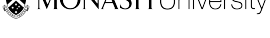
# From DPLL to CDCL

*How SAT solvers learn from conflicts*

---

Alexey Ignatiev

ACP Summer School 2026



# Realm of automated problem solving


SAT is the engines' engine
At the heart of (almost) every automated reasoning technology
N
W
E
S
SAT
The Core Engine
Resolution
Unit Propagation (UP)
Unique Implication Point (UIP)
Clause Learning
Proof Logging
Inprocessing
CDCL
The Main Power
Logics & Theorem Proving
Propositional Theorem Proving
SMT Solving
Modal, Temporal & Description Logics
Higher-Order Reasoning
Satisfiability Modulo Theories (SMT)
Theory Combines
EUF / LIA / NRA
Arrays & Sequences
Bit-Vectors
Datatypes
Knowledge & AI
Planning
Abduction & Diagnosis
Planning as SAT
Robotics
Game Playing
Search & CSP
Constraint Satisfaction
Graph Coloring
Scheduling
Resource Allocation
Timetabling
Configuration
Lazy Clause Generation (LCG)
Constraint Propagation
Explanation Generation
Conflict Analysis
Domain-Specific LCG
Optimization
Pseudo-Boolean Optimization
MaxSAT
Weighted MaxSAT
Partial MaxSAT
Core-Guided MaxSAT
Cardinality & PB Constraints
At-Most-One
At-Least-One
General PB
MaxSAT & PB Optimization
Cost-Function Optimization
Multi-Objective Optimization
Data & Analytics
Data Mining
ML & AI Applications
Feature Selection
Structure Learning
Answer Set Programming (ASP)
Stable Model Semantics
Grounding
Model Generation
Conflict-Driven ASP
Verification
Model Checking
Bounded Model Checking
Property Directed Reachability
Software Verification
Hardware Verification
Programming & Compilers
Program Analysis
Optimizing Compilers
Register Allocation
Dead Code Elimination
Loop Transformations
Mixed Integer Programming (MIP)
MIP via SAT
Cutting Planes
Branch-and-Cut
Indicator Constraints
How to read the map
SAT empowers the landscape in many ways.
Built on SAT
(uses SAT as a component)
Powered by SAT
(relies on SAT solving as the main engine)
Optimized by SAT
(improved via SAT techniques)
Shared Foundations
(technologies that both use and extend SAT in different ways)
Emerging Frontiers
(active research areas)
Many paths.
One engine at the heart.
Countless possibilities.
SAT lies at the heart, providing the fundamental reasoning engine.
These technologies either build on it, are powered by it, or are optimized by it.

# What is SAT?

# Propositional logic

- **propositions:** [BHvMW09]
  - **variables:** $w, x, y, z, a, b, c, \ldots$

# Propositional logic

- **propositions:** [BHvMW09]
  - **variables:** $w, x, y, z, a, b, c, \ldots$
  - **literals:** $w, \bar{x}, \bar{y}, a, \ldots$, but also $\neg w, \neg y, \ldots$

# Propositional logic

- **propositions:** [BHvMW09]
  - **variables:** $w, x, y, z, a, b, c, \ldots$
  - **literals:** $w, \bar{x}, \bar{y}, a, \ldots$, but also $\neg w, \neg y, \ldots$
- **logical connectives:** $\neg, \wedge, \vee, \rightarrow, \leftrightarrow, \ldots$

# Propositional logic

- **propositions:** [BHvMW09]
  - **variables:** $w, x, y, z, a, b, c, \ldots$
  - **literals:** $w, \bar{x}, \bar{y}, a, \ldots$, but also $\neg w, \neg y, \ldots$
- **logical connectives:** $\neg, \bigwedge, \bigvee, \rightarrow, \leftrightarrow, \ldots$

**Example**

$r \triangleq$ *"it is raining"*

$s \triangleq$ *"it is sunny"*

$h \triangleq$ *"time to do hiking"*

$g \triangleq$ *"time to play computer games"*

# Propositional logic

- **propositions:** [BHvMW09]
  - **variables:** $w, x, y, z, a, b, c, \ldots$
  - **literals:** $w, \bar{x}, \bar{y}, a, \ldots$, but also $\neg w, \neg y, \ldots$
- **logical connectives:** $\neg, \wedge, \vee, \rightarrow, \leftrightarrow, \ldots$

**Example**

| | | |
|---|---|---|
| $r \triangleq$ *"it is raining"* | $\neg(s \wedge r)$ | *can't be sunny **and** raining* |
| $s \triangleq$ *"it is sunny"* | $s \rightarrow h$ | *if it's sunny, do hiking!* |
| $h \triangleq$ *"time to do hiking"* | $r \rightarrow g$ | *if it's raining, play games!* |
| $g \triangleq$ *"time to play computer games"* | $\neg(h \wedge g)$ | *don't do hiking **and** gaming* |

# Formulas in conjunctive normal form

- **clause:**
  - **disjunction** of literals, e.g. $(\neg x \vee y \vee z)$

## Formulas in conjunctive normal form

- **clause:**
  - **disjunction** of literals, e.g. $(\neg x \vee y \vee z)$
- **CNF formula:**
  - **conjunction** of clauses, e.g. $(\neg x \vee y) \wedge (\neg y \vee z)$

# Formulas in conjunctive normal form

- **clause:**
  - **disjunction** of literals, e.g. $(\neg x \vee y \vee z)$
- **CNF formula:**
  - **conjunction** of clauses, e.g. $(\neg x \vee y) \wedge (\neg y \vee z)$

**Example**

| | | |
|---|---|---|
| $r \triangleq$ *"it is raining"* | $\neg(s \wedge r)$ | *can't be sunny **and** raining* |
| $s \triangleq$ *"it is sunny"* | $s \rightarrow h$ | *if it's sunny, do hiking!* |
| $h \triangleq$ *"time to do hiking"* | $r \rightarrow g$ | *if it's raining, play games!* |
| $g \triangleq$ *"time to play computer games"* | $\neg(h \wedge g)$ | *don't do hiking **and** gaming* |

# Formulas in conjunctive normal form

- **clause:**
  - **disjunction** of literals, e.g. $(\neg x \vee y \vee z)$
- **CNF formula:**
  - **conjunction** of clauses, e.g. $(\neg x \vee y) \wedge (\neg y \vee z)$

**Example**

| | | |
|---|---|---|
| $r \triangleq$ *"it is raining"* | $\neg(s \wedge r)$ | *can't be sunny **and** raining* |
| $s \triangleq$ *"it is sunny"* | $s \to h$ | *if it's sunny, do hiking!* |
| $h \triangleq$ *"time to do hiking"* | $r \to g$ | *if it's raining, play games!* |
| $g \triangleq$ *"time to play computer games"* | $\neg(h \wedge g)$ | *don't do hiking **and** gaming* |

$$\mathcal{F} = (\neg s \vee \neg r) \wedge (\neg s \vee h) \wedge (\neg r \vee g) \wedge (\neg h \vee \neg g)$$

## Satisfying assignments

- **model** (satisfying assignment) is a **mapping** $\mu$ from variables to $\{0, 1\}$

## Satisfying assignments

- **model** (satisfying assignment) is a **mapping** $\mu$ from variables to $\{0, 1\}$
- models can be **extended** to literals:

$$\mu(x) = 1 \Leftrightarrow \mu(\neg x) = 0$$
$$\mu(x) = 0 \Leftrightarrow \mu(\neg x) = 1$$

## Satisfying assignments

- **model** (satisfying assignment) is a **mapping** $\mu$ from variables to $\{0, 1\}$
- models can be **extended** to literals:

$$\mu(x) = 1 \Leftrightarrow \mu(\neg x) = 0$$
$$\mu(x) = 0 \Leftrightarrow \mu(\neg x) = 1$$

- **safisfied** clause: $\mu(l_1 \vee \ldots \vee l_k) = 1 \Leftrightarrow \exists_{l_i}\, \mu(l_i) = 1$

## Satisfying assignments

- **model** (satisfying assignment) is a **mapping** $\mu$ from variables to $\{0, 1\}$
- models can be **extended** to literals:

$$\mu(x) = 1 \Leftrightarrow \mu(\neg x) = 0$$
$$\mu(x) = 0 \Leftrightarrow \mu(\neg x) = 1$$

- **safisfied** clause: $\mu(l_1 \vee \ldots \vee l_k) = 1 \Leftrightarrow \exists_{l_i}\, \mu(l_i) = 1$
- **satisfied** CNF: $\mu(c_1 \wedge \ldots \wedge c_m) = 1 \Leftrightarrow \forall_{c_j}\, \mu(c_j) = 1$

## Satisfying assignments

- **model** (satisfying assignment) is a **mapping** $\mu$ from variables to $\{0, 1\}$
- models can be **extended** to literals:

$$\mu(x) = 1 \Leftrightarrow \mu(\neg x) = 0$$
$$\mu(x) = 0 \Leftrightarrow \mu(\neg x) = 1$$

- **safisfied** clause: $\mu(l_1 \vee \ldots \vee l_k) = 1 \Leftrightarrow \exists_{l_i}\, \mu(l_i) = 1$
- **satisfied** CNF: $\mu(c_1 \wedge \ldots \wedge c_m) = 1 \Leftrightarrow \forall_{c_j}\, \mu(c_j) = 1$

**Example** $\mathcal{F} = (\neg s \vee \neg r) \wedge (\neg s \vee h) \wedge (\neg r \vee g) \wedge (\neg h \vee \neg g)$

## Satisfying assignments

- **model** (satisfying assignment) is a **mapping** $\mu$ from variables to $\{0, 1\}$
- models can be **extended** to literals:

$$\mu(x) = 1 \Leftrightarrow \mu(\neg x) = 0$$
$$\mu(x) = 0 \Leftrightarrow \mu(\neg x) = 1$$

- **safisfied** clause: $\mu(l_1 \vee \ldots \vee l_k) = 1 \Leftrightarrow \exists_{l_i}\, \mu(l_i) = 1$
- **satisfied** CNF: $\mu(c_1 \wedge \ldots \wedge c_m) = 1 \Leftrightarrow \forall_{c_j}\, \mu(c_j) = 1$

**Example** $\mathcal{F} = (\neg s \vee \neg r) \wedge (\neg s \vee h) \wedge (\neg r \vee g) \wedge (\neg h \vee \neg g)$

$$\mu_1 = \{r = 1, s = 1, h = 1, g = 1\}$$

## Satisfying assignments

- **model** (satisfying assignment) is a **mapping** $\mu$ from variables to $\{0, 1\}$
- models can be **extended** to literals:

$$\mu(x) = 1 \Leftrightarrow \mu(\neg x) = 0$$
$$\mu(x) = 0 \Leftrightarrow \mu(\neg x) = 1$$

- **safisfied** clause: $\mu(l_1 \vee \ldots \vee l_k) = 1 \Leftrightarrow \exists_{l_i}\, \mu(l_i) = 1$
- **satisfied** CNF: $\mu(c_1 \wedge \ldots \wedge c_m) = 1 \Leftrightarrow \forall_{c_j}\, \mu(c_j) = 1$

**Example** $\mathcal{F} = (\neg s \vee \neg r) \wedge (\neg s \vee h) \wedge (\neg r \vee g) \wedge (\neg h \vee \neg g)$

$$\mu_1 = \{r = 1, s = 1, h = 1, g = 1\} \quad \Rightarrow \quad \mu_1(\mathcal{F}) = 0$$

## Satisfying assignments

- **model** (satisfying assignment) is a **mapping** $\mu$ from variables to $\{0, 1\}$
- models can be **extended** to literals:

$$\mu(x) = 1 \Leftrightarrow \mu(\neg x) = 0$$
$$\mu(x) = 0 \Leftrightarrow \mu(\neg x) = 1$$

- **safisfied** clause: $\mu(l_1 \vee \ldots \vee l_k) = 1 \Leftrightarrow \exists_{l_i}\, \mu(l_i) = 1$
- **satisfied** CNF: $\mu(c_1 \wedge \ldots \wedge c_m) = 1 \Leftrightarrow \forall_{c_j}\, \mu(c_j) = 1$

**Example** $\mathcal{F} = (\neg s \vee \neg r) \wedge (\neg s \vee h) \wedge (\neg r \vee g) \wedge (\neg h \vee \neg g)$

$$\mu_1 = \{r = 1, s = 1, h = 1, g = 1\} \quad \Rightarrow \quad \mu_1(\mathcal{F}) = 0$$
$$\mu_2 = \{r = 1, s = 0, h = 0, g = 1\}$$

## Satisfying assignments

- **model** (satisfying assignment) is a **mapping** $\mu$ from variables to $\{0, 1\}$
- models can be **extended** to literals:

$$\mu(x) = 1 \Leftrightarrow \mu(\neg x) = 0$$
$$\mu(x) = 0 \Leftrightarrow \mu(\neg x) = 1$$

- **safisfied** clause: $\mu(l_1 \vee \ldots \vee l_k) = 1 \Leftrightarrow \exists_{l_i}\, \mu(l_i) = 1$
- **satisfied** CNF: $\mu(c_1 \wedge \ldots \wedge c_m) = 1 \Leftrightarrow \forall_{c_j}\, \mu(c_j) = 1$

**Example** $\mathcal{F} = (\neg s \vee \neg r) \wedge (\neg s \vee h) \wedge (\neg r \vee g) \wedge (\neg h \vee \neg g)$

$$\mu_1 = \{r = 1, s = 1, h = 1, g = 1\} \quad \Rightarrow \quad \mu_1(\mathcal{F}) = 0$$
$$\mu_2 = \{r = 1, s = 0, h = 0, g = 1\} \quad \Rightarrow \quad \mu_2(\mathcal{F}) = 1$$

## Satisfying assignments

- **model** (satisfying assignment) is a **mapping** $\mu$ from variables to $\{0, 1\}$
- models can be **extended** to literals:

$$\mu(x) = 1 \Leftrightarrow \mu(\neg x) = 0$$
$$\mu(x) = 0 \Leftrightarrow \mu(\neg x) = 1$$

