Automated Reasoning and Formal Verification
Module I: Automated Reasoning
Ch. 01: **Propositional Satisfiability (SAT)**

Roberto Sebastiani

DISI, Università di Trento, Italy – roberto.sebastiani@unitn.it
URL: https://disi.unitn.it/rseba/DIDATTICA/arfv2026/
Teaching assistant: Gabriele Masina – gabriele.masina@unitn.it

M.S. in Computer Science, Mathematics, & Artificial Intelligence Systems
Academic year 2025-2026

last update: Tuesday 3rd March, 2026, 12:07

Outline

Outline

Propositional Logic (aka Boolean Logic)

Basic Definitions

- Propositional formula (aka Boolean formula)
  - $\top, \bot$ are formulas
  - a propositional atom $A_1, A_2, A_3, ...$ is a formula;
  - if φ_1 and φ_2 are formulas, then
    $\neg\varphi_1, \varphi_1 \wedge \varphi_2, \varphi_1 \vee \varphi_2, \varphi_1 \rightarrow \varphi_2, \varphi_1 \leftarrow \varphi_2, \varphi_1 \leftrightarrow \varphi_2, \varphi_1 \oplus \varphi_2$
    are formulas.
- Ex: $\varphi \stackrel{\text{def}}{=} (\neg(A_1 \rightarrow A_2)) \wedge (A_3 \leftrightarrow (\neg A_1 \oplus (A_2 \vee \neg A_4)))$
- *Atoms*(φ): the set $\{A_1, ..., A_N\}$ of atoms occurring in φ.
  - Ex: $Atoms(\varphi) = \{A_1, A_2, A_3, A_4\}$
- Literal: a propositional atom A_i (positive literal) or its negation $\neg A_i$ (negative literal)
  - Notation: if $l := \neg A_i$, then $\neg l := A_i$
- Clause: a disjunction of literals $\bigvee_j l_j$ (e.g., $(A_1 \vee \neg A_2 \vee A_3 \vee ...)$)
- Cube: a conjunction of literals $\bigwedge_j l_j$ (e.g., $(A_1 \wedge \neg A_2 \wedge A_3 \wedge ...)$)

Semantics of Boolean operators

Truth Table

α	β	$\neg\alpha$	$\alpha\wedge\beta$	$\alpha\vee\beta$	$\alpha\rightarrow\beta$	$\alpha\leftarrow\beta$	$\alpha\leftrightarrow\beta$	$\alpha\oplus\beta$
$\bot$	$\bot$	$\top$	$\bot$	$\bot$	$\top$	$\top$	$\top$	$\bot$
$\bot$	$\top$	$\top$	$\bot$	$\top$	$\top$	$\bot$	$\bot$	$\top$
$\top$	$\bot$	$\bot$	$\bot$	$\top$	$\bot$	$\top$	$\bot$	$\top$
$\top$	$\top$	$\bot$	$\top$	$\top$	$\top$	$\top$	$\top$	$\bot$

English meaning of connectives

English	Logic
A and B	$A \wedge B$
A if B | A when B | A whenever B	$A \leftarrow B$
if A, then B | A implies B | A forces B | A requires B	$A \rightarrow B$
A precisely when B | A if and only if B	$A \leftrightarrow B$
A or B (or both) | A unless B	$A \vee B$ (logical or)
either A or B (but not both)	$A \oplus B$ (exclusive or)

Semantics of Boolean operators (cont.)

Note

- $\land$, $\lor$, $\leftrightarrow$ and $\oplus$ are commutative:
$$\begin{array}{lcl} (\alpha \land \beta) & \Longleftrightarrow & (\beta \land \alpha) \\ (\alpha \lor \beta) & \Longleftrightarrow & (\beta \lor \alpha) \\ (\alpha \leftrightarrow \beta) & \Longleftrightarrow & (\beta \leftrightarrow \alpha) \\ (\alpha \oplus \beta) & \Longleftrightarrow & (\beta \oplus \alpha) \end{array}$$
- $\land$, $\lor$, $\leftrightarrow$ and $\oplus$ are associative:
$$\begin{array}{lclcl} ((\alpha \land \beta) \land \gamma) & \Longleftrightarrow & (\alpha \land (\beta \land \gamma)) & \Longleftrightarrow & (\alpha \land \beta \land \gamma) \\ ((\alpha \lor \beta) \lor \gamma) & \Longleftrightarrow & (\alpha \lor (\beta \lor \gamma)) & \Longleftrightarrow & (\alpha \lor \beta \lor \gamma) \\ ((\alpha \leftrightarrow \beta) \leftrightarrow \gamma) & \Longleftrightarrow & (\alpha \leftrightarrow (\beta \leftrightarrow \gamma)) & \Longleftrightarrow & (\alpha \leftrightarrow \beta \leftrightarrow \gamma) \\ ((\alpha \oplus \beta) \oplus \gamma) & \Longleftrightarrow & (\alpha \oplus (\beta \oplus \gamma)) & \Longleftrightarrow & (\alpha \oplus \beta \oplus \gamma) \end{array}$$
- $\rightarrow$, $\leftarrow$ are neither commutative nor associative:
$$\begin{array}{lcl} (\alpha \rightarrow \beta) & \not\Longleftrightarrow & (\beta \rightarrow \alpha) \\ ((\alpha \rightarrow \beta) \rightarrow \gamma) & \not\Longleftrightarrow & (\alpha \rightarrow (\beta \rightarrow \gamma)) \end{array}$$

Equivalences with Boolean Operators

$$
\begin{array}{rcl}
\neg\neg\alpha & \Longleftrightarrow & \alpha \\
(\alpha \vee \beta) & \Longleftrightarrow & \neg(\neg\alpha \wedge \neg\beta) \\
\neg(\alpha \vee \beta) & \Longleftrightarrow & (\neg\alpha \wedge \neg\beta) \\
(\alpha \wedge \beta) & \Longleftrightarrow & \neg(\neg\alpha \vee \neg\beta) \\
\neg(\alpha \wedge \beta) & \Longleftrightarrow & (\neg\alpha \vee \neg\beta) \\
(\alpha \rightarrow \beta) & \Longleftrightarrow & (\neg\alpha \vee \beta) \\
\neg(\alpha \rightarrow \beta) & \Longleftrightarrow & (\alpha \wedge \neg\beta) \\
(\alpha \leftarrow \beta) & \Longleftrightarrow & (\alpha \vee \neg\beta) \\
\neg(\alpha \leftarrow \beta) & \Longleftrightarrow & (\neg\alpha \wedge \beta) \\
(\alpha \leftrightarrow \beta) & \Longleftrightarrow & ((\alpha \rightarrow \beta) \wedge (\alpha \leftarrow \beta)) \\
 & \Longleftrightarrow & ((\neg\alpha \vee \beta) \wedge (\alpha \vee \neg\beta)) \\
\neg(\alpha \leftrightarrow \beta) & \Longleftrightarrow & (\neg\alpha \leftrightarrow \beta) \\
 & \Longleftrightarrow & (\alpha \leftrightarrow \neg\beta) \\
 & \Longleftrightarrow & ((\alpha \vee \beta) \wedge (\neg\alpha \vee \neg\beta)) \\
(\alpha \oplus \beta) & \Longleftrightarrow & \neg(\alpha \leftrightarrow \beta)
\end{array}
$$

Boolean logic can be expressed in terms of $\{\neg, \wedge\}$ (or $\{\neg, \vee\}$) only!

Equivalences with Boolean Operators

$$
\begin{array}{rcl}
\neg\neg\alpha & \Longleftrightarrow & \alpha \\
(\alpha \vee \beta) & \Longleftrightarrow & \neg(\neg\alpha \wedge \neg\beta) \\
\neg(\alpha \vee \beta) & \Longleftrightarrow & (\neg\alpha \wedge \neg\beta) \\
(\alpha \wedge \beta) & \Longleftrightarrow & \neg(\neg\alpha \vee \neg\beta) \\
\neg(\alpha \wedge \beta) & \Longleftrightarrow & (\neg\alpha \vee \neg\beta) \\
(\alpha \rightarrow \beta) & \Longleftrightarrow & (\neg\alpha \vee \beta) \\
\neg(\alpha \rightarrow \beta) & \Longleftrightarrow & (\alpha \wedge \neg\beta) \\
(\alpha \leftarrow \beta) & \Longleftrightarrow & (\alpha \vee \neg\beta) \\
\neg(\alpha \leftarrow \beta) & \Longleftrightarrow & (\neg\alpha \wedge \beta) \\
(\alpha \leftrightarrow \beta) & \Longleftrightarrow & ((\alpha \rightarrow \beta) \wedge (\alpha \leftarrow \beta)) \\
 & \Longleftrightarrow & ((\neg\alpha \vee \beta) \wedge (\alpha \vee \neg\beta)) \\
\neg(\alpha \leftrightarrow \beta) & \Longleftrightarrow & (\neg\alpha \leftrightarrow \beta) \\
 & \Longleftrightarrow & (\alpha \leftrightarrow \neg\beta) \\
 & \Longleftrightarrow & ((\alpha \vee \beta) \wedge (\neg\alpha \vee \neg\beta)) \\
(\alpha \oplus \beta) & \Longleftrightarrow & \neg(\alpha \leftrightarrow \beta)
\end{array}
$$

Boolean logic can be expressed in terms of $\{\neg, \wedge\}$ (or $\{\neg, \vee\}$) only!

Tree & DAG Representations of Formulas

- Formulas can be represented either as trees or as DAGS (Directed Acyclic Graphs)
- DAG representation can be up to exponentially smaller
  - in particular, when $\leftrightarrow$'s are involved

$$(A_1 \leftrightarrow A_2) \leftrightarrow (A_3 \leftrightarrow A_4)$$

Tree & DAG Representations of Formulas

- Formulas can be represented either as trees or as DAGS (Directed Acyclic Graphs)
- DAG representation can be up to exponentially smaller
  - in particular, when $\leftrightarrow$'s are involved

$$(A_1 \leftrightarrow A_2) \leftrightarrow (A_3 \leftrightarrow A_4)$$
$$\Downarrow$$
$$(((A_1 \leftrightarrow A_2) \rightarrow (A_3 \leftrightarrow A_4)) \wedge$$
$$((A_3 \leftrightarrow A_4) \rightarrow (A_1 \leftrightarrow A_2)))$$

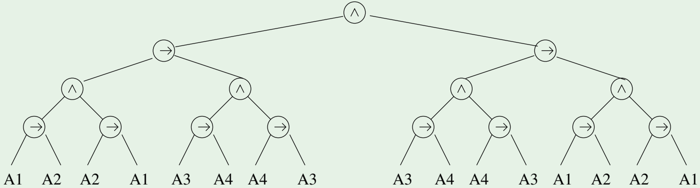
Tree & DAG Representations of Formulas

- Formulas can be represented either as trees or as DAGS (Directed Acyclic Graphs)
- DAG representation can be up to exponentially smaller
  - in particular, when $\leftrightarrow$'s are involved

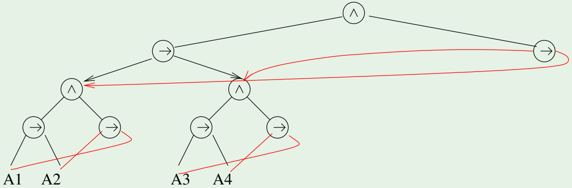
$$(A_1 \leftrightarrow A_2) \leftrightarrow (A_3 \leftrightarrow A_4)$$
$$\Downarrow$$
$$(((A_1 \leftrightarrow A_2) \rightarrow (A_3 \leftrightarrow A_4)) \wedge$$
$$((A_3 \leftrightarrow A_4) \rightarrow (A_1 \leftrightarrow A_2)))$$
$$\Downarrow$$
$$(((A_1 \rightarrow A_2) \wedge (A_2 \rightarrow A_1)) \rightarrow ((A_3 \rightarrow A_4) \wedge (A_4 \rightarrow A_3))) \wedge$$
$$(((A_3 \rightarrow A_4) \wedge (A_4 \rightarrow A_3)) \rightarrow (((A_1 \rightarrow A_2) \wedge (A_2 \rightarrow A_1))))$$

Tree & DAG Representations of Formulas: Example

$$(((A_1 \to A_2) \land (A_2 \to A_1)) \to ((A_3 \to A_4) \land (A_4 \to A_3))) \land$$
$$(((A_3 \to A_4) \land (A_4 \to A_3)) \to (((A_1 \to A_2) \land (A_2 \to A_1))))$$



Tree Representation



DAG Representation

Semantics: Basic Definitions

- Total truth assignment μ for φ:
  $\mu : Atoms(\varphi) \longmapsto \{\top, \bot\}$.
  - represents a possible world or a possible state of the world
- Partial Truth assignment μ for φ:
  $\mu : \mathcal{A} \longmapsto \{\top, \bot\}$, $\mathcal{A} \subset Atoms(\varphi)$.
  - represents 2^k total assignments, k is $\#$ unassigned variables
- Notation: set and formula representations of an assignment
  - μ can be represented as a set of literals:
    EX: $\{\mu(A_1) := \top, \mu(A_2) := \bot\} \implies \{A_1, \neg A_2\}$
  - μ can be represented as a formula (cube):
    EX: $\{\mu(A_1) := \top, \mu(A_2) := \bot\} \implies (A_1 \wedge \neg A_2)$

Semantics: Basic Definitions [cont.]

- A total truth assignment μ satisfies φ (μ is a model of φ, $\mu \models \varphi$):

$$
\begin{array}{l}
\mu \models A_i \iff \mu(A_i) = \top \\
\mu \models \neg\varphi \iff \textit{not } \mu \models \varphi \\
\mu \models \alpha \wedge \beta \iff \mu \models \alpha \textit{ and } \mu \models \beta \\
\mu \models \alpha \vee \beta \iff \mu \models \alpha \textit{ or } \mu \models \beta \\
\mu \models \alpha \to \beta \iff \textit{ if } \mu \models \alpha, \textit{ then } \mu \models \beta \\
\mu \models \alpha \leftrightarrow \beta \iff \mu \models \alpha \textit{ iff } \mu \models \beta \\
\mu \models \alpha \oplus \beta \iff \mu \models \alpha \textit{ iff not } \mu \models \beta
\end{array}
$$

- $M(\varphi) \stackrel{\text{def}}{=} \{\mu \mid \mu \models \varphi\}$ (the set of models of φ)
- A partial truth assignment μ satisfies φ iff all total assignments extending μ satisfy φ
  - Ex: $\{A_1\} \models (A_1 \vee A_2)$ because both $\{A_1, A_2\} \models (A_1 \vee A_2)$ and $\{A_1, \neg A_2\} \models (A_1 \vee A_2)$
- φ is satisfiable iff $\mu \models \varphi$ for some μ (i.e. $M(\varphi) \neq \emptyset$)
- α entails β ($\alpha \models \beta$): $\alpha \models \beta$ iff $\mu \models \alpha \Longrightarrow \mu \models \beta$ for all μs (i.e., $M(\alpha) \subseteq M(\beta)$)
- φ is valid ($\models \varphi$): $\models \varphi$ iff $\mu \models \varphi$ for all μs (i.e., $\mu \in M(\varphi)$ for all μs)

Semantics: Basic Definitions [cont.]

- A total truth assignment μ satisfies φ (μ is a model of φ, $\mu \models \varphi$):

$$
\begin{array}{l}
\mu \models A_i \iff \mu(A_i) = \top \\
\mu \models \neg\varphi \iff \textit{not } \mu \models \varphi \\
\mu \models \alpha \wedge \beta \iff \mu \models \alpha \textit{ and } \mu \models \beta \\
\mu \models \alpha \vee \beta \iff \mu \models \alpha \textit{ or } \mu \models \beta \\
\mu \models \alpha \rightarrow \beta \iff \textit{ if } \mu \models \alpha, \textit{ then } \mu \models \beta \\
\mu \models \alpha \leftrightarrow \beta \iff \mu \models \alpha \textit{ iff } \mu \models \beta \\
\mu \models \alpha \oplus \beta \iff \mu \models \alpha \textit{ iff not } \mu \models \beta
\end{array}
$$

- $M(\varphi) \stackrel{\text{def}}{=} \{\mu \mid \mu \models \varphi\}$ (the set of models of φ)
- A partial truth assignment μ satisfies φ iff all total assignments extending μ satisfy φ
  - Ex: $\{A_1\} \models (A_1 \vee A_2))$ because both $\{A_1, A_2\} \models (A_1 \vee A_2)$ and $\{A_1, \neg A_2\} \models (A_1 \vee A_2)$
- φ is satisfiable iff $\mu \models \varphi$ for some μ (i.e. $M(\varphi) \neq \emptyset$)
- α entails β ($\alpha \models \beta$): $\alpha \models \beta$ iff $\mu \models \alpha \Longrightarrow \mu \models \beta$ for all μs (i.e., $M(\alpha) \subseteq M(\beta)$)
- φ is valid ($\models \varphi$): $\models \varphi$ iff $\mu \models \varphi$ for all μs (i.e., $\mu \in M(\varphi)$ for all μs)

Semantics: Basic Definitions [cont.]

- A total truth assignment μ satisfies φ (μ is a model of φ, $\mu \models \varphi$):

$$\begin{array}{l}
\mu \models A_i \iff \mu(A_i) = \top \\
\mu \models \neg\varphi \iff \textit{not } \mu \models \varphi \\
\mu \models \alpha \wedge \beta \iff \mu \models \alpha \textit{ and } \mu \models \beta \\
\mu \models \alpha \vee \beta \iff \mu \models \alpha \textit{ or } \mu \models \beta \\
\mu \models \alpha \rightarrow \beta \iff \textit{ if } \mu \models \alpha, \textit{ then } \mu \models \beta \\
\mu \models \alpha \leftrightarrow \beta \iff \mu \models \alpha \textit{ iff } \mu \models \beta \\
\mu \models \alpha \oplus \beta \iff \mu \models \alpha \textit{ iff not } \mu \models \beta
\end{array}$$

- $M(\varphi) \stackrel{\text{def}}{=} \{\mu \mid \mu \models \varphi\}$ (the set of models of φ)
- A partial truth assignment μ satisfies φ iff all total assignments extending μ satisfy φ
  - Ex: $\{A_1\} \models (A_1 \vee A_2)$) because both $\{A_1, A_2\} \models (A_1 \vee A_2)$ and $\{A_1, \neg A_2\} \models (A_1 \vee A_2)$
- φ is satisfiable iff $\mu \models \varphi$ for some μ (i.e. $M(\varphi) \neq \emptyset$)
- α entails β ($\alpha \models \beta$): $\alpha \models \beta$ iff $\mu \models \alpha \Longrightarrow \mu \models \beta$ for all μs (i.e., $M(\alpha) \subseteq M(\beta)$)
- φ is valid ($\models \varphi$): $\models \varphi$ iff $\mu \models \varphi$ for all μs (i.e., $\mu \in M(\varphi)$ for all μs)

Semantics: Basic Definitions [cont.]

- A total truth assignment μ satisfies φ (μ is a model of φ, $\mu \models \varphi$):

$$\begin{array}{l}
\mu \models A_i \iff \mu(A_i) = \top \\
\mu \models \neg\varphi \iff \textit{not } \mu \models \varphi \\
\mu \models \alpha \wedge \beta \iff \mu \models \alpha \textit{ and } \mu \models \beta \\
\mu \models \alpha \vee \beta \iff \mu \models \alpha \textit{ or } \mu \models \beta \\
\mu \models \alpha \rightarrow \beta \iff \textit{ if } \mu \models \alpha, \textit{ then } \mu \models \beta \\
\mu \models \alpha \leftrightarrow \beta \iff \mu \models \alpha \textit{ iff } \mu \models \beta \\
\mu \models \alpha \oplus \beta \iff \mu \models \alpha \textit{ iff not } \mu \models \beta
\end{array}$$

- $M(\varphi) \stackrel{\text{def}}{=} \{\mu \mid \mu \models \varphi\}$ (the set of models of φ)
- A partial truth assignment μ satisfies φ iff all total assignments extending μ satisfy φ
  - Ex: $\{A_1\} \models (A_1 \vee A_2)$) because both $\{A_1, A_2\} \models (A_1 \vee A_2)$ and $\{A_1, \neg A_2\} \models (A_1 \vee A_2)$
- φ is satisfiable iff $\mu \models \varphi$ for some μ (i.e. $M(\varphi) \neq \emptyset$)
- α entails β ($\alpha \models \beta$): $\alpha \models \beta$ iff $\mu \models \alpha \Longrightarrow \mu \models \beta$ for all μs
  (i.e., $M(\alpha) \subseteq M(\beta)$)
- φ is valid ($\models \varphi$): $\models \varphi$ iff $\mu \models \varphi$ for all μs
  (i.e., $\mu \in M(\varphi)$ for all μs)

Semantics: Basic Definitions [cont.]

- A total truth assignment μ satisfies φ (μ is a model of φ, $\mu \models \varphi$):

$$
\begin{array}{l}
\mu \models A_i \iff \mu(A_i) = \top \\
\mu \models \neg\varphi \iff \textit{not } \mu \models \varphi \\
\mu \models \alpha \wedge \beta \iff \mu \models \alpha \textit{ and } \mu \models \beta \\
\mu \models \alpha \vee \beta \iff \mu \models \alpha \textit{ or } \mu \models \beta \\
\mu \models \alpha \rightarrow \beta \iff \textit{ if } \mu \models \alpha, \textit{ then } \mu \models \beta \\
\mu \models \alpha \leftrightarrow \beta \iff \mu \models \alpha \textit{ iff } \mu \models \beta \\
\mu \models \alpha \oplus \beta \iff \mu \models \alpha \textit{ iff not } \mu \models \beta
\end{array}
$$

- $M(\varphi) \stackrel{\text{def}}{=} \{\mu \mid \mu \models \varphi\}$ (the set of models of φ)
- A partial truth assignment μ satisfies φ iff all total assignments extending μ satisfy φ
  - Ex: $\{A_1\} \models (A_1 \vee A_2)$) because both $\{A_1, A_2\} \models (A_1 \vee A_2)$ and $\{A_1, \neg A_2\} \models (A_1 \vee A_2)$
- φ is satisfiable iff $\mu \models \varphi$ for some μ (i.e. $M(\varphi) \neq \emptyset$)
- α entails β ($\alpha \models \beta$): $\alpha \models \beta$ iff $\mu \models \alpha \Longrightarrow \mu \models \beta$ for all μs (i.e., $M(\alpha) \subseteq M(\beta)$)
- φ is valid ($\models \varphi$): $\models \varphi$ iff $\mu \models \varphi$ for all μs (i.e., $\mu \in M(\varphi)$ for all μs)

Properties & Results

Property

φ is valid iff $\neg\varphi$ is not satisfiable

Deduction Theorem

$\alpha \models \beta$ iff $\alpha \rightarrow \beta$ is valid ($\models \alpha \rightarrow \beta$)

Corollary

$\alpha \models \beta$ iff $\alpha \wedge \neg\beta$ is not satisfiable

Validity and entailment checking can be straightforwardly reduced to (un)satisfiability checking!

Properties & Results

Property

φ is valid iff $\neg\varphi$ is not satisfiable

Deduction Theorem

$\alpha \models \beta$ iff $\alpha \rightarrow \beta$ is valid ($\models \alpha \rightarrow \beta$)

Corollary

$\alpha \models \beta$ iff $\alpha \wedge \neg\beta$ is not satisfiable

Validity and entailment checking can be straightforwardly reduced to (un)satisfiability checking!

Properties & Results

Property

φ is valid iff $\neg\varphi$ is not satisfiable

Deduction Theorem

$\alpha \models \beta$ iff $\alpha \rightarrow \beta$ is valid ($\models \alpha \rightarrow \beta$)

Corollary

$\alpha \models \beta$ iff $\alpha \wedge \neg\beta$ is not satisfiable

Validity and entailment checking can be straightforwardly reduced to (un)satisfiability checking!

Properties & Results

Property

φ is valid iff $\neg\varphi$ is not satisfiable

Deduction Theorem

$\alpha \models \beta$ iff $\alpha \to \beta$ is valid ($\models \alpha \to \beta$)

Corollary

$\alpha \models \beta$ iff $\alpha \wedge \neg\beta$ is not satisfiable

Validity and entailment checking can be straightforwardly reduced to (un)satisfiability checking!

Equivalence and Equi-Satisfiability

- α and β are equivalent iff, for every μ, $\mu \models \alpha$ iff $\mu \models \beta$
  (i.e., if $M(\alpha) = M(\beta)$)
- α and β are equi-satisfiable iff exists μ_1 s.t. $\mu_1 \models \alpha$ iff exists μ_2 s.t. $\mu_2 \models \beta$
  (i.e., if $M(\alpha) \neq \emptyset$ iff $M(\beta) \neq \emptyset$)
- α, β equivalent
  $\Downarrow \quad \not\Uparrow$
  α, β equi-satisfiable
- EX: $A_1 \vee A_2$ and $(A_1 \vee \neg A_3) \wedge (A_3 \vee A_2)$ are equi-satisfiable, not equivalent.
  $\{\neg A_1, A_2, A_3\} \models (A_1 \vee A_2)$, but $\{\neg A_1, A_2, A_3\} \not\models (A_1 \vee \neg A_3) \wedge (A_3 \vee A_2)$
- Typically used when β is the result of applying some transformation T to α: $\beta \stackrel{\text{def}}{=} T(\alpha)$:
  - T is validity-preserving [resp. satisfiability-preserving] iff
    $T(\alpha)$ and α are equivalent [resp. equi-satisfiable]

Equivalence and Equi-Satisfiability

- α and β are equivalent iff, for every μ, $\mu \models \alpha$ iff $\mu \models \beta$ (i.e., if $M(\alpha) = M(\beta)$)
- α and β are equi-satisfiable iff exists μ_1 s.t. $\mu_1 \models \alpha$ iff exists μ_2 s.t. $\mu_2 \models \beta$ (i.e., if $M(\alpha) \neq \emptyset$ iff $M(\beta) \neq \emptyset$)
- α, β equivalent
  $$\Downarrow \quad \not\Uparrow$$
  α, β equi-satisfiable
- EX: $A_1 \vee A_2$ and $(A_1 \vee \neg A_3) \wedge (A_3 \vee A_2)$ are equi-satisfiable, not equivalent. $\{\neg A_1, A_2, A_3\} \models (A_1 \vee A_2)$, but $\{\neg A_1, A_2, A_3\} \not\models (A_1 \vee \neg A_3) \wedge (A_3 \vee A_2)$
- Typically used when β is the result of applying some transformation T to α: $\beta \stackrel{\text{def}}{=} T(\alpha)$:
  - T is validity-preserving [resp. satisfiability-preserving] iff $T(\alpha)$ and α are equivalent [resp. equi-satisfiable]

Equivalence and Equi-Satisfiability

- α and β are equivalent iff, for every μ, $\mu \models \alpha$ iff $\mu \models \beta$ (i.e., if $M(\alpha) = M(\beta)$)
- α and β are equi-satisfiable iff exists μ_1 s.t. $\mu_1 \models \alpha$ iff exists μ_2 s.t. $\mu_2 \models \beta$ (i.e., if $M(\alpha) \neq \emptyset$ iff $M(\beta) \neq \emptyset$)
- α, β equivalent
  $\Downarrow \quad \not\Uparrow$
  α, β equi-satisfiable
- EX: $A_1 \vee A_2$ and $(A_1 \vee \neg A_3) \wedge (A_3 \vee A_2)$ are equi-satisfiable, not equivalent. $\{\neg A_1, A_2, A_3\} \models (A_1 \vee A_2)$, but $\{\neg A_1, A_2, A_3\} \not\models (A_1 \vee \neg A_3) \wedge (A_3 \vee A_2)$
- Typically used when β is the result of applying some transformation T to α: $\beta \stackrel{\text{def}}{=} T(\alpha)$:
  - T is validity-preserving [resp. satisfiability-preserving] iff $T(\alpha)$ and α are equivalent [resp. equi-satisfiable]

Equivalence and Equi-Satisfiability

- α and β are equivalent iff, for every μ, $\mu \models \alpha$ iff $\mu \models \beta$
  (i.e., if $M(\alpha) = M(\beta)$)
- α and β are equi-satisfiable iff exists μ_1 s.t. $\mu_1 \models \alpha$ iff exists μ_2 s.t. $\mu_2 \models \beta$
  (i.e., if $M(\alpha) \neq \emptyset$ iff $M(\beta) \neq \emptyset$)
- α, β equivalent
  $\Downarrow \quad \not\Uparrow$
  α, β equi-satisfiable
- EX: $A_1 \vee A_2$ and $(A_1 \vee \neg A_3) \wedge (A_3 \vee A_2)$ are equi-satisfiable, not equivalent.
  $\{\neg A_1, A_2, A_3\} \models (A_1 \vee A_2)$, but $\{\neg A_1, A_2, A_3\} \not\models (A_1 \vee \neg A_3) \wedge (A_3 \vee A_2)$
- Typically used when β is the result of applying some transformation T to α: $\beta \stackrel{\text{def}}{=} T(\alpha)$:
  - T is validity-preserving [resp. satisfiability-preserving] iff
    $T(\alpha)$ and α are equivalent [resp. equi-satisfiable]

Equivalence and Equi-Satisfiability

- α and β are equivalent iff, for every μ, $\mu \models \alpha$ iff $\mu \models \beta$
  (i.e., if $M(\alpha) = M(\beta)$)
- α and β are equi-satisfiable iff exists μ_1 s.t. $\mu_1 \models \alpha$ iff exists μ_2 s.t. $\mu_2 \models \beta$
  (i.e., if $M(\alpha) \neq \emptyset$ iff $M(\beta) \neq \emptyset$)
- α, β equivalent

  $\Downarrow \quad \not\Uparrow$

  α, β equi-satisfiable
- EX: $A_1 \vee A_2$ and $(A_1 \vee \neg A_3) \wedge (A_3 \vee A_2)$ are equi-satisfiable, not equivalent.
  $\{\neg A_1, A_2, A_3\} \models (A_1 \vee A_2)$, but $\{\neg A_1, A_2, A_3\} \not\models (A_1 \vee \neg A_3) \wedge (A_3 \vee A_2)$
- Typically used when β is the result of applying some transformation T to α: $\beta \stackrel{\text{def}}{=} T(\alpha)$:
  - T is validity-preserving [resp. satisfiability-preserving] iff
    $T(\alpha)$ and α are equivalent [resp. equi-satisfiable]

Boolean Quantification

Shannon's expansion:

- If v is a Boolean variable and f is a Boolean formula, then

$$\begin{array}{lcl} \exists v.\varphi & := & \varphi|_{v=\bot} \vee \varphi|_{v=\top} \\ \forall v.\varphi & := & \varphi|_{v=\bot} \wedge \varphi|_{v=\top} \end{array}$$

- v does no more occur in $\exists v.\varphi$ and $\forall v.\varphi$!!
- Multi-variable quantification: $\exists (w_1, \ldots, w_n).\varphi \;:=\; \exists w_1 \ldots \exists w_n.\varphi$

- Intuition:
  - $\mu \models \exists v.\varphi$ iff exists $truthvalue \in \{\top, \bot\}$ s.t. $\mu \cup \{v := truthvalue\} \models \varphi$
  - $\mu \models \forall v.\varphi$ iff forall $truthvalue \in \{\top, \bot\}$, $\mu \cup \{v := truthvalue\} \models \varphi$
- Example: $\exists (b, c).((a \wedge b) \vee (c \wedge d)) \;=\; a \vee d$

Note

Naive expansion of quantifiers to propositional logic may cause a blow-up in size of the formulae

Boolean Quantification

Shannon's expansion:

- If v is a Boolean variable and f is a Boolean formula, then

$$\begin{aligned} \exists v.\varphi &:= \varphi|_{v=\bot} \vee \varphi|_{v=\top} \\ \forall v.\varphi &:= \varphi|_{v=\bot} \wedge \varphi|_{v=\top} \end{aligned}$$

- v does no more occur in $\exists v.\varphi$ and $\forall v.\varphi$!!
- Multi-variable quantification: $\exists(w_1,\ldots,w_n).\varphi \;:=\; \exists w_1 \ldots \exists w_n.\varphi$

- Intuition:
  - $\mu \models \exists v.\varphi$ iff exists $truthvalue \in \{\top,\bot\}$ s.t. $\mu \cup \{v = truthvalue\} \models \varphi$
  - $\mu \models \forall v.\varphi$ iff forall $truthvalue \in \{\top,\bot\}$, $\mu \cup \{v = truthvalue\} \models \varphi$
- Example: $\exists(b,c).((a \wedge b) \vee (c \wedge d)) \;=\; a \vee d$

Note

Naive expansion of quantifiers to propositional logic may cause a blow-up in size of the formulae

Boolean Quantification

Shannon's expansion:

- If v is a Boolean variable and f is a Boolean formula, then
  $$\begin{array}{lcl} \exists v.\varphi & := & \varphi|_{v=\bot} \vee \varphi|_{v=\top} \\ \forall v.\varphi & := & \varphi|_{v=\bot} \wedge \varphi|_{v=\top} \end{array}$$
- v does no more occur in $\exists v.\varphi$ and $\forall v.\varphi$!!
- Multi-variable quantification: $\exists (w_1, \ldots, w_n).\varphi \; := \; \exists w_1 \ldots \exists w_n.\varphi$

- Intuition:
  - $\mu \models \exists v.\varphi$ iff exists *truthvalue* $\in \{\top, \bot\}$ s.t. $\mu \cup \{v = \mathit{truthvalue}\} \models \varphi$
  - $\mu \models \forall v.\varphi$ iff forall *truthvalue* $\in \{\top, \bot\}$, $\mu \cup \{v = \mathit{truthvalue}\} \models \varphi$
- Example: $\exists (b, c).((a \wedge b) \vee (c \wedge d)) \; = \; a \vee d$

Note

Naive expansion of quantifiers to propositional logic may cause a blow-up in size of the formulae

Boolean Quantification

Shannon's expansion:

- If v is a Boolean variable and f is a Boolean formula, then
  $$\begin{aligned} \exists v.\varphi &:= \varphi|_{v=\bot} \vee \varphi|_{v=\top} \\ \forall v.\varphi &:= \varphi|_{v=\bot} \wedge \varphi|_{v=\top} \end{aligned}$$
- v does no more occur in $\exists v.\varphi$ and $\forall v.\varphi$!!
- Multi-variable quantification: $\exists (w_1, \ldots, w_n).\varphi \ := \ \exists w_1 \ldots \exists w_n.\varphi$

- Intuition:
  - $\mu \models \exists v.\varphi$ iff exists *truthvalue* $\in \{\top, \bot\}$ s.t. $\mu \cup \{v := \mathit{truthvalue}\} \models \varphi$
  - $\mu \models \forall v.\varphi$ iff forall *truthvalue* $\in \{\top, \bot\}$, $\mu \cup \{v := \mathit{truthvalue}\} \models \varphi$
- Example: $\exists (b, c).((a \wedge b) \vee (c \wedge d)) \ = \ a \vee d$

Note

Naive expansion of quantifiers to propositional logic may cause a blow-up in size of the formulae

Boolean Quantification

Shannon's expansion:

- If v is a Boolean variable and f is a Boolean formula, then

$$\begin{array}{lcl} \exists v.\varphi & := & \varphi|_{v=\perp} \vee \varphi|_{v=\top} \\ \forall v.\varphi & := & \varphi|_{v=\perp} \wedge \varphi|_{v=\top} \end{array}$$

- v does no more occur in $\exists v.\varphi$ and $\forall v.\varphi$!!
- Multi-variable quantification: $\exists(w_1, \ldots, w_n).\varphi \;:=\; \exists w_1 \ldots \exists w_n.\varphi$

- Intuition:
  - $\mu \models \exists v.\varphi$ iff exists *truthvalue* $\in \{\top, \perp\}$ s.t. $\mu \cup \{v := \mathit{truthvalue}\} \models \varphi$
  - $\mu \models \forall v.\varphi$ iff forall *truthvalue* $\in \{\top, \perp\}$, $\mu \cup \{v := \mathit{truthvalue}\} \models \varphi$
- Example: $\exists(b, c).((a \wedge b) \vee (c \wedge d)) \;=\; a \vee d$

Note

Naive expansion of quantifiers to propositional logic may cause a blow-up in size of the formulae

Boolean Quantification

Shannon's expansion:

- If v is a Boolean variable and f is a Boolean formula, then
$$\exists v.\varphi \quad := \quad \varphi|_{v=\bot} \vee \varphi|_{v=\top}$$
$$\forall v.\varphi \quad := \quad \varphi|_{v=\bot} \wedge \varphi|_{v=\top}$$
- v does no more occur in $\exists v.\varphi$ and $\forall v.\varphi$!!
- Multi-variable quantification: $\exists(w_1, \ldots, w_n).\varphi \ := \ \exists w_1 \ldots \exists w_n.\varphi$

- Intuition:
  - $\mu \models \exists v.\varphi$ iff exists *truthvalue* $\in \{\top, \bot\}$ s.t. $\mu \cup \{v := \mathit{truthvalue}\} \models \varphi$
  - $\mu \models \forall v.\varphi$ iff forall *truthvalue* $\in \{\top, \bot\}$, $\mu \cup \{v := \mathit{truthvalue}\} \models \varphi$
- Example: $\exists(b, c).((a \wedge b) \vee (c \wedge d)) \ = \ a \vee d$

Note

Naive expansion of quantifiers to propositional logic may cause a blow-up in size of the formulae

Complexity

NP-Completeness of SAT

- For N variables, there are up to 2^N truth assignments to be checked.
- The problem of deciding the satisfiability of a propositional formula is NP-complete

$\Longrightarrow$ The most important logical problems (validity, inference, entailment, equivalence, ...) can be straightforwardly reduced to (un)satisfiability, and are thus (co)NP-complete.

$$\Downarrow$$

No existing worst-case-polynomial algorithm.

