Solvers Principles and Architecture (SPA)

Part 1

Anatomy of SAT Solvers

Master Sciences Informatique (Sif)
September, 2019
Rennes

Khalil Ghorbal
khalil.ghorbal@inria.fr

Outline

Logic in a Nut Shell

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!

One can interpret all expressions in W over the set $\{0, 1\}$ by giving an interpretation of the basic constructors that generate W.
A symbol s can be either 0 or 1.

s	$\neg$
0	1
1	0

One can interpret all expressions in W over the set $\{0, 1\}$ by giving an interpretation of the basic constructors that generate W.
A symbol s can be either 0 or 1.

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

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

- 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:

$$\forall \sigma. \left([\![w]\!]_\sigma = 0\right) \quad .$$

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:

$$\forall \sigma. \ ([\![w]\!]_\sigma = 0) \quad .$$

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:

$$\forall \sigma.\ ([\![w]\!]_\sigma = 0) \quad .$$

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:

$$\forall \sigma. \ ([\![w]\!]_\sigma = 0) \quad .$$

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:

$$\forall \sigma. \left([\![w]\!]_\sigma = 0\right) \quad .$$

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)

$$w_1, \ldots, w_n \models w \quad \text{if and only if} \quad \forall \sigma.\ (\llbracket \wedge_i w_i \rrbracket_\sigma = 1 \longrightarrow \llbracket w \rrbracket_\sigma = 1)$$

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)

$$w_1, \ldots, w_n \models w \quad \text{if and only if} \quad \forall \sigma.\ (\llbracket \wedge_i w_i \rrbracket_\sigma = 1 \longrightarrow \llbracket w \rrbracket_\sigma = 1)$$

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)

$$w_1, \ldots, w_n \models w \quad \text{if and only if} \quad \forall\sigma.\ (\llbracket \wedge_i w_i \rrbracket_\sigma = 1 \longrightarrow \llbracket w \rrbracket_\sigma = 1)$$

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)

$$w_1, \ldots, w_n \models w \quad \text{if and only if} \quad \forall \sigma.\ (\llbracket \wedge_i w_i \rrbracket_\sigma = 1 \longrightarrow \llbracket w \rrbracket_\sigma = 1)$$

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

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

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

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

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

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

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

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

$$w_1 \sim w_2 \quad \text{if and only if} \quad \forall \sigma.\ ([\![w_1]\!]_\sigma = 1 \longleftrightarrow [\![w_2]\!]_\sigma = 1)$$

Equisatisfiability

$$w_1 \sim_{SAT} w_2 \quad \text{if and only if} \quad (\exists \sigma.\ [\![w_1]\!]_\sigma = 1) \longleftrightarrow (\exists \sigma.\ [\![w_2]\!]_\sigma = 1)$$

Equisatisfiability does not imply tautological equivalence!

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

- **Literal**: propositional symbol (atomic expression) or its negation
- **Clause**: disjunction of one or more literals
- **Positive** Occurrence: if the symbol occurs unnegated in a clause
- **Negative** Occurrence: if the symbol occurs negated in a clause

- **Literal**: propositional symbol (atomic expression) or its negation
- **Clause**: disjunction of one or more literals
- **Positive** Occurrence: if the symbol occurs unnegated in a clause
- **Negative** Occurrence: if the symbol occurs negated in a clause

An expression w is in CNF if and only if it has the form

$$w = \bigwedge_i \bigvee_j \ell_{ij}$$

- Each ℓ_{ij} is a literal
- Thus, a CNF is a conjunction of clauses

Example: $\underbrace{(s_1 \vee \neg s_3)}_{c_1} \wedge \underbrace{(\neg s_2 \vee s_3 \vee s_4)}_{c_2}$

- 4 variable symbols: s_1, s_2, s_3, and s_4
- clause c_1 (resp. c_2) has 2 (resp. 3) literals
- s_3 is negative in c_1 and positive in c_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
- $w_1 \sim w_2 := (s_1 \vee s_3) \wedge (s_1 \vee s_4) \wedge (s_2 \vee s_3) \wedge (s_2 \vee s_4)$ (2^2 clauses)
- $w_1 := (s_1 \wedge s_2) \vee (s_3 \wedge s_4) \vee (s_5 \wedge s_6) \cdots \vee (s_n \wedge s_{n+1})$
- Now $w_1 \sim w_2$, and w_2 in CNF, but w_2 has 2^{n-1} clauses!
- We seek to avoid such exponential cost for the CNF reduction