- **safisfied** clause: $\mu(l_1 \vee \ldots \vee l_k) = 1 \Leftrightarrow \exists_{l_i}\, \mu(l_i) = 1$
- **satisfied** CNF: $\mu(c_1 \wedge \ldots \wedge c_m) = 1 \Leftrightarrow \forall_{c_j}\, \mu(c_j) = 1$

**Example** $\mathcal{F} = (\neg s \vee \neg r) \wedge (\neg s \vee h) \wedge (\neg r \vee g) \wedge (\neg h \vee \neg g)$

$$\mu_1 = \{r = 1, s = 1, h = 1, g = 1\} \quad \Rightarrow \quad \mu_1(\mathcal{F}) = 0$$
$$\mu_2 = \{r = 1, s = 0, h = 0, g = 1\} \quad \Rightarrow \quad \mu_2(\mathcal{F}) = 1$$
$$\mu_3 = \{r = 0, s = 1, h = 1, g = 0\}$$

## Satisfying assignments

- **model** (satisfying assignment) is a **mapping** $\mu$ from variables to $\{0, 1\}$
- models can be **extended** to literals:

$$\mu(x) = 1 \Leftrightarrow \mu(\neg x) = 0$$
$$\mu(x) = 0 \Leftrightarrow \mu(\neg x) = 1$$

- **safisfied** clause: $\mu(l_1 \vee \ldots \vee l_k) = 1 \Leftrightarrow \exists_{l_i}\, \mu(l_i) = 1$
- **satisfied** CNF: $\mu(c_1 \wedge \ldots \wedge c_m) = 1 \Leftrightarrow \forall_{c_j}\, \mu(c_j) = 1$

**Example** $\mathcal{F} = (\neg s \vee \neg r) \wedge (\neg s \vee h) \wedge (\neg r \vee g) \wedge (\neg h \vee \neg g)$

$$\mu_1 = \{r = 1, s = 1, h = 1, g = 1\} \Rightarrow \mu_1(\mathcal{F}) = 0$$
$$\mu_2 = \{r = 1, s = 0, h = 0, g = 1\} \Rightarrow \mu_2(\mathcal{F}) = 1$$
$$\mu_3 = \{r = 0, s = 1, h = 1, g = 0\} \Rightarrow \mu_3(\mathcal{F}) = 1$$

## SAT is the decision problem for propositional logic

given a ***propositional*** formula in **CNF**,

## SAT is the decision problem for propositional logic

given a ***propositional*** formula in **CNF**,

decide whether or not it is **satisfiable**

## SAT is the decision problem for propositional logic

given a ***propositional*** formula in **CNF**,

decide whether or not it is **satisfiable**

**Example**

$\mathcal{F}_1 = (x_1) \wedge (\neg x_1 \vee \neg x_2)$

given a ***propositional*** formula in **CNF**,

decide whether or not it is **satisfiable**

**Example**

$\mathcal{F}_1 = (x_1) \wedge (\neg x_1 \vee \neg x_2)$ — satisfiable, e.g. $\mu = \{x_1 = 1, x_2 = 0\}$

given a ***propositional*** formula in **CNF**,

decide whether or not it is **satisfiable**

**Example**

$\mathcal{F}_1 = (x_1) \wedge (\neg x_1 \vee \neg x_2)$ — satisfiable, e.g. $\mu = \{x_1 = 1, x_2 = 0\}$

$\mathcal{F}_2 = (x_1) \wedge (\neg x_1 \vee \neg x_2) \wedge (x_2)$

given a ***propositional*** formula in **CNF**,

decide whether or not it is **satisfiable**

**Example**

$\mathcal{F}_1 = (x_1) \wedge (\neg x_1 \vee \neg x_2)$ — satisfiable, e.g. $\mu = \{x_1 = 1, x_2 = 0\}$

$\mathcal{F}_2 = (x_1) \wedge (\neg x_1 \vee \neg x_2) \wedge (x_2)$ — unsatisfiable

# SAT is hard

# SAT is $\mathbb{NP}$-complete

[Coo71, Kar72]

$$\mathrm{SAT} \in \mathbb{NP}$$

$$\forall_{A \in \mathbb{NP}} \quad A \propto \mathrm{SAT}$$

## SAT is $\mathbb{NP}$-complete

[Coo71, Kar72]

$$\text{SAT} \in \mathbb{NP}$$

$$\forall_{A \in \mathbb{NP}} \quad A \propto \text{SAT}$$



SAT is **at least as hard as** any other problem in $\mathbb{NP}$

## SAT is $\mathbb{NP}$-complete

[Coo71, Kar72]

$$\text{SAT} \in \mathbb{NP}$$

$$\forall_{A \in \mathbb{NP}} \quad A \propto \text{SAT}$$



SAT is **at least as hard as** any other problem in $\mathbb{NP}$



**so why bother?**

[Coo71, Kar72]

$$\text{SAT} \in \mathbb{NP}$$

$$\forall_{A \in \mathbb{NP}} \quad A \propto \text{SAT}$$



SAT is **at least as hard as** any other problem in $\mathbb{NP}$



**so why bother?** — $\mathbb{NP}$**-complete** = **"highly applicable"**!

\+ **SAT is not hard** ***"on average"***

## Grasping the search space. . .

- Number of seconds since the Big Bang: $\approx 10^{17}$

## Grasping the search space. . .

- Number of seconds since the Big Bang: $\approx 10^{17}$
- Number of fundamental particles in observable universe: $\approx 10^{80}$–$10^{85}$

## Grasping the search space. . .

- Number of seconds since the Big Bang: $\approx 10^{17}$
- Number of fundamental particles in observable universe: $\approx 10^{80}$–$10^{85}$
- Search space with **15775** propositional variables (worst case):
  - number of assignments: $> \mathbf{10^{4748}!}$
  - (SAT solvers **in late 90s** – but formula dependent)

# Grasping the search space. . .

- Number of seconds since the Big Bang: $\approx 10^{17}$
- Number of fundamental particles in observable universe: $\approx 10^{80}$–$10^{85}$
- Search space with **15775** propositional variables (worst case):
  - number of assignments: $> \mathbf{10^{4748}!}$
  - (SAT solvers **in late 90s** – but formula dependent)
- Search space with **2832875** propositional variables (worst case):
  - number of assignments: $\gg \mathbf{10^{840000}!!}$
  - (SAT solvers **at present** – but formula dependent)

## CDCL SAT disruption

- **CDCL SAT solving** is a success story of Computer Science

## CDCL SAT disruption

- **CDCL SAT solving** is a success story of Computer Science
  - **Conflict-Driven Clause Learning** (CDCL) [MSS96b, MSS99, ZMMM01, MMZ+01]

## CDCL SAT disruption

- **CDCL SAT solving** is a success story of Computer Science
  - **Conflict-Driven Clause Learning** (CDCL) [MSS96b, MSS99, ZMMM01, MMZ+01]
  - CDCL SAT has impacted **many** different fields

# CDCL SAT disruption

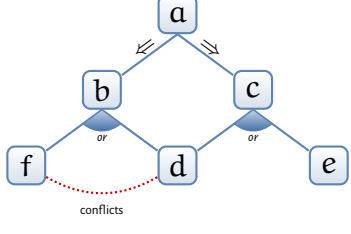
- **CDCL SAT solving** is a success story of Computer Science
  - **Conflict-Driven Clause Learning** (CDCL) [MSS96b, MSS99, ZMMM01, MMZ+01]
  - CDCL SAT has impacted **many** different fields
  - Hundreds (thousands?) of practical applications



# Package Dependency as reducibility example

can single package $\alpha$ be installed in repository $\mathcal{R}$? [DDL+06]

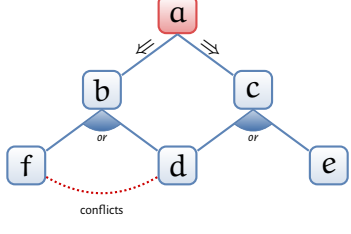
**Example**



# Package Dependency as reducibility example

can single package $a$ be installed in repository $R$? [DDL+06]

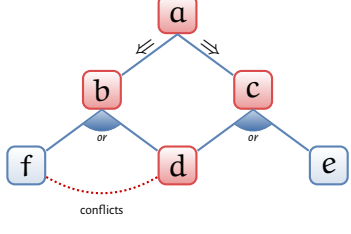
**Example**



# Package Dependency as reducibility example

can single package $\alpha$ be installed in repository $\mathcal{R}$? [DDL+06]

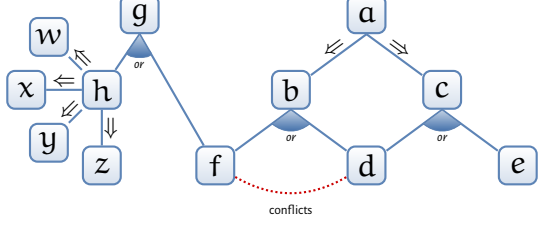
**Example**



# Package Dependency as reducibility example

can single package $\alpha$ be installed in repository $\mathcal{R}$? [DDL+06]

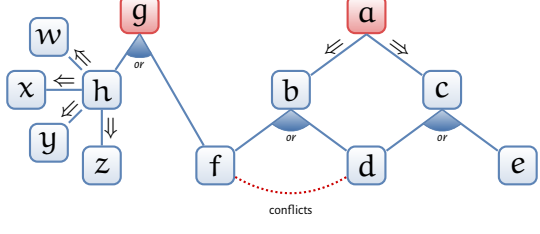
**Example**



# Package Dependency as reducibility example

can single package $\alpha$ be installed in repository $\mathcal{R}$? [DDL+06]

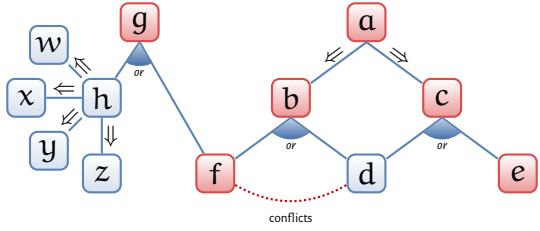
**Example**



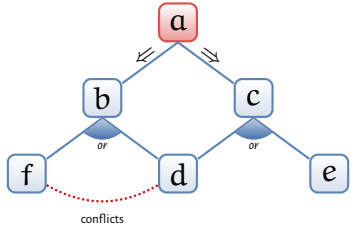
# Package Dependency as reducibility example

can single package $\alpha$ be installed in repository $\mathcal{R}$? [DDL+06]

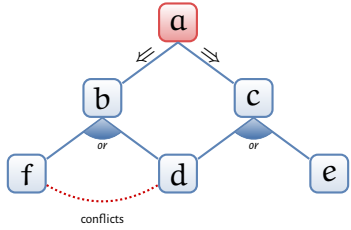
**Example**



# How to solve it with SAT?



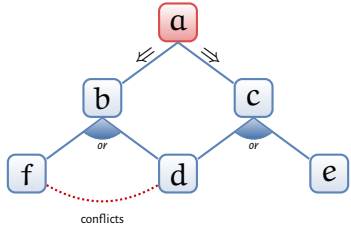
# How to solve it with SAT?




**SAT formulation:**

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

# How to solve it with SAT?




**SAT formulation:**

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

$$\mu = \{a = 1, b = 1, c = 1, d = 1, f = 0, e = 0\}$$

# SAT in practice

# DIMACS CNF format

- **preamble**: `p cnf N M` — $\mathbf{N}$ and $\mathbf{M}$ are numbers of **variables** and **clauses**

# DIMACS CNF format

- **preamble**: `p cnf N M` — $\mathbf{N}$ and $\mathbf{M}$ are numbers of **variables** and **clauses**
- **variable**: $\eta(v) \in \mathbb{N}$ — **unique integer index** per variable

# DIMACS CNF format

- **preamble**: `p cnf N M` — $\mathbf{N}$ and $\mathbf{M}$ are numbers of **variables** and **clauses**
- **variable**: $\eta(v) \in \mathbb{N}$ — **unique integer index** per variable
- **literal**: $\neg v$ is represented as -$\eta(v)$

# DIMACS CNF format

- **preamble**: `p cnf N M` — $\mathbf{N}$ and $\mathbf{M}$ are numbers of **variables** and **clauses**
- **variable**: $\eta(v) \in \mathbb{N}$ — **unique integer index** per variable
- **literal**: $\neg v$ is represented as $-\eta(v)$
- **clause**: $0$-terminated list of literals

# DIMACS CNF format

- **preamble**: `p cnf N M` — $N$ and $M$ are numbers of **variables** and **clauses**
- **variable**: $\eta(v) \in \mathbb{N}$ — **unique integer index** per variable
- **literal**: $\neg v$ is represented as $-\eta(v)$
- **clause**: 0-terminated list of literals

**Example**

$$
\begin{aligned}
&(\neg a \vee b)\\
\wedge&(\neg a \vee c)\\
\wedge&(\neg b \vee f \vee d)\\
\wedge&(\neg c \vee d \vee e)\\
\wedge&(\neg f \vee \neg d)\\
\wedge&(a)
\end{aligned}
$$

# DIMACS CNF format

- **preamble**: `p cnf N M` — $\mathbf{N}$ and $\mathbf{M}$ are numbers of **variables** and **clauses**
- **variable**: $\eta(v) \in \mathbb{N}$ — **unique integer index** per variable
- **literal**: $\neg v$ is represented as -$\eta(v)$
- **clause**: 0-terminated list of literals

