Solvers Principles and Architecture (SPA)

Lecture 1

SAT Solvers

Master Sciences Informatique (Sif)
September 25th, 2017
Rennes

Khalil Ghorbal
khalil.ghorbal@inria.fr

Specification

Formal specification of the system:

- What the system is expected to do
- What properties the system has to satisfy

Modeling

Mathematical model describing the behavior of the system:

- Finite state or hybrid automata
- Markov chain

Proof

Does the model satisfy the specification?
When successful: no **corner cases** left!

Outline

Formally, a **logic** is a pair of **syntax** and **semantics**.

Syntax

- Alphabet: set of symbols
- Expressions: sequences of symbols
- Rules: identifying **well-formed** expressions

Semantics

- **Meaning**: what is meant by well-formed expressions
- Rules: infer the meaning from subexpressions

Alphabet

(	left parenthesis
)	right parenthesis
$\neg$	Negation
$\wedge$	Conjunction
$\vee$	Disjunction (inclusive)
$\longleftarrow$	Implication
$\longleftrightarrow$	Equivalence
0	Propositional symbol “False”
1	Propositional symbol “True”
s_i	ith propositional symbol

Expression

Sequence of symbols from the alphabet.

$\langle (, a_1, \wedge, a_2,) \rangle$ $\quad$ $(a_1 \wedge a_2)$

$\langle (,), \vee, a_1, \neg, a_2 \rangle$ $\quad$ $() \vee a_1 \neg a_3$

We want to further **restrict** the allowed combinations.

Well-formed formulas (wff) are defined **inductively**

S: the set of expressions with a single propositional symbol

$$S = \{0, 1, s_1, s_2, \dots\}$$

W: the set of wffs is **freely generated** from S as follows

$$w ::= s \mid (w) \mid \neg w \mid w \wedge w \mid w \vee w \mid w \longrightarrow w \mid w \longleftrightarrow w$$

So far we only manipulated **symbols** or wooden pieces!

Semantics with Truth Table

s	$\neg$
0	1
1	0

Semantics with Truth Table

s	$\neg$
0	1
1	0

s_1	s_2	$\wedge$
0	0	0
0	1	0
1	0	0
1	1	1

s_1	s_2	$\vee$
0	0	0
0	1	1
1	0	1
1	1	1

Semantics with Truth Table

s	$\neg$
0	1
1	0

s_1	s_2	$\wedge$
0	0	0
0	1	0
1	0	0
1	1	1

s_1	s_2	$\vee$
0	0	0
0	1	1
1	0	1
1	1	1

s_1	s_2	$\rightarrow$
0	0	1
0	1	1
1	0	0
1	1	1

Intuition

Given a **context**, that is a **truth value** for each propositional symbol, we can determine the truth value of any wff in our context.

Boolean Algebra

- Field structure: $\mathbb{Z}/2\mathbb{Z} = \mathcal{B} = \{0, 1\}$
- "$+$": 0 is the identity, 1 is its own inverse: $1 + 1 = 0$
- "$\times$": standard multiplication operator, where 1 is the identity element

- **context**: $\sigma : S \to \mathcal{B}$. A valuation of all propositional symbols
- σ satisfies $\sigma(0) = 0$ and $\sigma(1) = 1$
- Define $[\![]\!]_\sigma : W \to \mathcal{B}$
- $[\![]\!]_\sigma$ is **well-defined** since W is freely generated

Semantics of the Transfer Functions

$$
\begin{aligned}
[\![s]\!]_\sigma &= \sigma(s) \\
[\![\neg w]\!]_\sigma &= 1 + [\![w]\!]_\sigma \\
[\![w_1 \wedge w_2]\!]_\sigma &= [\![w_1]\!]_\sigma \times [\![w_2]\!]_\sigma \\
[\![w_1 \vee w_2]\!]_\sigma &= [\![w_1]\!]_\sigma + [\![w_2]\!]_\sigma + [\![w_1]\!]_\sigma \times [\![w_2]\!]_\sigma \\
[\![w_1 \to w_2]\!]_\sigma &= 1 + [\![w_1]\!]_\sigma + [\![w_1]\!]_\sigma \times [\![w_2]\!]_\sigma \\
[\![w_1 \longleftrightarrow w_2]\!]_\sigma &= 1 + [\![w_1]\!]_\sigma + [\![w_2]\!]_\sigma
\end{aligned}
$$

Definitions

- σ: context, valuation, truth assignment
- σ **satisfies** w if and only if $[\![w]\!]_\sigma = 1$
- w is **satisfiable** if there exists σ such that σ satisfies w
- w is **unsatisfiable** if there is no σ such that σ satisfies w

Example:

- $(s_1 \vee s_2) \wedge (\neg s_1 \vee \neg s_2)$ is satisfiable
- $(s_1 \vee s_2) \wedge (\neg s_1 \vee \neg s_2) \wedge (s_1 \leftrightarrow s_2)$ is unsatisfiable

- σ: context, valuation, truth assignment

- σ **satisfies** w if and only if $[\![w]\!]_\sigma = 1$
- w is **satisfiable** if there exists σ such that σ satisfies w
- w is **unsatisfiable** if there is no σ such that σ satisfies w

Example:

- $(s_1 \vee s_2) \wedge (\neg s_1 \vee \neg s_2)$ is satisfiable
- $(s_1 \vee s_2) \wedge (\neg s_1 \vee \neg s_2) \wedge (s_1 \leftrightarrow s_2)$ is unsatisfiable

- σ: context, valuation, truth assignment

- σ **satisfies** w if and only if $[\![w]\!]_\sigma = 1$
- w is **satisfiable** if there exists σ such that σ satisfies w
- w is **unsatisfiable** if there is no σ such that σ satisfies w

Example:

- $(s_1 \vee s_2) \wedge (\neg s_1 \vee \neg s_2)$ is satisfiable
- $(s_1 \vee s_2) \wedge (\neg s_1 \vee \neg s_2) \wedge (s_1 \leftrightarrow s_2)$ is unsatisfiable

Tautological Implication (w_i are wffs)

$$w_1, \ldots, w_n \models w \quad \text{if and only if} \quad \forall \sigma. \left(\bigwedge_i [\![w_i]\!]_\sigma = 1 \longrightarrow [\![w]\!]_\sigma = 1 \right)$$

Every truth assignment that satisfies all w_i satisfies necessarily w

Definitions

- $\models w$ (or $1 \models w$): w is a tautology or w is **valid**
- $w_1 \sim w_2$: $w_1 \models w_2$ and $w_2 \models w_1$ (tautological equivalence)
- e.g. $s_1 \rightarrow s_2 \sim \neg s_1 \vee s_2$

Tautological Implication (w_i are wffs)

$$w_1, \ldots, w_n \models w \quad \text{if and only if} \quad \forall \sigma. \left(\bigwedge_i [\![w_i]\!]_\sigma = 1 \longrightarrow [\![w]\!]_\sigma = 1 \right)$$