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
- $w_1 \sim w_2 := (s_1 \vee s_3) \wedge (s_1 \vee s_4) \wedge (s_2 \vee s_3) \wedge (s_2 \vee s_4) \quad (2^2 \text{ clauses})$
- $w_1 := (s_1 \wedge s_2) \vee (s_3 \wedge s_4) \vee (s_5 \wedge s_6) \cdots \vee (s_n \wedge s_{n+1})$
- Now $w_1 \sim w_2$, and w_2 in CNF, but w_2 has 2^{n-1} clauses!
- We seek to avoid such exponential cost for the CNF reduction

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
- $w_1 \sim w_2 := (s_1 \vee s_3) \wedge (s_1 \vee s_4) \wedge (s_2 \vee s_3) \wedge (s_2 \vee s_4) \quad (2^2 \text{ clauses})$
- $w_1 := (s_1 \wedge s_2) \vee (s_3 \wedge s_4) \vee (s_5 \wedge s_6) \cdots \vee (s_n \wedge s_{n+1})$
- Now $w_1 \sim w_2$, and w_2 in CNF, but w_2 has 2^{n-1} clauses!
- We seek to avoid such exponential cost for the CNF reduction

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
- $w_1 \sim w_2 := (s_1 \vee s_3) \wedge (s_1 \vee s_4) \wedge (s_2 \vee s_3) \wedge (s_2 \vee s_4) \quad (2^2$ clauses)
- $w_1 := (s_1 \wedge s_2) \vee (s_3 \wedge s_4) \vee (s_5 \wedge s_6) \cdots \vee (s_n \wedge s_{n+1})$
- Now $w_1 \sim w_2$, and w_2 in CNF, but w_2 has 2^{n-1} clauses!
- We seek to avoid such exponential cost for the CNF reduction

Trick: Converting an expression by **adding** new propositional variables and **substituting** for nested operations. We avoid the exponential cost at the price of losing the (tautological) equivalence.

Example

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

- $p_1 \leftrightarrow (s_1 \wedge s_2)$
- $p_2 \leftrightarrow (s_3 \wedge s_4)$
- $p_3 \leftrightarrow p_1 \vee p_2$
- $w \sim_{SAT} (p_1 \leftrightarrow (s_1 \wedge s_2)) \wedge (p_2 \leftrightarrow (s_3 \wedge s_4)) \wedge (p_3 \leftrightarrow p_1 \vee p_2) \wedge p_3$

Trick: Converting an expression by **adding** new propositional variables and **substituting** for nested operations. We avoid the exponential cost at the price of losing the (tautological) equivalence.

Example

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

- $p_1 \leftrightarrow (s_1 \wedge s_2)$
- $p_2 \leftrightarrow (s_3 \wedge s_4)$
- $p_3 \leftrightarrow p_1 \vee p_2$
- $w \sim_{SAT} (p_1 \leftrightarrow (s_1 \wedge s_2)) \wedge (p_2 \leftrightarrow (s_3 \wedge s_4)) \wedge (p_3 \leftrightarrow p_1 \vee p_2) \wedge p_3$

- $p \leftrightarrow (\ell_1 \wedge \ell_2) \sim (\neg p \vee \ell_1) \wedge (\neg p \vee \ell_2) \wedge (\neg \ell_1 \vee \neg \ell_2 \vee p)$ (CNF)
- $p \leftrightarrow (\ell_1 \vee \ell_2) \sim \neg p \leftrightarrow (\neg \ell_1 \wedge \neg \ell_2)$
- $p \leftrightarrow \ell \sim p \leftrightarrow \ell \wedge 1$

- Each operator (gate) adds at most 3 clauses.
- An expression with m operators becomes a CNF
    - with at most $3m + 1$, $O(m)$, clauses, and
    - an **additional** m propositional variables
- **Linear increase in size**

- $p \leftrightarrow (\ell_1 \wedge \ell_2) \sim (\neg p \vee \ell_1) \wedge (\neg p \vee \ell_2) \wedge (\neg \ell_1 \vee \neg \ell_2 \vee p)$ (CNF)
- $p \leftrightarrow (\ell_1 \vee \ell_2) \sim \neg p \leftrightarrow (\neg \ell_1 \wedge \neg \ell_2)$
- $p \leftrightarrow \ell \sim p \leftrightarrow \ell \wedge 1$