POLARITY of subformulas

Polarity: the number of nested negations modulo 2.

- Positive/negative occurrences
  - φ occurs positively in φ;
  - if $\neg\varphi_1$ occurs positively [negatively] in φ, then φ_1 occurs negatively [positively] in φ
  - if $\varphi_1 \wedge \varphi_2$ or $\varphi_1 \vee \varphi_2$ occur positively [negatively] in φ, then φ_1 and φ_2 occur positively [negatively] in φ;
  - if $\varphi_1 \rightarrow \varphi_2$ occurs positively [negatively] in φ, then φ_1 occurs negatively [positively] in φ and φ_2 occurs positively [negatively] in φ;
  - if $\varphi_1 \leftrightarrow \varphi_2$ or $\varphi_1 \oplus \varphi_2$ occurs in φ, then φ_1 and φ_2 occur positively and negatively in φ;

Negative Normal Form (NNF)

- φ is in Negative normal form iff it is given only by the recursive applications of $\wedge, \vee$ to literals.
- every φ can be reduced into NNF:
  (i) substituting all $\rightarrow$'s and $\leftrightarrow$'s:

$$\begin{array}{lcl} \alpha \rightarrow \beta & \Longrightarrow & \neg\alpha \vee \beta \\ \alpha \leftrightarrow \beta & \Longrightarrow & (\neg\alpha \vee \beta) \wedge (\alpha \vee \neg\beta) \end{array}$$

  (ii) pushing down negations recursively:

$$\begin{array}{lcl} \neg(\alpha \wedge \beta) & \Longrightarrow & \neg\alpha \vee \neg\beta \\ \neg(\alpha \vee \beta) & \Longrightarrow & \neg\alpha \wedge \neg\beta \\ \neg\neg\alpha & \Longrightarrow & \alpha \end{array}$$

- Every non-atomic subformula in $NNF(\varphi)$ occurs with positive polarity only
  $\Longrightarrow$ a subformula ψ occurring with both polarities in φ is encoded as both $NNF(\psi)$ and $NNF(\neg\psi)$
- The reduction is linear if a DAG representation is used.
- Preserves the equivalence of formulas.

Negative Normal Form (NNF)

- φ is in Negative normal form iff it is given only by the recursive applications of $\wedge, \vee$ to literals.
- every φ can be reduced into NNF:
  (i) substituting all $\to$'s and $\leftrightarrow$'s:

$$\begin{array}{lll} \alpha \to \beta & \Longrightarrow & \neg\alpha \vee \beta \\ \alpha \leftrightarrow \beta & \Longrightarrow & (\neg\alpha \vee \beta) \wedge (\alpha \vee \neg\beta) \end{array}$$

  (ii) pushing down negations recursively:

$$\begin{array}{lll} \neg(\alpha \wedge \beta) & \Longrightarrow & \neg\alpha \vee \neg\beta \\ \neg(\alpha \vee \beta) & \Longrightarrow & \neg\alpha \wedge \neg\beta \\ \neg\neg\alpha & \Longrightarrow & \alpha \end{array}$$

- Every non-atomic subformula in $NNF(\varphi)$ occurs with positive polarity only
  $\Longrightarrow$ a subformula ψ occurring with both polarities in φ is encoded as both $NNF(\psi)$ and $NNF(\neg\psi)$
- The reduction is linear if a DAG representation is used.
- Preserves the equivalence of formulas.

Negative Normal Form (NNF)

- φ is in Negative normal form iff it is given only by the recursive applications of $\wedge, \vee$ to literals.
- every φ can be reduced into NNF:

  (i) substituting all $\rightarrow$'s and $\leftrightarrow$'s:

$$\begin{array}{lll} \alpha \rightarrow \beta & \Longrightarrow & \neg\alpha \vee \beta \\ \alpha \leftrightarrow \beta & \Longrightarrow & (\neg\alpha \vee \beta) \wedge (\alpha \vee \neg\beta) \end{array}$$

  (ii) pushing down negations recursively:

$$\begin{array}{lll} \neg(\alpha \wedge \beta) & \Longrightarrow & \neg\alpha \vee \neg\beta \\ \neg(\alpha \vee \beta) & \Longrightarrow & \neg\alpha \wedge \neg\beta \\ \neg\neg\alpha & \Longrightarrow & \alpha \end{array}$$

- Every non-atomic subformula in $NNF(\varphi)$ occurs with positive polarity only

  $\Longrightarrow$ a subformula ψ occurring with both polarities in φ is encoded as both $NNF(\psi)$ and $NNF(\neg\psi)$
- The reduction is linear if a DAG representation is used.
- Preserves the equivalence of formulas.

Negative Normal Form (NNF)

- φ is in Negative normal form iff it is given only by the recursive applications of $\wedge, \vee$ to literals.
- every φ can be reduced into NNF:

  (i) substituting all $\rightarrow$'s and $\leftrightarrow$'s:

$$\begin{array}{lcl} \alpha \rightarrow \beta & \Longrightarrow & \neg\alpha \vee \beta \\ \alpha \leftrightarrow \beta & \Longrightarrow & (\neg\alpha \vee \beta) \wedge (\alpha \vee \neg\beta) \end{array}$$

  (ii) pushing down negations recursively:

$$\begin{array}{lcl} \neg(\alpha \wedge \beta) & \Longrightarrow & \neg\alpha \vee \neg\beta \\ \neg(\alpha \vee \beta) & \Longrightarrow & \neg\alpha \wedge \neg\beta \\ \neg\neg\alpha & \Longrightarrow & \alpha \end{array}$$

- Every non-atomic subformula in *NNF*(φ) occurs with positive polarity only

  $\Longrightarrow$ a subformula ψ occurring with both polarities in φ is encoded as both *NNF*(ψ) and *NNF*$(\neg\psi)$
- The reduction is linear if a DAG representation is used.
- Preserves the equivalence of formulas.

Negative Normal Form (NNF)

- φ is in Negative normal form iff it is given only by the recursive applications of $\wedge, \vee$ to literals.
- every φ can be reduced into NNF:
  (i) substituting all $\rightarrow$'s and $\leftrightarrow$'s:

$$\begin{array}{lcl} \alpha \rightarrow \beta & \Longrightarrow & \neg\alpha \vee \beta \\ \alpha \leftrightarrow \beta & \Longrightarrow & (\neg\alpha \vee \beta) \wedge (\alpha \vee \neg\beta) \end{array}$$

  (ii) pushing down negations recursively:

$$\begin{array}{lcl} \neg(\alpha \wedge \beta) & \Longrightarrow & \neg\alpha \vee \neg\beta \\ \neg(\alpha \vee \beta) & \Longrightarrow & \neg\alpha \wedge \neg\beta \\ \neg\neg\alpha & \Longrightarrow & \alpha \end{array}$$

- Every non-atomic subformula in $NNF(\varphi)$ occurs with positive polarity only
  $\Longrightarrow$ a subformula ψ occurring with both polarities in φ is encoded as both $NNF(\psi)$ and $NNF(\neg\psi)$
- The reduction is linear if a DAG representation is used.
- Preserves the equivalence of formulas.

NNF: Example

$$(A_1 \leftrightarrow A_2) \leftrightarrow (A_3 \leftrightarrow A_4)$$

NNF: Example

$$(A_1 \leftrightarrow A_2) \leftrightarrow (A_3 \leftrightarrow A_4)$$
$$\Downarrow$$
$$((((A_1 \rightarrow A_2) \wedge (A_1 \leftarrow A_2)) \rightarrow ((A_3 \rightarrow A_4) \wedge (A_3 \leftarrow A_4)))\wedge$$
$$(((A_1 \rightarrow A_2) \wedge (A_1 \leftarrow A_2)) \leftarrow ((A_3 \rightarrow A_4) \wedge (A_3 \leftarrow A_4))))$$

NNF: Example

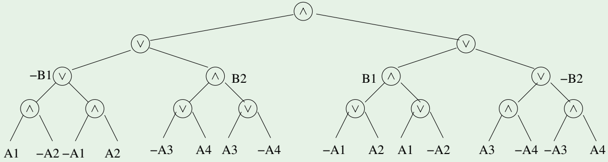
$$(A_1 \leftrightarrow A_2) \leftrightarrow (A_3 \leftrightarrow A_4)$$
$$\Downarrow$$
$$((((A_1 \rightarrow A_2) \wedge (A_1 \leftarrow A_2)) \rightarrow ((A_3 \rightarrow A_4) \wedge (A_3 \leftarrow A_4)))\wedge$$
$$(((A_1 \rightarrow A_2) \wedge (A_1 \leftarrow A_2)) \leftarrow ((A_3 \rightarrow A_4) \wedge (A_3 \leftarrow A_4))))$$
$$\Downarrow$$
$$((\neg((\neg A_1 \vee A_2) \wedge (A_1 \vee \neg A_2)) \vee ((\neg A_3 \vee A_4) \wedge (A_3 \vee \neg A_4)))\wedge$$
$$(((\neg A_1 \vee A_2) \wedge (A_1 \vee \neg A_2)) \vee \neg((\neg A_3 \vee A_4) \wedge (A_3 \vee \neg A_4))))$$

NNF: Example

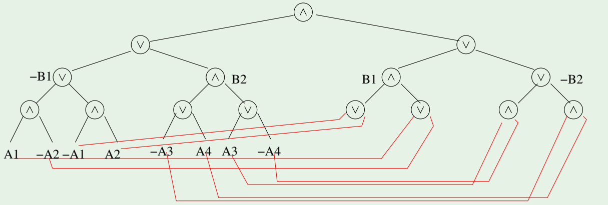
$$(A_1 \leftrightarrow A_2) \leftrightarrow (A_3 \leftrightarrow A_4)$$
$$\Downarrow$$
$$((((A_1 \rightarrow A_2) \wedge (A_1 \leftarrow A_2)) \rightarrow ((A_3 \rightarrow A_4) \wedge (A_3 \leftarrow A_4)))\wedge$$
$$(((A_1 \rightarrow A_2) \wedge (A_1 \leftarrow A_2)) \leftarrow ((A_3 \rightarrow A_4) \wedge (A_3 \leftarrow A_4))))$$
$$\Downarrow$$
$$((\neg((\neg A_1 \vee A_2) \wedge (A_1 \vee \neg A_2)) \vee ((\neg A_3 \vee A_4) \wedge (A_3 \vee \neg A_4)))\wedge$$
$$(((\neg A_1 \vee A_2) \wedge (A_1 \vee \neg A_2)) \vee \neg((\neg A_3 \vee A_4) \wedge (A_3 \vee \neg A_4))))$$
$$\Downarrow$$
$$((((A_1 \wedge \neg A_2) \vee (\neg A_1 \wedge A_2)) \vee ((\neg A_3 \vee A_4) \wedge (A_3 \vee \neg A_4)))\wedge$$
$$(((\neg A_1 \vee A_2) \wedge (A_1 \vee \neg A_2)) \vee ((A_3 \wedge \neg A_4) \vee (\neg A_3 \wedge A_4))))$$

NNF: Example [cont.]

Note



Tree Representation



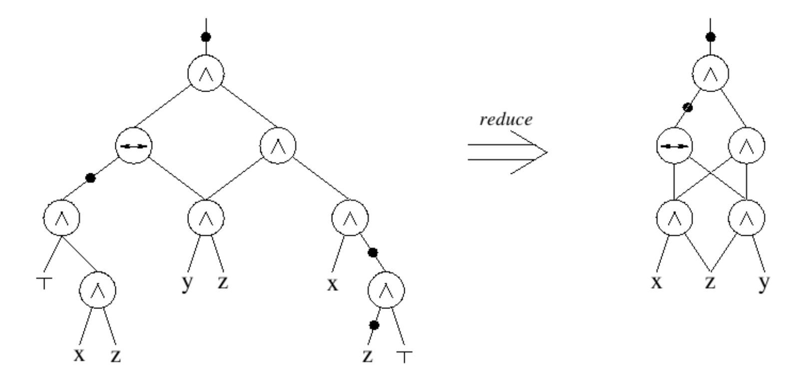
DAG Representation

For each non-literal subformula φ, φ and $\neg\varphi$ have different representations
$\Longrightarrow$ they are not shared.

Optimized polynomial representations

And-Inverter Graphs, Reduced Boolean Circuits, Boolean Expression Diagrams

- Maximize the sharing in DAG representations:
  $\{\wedge, \leftrightarrow, \neg\}$-only, negations on arcs, sorting of subformulae, lifting of $\neg$'s over $\leftrightarrow$'s,...



Conjunctive Normal Form (CNF)

- φ is in Conjunctive normal form iff it is a conjunction of disjunctions of literals:

$$\bigwedge_{i=1}^{L} \bigvee_{j_i=1}^{K_i} l_{j_i}$$

- the disjunctions of literals $\bigvee_{j_i=1}^{K_i} l_{j_i}$ are called clauses
- Easier to handle: list of lists of literals.
  $\Longrightarrow$ no reasoning on the recursive structure of the formula

Classic CNF Conversion $CNF(\varphi)$

- Every φ can be reduced into CNF by, e.g.,
  (i) expanding implications and equivalences:
  $$\begin{array}{lll} \alpha \to \beta & \Longrightarrow & \neg\alpha \vee \beta \\ \alpha \leftrightarrow \beta & \Longrightarrow & (\neg\alpha \vee \beta) \wedge (\alpha \vee \neg\beta) \end{array}$$
  (ii) pushing down negations recursively:
  $$\begin{array}{lll} \neg(\alpha \wedge \beta) & \Longrightarrow & \neg\alpha \vee \neg\beta \\ \neg(\alpha \vee \beta) & \Longrightarrow & \neg\alpha \wedge \neg\beta \\ \neg\neg\alpha & \Longrightarrow & \alpha \end{array}$$
  $\Longrightarrow$ Negation Normal Form, NNF (see previous slides)
  (iii) applying recursively the DeMorgan's Rule: $(\alpha \wedge \beta) \vee \gamma \quad \Longrightarrow \quad (\alpha \vee \gamma) \wedge (\beta \vee \gamma)$
- Resulting formula worst-case exponential:
  - ex: $||\text{CNF}(\bigvee_{i=1}^{N}(l_{i1} \wedge l_{i2})|| = ||(l_{11} \vee l_{21} \vee ... \vee l_{N1}) \wedge (l_{12} \vee l_{21} \vee ... \vee l_{N1}) \wedge ... \wedge (l_{12} \vee l_{22} \vee ... \vee l_{N2})|| = 2^N$
- $Atoms(CNF(\varphi)) = Atoms(\varphi)$
- $CNF(\varphi)$ is equivalent to φ.
- Rarely used in practice.

Classic CNF Conversion $CNF(\varphi)$

- Every φ can be reduced into CNF by, e.g.,
  (i) expanding implications and equivalences:
  $$\begin{array}{lll} \alpha \rightarrow \beta & \Longrightarrow & \neg\alpha \vee \beta \\ \alpha \leftrightarrow \beta & \Longrightarrow & (\neg\alpha \vee \beta) \wedge (\alpha \vee \neg\beta) \end{array}$$
  (ii) pushing down negations recursively:
  $$\begin{array}{lll} \neg(\alpha \wedge \beta) & \Longrightarrow & \neg\alpha \vee \neg\beta \\ \neg(\alpha \vee \beta) & \Longrightarrow & \neg\alpha \wedge \neg\beta \\ \neg\neg\alpha & \Longrightarrow & \alpha \end{array}$$
  $\Longrightarrow$ Negation Normal Form, NNF (see previous slides)
  (iii) applying recursively the DeMorgan's Rule: $(\alpha \wedge \beta) \vee \gamma \quad \Longrightarrow \quad (\alpha \vee \gamma) \wedge (\beta \vee \gamma)$
- Resulting formula worst-case exponential:
  - ex: $||CNF(\bigvee_{i=1}^{N}(l_{i1} \wedge l_{i2})|| = ||(l_{11} \vee l_{21} \vee \ldots \vee l_{N1}) \wedge (l_{12} \vee l_{21} \vee \ldots \vee l_{N1}) \wedge \ldots \wedge (l_{12} \vee l_{22} \vee \ldots \vee l_{N2})|| = 2^N$
- $Atoms(CNF(\varphi)) = Atoms(\varphi)$
- $CNF(\varphi)$ is equivalent to φ.
- Rarely used in practice.

Classic CNF Conversion $CNF(\varphi)$

- Every φ can be reduced into CNF by, e.g.,
  (i) expanding implications and equivalences:
  $$\begin{array}{lll} \alpha \to \beta & \Longrightarrow & \neg\alpha \vee \beta \\ \alpha \leftrightarrow \beta & \Longrightarrow & (\neg\alpha \vee \beta) \wedge (\alpha \vee \neg\beta) \end{array}$$
  (ii) pushing down negations recursively:
  $$\begin{array}{lll} \neg(\alpha \wedge \beta) & \Longrightarrow & \neg\alpha \vee \neg\beta \\ \neg(\alpha \vee \beta) & \Longrightarrow & \neg\alpha \wedge \neg\beta \\ \neg\neg\alpha & \Longrightarrow & \alpha \end{array}$$
  $\Longrightarrow$ Negation Normal Form, NNF (see previous slides)
  (iii) applying recursively the DeMorgan's Rule: $(\alpha \wedge \beta) \vee \gamma \quad \Longrightarrow \quad (\alpha \vee \gamma) \wedge (\beta \vee \gamma)$
- Resulting formula worst-case exponential:
  - ex: $||\text{CNF}(\bigvee_{i=1}^{N}(l_{i1} \wedge l_{i2})|| = ||(l_{11} \vee l_{21} \vee ... \vee l_{N1}) \wedge (l_{12} \vee l_{21} \vee ... \vee l_{N1}) \wedge ... \wedge (l_{12} \vee l_{22} \vee ... \vee l_{N2})|| = 2^N$
- $Atoms(CNF(\varphi)) = Atoms(\varphi)$
- $CNF(\varphi)$ is equivalent to φ.
- Rarely used in practice.

Classic CNF Conversion $CNF(\varphi)$

- Every φ can be reduced into CNF by, e.g.,
  (i) expanding implications and equivalences:
  $$\begin{array}{lll} \alpha \to \beta & \Longrightarrow & \neg\alpha \vee \beta \\ \alpha \leftrightarrow \beta & \Longrightarrow & (\neg\alpha \vee \beta) \wedge (\alpha \vee \neg\beta) \end{array}$$
  (ii) pushing down negations recursively:
  $$\begin{array}{lll} \neg(\alpha \wedge \beta) & \Longrightarrow & \neg\alpha \vee \neg\beta \\ \neg(\alpha \vee \beta) & \Longrightarrow & \neg\alpha \wedge \neg\beta \\ \neg\neg\alpha & \Longrightarrow & \alpha \end{array}$$
  $\Longrightarrow$ Negation Normal Form, NNF (see previous slides)
  (iii) applying recursively the DeMorgan's Rule: $(\alpha \wedge \beta) \vee \gamma \quad \Longrightarrow \quad (\alpha \vee \gamma) \wedge (\beta \vee \gamma)$
- Resulting formula worst-case exponential:
  - ex: $||\text{CNF}(\bigvee_{i=1}^{N}(l_{i1} \wedge l_{i2})|| = ||(l_{11} \vee l_{21} \vee \ldots \vee l_{N1}) \wedge (l_{12} \vee l_{21} \vee \ldots \vee l_{N1}) \wedge \ldots \wedge (l_{12} \vee l_{22} \vee \ldots \vee l_{N2})|| = 2^N$
- $Atoms(CNF(\varphi)) = Atoms(\varphi)$
- $CNF(\varphi)$ is equivalent to φ.
- Rarely used in practice.

Classic CNF Conversion $CNF(\varphi)$

- Every φ can be reduced into CNF by, e.g.,
  (i) expanding implications and equivalences:
  $$\alpha \rightarrow \beta \Longrightarrow \neg\alpha \vee \beta$$
  $$\alpha \leftrightarrow \beta \Longrightarrow (\neg\alpha \vee \beta) \wedge (\alpha \vee \neg\beta)$$
  (ii) pushing down negations recursively:
  $$\neg(\alpha \wedge \beta) \Longrightarrow \neg\alpha \vee \neg\beta$$
  $$\neg(\alpha \vee \beta) \Longrightarrow \neg\alpha \wedge \neg\beta$$
  $$\neg\neg\alpha \Longrightarrow \alpha$$
  $\Longrightarrow$ Negation Normal Form, NNF (see previous slides)
  (iii) applying recursively the DeMorgan's Rule: $(\alpha \wedge \beta) \vee \gamma \Longrightarrow (\alpha \vee \gamma) \wedge (\beta \vee \gamma)$
- Resulting formula worst-case exponential:
  - ex: $||\text{CNF}(\bigvee_{i=1}^{N}(l_{i1} \wedge l_{i2})|| = ||(l_{11} \vee l_{21} \vee ... \vee l_{N1}) \wedge (l_{12} \vee l_{21} \vee ... \vee l_{N1}) \wedge ... \wedge (l_{12} \vee l_{22} \vee ... \vee l_{N2})|| = 2^N$
- $Atoms(CNF(\varphi)) = Atoms(\varphi)$
- $CNF(\varphi)$ is equivalent to φ.
- Rarely used in practice.

Classic CNF Conversion $CNF(\varphi)$

- Every φ can be reduced into CNF by, e.g.,

  (i) expanding implications and equivalences:

  $$\begin{array}{lcl} \alpha \rightarrow \beta & \Longrightarrow & \neg\alpha \vee \beta \\ \alpha \leftrightarrow \beta & \Longrightarrow & (\neg\alpha \vee \beta) \wedge (\alpha \vee \neg\beta) \end{array}$$

  (ii) pushing down negations recursively:

  $$\begin{array}{lcl} \neg(\alpha \wedge \beta) & \Longrightarrow & \neg\alpha \vee \neg\beta \\ \neg(\alpha \vee \beta) & \Longrightarrow & \neg\alpha \wedge \neg\beta \\ \neg\neg\alpha & \Longrightarrow & \alpha \end{array}$$

  $\Longrightarrow$ Negation Normal Form, NNF (see previous slides)

  (iii) applying recursively the DeMorgan's Rule: $(\alpha \wedge \beta) \vee \gamma \quad \Longrightarrow \quad (\alpha \vee \gamma) \wedge (\beta \vee \gamma)$
- Resulting formula worst-case exponential:
  - ex: $||\text{CNF}(\bigvee_{i=1}^{N}(l_{i1} \wedge l_{i2})|| = ||(l_{11} \vee l_{21} \vee ... \vee l_{N1}) \wedge (l_{12} \vee l_{21} \vee ... \vee l_{N1}) \wedge ... \wedge (l_{12} \vee l_{22} \vee ... \vee l_{N2})|| = 2^N$
- $Atoms(CNF(\varphi)) = Atoms(\varphi)$
- $CNF(\varphi)$ is equivalent to φ.
- Rarely used in practice.

Classic CNF Conversion $CNF(\varphi)$

- Every φ can be reduced into CNF by, e.g.,
  (i) expanding implications and equivalences:
  $$\alpha \rightarrow \beta \quad \Longrightarrow \quad \neg\alpha \vee \beta$$
  $$\alpha \leftrightarrow \beta \quad \Longrightarrow \quad (\neg\alpha \vee \beta) \wedge (\alpha \vee \neg\beta)$$
  (ii) pushing down negations recursively:
  $$\neg(\alpha \wedge \beta) \quad \Longrightarrow \quad \neg\alpha \vee \neg\beta$$
  $$\neg(\alpha \vee \beta) \quad \Longrightarrow \quad \neg\alpha \wedge \neg\beta$$
  $$\neg\neg\alpha \quad \Longrightarrow \quad \alpha$$
  $\Longrightarrow$ Negation Normal Form, NNF (see previous slides)
  (iii) applying recursively the DeMorgan's Rule: $(\alpha \wedge \beta) \vee \gamma \quad \Longrightarrow \quad (\alpha \vee \gamma) \wedge (\beta \vee \gamma)$
- Resulting formula worst-case exponential:
  - ex: $||\mathrm{CNF}(\bigvee_{i=1}^{N}(l_{i1} \wedge l_{i2})|| = ||(l_{11} \vee l_{21} \vee ... \vee l_{N1}) \wedge (l_{12} \vee l_{21} \vee ... \vee l_{N1}) \wedge ... \wedge (l_{12} \vee l_{22} \vee ... \vee l_{N2})|| = 2^N$
- $Atoms(CNF(\varphi)) = Atoms(\varphi)$
- $CNF(\varphi)$ is equivalent to φ.
- Rarely used in practice.

Labeling CNF conversion $CNF_{label}(\varphi)$

Labeling CNF conversion $CNF_{label}(\varphi)$ (aka Tseitin's CNF-ization)

- Every φ can be reduced into CNF by, e.g., applying recursively bottom-up the rules:

$$\begin{array}{lll}
\varphi & \Longrightarrow & \varphi[(l_i \vee l_j)|B] \wedge CNF(B \leftrightarrow (l_i \vee l_j)) \\
\varphi & \Longrightarrow & \varphi[(l_i \wedge l_j)|B] \wedge CNF(B \leftrightarrow (l_i \wedge l_j)) \\
\varphi & \Longrightarrow & \varphi[(l_i \leftrightarrow l_j)|B] \wedge CNF(B \leftrightarrow (l_i \leftrightarrow l_j))
\end{array}$$

  l_i, l_j being literals and B being a "new" variable.
- Worst-case linear!
- $Atoms(CNF_{label}(\varphi)) \supseteq Atoms(\varphi)$
- $CNF_{label}(\varphi)$ is equi-satisfiable (but not equivalent) to φ.
  - moreover: $\exists B_1, ..., B_k.CNF_{label}(\varphi)$ equivalent to φ, s.t. $B_1, ..., B_k$ all fresh variables introduced
- Much more used than classic conversion in practice

Remark

With $CNF_{label}(\varphi)$, it is not a good idea to convert φ into $NNF(\varphi)$ upfront. Why?

Labeling CNF conversion $CNF_{label}(\varphi)$

Labeling CNF conversion $CNF_{label}(\varphi)$ (aka Tseitin's CNF-ization)

- Every φ can be reduced into CNF by, e.g., applying recursively bottom-up the rules:
  $$\begin{array}{lcl} \varphi & \Longrightarrow & \varphi[(l_i \vee l_j)|B] \wedge CNF(B \leftrightarrow (l_i \vee l_j)) \\ \varphi & \Longrightarrow & \varphi[(l_i \wedge l_j)|B] \wedge CNF(B \leftrightarrow (l_i \wedge l_j)) \\ \varphi & \Longrightarrow & \varphi[(l_i \leftrightarrow l_j)|B] \wedge CNF(B \leftrightarrow (l_i \leftrightarrow l_j)) \end{array}$$
  l_i, l_j being literals and B being a "new" variable.
- Worst-case linear!
- $Atoms(CNF_{label}(\varphi)) \supseteq Atoms(\varphi)$
- $CNF_{label}(\varphi)$ is equi-satisfiable (but not equivalent) to φ.
  - moreover: $\exists B_1, ..., B_k . CNF_{label}(\varphi)$ equivalent to φ, s.t. $B_1, ..., B_k$ all fresh variables introduced
- Much more used than classic conversion in practice

Remark

With $CNF_{label}(\varphi)$, it is not a good idea to convert φ into $NNF(\varphi)$ upfront. Why?

Labeling CNF conversion $CNF_{label}(\varphi)$

Labeling CNF conversion $CNF_{label}(\varphi)$ (aka Tseitin's CNF-ization)

- Every φ can be reduced into CNF by, e.g., applying recursively bottom-up the rules:

$$\begin{array}{lcl} \varphi & \Longrightarrow & \varphi[(l_i \vee l_j)|B] \wedge CNF(B \leftrightarrow (l_i \vee l_j)) \\ \varphi & \Longrightarrow & \varphi[(l_i \wedge l_j)|B] \wedge CNF(B \leftrightarrow (l_i \wedge l_j)) \\ \varphi & \Longrightarrow & \varphi[(l_i \leftrightarrow l_j)|B] \wedge CNF(B \leftrightarrow (l_i \leftrightarrow l_j)) \end{array}$$

  l_i, l_j being literals and B being a "new" variable.
- Worst-case linear!
- $Atoms(CNF_{label}(\varphi)) \supseteq Atoms(\varphi)$
- $CNF_{label}(\varphi)$ is equi-satisfiable (but not equivalent) to φ.
  - moreover: $\exists B_1, ..., B_k.CNF_{label}(\varphi)$ equivalent to φ, s.t. $B_1, ..., B_k$ all fresh variables introduced
- Much more used than classic conversion in practice

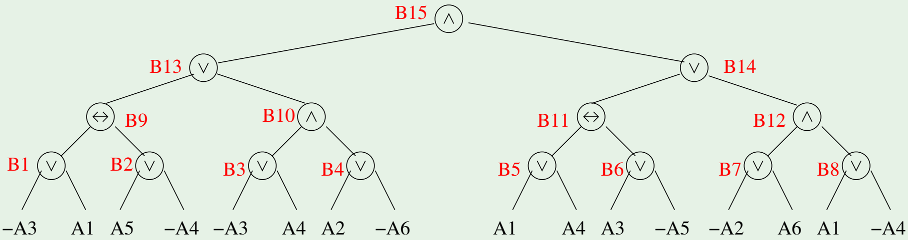
Remark

With $CNF_{label}(\varphi)$, it is not a good idea to convert φ into $NNF(\varphi)$ upfront. Why?

Labeling CNF conversion $CNF_{label}(\varphi)$ (cont.)

$CNF(B \leftrightarrow (l_i \vee l_j))$	$\Longleftrightarrow$	$(\neg B \vee l_i \vee l_j) \wedge$ $(B \vee \neg l_i) \wedge$ $(B \vee \neg l_j)$
$CNF(B \leftrightarrow (l_i \wedge l_j))$	$\Longleftrightarrow$	$(\neg B \vee l_i) \wedge$ $(\neg B \vee l_j) \wedge$ $(B \vee \neg l_i \vee \neg l_j)$
$CNF(B \leftrightarrow (l_i \leftrightarrow l_j))$	$\Longleftrightarrow$	$(\neg B \vee \neg l_i \vee l_j) \wedge$ $(\neg B \vee l_i \vee \neg l_j) \wedge$ $(B \vee l_i \vee l_j) \wedge$ $(B \vee \neg l_i \vee \neg l_j)$

Labeling CNF Conversion CNF_{label} – Example



$$
\begin{array}{lcl}
CNF(B_1 \leftrightarrow (\neg A_3 \vee A_1)) \wedge & & (\neg B_1 \vee \neg A_3 \vee A_1) \wedge (B_1 \vee A_3) \wedge (B_1 \vee \neg A_1) \wedge \\
\ldots \wedge & & \ldots \wedge \\
CNF(B_8 \leftrightarrow (A_1 \vee \neg A_4)) \wedge & & (\neg B_8 \vee A_1 \vee \neg A_4) \wedge (B_8 \vee \neg A_1) \wedge (B_8 \vee A_4) \wedge \\
CNF(B_9 \leftrightarrow (B_1 \leftrightarrow B_2)) \wedge & & (\neg B_9 \vee \neg B_1 \vee B_2) \wedge (\neg B_9 \vee B_1 \vee \neg B_2) \wedge \\
 & & (B_9 \vee B_1 \vee B_2) \wedge (B_9 \vee \neg B_1 \vee \neg B_2) \wedge \\
\ldots \wedge & = & \ldots \wedge \\
CNF(B_{12} \leftrightarrow (B_7 \wedge B_8)) \wedge & & (B_{12} \vee \neg B_7 \vee \neg B_8) \wedge (\neg B_{12} \vee B_7) \wedge (\neg B_{12} \vee B_8) \wedge \\
CNF(B_{13} \leftrightarrow (B_9 \vee B_{10})) \wedge & & (\neg B_{13} \vee B_9 \vee B_{10}) \wedge (B_{13} \vee \neg B_9) \wedge (B_{13} \vee \neg B_{10}) \wedge \\
CNF(B_{14} \leftrightarrow (B_{11} \vee B_{12})) \wedge & & (\neg B_{14} \vee B_{11} \vee B_{12}) \wedge (B_{14} \vee \neg B_{11}) \wedge (B_{14} \vee \neg B_{12}) \wedge \\
CNF(B_{15} \leftrightarrow (B_{13} \wedge B_{14})) \wedge & & (B_{15} \vee \neg B_{13} \vee \neg B_{14}) \wedge (\neg B_{15} \vee B_{13}) \wedge (\neg B_{15} \vee B_{14}) \wedge \\
B_{15} & & B_{15}
\end{array}
$$

Improved Labeling CNF conversion CNF_{label}

Polarity-based Labeling CNF conversion $CNF_{label}(\varphi)$ (aka Plaisted&Greenbaum CNF-ization)

- As in the previous case, applying instead the rules:

$$
\begin{array}{llllll}
\varphi & \Longrightarrow & \varphi[(l_i \vee l_j)|B] & \wedge\ CNF(B \rightarrow (l_i \vee l_j)) & \textit{if } (l_i \vee l_j) \textit{ pos.} \\
\varphi & \Longrightarrow & \varphi[(l_i \vee l_j)|B] & \wedge\ CNF((l_i \vee l_j) \rightarrow B) & \textit{if } (l_i \vee l_j) \textit{ neg.} \\
\varphi & \Longrightarrow & \varphi[(l_i \wedge l_j)|B] & \wedge\ CNF(B \rightarrow (l_i \wedge l_j)) & \textit{if } (l_i \wedge l_j) \textit{ pos.} \\
\varphi & \Longrightarrow & \varphi[(l_i \wedge l_j)|B] & \wedge\ CNF((l_i \wedge l_j) \rightarrow B) & \textit{if } (l_i \wedge l_j) \textit{ neg.} \\
\varphi & \Longrightarrow & \varphi[(l_i \leftrightarrow l_j)|B] & \wedge\ CNF(B \rightarrow (l_i \leftrightarrow l_j)) & \textit{if } (l_i \leftrightarrow l_j) \textit{ pos.} \\
\varphi & \Longrightarrow & \varphi[(l_i \leftrightarrow l_j)|B] & \wedge\ CNF((l_i \leftrightarrow l_j) \rightarrow B) & \textit{if } (l_i \leftrightarrow l_j) \textit{ neg.}
\end{array}
$$

- Smaller in size:

$$
\begin{array}{ll}
CNF(B \rightarrow (l_i \vee l_j)) & = (\neg B \vee l_i \vee l_j) \\
CNF(((l_i \vee l_j) \rightarrow B)) & = (\neg l_i \vee B) \wedge (\neg l_j \vee B)
\end{array}
$$

Improved Labeling CNF conversion CNF_{label}

Polarity-based Labeling CNF conversion $CNF_{label}(\varphi)$ (aka Plaisted&Greenbaum CNF-ization)

- As in the previous case, applying instead the rules:

$$
\begin{array}{lllll}
\varphi & \Longrightarrow & \varphi[(l_i \vee l_j)|B] & \wedge\, CNF(B \rightarrow (l_i \vee l_j)) & \text{if } (l_i \vee l_j) \text{ pos.} \\
\varphi & \Longrightarrow & \varphi[(l_i \vee l_j)|B] & \wedge\, CNF((l_i \vee l_j) \rightarrow B) & \text{if } (l_i \vee l_j) \text{ neg.} \\
\varphi & \Longrightarrow & \varphi[(l_i \wedge l_j)|B] & \wedge\, CNF(B \rightarrow (l_i \wedge l_j)) & \text{if } (l_i \wedge l_j) \text{ pos.} \\
\varphi & \Longrightarrow & \varphi[(l_i \wedge l_j)|B] & \wedge\, CNF((l_i \wedge l_j) \rightarrow B) & \text{if } (l_i \wedge l_j) \text{ neg.} \\
\varphi & \Longrightarrow & \varphi[(l_i \leftrightarrow l_j)|B] & \wedge\, CNF(B \rightarrow (l_i \leftrightarrow l_j)) & \text{if } (l_i \leftrightarrow l_j) \text{ pos.} \\
\varphi & \Longrightarrow & \varphi[(l_i \leftrightarrow l_j)|B] & \wedge\, CNF((l_i \leftrightarrow l_j) \rightarrow B) & \text{if } (l_i \leftrightarrow l_j) \text{ neg.}
\end{array}
$$

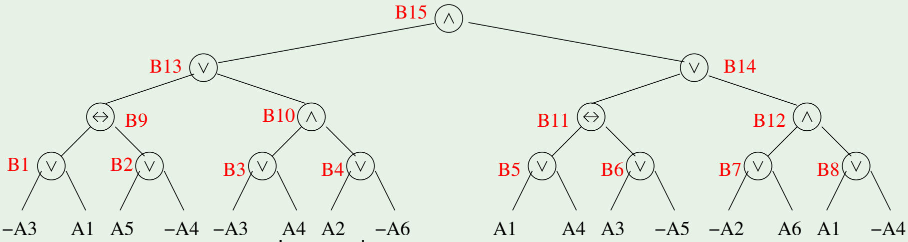
- Smaller in size:

$$
\begin{array}{ll}
CNF(B \rightarrow (l_i \vee l_j)) & = (\neg B \vee l_i \vee l_j) \\
CNF(((l_i \vee l_j) \rightarrow B)) & = (\neg l_i \vee B) \wedge (\neg l_j \vee B)
\end{array}
$$

Labeling CNF conversion $CNF_{label}(\varphi)$ (cont.)