**Example**

$$
\begin{aligned}
(\neg a \vee b) \\
\wedge(\neg a \vee c) \\
\wedge(\neg b \vee f \vee d) \\
\wedge(\neg c \vee d \vee e) \\
\wedge(\neg f \vee \neg d) \\
\wedge(a)
\end{aligned}
\qquad
\begin{aligned}
a \triangleq 1, \neg a \triangleq -1 \\
b \triangleq 2, \neg b \triangleq -2 \\
c \triangleq 3, \neg c \triangleq -3 \\
d \triangleq 4, \neg d \triangleq -4 \\
e \triangleq 5, \neg e \triangleq -5 \\
f \triangleq 6, \neg f \triangleq -6
\end{aligned}
$$

# DIMACS CNF format

- **preamble**: `p cnf N M` — $N$ and $M$ are numbers of **variables** and **clauses**
- **variable**: $\eta(v) \in \mathbb{N}$ — **unique integer index** per variable
- **literal**: $\neg v$ is represented as -$\eta(v)$
- **clause**: `0`-terminated list of literals

**Example**

$$
\begin{aligned}
&(\neg a \vee b) \\
\wedge&(\neg a \vee c) \\
\wedge&(\neg b \vee f \vee d) \\
\wedge&(\neg c \vee d \vee e) \\
\wedge&(\neg f \vee \neg d) \\
\wedge&(a)
\end{aligned}
\qquad
\begin{aligned}
a &\triangleq 1, \neg a \triangleq -1 \\
b &\triangleq 2, \neg b \triangleq -2 \\
c &\triangleq 3, \neg c \triangleq -3 \\
d &\triangleq 4, \neg d \triangleq -4 \\
e &\triangleq 5, \neg e \triangleq -5 \\
f &\triangleq 6, \neg f \triangleq -6
\end{aligned}
$$

```
p cnf 6 6
-1 2 0
-1 3 0
-2 6 4 0
-3 4 5 0
-6 -5 0
1 0
```

# PySAT — SAT in Python

- `$ pip install python-sat`

# PySAT — SAT in Python

- `$ pip install python-sat`
- `https://pysathq.github.io/` [IMM18, ITK24]

# PySAT — SAT in Python

- `$ pip install python-sat`
- `https://pysathq.github.io/`

[IMM18, ITK24]

**Example**

```python
from pysat.formula import CNF
from pysat.solvers import Solver

cnf = CNF(from_clauses=[[-1, 2], [-1, 3], [-2, 6, 4], [-3, 4, 5], [-6, 5], [1]])
with Solver(name='minisat22', bootstrap_with=cnf) as solver:
    if solver.solve() == True:
        model = solver.get_model()
        print('model:', model)
    else:
        print('unsatisfiable')
```

# exercise time

# DP and D(P)LL

- **Resolution rule** [DP60, Rob65]

$$\frac{(\alpha \vee x) \qquad (\beta \vee \bar{x})}{(\alpha \vee \beta)}$$

  - $(\alpha \vee \beta)$ is a **resolvent**

- **Resolution rule** [DP60, Rob65]

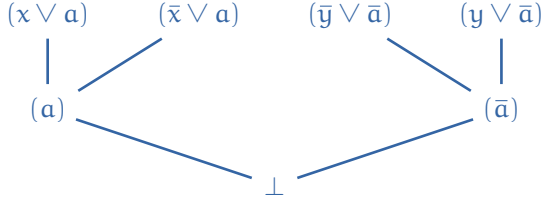
$$\frac{(\alpha \vee x) \qquad (\beta \vee \bar{x})}{(\alpha \vee \beta)}$$

  - $(\alpha \vee \beta)$ is a **resolvent**
  - **Complete proof** system for propositional logic

- **Resolution rule** [DP60, Rob65]

$$\frac{(\alpha \vee x) \qquad (\beta \vee \bar{x})}{(\alpha \vee \beta)}$$

  - $(\alpha \vee \beta)$ is a **resolvent**
  - **Complete proof** system for propositional logic

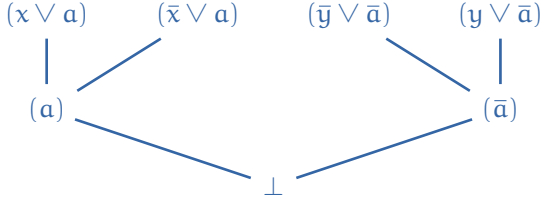



  - $\bot$ is an **empty clause** – **no way** to satisfy!

- **Resolution rule** [DP60, Rob65]

$$\frac{(\alpha \vee x) \qquad (\beta \vee \bar{x})}{(\alpha \vee \beta)}$$

  - $(\alpha \vee \beta)$ is a **resolvent**
  - **Complete proof** system for propositional logic




  - $\bot$ is an **empty clause** – **no way** to satisfy!
  - Extensively used in modern (CDCL) SAT solvers [BHvMW09]

# DP procedure

- **dates back to late 50s:** [DP60]
  - solely **resolution-based**

# DP procedure

- **dates back to late 50s:** [DP60]
  - solely **resolution-based**
  - ***eliminates*** variables **exhaustively**

# DP procedure

- **dates back to late 50s:** [DP60]
  - solely **resolution-based**
  - ***eliminates*** variables **exhaustively**
  - exponential in space

# DP procedure

- **dates back to late 50s:** [DP60]
  - solely **resolution-based**
  - ***eliminates*** variables **exhaustively**
  - exponential in space

```
while True:
    if 𝓕 = ⊤:
        return True  # SATISFIABLE
    if ⊥ ∈ 𝓕:
        return False  # UNSATISFIABLE
    pick remaining variable x
    add all resolvents on x
    remove all clauses with x and ¬x
```

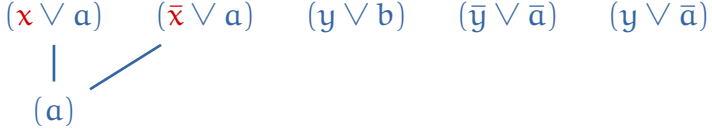
## DP Procedure (example)

$$(x \vee a) \quad (\bar{x} \vee a) \quad (y \vee b) \quad (\bar{y} \vee \bar{a}) \quad (y \vee \bar{a})$$

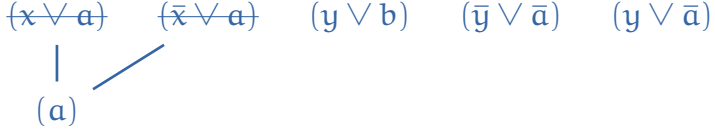
# DP Procedure (example)

$$(x \vee a) \quad (\bar{x} \vee a) \quad (y \vee b) \quad (\bar{y} \vee \bar{a}) \quad (y \vee \bar{a})$$

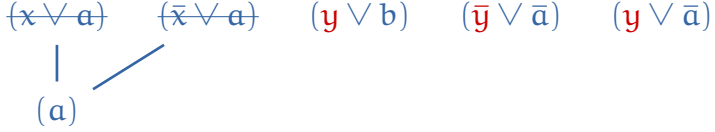
# DP Procedure (example)



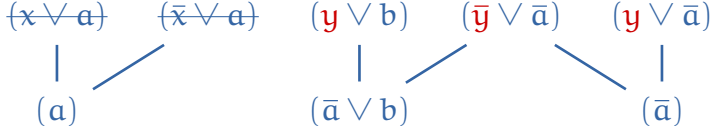
# DP Procedure (example)



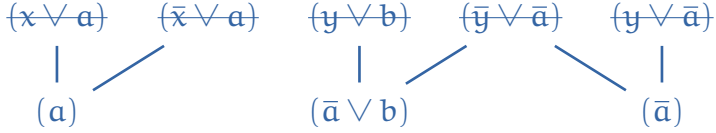
# DP Procedure (example)



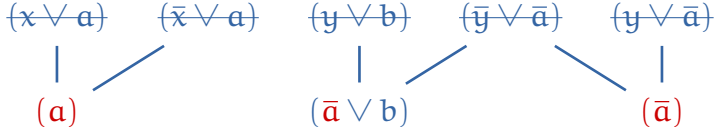
# DP Procedure (example)



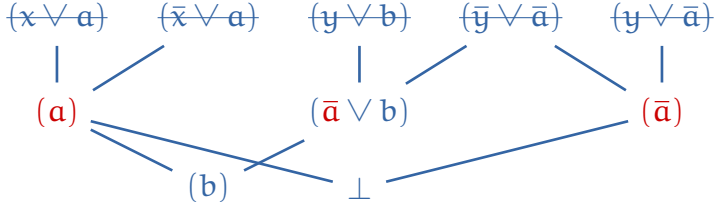
# DP Procedure (example)



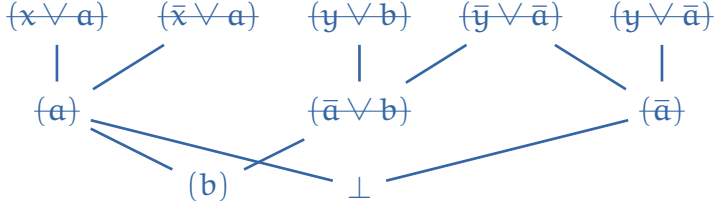
# DP Procedure (example)



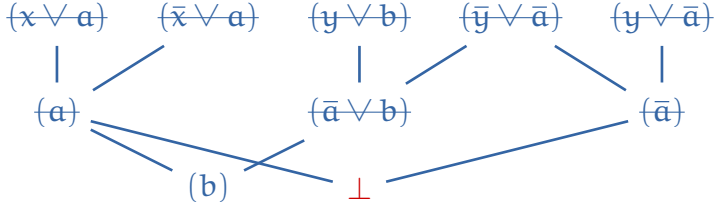
# DP Procedure (example)



# DP Procedure (example)



# DP Procedure (example)



# Unit propagation

$$\begin{aligned}\mathcal{F} \quad = \quad & (r) \wedge (\bar{r} \vee s) \wedge \\ & (\bar{w} \vee a) \wedge (\bar{x} \vee \bar{a} \vee b) \wedge \\ & (\bar{y} \vee \bar{z} \vee c) \wedge (\bar{b} \vee \bar{c} \vee d)\end{aligned}$$

# Unit propagation

$$\begin{aligned}\mathcal{F} \quad = \quad & (r) \wedge (\bar{r} \vee s) \wedge \\ & (\overline{w} \vee a) \wedge (\bar{x} \vee \bar{a} \vee b) \wedge \\ & (\bar{y} \vee \bar{z} \vee c) \wedge (\overline{b} \vee \bar{c} \vee d)\end{aligned}$$

- **Decisions / Variable Branchings:**
  $w = 1, x = 1, y = 1, z = 1$

# Unit propagation

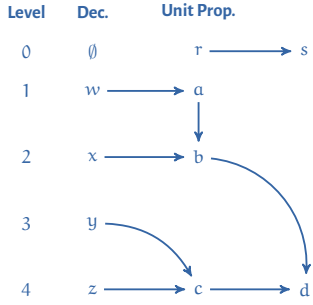
$$\begin{aligned}\mathcal{F} \quad = \quad & (r) \wedge (\bar{r} \vee s) \wedge \\ & (\overline{w} \vee a) \wedge (\bar{x} \vee \bar{a} \vee b) \wedge \\ & (\bar{y} \vee \bar{z} \vee c) \wedge (\overline{b} \vee \bar{c} \vee d)\end{aligned}$$

- **Decisions / Variable Branchings:**
  $w = 1, x = 1, y = 1, z = 1$

  - **Unit clause rule: if clause is unit, its sole literal must be satisfied**

# Unit propagation

$$\mathcal{F} = (r) \wedge (\bar{r} \vee s) \wedge (\overline{w} \vee a) \wedge (\bar{x} \vee \bar{a} \vee b) \wedge (\bar{y} \vee \bar{z} \vee c) \wedge (\bar{b} \vee \bar{c} \vee d)$$

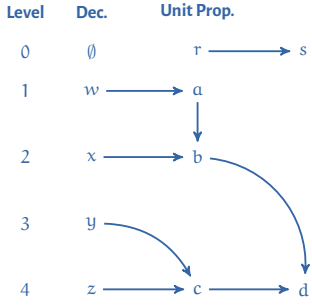



- **Decisions / Variable Branchings:**
  $w = 1, x = 1, y = 1, z = 1$

- **Unit clause rule: if clause is unit, its sole literal must be satisfied**

# Unit propagation

$$\mathcal{F} = (r) \wedge (\bar{r} \vee s) \wedge (\bar{w} \vee a) \wedge (\bar{x} \vee \bar{a} \vee b) \wedge (\bar{y} \vee \bar{z} \vee c) \wedge (\bar{b} \vee \bar{c} \vee d)$$




- **Decisions / Variable Branchings:**
  $w = 1, x = 1, y = 1, z = 1$

- **Unit clause rule: if clause is unit, its sole literal must be satisfied**

- Additional definitions:
  - Antecedent (or reason) of an implied assignment
    - $(\bar{b} \vee \bar{c} \vee d)$ for $d$
  - Associate assignment with decision levels
    - $w = 1 @ 1, x = 1 @ 2, y = 1 @ 3, z = 1 @ 4$
    - $r = 1 @ 0, d = 1 @ 4, \ldots$

# D(P)LL algorithm – simplistic view

- **dates back to 1962:** [DLL62]
  - based on **unit propagation + branching + backtracking**

# D(P)LL algorithm – simplistic view

- **dates back to 1962:** [DLL62]
  - based on **unit propagation + branching + backtracking**
  - **improves** the DP procedure

# D(P)LL algorithm – simplistic view

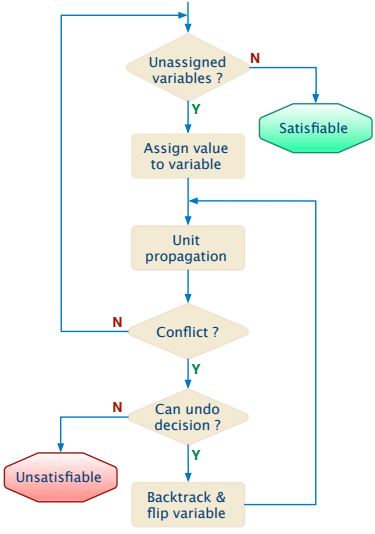
- **dates back to 1962:** [DLL62]
  - based on **unit propagation + branching + backtracking**
  - **improves** the DP procedure
  - exponential in time