Every truth assignment that satisfies all w_i satisfies necessarily w

Definitions

- $\models w$ (or $1 \models w$): w is a tautology or w is **valid**
- $w_1 \sim w_2$: $w_1 \models w_2$ and $w_2 \models w_1$ (tautological equivalence)
- e.g. $s_1 \rightarrow s_2 \sim \neg s_1 \vee s_2$

Tautological Implication (w_i are wffs)

$$w_1, \ldots, w_n \models w \quad \text{if and only if} \quad \forall \sigma. \left(\bigwedge_i [\![w_i]\!]_\sigma = 1 \longrightarrow [\![w]\!]_\sigma = 1 \right)$$

Every truth assignment that satisfies all w_i satisfies necessarily w

Definitions

- $\models w$ (or $1 \models w$): w is a tautology or w is **valid**
- $w_1 \sim w_2$: $w_1 \models w_2$ and $w_2 \models w_1$ (tautological equivalence)
- e.g. $s_1 \rightarrow s_2 \sim \neg s_1 \vee s_2$

Tautological Implication (w_i are wffs)

$$w_1, \ldots, w_n \models w \quad \text{if and only if} \quad \forall \sigma. \left(\bigwedge_i [\![w_i]\!]_\sigma = 1 \longrightarrow [\![w]\!]_\sigma = 1 \right)$$

Every truth assignment that satisfies all w_i satisfies necessarily w

Definitions

- $\models w$ (or $1 \models w$): w is a tautology or w is **valid**
- $w_1 \sim w_2$: $w_1 \models w_2$ and $w_2 \models w_1$ (tautological equivalence)
- e.g. $s_1 \rightarrow s_2 \sim \neg s_1 \vee s_2$

Tautological Implication as Satisfiability Problem

$$w_1, \ldots, w_n \models w \quad \text{if and only if} \quad \bigwedge_i w_i \wedge \neg w \quad \text{is unsatisfiable}$$

Example

- $s_1, s_1 \rightarrow s_2 \models s_2$ iff $s_1 \wedge (s_1 \rightarrow s_2) \wedge \neg s_2$ is unsat.
- $s, \neg s \models (s \wedge \neg s)$ iff $s \wedge \neg s \wedge \neg(s \wedge \neg s)$ is unsat

Tautological Implication as Satisfiability Problem

$$w_1, \ldots, w_n \models w \quad \text{if and only if} \quad \bigwedge_i w_i \wedge \neg w \quad \text{is unsatisfiable}$$

Example

- $s_1, s_1 \rightarrow s_2 \models s_2$ iff $s_1 \wedge (s_1 \rightarrow s_2) \wedge \neg s_2$ is unsat.
- $s, \neg s \models (s \wedge \neg s)$ iff $s \wedge \neg s \wedge \neg(s \wedge \neg s)$ is unsat

Tautological Implication as Satisfiability Problem

$$w_1, \ldots, w_n \models w \quad \text{if and only if} \quad \bigwedge_i w_i \wedge \neg w \quad \text{is unsatisfiable}$$

Example

- $s_1, s_1 \rightarrow s_2 \models s_2$ iff $s_1 \wedge (s_1 \rightarrow s_2) \wedge \neg s_2$ is unsat.
- $s, \neg s \models (s \wedge \neg s)$ iff $s \wedge \neg s \wedge \neg(s \wedge \neg s)$ is unsat

Outline

SAT Decision Problem

Given a well-formed formula w as an input, if there exists a σ that **satisfies** w return **True**, otherwise return **False**

$$s_1 \wedge (s_2 \vee \neg s_1) \wedge (s_3 \vee \neg s_2)$$

s_1	s_2	s_3	s_1	$\wedge$	$((s_2$	$\vee$	$\neg s_1)$	$\wedge$	$(s_3$	$\vee$	$\neg s_2))$

$$s_1 \wedge (s_2 \vee \neg s_1) \wedge (s_3 \vee \neg s_2)$$

	s_1	s_2	s_3	s_1	$\wedge$	$((s_2$	$\vee$	$\neg s_1)$	$\wedge$	$(s_3$	$\vee$	$\neg s_2))$
(1)	0	0	0		0		1	1	1		1	1

$$s_1 \wedge (s_2 \vee \neg s_1) \wedge (s_3 \vee \neg s_2)$$

	s_1	s_2	s_3	s_1	$\wedge$	$((s_2$	$\vee$	$\neg s_1)$	$\wedge$	$(s_3$	$\vee$	$\neg s_2))$
(1)	0	0	0		0		1	1	1		1	1
(2)	0	0	1		0		1	1	1		1	1

$$s_1 \wedge (s_2 \vee \neg s_1) \wedge (s_3 \vee \neg s_2)$$

	s_1	s_2	s_3	s_1	$\wedge$	$((s_2$	$\vee$	$\neg s_1)$	$\wedge$	$(s_3$	$\vee$	$\neg s_2))$
(1)	0	0	0		0		1	1	1		1	1
(2)	0	0	1		0		1	1	1		1	1
(3)	0	1	0		0		1	1	0		0	0

$$s_1 \wedge (s_2 \vee \neg s_1) \wedge (s_3 \vee \neg s_2)$$

	s_1	s_2	s_3	s_1	$\wedge$	$((s_2$	$\vee$	$\neg s_1)$	$\wedge$	$(s_3$	$\vee$	$\neg s_2))$
(1)	0	0	0		0		1	1	1		1	1
(2)	0	0	1		0		1	1	1		1	1
(3)	0	1	0		0		1	1	0		0	0
(4)	0	1	1		0		1	1	1		1	0

$$s_1 \wedge (s_2 \vee \neg s_1) \wedge (s_3 \vee \neg s_2)$$

	s_1	s_2	s_3	s_1	$\wedge$	$((s_2$	$\vee$	$\neg s_1)$	$\wedge$	$(s_3$	$\vee$	$\neg s_2))$
(1)	0	0	0		0		1	1	1		1	1
(2)	0	0	1		0		1	1	1		1	1
(3)	0	1	0		0		1	1	0		0	0
(4)	0	1	1		0		1	1	1		1	0
(5)	1	0	0		0		0	0	0		1	1

$$s_1 \wedge (s_2 \vee \neg s_1) \wedge (s_3 \vee \neg s_2)$$

	s_1	s_2	s_3	s_1	$\wedge$	$((s_2$	$\vee$	$\neg s_1)$	$\wedge$	$(s_3$	$\vee$	$\neg s_2))$
(1)	0	0	0		0		1	1	1		1	1
(2)	0	0	1		0		1	1	1		1	1
(3)	0	1	0		0		1	1	0		0	0
(4)	0	1	1		0		1	1	1		1	0
(5)	1	0	0		0		0	0	0		1	1
(6)	1	0	1		0		0	0	0		1	1