- Each operator (gate) adds at most 3 clauses.
- An expression with m operators becomes a CNF
  - with at most $3m + 1$, $O(m)$, clauses, and
  - an **additional** m propositional variables
- **Linear increase in size**

Converting to CNF

Tseytin Transformation [Tseytin 1970]

- $p \leftrightarrow (\ell_1 \wedge \ell_2) \sim (\neg p \vee \ell_1) \wedge (\neg p \vee \ell_2) \wedge (\neg \ell_1 \vee \neg \ell_2 \vee p) \quad$ (CNF)
- $p \leftrightarrow (\ell_1 \vee \ell_2) \sim \neg p \leftrightarrow (\neg \ell_1 \wedge \neg \ell_2)$
- $p \leftrightarrow \ell \sim p \leftrightarrow \ell \wedge 1$

- Each operator (gate) adds at most 3 clauses.
- An expression with m operators becomes a CNF
  - with at most $3m + 1$, $O(m)$, clauses, and
  - an **additional** m propositional variables
- **Linear increase in size**

- $p \leftrightarrow (\ell_1 \wedge \ell_2) \sim (\neg p \vee \ell_1) \wedge (\neg p \vee \ell_2) \wedge (\neg \ell_1 \vee \neg \ell_2 \vee p)$ (CNF)
- $p \leftrightarrow (\ell_1 \vee \ell_2) \sim \neg p \leftrightarrow (\neg \ell_1 \wedge \neg \ell_2)$
- $p \leftrightarrow \ell \sim p \leftrightarrow \ell \wedge 1$

- Each operator (gate) adds at most 3 clauses.
- An expression with m operators becomes a CNF
  - with at most $3m + 1$, $O(m)$, clauses, and
  - an **additional** m propositional variables
- **Linear increase in size**

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

Tseytin Transformation

Let w be a wff expression (a.k.a. Boolean function) of size n. Then, using Tseytin Transformation, w can be converted, in **polynomial time**, into an **equisatisfiable** expression w' in **CNF**.

So, one may assume that we already have a CNF to begin with.

Given a well-formed formula w as an input, if there exists a σ that **satisfies** w return **SAT** (with σ), otherwise return **UNSAT**.

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

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
(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 cases 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 (that is why we call it "complete")
- **No known Polynomial time** algorithm for solving SAT (otherwise P=NP)
- Yet, modern SAT Solvers are arguably **efficient**, why?

SAT Facts

- Brute force algorithm: **exponential complexity**:
- 2^n cases 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 (that is why we call it "complete")
- **No known Polynomial time** algorithm for solving SAT (otherwise P=NP)
- Yet, modern SAT Solvers are arguably **efficient**, why?

SAT Facts

- Brute force algorithm: **exponential complexity**:
- 2^n cases 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 (that is why we call it "complete")
- **No known Polynomial time** algorithm for solving SAT (otherwise P=NP)
- Yet, modern SAT Solvers are arguably **efficient**, why?

- Brute force algorithm: **exponential complexity**:
- 2^n cases 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 (that is why we call it "complete")
- **No known Polynomial time** algorithm for solving SAT (otherwise P=NP)
- Yet, modern SAT Solvers are arguably **efficient**, why?

SAT Facts

- Brute force algorithm: **exponential complexity**:
- 2^n cases 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 (that is why we call it "complete")
- **No known Polynomial time** algorithm for solving SAT (otherwise P=NP)
- Yet, modern SAT Solvers are arguably **efficient**, why?

Satisfiability-Preserving Transformations

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

DP Algorithm

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

- if the clause has the form $s \land \neg s$ 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 expressions)

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 $\ell \vee w$
- $(\ell \vee w_1) \wedge \cdots (\ell \vee w_m) \wedge w' \quad \sim_{SAT} \quad w' \qquad (w'$ has no ℓ in it$)$

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

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! $\sigma = \{1, 7, \bar{2}, \bar{4}, \bar{5}\}$ (with anything for $\{6, 8\}$)

Unit clause is a clause with only **one literal**, ℓ say
A CNF containing a unit clause ℓ has the form

$$\ell \wedge (\ell \vee w_1) \wedge (\neg\ell \vee w_2) \wedge w_3$$

Remove all the clauses containing ℓ

- $\ell \wedge (\ell \vee w_1) \wedge \cdots \wedge (\ell \vee w_m) \wedge w' \quad \sim_{SAT} \quad w'$