# D(P)LL algorithm – simplistic view

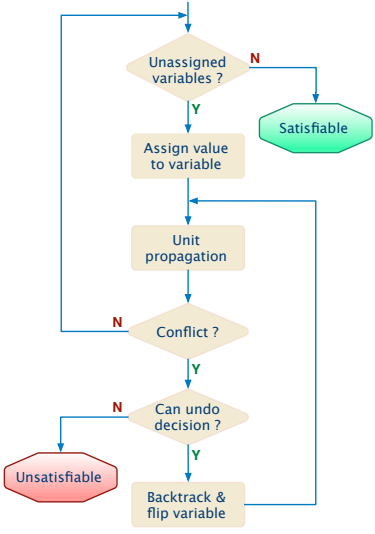
- **dates back to 1962:** [DLL62]
  - based on **unit propagation + branching + backtracking**
  - **improves** the DP procedure
  - exponential in time

```
def DPLL(ℱ):
    ℱ ← UP(ℱ)
    if ℱ = ⊤:
        return True  # SATISFIABLE
    if ⊥ ∈ ℱ:
        return False  # UNSATISFIABLE
    pick remaining variable x and literal l ∈ {x, ¬x}
    if DPLL(ℱ ∧ {l}) = True:
        return True  # SATISFIABLE
    return DPLL(ℱ ∧ {¬l})
```

# D(P)LL algorithm

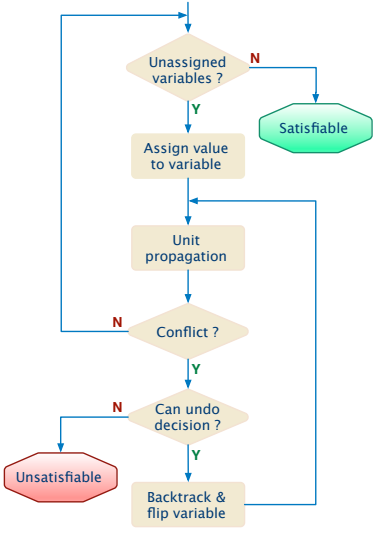


# D(P)LL algorithm



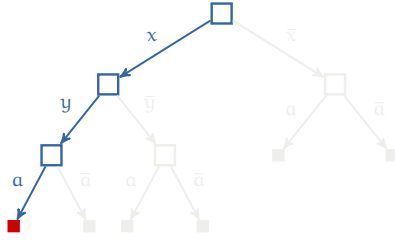

$$\mathcal{F} = (x \vee y) \wedge (a \vee b) \wedge (\bar{a} \vee b) \wedge (a \vee \bar{b}) \wedge (\bar{a} \vee \bar{b})$$

# D(P)LL algorithm

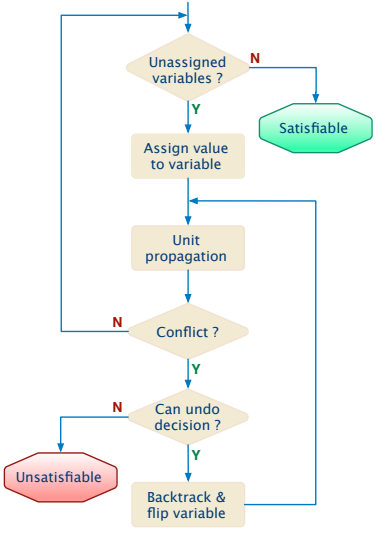



$\mathcal{F} = (x \vee y) \wedge (a \vee b) \wedge (\bar{a} \vee b) \wedge (a \vee \bar{b}) \wedge (\bar{a} \vee \bar{b})$

| Level | Dec. | Unit Prop. |
|---|---|---|
| 0 | $\emptyset$ | |
| 1 | $x$ | |
| 2 | $y$ | |
| 3 | $a$ | $a \to b \to \bot$ |

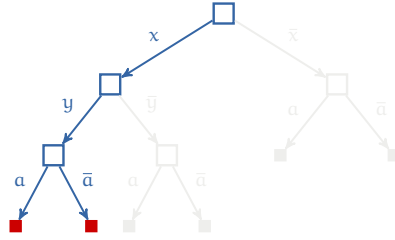


# D(P)LL algorithm

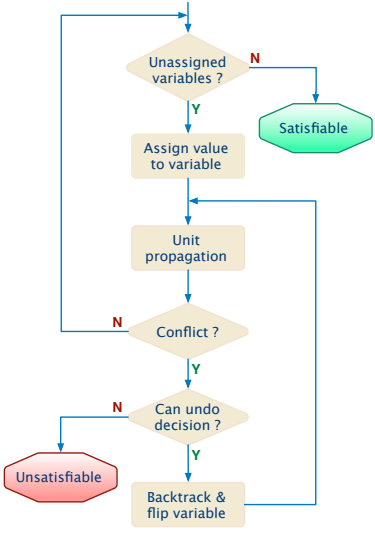



$$\mathcal{F} = (x \vee y) \wedge (a \vee b) \wedge (\bar{a} \vee b) \wedge (a \vee \bar{b}) \wedge (\bar{a} \vee \bar{b})$$

| Level | Dec. | Unit Prop. |
|---|---|---|
| 0 | $\emptyset$ | |
| 1 | $x$ | |
| 2 | $y$ | |
| 3 | $\bar{a}$ | $\rightarrow \bar{b} \rightarrow \bot$ |

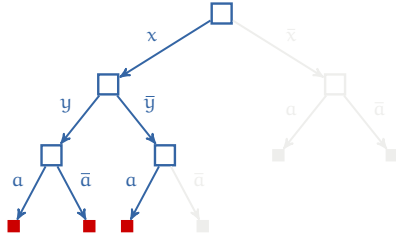


# D(P)LL algorithm

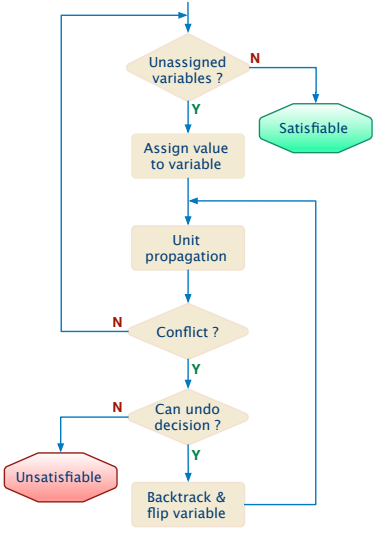



$$\mathcal{F} = (x \vee y) \wedge (a \vee b) \wedge (\bar{a} \vee b) \wedge (a \vee \bar{b}) \wedge (\bar{a} \vee \bar{b})$$

| Level | Dec. | Unit Prop. |
|---|---|---|
| 0 | $\emptyset$ | |
| 1 | $x$ | |
| 2 | $\bar{y}$ | |
| 3 | $a$ | $a \to b \to \bot$ |

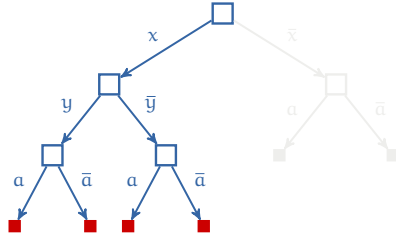


# D(P)LL algorithm

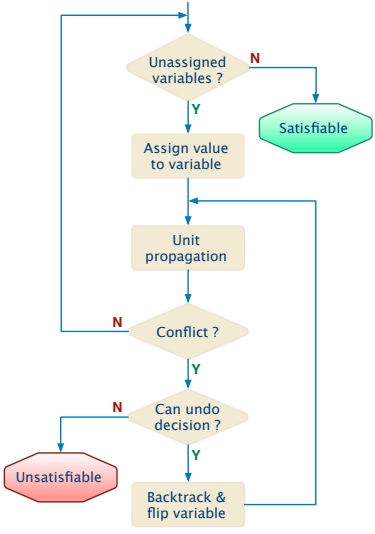



$$\mathcal{F} = (x \vee y) \wedge (a \vee b) \wedge (\bar{a} \vee b) \wedge (a \vee \bar{b}) \wedge (\bar{a} \vee \bar{b})$$

| Level | Dec. | Unit Prop. |
|---|---|---|
| 0 | $\emptyset$ | |
| 1 | $x$ | |
| 2 | $\bar{y}$ | |
| 3 | $\bar{a}$ | $\bar{a} \to \bar{b} \to \bot$ |

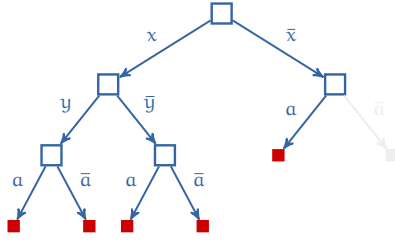


# D(P)LL algorithm

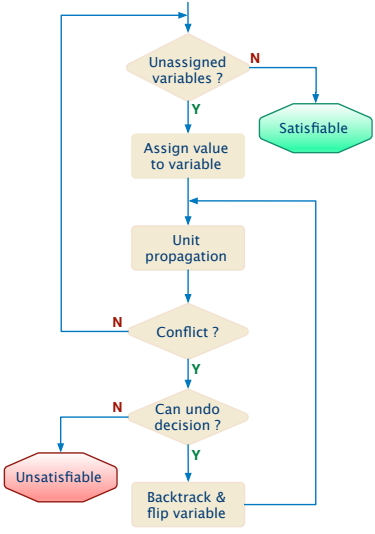



$\mathcal{F} = (x \vee y) \wedge (a \vee b) \wedge (\bar{a} \vee b) \wedge (a \vee \bar{b}) \wedge (\bar{a} \vee \bar{b})$

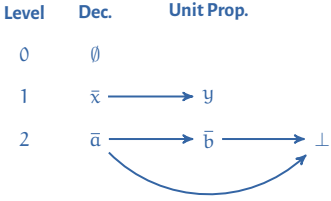
| Level | Dec. | Unit Prop. |
|---|---|---|
| 0 | $\emptyset$ | |
| 1 | $\bar{x}$ | $y$ |
| 2 | $a$ | $b$, $\bot$ |

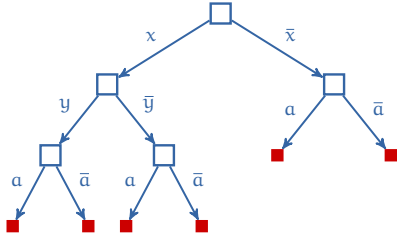


# D(P)LL algorithm




$$\mathcal{F} = (x \vee y) \wedge (a \vee b) \wedge (\bar{a} \vee b) \wedge (a \vee \bar{b}) \wedge (\bar{a} \vee \bar{b})$$



- **dates back to late 50s – early 60s:**
  - **1st version – DP:**
    - solely **resolution-based**
    - ***eliminates*** variables **exhaustively**
    - exponential in space

# DP / D(P)LL algorithm

- **dates back to late 50s – early 60s:**
  - **1st version – DP:**
    - solely **resolution-based**
    - ***eliminates*** variables **exhaustively**
    - exponential in space
  - **2nd version – D(P)LL**
    - based on **unit propagation + branching + backtracking**
    - **improves** the DP procedure
    - exponential in time

# DP / D(P)LL algorithm

- **dates back to late 50s – early 60s:**
  - **1st version – DP:**
    - solely **resolution-based**
    - ***eliminates*** variables **exhaustively**
    - exponential in space
  - **2nd version – D(P)LL**
    - based on **unit propagation + branching + backtracking**
    - **improves** the DP procedure
    - exponential in time
- **. . . D(P)LL gave rise to CDCL in 1995**

# exercise time

# What is CDCL?

# What is a CDCL?

- **Conflict-Driven Clause Learning**

# What is a CDCL?

- **Conflict-Driven Clause Learning**
- Extend **DPLL** SAT solver with: [DP60, DLL62]
  - **Clause learning** & **non-chronological backtracking** [MS95, MSS96b, MSS99]
  - **Search restarts** [GSC97, BMS00, Hua07, Bie08, LOM+18]
  - **Lazy data structures**
  - **Conflict-guided branching**
  - **Heuristic clause minimization** [SB09, Gel09, LLX+17]
  - ...

# What is a CDCL?

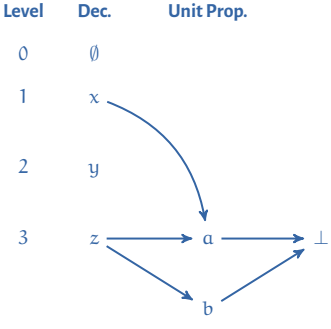
- **Conflict-Driven Clause Learning**
- Extend **DPLL** SAT solver with: [DP60, DLL62]
  - **Clause learning & non-chronological backtracking** [MS95, MSS96b, MSS99]
    - + Exploit UIPs [MS95, MSS99, ZMMM01, SSS12]
    - + Opportunistically delete clauses [MSS96b, MSS99, GN02, AS09]
  - **Search restarts** [GSC97, BMS00, Hua07, Bie08, LOM+18]
  - **Lazy data structures**
    - Watched literals [MMZ+01]
  - **Conflict-guided branching**
    - Lightweight branching heuristics [MMZ+01]
    - Phase saving [PD07]
  - **Heuristic clause minimization** [SB09, Gel09, LLX+17]
  - ...