$CNF(B \rightarrow (l_i \vee l_j))$	$\Longleftrightarrow$	$(\neg B \vee l_i \vee l_j)$
$CNF(B \leftarrow (l_i \vee l_j))$	$\Longleftrightarrow$	$(B \vee \neg l_i) \wedge$ $(B \vee \neg l_j)$
$CNF(B \rightarrow (l_i \wedge l_j))$	$\Longleftrightarrow$	$(\neg B \vee l_i) \wedge$ $(\neg B \vee l_j)$
$CNF(B \leftarrow (l_i \wedge l_j))$	$\Longleftrightarrow$	$(B \vee \neg l_i \neg l_j)$
$CNF(B \rightarrow (l_i \leftrightarrow l_j))$	$\Longleftrightarrow$	$(\neg B \vee \neg l_i \vee l_j) \wedge$ $(\neg B \vee l_i \vee \neg l_j)$
$CNF(B \leftarrow (l_i \leftrightarrow l_j))$	$\Longleftrightarrow$	$(B \vee l_i \vee l_j) \wedge$ $(B \vee \neg l_i \vee \neg l_j)$

Improved Labeling CNF conversion CNF_{label} – example



Basic

$CNF(B_1 \leftrightarrow (\neg A_3 \vee A_1)) \quad \wedge$
$\ldots \quad \wedge$
$CNF(B_8 \leftrightarrow (A_1 \vee \neg A_4)) \quad \wedge$
$CNF(B_9 \leftrightarrow (B_1 \leftrightarrow B_2)) \quad \wedge$
$\ldots \quad \wedge$
$CNF(B_{12} \leftrightarrow (B_7 \wedge B_8)) \quad \wedge$
$CNF(B_{13} \leftrightarrow (B_9 \vee B_{10})) \quad \wedge$
$CNF(B_{14} \leftrightarrow (B_{11} \vee B_{12})) \quad \wedge$
$CNF(B_{15} \leftrightarrow (B_{13} \wedge B_{14})) \quad \wedge$
B_{15}

Improved

$CNF(B_1 \leftrightarrow (\neg A_3 \vee A_1)) \quad \wedge$
$\ldots \quad \wedge$
$CNF(B_8 \rightarrow (A_1 \vee \neg A_4)) \quad \wedge$
$CNF(B_9 \rightarrow (B_1 \leftrightarrow B_2)) \quad \wedge$
$\ldots \quad \wedge$
$CNF(B_{12} \rightarrow (B_7 \wedge B_8)) \quad \wedge$
$CNF(B_{13} \rightarrow (B_9 \vee B_{10})) \quad \wedge$
$CNF(B_{14} \rightarrow (B_{11} \vee B_{12})) \quad \wedge$
$CNF(B_{15} \rightarrow (B_{13} \wedge B_{14})) \quad \wedge$
B_{15}

Labeling CNF conversion CNF_{label} – further improvements

- Do not apply CNF_{label} when not necessary:
  (e.g., $CNF_{label}(\varphi_1 \wedge \varphi_2) \Longrightarrow CNF_{label}(\varphi_1) \wedge \varphi_2$, if φ_2 already in CNF)
- Apply DeMorgan's rules where it is more effective:
  (e.g., $CNF_{label}(\varphi_1 \wedge (A \rightarrow (B \wedge C))) \Longrightarrow CNF_{label}(\varphi_1) \wedge (\neg A \vee B) \wedge (\neg A \vee C)$)
- Exploit the associativity of $\wedge$'s and $\vee$'s:
  $\ldots \underbrace{(A_1 \vee (A_2 \vee A_3))}_{B} \ldots \Longrightarrow \ldots CNF(B \leftrightarrow (A_1 \vee A_2 \vee A_3))\ldots$
- Before applying CNF_{label}, rewrite the initial formula so that to maximize the sharing of subformulas (RBC, BED)
- ...

Exercises

1. Consider the following Boolean formula φ:
   $\neg(((\neg A_1 \to A_2) \wedge (\neg A_3 \to A_4)) \vee ((A_5 \to A_6) \wedge (A_7 \to \neg A_8)))$
   Compute the Negative Normal Form of φ
2. Consider the following Boolean formula φ:
   $((\neg A_1 \wedge A_2) \vee (A_7 \wedge A_4) \vee (\neg A_3 \wedge \neg A_2) \vee (A_5 \wedge \neg A_4))$
   1. Produce the CNF formula $CNF(\varphi)$.
   2. Produce the CNF formula $CNF_{label}(\varphi)$.
   3. Produce the CNF formula $CNF_{label}(\varphi)$ (improved version)

   Do the same for the formula in Exercise (1)
3. Do the same as with Exercises (1) and (2), with the following formulas:
   - $(((A_1 \oplus A_2) \oplus A_3) \oplus A_4)$
   - $\neg((\neg A_1 \vee A_2) \wedge (A_7 \vee A_4) \wedge (\neg A_3 \vee \neg A_2) \wedge (A_5 \vee \neg A_4))$
4. For the same formulas as in Exercises (1), (2) and (3), compute $CNF_{label}(NNF(\varphi))$, and compare the result with $CNF_{label}(\varphi)$. What do you notice?

Outline

Outline

Propositional Reasoning: Generalities

- Automated Reasoning in Propositional Logic fundamental task
  - AI, formal verification, circuit synthesis, operational research,....
- Important in AI: $KB \models \alpha$: entail fact α from knowledge base KB (aka Model Checking: $M(KB) \subseteq M(\alpha)$)
  - typically $|KB| \gg |\alpha|$
- All propositional reasoning tasks reduced to satisfiability (SAT)
  - $KB \models \alpha \Longrightarrow$ SAT($KB \wedge \neg\alpha$)=false
  - input formula CNF-ized and fed to a SAT solver
- Current SAT solvers dramatically efficient:
  - handle industrial problems with $10^6 - 10^7$ variables & clauses!
  - used as backend engines in a variety of systems

Propositional Reasoning: Generalities

- Automated Reasoning in Propositional Logic fundamental task
  - AI, formal verification, circuit synthesis, operational research,....
- Important in AI: $KB \models \alpha$: entail fact α from knowledge base KB (aka Model Checking: $M(KB) \subseteq M(\alpha)$)
  - typically $|KB| \gg |\alpha|$
- All propositional reasoning tasks reduced to satisfiability (SAT)
  - $KB \models \alpha \Longrightarrow \text{SAT}(KB \wedge \neg\alpha)\text{=false}$
  - input formula CNF-ized and fed to a SAT solver
- Current SAT solvers dramatically efficient:
  - handle industrial problems with $10^6 - 10^7$ variables & clauses!
  - used as backend engines in a variety of systems

Propositional Reasoning: Generalities

- Automated Reasoning in Propositional Logic fundamental task
  - AI, formal verification, circuit synthesis, operational research,....
- Important in AI: $KB \models \alpha$: entail fact α from knowledge base KB (aka Model Checking: $M(KB) \subseteq M(\alpha)$)
  - typically $|KB| \gg |\alpha|$
- All propositional reasoning tasks reduced to satisfiability (SAT)
  - $KB \models \alpha \Longrightarrow \text{SAT}(KB \wedge \neg\alpha)\text{=false}$
  - input formula CNF-ized and fed to a SAT solver
- Current SAT solvers dramatically efficient:
  - handle industrial problems with $10^6 - 10^7$ variables & clauses!
  - used as backend engines in a variety of systems

Propositional Reasoning: Generalities

- Automated Reasoning in Propositional Logic fundamental task
  - AI, formal verification, circuit synthesis, operational research,....
- Important in AI: $KB \models \alpha$: entail fact α from knowledge base KB (aka Model Checking: $M(KB) \subseteq M(\alpha)$)
  - typically $|KB| \gg |\alpha|$
- All propositional reasoning tasks reduced to satisfiability (SAT)
  - $KB \models \alpha \Longrightarrow$ SAT($KB \wedge \neg\alpha$)=false
  - input formula CNF-ized and fed to a SAT solver
- Current SAT solvers dramatically efficient:
  - handle industrial problems with $10^6 - 10^7$ variables & clauses!
  - used as backend engines in a variety of systems

Truth Tables

- Exhaustive evaluation of all subformulas:

φ_1	φ_2	$\varphi_1 \wedge \varphi_2$	$\varphi_1 \vee \varphi_2$	$\varphi_1 \rightarrow \varphi_2$	$\varphi_1 \leftrightarrow \varphi_2$
$\bot$	$\bot$	$\bot$	$\bot$	$\top$	$\top$
$\bot$	$\top$	$\bot$	$\top$	$\top$	$\bot$
$\top$	$\bot$	$\bot$	$\top$	$\bot$	$\bot$
$\top$	$\top$	$\top$	$\top$	$\top$	$\top$

- Requires polynomial space (draw one line at a time).
- Requires analyzing $2^{|Atoms(\varphi)|}$ lines.
- Never used in practice.

Outline

The Resolution Rule

- Resolution: deduction of a new clause from a pair of clauses with exactly one incompatible variable (resolvent):

$$\frac{(\overbrace{l_1 \vee \ldots \vee l_k}^{common} \vee \overbrace{l}^{resolvent} \vee \overbrace{l'_{k+1} \vee \ldots \vee l'_m}^{C'}) \qquad (\overbrace{l_1 \vee \ldots \vee l_k}^{common} \vee \overbrace{\neg l}^{resolvent} \vee \overbrace{l''_{k+1} \vee \ldots \vee l''_n}^{C''})}{(\underbrace{l_1 \vee \ldots \vee l_k}_{common} \vee \underbrace{l'_{k+1} \vee \ldots \vee l'_m}_{C'} \vee \underbrace{l''_{k+1} \vee \ldots \vee l''_n}_{C''})}$$

- Ex: $\dfrac{(A \vee B \vee C \vee D \vee E) \qquad (A \vee B \vee \neg C \vee F)}{(A \vee B \vee D \vee E \vee F)}$

- Note: many standard inference rules subcases of resolution: (recall that $\alpha \to \beta \iff \neg\alpha \vee \beta$)

$$\frac{A \to B \quad B \to C}{A \to C}\,(trans.) \qquad \frac{A \quad A \to B}{B}\,(m.\ ponens) \qquad \frac{\neg B \quad A \to B}{\neg A}\,(m.\ tollens)$$

The Resolution Rule

- Resolution: deduction of a new clause from a pair of clauses with exactly one incompatible variable (resolvent):

$$\frac{(\overbrace{l_1 \vee \ldots \vee l_k}^{common} \vee \overbrace{l}^{resolvent} \vee \overbrace{l'_{k+1} \vee \ldots \vee l'_m}^{C'}) \qquad (\overbrace{l_1 \vee \ldots \vee l_k}^{common} \vee \overbrace{\neg l}^{resolvent} \vee \overbrace{l''_{k+1} \vee \ldots \vee l''_n}^{C''})}{(\underbrace{l_1 \vee \ldots \vee l_k}_{common} \vee \underbrace{l'_{k+1} \vee \ldots \vee l'_m}_{C'} \vee \underbrace{l''_{k+1} \vee \ldots \vee l''_n}_{C''})}$$

- Ex: $\dfrac{(A \vee B \vee C \vee D \vee E) \qquad (A \vee B \vee \neg C \vee F)}{(A \vee B \vee D \vee E \vee F)}$

- Note: many standard inference rules subcases of resolution:
(recall that $\alpha \to \beta \iff \neg\alpha \vee \beta$)

$$\frac{A \to B \quad B \to C}{A \to C}\ (trans.) \qquad \frac{A \quad A \to B}{B}\ (m.\ ponens) \qquad \frac{\neg B \quad A \to B}{\neg A}\ (m.\ tollens)$$

The Resolution Rule

- Resolution: deduction of a new clause from a pair of clauses with exactly one incompatible variable (resolvent):

$$\frac{(\overbrace{l_1 \vee \ldots \vee l_k}^{common} \vee \overbrace{l}^{resolvent} \vee \overbrace{l'_{k+1} \vee \ldots \vee l'_m}^{C'}) \qquad (\overbrace{l_1 \vee \ldots \vee l_k}^{common} \vee \overbrace{\neg l}^{resolvent} \vee \overbrace{l''_{k+1} \vee \ldots \vee l''_n}^{C''})}{(\underbrace{l_1 \vee \ldots \vee l_k}_{common} \vee \underbrace{l'_{k+1} \vee \ldots \vee l'_m}_{C'} \vee \underbrace{l''_{k+1} \vee \ldots \vee l''_n}_{C''})}$$

- Ex: $\dfrac{(A \vee B \vee C \vee D \vee E) \qquad (A \vee B \vee \neg C \vee F)}{(A \vee B \vee D \vee E \vee F)}$
- Note: many standard inference rules subcases of resolution: (recall that $\alpha \to \beta \iff \neg\alpha \vee \beta$)

$$\frac{A \to B \quad B \to C}{A \to C}\ (trans.) \qquad \frac{A \quad A \to B}{B}\ (m.\ ponens) \qquad \frac{\neg B \quad A \to B}{\neg A}\ (m.\ tollens)$$

Improvements: Subsumption & Unit Propagation

Alternative "set" notation (Γ clause set):

$$\frac{\Gamma, \phi_1, ..\phi_n}{\Gamma, \phi'_1, ..\phi'_{n'}} \qquad \left(e.g., \frac{\Gamma, C_1 \vee p, C_2 \vee \neg p}{\Gamma, C_1 \vee p, C_2 \vee \neg p, C_1 \vee C_2,} \right)$$

- Removal of valid clauses: $\dfrac{\Gamma \wedge (p \vee \neg p \vee C)}{\Gamma}$
- Clause Subsumption (C clause): $\dfrac{\Gamma \wedge C \wedge (C \vee \bigvee_i l_i)}{\Gamma \wedge (C)}$
- Unit Resolution: $\dfrac{\Gamma \wedge (l) \wedge (\neg l \vee \bigvee_i l_i)}{\Gamma \wedge (l) \wedge (\bigvee_i l_i)}$
- Unit Subsumption: $\dfrac{\Gamma \wedge (l) \wedge (l \vee \bigvee_i l_i)}{\Gamma \wedge (l)}$
- Unit Propagation = Unit Resolution + Unit Subsumption

"Deterministic" rule: applied before other "non-deterministic" rules!

Improvements: Subsumption & Unit Propagation

Alternative "set" notation (Γ clause set):

$$\frac{\Gamma, \phi_1, ..\phi_n}{\Gamma, \phi'_1, ..\phi'_{n'}} \qquad \left(e.g., \frac{\Gamma, C_1 \vee p, C_2 \vee \neg p}{\Gamma, C_1 \vee p, C_2 \vee \neg p, C_1 \vee C_2,} \right)$$

- Removal of valid clauses: $$\frac{\Gamma \wedge (p \vee \neg p \vee C)}{\Gamma}$$
- Clause Subsumption (C clause): $$\frac{\Gamma \wedge C \wedge (C \vee \bigvee_i l_i)}{\Gamma \wedge (C)}$$
- Unit Resolution: $$\frac{\Gamma \wedge (l) \wedge (\neg l \vee \bigvee_i l_i)}{\Gamma \wedge (l) \wedge (\bigvee_i l_i)}$$
- Unit Subsumption: $$\frac{\Gamma \wedge (l) \wedge (l \vee \bigvee_i l_i)}{\Gamma \wedge (l)}$$
- Unit Propagation = Unit Resolution + Unit Subsumption

"Deterministic" rule: applied before other "non-deterministic" rules!

Improvements: Subsumption & Unit Propagation

Alternative "set" notation (Γ clause set):

$$\frac{\Gamma, \phi_1, ..\phi_n}{\Gamma, \phi'_1, ..\phi'_{n'}} \qquad \left(e.g., \frac{\Gamma, C_1 \vee p, C_2 \vee \neg p}{\Gamma, C_1 \vee p, C_2 \vee \neg p, C_1 \vee C_2,} \right)$$

- Removal of valid clauses: $$\frac{\Gamma \wedge (p \vee \neg p \vee C)}{\Gamma}$$
- Clause Subsumption (C clause): $$\frac{\Gamma \wedge C \wedge (C \vee \bigvee_i l_i)}{\Gamma \wedge (C)}$$
- Unit Resolution: $$\frac{\Gamma \wedge (l) \wedge (\neg l \vee \bigvee_i l_i)}{\Gamma \wedge (l) \wedge (\bigvee_i l_i)}$$
- Unit Subsumption: $$\frac{\Gamma \wedge (l) \wedge (l \vee \bigvee_i l_i)}{\Gamma \wedge (l)}$$
- Unit Propagation = Unit Resolution + Unit Subsumption

"Deterministic" rule: applied before other "non-deterministic" rules!

Improvements: Subsumption & Unit Propagation

Alternative "set" notation (Γ clause set):

$$\frac{\Gamma, \phi_1, ..\phi_n}{\Gamma, \phi'_1, ..\phi'_{n'}} \qquad \left(e.g., \frac{\Gamma, C_1 \vee p, C_2 \vee \neg p}{\Gamma, C_1 \vee p, C_2 \vee \neg p, C_1 \vee C_2,} \right)$$

- Removal of valid clauses: $$\frac{\Gamma \wedge (p \vee \neg p \vee C)}{\Gamma}$$
- Clause Subsumption (C clause): $$\frac{\Gamma \wedge C \wedge (C \vee \bigvee_i l_i)}{\Gamma \wedge (C)}$$
- Unit Resolution: $$\frac{\Gamma \wedge (l) \wedge (\neg l \vee \bigvee_i l_i)}{\Gamma \wedge (l) \wedge (\bigvee_i l_i)}$$
- Unit Subsumption: $$\frac{\Gamma \wedge (l) \wedge (l \vee \bigvee_i l_i)}{\Gamma \wedge (l)}$$
- Unit Propagation = Unit Resolution + Unit Subsumption

"Deterministic" rule: applied before other "non-deterministic" rules!

Improvements: Subsumption & Unit Propagation

Alternative "set" notation (Γ clause set):

$$\frac{\Gamma, \phi_1, ..\phi_n}{\Gamma, \phi'_1, ..\phi'_{n'}} \qquad \left(e.g., \frac{\Gamma, C_1 \vee p, C_2 \vee \neg p}{\Gamma, C_1 \vee p, C_2 \vee \neg p, C_1 \vee C_2,} \right)$$

- Removal of valid clauses: $$\frac{\Gamma \wedge (p \vee \neg p \vee C)}{\Gamma}$$
- Clause Subsumption (C clause): $$\frac{\Gamma \wedge C \wedge (C \vee \bigvee_i l_i)}{\Gamma \wedge (C)}$$
- Unit Resolution: $$\frac{\Gamma \wedge (l) \wedge (\neg l \vee \bigvee_i l_i)}{\Gamma \wedge (l) \wedge (\bigvee_i l_i)}$$
- Unit Subsumption: $$\frac{\Gamma \wedge (l) \wedge (l \vee \bigvee_i l_i)}{\Gamma \wedge (l)}$$
- Unit Propagation = Unit Resolution + Unit Subsumption

"Deterministic" rule: applied before other "non-deterministic" rules!

Improvements: Subsumption & Unit Propagation

Alternative "set" notation (Γ clause set):

$$\frac{\Gamma, \phi_1, ..\phi_n}{\Gamma, \phi'_1, ..\phi'_{n'}} \quad \left(e.g., \frac{\Gamma, C_1 \vee p, C_2 \vee \neg p}{\Gamma, C_1 \vee p, C_2 \vee \neg p, C_1 \vee C_2,} \right)$$

- Removal of valid clauses: $$\frac{\Gamma \wedge (p \vee \neg p \vee C)}{\Gamma}$$
- Clause Subsumption (C clause): $$\frac{\Gamma \wedge C \wedge (C \vee \bigvee_i l_i)}{\Gamma \wedge (C)}$$
- Unit Resolution: $$\frac{\Gamma \wedge (l) \wedge (\neg l \vee \bigvee_i l_i)}{\Gamma \wedge (l) \wedge (\bigvee_i l_i)}$$
- Unit Subsumption: $$\frac{\Gamma \wedge (l) \wedge (l \vee \bigvee_i l_i)}{\Gamma \wedge (l)}$$
- Unit Propagation = Unit Resolution + Unit Subsumption

"Deterministic" rule: applied before other "non-deterministic" rules!

Improvements: Subsumption & Unit Propagation

Alternative "set" notation (Γ clause set):

$$\frac{\Gamma, \phi_1, ..\phi_n}{\Gamma, \phi'_1, ..\phi'_{n'}} \qquad \left(e.g., \ \frac{\Gamma, C_1 \vee p, C_2 \vee \neg p}{\Gamma, C_1 \vee p, C_2 \vee \neg p, C_1 \vee C_2,} \right)$$

- Removal of valid clauses: $$\frac{\Gamma \wedge (p \vee \neg p \vee C)}{\Gamma}$$
- Clause Subsumption (C clause): $$\frac{\Gamma \wedge C \wedge (C \vee \bigvee_i l_i)}{\Gamma \wedge (C)}$$
- Unit Resolution: $$\frac{\Gamma \wedge (l) \wedge (\neg l \vee \bigvee_i l_i)}{\Gamma \wedge (l) \wedge (\bigvee_i l_i)}$$
- Unit Subsumption: $$\frac{\Gamma \wedge (l) \wedge (l \vee \bigvee_i l_i)}{\Gamma \wedge (l)}$$
- Unit Propagation = Unit Resolution + Unit Subsumption

"Deterministic" rule: applied before other "non-deterministic" rules!

Remark

What happens with more than 1 resolvent?

- Common mistake: the following is <u>not</u> a correct application of the resolution rule:

$$\frac{\Gamma,\ (C_1 \vee l_1 \vee l_2),\ (C_2 \vee \neg l_1 \vee \neg l_2)}{\Gamma,\ (C_1 \vee l_1 \vee l_2),\ (C_2 \vee \neg l_1 \vee \neg l_2),\ (C_1 \vee C_2)}$$

- Rather, a correct application would be:

$$\frac{\Gamma,\ (C_1 \vee l_1 \vee l_2),\ (C_2 \vee \neg l_1 \vee \neg l_2)}{\Gamma,\ (C_1 \vee l_1 \vee l_2),\ (C_2 \vee \neg l_1 \vee \neg l_2),\ (C_1 \vee l_2 \vee C_2 \vee \neg l_2)}$$

... but $(C_1 \vee l_2 \vee C_2 \vee \neg l_2)$ is valid and should be removed

$\Longrightarrow$ no clause is produced

Remark

What happens with more than 1 resolvent?

- Common mistake: the following is not a correct application of the resolution rule:

$$\frac{\Gamma,\ (C_1 \vee l_1 \vee l_2),\ (C_2 \vee \neg l_1 \vee \neg l_2)}{\Gamma,\ (C_1 \vee l_1 \vee l_2),\ (C_2 \vee \neg l_1 \vee \neg l_2),\ (C_1 \vee C_2)}$$

- Rather, a correct application would be:

$$\frac{\Gamma,\ (C_1 \vee l_1 \vee l_2),\ (C_2 \vee \neg l_1 \vee \neg l_2)}{\Gamma,\ (C_1 \vee l_1 \vee l_2),\ (C_2 \vee \neg l_1 \vee \neg l_2),\ (C_1 \vee l_2 \vee C_2 \vee \neg l_2)}$$

... but $(C_1 \vee l_2 \vee C_2 \vee \neg l_2)$ is valid and should be removed

$\Longrightarrow$ no clause is produced

Remark

What happens with more than 1 resolvent?

- Common mistake: the following is not a correct application of the resolution rule:

$$\frac{\Gamma,\ (C_1 \vee l_1 \vee l_2),\ (C_2 \vee \neg l_1 \vee \neg l_2)}{\Gamma,\ (C_1 \vee l_1 \vee l_2),\ (C_2 \vee \neg l_1 \vee \neg l_2),\ (C_1 \vee C_2)}$$

- Rather, a correct application would be:

$$\frac{\Gamma,\ (C_1 \vee l_1 \vee l_2),\ (C_2 \vee \neg l_1 \vee \neg l_2)}{\Gamma,\ (C_1 \vee l_1 \vee l_2),\ (C_2 \vee \neg l_1 \vee \neg l_2),\ (C_1 \vee l_2 \vee C_2 \vee \neg l_2)}$$

... but $(C_1 \vee l_2 \vee C_2 \vee \neg l_2)$ is valid and should be removed

$\Longrightarrow$ no clause is produced

Basic Propositional Inference: Resolution [33, 10]

- Assume input formula in CNF
  - if not, apply Tseitin CNF-ization first

$\Longrightarrow$ φ is represented as a set of clauses

- Search for a refutation of φ (is φ unsatisfiable?)
  - recall: $\alpha \models \beta$ iff $\alpha \wedge \neg\beta$ unsatisfiable
- Basic idea: apply iteratively the resolution rule to pairs of clauses with a conflicting literal, producing novel clauses, until either
  - a false clause is generated, or
  - the resolution rule is no more applicable
- Correct: if returns an empty clause, then φ unsat ($\alpha \models \beta$)
- Complete: if φ unsat ($\alpha \models \beta$), then it returns an empty clause
- Time-inefficient
- Very Memory-inefficient (exponential in memory)
- Many different strategies

Basic Propositional Inference: Resolution [33, 10]

- Assume input formula in CNF
  - if not, apply Tseitin CNF-ization first

$\Longrightarrow$ φ is represented as a set of clauses

- Search for a refutation of φ (is φ unsatisfiable?)
  - recall: $\alpha \models \beta$ iff $\alpha \wedge \neg\beta$ unsatisfiable
- Basic idea: apply iteratively the resolution rule to pairs of clauses with a conflicting literal, producing novel clauses, until either
  - a false clause is generated, or
  - the resolution rule is no more applicable
- Correct: if returns an empty clause, then φ unsat ($\alpha \models \beta$)
- Complete: if φ unsat ($\alpha \models \beta$), then it returns an empty clause
- Time-inefficient
- Very Memory-inefficient (exponential in memory)
- Many different strategies

Basic Propositional Inference: Resolution [33, 10]

- Assume input formula in CNF
  - if not, apply Tseitin CNF-ization first

$\Longrightarrow$ φ is represented as a set of clauses

- Search for a refutation of φ (is φ unsatisfiable?)
  - recall: $\alpha \models \beta$ iff $\alpha \wedge \neg\beta$ unsatisfiable
- Basic idea: apply iteratively the resolution rule to pairs of clauses with a conflicting literal, producing novel clauses, until either
  - a false clause is generated, or
  - the resolution rule is no more applicable
- Correct: if returns an empty clause, then φ unsat ($\alpha \models \beta$)
- Complete: if φ unsat ($\alpha \models \beta$), then it returns an empty clause
- Time-inefficient
- Very Memory-inefficient (exponential in memory)
- Many different strategies

Basic Propositional Inference: Resolution [33, 10]

- Assume input formula in CNF
  - if not, apply Tseitin CNF-ization first

$\Longrightarrow$ φ is represented as a set of clauses

- Search for a refutation of φ (is φ unsatisfiable?)
  - recall: $\alpha \models \beta$ iff $\alpha \wedge \neg\beta$ unsatisfiable
- Basic idea: apply iteratively the resolution rule to pairs of clauses with a conflicting literal, producing novel clauses, until either
  - a false clause is generated, or
  - the resolution rule is no more applicable
- Correct: if returns an empty clause, then φ unsat ($\alpha \models \beta$)
- Complete: if φ unsat ($\alpha \models \beta$), then it returns an empty clause
- Time-inefficient
- Very Memory-inefficient (exponential in memory)
- Many different strategies

Basic Propositional Inference: Resolution [33, 10]

- Assume input formula in CNF
  - if not, apply Tseitin CNF-ization first

$\Longrightarrow$ φ is represented as a set of clauses

- Search for a refutation of φ (is φ unsatisfiable?)
  - recall: $\alpha \models \beta$ iff $\alpha \wedge \neg\beta$ unsatisfiable
- Basic idea: apply iteratively the resolution rule to pairs of clauses with a conflicting literal, producing novel clauses, until either
  - a false clause is generated, or
  - the resolution rule is no more applicable
- Correct: if returns an empty clause, then φ unsat ($\alpha \models \beta$)
- Complete: if φ unsat ($\alpha \models \beta$), then it returns an empty clause
- Time-inefficient
- Very Memory-inefficient (exponential in memory)
- Many different strategies

Basic Propositional Inference: Resolution [33, 10]

- Assume input formula in CNF
  - if not, apply Tseitin CNF-ization first

$\Longrightarrow$ φ is represented as a set of clauses

- Search for a refutation of φ (is φ unsatisfiable?)
  - recall: $\alpha \models \beta$ iff $\alpha \wedge \neg\beta$ unsatisfiable
- Basic idea: apply iteratively the resolution rule to pairs of clauses with a conflicting literal, producing novel clauses, until either
  - a false clause is generated, or
  - the resolution rule is no more applicable
- Correct: if returns an empty clause, then φ unsat ($\alpha \models \beta$)
- Complete: if φ unsat ($\alpha \models \beta$), then it returns an empty clause
- Time-inefficient
- Very Memory-inefficient (exponential in memory)
- Many different strategies

Basic Propositional Inference: Resolution [33, 10]

- Assume input formula in CNF
  - if not, apply Tseitin CNF-ization first

$\Longrightarrow$ φ is represented as a set of clauses

- Search for a refutation of φ (is φ unsatisfiable?)
  - recall: $\alpha \models \beta$ iff $\alpha \land \neg\beta$ unsatisfiable
- Basic idea: apply iteratively the resolution rule to pairs of clauses with a conflicting literal, producing novel clauses, until either
  - a false clause is generated, or
  - the resolution rule is no more applicable
- Correct: if returns an empty clause, then φ unsat ($\alpha \models \beta$)
- Complete: if φ unsat ($\alpha \models \beta$), then it returns an empty clause
- Time-inefficient
- Very Memory-inefficient (exponential in memory)
- Many different strategies

Basic Propositional Inference: Resolution [33, 10]

- Assume input formula in CNF
  - if not, apply Tseitin CNF-ization first

$\Longrightarrow$ φ is represented as a set of clauses

- Search for a refutation of φ (is φ unsatisfiable?)
  - recall: $\alpha \models \beta$ iff $\alpha \wedge \neg\beta$ unsatisfiable
- Basic idea: apply iteratively the resolution rule to pairs of clauses with a conflicting literal, producing novel clauses, until either
  - a false clause is generated, or
  - the resolution rule is no more applicable
- Correct: if returns an empty clause, then φ unsat ($\alpha \models \beta$)
- Complete: if φ unsat ($\alpha \models \beta$), then it returns an empty clause
- Time-inefficient
- Very Memory-inefficient (exponential in memory)
- Many different strategies

Resolution: basic strategy [10]

function $DP(\Gamma)$
  if $\bot \in \Gamma$ /* unsat */
    then return *False*;
  if (Resolve() is no more applicable to Γ) /* sat */
    then return *True*;
  if {a unit clause (l) occurs in Γ} /* unit */
    then $\Gamma := Unit\_Propagate(l, \Gamma)$);
    return $DP(\Gamma)$
  $A := select\text{-}variable(\Gamma)$; /* resolve */
  $\Gamma = \Gamma \cup \bigcup_{A \in C', \neg A \in C''} \{Resolve(C', C'')\} \setminus \bigcup_{A \in C', \neg A \in C''} \{C', C''\}$;
  return $DP(\Gamma)$

Hint: drops one variable $A \in Atoms(\Gamma)$ at a time

Resolution: Examples

$$(A_1 \vee A_2)\ \ (A_1 \vee \neg A_2)\ \ (\neg A_1 \vee A_2)\ \ (\neg A_1 \vee \neg A_2)$$

$$\Downarrow$$

$$(A_2)\ \ (A_2 \vee \neg A_2)\ \ (\neg A_2 \vee A_2)\ \ (\neg A_2)$$

$$\Downarrow$$

$$\perp$$

$\Longrightarrow$ UNSAT

Resolution: Examples

$$(A_1 \vee A_2)\ \ (A_1 \vee \neg A_2)\ \ (\neg A_1 \vee A_2)\ \ (\neg A_1 \vee \neg A_2)$$

$$\Downarrow$$

$$(A_2)\ \ (A_2 \vee \neg A_2)\ \ (\neg A_2 \vee A_2)\ \ (\neg A_2)$$

$$\Downarrow$$

$$\bot$$

$\Longrightarrow$ UNSAT

Resolution: Examples

$$(A_1 \vee A_2) \;\; (A_1 \vee \neg A_2) \;\; (\neg A_1 \vee A_2) \;\; (\neg A_1 \vee \neg A_2)$$
$$\Downarrow$$
$$(A_2) \;\; (A_2 \vee \neg A_2) \;\; (\neg A_2 \vee A_2) \;\; (\neg A_2)$$
$$\Downarrow$$
$$\perp$$

$\Longrightarrow$ UNSAT

Resolution: Examples

$$(A_1 \vee A_2)\ \ (A_1 \vee \neg A_2)\ \ (\neg A_1 \vee A_2)\ \ (\neg A_1 \vee \neg A_2)$$
$$\Downarrow$$
$$(A_2)\ \ (A_2 \vee \neg A_2)\ \ (\neg A_2 \vee A_2)\ \ (\neg A_2)$$
$$\Downarrow$$
$$\bot$$

$\Longrightarrow$ UNSAT

Resolution: Examples (cont.)

$$(A \vee B \vee C) \;\; (B \vee \neg C \vee \neg F) \;\; (\neg B \vee E)$$

$$\Downarrow$$

$$(A \vee C \vee E) \;\; (\neg C \vee \neg F \vee E)$$

$$\Downarrow$$

$$(A \vee E \vee \neg F)$$

$\Longrightarrow$ SAT

Resolution: Examples (cont.)

$$(A \lor B \lor C) \;\; (B \lor \neg C \lor \neg F) \;\; (\neg B \lor E)$$
$$\Downarrow$$
$$(A \lor C \lor E) \;\; (\neg C \lor \neg F \lor E)$$
$$\Downarrow$$
$$(A \lor E \lor \neg F)$$

$\Longrightarrow$ SAT

Resolution: Examples (cont.)

$$(A \lor B \lor C) \;\; (B \lor \neg C \lor \neg F) \;\; (\neg B \lor E)$$
$$\Downarrow$$
$$(A \lor C \lor E) \;\; (\neg C \lor \neg F \lor E)$$
$$\Downarrow$$
$$(A \lor E \lor \neg F)$$

$\Longrightarrow$ SAT

Resolution: Examples (cont.)

$$(A \lor B \lor C) \;\; (B \lor \neg C \lor \neg F) \;\; (\neg B \lor E)$$
$$\Downarrow$$
$$(A \lor C \lor E) \;\; (\neg C \lor \neg F \lor E)$$
$$\Downarrow$$
$$(A \lor E \lor \neg F)$$

$\Longrightarrow$ SAT

Resolution: Examples

$$(A \vee B) \;\; (A \vee \neg B) \;\; (\neg A \vee C) \;\; (\neg A \vee \neg C)$$

$$\Downarrow$$

$$(A) \;\; (\neg A \vee C) \;\; (\neg A \vee \neg C)$$

$$\Downarrow$$

$$(C) \;\; (\neg C)$$

$$\Downarrow$$

$$\bot$$

$\Longrightarrow$ UNSAT

Resolution: Examples

$$(A \lor B) \;\; (A \lor \neg B) \;\; (\neg A \lor C) \;\; (\neg A \lor \neg C)$$
$$\Downarrow$$
$$(A) \;\; (\neg A \lor C) \;\; (\neg A \lor \neg C)$$
$$\Downarrow$$
$$(C) \;\; (\neg C)$$
$$\Downarrow$$
$$\bot$$

$\Longrightarrow$ UNSAT

Resolution: Examples

$$(A \vee B) \;\; (A \vee \neg B) \;\; (\neg A \vee C) \;\; (\neg A \vee \neg C)$$
$$\Downarrow$$
$$(A) \;\; (\neg A \vee C) \;\; (\neg A \vee \neg C)$$
$$\Downarrow$$
$$(C) \;\; (\neg C)$$
$$\Downarrow$$
$$\perp$$

$\Longrightarrow$ UNSAT

Resolution: Examples

$$(A \vee B) \;\; (A \vee \neg B) \;\; (\neg A \vee C) \;\; (\neg A \vee \neg C)$$
$$\Downarrow$$
$$(A) \;\; (\neg A \vee C) \;\; (\neg A \vee \neg C)$$
$$\Downarrow$$
$$(C) \;\; (\neg C)$$
$$\Downarrow$$
$$\bot$$

$\Longrightarrow$ UNSAT

Resolution – summary

- Requires CNF
- Γ may blow up
  $\Longrightarrow$ May require exponential space
- Not very much used in Boolean reasoning (unless integrated with DPLL procedure in recent implementations)

Outline

Semantic tableaux [39]

- Search for an assignment satisfying φ
- applies recursively elimination rules to the connectives
- If a branch contains A_i and $\neg A_i$, (ψ_i and $\neg\psi_i$) for some i, the branch is closed, otherwise it is open.
- if no rule can be applied to an open branch μ, then $\mu \models \varphi$;
- if all branches are closed, the formula is not satisfiable;