$$s_1 \wedge (s_2 \vee \neg s_1) \wedge (s_3 \vee \neg s_2)$$

	s_1	s_2	s_3	s_1	$\wedge$	$((s_2$	$\vee$	$\neg s_1)$	$\wedge$	$(s_3$	$\vee$	$\neg s_2))$
(1)	0	0	0		0		1	1	1		1	1
(2)	0	0	1		0		1	1	1		1	1
(3)	0	1	0		0		1	1	0		0	0
(4)	0	1	1		0		1	1	1		1	0
(5)	1	0	0		0		0	0	0		1	1
(6)	1	0	1		0		0	0	0		1	1
(7)	1	1	0		0		1	0	0		0	0

$$s_1 \wedge (s_2 \vee \neg s_1) \wedge (s_3 \vee \neg s_2)$$

	s_1	s_2	s_3	s_1	$\wedge$	$((s_2$	$\vee$	$\neg s_1)$	$\wedge$	$(s_3$	$\vee$	$\neg s_2))$
(1)	0	0	0		0		1	1	1		1	1
(2)	0	0	1		0		1	1	1		1	1
(3)	0	1	0		0		1	1	0		0	0
(4)	0	1	1		0		1	1	1		1	0
(5)	1	0	0		0		0	0	0		1	1
(6)	1	0	1		0		0	0	0		1	1
(7)	1	1	0		0		1	0	0		0	0
(8)	1	1	1		1		1	0	1		1	0

SAT Facts

- Brute force algorithm: **exponential complexity**:
- 2^n for n propositional symbol
- SAT is the first problem to be proven to be **NP-complete** [Cook 1971]
- SAT solves any decision problem in NP
- Modern SAT Solvers are arguably **efficient**, why?
- SAT expects an input in **Conjunctive Normal Form**

SAT Facts

- Brute force algorithm: **exponential complexity**:
- 2^n for n propositional symbol
- SAT is the first problem to be proven to be **NP-complete** [Cook 1971]

- SAT solves any decision problem in NP
- Modern SAT Solvers are arguably **efficient**, why?
- SAT expects an input in **Conjunctive Normal Form**

SAT Facts

- Brute force algorithm: **exponential complexity**:
- 2^n for n propositional symbol
- SAT is the first problem to be proven to be **NP-complete** [Cook 1971]

- SAT solves any decision problem in NP
- Modern SAT Solvers are arguably **efficient**, why?
- SAT expects an input in **Conjunctive Normal Form**

SAT Facts

- Brute force algorithm: **exponential complexity**:
- 2^n for n propositional symbol
- SAT is the first problem to be proven to be **NP-complete** [Cook 1971]

- SAT solves any decision problem in NP
- Modern SAT Solvers are arguably **efficient**, why?
- SAT expects an input in **Conjunctive Normal Form**

SAT Facts

- Brute force algorithm: **exponential complexity**:
- 2^n for n propositional symbol
- SAT is the first problem to be proven to be **NP-complete** [Cook 1971]

- SAT solves any decision problem in NP
- Modern SAT Solvers are arguably **efficient**, why?
- SAT expects an input in **Conjunctive Normal Form**

Recall (Tautological) Equivalence

$$w_1 \sim w_2 \quad \text{if and only if} \quad \forall \sigma.\ (\llbracket w_1 \rrbracket_\sigma = 1 \longleftrightarrow \llbracket w_2 \rrbracket_\sigma = 1)$$

Equisatisfiability

$$w_1 \sim_{SAT} w_2 \quad \text{if and only if} \quad \exists \sigma.\ \llbracket w_1 \rrbracket_\sigma = 1 \longleftrightarrow \exists \sigma.\ \llbracket w_2 \rrbracket_\sigma = 1$$

Equisatisfiability does not imply tautological equivalence!

- $w_1 := s_1 \wedge (s_1 \leftrightarrow s_2)$ and $w_2 := s_1$
- $w_1 \sim_{SAT} w_2$ but $w_1 \not\sim w_2$

Recall (Tautological) Equivalence

$$w_1 \sim w_2 \quad \text{if and only if} \quad \forall \sigma.\ (\llbracket w_1 \rrbracket_\sigma = 1 \longleftrightarrow \llbracket w_2 \rrbracket_\sigma = 1)$$

Equisatisfiability

$$w_1 \sim_{SAT} w_2 \quad \text{if and only if} \quad \exists \sigma.\ \llbracket w_1 \rrbracket_\sigma = 1 \longleftrightarrow \exists \sigma.\ \llbracket w_2 \rrbracket_\sigma = 1$$

Equisatisfiability does not imply tautological equivalence!

- $w_1 := s_1 \wedge (s_1 \leftrightarrow s_2)$ and $w_2 := s_1$
- $w_1 \sim_{SAT} w_2$ but $w_1 \not\sim w_2$

Recall (Tautological) Equivalence

$$w_1 \sim w_2 \quad \text{if and only if} \quad \forall \sigma.\ (\llbracket w_1 \rrbracket_\sigma = 1 \longleftrightarrow \llbracket w_2 \rrbracket_\sigma = 1)$$

Equisatisfiability

$$w_1 \sim_{SAT} w_2 \quad \text{if and only if} \quad \exists \sigma.\ \llbracket w_1 \rrbracket_\sigma = 1 \longleftrightarrow \exists \sigma.\ \llbracket w_2 \rrbracket_\sigma = 1$$

Equisatisfiability does not imply tautological equivalence!

- $w_1 := s_1 \wedge (s_1 \leftrightarrow s_2)$ and $w_2 := s_1$
- $w_1 \sim_{SAT} w_2$ but $w_1 \not\sim w_2$

Converting a wff w to an equivalent formula in CNF using De Morgan's Laws and distributivity may increase the number of logical operations (Boolean gates) **exponentially**.

Example

- $w_1 := (s_1 \wedge s_2) \vee (s_3 \wedge s_4)$, by distributivity
- $w_2 := (s_1 \vee s_3) \wedge (s_1 \vee s_4) \wedge (s_2 \vee s_3) \wedge (s_2 \vee s_4)$
- Add extra pairs to w_1, $(s_1 \wedge s_2) \vee (s_3 \wedge s_4) \vee (s_5 \wedge s_6) \ldots$
- The number of the involved logical operations is exponential

Converting a wff w to an equivalent formula in CNF using De Morgan's Laws and distributivity may increase the number of logical operations (Boolean gates) **exponentially**.

Example

- $w_1 := (s_1 \wedge s_2) \vee (s_3 \wedge s_4)$, by distributivity
- $w_2 := (s_1 \vee s_3) \wedge (s_1 \vee s_4) \wedge (s_2 \vee s_3) \wedge (s_2 \vee s_4)$
- Add extra pairs to w_1, $(s_1 \wedge s_2) \vee (s_3 \wedge s_4) \vee (s_5 \wedge s_6) \ldots$
- The number of the involved logical operations is exponential

Converting a wff w to an equivalent formula in CNF using De Morgan's Laws and distributivity may increase the number of logical operations (Boolean gates) **exponentially**.