# What is a CDCL?

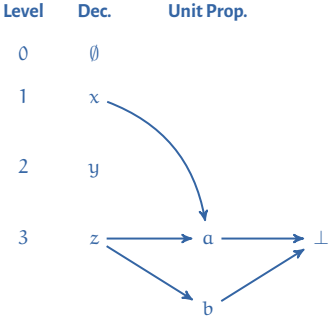
- **Conflict-Driven Clause Learning**
- Extend **DPLL** SAT solver with: [DP60, DLL62]
  - **Clause learning & non-chronological backtracking** [MS95, MSS96b, MSS99]
    - + Exploit UIPs [MS95, MSS99, ZMMM01, SSS12]
    - + Opportunistically delete clauses [MSS96b, MSS99, GN02, AS09]

# Clause learning and backjumping

# Clause learning



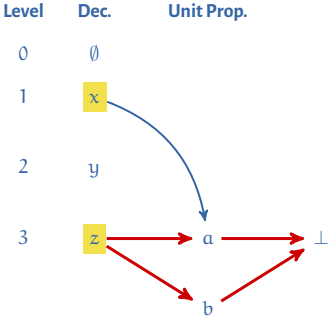
# Clause learning




- **Analyze conflict**

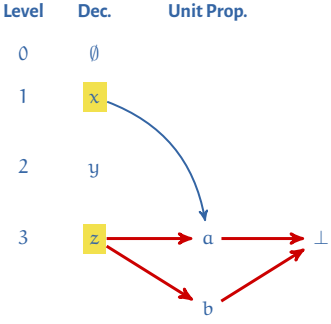
[MS95, MSS96a, MSS96a, MSS96b, MSS99]

# Clause learning



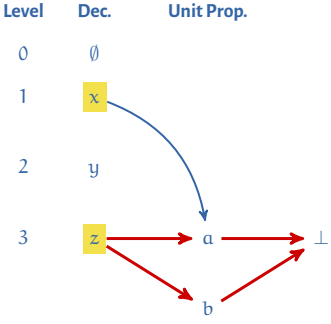

- **Analyze conflict** [MS95, MSS96a, MSS96a, MSS96b, MSS99]
  - **Reasons: $x$ and $z$**
    - Current decision variable & literals assigned at earlier decision levels

# Clause learning




- **Analyze conflict** [MS95, MSS96a, MSS96a, MSS96b, MSS99]
  - **Reasons: $x$ and $z$**
    - Current decision variable & literals assigned at earlier decision levels
  - **Create new clause:** $(\bar{x} \vee \bar{z})$

# Clause learning



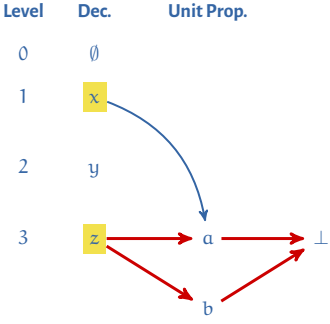

$(\bar{a} \vee \bar{b})$ $(\bar{z} \vee b)$ $(\bar{x} \vee \bar{z} \vee a)$

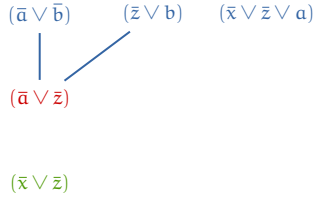
$(\bar{a} \vee \bar{z})$

$(\bar{x} \vee \bar{z})$

- **Analyze conflict** [MS95, MSS96a, MSS96a, MSS96b, MSS99]
  - **Reasons: $x$ and $z$**
    - Current decision variable & literals assigned at earlier decision levels
  - **Create new clause: $(\bar{x} \vee \bar{z})$**
- **Clause learning vs. resolution**

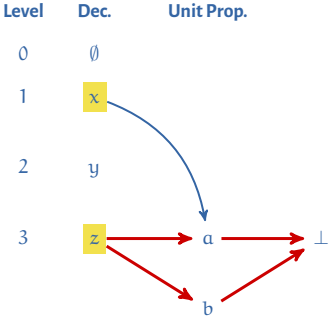
# Clause learning

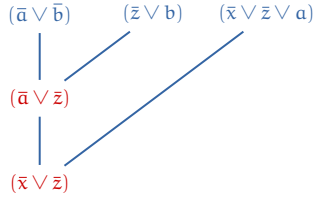



- **Analyze conflict** [MS95, MSS96a, MSS96a, MSS96b, MSS99]
  - **Reasons: $x$ and $z$**
    - Current decision variable & literals assigned at earlier decision levels
  - **Create new clause: $(\bar{x} \vee \bar{z})$**
- **Clause learning vs. resolution**

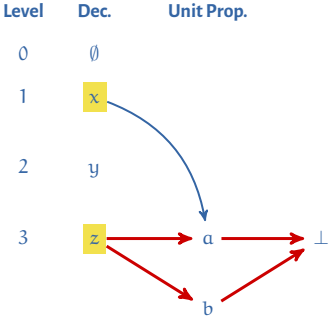
# Clause learning

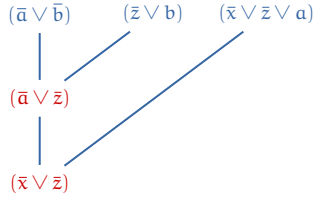



- **Analyze conflict** [MS95, MSS96a, MSS96a, MSS96b, MSS99]
  - **Reasons: $x$ and $z$**
    - Current decision variable & literals assigned at earlier decision levels
  - **Create new clause: $(\bar{x} \vee \bar{z})$**
- **Clause learning vs. resolution**

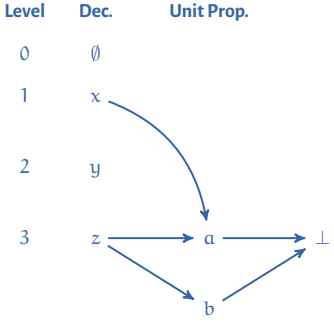
# Clause learning



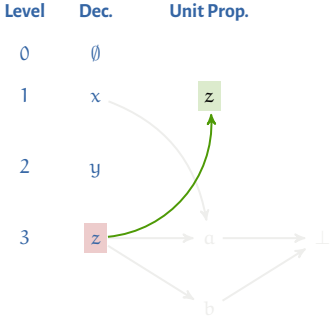




- **Analyze conflict** [MS95, MSS96a, MSS96a, MSS96b, MSS99]
  - **Reasons: $x$ and $z$**
    - Current decision variable & literals assigned at earlier decision levels
  - **Create new clause: $(\bar{x} \vee \bar{z})$**
- **Clause learning vs. resolution**
  - **Learned clauses result from (selected) resolution operations**

# Clause learning + backtracking

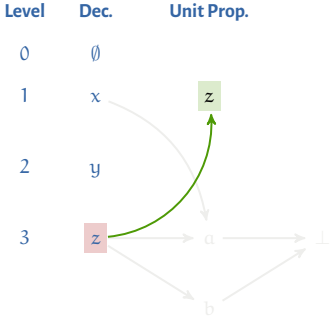


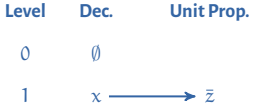
# Clause learning + backtracking




- **Clause $(\bar{x} \vee \bar{z})$ is asserting at decision level 1**

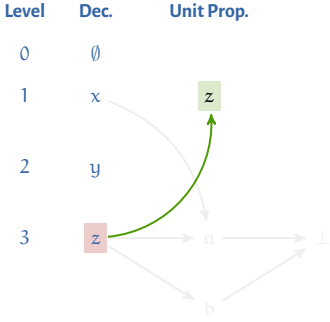
# Clause learning + backtracking

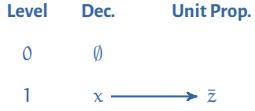



- **Clause $(\bar{x} \vee \bar{z})$ is asserting at decision level 1**

# Clause learning + backtracking

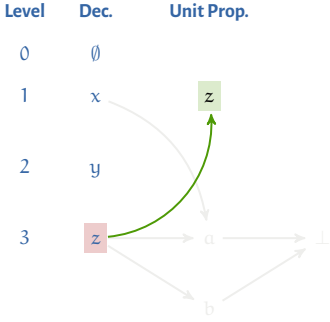


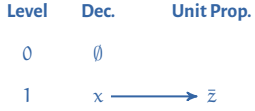




- **Clause $(\bar{x} \vee \bar{z})$ is asserting at decision level 1**
- **Learned clauses are asserting**

[MS95, MSS96b, MSS99]

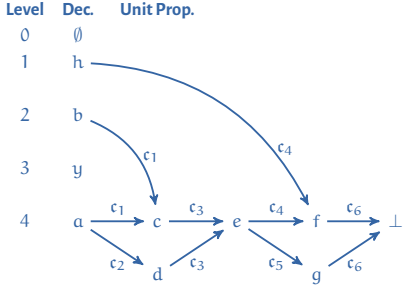
# Clause learning + backtracking



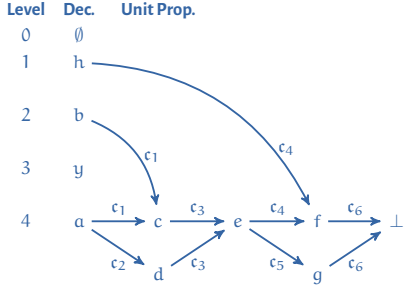




- **Clause $(\bar{x} \vee \bar{z})$ is asserting at decision level 1**
- **Learned clauses are asserting** [MS95, MSS96b, MSS99]
- **Backtracking differs from plain DPLL:**
  - **Non-chronological backtracking - backjumping!**

# Example – conflict analysis

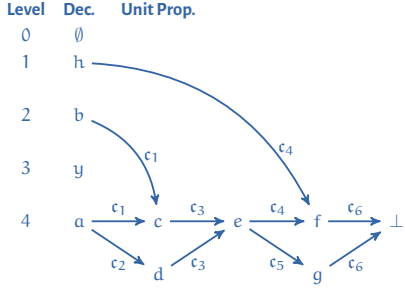


# Example – conflict analysis



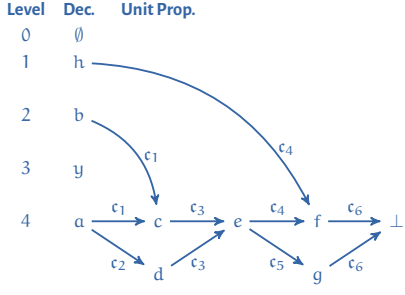

| Step | Var Queue | Extract Var | Antecedent | Recorded Lits | Added to Queue |
|---|---|---|---|---|---|
| 0 | – | $\bot$ | $c_6$ | $\emptyset$ | $\{f, g\}$ |

# Example – conflict analysis



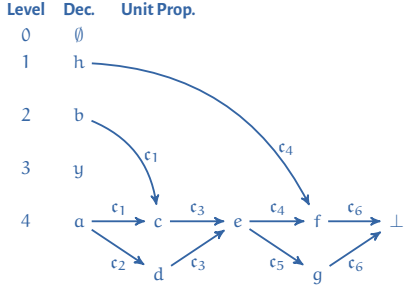

| Step | Var Queue | Extract Var | Antecedent | Recorded Lits | Added to Queue |
|---|---|---|---|---|---|
| 0 | – | $\bot$ | $c_6$ | $\emptyset$ | $\{f, g\}$ |
| 1 | $[f, g]$ | $f$ | $c_4$ | $\{\overline{h}\}$ | $\{e\}$ |

# Example – conflict analysis



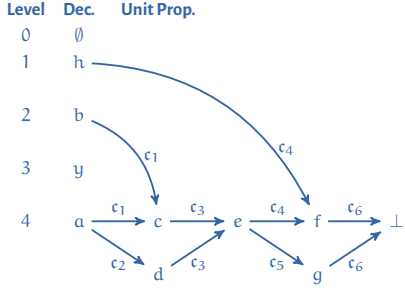

| Step | Var Queue | Extract Var | Antecedent | Recorded Lits | Added to Queue |
|---|---|---|---|---|---|
| 0 | – | $\bot$ | $c_6$ | $\emptyset$ | $\{f, g\}$ |
| 1 | $[f, g]$ | $f$ | $c_4$ | $\{\bar{h}\}$ | $\{e\}$ |
| 2 | $[g, e]$ | $g$ | $c_5$ | $\{\bar{h}\}$ | $\emptyset$ |

# Example – conflict analysis



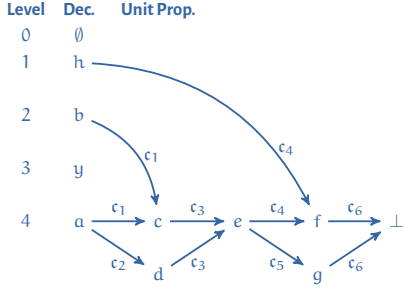

| Step | Var Queue | Extract Var | Antecedent | Recorded Lits | Added to Queue |
|---|---|---|---|---|---|
| 0 | – | $\bot$ | $c_6$ | $\emptyset$ | $\{f, g\}$ |
| 1 | $[f, g]$ | $f$ | $c_4$ | $\{\bar{h}\}$ | $\{e\}$ |
| 2 | $[g, e]$ | $g$ | $c_5$ | $\{\bar{h}\}$ | $\emptyset$ |
| 3 | $[e]$ | $e$ | $c_3$ | $\{\bar{h}\}$ | $\{c, d\}$ |

# Example – conflict analysis



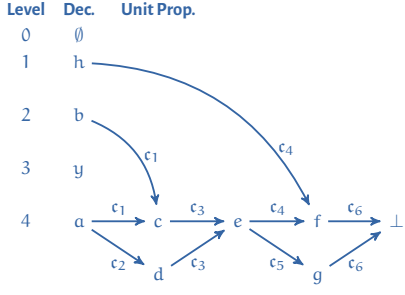

| Step | Var Queue | Extract Var | Antecedent | Recorded Lits | Added to Queue |
|---|---|---|---|---|---|
| 0 | – | $\bot$ | $c_6$ | $\emptyset$ | $\{f, g\}$ |
| 1 | $[f, g]$ | $f$ | $c_4$ | $\{\bar{h}\}$ | $\{e\}$ |
| 2 | $[g, e]$ | $g$ | $c_5$ | $\{\bar{h}\}$ | $\emptyset$ |
| 3 | $[e]$ | $e$ | $c_3$ | $\{\bar{h}\}$ | $\{c, d\}$ |
| 4 | $[c, d]$ | $c$ | $c_1$ | $\{\bar{h}, \bar{b}\}$ | $\{a\}$ |