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

- $\ell \wedge (\neg\ell \vee w_1) \wedge \cdots \wedge (\neg\ell \vee w_m) \wedge w' \quad \sim_{SAT} \quad w_1 \wedge \cdots \wedge w_m \wedge w'$

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

If ℓ or its negation do not appear in the wff w, then

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

Generalizing to several clauses:

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

Converting back to a CNF

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

If ℓ or its negation do not appear in the wff w, then

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

Generalizing to several clauses:

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

Converting back to a CNF

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

If ℓ or its negation do not appear in the wff w, then

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

Generalizing to several clauses:

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

Converting back to a CNF

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

To summarize, if ℓ or its negation do not appear in the wff w, then

$$\bigwedge_{i=1}^{r}(\ell \vee a_i) \wedge \bigwedge_{j=1}^{s}(\neg\ell \vee b_j) \wedge w \quad \sim_{SAT} \quad \left(\bigwedge_{i=1}^{r}\bigwedge_{j=1}^{s}(a_i \vee b_j)\right) \wedge w$$

- **Before** applying the resolution rule the CNF had $r+s$ clauses containing ℓ or its negation
- **After** applying the rule, the so obtained CNF has rs clauses ...
- and ℓ is **resolved** (eliminated, simplified).

- Thus, **no explicit assignment** is required for ℓ.

Satisfiability-Preserving Transformations

- **Pure literal rule** or affirmative-negative rule
- **Unit propagation** or 1-literal rule
- **Resolution rule** or rule for eliminating literals

In practice

- Pure literal rule is expensive to detect dynamically
- Unit propagation consumes the most significant runtime
- Resolution rule can exhaust rapidly the available memory

Satisfiability-Preserving Transformations

- **Pure literal rule** or affirmative-negative rule
- **Unit propagation** or 1-literal rule
- **Resolution rule** or rule for eliminating literals

In practice

- Pure literal rule is expensive to detect dynamically
- Unit propagation consumes the most significant runtime
- Resolution rule can exhaust rapidly the available memory

$$C_1 := x \vee y, \quad C_2 := \neg x \vee y \vee \neg z, \quad C_3 := x \vee z, \quad C_4 := x \vee \neg z$$

Suppose $\sigma = \{z\}$, that is z is assigned 1.
Let $\#C := (C(\ell = 0), C(\ell = 1))$, then

$$\#C_1 = (0, 0), \quad \#C_2 = (1, 0), \quad \#C_3 = (0, 1), \quad \#C_4 = (1, 0),$$

The pair of lists associated with the variable x is

$$P_x := \{C_1, C_3, C_4\}, \quad N_x := \{C_2\}$$

Observe that $C_3(\ell = 1) = 1$, so C_3 is already satisfied by σ.

If x is assigned to 0, σ becomes $\{z, \bar{x}\}$, and the counters $\#C_i$ become

$$\#C_1 = (1,0), \quad \#C_2 = (1,1), \quad \#C_3 = (1,1), \quad \#C_4 = (2,0)$$

- $C_2(\ell = 1) = 1$: C_2 becomes satisfied.
- $C_4(\ell = 0) = 2 = |C_4|$: C_4 becomes **conflicting**.

If x is assigned to 1, σ becomes $\{z, x\}$, and the counters $\#C_i$ become

$$\#C_1 = (0,1), \quad \#C_2 = (2,0), \quad \#C_3 = (0,2), \quad \#C_4 = (1,1)$$

- $C_1(\ell = 1) = C_4(\ell = 1) = 1$: C_1 and C_4 become satisfied.
- $C_2(\ell = 0) = 2 = -1 + 3 = -1 + |C_2|$: C_2 becomes a **unit clause**.