Example

- $w_1 := (s_1 \wedge s_2) \vee (s_3 \wedge s_4)$, by distributivity
- $w_2 := (s_1 \vee s_3) \wedge (s_1 \vee s_4) \wedge (s_2 \vee s_3) \wedge (s_2 \vee s_4)$
- Add extra pairs to w_1, $(s_1 \wedge s_2) \vee (s_3 \wedge s_4) \vee (s_5 \wedge s_6) \ldots$
- The number of the involved logical operations is exponential

Converting a wff w to an equivalent formula in CNF using De Morgan's Laws and distributivity may increase the number of logical operations (Boolean gates) **exponentially**.

Example

- $w_1 := (s_1 \wedge s_2) \vee (s_3 \wedge s_4)$, by distributivity
- $w_2 := (s_1 \vee s_3) \wedge (s_1 \vee s_4) \wedge (s_2 \vee s_3) \wedge (s_2 \vee s_4)$
- Add extra pairs to w_1, $(s_1 \wedge s_2) \vee (s_3 \wedge s_4) \vee (s_5 \wedge s_6) \ldots$
- The number of the involved logical operations is exponential

Idea: Converting a w by adding new propositional variables and **substitute** for nested operations.

Example

$$\underbrace{\underbrace{(s_1 \wedge s_2)}_{p_1} \vee \underbrace{(s_3 \wedge s_4)}_{p_2}}_{p_3}$$

- $p_1 \leftrightarrow (s_1 \wedge s_2)$
- $p_2 \leftrightarrow (s_3 \wedge s_4)$
- $p_3 \leftrightarrow p_1 \vee p_2$
- CNF: $(p_1 \leftrightarrow (s_1 \wedge s_2)) \wedge (p_2 \leftrightarrow (s_3 \wedge s_4)) \wedge (p_3 \leftrightarrow p_1 \vee p_2) \wedge p_3$

Idea: Converting a w by adding new propositional variables and **substitute** for nested operations.

Example

$$\underbrace{\underbrace{(s_1 \wedge s_2)}_{p_1} \vee \underbrace{(s_3 \wedge s_4)}_{p_2}}_{p_3}$$

- $p_1 \leftrightarrow (s_1 \wedge s_2)$
- $p_2 \leftrightarrow (s_3 \wedge s_4)$
- $p_3 \leftrightarrow p_1 \vee p_2$
- CNF: $(p_1 \leftrightarrow (s_1 \wedge s_2)) \wedge (p_2 \leftrightarrow (s_3 \wedge s_4)) \wedge (p_3 \leftrightarrow p_1 \vee p_2) \wedge p_3$

Converting to CNF

Tseytin Transformation [Tseytin 1970]

- $p \leftrightarrow (s_1 \circ s_2)$ has at most 3 clauses for any $\circ \in \{\neg, \wedge, \vee, \bar{\wedge}, \bar{\vee}\}$
- For n logical operation, the increase is **linear** $O(n)$
- Drawback: the number of propositional variables increases (linearly)!

Converting to CNF

Tseytin Transformation [Tseytin 1970]

- $p \leftrightarrow (s_1 \circ s_2)$ has at most 3 clauses for any $\circ \in \{\neg, \wedge, \vee, \bar{\wedge}, \bar{\vee}\}$
- For n logical operation, the increase is **linear** $O(n)$
- Drawback: the number of propositional variables increases (linearly)!

Converting to CNF

Tseytin Transformation [Tseytin 1970]

- $p \leftrightarrow (s_1 \circ s_2)$ has at most 3 clauses for any $\circ \in \{\neg, \wedge, \vee, \bar{\wedge}, \bar{\vee}\}$
- For n logical operation, the increase is **linear** $O(n)$
- Drawback: the number of propositional variables increases (linearly)!

DiMaCS Standard

- Each propositional variable is represented by a positive integer
- A negative integer refers to negative occurrences
- Clauses are given as sequences of integers separated by spaces
- A 0 terminates the clause

Example:

- $(s_1 \vee \neg s_3) \wedge (\neg s_2 \vee s_3 \vee s_4)$
- 1 -3 0 -2 3 4 0

- Each propositional variable is represented by a positive integer
- A negative integer refers to negative occurrences
- Clauses are given as sequences of integers separated by spaces
- A 0 terminates the clause

Example:

- $(s_1 \vee \neg s_3) \wedge (\neg s_2 \vee s_3 \vee s_4)$
- 1 -3 0 -2 3 4 0

Outline

- **Literal**: propositional symbol (atomic formula) or its negation
- **Clause**: disjunction of one or more literals
- **Conjunctive Normal Form** (CNF): conjunction of clauses
- **Positive** Occurrence: if the symbol occurs unnegated in a clause
- **Negative** Occurrence: if the symbol occurs negated in a clause

$$(s_1 \vee \neg s_3) \wedge (\neg s_2 \vee s_3 \vee s_4)$$

- **Literal**: propositional symbol (atomic formula) or its negation
- **Clause**: disjunction of one or more literals
- **Conjunctive Normal Form** (CNF): conjunction of clauses
- **Positive** Occurrence: if the symbol occurs unnegated in a clause
- **Negative** Occurrence: if the symbol occurs negated in a clause

$$(s_1 \vee \neg s_3) \wedge (\neg s_2 \vee s_3 \vee s_4)$$

- **Literal**: propositional symbol (atomic formula) or its negation
- **Clause**: disjunction of one or more literals
- **Conjunctive Normal Form** (CNF): conjunction of clauses
- **Positive** Occurrence: if the symbol occurs unnegated in a clause
- **Negative** Occurrence: if the symbol occurs negated in a clause

$$(s_1 \vee \neg s_3) \wedge (\neg s_2 \vee s_3 \vee s_4)$$

- **Literal**: propositional symbol (atomic formula) or its negation
- **Clause**: disjunction of one or more literals
- **Conjunctive Normal Form** (CNF): conjunction of clauses
- **Positive** Occurrence: if the symbol occurs unnegated in a clause
- **Negative** Occurrence: if the symbol occurs negated in a clause

$$(s_1 \vee \neg s_3) \wedge (\neg s_2 \vee s_3 \vee s_4)$$

- **Literal**: propositional symbol (atomic formula) or its negation
- **Clause**: disjunction of one or more literals
- **Conjunctive Normal Form** (CNF): conjunction of clauses
- **Positive** Occurrence: if the symbol occurs unnegated in a clause
- **Negative** Occurrence: if the symbol occurs negated in a clause

$$(s_1 \vee \neg s_3) \wedge (\neg s_2 \vee s_3 \vee s_4)$$

Satisfiability-Preserving Transformations

- **Pure literal rule** or affirmative-negative rule
- **Unit propagation** or 1-literal rule
- **Resolution rule** or rule for eliminating literals (atomic formulas)

DP Algorithm

Iteratively apply the rules till **reducing the problem to a unique clause**

- if the clause has the form $s \wedge \neg s$ the problem is unsat
- otherwise, the problem is sat

Satisfiability-Preserving Transformations

- **Pure literal rule** or affirmative-negative rule
- **Unit propagation** or 1-literal rule
- **Resolution rule** or rule for eliminating literals (atomic formulas)

DP Algorithm

Iteratively apply the rules till **reducing the problem to a unique clause**

- if the clause has the form $s \wedge \neg s$ the problem is unsat
- otherwise, the problem is sat

Pure literal i.e. appears **only positively** or **only negatively**, ℓ say

Delete all clauses containing that literal

- A clause containing ℓ has the form $\ell \vee w$
- $(\ell \leftrightarrow 1) \wedge (\ell \vee w_1) \wedge w_2 \quad \sim_{SAT} \quad w_2$
- Repeat till no more clauses contain ℓ

➻ Augment σ such that $[\![\ell]\!]_\sigma = 1$

Nota: not used dynamically while solving as expensive to detect.