# Example – conflict analysis



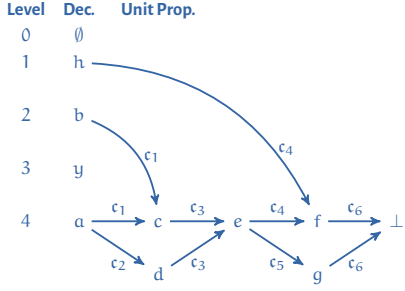

| Step | Var Queue | Extract Var | Antecedent | Recorded Lits | Added to Queue |
|---|---|---|---|---|---|
| 0 | – | $\bot$ | $c_6$ | $\emptyset$ | $\{f, g\}$ |
| 1 | $[f, g]$ | $f$ | $c_4$ | $\{\bar{h}\}$ | $\{e\}$ |
| 2 | $[g, e]$ | $g$ | $c_5$ | $\{\bar{h}\}$ | $\emptyset$ |
| 3 | $[e]$ | $e$ | $c_3$ | $\{\bar{h}\}$ | $\{c, d\}$ |
| 4 | $[c, d]$ | $c$ | $c_1$ | $\{\bar{h}, \bar{b}\}$ | $\{a\}$ |
| 5 | $[d, a]$ | $d$ | $c_2$ | $\{\bar{h}, \bar{b}\}$ | $\emptyset$ |

# Example – conflict analysis




| Step | Var Queue | Extract Var | Antecedent | Recorded Lits | Added to Queue |
|---|---|---|---|---|---|
| 0 | – | $\bot$ | $c_6$ | $\emptyset$ | $\{f, g\}$ |
| 1 | $[f, g]$ | $f$ | $c_4$ | $\{\bar{h}\}$ | $\{e\}$ |
| 2 | $[g, e]$ | $g$ | $c_5$ | $\{\bar{h}\}$ | $\emptyset$ |
| 3 | $[e]$ | $e$ | $c_3$ | $\{\bar{h}\}$ | $\{c, d\}$ |
| 4 | $[c, d]$ | $c$ | $c_1$ | $\{\bar{h}, \bar{b}\}$ | $\{a\}$ |
| 5 | $[d, a]$ | $d$ | $c_2$ | $\{\bar{h}, \bar{b}\}$ | $\emptyset$ |
| 6 | $[a]$ | $a$ | **dec var** | $\{\bar{h}, \bar{b}, \bar{a}\}$ | – |

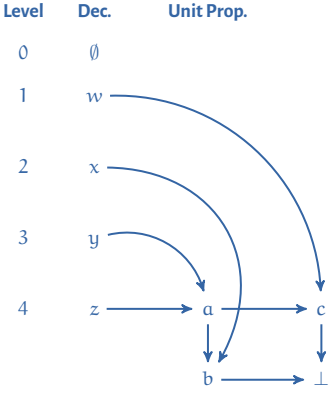
# Example – conflict analysis



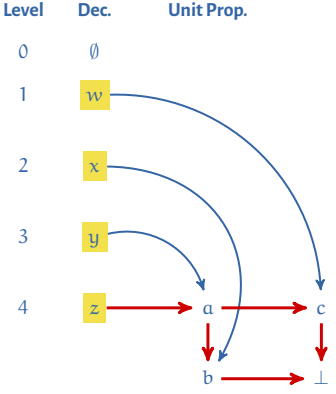

| Step | Var Queue | Extract Var | Antecedent | Recorded Lits | Added to Queue |
|---|---|---|---|---|---|
| 0 | – | $\bot$ | $c_6$ | $\emptyset$ | $\{f, g\}$ |
| 1 | $[f, g]$ | $f$ | $c_4$ | $\{\overline{h}\}$ | $\{e\}$ |
| 2 | $[g, e]$ | $g$ | $c_5$ | $\{\overline{h}\}$ | $\emptyset$ |
| 3 | $[e]$ | $e$ | $c_3$ | $\{\overline{h}\}$ | $\{c, d\}$ |
| 4 | $[c, d]$ | $c$ | $c_1$ | $\{\overline{h}, \overline{b}\}$ | $\{a\}$ |
| 5 | $[d, a]$ | $d$ | $c_2$ | $\{\overline{h}, \overline{b}\}$ | $\emptyset$ |
| 6 | $[a]$ | $a$ | **dec var** | $\{\overline{h}, \overline{b}, \overline{a}\}$ | – |
| 7 | $[]$ | – | – | $\{\overline{h}, \overline{b}, \overline{a}\}$ | – |

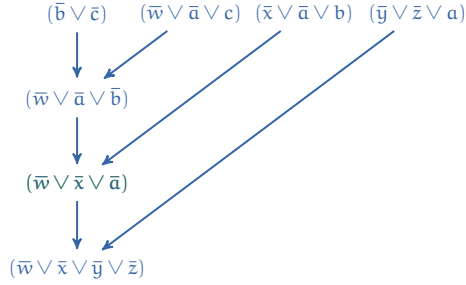
# Unique implication points

# Unique implication points (UIPs)



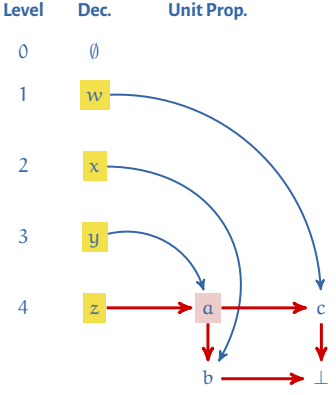
# Unique implication points (UIPs)

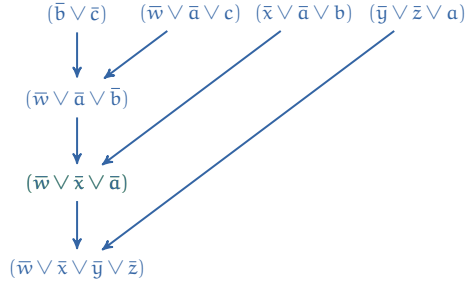



- **Learn clause** $(\overline{w} \vee \overline{x} \vee \overline{y} \vee \overline{z})$

# Unique implication points (UIPs)

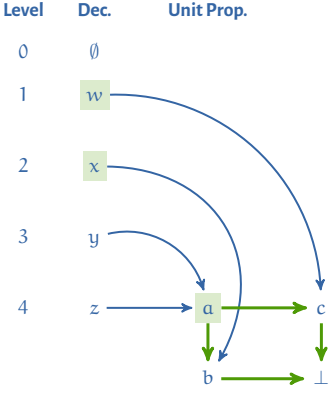


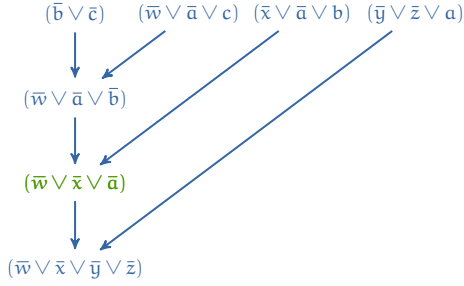




- **Learn clause** $(\bar{w} \vee \bar{x} \vee \bar{y} \vee \bar{z})$
- **But $a$ is an UIP**
  - **Dominator in DAG for decision level 4**

[MS95, MSS99]

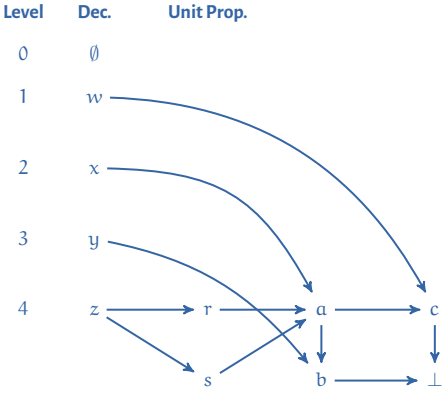
# Unique implication points (UIPs)



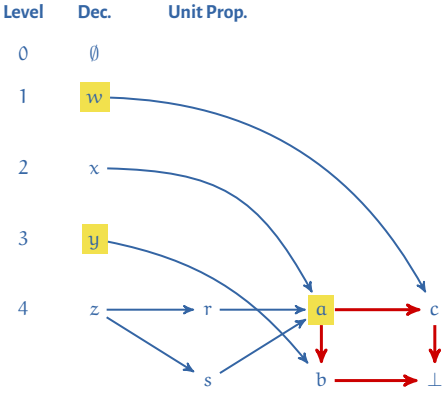




- ~~**Learn** clause $(\overline{w} \vee \overline{x} \vee \overline{y} \vee \overline{z})$~~
- **But $a$ is an UIP** [MS95, MSS99]
  - **Dominator in DAG for decision level 4**
- **Learn clause $(\overline{w} \vee \overline{x} \vee \overline{a})$**

# Multiple UIPs

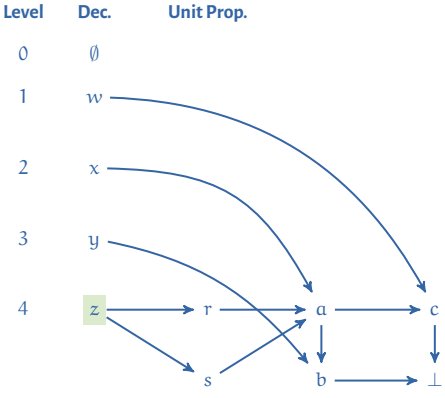


# Multiple UIPs



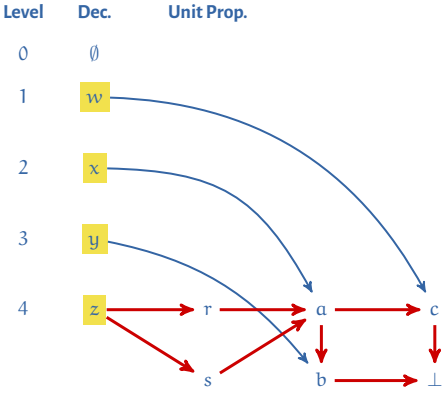

- **First UIP**:
  - **Learn clause** $(\overline{w} \vee \overline{y} \vee \overline{a})$

# Multiple UIPs



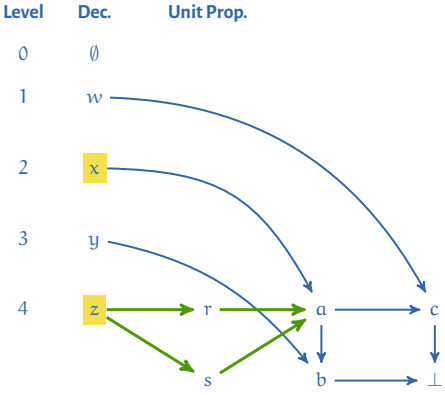

- **First UIP**:
  - **Learn clause** $(\overline{w} \vee \overline{y} \vee \overline{a})$
- **But there can be multiple UIPs!**

# Multiple UIPs



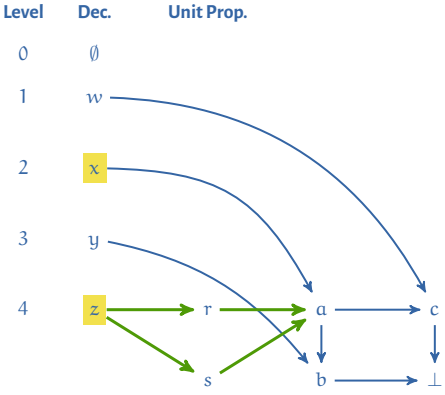

- **First UIP**:
  - **Learn clause** $(\overline{w} \vee \overline{y} \vee \overline{a})$
- **But there can be multiple UIPs!**
- **Second UIP**:
  - **Learn clause** $(\overline{w} \vee \overline{x} \vee \overline{y} \vee \overline{z})$

# Multiple UIPs



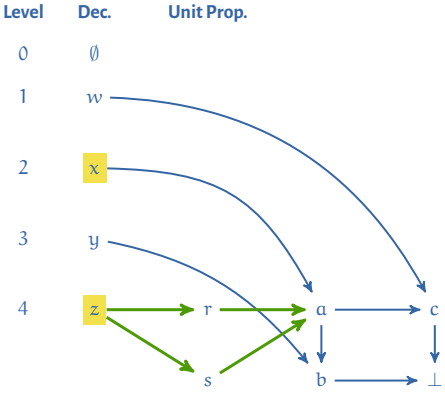

- **First UIP**:
  - **Learn clause** $(\overline{w} \vee \overline{y} \vee \overline{a})$
- **But there can be multiple UIPs!**
- **Second UIP**:
  - **Learn clause** $(\overline{w} \vee \overline{x} \vee \overline{y} \vee \overline{z})$
  - Optionally, learn $(\overline{x} \vee \overline{z} \vee a)$
  - (clause is **not** asserting)

# Multiple UIPs




- **First UIP**:
  - **Learn clause** $(\overline{w} \vee \overline{y} \vee \overline{a})$
- **But there can be multiple UIPs!**
- **Second UIP**:
  - **Learn clause** $(\overline{w} \vee \overline{x} \vee \overline{y} \vee \overline{z})$
  - Optionally, learn $(\overline{x} \vee \overline{z} \vee a)$
  - (clause is **not** asserting)
- **In practice smaller clauses more effective**