Tableau elimination rules

$$\frac{\Gamma, (\varphi_1 \wedge \varphi_2)}{\Gamma, \varphi_1, \varphi_2} \qquad \frac{\Gamma, \neg(\varphi_1 \vee \varphi_2)}{\Gamma, \neg\varphi_1, \neg\varphi_2} \qquad \frac{\Gamma, \neg(\varphi_1 \rightarrow \varphi_2)}{\Gamma, \varphi_1, \neg\varphi_2} \qquad (\wedge\text{-elimination})$$

$$\frac{\Gamma, \neg\neg\varphi}{\Gamma, \varphi} \qquad (\neg\neg\text{-elimination})$$

$$\frac{\Gamma, (\varphi_1 \vee \varphi_2)}{\Gamma, \varphi_1 \qquad \Gamma, \varphi_2} \qquad \frac{\Gamma, \neg(\varphi_1 \wedge \varphi_2)}{\Gamma, \neg\varphi_1 \qquad \Gamma, \neg\varphi_2} \qquad \frac{\Gamma, (\varphi_1 \rightarrow \varphi_2)}{\Gamma, \neg\varphi_1 \qquad \Gamma, \varphi_2} \qquad (\vee\text{-elimination})$$

$$\frac{\Gamma, (\varphi_1 \leftrightarrow \varphi_2)}{\Gamma, \varphi_1, \varphi_2 \qquad \Gamma, \neg\varphi_1\neg\varphi_2} \qquad \frac{\Gamma, \neg(\varphi_1 \leftrightarrow \varphi_2)}{\Gamma, \varphi_1, \neg\varphi_2 \qquad \Gamma, \neg\varphi_1\varphi_2} \qquad (\leftrightarrow\text{ -elimination}).$$

Semantic Tableaux – Example

$$\varphi = (A_1 \vee A_2) \wedge (A_1 \vee \neg A_2) \wedge (\neg A_1 \vee A_2) \wedge (\neg A_1 \vee \neg A_2)$$

Semantic Tableaux – Example

$$\varphi = (A_1 \vee A_2) \wedge (A_1 \vee \neg A_2) \wedge (\neg A_1 \vee A_2) \wedge (\neg A_1 \vee \neg A_2)$$

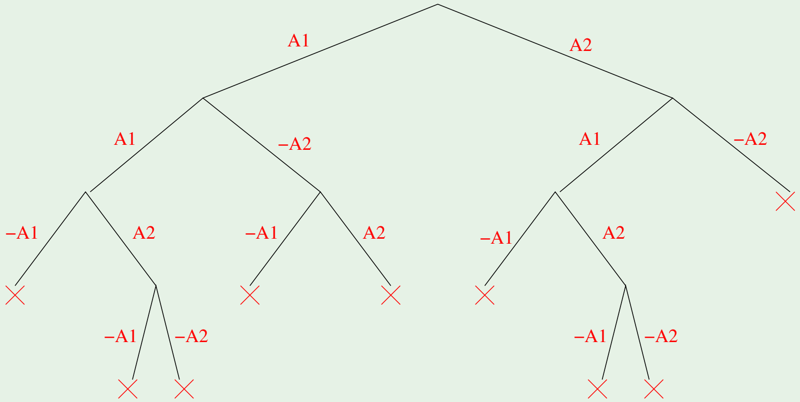


Tableau algorithm

function *Tableau*(Γ)
  if $A_i \in \Gamma$ **and** $\neg A_i \in \Gamma$ /* branch closed */
    then return *False*;
  if $(\varphi_1 \wedge \varphi_2) \in \Gamma$ /* $\wedge$-elimination */
    then return *Tableau*($\Gamma \cup \{\varphi_1, \varphi_2\} \backslash \{(\varphi_1 \wedge \varphi_2)\}$);
  if $(\neg\neg\varphi_1) \in \Gamma$ /* $\neg\neg$-elimination */
    then return *Tableau*($\Gamma \cup \{\varphi_1\} \backslash \{(\neg\neg\varphi_1)\}$);
  if $(\varphi_1 \vee \varphi_2) \in \Gamma$ /* $\vee$-elimination */
    then return *Tableau*($\Gamma \cup \{\varphi_1\} \backslash \{(\varphi_1 \vee \varphi_2)\}$) **or**
      Tableau($\Gamma \cup \{\varphi_2\} \backslash \{(\varphi_1 \vee \varphi_2)\}$);
  ...
  return *True*; /* branch expanded */

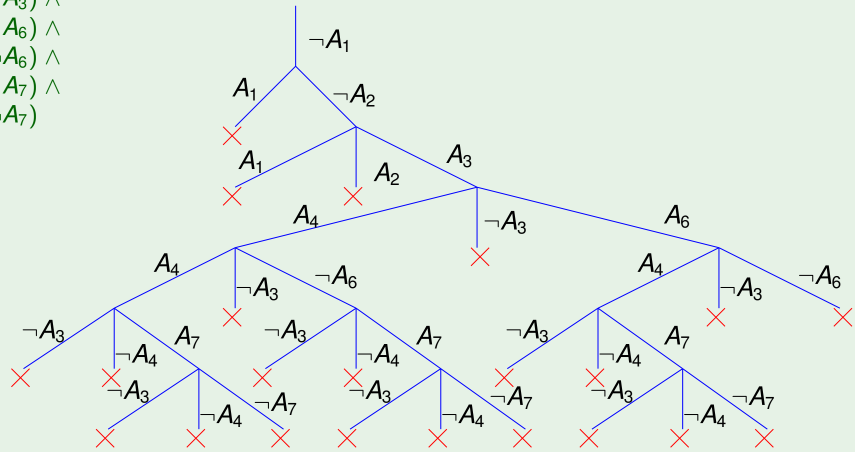
Semantic Tableaux: Example

$$
\begin{array}{rcrcrl}
 & & & & (\neg A_1) & \wedge \\
 & & (\ A_1 & \vee & \neg A_2) & \wedge \\
(\ A_1 & \vee & A_2 & \vee & A_3) & \wedge \\
(\ A_4 & \vee & \neg A_3 & \vee & A_6) & \wedge \\
(\ A_4 & \vee & \neg A_3 & \vee & \neg A_6) & \wedge \\
(\neg A_3 & \vee & \neg A_4 & \vee & A_7) & \wedge \\
(\neg A_3 & \vee & \neg A_4 & \vee & \neg A_7) &
\end{array}
$$

$\Longrightarrow$ unsat

Semantic Tableaux: Example

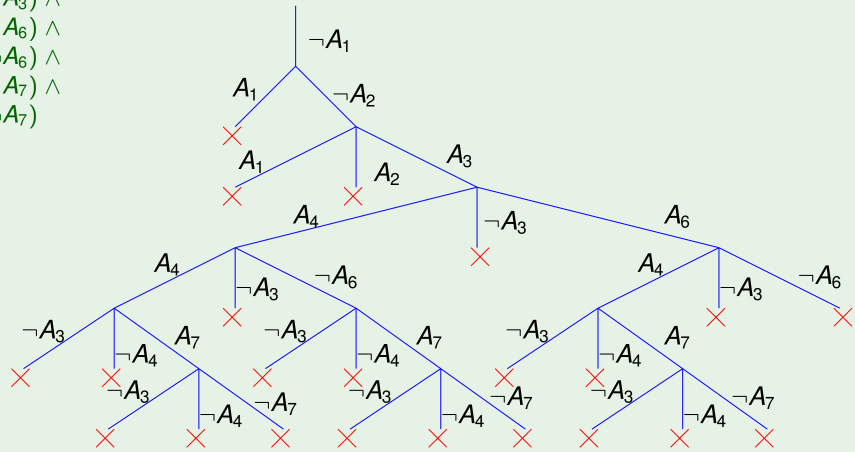
$$
\begin{array}{lllllr}
 & & & & (\neg A_1) & \wedge \\
 & & (\;A_1 & \vee & \neg A_2) & \wedge \\
(\;A_1 & \vee & A_2 & \vee & A_3) & \wedge \\
(\;A_4 & \vee & \neg A_3 & \vee & A_6) & \wedge \\
(\;A_4 & \vee & \neg A_3 & \vee & \neg A_6) & \wedge \\
(\neg A_3 & \vee & \neg A_4 & \vee & A_7) & \wedge \\
(\neg A_3 & \vee & \neg A_4 & \vee & \neg A_7) &
\end{array}
$$



$\Longrightarrow$ unsat

Semantic Tableaux: Example

$$
\begin{array}{lllllll}
 & & & & & (\neg A_1) & \wedge \\
 & & (A_1 & \vee & & \neg A_2) & \wedge \\
(A_1 & \vee & A_2 & \vee & & A_3) & \wedge \\
(A_4 & \vee & \neg A_3 & \vee & & A_6) & \wedge \\
(A_4 & \vee & \neg A_3 & \vee & & \neg A_6) & \wedge \\
(\neg A_3 & \vee & \neg A_4 & \vee & & A_7) & \wedge \\
(\neg A_3 & \vee & \neg A_4 & \vee & & \neg A_7) &
\end{array}
$$



$\Longrightarrow$ unsat

Semantic Tableaux – Summary

- Handles all propositional formulas (CNF not required).
- Branches on disjunctions
- Intuitive, modular, easy to extend
  $\Longrightarrow$ loved by logicians.
- Rather inefficient
  $\Longrightarrow$ avoided by computer scientists.
- Requires polynomial space

Outline

DPLL [10, 9]

- Davis-Putnam-Longeman-Loveland procedure (DPLL)
- Tries to build an assignment μ satisfying φ;
- At each step assigns a truth value to (all instances of) one atom.
- Performs deterministic choices first.

DPLL rules

$$\frac{\varphi_1 \wedge (l)}{\varphi_1[l|\top]}\ (Unit)$$

$$\frac{\varphi}{\varphi[l|\top]}\ (l\ Pure)$$

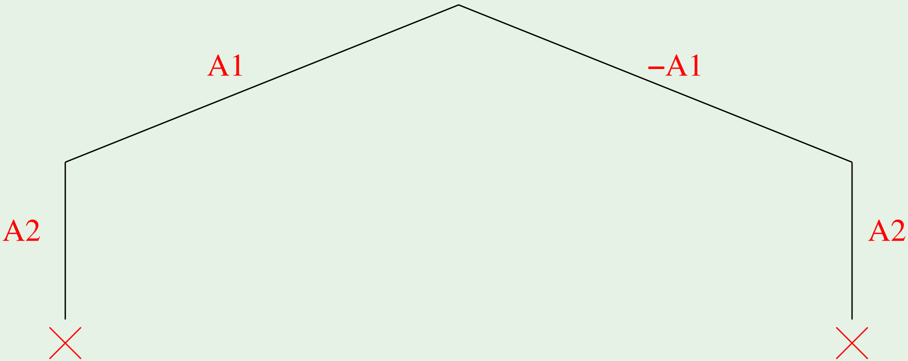
$$\frac{\varphi}{\varphi[l|\top] \quad \varphi[l|\bot]}\ (split)$$

(l is a pure literal in φ iff it occurs only positively).

- Split applied if and only if the others cannot be applied.
- Richer formalisms described in [40, 29, 30]

DPLL – example

$$\varphi = (A_1 \vee A_2) \wedge (A_1 \vee \neg A_2) \wedge (\neg A_1 \vee A_2) \wedge (\neg A_1 \vee \neg A_2)$$



function *DPLL*(φ, μ)
    if $\varphi = \top$ /* base */
        then return *True*;
    if $\varphi = \bot$ /* backtrack */
        then return *False*;
    if {a unit clause (l) occurs in φ}   /* unit */
        then return *DPLL*(*assign*(l, φ)$, \mu \wedge l$);
    if {a literal l occurs *pure* in φ}   /* pure */
        then return *DPLL*(*assign*(l, φ)$, \mu \wedge l$);
    l := *choose-literal*(φ);   /* split */
    return *DPLL*(*assign*(l, φ)$, \mu \wedge l$) **or**
            DPLL(*assign*($\neg l, \varphi$)$, \mu \wedge \neg l$);

- The pure-literal rule is nowadays obsolete.
- *choose-literal*(φ) picks only variables still occurring in the formula

The DPLL Algorithm

function *DPLL*(φ, μ)
    if $\varphi = \top$ /* base */
        then return *True*;
    if $\varphi = \bot$ /* backtrack */
        then return *False*;
    if {a unit clause (l) occurs in φ} /* unit */
        then return *DPLL*(*assign*(l, φ)$, \mu \wedge l$);

    l := *choose-literal*(φ); /* split */
    return *DPLL*(*assign*(l, φ)$, \mu \wedge l$) **or**
        DPLL(*assign*($\neg l, \varphi$)$, \mu \wedge \neg l$);

- The pure-literal rule is nowadays obsolete.
- *choose-literal*(φ) picks only variables still occurring in the formula

The DPLL Algorithm

```
function DPLL(φ, μ)
    if φ = ⊤                                        /* base      */
        then return True;
    if φ = ⊥                                        /* backtrack */
        then return False;
    if {a unit clause (l) occurs in φ}              /* unit      */
        then return DPLL(assign(l, φ), μ ∧ l);


    l := choose-literal(φ);                         /* split     */
    return  DPLL(assign(l, φ), μ ∧ l)  or
            DPLL(assign(¬l, φ), μ ∧ ¬l);
```

- The pure-literal rule is nowadays obsolete.
- *choose-literal*(φ) picks only variables still occurring in the formula

DPLL – example

DPLL (without pure-literal rule)

Here "choose-literal" selects variables in alphabetic order, selecting true first.

$$
\begin{array}{l}
(\neg C) \wedge \\
(\ B \vee \ A \vee \ C) \wedge \\
(\neg A \vee \ D) \wedge \\
(\neg E \vee \neg A \vee \ F) \wedge \\
(\neg E \vee \neg F \vee \neg A) \wedge \\
(\ G \vee \neg A \vee \ E) \wedge \\
(\ E \vee \neg G \vee \neg A) \wedge \\
(\ A \vee \ H \vee \ C) \wedge \\
(\neg H \vee \neg I \vee \ A) \wedge \\
(\ I \vee \ L \vee \ M) \wedge \\
(\neg L \vee \ C \vee \neg M) \wedge \\
(\ A \vee \neg L \vee \ M) \wedge \\
(\ L \vee \ N \vee \neg H) \wedge \\
(\ I \vee \ L \vee \neg N)
\end{array}
$$

Remark: "choose-literal" selects only variables which still occur in the formula, after simplification. E.g., in the leftmost branch, after assigning $\neg C$, A, D, it does not select B because the clause $(B \vee A \vee C)$ has been simplified into true, and as such is no more part of the formula, so that B does not occur in the formula anymore.

DPLL – example

DPLL (without pure-literal rule)

Here “choose-literal” selects variables in alphabetic order, selecting true first.

$$
\begin{array}{l}
(\neg C) \wedge \\
(\ B \vee \ A \vee \ C) \wedge \\
(\neg A \vee \ D) \wedge \\
(\neg E \vee \neg A \vee \ F) \wedge \\
(\neg E \vee \neg F \vee \neg A) \wedge \\
(\ G \vee \neg A \vee \ E) \wedge \\
(\ E \vee \neg G \vee \neg A) \wedge \\
(\ A \vee \ H \vee \ C) \wedge \\
(\neg H \vee \neg I \vee \ A) \wedge \\
(\ I \vee \ L \vee \ M) \wedge \\
(\neg L \vee \ C \vee \neg M) \wedge \\
(\ A \vee \neg L \vee \ M) \wedge \\
(\ L \vee \ N \vee \neg H) \wedge \\
(\ I \vee \ L \vee \neg N)
\end{array}
$$

$\neg C$

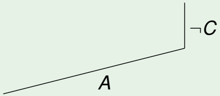
Remark: “choose-literal” selects only variables which still occur in the formula, after simplification. E.g., in the leftmost branch, after assigning $\neg C$, A, D, it does not select B because the clause $(B \vee A \vee C)$ has been simplified into true, and as such is no more part of the formula, so that B does not occur in the formula anymore.

DPLL – example

DPLL (without pure-literal rule)

Here "choose-literal" selects variables in alphabetic order, selecting true first.

$$
\begin{array}{l}
(\neg C) \wedge \\
(\;B \vee \;A \vee \;C) \wedge \\
(\neg A \vee \;D) \wedge \\
(\neg E \vee \neg A \vee \;F) \wedge \\
(\neg E \vee \neg F \vee \neg A) \wedge \\
(\;G \vee \neg A \vee \;E) \wedge \\
(\;E \vee \neg G \vee \neg A) \wedge \\
(\;A \vee \;H \vee \;C) \wedge \\
(\neg H \vee \neg I \vee \;A) \wedge \\
(\;I \vee \;L \vee \;M) \wedge \\
(\neg L \vee \;C \vee \neg M) \wedge \\
(\;A \vee \neg L \vee \;M) \wedge \\
(\;L \vee \;N \vee \neg H) \wedge \\
(\;I \vee \;L \vee \neg N)
\end{array}
$$



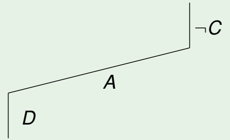
Remark: "choose-literal" selects only variables which still occur in the formula, after simplification. E.g., in the leftmost branch, after assigning $\neg C$, A, D, it does not select B because the clause $(B \vee A \vee C)$ has been simplified into true, and as such is no more part of the formula, so that B does not occur in the formula anymore.

DPLL – example

DPLL (without pure-literal rule)

Here "choose-literal" selects variables in alphabetic order, selecting true first.

$$
\begin{aligned}
&(\neg C) \wedge \\
&(B \vee A \vee C) \wedge \\
&(\neg A \vee D) \wedge \\
&(\neg E \vee \neg A \vee F) \wedge \\
&(\neg E \vee \neg F \vee \neg A) \wedge \\
&(G \vee \neg A \vee E) \wedge \\
&(E \vee \neg G \vee \neg A) \wedge \\
&(A \vee H \vee C) \wedge \\
&(\neg H \vee \neg I \vee A) \wedge \\
&(I \vee L \vee M) \wedge \\
&(\neg L \vee C \vee \neg M) \wedge \\
&(A \vee \neg L \vee M) \wedge \\
&(L \vee N \vee \neg H) \wedge \\
&(I \vee L \vee \neg N)
\end{aligned}
$$



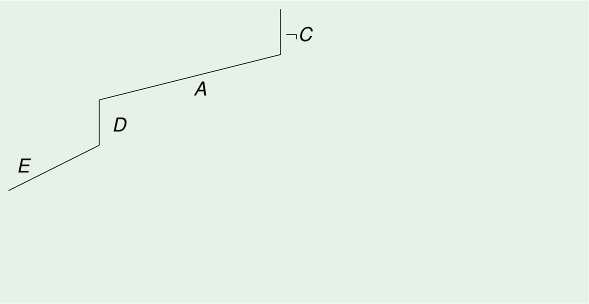
Remark: "choose-literal" selects only variables which still occur in the formula, after simplification. E.g., in the leftmost branch, after assigning $\neg C$, A, D, it does not select B because the clause $(B \vee A \vee C)$ has been simplified into true, and as such is no more part of the formula, so that B does not occur in the formula anymore.

DPLL – example

DPLL (without pure-literal rule)

Here "choose-literal" selects variables in alphabetic order, selecting true first.

$$
\begin{array}{l}
(\neg C) \wedge \\
(B \vee A \vee C) \wedge \\
(\neg A \vee D) \wedge \\
(\neg E \vee \neg A \vee F) \wedge \\
(\neg E \vee \neg F \vee \neg A) \wedge \\
(G \vee \neg A \vee E) \wedge \\
(E \vee \neg G \vee \neg A) \wedge \\
(A \vee H \vee C) \wedge \\
(\neg H \vee \neg I \vee A) \wedge \\
(I \vee L \vee M) \wedge \\
(\neg L \vee C \vee \neg M) \wedge \\
(A \vee \neg L \vee M) \wedge \\
(L \vee N \vee \neg H) \wedge \\
(I \vee L \vee \neg N)
\end{array}
$$



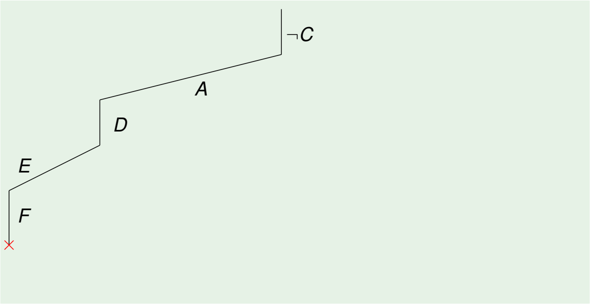
Remark: "choose-literal" selects only variables which still occur in the formula, after simplification. E.g., in the leftmost branch, after assigning $\neg C$, A, D, it does not select B because the clause $(B \vee A \vee C)$ has been simplified into true, and as such is no more part of the formula, so that B does not occur in the formula anymore.

DPLL – example

DPLL (without pure-literal rule)

Here "choose-literal" selects variables in alphabetic order, selecting true first.

$$
\begin{array}{l}
(\neg C) \wedge \\
(\ B \vee \ A \vee \ C) \wedge \\
(\neg A \vee \ D) \wedge \\
(\neg E \vee \neg A \vee \ F) \wedge \\
(\neg E \vee \neg F \vee \neg A) \wedge \\
(\ G \vee \neg A \vee \ E) \wedge \\
(\ E \vee \neg G \vee \neg A) \wedge \\
(\ A \vee \ H \vee \ C) \wedge \\
(\neg H \vee \neg I \vee \ A) \wedge \\
(\ I \vee \ L \vee \ M) \wedge \\
(\neg L \vee \ C \vee \neg M) \wedge \\
(\ A \vee \neg L \vee \ M) \wedge \\
(\ L \vee \ N \vee \neg H) \wedge \\
(\ I \vee \ L \vee \neg N)
\end{array}
$$



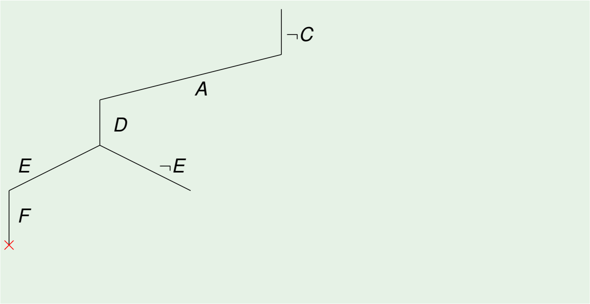
Remark: "choose-literal" selects only variables which still occur in the formula, after simplification. E.g., in the leftmost branch, after assigning $\neg C$, A, D, it does not select B because the clause $(B \vee A \vee C)$ has been simplified into true, and as such is no more part of the formula, so that B does not occur in the formula anymore.

DPLL – example

DPLL (without pure-literal rule)

Here "choose-literal" selects variables in alphabetic order, selecting true first.

$$
\begin{aligned}
&(\neg C) \land \\
&(B \lor A \lor C) \land \\
&(\neg A \lor D) \land \\
&(\neg E \lor \neg A \lor F) \land \\
&(\neg E \lor \neg F \lor \neg A) \land \\
&(G \lor \neg A \lor E) \land \\
&(E \lor \neg G \lor \neg A) \land \\
&(A \lor H \lor C) \land \\
&(\neg H \lor \neg I \lor A) \land \\
&(I \lor L \lor M) \land \\
&(\neg L \lor C \lor \neg M) \land \\
&(A \lor \neg L \lor M) \land \\
&(L \lor N \lor \neg H) \land \\
&(I \lor L \lor \neg N)
\end{aligned}
$$



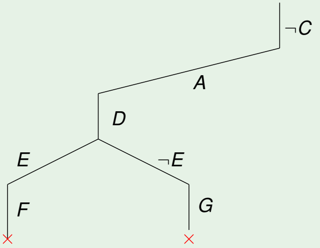
Remark: "choose-literal" selects only variables which still occur in the formula, after simplification. E.g., in the leftmost branch, after assigning $\neg C$, A, D, it does not select B because the clause $(B \lor A \lor C)$ has been simplified into true, and as such is no more part of the formula, so that B does not occur in the formula anymore.

DPLL – example

DPLL (without pure-literal rule)

Here "choose-literal" selects variables in alphabetic order, selecting true first.

$$
\begin{array}{l}
(\neg C) \wedge \\
(B \vee A \vee C) \wedge \\
(\neg A \vee D) \wedge \\
(\neg E \vee \neg A \vee F) \wedge \\
(\neg E \vee \neg F \vee \neg A) \wedge \\
(G \vee \neg A \vee E) \wedge \\
(E \vee \neg G \vee \neg A) \wedge \\
(A \vee H \vee C) \wedge \\
(\neg H \vee \neg I \vee A) \wedge \\
(I \vee L \vee M) \wedge \\
(\neg L \vee C \vee \neg M) \wedge \\
(A \vee \neg L \vee M) \wedge \\
(L \vee N \vee \neg H) \wedge \\
(I \vee L \vee \neg N)
\end{array}
$$



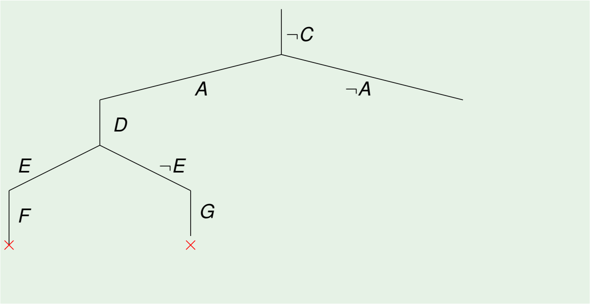
Remark: "choose-literal" selects only variables which still occur in the formula, after simplification. E.g., in the leftmost branch, after assigning $\neg C$, A, D, it does not select B because the clause $(B \vee A \vee C)$ has been simplified into true, and as such is no more part of the formula, so that B does not occur in the formula anymore.

DPLL – example

DPLL (without pure-literal rule)

Here “choose-literal” selects variables in alphabetic order, selecting true first.

$$
\begin{array}{l}
(\neg C) \wedge \\
(B \vee A \vee C) \wedge \\
(\neg A \vee D) \wedge \\
(\neg E \vee \neg A \vee F) \wedge \\
(\neg E \vee \neg F \vee \neg A) \wedge \\
(G \vee \neg A \vee E) \wedge \\
(E \vee \neg G \vee \neg A) \wedge \\
(A \vee H \vee C) \wedge \\
(\neg H \vee \neg I \vee A) \wedge \\
(I \vee L \vee M) \wedge \\
(\neg L \vee C \vee \neg M) \wedge \\
(A \vee \neg L \vee M) \wedge \\
(L \vee N \vee \neg H) \wedge \\
(I \vee L \vee \neg N)
\end{array}
$$



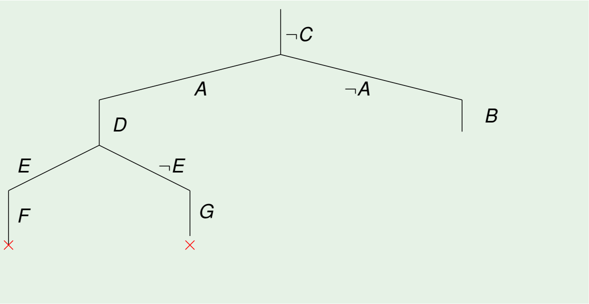
Remark: “choose-literal” selects only variables which still occur in the formula, after simplification. E.g., in the leftmost branch, after assigning $\neg C$, A, D, it does not select B because the clause $(B \vee A \vee C)$ has been simplified into true, and as such is no more part of the formula, so that B does not occur in the formula anymore.

DPLL – example

DPLL (without pure-literal rule)

Here “choose-literal” selects variables in alphabetic order, selecting true first.

$$
\begin{array}{l}
(\neg C) \wedge \\
(B \vee A \vee C) \wedge \\
(\neg A \vee D) \wedge \\
(\neg E \vee \neg A \vee F) \wedge \\
(\neg E \vee \neg F \vee \neg A) \wedge \\
(G \vee \neg A \vee E) \wedge \\
(E \vee \neg G \vee \neg A) \wedge \\
(A \vee H \vee C) \wedge \\
(\neg H \vee \neg I \vee A) \wedge \\
(I \vee L \vee M) \wedge \\
(\neg L \vee C \vee \neg M) \wedge \\
(A \vee \neg L \vee M) \wedge \\
(L \vee N \vee \neg H) \wedge \\
(I \vee L \vee \neg N)
\end{array}
$$



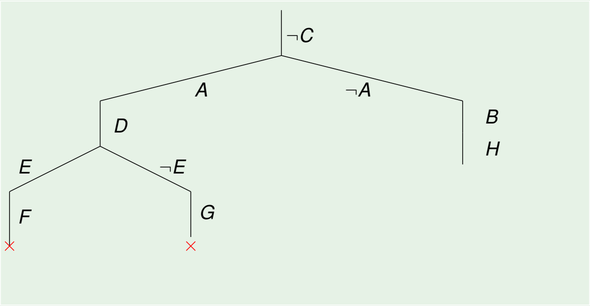
Remark: “choose-literal” selects only variables which still occur in the formula, after simplification. E.g., in the leftmost branch, after assigning $\neg C$, A, D, it does not select B because the clause $(B \vee A \vee C)$ has been simplified into true, and as such is no more part of the formula, so that B does not occur in the formula anymore.

DPLL – example

DPLL (without pure-literal rule)

Here “choose-literal” selects variables in alphabetic order, selecting true first.

$$
\begin{array}{l}
(\neg C) \wedge \\
(B \vee A \vee C) \wedge \\
(\neg A \vee D) \wedge \\
(\neg E \vee \neg A \vee F) \wedge \\
(\neg E \vee \neg F \vee \neg A) \wedge \\
(G \vee \neg A \vee E) \wedge \\
(E \vee \neg G \vee \neg A) \wedge \\
(A \vee H \vee C) \wedge \\
(\neg H \vee \neg I \vee A) \wedge \\
(I \vee L \vee M) \wedge \\
(\neg L \vee C \vee \neg M) \wedge \\
(A \vee \neg L \vee M) \wedge \\
(L \vee N \vee \neg H) \wedge \\
(I \vee L \vee \neg N)
\end{array}
$$



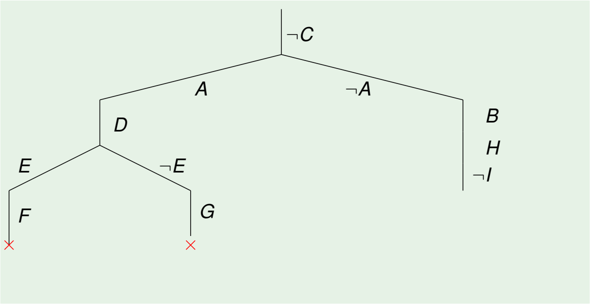
Remark: “choose-literal” selects only variables which still occur in the formula, after simplification. E.g., in the leftmost branch, after assigning $\neg C$, A, D, it does not select B because the clause $(B \vee A \vee C)$ has been simplified into true, and as such is no more part of the formula, so that B does not occur in the formula anymore.

DPLL – example

DPLL (without pure-literal rule)

Here "choose-literal" selects variables in alphabetic order, selecting true first.

$$
\begin{array}{l}
(\neg C) \wedge \\
(B \vee A \vee C) \wedge \\
(\neg A \vee D) \wedge \\
(\neg E \vee \neg A \vee F) \wedge \\
(\neg E \vee \neg F \vee \neg A) \wedge \\
(G \vee \neg A \vee E) \wedge \\
(E \vee \neg G \vee \neg A) \wedge \\
(A \vee H \vee C) \wedge \\
(\neg H \vee \neg I \vee A) \wedge \\
(I \vee L \vee M) \wedge \\
(\neg L \vee C \vee \neg M) \wedge \\
(A \vee \neg L \vee M) \wedge \\
(L \vee N \vee \neg H) \wedge \\
(I \vee L \vee \neg N)
\end{array}
$$



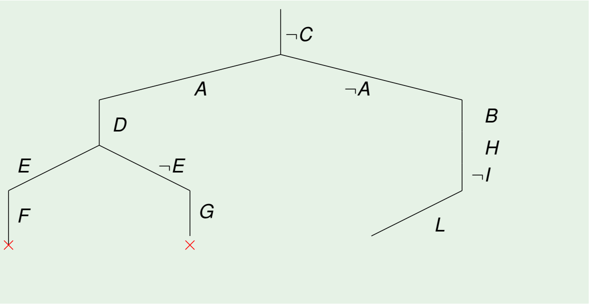
Remark: "choose-literal" selects only variables which still occur in the formula, after simplification. E.g., in the leftmost branch, after assigning $\neg C$, A, D, it does not select B because the clause $(B \vee A \vee C)$ has been simplified into true, and as such is no more part of the formula, so that B does not occur in the formula anymore.

DPLL – example

DPLL (without pure-literal rule)

Here “choose-literal” selects variables in alphabetic order, selecting true first.

$$
\begin{array}{l}
(\neg C) \wedge \\
(B \vee A \vee C) \wedge \\
(\neg A \vee D) \wedge \\
(\neg E \vee \neg A \vee F) \wedge \\
(\neg E \vee \neg F \vee \neg A) \wedge \\
(G \vee \neg A \vee E) \wedge \\
(E \vee \neg G \vee \neg A) \wedge \\
(A \vee H \vee C) \wedge \\
(\neg H \vee \neg I \vee A) \wedge \\
(I \vee L \vee M) \wedge \\
(\neg L \vee C \vee \neg M) \wedge \\
(A \vee \neg L \vee M) \wedge \\
(L \vee N \vee \neg H) \wedge \\
(I \vee L \vee \neg N)
\end{array}
$$



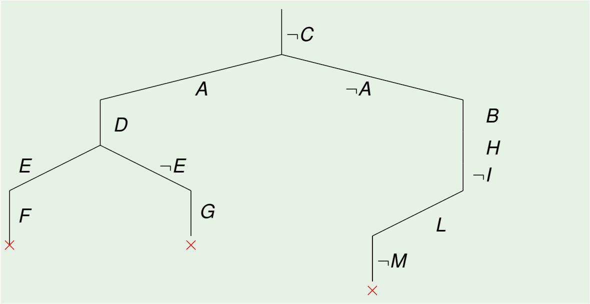
Remark: “choose-literal” selects only variables which still occur in the formula, after simplification. E.g., in the leftmost branch, after assigning $\neg C$, A, D, it does not select B because the clause $(B \vee A \vee C)$ has been simplified into true, and as such is no more part of the formula, so that B does not occur in the formula anymore.

DPLL – example

DPLL (without pure-literal rule)

Here "choose-literal" selects variables in alphabetic order, selecting true first.

$$
\begin{aligned}
&(\neg C) \wedge \\
&(B \vee A \vee C) \wedge \\
&(\neg A \vee D) \wedge \\
&(\neg E \vee \neg A \vee F) \wedge \\
&(\neg E \vee \neg F \vee \neg A) \wedge \\
&(G \vee \neg A \vee E) \wedge \\
&(E \vee \neg G \vee \neg A) \wedge \\
&(A \vee H \vee C) \wedge \\
&(\neg H \vee \neg I \vee A) \wedge \\
&(I \vee L \vee M) \wedge \\
&(\neg L \vee C \vee \neg M) \wedge \\
&(A \vee \neg L \vee M) \wedge \\
&(L \vee N \vee \neg H) \wedge \\
&(I \vee L \vee \neg N)
\end{aligned}
$$



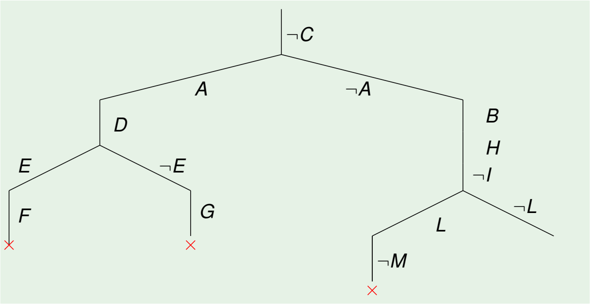
Remark: "choose-literal" selects only variables which still occur in the formula, after simplification. E.g., in the leftmost branch, after assigning $\neg C$, A, D, it does not select B because the clause $(B \vee A \vee C)$ has been simplified into true, and as such is no more part of the formula, so that B does not occur in the formula anymore.

DPLL – example

DPLL (without pure-literal rule)

Here "choose-literal" selects variables in alphabetic order, selecting true first.

$$
\begin{array}{l}
(\neg C) \wedge \\
(B \vee A \vee C) \wedge \\
(\neg A \vee D) \wedge \\
(\neg E \vee \neg A \vee F) \wedge \\
(\neg E \vee \neg F \vee \neg A) \wedge \\
(G \vee \neg A \vee E) \wedge \\
(E \vee \neg G \vee \neg A) \wedge \\
(A \vee H \vee C) \wedge \\
(\neg H \vee \neg I \vee A) \wedge \\
(I \vee L \vee M) \wedge \\
(\neg L \vee C \vee \neg M) \wedge \\
(A \vee \neg L \vee M) \wedge \\
(L \vee N \vee \neg H) \wedge \\
(I \vee L \vee \neg N)
\end{array}
$$



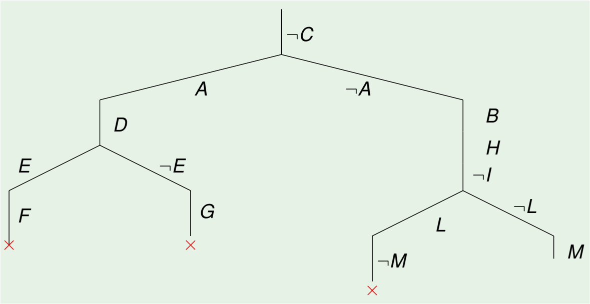
Remark: "choose-literal" selects only variables which still occur in the formula, after simplification. E.g., in the leftmost branch, after assigning $\neg C$, A, D, it does not select B because the clause $(B \vee A \vee C)$ has been simplified into true, and as such is no more part of the formula, so that B does not occur in the formula anymore.