Example of Preprocessing with Pure Literal Rule

(1 and 7)	($\bar{2}$)	($\bar{4}$ and $\bar{5}$)
$1 \vee 2$	$1 \vee 2$	$1 \vee 2$
$1 \vee 3 \vee 8$	$1 \vee 3 \vee 8$	$1 \vee 3 \vee 8$
$\bar{2} \vee \bar{3} \vee 4$	$\bar{2} \vee \bar{3} \vee 4$	$\bar{2} \vee \bar{3} \vee 4$
$\bar{4} \vee 5 \vee 7$	$\bar{4} \vee 5 \vee 7$	$\bar{4} \vee 5 \vee 7$
$\bar{4} \vee 6 \vee 8$	$\bar{4} \vee 6 \vee 8$	$\bar{4} \vee 6 \vee 8$
$\bar{5} \vee \bar{6}$	$\bar{5} \vee \bar{6}$	$\bar{5} \vee \bar{6}$
$7 \vee \bar{8}$	$7 \vee \bar{8}$	$7 \vee \bar{8}$
$7 \vee \bar{9} \vee 10$	$7 \vee \bar{9} \vee 10$	$7 \vee \bar{9} \vee 10$

↦ SAT! $\sigma = \{1, 7, \bar{2}, \bar{4}, \bar{5}\}$

Unit clause is a clause with only **one literal**, ℓ say

Remove all the clauses containing ℓ

- A clause containing ℓ has the form $\ell \vee w$
- $(\ell \leftrightarrow 1) \wedge (\ell \vee w) \quad \sim_{SAT} \quad 1$
- Repeat till no more clauses contain ℓ

Remove all instances of $\neg\ell$ from all the clauses

- A clause containing $\neg\ell$ has the form $\ell \vee w$
- $[\![\neg\ell]\!]_\sigma = 0$
- $(\ell \leftrightarrow 1) \wedge (\neg\ell \vee w) \quad \sim_{SAT} \quad w$
- Repeat till no more clauses contain $\neg\ell$

↠ Augment σ such that $[\![\ell]\!]_\sigma = 1$

Boolean Constraint Propagation (BCP)

- Unit propagation is a typical instance of BCP
- Consumes the most significant runtime of modern solvers

Several **heuristics** proved efficient

- Counter-based (GRASP) [Marques-Silva, Sakallah, 1996]
- Head/Tail lists (SATO) [Zhang, Stickel, 1996]
- 2-literal watching (Chaff) [Moskewicz et al. 2001]

Counter-Based Algorithm for BCP

- Denote by $|C|$ the total number of literals in C
- Each clause C has two counters:
  - $C(\ell = 0) := \#_\ell$ such that $[\![\ell]\!]_\sigma = 0$
  - $C(\ell = 1) := \#_\ell$ such that $[\![\ell]\!]_\sigma = 0$
- Each variable s has two lists of clauses:
  - P_s: set of clauses where the variable occurs positively
  - N_s: set of clauses where the variable occurs negatively

If s is assigned, $C(\ell = 0)$ and $C(\ell = 1)$ for all C in $P_s \cup N_s$ are updated

- If $C(\ell = 0) = |C|$ then C is a **conflicting clause** (more later)
- If $C(\ell = 0) = -1 + |C|$ and $C(\ell = 1) = 0$ then it is a **unit clause**

Counter-Based Algorithm for BCP

- Denote by $|C|$ the total number of literals in C
- Each clause C has two counters:
  - $C(\ell = 0) := \#_\ell$ such that $[\![\ell]\!]_\sigma = 0$
  - $C(\ell = 1) := \#_\ell$ such that $[\![\ell]\!]_\sigma = 0$
- Each variable s has two lists of clauses:
  - P_s: set of clauses where the variable occurs positively
  - N_s: set of clauses where the variable occurs negatively

If s is assigned, $C(\ell = 0)$ and $C(\ell = 1)$ for all C in $P_s \cup N_s$ are updated

- If $C(\ell = 0) = |C|$ then C is a **conflicting clause** (more later)
- If $C(\ell = 0) = -1 + |C|$ and $C(\ell = 1) = 0$ then it is a **unit clause**

Counter-Based Algorithm for BCP

- Denote by $|C|$ the total number of literals in C
- Each clause C has two counters:
  - $C(\ell = 0) := \#_\ell$ such that $[\![\ell]\!]_\sigma = 0$
  - $C(\ell = 1) := \#_\ell$ such that $[\![\ell]\!]_\sigma = 0$

- Each variable s has two lists of clauses:
  - P_s: set of clauses where the variable occurs positively
  - N_s: set of clauses where the variable occurs negatively

If s is assigned, $C(\ell = 0)$ and $C(\ell = 1)$ for all C in $P_s \cup N_s$ are updated

- If $C(\ell = 0) = |C|$ then C is a **conflicting clause** (more later)
- If $C(\ell = 0) = -1 + |C|$ and $C(\ell = 1) = 0$ then it is a **unit clause**

Counter-Based Algorithm for BCP

- Denote by $|C|$ the total number of literals in C
- Each clause C has two counters:
  - $C(\ell = 0) := \#_\ell$ such that $[\![\ell]\!]_\sigma = 0$
  - $C(\ell = 1) := \#_\ell$ such that $[\![\ell]\!]_\sigma = 0$

- Each variable s has two lists of clauses:
  - P_s: set of clauses where the variable occurs positively
  - N_s: set of clauses where the variable occurs negatively

If s is assigned, $C(\ell = 0)$ and $C(\ell = 1)$ for all C in $P_s \cup N_s$ are updated

- If $C(\ell = 0) = |C|$ then C is a **conflicting clause** (more later)
- If $C(\ell = 0) = -1 + |C|$ and $C(\ell = 1) = 0$ then it is a **unit clause**

Counter-Based Algorithm for BCP

- Denote by $|C|$ the total number of literals in C
- Each clause C has two counters:
  - $C(\ell = 0) := \#_\ell$ such that $[\![\ell]\!]_\sigma = 0$
  - $C(\ell = 1) := \#_\ell$ such that $[\![\ell]\!]_\sigma = 0$

- Each variable s has two lists of clauses:
  - P_s: set of clauses where the variable occurs positively
  - N_s: set of clauses where the variable occurs negatively

If s is assigned, $C(\ell = 0)$ and $C(\ell = 1)$ for all C in $P_s \cup N_s$ are updated

- If $C(\ell = 0) = |C|$ then C is a **conflicting clause** (more later)
- If $C(\ell = 0) = -1 + |C|$ and $C(\ell = 1) = 0$ then it is a **unit clause**