# Multiple UIPs



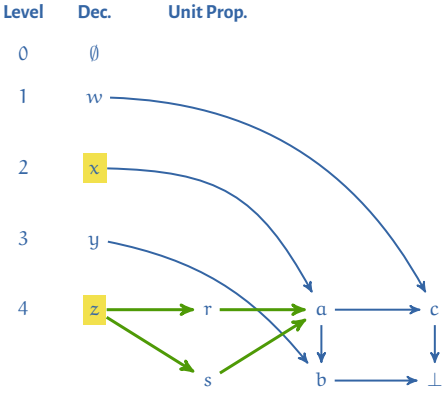

- Multiple UIPs proposed in GRASP
  - First UIP learning proposed in Chaff

[MS95, MSS99]

[MMZ+01]

- **First UIP**:
  - **Learn clause** $(\overline{w} \vee \overline{y} \vee \overline{a})$
- **But there can be multiple UIPs!**
- **Second UIP**:
  - **Learn clause** $(\overline{w} \vee \overline{x} \vee \overline{y} \vee \overline{z})$
  - Optionally, learn $(\overline{x} \vee \overline{z} \vee a)$
  - (clause is **not** asserting)
- **In practice smaller clauses more effective**

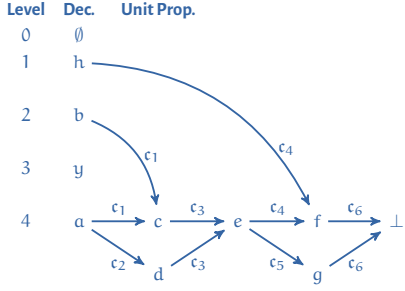
# Multiple UIPs



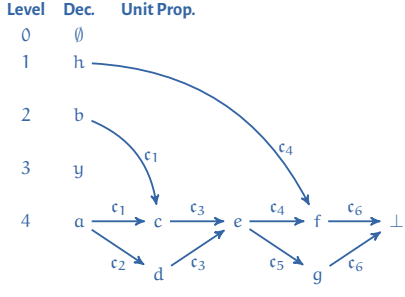

- **First UIP**:
  - **Learn clause** $(\overline{w} \vee \overline{y} \vee \overline{a})$
- **But there can be multiple UIPs!**
- **Second UIP**:
  - **Learn clause** $(\overline{w} \vee \overline{x} \vee \overline{y} \vee \overline{z})$
  - Optionally, learn $(\overline{x} \vee \overline{z} \vee a)$
  - (clause is **not** asserting)
- **In practice smaller clauses more effective**

- Multiple UIPs proposed in GRASP [MS95, MSS99]
  - First UIP learning proposed in Chaff [MMZ+01]
- Not used in recent state of the art CDCL SAT solvers
- But recent results show it can be beneficial on some instances [SSS12]

# Example – conflict analysis with UIP(s)

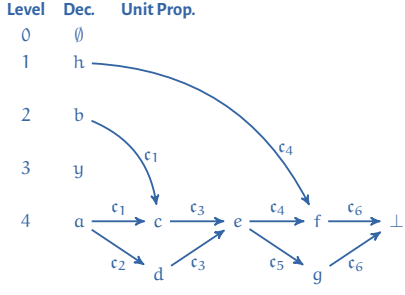


# Example – conflict analysis with UIP(s)



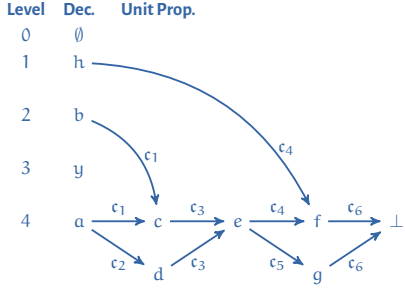

| Step | Var Queue | Extract Var | Antecedent | Recorded Lits | Added to Queue |
|---|---|---|---|---|---|
| 0 | – | $\bot$ | $c_6$ | $\emptyset$ | $\{f, g\}$ |

# Example – conflict analysis with UIP(s)



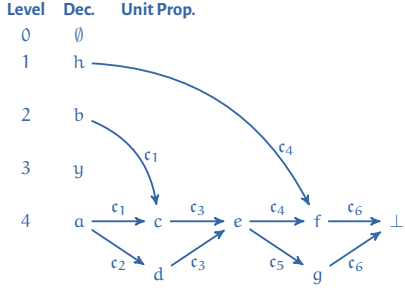

| Step | Var Queue | Extract Var | Antecedent | Recorded Lits | Added to Queue |
|---|---|---|---|---|---|
| 0 | – | $\bot$ | $c_6$ | $\emptyset$ | $\{f, g\}$ |
| 1 | $[f, g]$ | $f$ | $c_4$ | $\{\bar{h}\}$ | $\{e\}$ |

# Example – conflict analysis with UIP(s)



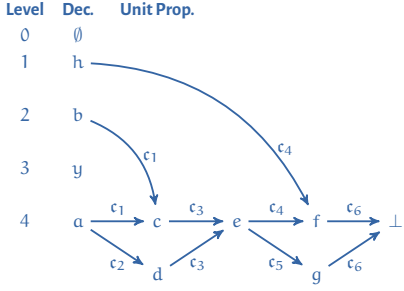

| Step | Var Queue | Extract Var | Antecedent | Recorded Lits | Added to Queue |
|---|---|---|---|---|---|
| 0 | – | $\bot$ | $c_6$ | $\emptyset$ | $\{f, g\}$ |
| 1 | $[f, g]$ | $f$ | $c_4$ | $\{\bar{h}\}$ | $\{e\}$ |
| 2 | $[g, e]$ | $g$ | $c_5$ | $\{\bar{h}\}$ | $\emptyset$ |

# Example – conflict analysis with UIP(s)




| Step | Var Queue | Extract Var | Antecedent | Recorded Lits | Added to Queue |
|---|---|---|---|---|---|
| 0 | – | $\bot$ | $c_6$ | $\emptyset$ | $\{f, g\}$ |
| 1 | $[f, g]$ | $f$ | $c_4$ | $\{\bar{h}\}$ | $\{e\}$ |
| 2 | $[g, e]$ | $g$ | $c_5$ | $\{\bar{h}\}$ | $\emptyset$ |
| 3 | $[e]$ | $e$ | $c_3$ | $\{\bar{h}, \bar{e}\}$ | $\emptyset$ |

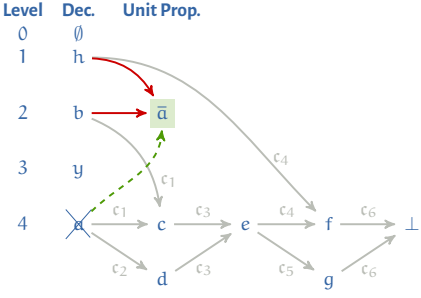
# Example – conflict analysis with UIP(s)



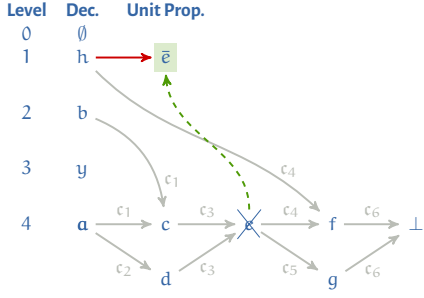

| Step | Var Queue | Extract Var | Antecedent | Recorded Lits | Added to Queue |
|---|---|---|---|---|---|
| 0 | – | $\bot$ | $c_6$ | $\emptyset$ | $\{f, g\}$ |
| 1 | $[f, g]$ | $f$ | $c_4$ | $\{\bar{h}\}$ | $\{e\}$ |
| 2 | $[g, e]$ | $g$ | $c_5$ | $\{\bar{h}\}$ | $\emptyset$ |
| 3 | $[e]$ | $e$ | $c_3$ | $\{\bar{h}, \bar{e}\}$ | $\emptyset$ |
| 4 | $[]$ | – | – | $\{\bar{h}, \bar{e}\}$ | – |

# Example (Cont.) – backjumping

**Without UIP:**




**With UIP:**



# exercise time

# Branching

# Which variable to branch on – static branching

- **Static heuristics**
  - **largest individual sum (LIS)** [MS99]
    - number of variable occurrences in clauses, i.e.
    - $L(l) = \sum_{l \in c \wedge c \in \mathcal{F}} 1$

# Which variable to branch on – static branching

- **Static heuristics**
  - **largest individual sum (LIS)** [MS99]
    - number of variable occurrences in clauses, i.e.
    - $L(l) = \sum_{l \in c \wedge c \in \mathcal{F}} 1$
  - **normalized LIS (Jeroslow-Wang)** [JW90]
    - $J(l) = \sum_{l \in c \wedge c \in \mathcal{F}} 2^{-|c|}$
  - ...

# Which variable to branch on – static branching

- **Static heuristics**
  - **largest individual sum (LIS)** [MS99]
    - number of variable occurrences in clauses, i.e.
    - $L(l) = \sum_{l \in c \wedge c \in \mathcal{F}} 1$
  - **normalized LIS (Jeroslow-Wang)** [JW90]
    - $J(l) = \sum_{l \in c \wedge c \in \mathcal{F}} 2^{-|c|}$
  - ...
  - **precomputed once**
  - **poor practical performance**

# Which variable to branch on – static branching

- **Static heuristics**
  - **largest individual sum (LIS)** [MS99]
    - number of variable occurrences in clauses, i.e.
    - $L(l) = \sum_{l \in c \wedge c \in \mathcal{F}} 1$
  - **normalized LIS (Jeroslow-Wang)** [JW90]
    - $J(l) = \sum_{l \in c \wedge c \in \mathcal{F}} 2^{-|c|}$
  - ...
  - **precomputed once**
  - **poor practical performance**

**Example**

$$\mathcal{F} = (x) \wedge (\neg x \vee y \vee z) \wedge (\neg y \vee z \vee w) \wedge (y \vee \neg z \vee w)$$

# Which variable to branch on – static branching

- **Static heuristics**
  - **largest individual sum (LIS)** [MS99]
    - number of variable occurrences in clauses, i.e.
    - $L(l) = \sum_{l \in c \wedge c \in \mathcal{F}} 1$
  - **normalized LIS (Jeroslow-Wang)** [JW90]
    - $J(l) = \sum_{l \in c \wedge c \in \mathcal{F}} 2^{-|c|}$
  - ...
  - **precomputed once**
  - **poor practical performance**

**Example**

$$\mathcal{F} = (x) \wedge (\neg x \vee y \vee z) \wedge (\neg y \vee z \vee w) \wedge (y \vee \neg z \vee w)$$

$$L(x) = 2, L(y) = 3, L(z) = 3, L(w) = 2$$

# Which variable to branch on – static branching

- **Static heuristics**
  - **largest individual sum (LIS)** [MS99]
    - number of variable occurrences in clauses, i.e.
    - $L(l) = \sum_{l \in c \wedge c \in \mathcal{F}} 1$
  - **normalized LIS (Jeroslow-Wang)** [JW90]
    - $J(l) = \sum_{l \in c \wedge c \in \mathcal{F}} 2^{-|c|}$
  - ...
  - **precomputed once**
  - **poor practical performance**

**Example**

$$\mathcal{F} = (x) \wedge (\neg x \vee y \vee z) \wedge (\neg y \vee z \vee w) \wedge (y \vee \neg z \vee w)$$

$$L(x) = 2, L(y) = 3, L(z) = 3, L(w) = 2$$

$$J(x) = \frac{5}{8}, J(y) = \frac{3}{8}, J(z) = \frac{3}{8}, J(w) = \frac{2}{8}$$

# Which variable to branch on – dynamic branching

- **Dynamic heuristics**
  - **Dynamic LIS** [MS99]
    - Calculate for **remaining unsatisfied** clauses
    - Expensive to do

# Which variable to branch on – dynamic branching

- **Dynamic heuristics**
  - **Dynamic LIS** [MS99]
    - Calculate for **remaining unsatisfied** clauses
    - Expensive to do
  - **Variable state independent Decaying Sum (VSIDS)** [MMZ+01]

# Which variable to branch on – dynamic branching

- **Dynamic heuristics**
  - **Dynamic LIS** [MS99]
    - Calculate for **remaining unsatisfied** clauses
    - Expensive to do
  - **Variable state independent Decaying Sum (VSIDS)** [MMZ+01]
    - Initially weights of all variables are **set to 0**

# Which variable to branch on – dynamic branching

- **Dynamic heuristics**
  - **Dynamic LIS** [MS99]
    - Calculate for **remaining unsatisfied** clauses
    - Expensive to do
  - **Variable state independent Decaying Sum (VSIDS)** [MMZ+01]
    - Initially weights of all variables are **set to 0**
    - Weights of active variables are **incremented for each conflict**

# Which variable to branch on – dynamic branching

- **Dynamic heuristics**
  - **Dynamic LIS** [MS99]
    - Calculate for **remaining unsatisfied** clauses
    - Expensive to do
  - **Variable state independent Decaying Sum (VSIDS)** [MMZ+01]
    - Initially weights of all variables are **set to 0**
    - Weights of active variables are **incremented for each conflict**
    - All weights are **divided by a constant *periodically***

# Which variable to branch on – dynamic branching

- **Dynamic heuristics**
  - **Dynamic LIS** [MS99]
    - Calculate for **remaining unsatisfied** clauses
    - Expensive to do
  - **Variable state independent Decaying Sum (VSIDS)** [MMZ+01]
    - Initially weights of all variables are **set to 0**
    - Weights of active variables are **incremented for each conflict**
    - All weights are **divided by a constant *periodically***
    - Focus on variables that were **active in recent conflicts**
    - Can be interpreted as *reinforcement learning*
    - There are variants

# Which variable to branch on – dynamic branching

- **Dynamic heuristics**
  - **Dynamic LIS** [MS99]
    - Calculate for **remaining unsatisfied** clauses
    - Expensive to do
  - **Variable state independent Decaying Sum (VSIDS)** [MMZ+01]
    - Initially weights of all variables are **set to 0**
    - Weights of active variables are **incremented for each conflict**
    - All weights are **divided by a constant *periodically***
    - Focus on variables that were **active in recent conflicts**
    - Can be interpreted as *reinforcement learning*
    - There are variants
  - **Recent promising ML-based branching** [LGPC16a, LGPC16b]

# exercise time

# Lazy data structures

# How to implement unit propagation?

- **Recap states of a clause: unresolved, unit, falsified, satisfied**
- **Each literal $l$ should access clauses containing $l$ and $\neg l$**
  - Why?