DPLL – example

DPLL (without pure-literal rule)

Here “choose-literal” selects variables in alphabetic order, selecting true first.

$$
\begin{array}{l}
(\neg C) \wedge \\
(B \vee A \vee C) \wedge \\
(\neg A \vee D) \wedge \\
(\neg E \vee \neg A \vee F) \wedge \\
(\neg E \vee \neg F \vee \neg A) \wedge \\
(G \vee \neg A \vee E) \wedge \\
(E \vee \neg G \vee \neg A) \wedge \\
(A \vee H \vee C) \wedge \\
(\neg H \vee \neg I \vee A) \wedge \\
(I \vee L \vee M) \wedge \\
(\neg L \vee C \vee \neg M) \wedge \\
(A \vee \neg L \vee M) \wedge \\
(L \vee N \vee \neg H) \wedge \\
(I \vee L \vee \neg N)
\end{array}
$$



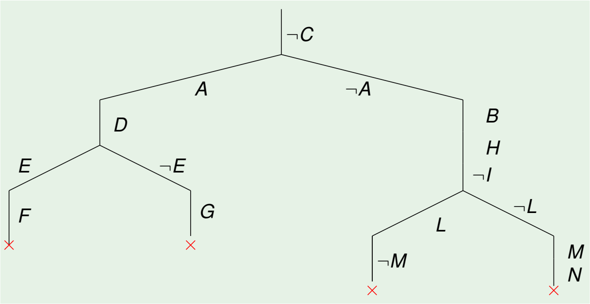
Remark: “choose-literal” selects only variables which still occur in the formula, after simplification. E.g., in the leftmost branch, after assigning $\neg C$, A, D, it does not select B because the clause $(B \vee A \vee C)$ has been simplified into true, and as such is no more part of the formula, so that B does not occur in the formula anymore.

DPLL – example

DPLL (without pure-literal rule)

Here "choose-literal" selects variables in alphabetic order, selecting true first.

$$
\begin{array}{l}
(\neg C) \wedge \\
(B \vee A \vee C) \wedge \\
(\neg A \vee D) \wedge \\
(\neg E \vee \neg A \vee F) \wedge \\
(\neg E \vee \neg F \vee \neg A) \wedge \\
(G \vee \neg A \vee E) \wedge \\
(E \vee \neg G \vee \neg A) \wedge \\
(A \vee H \vee C) \wedge \\
(\neg H \vee \neg I \vee A) \wedge \\
(I \vee L \vee M) \wedge \\
(\neg L \vee C \vee \neg M) \wedge \\
(A \vee \neg L \vee M) \wedge \\
(L \vee N \vee \neg H) \wedge \\
(I \vee L \vee \neg N)
\end{array}
$$



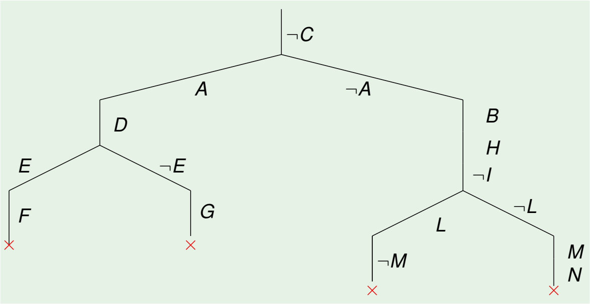
Remark: "choose-literal" selects only variables which still occur in the formula, after simplification. E.g., in the leftmost branch, after assigning $\neg C$, A, D, it does not select B because the clause $(B \vee A \vee C)$ has been simplified into true, and as such is no more part of the formula, so that B does not occur in the formula anymore.

DPLL – example

DPLL (without pure-literal rule)

Here “choose-literal” selects variables in alphabetic order, selecting true first.

$$
\begin{array}{l}
(\neg C) \wedge \\
(B \vee A \vee C) \wedge \\
(\neg A \vee D) \wedge \\
(\neg E \vee \neg A \vee F) \wedge \\
(\neg E \vee \neg F \vee \neg A) \wedge \\
(G \vee \neg A \vee E) \wedge \\
(E \vee \neg G \vee \neg A) \wedge \\
(A \vee H \vee C) \wedge \\
(\neg H \vee \neg I \vee A) \wedge \\
(I \vee L \vee M) \wedge \\
(\neg L \vee C \vee \neg M) \wedge \\
(A \vee \neg L \vee M) \wedge \\
(L \vee N \vee \neg H) \wedge \\
(I \vee L \vee \neg N)
\end{array}
$$



$\Longrightarrow$ UNSAT

Remark: “choose-literal” selects only variables which still occur in the formula, after simplification. E.g., in the leftmost branch, after assigning $\neg C$, A, D, it does not select B because the clause $(B \vee A \vee C)$ has been simplified into true, and as such is no more part of the formula, so that B does not occur in the formula anymore.

DPLL – summary

- Handles CNF formulas (non-CNF variant known [1, 15]).
- Branches on truth values
  $\Longrightarrow$ all instances of an atom assigned simultaneously
- Postpones branching as much as possible.
- Mostly ignored by logicians.
- (The grandfather of) the most efficient SAT algorithms
  $\Longrightarrow$ loved by computer scientists.
- Requires polynomial space
- Choose_literal() critical!
- Many very efficient implementations [42, 38, 2, 28].

Outline

Ordered Binary Decision Diagrams (OBDDs) [8]]

Canonical representation of Boolean formulas

- "If-then-else" binary direct acyclic graphs (DAGs) with one root and two leaves: 1, 0 (or $\top, \bot$; or T, F)
- Variable ordering $A_1, A_2, \ldots, A_n$ imposed a priori.
- Paths leading to 1 represent models
  Paths leading to 0 represent counter-models

Note

Some authors call them Reduced Ordered Binary Decision Diagrams (ROBDDs)

Ordered Binary Decision Diagrams (OBDDs) [8]]

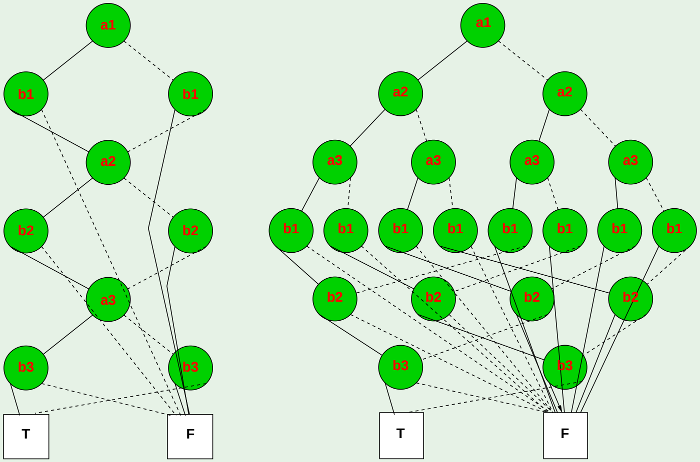
Canonical representation of Boolean formulas

- "If-then-else" binary direct acyclic graphs (DAGs) with one root and two leaves: 1, 0 (or $\top, \bot$; or T, F)
- Variable ordering $A_1, A_2, ..., A_n$ imposed a priori.
- Paths leading to 1 represent models
  Paths leading to 0 represent counter-models

Note

Some authors call them Reduced Ordered Binary Decision Diagrams (ROBDDs)

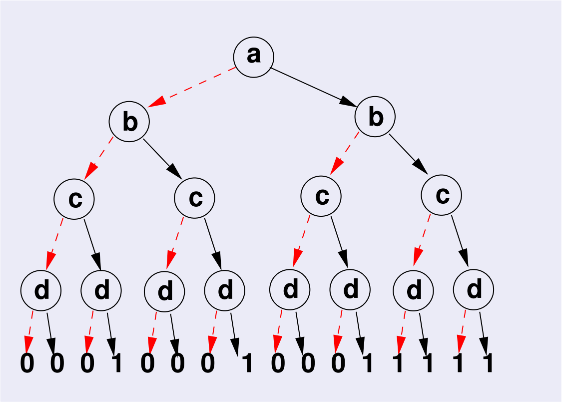
OBDD - Examples



OBDDs of $(a_1 \leftrightarrow b_1) \wedge (a_2 \leftrightarrow b_2) \wedge (a_3 \leftrightarrow b_3)$ with different variable orderings

Ordered Decision Trees

- Ordered Decision Tree:
  from root to leaves, variables are encountered always in the same order
- Example: Ordered Decision tree for $\varphi \stackrel{\text{def}}{=} (a \wedge b) \vee (c \wedge d)$



From Ordered Decision Trees to OBDD’s: reductions

- Recursive applications of the following reductions:
  - share subnodes: point to the same occurrence of a subtree (via hash consing)
  - remove redundancies: nodes with same left and right children can be eliminated: “if A then B else B” $\Longrightarrow$ “B”

From Ordered Decision Trees to OBDD's: reductions

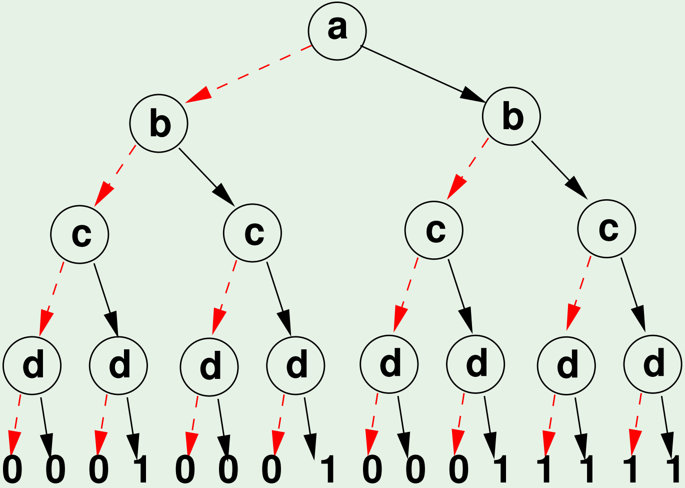
- Recursive applications of the following reductions:
  - share subnodes: point to the same occurrence of a subtree (via hash consing)
  - remove redundancies: nodes with same left and right children can be eliminated: "if A then B else B" $\Longrightarrow$ "B"

From Ordered Decision Trees to OBDD's: reductions

- Recursive applications of the following reductions:
  - share subnodes: point to the same occurrence of a subtree (via hash consing)
  - remove redundancies: nodes with same left and right children can be eliminated: "if A then B else B" $\Longrightarrow$ "B"

Reduction: example

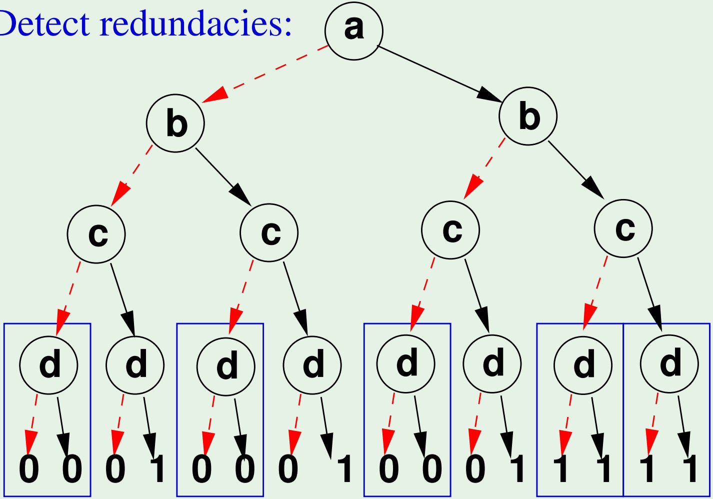
$\varphi \stackrel{\text{def}}{=} (a \wedge b) \vee (c \wedge d)$



Reduction: example

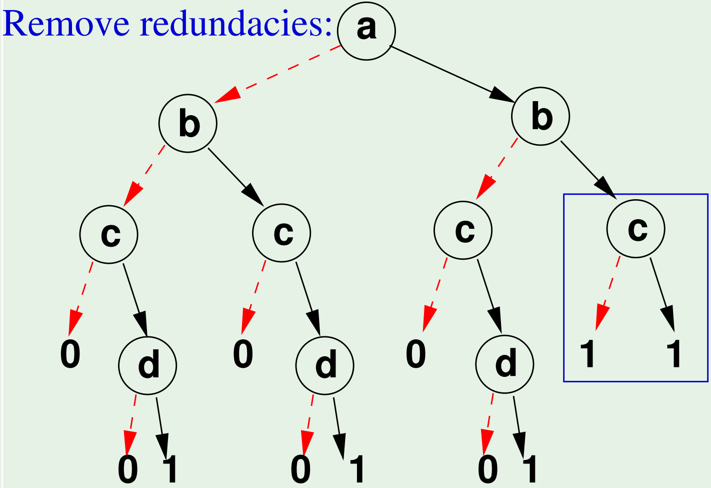
$\varphi \stackrel{\text{def}}{=} (a \wedge b) \vee (c \wedge d)$

Detect redundacies:



Reduction: example

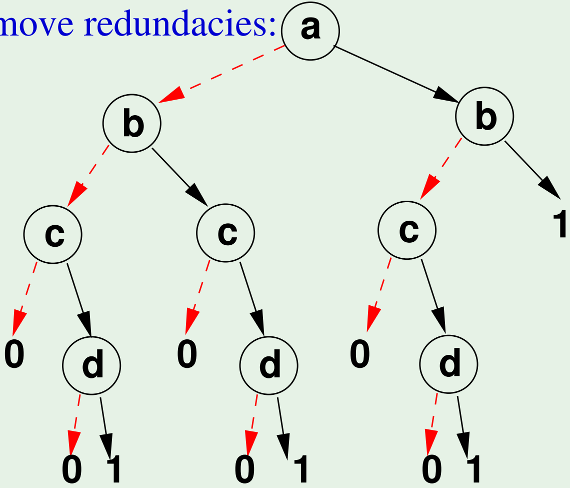
$\varphi \stackrel{\text{def}}{=} (a \land b) \lor (c \land d)$



Reduction: example

$\varphi \stackrel{\text{def}}{=} (a \wedge b) \vee (c \wedge d)$

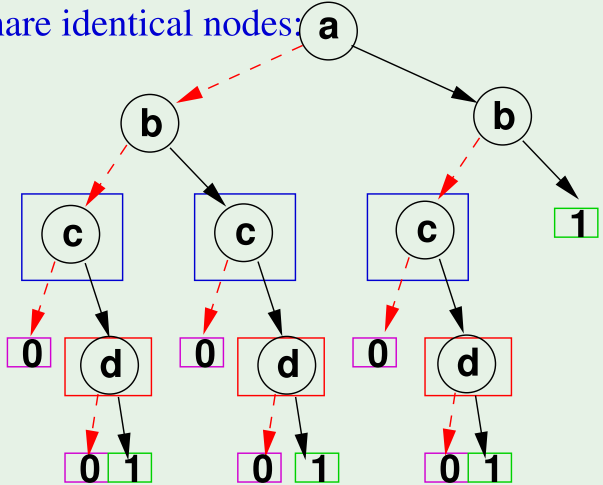
Remove redundacies:



Reduction: example

$\varphi \stackrel{\text{def}}{=} (a \wedge b) \vee (c \wedge d)$

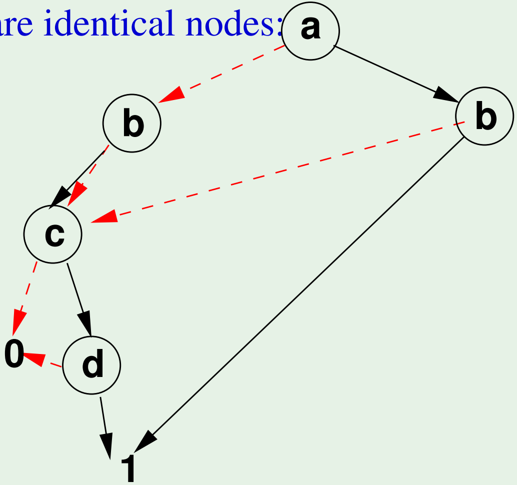
Share identical nodes:



Reduction: example

$\varphi \stackrel{\text{def}}{=} (a \wedge b) \vee (c \wedge d)$

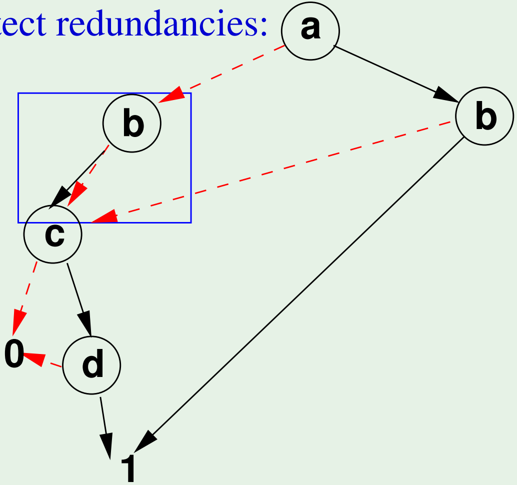
Share identical nodes:



Reduction: example

$\varphi \stackrel{\text{def}}{=} (a \land b) \lor (c \land d)$

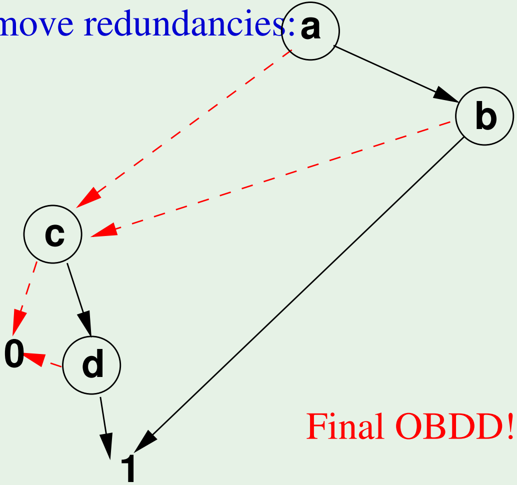
Detect redundancies:



Reduction: example

$\varphi \stackrel{\text{def}}{=} (a \wedge b) \vee (c \wedge d)$

Remove redundancies:



Final OBDD!

If-Then-Else Operators: "*ite*(...)"

If-Then-Else Operators: "*ite*(...)"

- $ite(\phi, \varphi^\top, \varphi^\bot)$: "If ϕ Then $\varphi^\top$ Else $\varphi^\bot$"
- $ite(\phi, \varphi^\top, \varphi^\bot) \stackrel{\text{def}}{=} ((\neg\phi \vee \varphi^\top) \wedge (\phi \vee \varphi^\bot)) \Longleftrightarrow ((\phi \wedge \varphi^\top) \vee (\neg\phi \wedge \varphi^\bot))$
- properties:

$$
\begin{aligned}
ite(\neg\phi, \varphi^\top, \varphi^\bot) &= ite(\phi, \varphi^\bot, \varphi^\top)\\
\neg ite(\phi, \varphi^\top, \varphi^\bot) &= ite(\phi, \neg\varphi^\top, \neg\varphi^\bot)\\
ite(\phi, \varphi_1^\top, \varphi_1^\bot) \; op \; ite(\phi, \varphi_2^\top, \varphi_2^\bot) &= ite(\phi, (\varphi_1^\top \; op \; \varphi_2^\top), (\varphi_1^\bot \; op \; \varphi_2^\bot))\\
ite(\phi_1, \varphi_1^\top, \varphi_1^\bot) \; op \; ite(\phi_2, \varphi_2^\top, \varphi_2^\bot) &= ite(\phi_1, (\varphi_1^\top \; op \; ite(\phi_2, \varphi_2^\top, \varphi_2^\bot)),\\
&\qquad\quad (\varphi_1^\bot \; op \; ite(\phi_2, \varphi_2^\top, \varphi_2^\bot)))\\
&= ite(\phi_2, (ite(\phi_1, \varphi_1^\top, \varphi_1^\bot) \; op \; \varphi_2^\top),\\
&\qquad\quad (ite(\phi_1, \varphi_1^\top, \varphi_1^\bot) \; op \; \varphi_2^\bot))
\end{aligned}
\qquad op \in \{\wedge, \vee, \rightarrow, \leftarrow, \leftrightarrow, \oplus\}
$$

If-Then-Else Operators: "*ite*(...)"

If-Then-Else Operators: "*ite*(...)"

- $ite(\phi, \varphi^{\top}, \varphi^{\bot})$: "If ϕ Then $\varphi^{\top}$ Else $\varphi^{\bot}$"
- $ite(\phi, \varphi^{\top}, \varphi^{\bot}) \stackrel{\text{def}}{=} ((\neg\phi \vee \varphi^{\top}) \wedge (\phi \vee \varphi^{\bot})) \Longleftrightarrow ((\phi \wedge \varphi^{\top}) \vee (\neg\phi \wedge \varphi^{\bot}))$
- properties:

$$
\begin{aligned}
ite(\neg\phi, \varphi^{\top}, \varphi^{\bot}) &= ite(\phi, \varphi^{\bot}, \varphi^{\top}) \\
\neg ite(\phi, \varphi^{\top}, \varphi^{\bot}) &= ite(\phi, \neg\varphi^{\top}, \neg\varphi^{\bot}) \\
ite(\phi, \varphi_1^{\top}, \varphi_1^{\bot})\ op\ ite(\phi, \varphi_2^{\top}, \varphi_2^{\bot}) &= ite(\phi, (\varphi_1^{\top}\ op\ \varphi_2^{\top}), (\varphi_1^{\bot}\ op\ \varphi_2^{\bot})) \\
ite(\phi_1, \varphi_1^{\top}, \varphi_1^{\bot})\ op\ ite(\phi_2, \varphi_2^{\top}, \varphi_2^{\bot}) &= ite(\phi_1, (\varphi_1^{\top}\ op\ ite(\phi_2, \varphi_2^{\top}, \varphi_2^{\bot})), \qquad op \in \{\wedge, \vee, \rightarrow, \leftarrow, \leftrightarrow, \oplus\} \\
&\qquad\quad (\varphi_1^{\bot}\ op\ ite(\phi_2, \varphi_2^{\top}, \varphi_2^{\bot}))) \\
&= ite(\phi_2, (ite(\phi_1, \varphi_1^{\top}, \varphi_1^{\bot})\ op\ \varphi_2^{\top}), \\
&\qquad\quad (ite(\phi_1, \varphi_1^{\top}, \varphi_1^{\bot})\ op\ \varphi_2^{\bot}))
\end{aligned}
$$

If-Then-Else Operators: "*ite*(...)"

If-Then-Else Operators: "*ite*(...)"

- $ite(\phi, \varphi^\top, \varphi^\bot)$: "If ϕ Then $\varphi^\top$ Else $\varphi^\bot$"
- $ite(\phi, \varphi^\top, \varphi^\bot) \stackrel{\text{def}}{=} ((\neg\phi \vee \varphi^\top) \wedge (\phi \vee \varphi^\bot)) \Longleftrightarrow ((\phi \wedge \varphi^\top) \vee (\neg\phi \wedge \varphi^\bot))$
- properties:

$$
\begin{aligned}
ite(\neg\phi, \varphi^\top, \varphi^\bot) &= ite(\phi, \varphi^\bot, \varphi^\top) \\
\neg ite(\phi, \varphi^\top, \varphi^\bot) &= ite(\phi, \neg\varphi^\top, \neg\varphi^\bot) \\
ite(\phi, \varphi_1^\top, \varphi_1^\bot) \; op \; ite(\phi, \varphi_2^\top, \varphi_2^\bot) &= ite(\phi, (\varphi_1^\top \; op \; \varphi_2^\top), (\varphi_1^\bot \; op \; \varphi_2^\bot)) \\
ite(\phi_1, \varphi_1^\top, \varphi_1^\bot) \; op \; ite(\phi_2, \varphi_2^\top, \varphi_2^\bot) &= ite(\phi_1, (\varphi_1^\top \; op \; ite(\phi_2, \varphi_2^\top, \varphi_2^\bot)), \\
&\qquad\quad (\varphi_1^\bot \; op \; ite(\phi_2, \varphi_2^\top, \varphi_2^\bot))) \\
&= ite(\phi_2, (ite(\phi_1, \varphi_1^\top, \varphi_1^\bot) \; op \; \varphi_2^\top), \\
&\qquad\quad (ite(\phi_1, \varphi_1^\top, \varphi_1^\bot) \; op \; \varphi_2^\bot))
\end{aligned}
$$

$op \in \{\wedge, \vee, \rightarrow, \leftarrow, \leftrightarrow, \oplus\}$

If-Then-Else Operators: "*ite*(...)"

If-Then-Else Operators: "*ite*(...)"

- $ite(\phi, \varphi^\top, \varphi^\bot)$: "If ϕ Then $\varphi^\top$ Else $\varphi^\bot$"
- $ite(\phi, \varphi^\top, \varphi^\bot) \stackrel{\text{def}}{=} ((\neg\phi \vee \varphi^\top) \wedge (\phi \vee \varphi^\bot)) \iff ((\phi \wedge \varphi^\top) \vee (\neg\phi \wedge \varphi^\bot))$
- properties:

$$
\begin{aligned}
ite(\neg\phi, \varphi^\top, \varphi^\bot) &= ite(\phi, \varphi^\bot, \varphi^\top) \\
\neg ite(\phi, \varphi^\top, \varphi^\bot) &= ite(\phi, \neg\varphi^\top, \neg\varphi^\bot) \\
ite(\phi, \varphi_1^\top, \varphi_1^\bot)\ op\ ite(\phi, \varphi_2^\top, \varphi_2^\bot) &= ite(\phi, (\varphi_1^\top\ op\ \varphi_2^\top), (\varphi_1^\bot\ op\ \varphi_2^\bot)) \\
ite(\phi_1, \varphi_1^\top, \varphi_1^\bot)\ op\ ite(\phi_2, \varphi_2^\top, \varphi_2^\bot) &= ite(\phi_1, (\varphi_1^\top\ op\ ite(\phi_2, \varphi_2^\top, \varphi_2^\bot)), \\
&\qquad\qquad (\varphi_1^\bot\ op\ ite(\phi_2, \varphi_2^\top, \varphi_2^\bot))) \\
&= ite(\phi_2, (ite(\phi_1, \varphi_1^\top, \varphi_1^\bot)\ op\ \varphi_2^\top), \\
&\qquad\qquad (ite(\phi_1, \varphi_1^\top, \varphi_1^\bot)\ op\ \varphi_2^\bot))
\end{aligned}
\qquad op \in \{\wedge, \vee, \rightarrow, \leftarrow, \leftrightarrow, \oplus\}
$$

If-Then-Else Operators: "*ite*(...)"

If-Then-Else Operators: "*ite*(...)"

- $ite(\phi, \varphi^\top, \varphi^\bot)$: "If ϕ Then $\varphi^\top$ Else $\varphi^\bot$"
- $ite(\phi, \varphi^\top, \varphi^\bot) \stackrel{\text{def}}{=} ((\neg\phi \vee \varphi^\top) \wedge (\phi \vee \varphi^\bot)) \Longleftrightarrow ((\phi \wedge \varphi^\top) \vee (\neg\phi \wedge \varphi^\bot))$
- properties:

$$
\begin{aligned}
ite(\neg\phi, \varphi^\top, \varphi^\bot) &= ite(\phi, \varphi^\bot, \varphi^\top)\\
\neg ite(\phi, \varphi^\top, \varphi^\bot) &= ite(\phi, \neg\varphi^\top, \neg\varphi^\bot)\\
ite(\phi, \varphi_1^\top, \varphi_1^\bot)\ op\ ite(\phi, \varphi_2^\top, \varphi_2^\bot) &= ite(\phi, (\varphi_1^\top\ op\ \varphi_2^\top), (\varphi_1^\bot\ op\ \varphi_2^\bot))\\
ite(\phi_1, \varphi_1^\top, \varphi_1^\bot)\ op\ ite(\phi_2, \varphi_2^\top, \varphi_2^\bot) &= ite(\phi_1, (\varphi_1^\top\ op\ ite(\phi_2, \varphi_2^\top, \varphi_2^\bot)),\\
&\qquad\qquad (\varphi_1^\bot\ op\ ite(\phi_2, \varphi_2^\top, \varphi_2^\bot)))\\
&= ite(\phi_2, (ite(\phi_1, \varphi_1^\top, \varphi_1^\bot)\, op\ \varphi_2^\top),\\
&\qquad\qquad (ite(\phi_1, \varphi_1^\top, \varphi_1^\bot)\, op\ \varphi_2^\bot))
\end{aligned}
\qquad op \in \{\wedge, \vee, \rightarrow, \leftarrow, \leftrightarrow, \oplus\}
$$

If-Then-Else Operators: “*ite*(...)”

If-Then-Else Operators: “*ite*(...)”

- $ite(\phi, \varphi^{\top}, \varphi^{\bot})$: “If ϕ Then $\varphi^{\top}$ Else $\varphi^{\bot}$”
- $ite(\phi, \varphi^{\top}, \varphi^{\bot}) \stackrel{\text{def}}{=} ((\neg\phi \vee \varphi^{\top}) \wedge (\phi \vee \varphi^{\bot})) \Longleftrightarrow ((\phi \wedge \varphi^{\top}) \vee (\neg\phi \wedge \varphi^{\bot}))$
- properties:

$$
\begin{aligned}
ite(\neg\phi, \varphi^{\top}, \varphi^{\bot}) &= ite(\phi, \varphi^{\bot}, \varphi^{\top}) \\
\neg ite(\phi, \varphi^{\top}, \varphi^{\bot}) &= ite(\phi, \neg\varphi^{\top}, \neg\varphi^{\bot}) \\
ite(\phi, \varphi_1^{\top}, \varphi_1^{\bot})\ op\ ite(\phi, \varphi_2^{\top}, \varphi_2^{\bot}) &= ite(\phi, (\varphi_1^{\top}\ op\ \varphi_2^{\top}), (\varphi_1^{\bot}\ op\ \varphi_2^{\bot})) \\
ite(\phi_1, \varphi_1^{\top}, \varphi_1^{\bot})\ op\ ite(\phi_2, \varphi_2^{\top}, \varphi_2^{\bot}) &= ite(\phi_1, (\varphi_1^{\top}\ op\ ite(\phi_2, \varphi_2^{\top}, \varphi_2^{\bot})), \\
&\qquad (\varphi_1^{\bot}\ op\ ite(\phi_2, \varphi_2^{\top}, \varphi_2^{\bot}))) \\
&= ite(\phi_2, (ite(\phi_1, \varphi_1^{\top}, \varphi_1^{\bot})\ op\ \varphi_2^{\top}), \\
&\qquad (ite(\phi_1, \varphi_1^{\top}, \varphi_1^{\bot})\ op\ \varphi_2^{\bot}))
\end{aligned}
$$

$op \in \{\wedge, \vee, \rightarrow, \leftarrow, \leftrightarrow, \oplus\}$

If-Then-Else Operators: “*ite*(...)”

If-Then-Else Operators: “*ite*(...)”

- $ite(\phi, \varphi^{\top}, \varphi^{\bot})$: “If ϕ Then $\varphi^{\top}$ Else $\varphi^{\bot}$”
- $ite(\phi, \varphi^{\top}, \varphi^{\bot}) \stackrel{\text{def}}{=} ((\neg\phi \vee \varphi^{\top}) \wedge (\phi \vee \varphi^{\bot})) \Longleftrightarrow ((\phi \wedge \varphi^{\top}) \vee (\neg\phi \wedge \varphi^{\bot}))$
- properties:

$$
\begin{aligned}
ite(\neg\phi, \varphi^{\top}, \varphi^{\bot}) &= ite(\phi, \varphi^{\bot}, \varphi^{\top}) \\
\neg ite(\phi, \varphi^{\top}, \varphi^{\bot}) &= ite(\phi, \neg\varphi^{\top}, \neg\varphi^{\bot}) \\
ite(\phi, \varphi_1^{\top}, \varphi_1^{\bot}) \ op\ ite(\phi, \varphi_2^{\top}, \varphi_2^{\bot}) &= ite(\phi, (\varphi_1^{\top}\ op\ \varphi_2^{\top}), (\varphi_1^{\bot}\ op\ \varphi_2^{\bot})) \\
ite(\phi_1, \varphi_1^{\top}, \varphi_1^{\bot}) \ op\ ite(\phi_2, \varphi_2^{\top}, \varphi_2^{\bot}) &= ite(\phi_1, (\varphi_1^{\top}\ op\ ite(\phi_2, \varphi_2^{\top}, \varphi_2^{\bot})), \qquad op \in \{\wedge, \vee, \rightarrow, \leftarrow, \leftrightarrow, \oplus\} \\
&\qquad (\varphi_1^{\bot}\ op\ ite(\phi_2, \varphi_2^{\top}, \varphi_2^{\bot}))) \\
&= ite(\phi_2, (ite(\phi_1, \varphi_1^{\top}, \varphi_1^{\bot})\ op\ \varphi_2^{\top}), \\
&\qquad (ite(\phi_1, \varphi_1^{\top}, \varphi_1^{\bot})\ op\ \varphi_2^{\bot}))
\end{aligned}
$$

If-Then-Else Operators: "*ite*(...)"

- $ite(\phi, \varphi^\top, \varphi^\bot)$: "If ϕ Then $\varphi^\top$ Else $\varphi^\bot$"
- $ite(\phi, \varphi^\top, \varphi^\bot) \stackrel{\text{def}}{=} ((\neg\phi \vee \varphi^\top) \wedge (\phi \vee \varphi^\bot)) \iff ((\phi \wedge \varphi^\top) \vee (\neg\phi \wedge \varphi^\bot))$
- properties:

$$\begin{aligned}
ite(\neg\phi, \varphi^\top, \varphi^\bot) &= ite(\phi, \varphi^\bot, \varphi^\top)\\
\neg ite(\phi, \varphi^\top, \varphi^\bot) &= ite(\phi, \neg\varphi^\top, \neg\varphi^\bot)\\
ite(\phi, \varphi_1^\top, \varphi_1^\bot)\ op\ ite(\phi, \varphi_2^\top, \varphi_2^\bot) &= ite(\phi, (\varphi_1^\top\ op\ \varphi_2^\top), (\varphi_1^\bot\ op\ \varphi_2^\bot))\\
ite(\phi_1, \varphi_1^\top, \varphi_1^\bot)\ op\ ite(\phi_2, \varphi_2^\top, \varphi_2^\bot) &= ite(\phi_1, (\varphi_1^\top\ op\ ite(\phi_2, \varphi_2^\top, \varphi_2^\bot)), \qquad op \in \{\wedge, \vee, \rightarrow, \leftarrow, \leftrightarrow, \oplus\}\\
&\qquad (\varphi_1^\bot\ op\ ite(\phi_2, \varphi_2^\top, \varphi_2^\bot)))\\
&= ite(\phi_2, (ite(\phi_1, \varphi_1^\top, \varphi_1^\bot)op\ \varphi_2^\top),\\
&\qquad (ite(\phi_1, \varphi_1^\top, \varphi_1^\bot)op\ \varphi_2^\bot))
\end{aligned}$$

Recursive structure of an OBDD

Assume the variable ordering $A_1, A_2, ..., A_n$:

$$
\begin{array}{rcl}
OBDD(\top, \{A_1, A_2, ..., A_n\}) & = & 1 \\
OBDD(\bot, \{A_1, A_2, ..., A_n\}) & = & 0 \\
OBDD(\varphi, \{A_1, A_2, ..., A_n\}) & = & \textit{if } A_1 \\
& & \textit{then } OBDD(\varphi[A_1|\top], \{A_2, ..., A_n\}) \\
& & \textit{else } OBDD(\varphi[A_1|\bot], \{A_2, ..., A_n\})
\end{array}
$$

Incrementally building an OBDD

- $obdd\_build(\top, \{...\}) := \top$,
- $obdd\_build(\bot, \{...\}) := \bot$,
- $obdd\_build(A_i, \{...\}) := ite(A_i, \top, \bot)$,
- $obdd\_build((\neg\varphi), \{A_1, ..., A_n\}) := apply(\neg, obdd\_build(\varphi, \{A_1, ..., A_n\}))$
- $obdd\_build((\varphi_1 \ op\ \varphi_2), \{A_1, ..., A_n\}) :=$
  $reduce($
  $apply(\quad op,$
  $obdd\_build(\varphi_1, \{A_1, ..., A_n\}), \quad op \in \{\wedge, \vee, \rightarrow, \leftarrow, \leftrightarrow, \oplus\}$
  $obdd\_build(\varphi_2, \{A_1, ..., A_n\})$
  $))$

Incrementally building an OBDD

- $obdd\_build(\top, \{...\}) := \top$,
- $obdd\_build(\bot, \{...\}) := \bot$,
- $obdd\_build(A_i, \{...\}) := ite(A_i, \top, \bot)$,
- $obdd\_build((\neg\varphi), \{A_1, ..., A_n\}) := apply(\neg, obdd\_build(\varphi, \{A_1, ..., A_n\}))$
- $obdd\_build((\varphi_1 \ op\ \varphi_2), \{A_1, ..., A_n\}) :=$
  $reduce($
  $\quad apply(\quad op,$
  $\qquad obdd\_build(\varphi_1, \{A_1, ..., A_n\}), \quad op \in \{\wedge, \vee, \rightarrow, \leftarrow, \leftrightarrow, \oplus\}$
  $\qquad obdd\_build(\varphi_2, \{A_1, ..., A_n\})$
  $))$

Incrementally building an OBDD

- $obdd\_build(\top, \{...\}) := \top$,
- $obdd\_build(\bot, \{...\}) := \bot$,
- $obdd\_build(A_i, \{...\}) := ite(A_i, \top, \bot)$,
- $obdd\_build((\neg\varphi), \{A_1, ..., A_n\}) := apply(\neg, obdd\_build(\varphi, \{A_1, ..., A_n\}))$
- $obdd\_build((\varphi_1\ op\ \varphi_2), \{A_1, ..., A_n\}) :=$
  $reduce($
  $apply(\quad op,$
  $obdd\_build(\varphi_1, \{A_1, ..., A_n\}), \quad op \in \{\wedge, \vee, \rightarrow, \leftarrow, \leftrightarrow, \oplus\}$
  $obdd\_build(\varphi_2, \{A_1, ..., A_n\})$
  $))$

Incrementally building an OBDD

- $obdd\_build(\top, \{...\}) := \top$,
- $obdd\_build(\bot, \{...\}) := \bot$,
- $obdd\_build(A_i, \{...\}) := ite(A_i, \top, \bot)$,
- $obdd\_build((\neg\varphi), \{A_1, ..., A_n\}) := apply(\neg, obdd\_build(\varphi, \{A_1, ..., A_n\}))$
- $obdd\_build((\varphi_1\ op\ \varphi_2), \{A_1, ..., A_n\}) :=$
  $reduce($
  $apply(\quad op,$
  $obdd\_build(\varphi_1, \{A_1, ..., A_n\}),\quad op \in \{\wedge, \vee, \rightarrow, \leftarrow, \leftrightarrow, \oplus\}$
  $obdd\_build(\varphi_2, \{A_1, ..., A_n\})$
  $))$

Incrementally building an OBDD

- $obdd\_build(\top, \{...\}) := \top$,
- $obdd\_build(\bot, \{...\}) := \bot$,
- $obdd\_build(A_i, \{...\}) := ite(A_i, \top, \bot)$,
- $obdd\_build((\neg\varphi), \{A_1, ..., A_n\}) := apply(\neg, obdd\_build(\varphi, \{A_1, ..., A_n\}))$
- $obdd\_build((\varphi_1\ op\ \varphi_2), \{A_1, ..., A_n\}) :=$

$$
\begin{array}{l}
reduce(\\
\quad apply(\quad op, \\
\qquad\qquad obdd\_build(\varphi_1, \{A_1, ..., A_n\}), \quad op \in \{\wedge, \vee, \rightarrow, \leftarrow, \leftrightarrow, \oplus\} \\
\qquad\qquad obdd\_build(\varphi_2, \{A_1, ..., A_n\}) \\
))
\end{array}
$$

Incrementally building an OBDD (cont.)