Counter-Based Algorithm for BCP

- Denote by $|C|$ the total number of literals in C
- Each clause C has two counters:
  - $C(\ell = 0) := \#_\ell$ such that $[\![\ell]\!]_\sigma = 0$
  - $C(\ell = 1) := \#_\ell$ such that $[\![\ell]\!]_\sigma = 0$

- Each variable s has two lists of clauses:
  - P_s: set of clauses where the variable occurs positively
  - N_s: set of clauses where the variable occurs negatively

If s is assigned, $C(\ell = 0)$ and $C(\ell = 1)$ for all C in $P_s \cup N_s$ are updated

- If $C(\ell = 0) = |C|$ then C is a **conflicting clause** (more later)
- If $C(\ell = 0) = -1 + |C|$ and $C(\ell = 1) = 0$ then it is a **unit clause**

Counter-Based Algorithm for BCP

- Denote by $|C|$ the total number of literals in C
- Each clause C has two counters:
  - $C(\ell = 0) := \#_\ell$ such that $[\![\ell]\!]_\sigma = 0$
  - $C(\ell = 1) := \#_\ell$ such that $[\![\ell]\!]_\sigma = 0$

- Each variable s has two lists of clauses:
  - P_s: set of clauses where the variable occurs positively
  - N_s: set of clauses where the variable occurs negatively

If s is assigned, $C(\ell = 0)$ and $C(\ell = 1)$ for all C in $P_s \cup N_s$ are updated

- If $C(\ell = 0) = |C|$ then C is a **conflicting clause** (more later)
- If $C(\ell = 0) = -1 + |C|$ and $C(\ell = 1) = 0$ then it is a **unit clause**

Counter-Based Algorithm for BCP

- Denote by $|C|$ the total number of literals in C
- Each clause C has two counters:
  - $C(\ell = 0) := \#_\ell$ such that $[\![\ell]\!]_\sigma = 0$
  - $C(\ell = 1) := \#_\ell$ such that $[\![\ell]\!]_\sigma = 0$

- Each variable s has two lists of clauses:
  - P_s: set of clauses where the variable occurs positively
  - N_s: set of clauses where the variable occurs negatively

If s is assigned, $C(\ell = 0)$ and $C(\ell = 1)$ for all C in $P_s \cup N_s$ are updated

- If $C(\ell = 0) = |C|$ then C is a **conflicting clause** (more later)
- If $C(\ell = 0) = -1 + |C|$ and $C(\ell = 1) = 0$ then it is a **unit clause**

Empirical fact: Find and perform literal prorogation is **expensive**

Watched Literals

- For every clause, pick two literals to be **watched**
- If a literal is assigned, check only those clauses in which the literal is watched
- When inspecting, if the propagation is not triggered, pick a new literal to watch

Empirical fact: Find and perform literal prorogation is **expensive**

Watched Literals

- For every clause, pick two literals to be **watched**
- If a literal is assigned, check only those clauses in which the literal is watched
- When inspecting, if the propagation is not triggered, pick a new literal to watch

Empirical fact: Find and perform literal prorogation is **expensive**

Watched Literals

- For every clause, pick two literals to be **watched**
- If a literal is assigned, check only those clauses in which the literal is watched
- When inspecting, if the propagation is not triggered, pick a new literal to watch

Empirical fact: Find and perform literal prorogation is **expensive**

Watched Literals

- For every clause, pick two literals to be **watched**
- If a literal is assigned, check only those clauses in which the literal is watched
- When inspecting, if the propagation is not triggered, pick a new literal to watch

If s does not appear in the wff w, then

$$(s \vee a) \wedge (\neg s \vee b) \wedge w \quad \sim_{SAT} \quad \underbrace{(a \vee b)}_{\textbf{resolvent}} \wedge w$$

$$\bigwedge_i (s \vee a_i) \wedge \bigwedge_j (\neg s \vee b_j) \wedge w \quad \sim_{SAT} \quad \left(\bigwedge_i a_i \vee \bigwedge_j b_j \right) \wedge w$$

$$\left(\bigwedge_i a_i \vee \bigwedge_j b_j \right) \wedge w \quad \sim \quad \left(\bigwedge_i \bigwedge_j (a_i \vee b_j) \right) \wedge w$$

Resolution Rule

- Simplifies by s. **No explicit assignment** for s!

If s does not appear in the wff w, then

$$(s \vee a) \wedge (\neg s \vee b) \wedge w \quad \sim_{SAT} \quad \underbrace{(a \vee b)}_{\text{resolvent}} \wedge w$$

$$\bigwedge_i (s \vee a_i) \wedge \bigwedge_j (\neg s \vee b_j) \wedge w \quad \sim_{SAT} \quad \left(\bigwedge_i a_i \vee \bigwedge_j b_j \right) \wedge w$$

$$\left(\bigwedge_i a_i \vee \bigwedge_j b_j \right) \wedge w \quad \sim \quad \left(\bigwedge_i \bigwedge_j (a_i \vee b_j) \right) \wedge w$$

Resolution Rule

- Simplifies by s. **No explicit assignment** for s!

If s does not appear in the wff w, then

$$(s \vee a) \wedge (\neg s \vee b) \wedge w \quad \sim_{SAT} \quad \underbrace{(a \vee b)}_{\text{resolvent}} \wedge w$$

$$\bigwedge_i (s \vee a_i) \wedge \bigwedge_j (\neg s \vee b_j) \wedge w \quad \sim_{SAT} \quad \left(\bigwedge_i a_i \vee \bigwedge_j b_j \right) \wedge w$$

$$\left(\bigwedge_i a_i \vee \bigwedge_j b_j \right) \wedge w \quad \sim \quad \left(\bigwedge_i \bigwedge_j (a_i \vee b_j) \right) \wedge w$$

Resolution Rule

- Simplifies by s. **No explicit assignment** for s!

If s does not appear in the wff w, then

$$(s \vee a) \wedge (\neg s \vee b) \wedge w \quad \sim_{SAT} \quad \underbrace{(a \vee b)}_{\text{resolvent}} \wedge w$$

$$\bigwedge_i (s \vee a_i) \wedge \bigwedge_j (\neg s \vee b_j) \wedge w \quad \sim_{SAT} \quad \left(\bigwedge_i a_i \vee \bigwedge_j b_j \right) \wedge w$$

$$\left(\bigwedge_i a_i \vee \bigwedge_j b_j \right) \wedge w \quad \sim \quad \left(\bigwedge_i \bigwedge_j (a_i \vee b_j) \right) \wedge w$$

Resolution Rule

- Simplifies by s. **No explicit assignment** for s!