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

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

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

- 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**

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

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

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

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

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]

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 variable s (which one?)
3. Test if $(w \wedge s)$ is SAT
4. Otherwise return the result for $(w \wedge \neg s)$

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 variable s (which one?)
3. Test if $(w \wedge s)$ is SAT
4. Otherwise return the result for $(w \wedge \neg s)$

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 variable s (which one?)
3. Test if $(w \wedge s)$ is SAT
4. Otherwise return the result for $(w \wedge \neg s)$

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 variable s (which one?)
3. Test if $(w \wedge s)$ is SAT
4. Otherwise return the result for $(w \wedge \neg s)$

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 variable s (which one?)
3. Test if $(w \wedge s)$ is SAT
4. Otherwise return the result for $(w \wedge \neg s)$

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 variable s (which one?)
3. Test if $(w \wedge s)$ is SAT
4. Otherwise return the result for $(w \wedge \neg s)$

$$w = c_1 \wedge c_2 \wedge c_3 \wedge c_4 \wedge c_5 \wedge c_6$$

$$\begin{aligned}
c_1 &= (x_5 \vee x_6) \\
c_2 &= (x_1 \vee x_7 \vee \neg x_2) \\
c_3 &= (x_1 \vee \neg x_3) \\
c_4 &= (x_2 \vee x_3 \vee x_4) \\
c_5 &= (\neg x_4 \vee \neg x_5) \\
c_6 &= (x_8 \vee \neg x_4 \vee \neg x_6)
\end{aligned}$$

Count the number of unresolved clause for each variable

$$x_1 : (2), x_2 : (2), x_3 : (2), \mathbf{x_4} : (\mathbf{3}), x_5 : (2), x_6 : (2), x_7 : (1), x_8 : (1)$$

Branch with the variable having the largest number (here x_4)

$x_4 = 1@1$ (i.e. x_4 set to 1 at decision level 1)

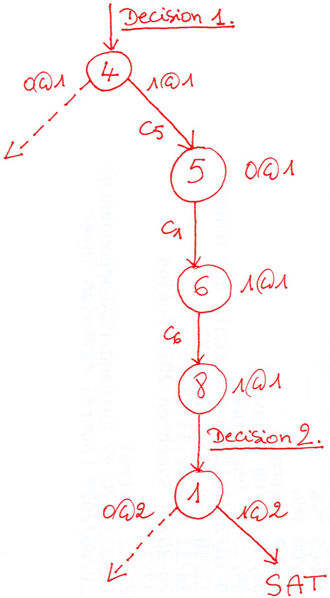
- c_4 becomes resolved
- by UP (c_5), $x_5 = 0@1$,
- by UP (c_1), $x_6 = 1@1$,
- by UP (c_6), $x_8 = 1@1$

Count the number of unresolved clause for each remaining variable (c_2 and c_3)

$$\mathbf{x_1} : (\mathbf{2}), x_2 : (1), x_3 : (1), x_7 : (1)$$

$x_1 = 1@2$

- c_2 and c_3 become resolved
- The algorithm halts with SAT, $\sigma = \{4, \bar{5}, 6, 8, 1\}$


Decision 1.
4
0@1
1@1
C5
5
0@1
C1
6
1@1
C6
8
1@1
Decision 2.
1
0@2
1@2
SAT

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)

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

Solving conflicts:

- If a conflict is detected at decision level $@d$, the decision variable of that level is flipped before starting the UP again.
- If a conflict is again detected, the algorithm goes to decision level $@(d-1)$ and so on.
- If decision level 0 reached, return UNSAT
- Essentially a **Depth First Search** technique.

Problem: several conflicts could be caused by the same assignement made at an early decision level.

The algorithm gets **stuck** in some sort of "local minimum" with an important number of conflicts.

Modern SAT solvers **learns** the conflicting clauses and attempt to **jumpback** to an early root of the conflict.

Two graphs are built iteratively

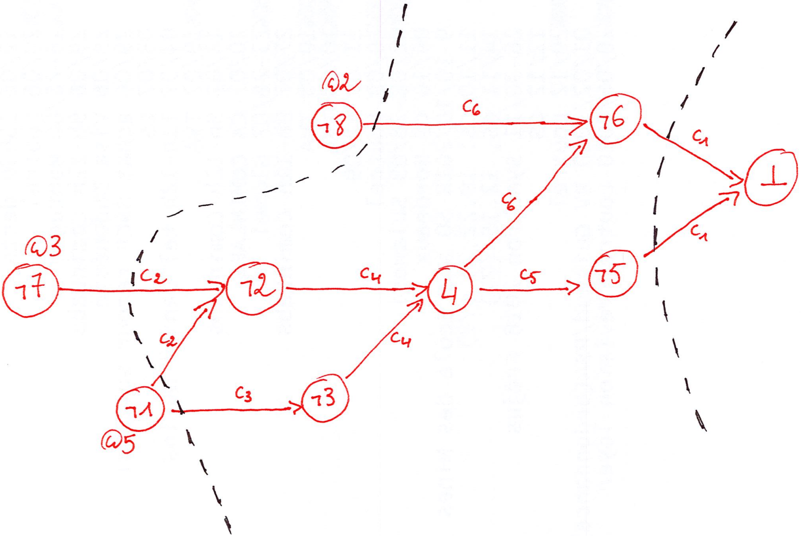
- Search graph (as the one we have already seen)
- **Implication graph**