- $apply\ (op,\ O_i, O_j) := (O_i\ op\ O_j)$ **if** $(O_i \in \{\top, \bot\}$ or $O_j \in \{\top, \bot\})$
- $apply\ (\neg,\ ite(A_i, \varphi_i^\top, \varphi_i^\bot)) :=$
  $ite(A_i, apply(\neg, \varphi_i^\top), apply(\neg, \varphi_i^\bot))$
- $apply\ (op,\ ite(A_i, \varphi_i^\top, \varphi_i^\bot),\ ite(A_j, \varphi_j^\top, \varphi_j^\bot)) :=$
  if $(A_i = A_j)$ **then** $ite(A_i,\ apply\ (op, \varphi_i^\top, \varphi_j^\top),\ apply\ (op, \varphi_i^\bot, \varphi_j^\bot))$
  if $(A_i < A_j)$ **then** $ite(A_i,\ apply\ (op, \varphi_i^\top, ite(A_j, \varphi_j^\top, \varphi_j^\bot)),\ apply\ (op, \varphi_i^\bot, ite(A_j, \varphi_j^\top, \varphi_j^\bot)))$
  if $(A_i > A_j)$ **then** $ite(A_j,\ apply\ (op, ite(A_i, \varphi_i^\top, \varphi_i^\bot), \varphi_j^\top),\ apply\ (op, ite(A_i, \varphi_i^\top, \varphi_i^\bot), \varphi_j^\bot))$

$op \in \{\wedge, \vee, \rightarrow, \leftarrow, \leftrightarrow, \oplus\}$

Incrementally building an OBDD (cont.)

- $apply\ (op,\ O_i, O_j) := (O_i\ op\ O_j)$ **if** $(O_i \in \{\top, \bot\}$ or $O_j \in \{\top, \bot\})$
- $apply\ (\neg,\ ite(A_i, \varphi_i^\top, \varphi_i^\bot)) :=$
  $ite(A_i, apply(\neg, \varphi_i^\top), apply(\neg, \varphi_i^\bot))$
- $apply\ (op,\ ite(A_i, \varphi_i^\top, \varphi_i^\bot),\ ite(A_j, \varphi_j^\top, \varphi_j^\bot)) :=$
  if $(A_i = A_j)$ **then** $ite(A_i,\ apply\ (op, \varphi_i^\top, \varphi_j^\top),$
  $apply\ (op, \varphi_i^\bot, \varphi_j^\bot))$
  if $(A_i < A_j)$ **then** $ite(A_i,\ apply\ (op, \varphi_i^\top, ite(A_j, \varphi_j^\top, \varphi_j^\bot)),$
  $apply\ (op, \varphi_i^\bot, ite(A_j, \varphi_j^\top, \varphi_j^\bot)))$
  if $(A_i > A_j)$ **then** $ite(A_j,\ apply\ (op, ite(A_i, \varphi_i^\top, \varphi_i^\bot), \varphi_j^\top),$
  $apply\ (op, ite(A_i, \varphi_i^\top, \varphi_i^\bot), \varphi_j^\bot))$

$op \in \{\wedge, \vee, \rightarrow, \leftarrow, \leftrightarrow, \oplus\}$

Incrementally building an OBDD (cont.)

- $apply\ (op,\ O_i, O_j) := (O_i\ op\ O_j)$ **if** $(O_i \in \{\top, \bot\}$ or $O_j \in \{\top, \bot\})$
- $apply\ (\neg,\ ite(A_i, \varphi_i^\top, \varphi_i^\bot)) :=$
  $ite(A_i, apply(\neg, \varphi_i^\top), apply(\neg, \varphi_i^\bot))$
- $apply\ (op,\ ite(A_i, \varphi_i^\top, \varphi_i^\bot),\ ite(A_j, \varphi_j^\top, \varphi_j^\bot)) :=$

$$
\begin{array}{ll}
\textbf{if } (A_i = A_j) \textbf{ then } ite(A_i, & apply\ (op, \varphi_i^\top, \varphi_j^\top), \\
& apply\ (op, \varphi_i^\bot, \varphi_j^\bot)) \\
\textbf{if } (A_i < A_j) \textbf{ then } ite(A_i, & apply\ (op, \varphi_i^\top, ite(A_j, \varphi_j^\top, \varphi_j^\bot)), \\
& apply\ (op, \varphi_i^\bot, ite(A_j, \varphi_j^\top, \varphi_j^\bot))) \\
\textbf{if } (A_i > A_j) \textbf{ then } ite(A_j, & apply\ (op, ite(A_i, \varphi_i^\top, \varphi_i^\bot), \varphi_j^\top), \\
& apply\ (op, ite(A_i, \varphi_i^\top, \varphi_i^\bot), \varphi_j^\bot))
\end{array}
$$

$op \in \{\wedge, \vee, \rightarrow, \leftarrow, \leftrightarrow, \oplus\}$

Incrementally building an OBDD: Examples

- Ex: build the obdd for $A_1 \vee A_2$ from those of A_1, A_2 (order: A_1, A_2):

$$\begin{aligned} & apply(\vee, \overbrace{ite(A_1, \top, \bot)}^{A_1}, \overbrace{ite(A_2, \top, \bot)}^{A_2}) \\ = \; & ite(A_1,\; apply(\vee, \top, ite(A_2, \top, \bot)),\; apply(\vee, \bot, ite(A_2, \top, \bot))\;) \\ = \; & ite(A_1,\; \top,\; ite(A_2, \top, \bot)\;) \end{aligned}$$

- Ex: build the obdd for $(A_1 \vee A_2) \wedge (A_1 \vee \neg A_2)$ from those of $(A_1 \vee A_2)$, $(A_1 \vee \neg A_2)$ (order: A_1, A_2):

$$\begin{aligned} & apply(\wedge, \overbrace{ite(A_1, \top, ite(A_2, \top, \bot))}^{(A_1 \vee A_2)}, \overbrace{ite(A_1, \top, ite(A_2, \bot, \top))}^{(A_1 \vee \neg A_2)}, \\ = \; & ite(A_1,\; apply(\wedge, \top, \top),\; apply(\wedge,\; ite(A_2, \top, \bot),\; ite(A_2, \bot, \top)) \\ = \; & ite(A_1,\; \top,\; ite(A_2,\; apply(\wedge, \top, \bot),\; apply(\wedge, \bot, \top))) \\ = \; & ite(A_1,\; \top,\; ite(A_2,\; \bot,\; \bot)) \\ = \; & ite(A_1,\; \top,\; \bot) \end{aligned}$$

Incrementally building an OBDD: Examples

- Ex: build the obdd for $A_1 \vee A_2$ from those of A_1, A_2 (order: A_1, A_2):

$$
\begin{array}{rl}
& apply(\vee, \overbrace{ite(A_1, \top, \bot)}^{A_1}, \overbrace{ite(A_2, \top, \bot)}^{A_2}) \\
= & ite(A_1,\ apply(\vee, \top, ite(A_2, \top, \bot)),\ apply(\vee, \bot, ite(A_2, \top, \bot))\ ) \\
= & ite(A_1,\ \top,\ ite(A_2, \top, \bot)\ )
\end{array}
$$

- Ex: build the obdd for $(A_1 \vee A_2) \wedge (A_1 \vee \neg A_2)$ from those of $(A_1 \vee A_2)$, $(A_1 \vee \neg A_2)$ (order: A_1, A_2):

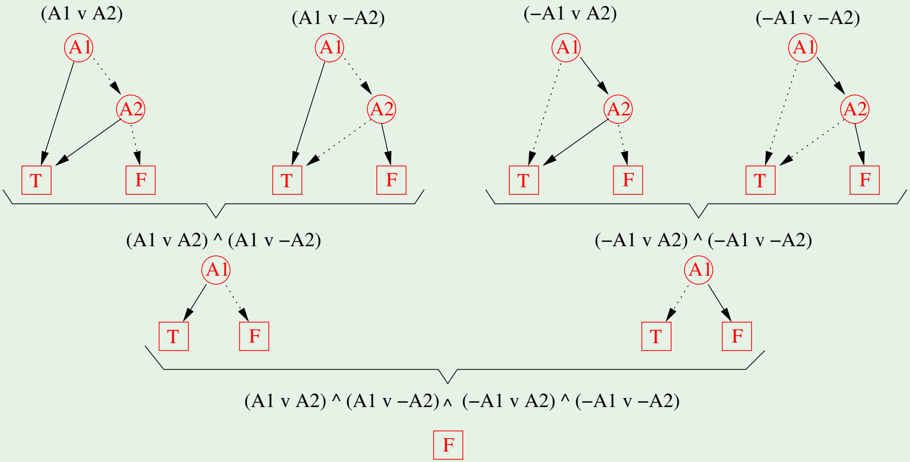
$$
\begin{array}{rl}
& apply(\wedge,\ \overbrace{ite(A_1, \top, ite(A_2, \top, \bot))}^{(A_1 \vee A_2)},\ \overbrace{ite(A_1, \top, ite(A_2, \bot, \top))}^{(A_1 \vee \neg A_2)}, \\
= & ite(A_1,\ apply(\wedge, \top, \top),\ apply(\wedge,\ ite(A_2, \top, \bot),\ ite(A_2, \bot, \top)) \\
= & ite(A_1,\ \top,\ ite(A_2,\ apply(\wedge, \top, \bot),\ apply(\wedge, \bot, \top))) \\
= & ite(A_1,\ \top,\ ite(A_2,\ \bot,\ \bot)) \\
= & ite(A_1,\ \top,\ \bot)
\end{array}
$$

OBDD incremental building – example

$$\varphi = (A_1 \vee A_2) \wedge (A_1 \vee \neg A_2) \wedge (\neg A_1 \vee A_2) \wedge (\neg A_1 \vee \neg A_2)$$

OBDD incremental building – example

$$\varphi = (A_1 \vee A_2) \wedge (A_1 \vee \neg A_2) \wedge (\neg A_1 \vee A_2) \wedge (\neg A_1 \vee \neg A_2)$$



Critical choice of variable Orderings in OBDD's

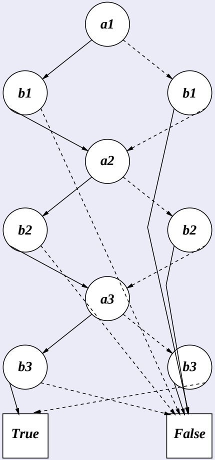
$$(a_1 \leftrightarrow b_1) \wedge (a_2 \leftrightarrow b_2) \wedge (a_3 \leftrightarrow b_3)$$

Linear size

Exponential size

Critical choice of variable Orderings in OBDD's

$$(a_1 \leftrightarrow b_1) \wedge (a_2 \leftrightarrow b_2) \wedge (a_3 \leftrightarrow b_3)$$

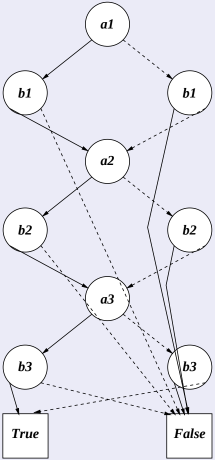


Linear size

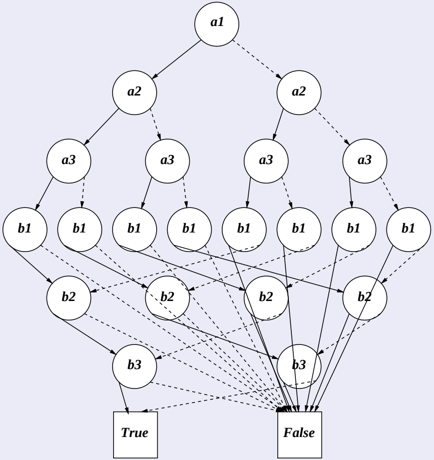
Exponential size

Critical choice of variable Orderings in OBDD's

$$(a_1 \leftrightarrow b_1) \wedge (a_2 \leftrightarrow b_2) \wedge (a_3 \leftrightarrow b_3)$$



Linear size



Exponential size

OBDD's as canonical representation of Boolean formulas

- An OBDD is a canonical representation of a Boolean formula: once the variable ordering is established, equivalent formulas are represented by the same OBDD:

$$\varphi_1 \leftrightarrow \varphi_2 \iff OBDD(\varphi_1) = OBDD(\varphi_2)$$

- equivalence check requires constant time!
  $\Longrightarrow$validity check requires constant time! $(\varphi \leftrightarrow \top)$
  $\Longrightarrow$(un)satisfiability check requires constant time! $(\varphi \leftrightarrow \bot)$
- the set of the paths from the root to 1 represent all the models of the formula
- the set of the paths from the root to 0 represent all the counter-models of the formula

OBDD's as canonical representation of Boolean formulas

- An OBDD is a canonical representation of a Boolean formula: once the variable ordering is established, equivalent formulas are represented by the same OBDD:

$$\varphi_1 \leftrightarrow \varphi_2 \iff OBDD(\varphi_1) = OBDD(\varphi_2)$$

- equivalence check requires constant time!
  $\Longrightarrow$validity check requires constant time! ($\varphi \leftrightarrow \top$)
  $\Longrightarrow$(un)satisfiability check requires constant time! ($\varphi \leftrightarrow \bot$)
- the set of the paths from the root to 1 represent all the models of the formula
- the set of the paths from the root to 0 represent all the counter-models of the formula

OBDD's as canonical representation of Boolean formulas

- An OBDD is a canonical representation of a Boolean formula: once the variable ordering is established, equivalent formulas are represented by the same OBDD:

$$\varphi_1 \leftrightarrow \varphi_2 \iff OBDD(\varphi_1) = OBDD(\varphi_2)$$

- equivalence check requires constant time!
  $\Longrightarrow$validity check requires constant time! ($\varphi \leftrightarrow \top$)
  $\Longrightarrow$(un)satisfiability check requires constant time! ($\varphi \leftrightarrow \bot$)
- the set of the paths from the root to 1 represent all the models of the formula
- the set of the paths from the root to 0 represent all the counter-models of the formula

Exponentiality of OBDD's

- The size of OBDD's may grow exponentially wrt. the number of variables in worst-case
- Consequence of the canonicity of OBDD's (unless P = co-NP)
- Example: there exist no polynomial-size OBDD representing the electronic circuit of a bitwise multiplier

Note

The size of intermediate OBDD's may be bigger than that of the final one (e.g., inconsistent formula)

Exponentiality of OBDD's

- The size of OBDD's may grow exponentially wrt. the number of variables in worst-case
- Consequence of the canonicity of OBDD's (unless P = co-NP)
- Example: there exist no polynomial-size OBDD representing the electronic circuit of a bitwise multiplier

Note

The size of intermediate OBDD's may be bigger than that of the final one (e.g., inconsistent formula)

Exponentiality of OBDD’s

- The size of OBDD’s may grow exponentially wrt. the number of variables in worst-case
- Consequence of the canonicity of OBDD’s (unless P = co-NP)
- Example: there exist no polynomial-size OBDD representing the electronic circuit of a bitwise multiplier

Note

The size of intermediate OBDD’s may be bigger than that of the final one (e.g., inconsistent formula)

Exponentiality of OBDD's

- The size of OBDD's may grow exponentially wrt. the number of variables in worst-case
- Consequence of the canonicity of OBDD's (unless P = co-NP)
- Example: there exist no polynomial-size OBDD representing the electronic circuit of a bitwise multiplier

Note

The size of intermediate OBDD's may be bigger than that of the final one (e.g., inconsistent formula)

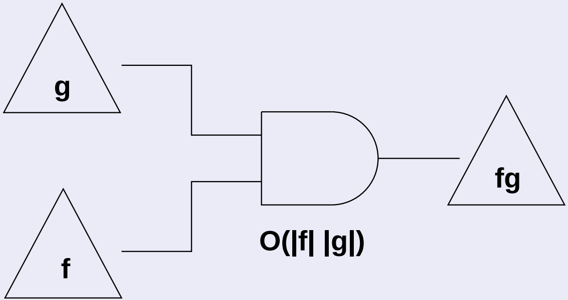
Useful Operations over OBDDs

- the equivalence check between two OBDDs is simple
  - are they the same OBDD? ($\Longrightarrow$ constant time)
- the size of a Boolean composition is up to the product of the size of the operands: $|f \; op \; g| = O(|f| \cdot |g|)$

(but typically much smaller on average).

Useful Operations over OBDDs

- the equivalence check between two OBDDs is simple
  - are they the same OBDD? ($\Longrightarrow$ constant time)
- the size of a Boolean composition is up to the product of the size of the operands: $|f\ op\ g| = O(|f| \cdot |g|)$



(but typically much smaller on average).

[Recall] Boolean Quantification

Shannon's expansion:

- If v is a Boolean variable and f is a Boolean formula, then

$$\begin{aligned} \exists v.\varphi &:= \varphi|_{v=\perp} \vee \varphi|_{v=\top} \\ \forall v.\varphi &:= \varphi|_{v=\perp} \wedge \varphi|_{v=\top} \end{aligned}$$

- v does no more occur in $\exists v.\varphi$ and $\forall v.\varphi$!!
- Multi-variable quantification: $\exists(w_1, \ldots, w_n).\varphi \ := \ \exists w_1 \ldots \exists w_n.\varphi$

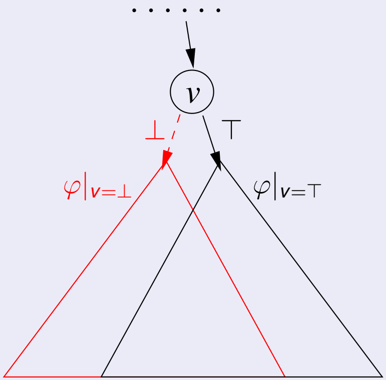
- Intuition:
  - $\mu \models \exists v.\varphi$ iff exists *truthvalue* $\in \{\top, \perp\}$ s.t. $\mu \cup \{v := \mathit{truthvalue}\} \models \varphi$
  - $\mu \models \forall v.\varphi$ iff forall *truthvalue* $\in \{\top, \perp\}$, $\mu \cup \{v := \mathit{truthvalue}\} \models \varphi$
- Example: $\exists(b, c).((a \wedge b) \vee (c \wedge d)) \ = \ a \vee d$

Note

Naive expansion of quantifiers to propositional logic may cause a blow-up in size of the formulae

OBDD's and Boolean quantification

- OBDD's handle quantification operations quite efficiently
  - if f is a sub-OBDD labeled by variable v, then $\varphi|_{v=\top}$ and $\varphi|_{v=\bot}$ are the "then" and "else" branches of f



$\Longrightarrow$ lots of sharing of subformulae!

Example

Let $\varphi \stackrel{\text{def}}{=} (A \wedge (B \vee C))$ and $\varphi' \stackrel{\text{def}}{=} \exists A.\forall B.\varphi$. Using the variable ordering "A, B, C", draw the OBDD corresponding to the formulas φ and φ'.

Example

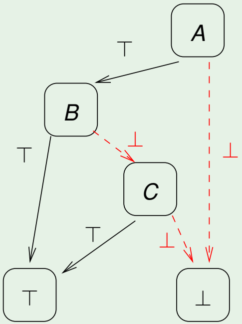
Let $\varphi \stackrel{\text{def}}{=} (A \wedge (B \vee C))$ and $\varphi' \stackrel{\text{def}}{=} \exists A.\forall B.\varphi$. Using the variable ordering "A, B, C", draw the OBDD corresponding to the formulas φ and φ'.

$\varphi \stackrel{\text{def}}{=} (A \wedge (B \vee C))$

Example

Let $\varphi \stackrel{\text{def}}{=} (A \wedge (B \vee C))$ and $\varphi' \stackrel{\text{def}}{=} \exists A.\forall B.\varphi$. Using the variable ordering “A, B, C”, draw the OBDD corresponding to the formulas φ and φ'.

$\varphi \stackrel{\text{def}}{=} (A \wedge (B \vee C))$



Example (cont.)

$\varphi' \stackrel{\text{def}}{=} \exists A.\forall B.(A \wedge (B \vee C))$

which corresponds to the following OBDD:

Example (cont.)

$\varphi' \stackrel{\text{def}}{=} \exists A.\forall B.(A \wedge (B \vee C))$

$$
\begin{array}{rllll}
\varphi' \stackrel{\text{def}}{=} & \exists A.\forall B.\varphi & & & \\
= & \forall B.(A \wedge (B \vee C)))[A := \top] & & \vee & (\forall B.(A \wedge (B \vee C)))[A := \bot] \\
= & \forall B.(B \vee C)) & & \vee & \forall B.\bot \\
= & ((B \vee C)[B := \top] & \wedge \quad (B \vee C)[B := \bot]) & \vee & \bot \\
= & (\top & \wedge \quad C) & & \\
= & C & & &
\end{array}
$$

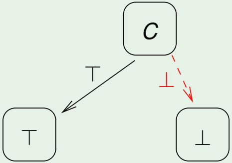
which corresponds to the following OBDD:

Example (cont.)

$\varphi' \stackrel{\text{def}}{=} \exists A.\forall B.(A \wedge (B \vee C))$

$$
\begin{array}{rllll}
\varphi' \stackrel{\text{def}}{=} & \exists A.\forall B.\varphi & & & \\
= & \forall B.(A \wedge (B \vee C)))[A := \top] & & \vee & (\forall B.(A \wedge (B \vee C)))[A := \bot] \\
= & \forall B.(B \vee C)) & & \vee & \forall B.\bot \\
= & ((B \vee C)[B := \top] & \wedge \quad (B \vee C)[B := \bot]) & \vee & \bot \\
= & (\top & \wedge \quad C) & & \\
= & C & & &
\end{array}
$$

which corresponds to the following OBDD:



OBDD – summary

- Factorize common parts of the search tree (DAG)
- Require setting a variable ordering a priori (critical!)
- Canonical representation of a Boolean formula.
- Once built, logical operations (satisfiability, validity, equivalence) immediate.
- Represents all models and counter-models of the formula.
- Require exponential space in worst-case
- Very efficient for some practical problems (circuits, symbolic model checking).

Exercises

1. Let
$$\varphi \stackrel{\text{def}}{=} \neg \left(\begin{array}{rcrl} & & (A_1) & \wedge \\ (A_1 & \rightarrow & A_2) & \wedge \\ (A_2 & \rightarrow & A_3) & \wedge \\ (A_3 & \rightarrow & A_4) & \wedge \\ (A_4 & \rightarrow & A_5) & \wedge \end{array} \right)$$
Using the variable ordering: " A_1 A_2, A_3, A_4, A_5", draw the OBDD corresponding to φ

2. Let
$$\varphi \stackrel{\text{def}}{=} A_1 \rightarrow (A_2 \leftrightarrow \left(\begin{array}{ll} (A_3 \vee A_6 \vee A_8) & \wedge \\ (A_5 \vee A_7 \vee A_8) & \wedge \\ (\neg A_4 \vee \neg A_6 \vee \neg A_8) & \wedge \\ (\neg A_6 \vee A_7 \vee \neg A_8) & \wedge \\ (\neg A_3 \vee A_6 \vee A_9) & \wedge \\ (\neg A_6 \vee \neg A_8 \vee \neg A_9) & \wedge \\ (A_3 \vee A_4 \vee \neg A_5) & \wedge \\ (\neg A_4 \vee \neg A_7 \vee \neg A_9) & \end{array} \right)).$$
Using the variable ordering: " A_1, A_3, A_4, A_5, A_6, A_7, A_8, A_9", draw the OBDD corresponding to the formula φ' defined as: $\varphi' \stackrel{\text{def}}{=} \forall A_2.\varphi$.

Exercises

1. Let
$$\varphi \stackrel{\text{def}}{=} \neg \left(\begin{array}{rrrl} & (& A_1) & \wedge \\ (A_1 \rightarrow & & A_2) & \wedge \\ (A_2 \rightarrow & & A_3) & \wedge \\ (A_3 \rightarrow & & A_4) & \wedge \\ (A_4 \rightarrow & & A_5) & \wedge \end{array} \right)$$
Using the variable ordering: " A_1 A_2, A_3, A_4, A_5", draw the OBDD corresponding to φ

2. Let
$$\varphi \stackrel{\text{def}}{=} A_1 \rightarrow (A_2 \leftrightarrow \left(\begin{array}{lllll} (A_3 \vee & A_6 \vee & A_8) & \wedge \\ (A_5 \vee & A_7 \vee & A_8) & \wedge \\ (\neg A_4 \vee & \neg A_6 \vee & \neg A_8) & \wedge \\ (\neg A_6 \vee & A_7 \vee & \neg A_8) & \wedge \\ (\neg A_3 \vee & A_6 \vee & A_9) & \wedge \\ (\neg A_6 \vee & \neg A_8 \vee & \neg A_9) & \wedge \\ (A_3 \vee & A_4 \vee & \neg A_5) & \wedge \\ (\neg A_4 \vee & \neg A_7 \vee & \neg A_9) & \end{array} \right)).$$
Using the variable ordering: " A_1, A_3, A_4, A_5, A_6, A_7, A_8, A_9", draw the OBDD corresponding to the formula φ' defined as: $\varphi' \stackrel{\text{def}}{=} \forall A_2.\varphi$.

Outline

Outline

DPLL: “Classic” chronological backtracking

DPLL implements “classic” chronological backtracking:

- variable assignments (literals) stored in a stack
- each variable assignments labeled as “unit”, “open”, “closed”
- when a conflict is encountered, the stack is popped up to the most recent open assignment l
- l is toggled, is labeled as “closed”, and the search proceeds.

DPLL Chronological Backtracking: Drawbacks

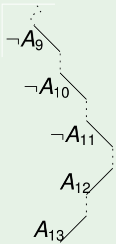
Chronological backtracking always backtracks to the most recent branching point, even though a higher backtrack could be possible
$\Longrightarrow$ lots of useless search!

DPLL Chronological Backtracking: Example

$c_1 : \neg A_1 \vee A_2$
$c_2 : \neg A_1 \vee A_3 \vee A_9$
$c_3 : \neg A_2 \vee \neg A_3 \vee A_4$
$c_4 : \neg A_4 \vee A_5 \vee A_{10}$
$c_5 : \neg A_4 \vee A_6 \vee A_{11}$
$c_6 : \neg A_5 \vee \neg A_6$
$c_7 : A_1 \vee A_7 \vee \neg A_{12}$
$c_8 : A_1 \vee A_8$
$c_9 : \neg A_7 \vee \neg A_8 \vee \neg A_{13}$
$\ldots$

DPLL Chronological Backtracking: Example

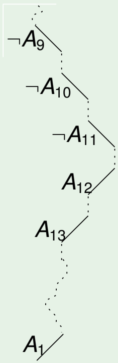
$c_1 : \neg A_1 \lor A_2$
$c_2 : \neg A_1 \lor A_3 \lor A_9$
$c_3 : \neg A_2 \lor \neg A_3 \lor A_4$
$c_4 : \neg A_4 \lor A_5 \lor A_{10}$
$c_5 : \neg A_4 \lor A_6 \lor A_{11}$
$c_6 : \neg A_5 \lor \neg A_6$
$c_7 : A_1 \lor A_7 \lor \neg A_{12}$
$c_8 : A_1 \lor A_8$
$c_9 : \neg A_7 \lor \neg A_8 \lor \neg A_{13}$
$\ldots$



$\{\ldots, \neg A_9, \ldots, \neg A_{10}, \ldots, \neg A_{11}, \ldots, A_{12}, \ldots, A_{13}, \ldots\}$
(initial assignment)

DPLL Chronological Backtracking: Example

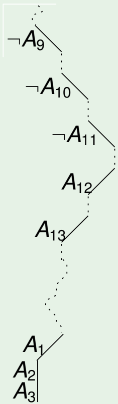
c_1 : $\neg A_1 \vee A_2$
c_2 : $\neg A_1 \vee A_3 \vee A_9$
c_3 : $\neg A_2 \vee \neg A_3 \vee A_4$
c_4 : $\neg A_4 \vee A_5 \vee A_{10}$
c_5 : $\neg A_4 \vee A_6 \vee A_{11}$
c_6 : $\neg A_5 \vee \neg A_6$
c_7 : $A_1 \vee A_7 \vee \neg A_{12}$ $\surd$
c_8 : $A_1 \vee A_8$ $\surd$
c_9 : $\neg A_7 \vee \neg A_8 \vee \neg A_{13}$
...



$\{..., \neg A_9, ..., \neg A_{10}, ..., \neg A_{11}, ..., A_{12}, ..., A_{13}, ..., A_1\}$
... (branch on A_1)

DPLL Chronological Backtracking: Example

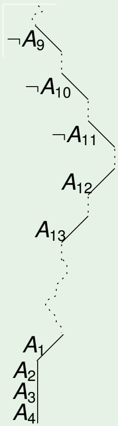
$c_1 : \neg A_1 \vee A_2$ √
$c_2 : \neg A_1 \vee A_3 \vee A_9$ √
$c_3 : \neg A_2 \vee \neg A_3 \vee A_4$
$c_4 : \neg A_4 \vee A_5 \vee A_{10}$
$c_5 : \neg A_4 \vee A_6 \vee A_{11}$
$c_6 : \neg A_5 \vee \neg A_6$
$c_7 : A_1 \vee A_7 \vee \neg A_{12}$ √
$c_8 : A_1 \vee A_8$ √
$c_9 : \neg A_7 \vee \neg A_8 \vee \neg A_{13}$
...



$\{..., \neg A_9, ..., \neg A_{10}, ..., \neg A_{11}, ..., A_{12}, ..., A_{13}, ..., A_1, A_2, A_3\}$
(unit A_2, A_3)

DPLL Chronological Backtracking: Example

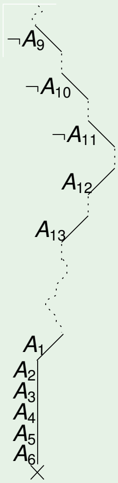
$c_1 : \neg A_1 \vee A_2$ $\surd$
$c_2 : \neg A_1 \vee A_3 \vee A_9$ $\surd$
$c_3 : \neg A_2 \vee \neg A_3 \vee A_4$ $\surd$
$c_4 : \neg A_4 \vee A_5 \vee A_{10}$
$c_5 : \neg A_4 \vee A_6 \vee A_{11}$
$c_6 : \neg A_5 \vee \neg A_6$
$c_7 : A_1 \vee A_7 \vee \neg A_{12}$ $\surd$
$c_8 : A_1 \vee A_8$ $\surd$
$c_9 : \neg A_7 \vee \neg A_8 \vee \neg A_{13}$
...



$\{..., \neg A_9, ..., \neg A_{10}, ..., \neg A_{11}, ..., A_{12}, ..., A_{13}, ..., A_1, A_2, A_3, A_4\}$
(unit A_4)

DPLL Chronological Backtracking: Example

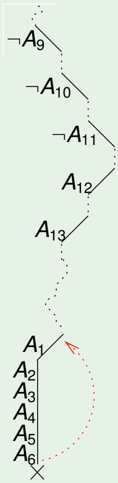
$c_1 : \neg A_1 \vee A_2$ $\surd$
$c_2 : \neg A_1 \vee A_3 \vee A_9$ $\surd$
$c_3 : \neg A_2 \vee \neg A_3 \vee A_4$ $\surd$
$c_4 : \neg A_4 \vee A_5 \vee A_{10}$ $\surd$
$c_5 : \neg A_4 \vee A_6 \vee A_{11}$ $\surd$
$c_6 : \neg A_5 \vee \neg A_6$ $\times$
$c_7 : A_1 \vee A_7 \vee \neg A_{12}$ $\surd$
$c_8 : A_1 \vee A_8$ $\surd$
$c_9 : \neg A_7 \vee \neg A_8 \vee \neg A_{13}$
...



$\{\ldots, \neg A_9, \ldots, \neg A_{10}, \ldots, \neg A_{11}, \ldots, A_{12}, \ldots, A_{13}, \ldots, A_1, A_2, A_3, A_4, A_5, A_6\}$
(unit A_5, A_6) $\Longrightarrow$ conflict

DPLL Chronological Backtracking: Example

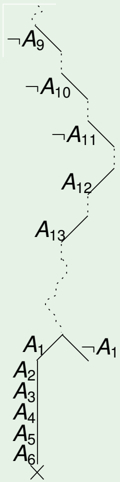
$c_1 : \neg A_1 \vee A_2$
$c_2 : \neg A_1 \vee A_3 \vee A_9$
$c_3 : \neg A_2 \vee \neg A_3 \vee A_4$
$c_4 : \neg A_4 \vee A_5 \vee A_{10}$
$c_5 : \neg A_4 \vee A_6 \vee A_{11}$
$c_6 : \neg A_5 \vee \neg A_6$
$c_7 : A_1 \vee A_7 \vee \neg A_{12}$
$c_8 : A_1 \vee A_8$
$c_9 : \neg A_7 \vee \neg A_8 \vee \neg A_{13}$
...