# How to implement unit propagation?

- **Recap states of a clause: unresolved, unit, falsified, satisfied**
- **Each literal $l$ should access clauses containing $l$ and $\neg l$**
  - Why? Unit propagation!

# How to implement unit propagation?

- **Recap states of a clause: unresolved, unit, falsified, satisfied**
- **Each literal $l$ should access clauses containing $l$ and $\neg l$**
  - Why? Unit propagation!
- **Clause with $k$ literals results in $k$ references, from literals to the clause**

# How to implement unit propagation?

- **Recap states of a clause: unresolved, unit, falsified, satisfied**
- **Each literal $l$ should access clauses containing $l$ and $\neg l$**
  - Why? Unit propagation!
- **Clause with $k$ literals results in $k$ references, from literals to the clause**
- **Number of clause references equals total number of literals**

# How to implement unit propagation?

- **Recap states of a clause: unresolved, unit, falsified, satisfied**
- **Each literal $l$ should access clauses containing $l$ and $\neg l$**
  - Why? Unit propagation!
- **Clause with $k$ literals results in $k$ references, from literals to the clause**
- **Number of clause references equals total number of literals**
  - Clause learning can generate **large** clauses
    - Worst-case size: $\mathcal{O}(n)$

# How to implement unit propagation?

- **Recap states of a clause: unresolved, unit, falsified, satisfied**
- **Each literal $l$ should access clauses containing $l$ and $\neg l$**
  - Why? Unit propagation!
- **Clause with $k$ literals results in $k$ references, from literals to the clause**
- **Number of clause references equals total number of literals**
  - Clause learning can generate **large** clauses
    - Worst-case size: $\mathcal{O}(n)$
  - Worst-case number of literals: $\mathcal{O}(m\,n)$

# How to implement unit propagation?

- **Recap states of a clause: unresolved, unit, falsified, satisfied**
- **Each literal $l$ should access clauses containing $l$ and $\neg l$**
  - Why? Unit propagation!
- **Clause with $k$ literals results in $k$ references, from literals to the clause**
- **Number of clause references equals total number of literals**
  - Clause learning can generate **large** clauses
    - Worst-case size: $\mathcal{O}(n)$
  - Worst-case number of literals: $\mathcal{O}(m\,n)$
  - In practice,

    Unit propagation slow-down worse than linear as clauses are learned **!**

# How to implement unit propagation?

- **Recap states of a clause: unresolved, unit, falsified, satisfied**
- **Each literal $l$ should access clauses containing $l$ and $\neg l$**
  - Why? Unit propagation!
- **Clause with $k$ literals results in $k$ references, from literals to the clause**
- **Number of clause references equals total number of literals**
  - Clause learning can generate **large** clauses
    - Worst-case size: $\mathcal{O}(n)$
  - Worst-case number of literals: $\mathcal{O}(m\,n)$
  - In practice,

    Unit propagation slow-down worse than linear as clauses are learned **!**
- **More efficient representation is needed: watched literals** [MMZ+01]

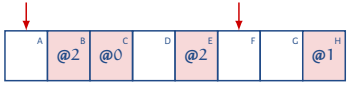
# How to implement unit propagation?

- **Recap states of a clause: unresolved, unit, falsified, satisfied**
- **Each literal $l$ should access clauses containing $l$ and $\neg l$**
  - Why? Unit propagation!
- **Clause with $k$ literals results in $k$ references, from literals to the clause**
- **Number of clause references equals total number of literals**
  - Clause learning can generate **large** clauses
    - Worst-case size: $\mathcal{O}(n)$
  - Worst-case number of literals: $\mathcal{O}(m\,n)$
  - In practice,

    Unit propagation slow-down worse than linear as clauses are learned **!**
- **More efficient representation is needed: watched literals** [MMZ+01]
  - Watched literals are one example of lazy data structures
    - But there are others, e.g. head-tail literals [ZS00]

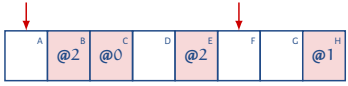
# Watched literals

# Watched literals




**Watch 2 unassigned literals in each clause**

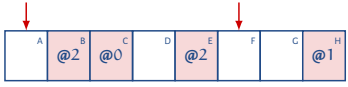
# Watched literals




**Watch 2 unassigned literals in each clause**

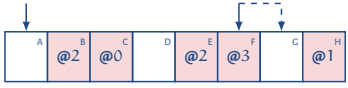
At DLevel 2: clause is unresolved

# Watched literals



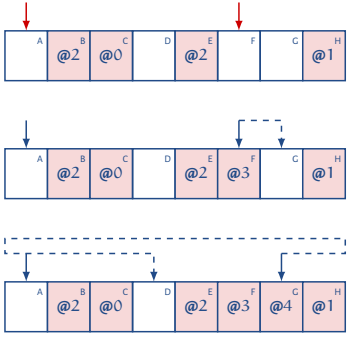

**Watch 2 unassigned literals in each clause**

At DLevel 2: clause is unresolved




At DLevel 3: watch updated

# Watched literals



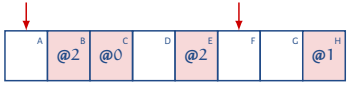

**Watch 2 unassigned literals in each clause**

At DLevel 2: clause is unresolved

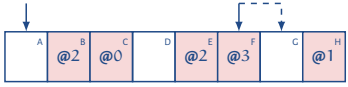
At DLevel 3: watch updated

At DLevel 4: watch updated

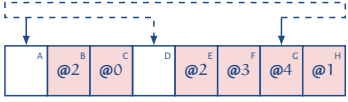
# Watched literals



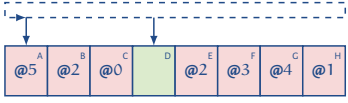

**Watch 2 unassigned literals in each clause**
At DLevel 2: clause is unresolved




At DLevel 3: watch updated

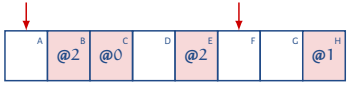



At DLevel 4: watch updated

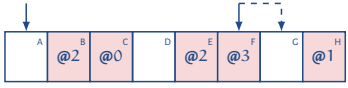



At DLevel 5: clause is unit
**Literal D assigned value 1; clause becomes satisfied**

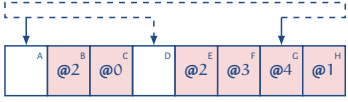
# Watched literals



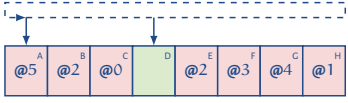

**Watch 2 unassigned literals in each clause**
At DLevel 2: clause is unresolved



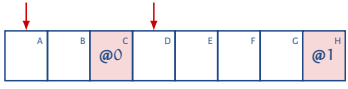

At DLevel 3: watch updated




At DLevel 4: watch updated

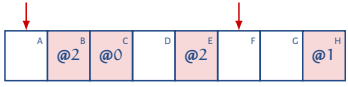



At DLevel 5: clause is unit
**Literal D assigned value 1; clause becomes satisfied**

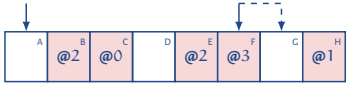



After backtracking to DLevel 1
**Watched literals untouched**

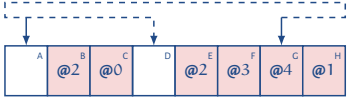
# Watched literals – different implementations exist!



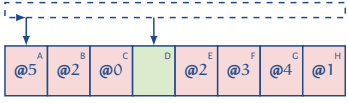

**Watch 2 unassigned literals in each clause**
At DLevel 2: clause is unresolved



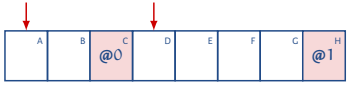

At DLevel 3: watch updated




At DLevel 4: watch updated




At DLevel 5: clause is unit
**Literal D assigned value 1; clause becomes satisfied**




After backtracking to DLevel 1
**Watched literals untouched**

# References i



Gilles Audemard and Laurent Simon.
**Predicting learnt clauses quality in modern SAT solvers.**
In *IJCAI*, pages 399–404, 2009.


Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors.
***Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.**


Armin Biere.
**PicoSAT essentials.**
*JSAT*, 4(2-4):75–97, 2008.


Luís Baptista and Joao Marques-Silva.
**Using randomization and learning to solve hard real-world instances of satisfiability.**
In *CP*, volume 1894 of *Lecture Notes in Computer Science*, pages 489–494. Springer, 2000.

# References ii



Stephen A. Cook.
**The complexity of theorem-proving procedures.**
In *STOC*, pages 151–158. ACM, 1971.


Roberto Di Cosmo, Berke Durak, Xavier Leroy, Fabio Mancinelli, and Jérôme Vouillon.
**Maintaining large software distributions: New challenges from the FOSS era.**
In *FRCSS*, volume 12, pages 7–20, 2006.


Martin Davis, George Logemann, and Donald W. Loveland.
**A machine program for theorem-proving.**
*Commun. ACM*, 5(7):394–397, 1962.


Martin Davis and Hilary Putnam.
**A computing procedure for quantification theory.**
*J. ACM*, 7(3):201–215, 1960.

# References iii



Allen Van Gelder.
**Improved conflict-clause minimization leads to improved propositional proof traces.**
In *SAT*, pages 141–146, 2009.


Evguenii I. Goldberg and Yakov Novikov.
**BerkMin: A fast and robust SAT-solver.**
In *DATE*, pages 142–149. IEEE Computer Society, 2002.


Carla P. Gomes, Bart Selman, and Nuno Crato.
**Heavy-tailed distributions in combinatorial search.**
In *CP*, volume 1330 of *Lecture Notes in Computer Science*, pages 121–135. Springer, 1997.


Jinbo Huang.
**The effect of restarts on the efficiency of clause learning.**
In *IJCAI*, pages 2318–2323, 2007.

# References iv



Alexey Ignatiev, Antonio Morgado, and Joao Marques-Silva.
**PySAT: A Python toolkit for prototyping with SAT oracles.**
In *SAT*, pages 428–437, 2018.


Alexey Ignatiev, Zi Li Tan, and Christos Karamanos.
**Towards universally accessible SAT technology.**
In *SAT*, pages 16:1–16:11, 2024.


Robert G. Jeroslow and Jinchang Wang.
**Solving propositional satisfiability problems.**
*Ann. Math. Artif. Intell.*, 1:167–187, 1990.


Richard M. Karp.
**Reducibility among combinatorial problems.**
In *Complexity of Computer Computations*, pages 85–103. Plenum Press, New York, 1972.

## References v



Jia Hui Liang, Vijay Ganesh, Pascal Poupart, and Krzysztof Czarnecki.
**Exponential recency weighted average branching heuristic for SAT solvers.**
In *AAAI*, pages 3434–3440, 2016.


Jia Hui Liang, Vijay Ganesh, Pascal Poupart, and Krzysztof Czarnecki.
**Learning rate based branching heuristic for SAT solvers.**
In *SAT*, pages 123–140, 2016.


Mao Luo, Chu-Min Li, Fan Xiao, Felip Manyà, and Zhipeng Lü.
**An effective learnt clause minimization approach for CDCL SAT solvers.**
In *IJCAI*, pages 703–711, 2017.


Jia Hui Liang, Chanseok Oh, Minu Mathew, Ciza Thomas, Chunxiao Li, and Vijay Ganesh.
**Machine learning-based restart policy for CDCL SAT solvers.**
In *SAT*, pages 94–110, 2018.

# References vi



Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik.
**Chaff: Engineering an efficient SAT solver.**
In *DAC*, pages 530–535. ACM, 2001.


J. Marques-Silva.
***Search Algorithms for Satisfiability Problems in Combinational Switching Circuits.***
PhD thesis, University of Michigan, May 1995.


Joao Marques-Silva.
**The impact of branching heuristics in propositional satisfiability algorithms.**
In *EPIA*, volume 1695 of *Lecture Notes in Computer Science*, pages 62–74. Springer, 1999.


Joao Marques-Silva and Karem A. Sakallah.
**Conflict analysis in search algorithms for propositional satisfiability.**
Technical Report RT-04-96, INESC, May 1996.

## References vii



Joao Marques-Silva and Karem A. Sakallah.
**GRASP - a new search algorithm for satisfiability.**
In *ICCAD*, pages 220–227, 1996.


Joao Marques-Silva and Karem A. Sakallah.
**GRASP: A search algorithm for propositional satisfiability.**
*IEEE Trans. Computers*, 48(5):506–521, 1999.


Knot Pipatsrisawat and Adnan Darwiche.
**A lightweight component caching scheme for satisfiability solvers.**
In *SAT*, volume 4501 of *Lecture Notes in Computer Science*, pages 294–299. Springer, 2007.


John Alan Robinson.
**A machine-oriented logic based on the resolution principle.**
*J. ACM*, 12(1):23–41, 1965.

## References viii



Niklas Sörensson and Armin Biere.
**Minimizing learned clauses.**
In *SAT*, volume 5584 of *Lecture Notes in Computer Science*, pages 237–243. Springer, 2009.


Ashish Sabharwal, Horst Samulowitz, and Meinolf Sellmann.
**Learning back-clauses in SAT.**
In *SAT*, pages 498–499, 2012.


Lintao Zhang, Conor F. Madigan, Matthew W. Moskewicz, and Sharad Malik.
**Efficient conflict driven learning in boolean satisfiability solver.**
In *ICCAD*, pages 279–285. IEEE Computer Society, 2001.


Hantao Zhang and Mark E. Stickel.
**Implementing the Davis-Putnam method.**
*J. Autom. Reasoning*, 24(1/2):277–296, 2000.