$$w = c_1 \wedge c_2 \wedge c_3 \wedge c_4 \wedge c_5 \wedge c_6$$

$$\begin{aligned}
c_1 &= (x_5 \vee x_6) \\
c_2 &= (x_1 \vee x_7 \vee \neg x_2) \\
c_3 &= (x_1 \vee \neg x_3) \\
c_4 &= (x_2 \vee x_3 \vee x_4) \\
c_5 &= (\neg x_4 \vee \neg x_5) \\
c_6 &= (x_8 \vee \neg x_4 \vee \neg x_6)
\end{aligned}$$

Assume the following decisions have been made:

$$x_8 = 0@2, \quad x_7 = 0@3, \quad x_1 = 0@5.$$


@2
¬8
c6
¬6
c1
⊥
@3
¬7
c2
¬2
c4
4
c5
¬5
c1
c6
c2
c4
c3
¬1
¬3
@5

- Let $\phi := \neg x_1 \wedge \neg x_7 \wedge \neg x_8$
- $w \wedge \phi$ is UNSAT
- Thus, $w \models \neg\phi$ (Tautological implication)
- Therefore, $w \sim w \wedge \neg\phi$
- $\neg\phi$ is a **learned clause**

Several clauses could be learned by **seperating** the sources from the conflict in the implication graph

$$\phi_1 := \neg x_4 \vee x_8 \qquad \phi_2 := \neg x_4 \vee x_6$$

For instance, by adding ϕ_1 as a new clause to w, with respect to the decision $x_8 = 0@2$, x_4 will be forced to 0 (instead of 1 which would lead inevitably to a conflict according to the implication graph).

In our example, one can jump back to three decision levels: 2, 3 and 5 (the current one).

Unit Implication Point strategy (used in in Chaff)

- One would want to backtrack to a decision that immediately exploits the learned clause to fix an additional variable **without necessarily changing that decision**.
- For instance, by learning ϕ_1 and backtracking to depth 2 (as the earliest decision involved in ϕ), x_4 will be set to 0 by UP.

Learn

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

Backjump

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

Forget

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

Restart

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

Variable State Independent Decaying Sum

VSIDS. [Moskewicz et al., 2001]

In modern solvers, **branching heuristics exploit the learned clauses**:

- 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

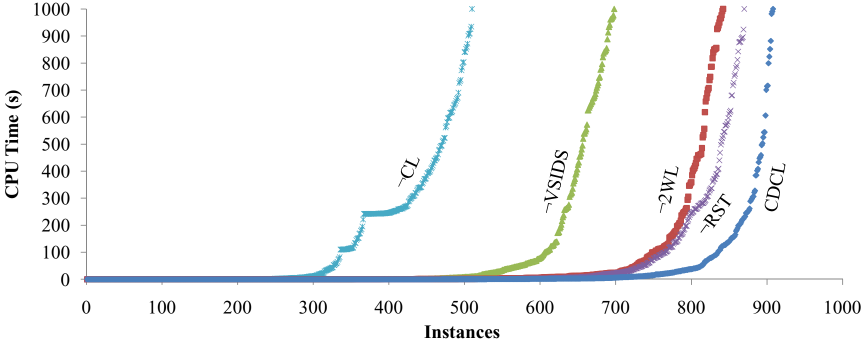
Outline

DPLL-CDCL Modern Decision Procedures

Zhang, Malik, 2002

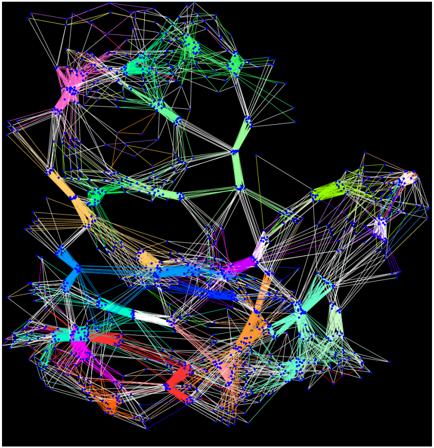
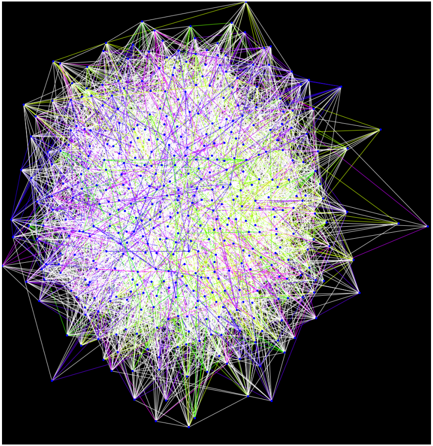
```
status = preprocess();
if (status!=UNKNOWN) return status;
while(true) {
    decide_next_branch();
    while (true) {
        status = deduce();
        if (status == CONFLICT) {
            blevel = analyze_conflict();
            if (blevel == 0)
                return UNSATISFIABLE;
         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

[Katebi et al., 2011]



Visualizing Boolean Functions

[SATGraf, Newsham et al., 2015]

SAT Competition

- Visit satlive.org

Applications

- Automated Theorem Proving (more later)
- Current industrial applications (hardware verification):
- hundreds of millions of variables and clauses for a couple of hours of computations

Alternative Approaches

- Stalmarck's method (generate UNSAT certificates)
- Greedy local search (more adapted for random expressions)

Data:

- A: a finite set of tasks
- ℓ: a measure (or time length) $\ell : A \to \mathbb{N}$
- m processors
- D: a deadline in $\mathbb{N}$

Multiprocessing Scheduling Problem (MSP):
Find a partition $A = A_1 \cup A_2 \cup \cdots \cup A_m$ of A into m disjoint sets such that

$$\max_{1 \leq i \leq m} \left\{ \sum_{a \in A_i} \ell(a) \right\} \leq D \ .$$

Question: Prove that MSP is **NP-complete**.

The problem is in **NP**: Given a partition, one can check the inequality by computing the max over i.

NP-completeness

- Reduce a known NP-complete problem (e.g. SAT) to the multiprocessing scheduling problem.
- Essentially, solve SAT by solving the given problem.

Hamiltonian Circuit Problem

Given a graph $G = (V, E)$, is there a vertex permutation $\pi : V \to V$ such that $\{v_{\pi(n)}, v_{\pi(1)}\} \in E$ and $\{v_{\pi(i)}, v_{\pi(i+1)}\} \in E$, $i = 1, \ldots, n-1$?

Partition Problem

Given a finite set A and a positive measure s on A, is there a subset A' of A, such that

$$\sum_{a \in A'} s(a) = \sum_{a \in A \setminus A'} s(a) \ ?$$

The partition problem is a particular instance of MSP with:

$$D = \frac{1}{2} \sum_{a \in A} s(a), \qquad m = 2, \qquad s = \ell.$$

Suppose we found a partition of A in two subsets $A_1 \cup A_2$ that solves this instance of MSP, we prove that it solves the partition problem.

Suppose that (without loss of generality):

$$\sum_{a\in A_1} s(a) \le \sum_{a\in A_2} s(a),$$

then, A_1, A_2, D solve MSP:

$$\max_{1\le i\le 2}\left\{\sum_{a\in A_i} s(a)\right\} = \sum_{a\in A_2} s(a) \le D = \frac{1}{2}\sum_{a\in A} s(a) = \frac{1}{2}\sum_{a\in A_1} s(a) + \frac{1}{2}\sum_{a\in A_2} s(a)$$

which implies

$$\frac{1}{2}\sum_{a\in A_2} s(a) \le \frac{1}{2}\sum_{a\in A_1} s(a)$$

Therefore

$$\sum_{a\in A_1} s(a) = \sum_{a\in A_2} s(a) = D.$$

NP-Complete Problems are Ubiquitous

- Graph Theory
- Network Design
- Sets and Partition
- Sequencing and Scheduling
- Algebra and Number Theory
- Games and Puzzles
- Automata and Languages
- Optimization
- Logic

Hence the importance of SAT Solvers ...

SAT Problems

- Equisatisfiability (CNF transformation)
- Proving tautological implications/equivalences

CDCL-DPLL Algorithm

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