$\{..., \neg A_9, ..., \neg A_{10}, ..., \neg A_{11}, ..., A_{12}, ..., A_{13}, ...\}$
$\Longrightarrow$ backtrack up to A_1

DPLL Chronological Backtracking: Example

$c_1 : \neg A_1 \vee A_2$ $\surd$
$c_2 : \neg A_1 \vee A_3 \vee A_9$ $\surd$
$c_3 : \neg A_2 \vee \neg A_3 \vee A_4$
$c_4 : \neg A_4 \vee A_5 \vee A_{10}$
$c_5 : \neg A_4 \vee A_6 \vee A_{11}$
$c_6 : \neg A_5 \vee \neg A_6$
$c_7 : A_1 \vee A_7 \vee \neg A_{12}$
$c_8 : A_1 \vee A_8$
$c_9 : \neg A_7 \vee \neg A_8 \vee \neg A_{13}$
$\ldots$



$\{\ldots, \neg A_9, \ldots, \neg A_{10}, \ldots, \neg A_{11}, \ldots, A_{12}, \ldots, A_{13}, \ldots, \neg A_1\}$
(unit $\neg A_1$)

DPLL Chronological Backtracking: Example

$c_1 : \neg A_1 \vee A_2$ $\surd$

$c_2 : \neg A_1 \vee A_3 \vee A_9$ $\surd$

$c_3 : \neg A_2 \vee \neg A_3 \vee A_4$

$c_4 : \neg A_4 \vee A_5 \vee A_{10}$

$c_5 : \neg A_4 \vee A_6 \vee A_{11}$

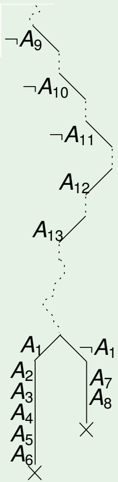
$c_6 : \neg A_5 \vee \neg A_6$

$c_7 : A_1 \vee A_7 \vee \neg A_{12}$ $\surd$

$c_8 : A_1 \vee A_8$ $\surd$

$c_9 : \neg A_7 \vee \neg A_8 \vee \neg A_{13}$ $\times$

$\ldots$

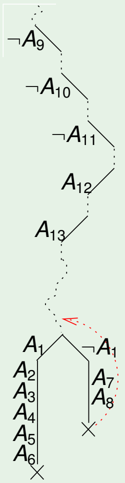


$\{\ldots, \neg A_9, \ldots, \neg A_{10}, \ldots, \neg A_{11}, \ldots, A_{12}, \ldots, A_{13}, \ldots, \neg A_1, A_7, A_8\}$

(unit $A_7,\ A_8$) $\Longrightarrow$ conflict

DPLL Chronological Backtracking: Example

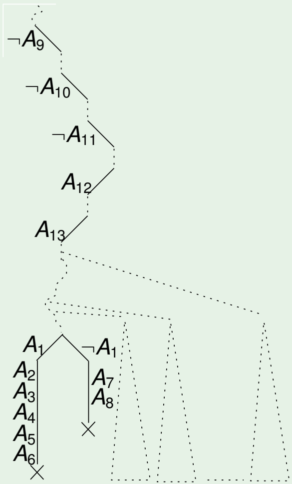
$c_1 : \neg A_1 \vee A_2$
$c_2 : \neg A_1 \vee A_3 \vee A_9$
$c_3 : \neg A_2 \vee \neg A_3 \vee A_4$
$c_4 : \neg A_4 \vee A_5 \vee A_{10}$
$c_5 : \neg A_4 \vee A_6 \vee A_{11}$
$c_6 : \neg A_5 \vee \neg A_6$
$c_7 : A_1 \vee A_7 \vee \neg A_{12}$
$c_8 : A_1 \vee A_8$
$c_9 : \neg A_7 \vee \neg A_8 \vee \neg A_{13}$
...



$\{..., \neg A_9, ..., \neg A_{10}, ..., \neg A_{11}, ..., A_{12}, ..., A_{13}, ...\}$
$\Longrightarrow$ backtrack to the most recent open branching point

DPLL Chronological Backtracking: Example

$c_1 : \neg A_1 \vee A_2$
$c_2 : \neg A_1 \vee A_3 \vee A_9$
$c_3 : \neg A_2 \vee \neg A_3 \vee A_4$
$c_4 : \neg A_4 \vee A_5 \vee A_{10}$
$c_5 : \neg A_4 \vee A_6 \vee A_{11}$
$c_6 : \neg A_5 \vee \neg A_6$
$c_7 : A_1 \vee A_7 \vee \neg A_{12}$
$c_8 : A_1 \vee A_8$
$c_9 : \neg A_7 \vee \neg A_8 \vee \neg A_{13}$
$\ldots$



$\{\ldots, \neg A_9, \ldots, \neg A_{10}, \ldots, \neg A_{11}, \ldots, A_{12}, \ldots, A_{13}, \ldots\}$
$\Longrightarrow$ lots of useless search before backtracking up to A_{13}!

Outline

1. Boolean Logics and SAT
2. Basic SAT-Solving Techniques
   - Generalities
   - Resolution
   - Tableaux
   - DPLL
3. Ordered Binary Decision Diagrams – OBDDs
4. Modern CDCL SAT Solvers
   - Limitations of Chronological Backtracking
   - Conflict-Driven Clause-Learning SAT solvers
   - Further Improvements
   - SAT Under Assumptions & Incremental SAT
5. SAT Functionalities: proofs, unsat cores, optimization

Modern Conflict-Driven Clause-Learning SAT Solvers

- Non-recursive, stack-based implementations
- Based on Conflict-Driven Clause-Learning (CDCL) schema
  - inspired to conflict-driven backjumping and learning in CSPs
  - learns implied clauses as nogoods
- Random restarts
  - abandon the current search tree and restart on top level
  - previously-learned clauses maintained
- Smart literal selection heuristics (ex: VSIDS)
  - “static”: scores updated only at the end of a branch
  - “local”: privileges variable in recently learned clauses
- Smart preprocessing/inprocessing technique to simplify formulas
- Smart indexing techniques (e.g. 2-watched literals)
  - efficiently do/undo assignments and reveal unit clauses
- Allow Incremental Calls (stack-based interface)
  - allow for reusing previous search on “similar” problems

Can handle industrial problems with $10^6 - 10^7$ variables and clauses!

Modern Conflict-Driven Clause-Learning SAT Solvers

- Non-recursive, stack-based implementations
- Based on Conflict-Driven Clause-Learning (CDCL) schema
  - inspired to conflict-driven backjumping and learning in CSPs
  - learns implied clauses as nogoods
- Random restarts
  - abandon the current search tree and restart on top level
  - previously-learned clauses maintained
- Smart literal selection heuristics (ex: VSIDS)
  - "static": scores updated only at the end of a branch
  - "local": privileges variable in recently learned clauses
- Smart preprocessing/inprocessing technique to simplify formulas
- Smart indexing techniques (e.g. 2-watched literals)
  - efficiently do/undo assignments and reveal unit clauses
- Allow Incremental Calls (stack-based interface)
  - allow for reusing previous search on "similar" problems

Can handle industrial problems with $10^6 - 10^7$ variables and clauses!

Modern Conflict-Driven Clause-Learning SAT Solvers

- Non-recursive, stack-based implementations
- Based on Conflict-Driven Clause-Learning (CDCL) schema
  - inspired to conflict-driven backjumping and learning in CSPs
  - learns implied clauses as nogoods
- Random restarts
  - abandon the current search tree and restart on top level
  - previously-learned clauses maintained
- Smart literal selection heuristics (ex: VSIDS)
  - "static": scores updated only at the end of a branch
  - "local": privileges variable in recently learned clauses
- Smart preprocessing/inprocessing technique to simplify formulas
- Smart indexing techniques (e.g. 2-watched literals)
  - efficiently do/undo assignments and reveal unit clauses
- Allow Incremental Calls (stack-based interface)
  - allow for reusing previous search on "similar" problems

Can handle industrial problems with $10^6 - 10^7$ variables and clauses!

Modern Conflict-Driven Clause-Learning SAT Solvers

- Non-recursive, stack-based implementations
- Based on Conflict-Driven Clause-Learning (CDCL) schema
  - inspired to conflict-driven backjumping and learning in CSPs
  - learns implied clauses as nogoods
- Random restarts
  - abandon the current search tree and restart on top level
  - previously-learned clauses maintained
- Smart literal selection heuristics (ex: VSIDS)
  - “static”: scores updated only at the end of a branch
  - “local”: privileges variable in recently learned clauses
- Smart preprocessing/inprocessing technique to simplify formulas
- Smart indexing techniques (e.g. 2-watched literals)
  - efficiently do/undo assignments and reveal unit clauses
- Allow Incremental Calls (stack-based interface)
  - allow for reusing previous search on “similar” problems

Can handle industrial problems with $10^6 - 10^7$ variables and clauses!

Modern Conflict-Driven Clause-Learning SAT Solvers

- Non-recursive, stack-based implementations
- Based on Conflict-Driven Clause-Learning (CDCL) schema
  - inspired to conflict-driven backjumping and learning in CSPs
  - learns implied clauses as nogoods
- Random restarts
  - abandon the current search tree and restart on top level
  - previously-learned clauses maintained
- Smart literal selection heuristics (ex: VSIDS)
  - "static": scores updated only at the end of a branch
  - "local": privileges variable in recently learned clauses
- Smart preprocessing/inprocessing technique to simplify formulas
- Smart indexing techniques (e.g. 2-watched literals)
  - efficiently do/undo assignments and reveal unit clauses
- Allow Incremental Calls (stack-based interface)
  - allow for reusing previous search on "similar" problems

Can handle industrial problems with $10^6 - 10^7$ variables and clauses!

Modern Conflict-Driven Clause-Learning SAT Solvers

- Non-recursive, stack-based implementations
- Based on Conflict-Driven Clause-Learning (CDCL) schema
  - inspired to conflict-driven backjumping and learning in CSPs
  - learns implied clauses as nogoods
- Random restarts
  - abandon the current search tree and restart on top level
  - previously-learned clauses maintained
- Smart literal selection heuristics (ex: VSIDS)
  - “static”: scores updated only at the end of a branch
  - “local”: privileges variable in recently learned clauses
- Smart preprocessing/inprocessing technique to simplify formulas
- Smart indexing techniques (e.g. 2-watched literals)
  - efficiently do/undo assignments and reveal unit clauses
- Allow Incremental Calls (stack-based interface)
  - allow for reusing previous search on “similar” problems

Can handle industrial problems with $10^6 - 10^7$ variables and clauses!

Modern Conflict-Driven Clause-Learning SAT Solvers

- Non-recursive, stack-based implementations
- Based on Conflict-Driven Clause-Learning (CDCL) schema
  - inspired to conflict-driven backjumping and learning in CSPs
  - learns implied clauses as nogoods
- Random restarts
  - abandon the current search tree and restart on top level
  - previously-learned clauses maintained
- Smart literal selection heuristics (ex: VSIDS)
  - "static": scores updated only at the end of a branch
  - "local": privileges variable in recently learned clauses
- Smart preprocessing/inprocessing technique to simplify formulas
- Smart indexing techniques (e.g. 2-watched literals)
  - efficiently do/undo assignments and reveal unit clauses
- Allow Incremental Calls (stack-based interface)
  - allow for reusing previous search on "similar" problems

Can handle industrial problems with $10^6 - 10^7$ variables and clauses!

Modern Conflict-Driven Clause-Learning SAT Solvers

- Non-recursive, stack-based implementations
- Based on Conflict-Driven Clause-Learning (CDCL) schema
  - inspired to conflict-driven backjumping and learning in CSPs
  - learns implied clauses as nogoods
- Random restarts
  - abandon the current search tree and restart on top level
  - previously-learned clauses maintained
- Smart literal selection heuristics (ex: VSIDS)
  - “static”: scores updated only at the end of a branch
  - “local”: privileges variable in recently learned clauses
- Smart preprocessing/inprocessing technique to simplify formulas
- Smart indexing techniques (e.g. 2-watched literals)
  - efficiently do/undo assignments and reveal unit clauses
- Allow Incremental Calls (stack-based interface)
  - allow for reusing previous search on “similar” problems

Can handle industrial problems with $10^6 - 10^7$ variables and clauses!

Modern Conflict-Driven Clause-Learning SAT Solvers

- Non-recursive, stack-based implementations
- Based on Conflict-Driven Clause-Learning (CDCL) schema
  - inspired to conflict-driven backjumping and learning in CSPs
  - learns implied clauses as nogoods
- Random restarts
  - abandon the current search tree and restart on top level
  - previously-learned clauses maintained
- Smart literal selection heuristics (ex: VSIDS)
  - "static": scores updated only at the end of a branch
  - "local": privileges variable in recently learned clauses
- Smart preprocessing/inprocessing technique to simplify formulas
- Smart indexing techniques (e.g. 2-watched literals)
  - efficiently do/undo assignments and reveal unit clauses
- Allow Incremental Calls (stack-based interface)
  - allow for reusing previous search on "similar" problems

Can handle industrial problems with $10^6 - 10^7$ variables and clauses!

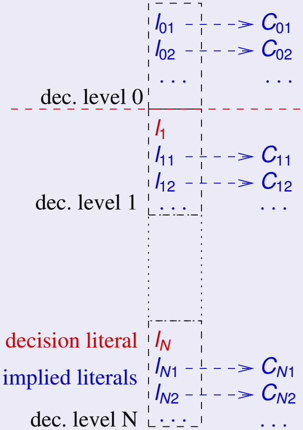
Modern Conflict-Driven Clause-Learning SAT Solvers

- Non-recursive, stack-based implementations
- Based on Conflict-Driven Clause-Learning (CDCL) schema
  - inspired to conflict-driven backjumping and learning in CSPs
  - learns implied clauses as nogoods
- Random restarts
  - abandon the current search tree and restart on top level
  - previously-learned clauses maintained
- Smart literal selection heuristics (ex: VSIDS)
  - "static": scores updated only at the end of a branch
  - "local": privileges variable in recently learned clauses
- Smart preprocessing/inprocessing technique to simplify formulas
- Smart indexing techniques (e.g. 2-watched literals)
  - efficiently do/undo assignments and reveal unit clauses
- Allow Incremental Calls (stack-based interface)
  - allow for reusing previous search on "similar" problems

Can handle industrial problems with $10^6 - 10^7$ variables and clauses!

Stack-based representation of a truth assignment μ

- assign one truth-value at a time (add one literal to a stack representing μ)
- stack partitioned into decision levels:
  - one decision literal
  - its implied literals
  - each implied literal tagged with the clause causing its unit-propagation (antecedent clause)
- equivalent to an implication graph



Implication graph

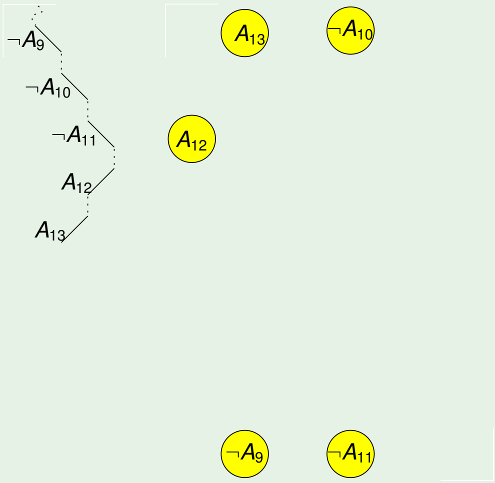
- An implication graph is a DAG s.t.:
  - each node represents a variable assignment (literal)
  - each edge $l_i \xmapsto{c} l$ is labeled with a clause
  - the node of a decision literal has no incoming edges
  - all edges incoming into a node l are labeled with the same clause c, s.t. $l_1 \xmapsto{c} l, ..., l_n \xmapsto{c} l$ iff $c = \neg l_1 \vee ... \vee \neg l_n \vee l$
    (c is said to be the antecedent clause of l)
  - when both l and $\neg l$ occur in the graph, we have a conflict.
- Intuition:
  - representation of the dependencies between literals in μ
  - the graph contains $l_1 \xmapsto{c} l, ..., l_n \xmapsto{c} l$ iff l has been obtained from $l_1, ..., l_n$ by unit propagation on c
  - a partition of the graph with all decision literals on one side and the conflict on the other represents a conflict set

Example

$c_1 : \neg A_1 \vee A_2$
$c_2 : \neg A_1 \vee A_3 \vee A_9$
$c_3 : \neg A_2 \vee \neg A_3 \vee A_4$
$c_4 : \neg A_4 \vee A_5 \vee A_{10}$
$c_5 : \neg A_4 \vee A_6 \vee A_{11}$
$c_6 : \neg A_5 \vee \neg A_6$
$c_7 : A_1 \vee A_7 \vee \neg A_{12}$
$c_8 : A_1 \vee A_8$
$c_9 : \neg A_7 \vee \neg A_8 \vee \neg A_{13}$
...

Example

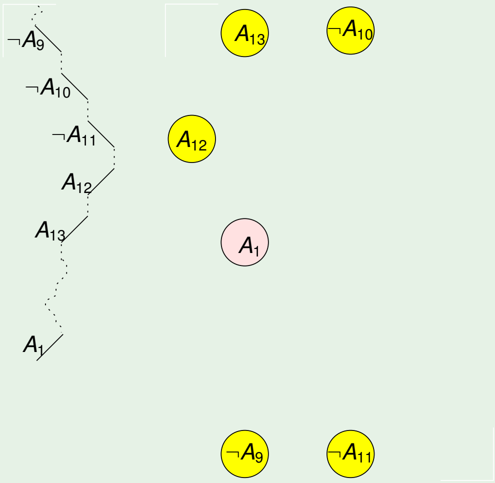
$c_1 : \neg A_1 \vee A_2$
$c_2 : \neg A_1 \vee A_3 \vee A_9$
$c_3 : \neg A_2 \vee \neg A_3 \vee A_4$
$c_4 : \neg A_4 \vee A_5 \vee A_{10}$
$c_5 : \neg A_4 \vee A_6 \vee A_{11}$
$c_6 : \neg A_5 \vee \neg A_6$
$c_7 : A_1 \vee A_7 \vee \neg A_{12}$
$c_8 : A_1 \vee A_8$
$c_9 : \neg A_7 \vee \neg A_8 \vee \neg A_{13}$
...



$\{..., \neg A_9, ..., \neg A_{10}, ..., \neg A_{11}, ..., A_{12}, ..., A_{13}, ...\}$
(Initial assignment. Note: $c_1, ..., c_9$ inconsistent.)

Example

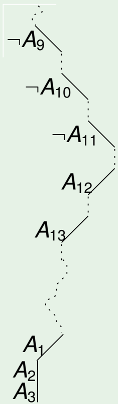
$c_1 : \neg A_1 \vee A_2$
$c_2 : \neg A_1 \vee A_3 \vee A_9$
$c_3 : \neg A_2 \vee \neg A_3 \vee A_4$
$c_4 : \neg A_4 \vee A_5 \vee A_{10}$
$c_5 : \neg A_4 \vee A_6 \vee A_{11}$
$c_6 : \neg A_5 \vee \neg A_6$
$c_7 : A_1 \vee A_7 \vee \neg A_{12}$ $\surd$
$c_8 : A_1 \vee A_8$ $\surd$
$c_9 : \neg A_7 \vee \neg A_8 \vee \neg A_{13}$
$\ldots$

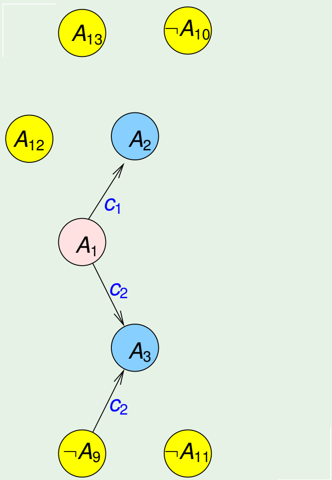


$\{\ldots, \neg A_9, \ldots, \neg A_{10}, \ldots, \neg A_{11}, \ldots, A_{12}, \ldots, A_{13}, \ldots, A_1\}$
... (decide A_1)

Example

$c_1 : \neg A_1 \vee A_2$ ✓
$c_2 : \neg A_1 \vee A_3 \vee A_9$ ✓
$c_3 : \neg A_2 \vee \neg A_3 \vee A_4$
$c_4 : \neg A_4 \vee A_5 \vee A_{10}$
$c_5 : \neg A_4 \vee A_6 \vee A_{11}$
$c_6 : \neg A_5 \vee \neg A_6$
$c_7 : A_1 \vee A_7 \vee \neg A_{12}$ ✓
$c_8 : A_1 \vee A_8$ ✓
$c_9 : \neg A_7 \vee \neg A_8 \vee \neg A_{13}$
...

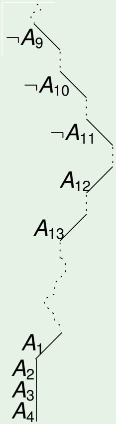


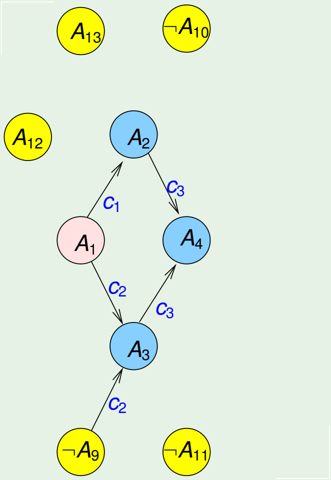


$\{..., \neg A_9, ..., \neg A_{10}, ..., \neg A_{11}, ..., A_{12}, ..., A_{13}, ..., A_1, A_2, A_3\}$
(unit A_2, A_3)

Example

c_1 : $\neg A_1 \vee A_2$ $\surd$
c_2 : $\neg A_1 \vee A_3 \vee A_9$ $\surd$
c_3 : $\neg A_2 \vee \neg A_3 \vee A_4$ $\surd$
c_4 : $\neg A_4 \vee A_5 \vee A_{10}$
c_5 : $\neg A_4 \vee A_6 \vee A_{11}$
c_6 : $\neg A_5 \vee \neg A_6$
c_7 : $A_1 \vee A_7 \vee \neg A_{12}$ $\surd$
c_8 : $A_1 \vee A_8$ $\surd$
c_9 : $\neg A_7 \vee \neg A_8 \vee \neg A_{13}$
...





$\{..., \neg A_9, ..., \neg A_{10}, ..., \neg A_{11}, ..., A_{12}, ..., A_{13}, ..., A_1, A_2, A_3, A_4\}$
(unit A_4)

Example

$c_1 : \neg A_1 \lor A_2$ $\surd$

$c_2 : \neg A_1 \lor A_3 \lor A_9$ $\surd$

$c_3 : \neg A_2 \lor \neg A_3 \lor A_4$ $\surd$

$c_4 : \neg A_4 \lor A_5 \lor A_{10}$ $\surd$

$c_5 : \neg A_4 \lor A_6 \lor A_{11}$ $\surd$

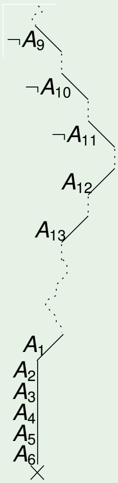
$c_6 : \neg A_5 \lor \neg A_6$ $\times$

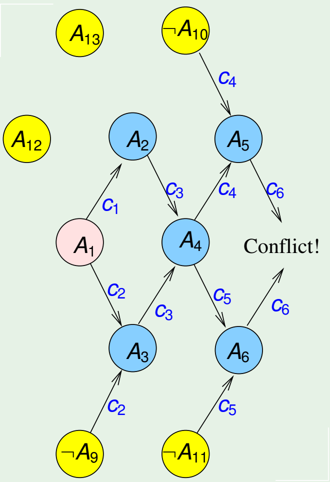
$c_7 : A_1 \lor A_7 \lor \neg A_{12}$ $\surd$

$c_8 : A_1 \lor A_8$ $\surd$

$c_9 : \neg A_7 \lor \neg A_8 \lor \neg A_{13}$

$\dots$





$\{\dots, \neg A_9, \dots, \neg A_{10}, \dots, \neg A_{11}, \dots, A_{12}, \dots, A_{13}, \dots, A_1, A_2, A_3, A_4, A_5, A_6\}$

(unit A_5, A_6) $\Longrightarrow$ conflict

Unique implication point - UIP [44]

- A node l in an implication graph is an unique implication point (UIP) for the last decision level iff every path from the last decision node to both the conflict nodes passes through l.
  - the most recent decision node is an UIP (last UIP)
  - all other UIP's have been assigned after the most recent decision
- Intuition: if the last decision had beed one of the UIPs instead of the last decision (last UIP), then we would have had the same failure on the same falsified clause.
  $\Longrightarrow$ We can pretend the UIP was the last decision.

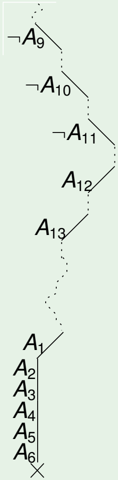
Unique implication point - UIP [44]

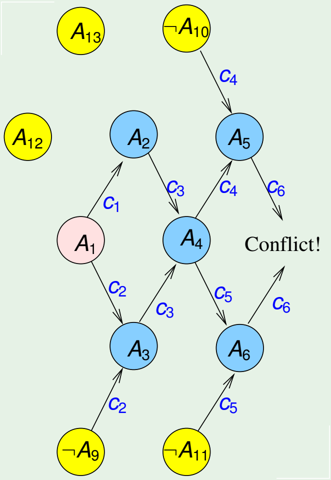
- A node I in an implication graph is an unique implication point (UIP) for the last decision level iff every path from the last decision node to both the conflict nodes passes through I.
  - the most recent decision node is an UIP (last UIP)
  - all other UIP's have been assigned after the most recent decision
- Intuition: if the last decision had beed one of the UIPs instead of the last decision (last UIP), then we would have had the same failure on the same falsified clause.
  $\Longrightarrow$ We can pretend the UIP was the last decision.

Unique implication point - UIP - example

$c_1 : \neg A_1 \vee A_2$ $\surd$
$c_2 : \neg A_1 \vee A_3 \vee A_9$ $\surd$
$c_3 : \neg A_2 \vee \neg A_3 \vee A_4$ $\surd$
$c_4 : \neg A_4 \vee A_5 \vee A_{10}$ $\surd$
$c_5 : \neg A_4 \vee A_6 \vee A_{11}$ $\surd$
$c_6 : \neg A_5 \vee \neg A_6$ $\times$
$c_7 : A_1 \vee A_7 \vee \neg A_{12}$ $\surd$
$c_8 : A_1 \vee A_8$ $\surd$
$c_9 : \neg A_7 \vee \neg A_8 \vee \neg A_{13}$
$\ldots$

- A_1 is the last UIP
- A_4 is the 1^{st} UIP





Schema of a CDCL DPLL solver [38, 45]

```
Function CDCL-SAT (formula: φ, assignment & μ) {
        status := preprocess(φ,μ);
        while (1) {
            while (1) {
                status := deduce(φ,μ);
                if (status == Sat)
                    return Sat;
                if (status == Conflict) {
                    ⟨blevel,η⟩ := analyze_conflict(φ,μ);
                    //η is a conflict set
                    if (blevel == 0)
                        return Unsat;
                    else backtrack(blevel,φ,μ);
                }
                else break;
            }
            decide_next_branch(φ,μ);
}       }
```

Schema of a CDCL DPLL solver [38, 45] (cont.)

- `preprocess(`φ, μ`)` simplifies φ into an easier equisatisfiable formula, updating μ.
- `decide_next_branch(`φ, μ`)` chooses a new decision literal from φ according to some heuristic, and adds it to μ
- `deduce(`φ, μ`)` performs all deterministic assignments (unit-propagations plus others), and updates φ, μ accordingly.
- `analyze_conflict(`φ, μ`)` Computes the subset η of μ causing the conflict (conflict set), and returns the "wrong-decision" level suggested by η
  - η is s.t. if we had decided all literals in η, we would have obtaned the same failure on the same falsified clause by unit-propagation
  - "`blevel==0`" means that η is entirely assigned at level 0, i.e., a conflict exists even without branching
- `backtrack(blevel,`φ, μ`)` undoes the branches up to blevel, and updates φ, μ accordingly

Backjumping and learning: general ideas [2, 38]

- When a branch μ fails:
  (i) conflict analysis: reveal the sub-assignment $\eta \subseteq \mu$ causing the failure (conflict set η)
  (ii) learning: add the conflict clause $C \stackrel{\text{def}}{=} \neg\eta$ to the clause set
  (iii) backjumping: use η to decide the point where to backtrack
- Jump back up much more than one decision level in the stack
  $\Longrightarrow$ may avoid lots of redundant search!!.
- We illustrate two main backjumping & learning strategies:
  - the original strategy presented in [38]
  - the state-of-the-art 1^{st}UIP strategy of [44]

Conflict analysis

1. $C :=$ falsified clause (conflicting clause)
2. repeat
   (i) resolve the current clause C with the antecedent clause of the last unit-propagated literal l in C

   until C verifies some given termination criteria

Conflict analysis

1. $C :=$ falsified clause (conflicting clause)
2. repeat
   (i) resolve the current clause C with the antecedent clause of the last unit-propagated literal l in C

   until C verifies some given termination criteria

criterion: decision

...until C contains only decision literals

$$
\dfrac{\neg A_1 \vee A_2 \qquad \dfrac{\neg A_1 \vee A_3 \vee A_9 \qquad \dfrac{\neg A_2 \vee \neg A_3 \vee A_4 \qquad \dfrac{\neg A_4 \vee A_5 \vee A_{10} \qquad \dfrac{\neg A_4 \vee A_6 \vee A_{11} \qquad \overbrace{\neg A_5 \vee \neg A_6}^{\textit{Conflicting cl.}}}{\neg A_4 \vee \neg A_5 \vee A_{11}}\ (A_6)}{\neg A_4 \vee A_{10} \vee A_{11}}\ (A_5)}{\neg A_2 \vee \neg A_3 \vee A_{10} \vee A_{11}}\ (A_4)}{\neg A_2 \vee \neg A_1 \vee A_9 \vee A_{10} \vee A_{11}}\ (A_3)}{\neg A_1 \vee A_9 \vee A_{10} \vee A_{11}}\ (A_2)
$$

Conflict analysis

1. $C :=$ falsified clause (conflicting clause)
2. repeat
   (i) resolve the current clause C with the antecedent clause of the last unit-propagated literal l in C

   until C verifies some given termination criteria

criterion: last UIP

... until C contains only one literal assigned at current decision level, and it is the decision literal (last UIP)

$$\dfrac{\neg A_1 \lor A_2 \qquad \dfrac{\neg A_1 \lor A_3 \lor A_9 \qquad \dfrac{\neg A_2 \lor \neg A_3 \lor A_4 \qquad \dfrac{\neg A_4 \lor A_5 \lor A_{10} \qquad \dfrac{\neg A_4 \lor A_6 \lor A_{11} \quad \overbrace{\neg A_5 \lor \neg A_6}^{\text{Conflicting } cl.}}{\neg A_4 \lor \neg A_5 \lor A_{11}}\,(A_6)}{\neg A_4 \lor A_{10} \lor A_{11}}\,(A_5)}{\neg A_2 \lor \neg A_3 \lor A_{10} \lor A_{11}}\,(A_4)}{\neg A_2 \lor \neg A_1 \lor A_9 \lor A_{10} \lor A_{11}}\,(A_3)}{\neg A_1 \lor A_9 \lor A_{10} \lor A_{11}}\,(A_2)$$

Conflict analysis

1. $C :=$ falsified clause (conflicting clause)
2. repeat
   (i) resolve the current clause C with the antecedent clause of the last unit-propagated literal l in C

   until C verifies some given termination criteria

criterion: 1st UIP

... until C contains only one literal assigned at current decision level (1st UIP)

$$\dfrac{\neg A_4 \lor A_5 \lor A_{10} \qquad \dfrac{\neg A_4 \lor A_6 \lor A_{11} \qquad \overbrace{\neg A_5 \lor \neg A_6}^{Conflicting\ \ cl.}}{\neg A_4 \lor \neg A_5 \lor A_{11}}\ (A_6)}{\underbrace{\neg A_4}_{1st\ \ UIP} \lor A_{10} \lor A_{11}}\ (A_5)$$

Conflict analysis

1. $C :=$ falsified clause (conflicting clause)
2. repeat
   (i) resolve the current clause C with the antecedent clause of the last unit-propagated literal l in C

   until C verifies some given termination criteria

Note:

$\varphi \models C$, so that C can be safely added to φ.

Note:

Equivalent to finding a partition in the implication graph of μ with all decision literals on one side and the conflict on the other.

Conflict analysis

1. $C :=$ falsified clause (conflicting clause)
2. repeat
   (i) resolve the current clause C with the antecedent clause of the last unit-propagated literal l in C

   until C verifies some given termination criteria

Note:

$\varphi \models C$, so that C can be safely added to φ.

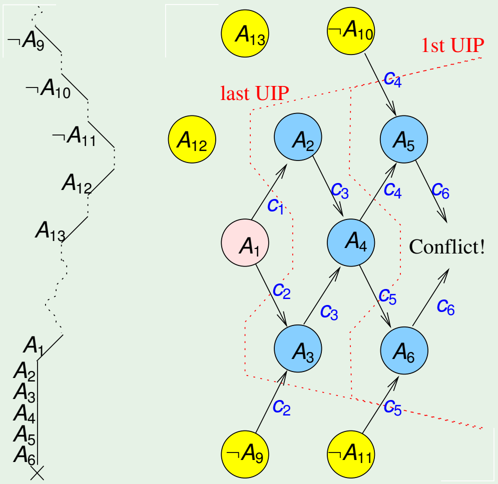
Note:

Equivalent to finding a partition in the implication graph of μ with all decision literals on one side and the conflict on the other.

Conflict analysis and implication graph - example

$c_1 : \neg A_1 \lor A_2$ $\surd$
$c_2 : \neg A_1 \lor A_3 \lor A_9$ $\surd$
$c_3 : \neg A_2 \lor \neg A_3 \lor A_4$ $\surd$
$c_4 : \neg A_4 \lor A_5 \lor A_{10}$ $\surd$
$c_5 : \neg A_4 \lor A_6 \lor A_{11}$ $\surd$
$c_6 : \neg A_5 \lor \neg A_6$ $\times$
$c_7 : A_1 \lor A_7 \lor \neg A_{12}$ $\surd$
$c_8 : A_1 \lor A_8$ $\surd$
$c_9 : \neg A_7 \lor \neg A_8 \lor \neg A_{13}$
...

Note: in this case decision and last-UIP criteria produce the same partition

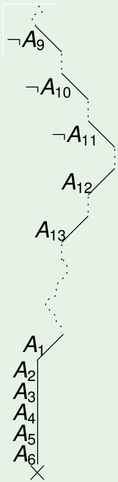


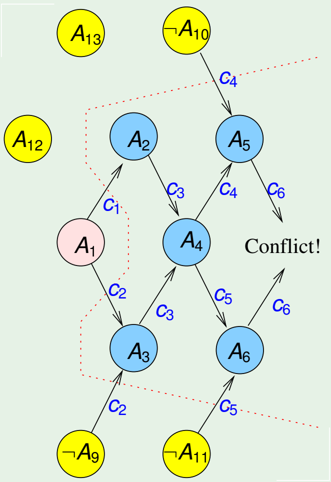
The original backjumping and learning strategy of [38]

- Idea: when a branch μ fails,
  (i) conflict analysis: find the conflict set $\eta \subseteq \mu$ by generating the conflict clause $C \stackrel{\text{def}}{=} \neg\eta$ via resolution from the falsified clause (conflicting clause) using the "Decision" criterion;
  (ii) learning: add the conflict clause C to the clause set
  (iii) backjumping: backtrack to the most recent branching point s.t. the stack does not fully contain η, and then unit-propagate the unassigned literal on C

The Original Backjumping Strategy: Example

$c_1 : \neg A_1 \lor A_2$ $\surd$
$c_2 : \neg A_1 \lor A_3 \lor A_9$ $\surd$
$c_3 : \neg A_2 \lor \neg A_3 \lor A_4$ $\surd$
$c_4 : \neg A_4 \lor A_5 \lor A_{10}$ $\surd$
$c_5 : \neg A_4 \lor A_6 \lor A_{11}$ $\surd$
$c_6 : \neg A_5 \lor \neg A_6$ $\times$
$c_7 : A_1 \lor A_7 \lor \neg A_{12}$ $\surd$
$c_8 : A_1 \lor A_8$ $\surd$
$c_9 : \neg A_7 \lor \neg A_8 \lor \neg A_{13}$
...

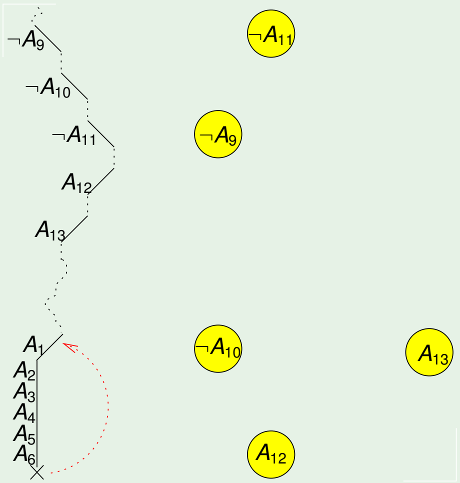




$\Longrightarrow$ Conflict set: $\{\neg A_9, \neg A_{10}, \neg A_{11}, A_1\}$ ("decision" schema)
$\Longrightarrow$ learn the conflict clause $c_{10} := A_9 \lor A_{10} \lor A_{11} \lor \neg A_1$

The Original Backjumping Strategy: Example

$c_1 : \neg A_1 \vee A_2$
$c_2 : \neg A_1 \vee A_3 \vee A_9$
$c_3 : \neg A_2 \vee \neg A_3 \vee A_4$
$c_4 : \neg A_4 \vee A_5 \vee A_{10}$
$c_5 : \neg A_4 \vee A_6 \vee A_{11}$
$c_6 : \neg A_5 \vee \neg A_6$
$c_7 : A_1 \vee A_7 \vee \neg A_{12}$
$c_8 : A_1 \vee A_8$
$c_9 : \neg A_7 \vee \neg A_8 \vee \neg A_{13}$
$c_{10} : A_9 \vee A_{10} \vee A_{11} \vee \neg A_1$
...