Splitting (or Branching) Rule

Davis-Logemann-Loveland 1962

Memory Consumption The resolution rule can cause a **quadratic expansion** every time it is applied exhausting rapidly the available memory

The DLL algorithm replaces the resolution rule with a **Splitting Rule**

1. Simplify by Unit Propagation and Pure Literals
2. **Recursively** pick a *hard core* variable s
3. Test if $(w \wedge s)$ is SAT
4. Otherwise return the result for $(w \wedge \neg s)$

Splitting (or Branching) Rule

Davis-Logemann-Loveland 1962

Memory Consumption The resolution rule can cause a **quadratic expansion** every time it is applied exhausting rapidly the available memory

The DLL algorithm replaces the resolution rule with a **Splitting Rule**

1. Simplify by Unit Propagation and Pure Literals
2. **Recursively** pick a *hard core* variable s
3. Test if $(w \wedge s)$ is SAT
4. Otherwise return the result for $(w \wedge \neg s)$

Splitting (or Branching) Rule

Davis-Logemann-Loveland 1962

Memory Consumption The resolution rule can cause a **quadratic expansion** every time it is applied exhausting rapidly the available memory

The DLL algorithm replaces the resolution rule with a **Splitting Rule**

1. Simplify by Unit Propagation and Pure Literals
2. **Recursively** pick a *hard core* variable s
3. Test if $(w \wedge s)$ is SAT
4. Otherwise return the result for $(w \wedge \neg s)$

Splitting (or Branching) Rule

Davis-Logemann-Loveland 1962

Memory Consumption The resolution rule can cause a **quadratic expansion** every time it is applied exhausting rapidly the available memory

The DLL algorithm replaces the resolution rule with a **Splitting Rule**

1. Simplify by Unit Propagation and Pure Literals
2. **Recursively** pick a *hard core* variable s
3. Test if $(w \wedge s)$ is SAT
4. Otherwise return the result for $(w \wedge \neg s)$

Splitting (or Branching) Rule

Davis-Logemann-Loveland 1962

Memory Consumption The resolution rule can cause a **quadratic expansion** every time it is applied exhausting rapidly the available memory

The DLL algorithm replaces the resolution rule with a **Splitting Rule**

1. Simplify by Unit Propagation and Pure Literals
2. **Recursively** pick a *hard core* variable s
3. Test if $(w \wedge s)$ is SAT
4. Otherwise return the result for $(w \wedge \neg s)$

Splitting (or Branching) Rule

Davis-Logemann-Loveland 1962

Memory Consumption The resolution rule can cause a **quadratic expansion** every time it is applied exhausting rapidly the available memory

The DLL algorithm replaces the resolution rule with a **Splitting Rule**

1. Simplify by Unit Propagation and Pure Literals
2. **Recursively** pick a *hard core* variable s
3. Test if $(w \wedge s)$ is SAT
4. Otherwise return the result for $(w \wedge \neg s)$

DPLL Modern Decision Procedure

Zhang, Malik, 2002

```
status = preprocess();
if (status!=UNKNOWN) return status;
while(true) {
    decide_next_branch();
    while (true) {
        status = deduce();
        if (status == CONFLICT) {
            blevel = analyze_conflict();
            if (blevel == 0)
                return UNSATISFIABLE;
            else backtrack(blevel);
        } else if (status == SATISFIABLE)
            return SATISFIABLE;
        else break;
    }
}
```

Which variable to branch with ?

Greedy Algorithms

- Exploit the statistics of the clause database
- Estimate the branching effect on each variable (cost function)
  - Ex1: Generate the largest number of implications
  - Ex2: Satisfy most clauses

Heuristcs

- Maximum occurences on minimum sized clauses (MOM)
- **Literal Count Heuristcs**

Dynamic Largest Individual Sum (DLIS) [Marques-Silva, 1999]

- Counts the number of unresolved clauses for each free variable
- Chooses the variable with the largest number
- State-dependent (recalculated each time before branching)

Variable State Independent Decaying Sum

VSIDS. [Moskewicz et al., 2001]

- Keeps two scores for each variable
- (# of pos occurences, # of neg occurences)
- Increases the score of a variable by a constant if it appears in a learned conflicting-clause
- Periodically, all the scores are divided by a constant
- Branch with the variable with the highest combined score

↣ Cheap to maintain (State Independent)

↣ Captures the **recently active** variables

Conflict-Driven Clause Learning (CDCL)

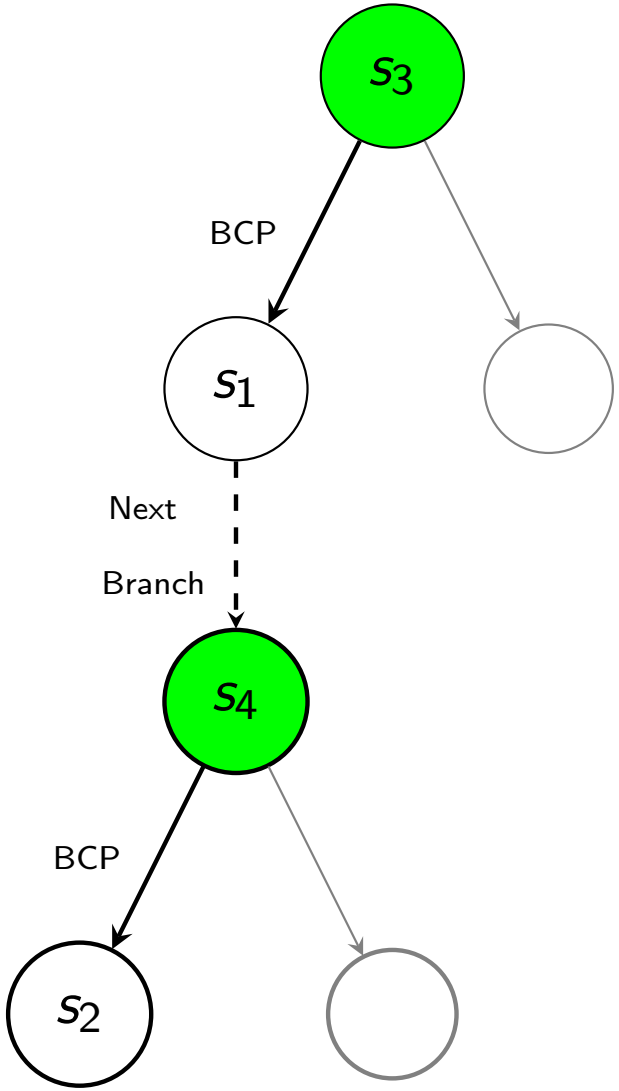
Marques-Silva,Sakallah,1996 and Bayardo,Schrag,1997

