mapsto \mathbf{ff}, g_3' \mapsto \mathbf{tt}, g_4' \mapsto \mathbf{tt}, o' \mapsto \mathbf{ff}\}$.
Thus, the circuits are actually not equivalent and differ on the input $\{a \mapsto \mathbf{tt}, b \mapsto \mathbf{tt}, c \mapsto \mathbf{tt}, d \mapsto \mathbf{tt}\}$.

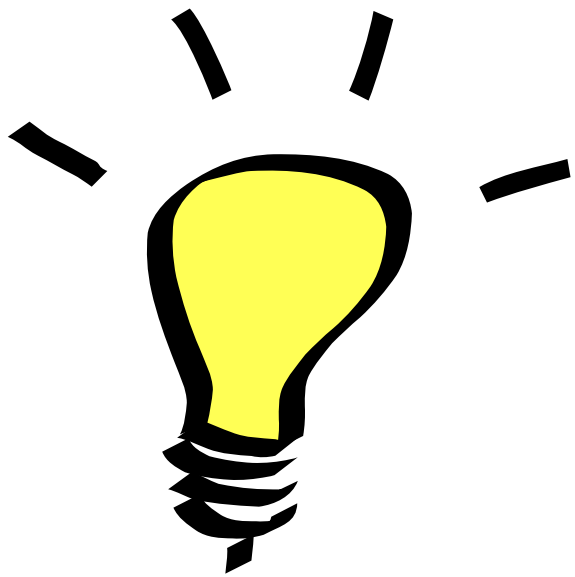
(Interactive)



Summary

Summary / List of Concepts

- Bounded Model Checking
- Encoding BMC in a Satisfiability problem
- Basic ideas of search-based satisfiability solving:
  - Branching
  - Unit Propogation
  - The pure literal rule
  - Conflict-driven clause learning
- Some theory on the limit of SAT solving
- Circuit encoding into SAT



References I