Modern SAT solvers essentially implements a

backtracking-based search algorithm

Two graphs are built iteratively

- Search graph
- Implication graph


s_3
BCP
s_1
Next
Branch
s_4
BCP
s_2

Conflicting Clause: a clause with all its literals assigned to 0

Backtrack when a conflict occurs

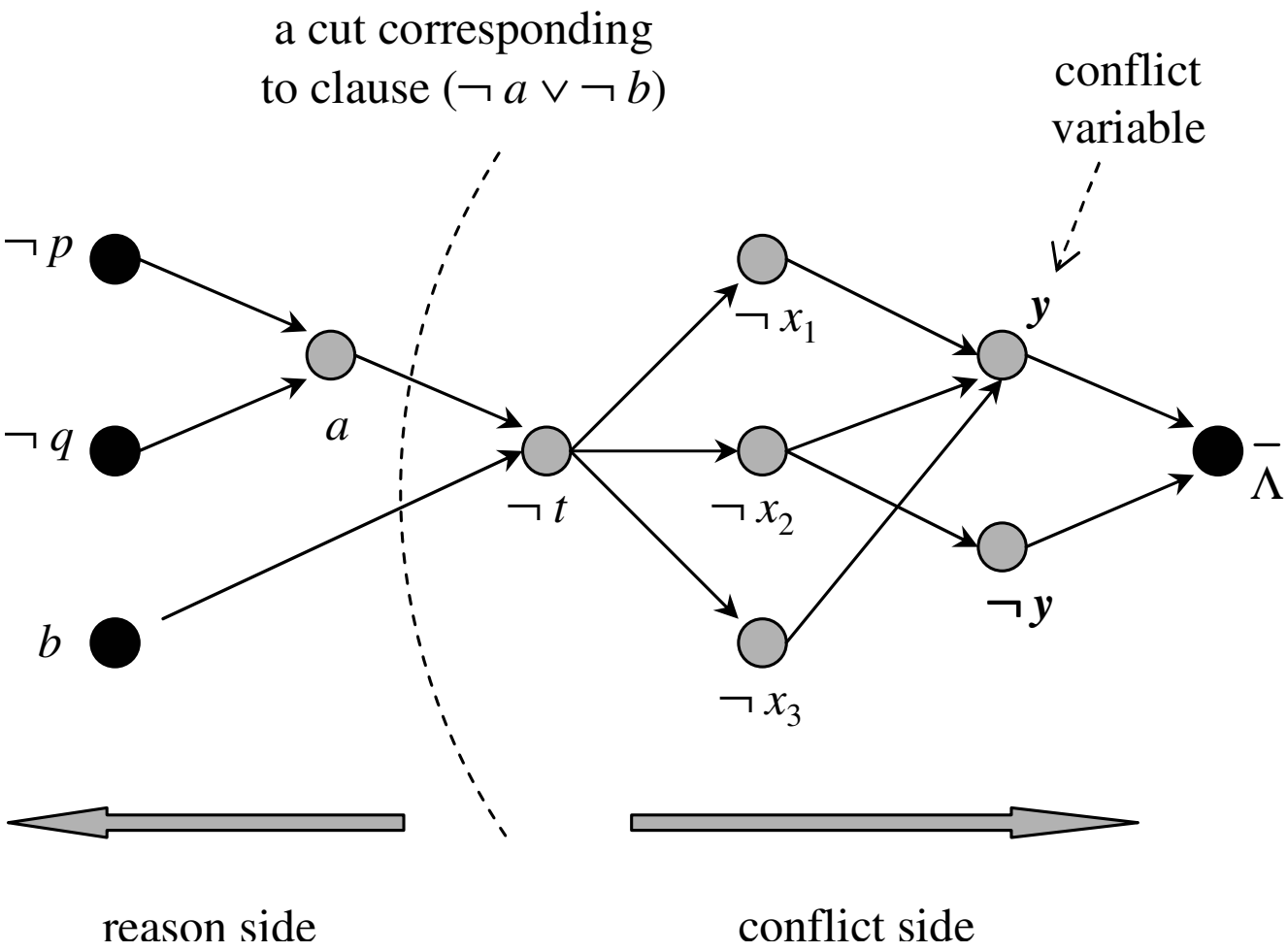
- No future decisions are possible
- Backtrack to the **immediately previous decision** made
- Flip the assignement and continue
- If decision level 0 reached, return unsat

Conflicting Clause: a clause with all its literals assigned to 0

Backtrack when a conflict occurs

- No future decisions are possible
- Backtrack to the **immediately previous decision** made
- Flip the assignement and continue
- If decision level 0 reached, return unsat

No incident edges for decision nodes!



DPLL: Backjump and Learn

Backjump

- Jump to a past decision that caused the conflict
- (not necessarily the latest like in backtracking)
- Not unique in general (heuristics)

Learn

- Add a new clause to avoid reaching the same conflict again
- Not unique in general (heuristics)

Example: $\bar{1} \vee 2,\ 3 \vee 4,\ \bar{5} \vee \bar{6},\ \bar{2} \vee \bar{5} \vee 6$ (Satisfiable)

DPLL: Backjump and Learn

Backjump

- Jump to a past decision that caused the conflict
- (not necessarily the latest like in backtracking)
- Not unique in general (heuristics)

Learn

- Add a new clause to avoid reaching the same conflict again
- Not unique in general (heuristics)

Example: $\bar{1} \vee 2,\ 3 \vee 4,\ \bar{5} \vee \bar{6},\ \bar{2} \vee \bar{5} \vee 6$ (Satisfiable)

DPLL: Backjump and Learn

Backjump

- Jump to a past decision that caused the conflict
- (not necessarily the latest like in backtracking)
- Not unique in general (heuristics)

Learn

- Add a new clause to avoid reaching the same conflict again
- Not unique in general (heuristics)

Example: $\bar{1} \vee 2,\ 3 \vee 4,\ \bar{5} \vee \bar{6},\ \bar{2} \vee \bar{5} \vee 6$ (Satisfiable)

DPLL: Forget and Restart

Mostly used in SMT Solvers

Forget

- When **too much clauses** are learned
- heuristics: those not frequently used by literal propagations

Restart

- If stuck, **restart** from the beginning (extreme backjumping)
- **Keep the learned clauses**

DPLL: Forget and Restart

Mostly used in SMT Solvers

Forget

- When **too much clauses** are learned
- heuristics: those not frequently used by literal propagations

Restart

- If stuck, **restart** from the beginning (extreme backjumping)
- **Keep the learned clauses**

SAT Problem

- Equisatisfiability
- SAT for proving tautological implications/equivalences
- CNF transformation

CDCL-DPLL Algorithm

- Unit Propagation
- Pure Literal
- Resolution/Splitting/Conflict Learning

Alternative Approaches: Stalmarck's method

- Designed to detect unsat formulas
- Works on conjunctions of triplet of the form $p \leftrightarrow q \wedge r$
- The transformation works like CNF, but we leave them as equivalences
- Saturation by deduction is performed till reaching a contradiction $(0 \leftrightarrow 1)$
- Otherwise, Split for each variable to deduce new equivalences
- Continue with higher level of saturations

Saturation Rules (example)

Works on triplet of the form $p \leftrightarrow q \wedge r$

- If $r \leftrightarrow 1$, then $p \leftrightarrow q$
- If $p \leftrightarrow 1$, then $q \leftrightarrow 1$ and $r \leftrightarrow 1$
- If $q \leftrightarrow 0$, then $p \leftrightarrow 0$
- If $q \leftrightarrow r$, then $p \leftrightarrow q$ and $p \leftrightarrow r$
- If $p \leftrightarrow \neg q$, then $q \leftrightarrow 1$ and $r \leftrightarrow 0$