$\{..., \neg A_9, ..., \neg A_{10}, ..., \neg A_{11}, ..., A_{12}, ..., A_{13}, ...\}$
$\Longrightarrow$ backtrack up to A_1

The Original Backjumping Strategy: Example

$c_1 : \neg A_1 \vee A_2$ $\surd$

$c_2 : \neg A_1 \vee A_3 \vee A_9$ $\surd$

$c_3 : \neg A_2 \vee \neg A_3 \vee A_4$

$c_4 : \neg A_4 \vee A_5 \vee A_{10}$

$c_5 : \neg A_4 \vee A_6 \vee A_{11}$

$c_6 : \neg A_5 \vee \neg A_6$

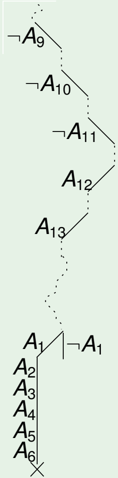
$c_7 : A_1 \vee A_7 \vee \neg A_{12}$

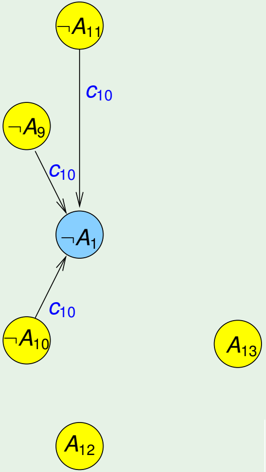
$c_8 : A_1 \vee A_8$

$c_9 : \neg A_7 \vee \neg A_8 \vee \neg A_{13}$

$c_{10} : A_9 \vee A_{10} \vee A_{11} \vee \neg A_1$ $\surd$

...



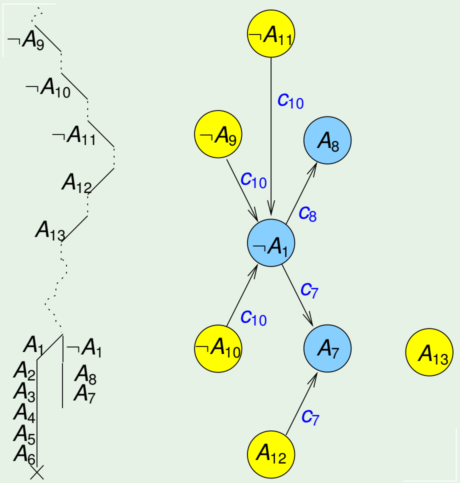


$\{..., \neg A_9, ..., \neg A_{10}, ..., \neg A_{11}, ..., A_{12}, ..., A_{13}, ..., \neg A_1\}$

(unit $\neg A_1$)

The Original Backjumping Strategy: Example

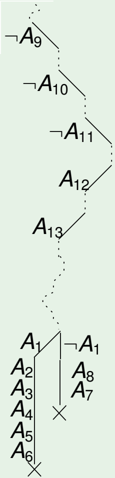
$c_1 : \neg A_1 \vee A_2$ $\surd$
$c_2 : \neg A_1 \vee A_3 \vee A_9$ $\surd$
$c_3 : \neg A_2 \vee \neg A_3 \vee A_4$
$c_4 : \neg A_4 \vee A_5 \vee A_{10}$
$c_5 : \neg A_4 \vee A_6 \vee A_{11}$
$c_6 : \neg A_5 \vee \neg A_6$
$c_7 : A_1 \vee A_7 \vee \neg A_{12}$ $\surd$
$c_8 : A_1 \vee A_8$ $\surd$
$c_9 : \neg A_7 \vee \neg A_8 \vee \neg A_{13}$
$c_{10} : A_9 \vee A_{10} \vee A_{11} \vee \neg A_1$ $\surd$
...

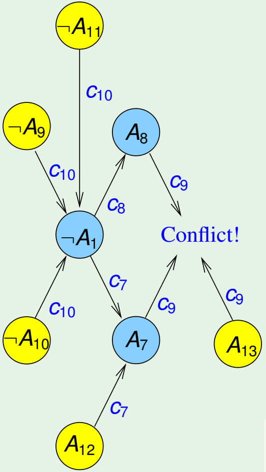


$\{..., \neg A_9, ..., \neg A_{10}, ..., \neg A_{11}, ..., A_{12}, ..., A_{13}, ..., \neg A_1, A_7, A_8\}$
(unit A_7, A_8)

The Original Backjumping Strategy: Example

$c_1 : \neg A_1 \vee A_2$ $\surd$
$c_2 : \neg A_1 \vee A_3 \vee A_9$ $\surd$
$c_3 : \neg A_2 \vee \neg A_3 \vee A_4$
$c_4 : \neg A_4 \vee A_5 \vee A_{10}$
$c_5 : \neg A_4 \vee A_6 \vee A_{11}$
$c_6 : \neg A_5 \vee \neg A_6$
$c_7 : A_1 \vee A_7 \vee \neg A_{12}$ $\surd$
$c_8 : A_1 \vee A_8$ $\surd$
$c_9 : \neg A_7 \vee \neg A_8 \vee \neg A_{13}$ $\times$
$c_{10} : A_9 \vee A_{10} \vee A_{11} \vee \neg A_1$ $\surd$
...





$\{\ldots, \neg A_9, \ldots, \neg A_{10}, \ldots, \neg A_{11}, \ldots, A_{12}, \ldots, A_{13}, \ldots, \neg A_1, A_7, A_8\}$
Conflict!

The Original Backjumping Strategy: Example

$c_1 : \neg A_1 \vee A_2$ √

$c_2 : \neg A_1 \vee A_3 \vee A_9$ √

$c_3 : \neg A_2 \vee \neg A_3 \vee A_4$

$c_4 : \neg A_4 \vee A_5 \vee A_{10}$

$c_5 : \neg A_4 \vee A_6 \vee A_{11}$

$c_6 : \neg A_5 \vee \neg A_6$

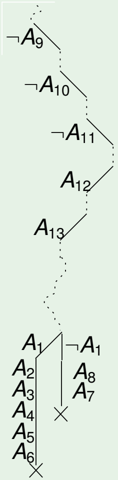
$c_7 : A_1 \vee A_7 \vee \neg A_{12}$ √

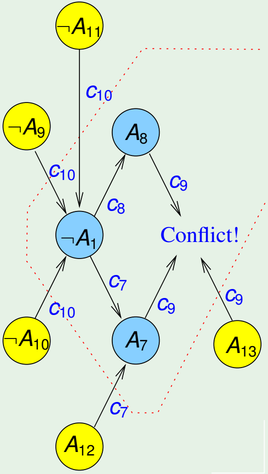
$c_8 : A_1 \vee A_8$ √

$c_9 : \neg A_7 \vee \neg A_8 \vee \neg A_{13}$ ×

$c_{10} : A_9 \vee A_{10} \vee A_{11} \vee \neg A_1$ √

...



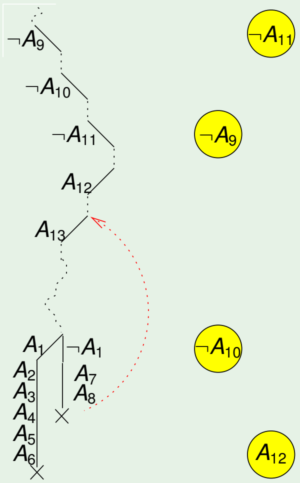


$\Longrightarrow$ conflict set: $\{\neg A_9, \neg A_{10}, \neg A_{11}, A_{12}, A_{13}\}$.

$\Longrightarrow$ learn $C_{11} := A_9 \vee A_{10} \vee A_{11} \vee \neg A_{12} \vee \neg A_{13}$

The Original Backjumping Strategy: Example

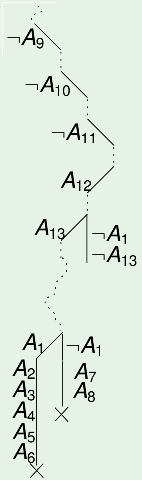
$c_1 : \neg A_1 \vee A_2$
$c_2 : \neg A_1 \vee A_3 \vee A_9$
$c_3 : \neg A_2 \vee \neg A_3 \vee A_4$
$c_4 : \neg A_4 \vee A_5 \vee A_{10}$
$c_5 : \neg A_4 \vee A_6 \vee A_{11}$
$c_6 : \neg A_5 \vee \neg A_6$
$c_7 : A_1 \vee A_7 \vee \neg A_{12}$
$c_8 : A_1 \vee A_8$
$c_9 : \neg A_7 \vee \neg A_8 \vee \neg A_{13}$
$c_{10} : A_9 \vee A_{10} \vee A_{11} \vee \neg A_1$
$c_{11} : A_9 \vee A_{10} \vee A_{11} \vee \neg A_{12} \vee \neg A_{13}$
...

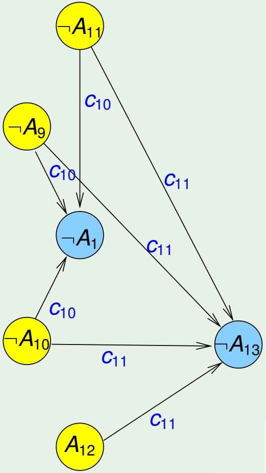


$\Longrightarrow$ backtrack to A_{13} $\Longrightarrow$ Lots of search saved!

The Original Backjumping Strategy: Example

$c_1 : \neg A_1 \vee A_2$ $\surd$
$c_2 : \neg A_1 \vee A_3 \vee A_9$ $\surd$
$c_3 : \neg A_2 \vee \neg A_3 \vee A_4$
$c_4 : \neg A_4 \vee A_5 \vee A_{10}$
$c_5 : \neg A_4 \vee A_6 \vee A_{11}$
$c_6 : \neg A_5 \vee \neg A_6$
$c_7 : A_1 \vee A_7 \vee \neg A_{12}$
$c_8 : A_1 \vee A_8$
$c_9 : \neg A_7 \vee \neg A_8 \vee \neg A_{13}$ $\surd$
$c_{10} : A_9 \vee A_{10} \vee A_{11} \vee \neg A_1$ $\surd$
$c_{11} : A_9 \vee A_{10} \vee A_{11} \vee \neg A_{12} \vee \neg A_{13}$ $\surd$
...





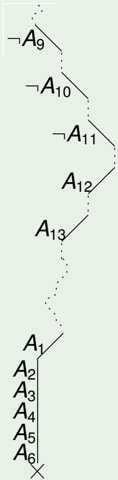
$\Longrightarrow$ backtrack to A_{13}, then set A_{13} and A_1 to $\perp$,...

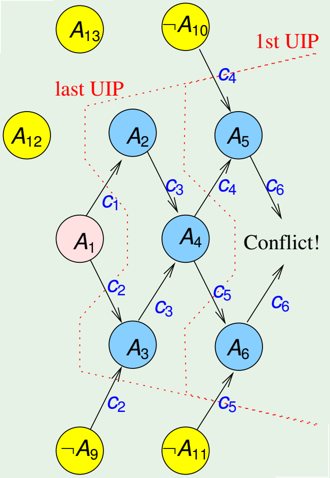
State-of-the-art backjumping and learning [44]

- Idea: when a branch μ fails,
  (i) conflict analysis: find the conflict set $\eta \subseteq \mu$ by generating the conflict clause $C \stackrel{\text{def}}{=} \neg\eta$ via resolution from the falsified clause, according to the 1^{st}UIP strategy
  (ii) learning: add the conflict clause C to the clause set
  (iii) backjumping: backtrack to the highest branching point s.t. the stack contains all-but-one literals in η, and then unit-propagate the unassigned literal on C

1st UIP strategy – example (7)

$c_1 : \neg A_1 \vee A_2$ $\surd$
$c_2 : \neg A_1 \vee A_3 \vee A_9$ $\surd$
$c_3 : \neg A_2 \vee \neg A_3 \vee A_4$ $\surd$
$c_4 : \neg A_4 \vee A_5 \vee A_{10}$ $\surd$
$c_5 : \neg A_4 \vee A_6 \vee A_{11}$ $\surd$
$c_6 : \neg A_5 \vee \neg A_6$ $\times$
$c_7 : A_1 \vee A_7 \vee \neg A_{12}$ $\surd$
$c_8 : A_1 \vee A_8$ $\surd$
$c_9 : \neg A_7 \vee \neg A_8 \vee \neg A_{13}$
...





$\Longrightarrow$ Conflict set: $\{\neg A_{10}, \neg A_{11}, A_4\}$, learn $c_{10} := A_{10} \vee A_{11} \vee \neg A_4$

1st UIP strategy and backjumping [44]

- The added conflict clause states the reason for the conflict
- The procedure backtracks to the most recent decision level of the variables in the conflict clause which are not the UIP.
- then the conflict clause forces the negation of the UIP by unit propagation.

E.g.: $c_{10} := A_{10} \vee A_{11} \vee \neg A_4$
$\Longrightarrow$ backtrack to A_{11}, then assign $\neg A_4$

1st UIP strategy – example (7)

c_1 : $\neg A_1 \lor A_2$ ✓

c_2 : $\neg A_1 \lor A_3 \lor A_9$ ✓

c_3 : $\neg A_2 \lor \neg A_3 \lor A_4$ ✓

c_4 : $\neg A_4 \lor A_5 \lor A_{10}$ ✓

c_5 : $\neg A_4 \lor A_6 \lor A_{11}$ ✓

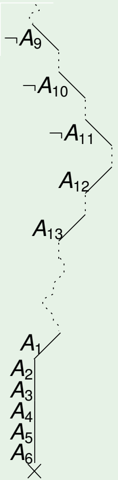
c_6 : $\neg A_5 \lor \neg A_6$ ×

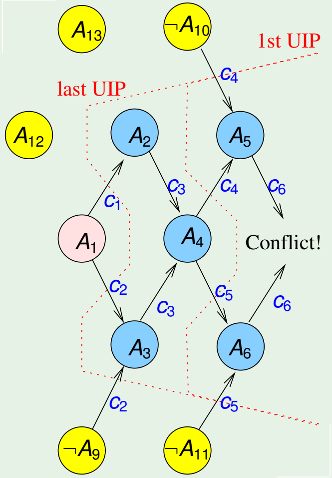
c_7 : $A_1 \lor A_7 \lor \neg A_{12}$ ✓

c_8 : $A_1 \lor A_8$ ✓

c_9 : $\neg A_7 \lor \neg A_8 \lor \neg A_{13}$

…

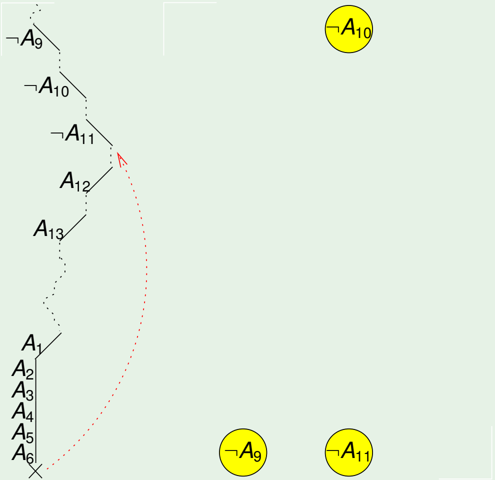




$\Longrightarrow$ Conflict set: $\{\neg A_{10}, \neg A_{11}, A_4\}$, learn $c_{10} := A_{10} \lor A_{11} \lor \neg A_4$

1st UIP strategy – example (8)

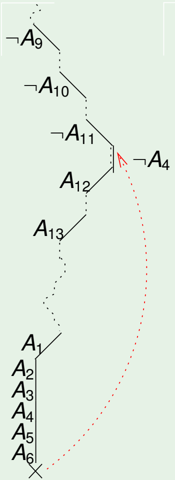
$c_1 : \neg A_1 \lor A_2$
$c_2 : \neg A_1 \lor A_3 \lor A_9$
$c_3 : \neg A_2 \lor \neg A_3 \lor A_4$
$c_4 : \neg A_4 \lor A_5 \lor A_{10}$
$c_5 : \neg A_4 \lor A_6 \lor A_{11}$
$c_6 : \neg A_5 \lor \neg A_6$
$c_7 : A_1 \lor A_7 \lor \neg A_{12}$
$c_8 : A_1 \lor A_8$
$c_9 : \neg A_7 \lor \neg A_8 \lor \neg A_{13}$
$c_{10} : A_{10} \lor A_{11} \lor \neg A_4$
...

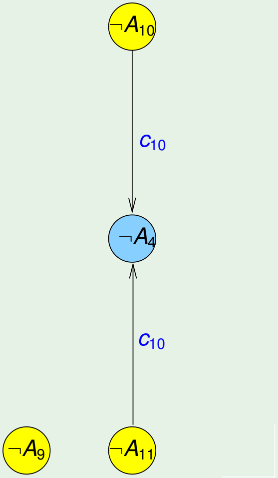


$\Longrightarrow$ backtrack up to A_{11} $\Longrightarrow \{..., \neg A_9, ..., \neg A_{10}, ..., \neg A_{11}\}$

1st UIP strategy – example (9)

$c_1 : \neg A_1 \vee A_2$
$c_2 : \neg A_1 \vee A_3 \vee A_9$
$c_3 : \neg A_2 \vee \neg A_3 \vee A_4$
$c_4 : \neg A_4 \vee A_5 \vee A_{10}$ $\surd$
$c_5 : \neg A_4 \vee A_6 \vee A_{11}$ $\surd$
$c_6 : \neg A_5 \vee \neg A_6$
$c_7 : A_1 \vee A_7 \vee \neg A_{12}$
$c_8 : A_1 \vee A_8$
$c_9 : \neg A_7 \vee \neg A_8 \vee \neg A_{13}$
$c_{10} : A_{10} \vee A_{11} \vee \neg A_4$ $\surd$
...





$\Longrightarrow$ unit propagate $\neg A_4 \Longrightarrow \{..., \neg A_9, ..., \neg A_{10}, ..., \neg A_{11}, A_4\}...$

1st UIP strategy and backjumping – intuition

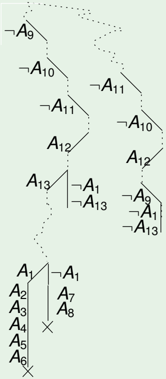
- An UIP is a single reason implying the conflict at the current level
- substituting the 1st UIP for the last UIP
  - does not enlarge the conflict
  - requires less resolution steps to compute C
  - may require involving less decision literals from other levels
- by backtracking to the most recent decision level of the variables in the conflict clause which are not the UIP:
  - jump higher
  - allows for assigning (the negation of) the UIP as high as possible in the search tree.

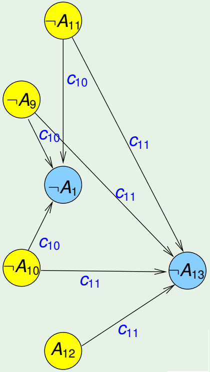
Learning [2, 38]

Idea: When a conflict set η is revealed, then $C \stackrel{\text{def}}{=} \neg\eta$ is added to φ
$\Longrightarrow$ the solver will no more generate an assignment containing η:
when $|\eta| - 1$ literals in η are assigned, the other is set $\perp$ by unit-propagation on C
$\Longrightarrow$ Drastic pruning of the search!

Learning – example

$c_1 : \neg A_1 \vee A_2$
$c_2 : \neg A_1 \vee A_3 \vee A_9$
$c_3 : \neg A_2 \vee \neg A_3 \vee A_4$
$c_4 : \neg A_4 \vee A_5 \vee A_{10}$
$c_5 : \neg A_4 \vee A_6 \vee A_{11}$
$c_6 : \neg A_5 \vee \neg A_6$
$c_7 : A_1 \vee A_7 \vee \neg A_{12}$
$c_8 : A_1 \vee A_8$
$c_9 : \neg A_7 \vee \neg A_8 \vee \neg A_{13}$ $\surd$
$c_{10} : A_9 \vee A_{10} \vee A_{11} \vee \neg A_1$ $\surd$
$c_{11} : A_9 \vee A_{10} \vee A_{11} \vee \neg A_{12} \vee \neg A_{13}$ $\surd$
...
$\Longrightarrow$ Unit: $\{\neg A_1, \neg A_{13}\}$





Drawbacks of Learning & Clause discharging

Problem with Learning

Learning can generate exponentially-many clauses

- may cause a blowup in space
- may drastically slow down BCP

A solution: clause discharging

Techniques to drop learned clauses when necessary

- according to their size
- according to their activity.

A clause is currently active if it occurs in the current implication graph (i.e., it is the antecedent clause of a literal in the current assignment).

Drawbacks of Learning & Clause discharging

Problem with Learning

Learning can generate exponentially-many clauses

- may cause a blowup in space
- may drastically slow down BCP

A solution: clause discharging

Techniques to drop learned clauses when necessary

- according to their size
- according to their activity.

A clause is currently active if it occurs in the current implication graph (i.e., it is the antecedent clause of a literal in the current assignment).

Drawbacks of Learning & Clause discharging

- Is clause-discharging safe?
- Yes, if done properly.

Property (see, e.g., [30])

In order to guarantee correctness, completeness & termination of a CDCL solver, it suffices to keep each clause until it is active.
$\Longrightarrow$ CDCL solvers require polynomial space

"Lazy" Strategy

- when a clause is involved in conflict analisis, increase its activity
- when needed, drop the least-active clauses

Drawbacks of Learning & Clause discharging

- Is clause-discharging safe?
- Yes, if done properly.

Property (see, e.g., [30])

In order to guarantee correctness, completeness & termination of a CDCL solver, it suffices to keep each clause until it is active.
$\Longrightarrow$ CDCL solvers require polynomial space

"Lazy" Strategy

- when a clause is involved in conflict analisis, increase its activity
- when needed, drop the least-active clauses

Drawbacks of Learning & Clause discharging

- Is clause-discharging safe?
- Yes, if done properly.

Property (see, e.g., [30])

In order to guarantee correctness, completeness & termination of a CDCL solver, it suffices to keep each clause until it is active.
$\Longrightarrow$ CDCL solvers require polynomial space

"Lazy" Strategy

- when a clause is involved in conflict analisis, increase its activity
- when needed, drop the least-active clauses

State-of-the-art backjumping and learning: intuitions

- Backjumping: allows for climbing up to many decision levels in the stack
  - intuition: " go back to the oldest decision where you'd have done something different if only you had known C"

  $\Longrightarrow$ may avoid lots of redundant search
- Learning: in future branches, when all-but-one literals in η are assigned, the remaining literal is assigned to false by unit-propagation:
  - intuition: "when you're about to repeat the mistake, do the opposite of the last step"

  $\Longrightarrow$ avoid finding the same conflict again

State-of-the-art backjumping and learning: intuitions

- Backjumping: allows for climbing up to many decision levels in the stack
  - intuition: " go back to the oldest decision where you'd have done something different if only you had known *C*"

  $\Longrightarrow$ may avoid lots of redundant search
- Learning: in future branches, when all-but-one literals in η are assigned, the remaining literal is assigned to false by unit-propagation:
  - intuition: "when you're about to repeat the mistake, do the opposite of the last step"

  $\Longrightarrow$ avoid finding the same conflict again

State-of-the-art backjumping and learning: intuitions

- Backjumping: allows for climbing up to many decision levels in the stack
  - intuition: " go back to the oldest decision where you'd have done something different if only you had known C"

  $\Longrightarrow$ may avoid lots of redundant search
- Learning: in future branches, when all-but-one literals in η are assigned, the remaining literal is assigned to false by unit-propagation:
  - intuition: "when you're about to repeat the mistake, do the opposite of the last step"

  $\Longrightarrow$ avoid finding the same conflict again

State-of-the-art backjumping and learning: intuitions

- Backjumping: allows for climbing up to many decision levels in the stack
  - intuition: " go back to the oldest decision where you'd have done something different if only you had known C"

  $\Longrightarrow$ may avoid lots of redundant search
- Learning: in future branches, when all-but-one literals in η are assigned, the remaining literal is assigned to false by unit-propagation:
  - intuition: "when you're about to repeat the mistake, do the opposite of the last step"

  $\Longrightarrow$ avoid finding the same conflict again

State-of-the-art backjumping and learning: intuitions

- **Backjumping:** allows for climbing up to many decision levels in the stack
  - intuition: " go back to the oldest decision where you'd have done something different if only you had known C"

  $\Longrightarrow$ may avoid lots of redundant search
- **Learning:** in future branches, when all-but-one literals in η are assigned, the remaining literal is assigned to false by unit-propagation:
  - intuition: "when you're about to repeat the mistake, do the opposite of the last step"

  $\Longrightarrow$ avoid finding the same conflict again

State-of-the-art backjumping and learning: intuitions

- Backjumping: allows for climbing up to many decision levels in the stack
  - intuition: " go back to the oldest decision where you'd have done something different if only you had known C"

  $\Longrightarrow$ may avoid lots of redundant search
- Learning: in future branches, when all-but-one literals in η are assigned, the remaining literal is assigned to false by unit-propagation:
  - intuition: "when you're about to repeat the mistake, do the opposite of the last step"

  $\Longrightarrow$ avoid finding the same conflict again

Remark: the “quality” of conflict sets

- Different ideas of “good” conflict set
  - Backjumping: if causes the highest backjump (“local” role)
  - Learning: if causes the maximum pruning (“global” role)
- Many different strategies implemented (see, e.g., [2, 38, 44])

Outline

Preprocessing/Inprocessing

- Part of `preprocess()` and `deduce()` steps respectively
- Simplify current formula into an equivalently-satisfiable one
- Must be fast (in particular inprocessing)
- Some techniques:
  - detect and remove subsumed clauses
  - detect & collapse equivalent literals
  - apply basic resolution steps
  - ...

Preprocessing/Inprocessing (cont.)

Detect and remove subsumed clauses:

$$\varphi_1 \wedge (l_2 \vee l_1) \wedge \varphi_2 \wedge (l_2 \vee l_3 \vee l_1) \wedge \varphi_3$$
$$\Downarrow$$
$$\varphi_1 \wedge (l_1 \vee l_2) \wedge \varphi_2 \wedge \varphi_3$$

Preprocessing/Inprocessing (cont.)

Detect & collapse equivalent literals [7]

Repeat:

(i) build the implication graph induced by binary clauses

(ii) detect strongly connected cycles $\Longrightarrow$ equivalence classes of literals

(iii) perform substitutions

(iv) perform unit and pure literal.

Until (no more simplification is possible).

- Ex:

$$\varphi_1 \wedge (\neg l_2 \vee l_1) \wedge \varphi_2 \wedge (\neg l_3 \vee l_2) \wedge \varphi_3 \wedge (\neg l_1 \vee l_3) \wedge \varphi_4$$
$$\Downarrow_{l_1 \leftrightarrow l_2 \leftrightarrow l_3}$$
$$(\varphi_1 \wedge \varphi_2 \wedge \varphi_3 \wedge \varphi_4)[l_2 \leftarrow l_1; l_3 \leftarrow l_1;]$$

- Very effective in many application domains.

Preprocessing/Inprocessing (cont.)

Detect & collapse equivalent literals [7]

Repeat:

(i) build the implication graph induced by binary clauses

(ii) detect strongly connected cycles $\Longrightarrow$ equivalence classes of literals

(iii) perform substitutions

(iv) perform unit and pure literal.

Until (no more simplification is possible).

- Ex:

$$\varphi_1 \wedge (\neg l_2 \vee l_1) \wedge \varphi_2 \wedge (\neg l_3 \vee l_2) \wedge \varphi_3 \wedge (\neg l_1 \vee l_3) \wedge \varphi_4$$
$$\Downarrow_{l_1 \leftrightarrow l_2 \leftrightarrow l_3}$$
$$(\varphi_1 \wedge \varphi_2 \wedge \varphi_3 \wedge \varphi_4)[l_2 \leftarrow l_1; l_3 \leftarrow l_1;]$$

- Very effective in many application domains.

Preprocessing/Inprocessing (cont.)

Detect & collapse equivalent literals [7]

Repeat:

(i) build the implication graph induced by binary clauses

(ii) detect strongly connected cycles $\Longrightarrow$ equivalence classes of literals

(iii) perform substitutions

(iv) perform unit and pure literal.

Until (no more simplification is possible).

- Ex:

$$\varphi_1 \wedge (\neg l_2 \vee l_1) \wedge \varphi_2 \wedge (\neg l_3 \vee l_2) \wedge \varphi_3 \wedge (\neg l_1 \vee l_3) \wedge \varphi_4$$
$$\Downarrow_{l_1 \leftrightarrow l_2 \leftrightarrow l_3}$$
$$(\varphi_1 \wedge \varphi_2 \wedge \varphi_3 \wedge \varphi_4)[l_2 \leftarrow l_1; l_3 \leftarrow l_1;]$$

- Very effective in many application domains.

Preprocessing/Inprocessing (cont.)

Detect & collapse equivalent literals [7]

Repeat:

(i) build the implication graph induced by binary clauses
(ii) detect strongly connected cycles $\Longrightarrow$ equivalence classes of literals
(iii) perform substitutions
(iv) perform unit and pure literal.

Until (no more simplification is possible).

- Ex:

$$\varphi_1 \land (\neg l_2 \lor l_1) \land \varphi_2 \land (\neg l_3 \lor l_2) \land \varphi_3 \land (\neg l_1 \lor l_3) \land \varphi_4$$
$$\Downarrow_{l_1 \leftrightarrow l_2 \leftrightarrow l_3}$$
$$(\varphi_1 \land \varphi_2 \land \varphi_3 \land \varphi_4)[l_2 \leftarrow l_1; l_3 \leftarrow l_1;]$$

- Very effective in many application domains.

Preprocessing/Inprocessing (cont.)

Detect & collapse equivalent literals [7]

Repeat:

(i) build the implication graph induced by binary clauses

(ii) detect strongly connected cycles $\Longrightarrow$ equivalence classes of literals

(iii) perform substitutions

(iv) perform unit and pure literal.

Until (no more simplification is possible).

- Ex:

$$\varphi_1 \wedge (\neg l_2 \vee l_1) \wedge \varphi_2 \wedge (\neg l_3 \vee l_2) \wedge \varphi_3 \wedge (\neg l_1 \vee l_3) \wedge \varphi_4$$
$$\Downarrow_{l_1 \leftrightarrow l_2 \leftrightarrow l_3}$$
$$(\varphi_1 \wedge \varphi_2 \wedge \varphi_3 \wedge \varphi_4)[l_2 \leftarrow l_1; l_3 \leftarrow l_1;]$$

- Very effective in many application domains.

Preprocessing/Inprocessing (cont.)

Detect & collapse equivalent literals [7]

Repeat:

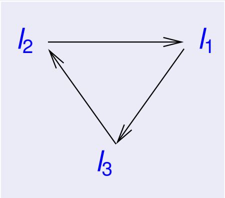
(i) build the implication graph induced by binary clauses

(ii) detect strongly connected cycles $\Longrightarrow$ equivalence classes of literals

(iii) perform substitutions

(iv) perform unit and pure literal.

Until (no more simplification is possible).



- Ex:

$$\varphi_1 \wedge (\neg l_2 \vee l_1) \wedge \varphi_2 \wedge (\neg l_3 \vee l_2) \wedge \varphi_3 \wedge (\neg l_1 \vee l_3) \wedge \varphi_4$$
$$\Downarrow_{l_1 \leftrightarrow l_2 \leftrightarrow l_3}$$
$$(\varphi_1 \wedge \varphi_2 \wedge \varphi_3 \wedge \varphi_4)[l_2 \leftarrow l_1; l_3 \leftarrow l_1;]$$

- Very effective in many application domains.

Preprocessing/Inprocessing (cont.)

Detect & collapse equivalent literals [7]

Repeat:

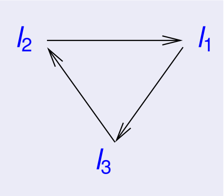
(i) build the implication graph induced by binary clauses

(ii) detect strongly connected cycles $\Longrightarrow$ equivalence classes of literals

(iii) perform substitutions

(iv) perform unit and pure literal.

Until (no more simplification is possible).



- Ex:

$$\varphi_1 \wedge (\neg l_2 \vee l_1) \wedge \varphi_2 \wedge (\neg l_3 \vee l_2) \wedge \varphi_3 \wedge (\neg l_1 \vee l_3) \wedge \varphi_4$$
$$\Downarrow_{l_1 \leftrightarrow l_2 \leftrightarrow l_3}$$
$$(\varphi_1 \wedge \varphi_2 \wedge \varphi_3 \wedge \varphi_4)[l_2 \leftarrow l_1; l_3 \leftarrow l_1;]$$

- Very effective in many application domains.

Preprocessing/Inprocessing (cont.)

Apply some basic steps of resolution (and simplify)

$$\varphi_1 \wedge (l_2 \vee l_1) \wedge \varphi_2 \wedge (l_2 \vee \neg l_1) \wedge \varphi_3$$
$$\Downarrow_{resolve}$$
$$\varphi_1 \wedge (l_2) \wedge \varphi_2 \wedge \varphi_3$$
$$\Downarrow_{unit-propagate}$$
$$(\varphi_1 \wedge \varphi_2 \wedge \varphi_3)[l_2 \leftarrow \top]$$

Literal-Decision Heuristics (aka Branching Heuristics)

- Implemented in `decide_next_branch()`
- Branch is the source of non-determinism for DPLL
  $\Longrightarrow$ critical for efficiency
- Many literal-decision heuristics in literature (for DPLL & CDCL)

Some Heuristics

- MOMS heuristics (DPLL): pick the literal occurring most often in the minimal size clauses
  $\Longrightarrow$ fast and simple, many variants
- Jeroslow-Wang (DPLL): choose the literal with maximum

$$score(l) := \Sigma_{l \in c \ \& \ c \in \varphi} \ 2^{-|c|}$$

  $\Longrightarrow$ estimates l's contribution to the satisfiability of φ
- Satz [21] (DPLL): selects a candidate set of literals, perform unit propagation, chooses the one leading to smaller clause set
  $\Longrightarrow$ maximizes the effects of unit propagation
- VSIDS [28] (CDCL+): variable state independent decaying sum
  - "static": scores updated only at the end of a branch
  - "local": privileges variable in recently learned clauses

Restarts [16]

Idea: (according to some strategy) restart the search

- abandon the current search tree and reconstruct a new one
- The clauses learned prior to the restart are still there after the restart and can help pruning the search space
- avoid getting stuck in certain areas of the search space

$\Longrightarrow$ may significantly reduce the overall search space

Outline

1. Boolean Logics and SAT
2. Basic SAT-Solving Techniques
   - Generalities
   - Resolution
   - Tableaux
   - DPLL
3. Ordered Binary Decision Diagrams – OBDDs
4. Modern CDCL SAT Solvers
   - Limitations of Chronological Backtracking
   - Conflict-Driven Clause-Learning SAT solvers
   - Further Improvements
   - SAT Under Assumptions & Incremental SAT
5. SAT Functionalities: proofs, unsat cores, optimization

SAT under assumptions: $SAT(\varphi, \{l_1, ..., l_n\})$ [12]

- Many SAT solvers allow for solving a CNF formula φ under a set of assumption literals $\mathcal{A} \stackrel{\text{def}}{=} \{l_1, ..., l_n\}$: $SAT(\varphi, \{l_1, ..., l_n\})$
  - $SAT(\varphi, \{l_1, ..., l_n\})$: same result as $SAT(\varphi \wedge \bigwedge_{i=1}^{n} l_i)$
  - often useful to call the same formula with different assumption lists: $SAT(\varphi, \mathcal{A}_1), SAT(\varphi, \mathcal{A}_2), ...$
- Idea:
  - $l_1, ..., l_n$ "decided" at decision level 0 before starting the search
  - if backjump to level 0 on $C \stackrel{\text{def}}{=} \neg\eta$ s.t. $\eta \subseteq \mathcal{A}$, then return UNSAT

Property

If the "decision" strategy for conflict analysis is used,
then η is the subset of assumptions causing the inconsistency

SAT under assumptions: $SAT(\varphi, \{l_1, ..., l_n\})$ [12]

- Many SAT solvers allow for solving a CNF formula φ under a set of assumption literals $\mathcal{A} \stackrel{\text{def}}{=} \{l_1, ..., l_n\}$: $SAT(\varphi, \{l_1, ..., l_n\})$
  - $SAT(\varphi, \{l_1, ..., l_n\})$: same result as $SAT(\varphi \wedge \bigwedge_{i=1}^{n} l_i)$
  - often useful to call the same formula with different assumption lists: $SAT(\varphi, \mathcal{A}_1), SAT(\varphi, \mathcal{A}_2), ...$
- Idea:
  - $l_1, ..., l_n$ "decided" at decision level 0 before starting the search
  - if backjump to level 0 on $C \stackrel{\text{def}}{=} \neg\eta$ s.t. $\eta \subseteq \mathcal{A}$, then return UNSAT

Property

If the "decision" strategy for conflict analysis is used,
then η is the subset of assumptions causing the inconsistency

SAT under assumptions: $SAT(\varphi, \{l_1, ..., l_n\})$ [12]

- Many SAT solvers allow for solving a CNF formula φ under a set of assumption literals $\mathcal{A} \stackrel{\text{def}}{=} \{l_1, ..., l_n\}$: $SAT(\varphi, \{l_1, ..., l_n\})$
  - $SAT(\varphi, \{l_1, ..., l_n\})$: same result as $SAT(\varphi \wedge \bigwedge_{i=1}^{n} l_i)$
  - often useful to call the same formula with different assumption lists: $SAT(\varphi, \mathcal{A}_1)$, $SAT(\varphi, \mathcal{A}_2)$, ...
- Idea:
  - $l_1, ..., l_n$ "decided" at decision level 0 before starting the search
  - if backjump to level 0 on $C \stackrel{\text{def}}{=} \neg\eta$ s.t. $\eta \subseteq \mathcal{A}$, then return UNSAT

Property

If the "decision" strategy for conflict analysis is used,
then η is the subset of assumptions causing the inconsistency

Selection of sub-formulas

Idea: select clauses [12, 23]

Let φ be $\bigwedge_{i=1}^{n} C_i$.

- let $S_1 \ldots S_n$ be fresh Boolean atoms (selection variables).
- let $\mathcal{A} \stackrel{\text{def}}{=} \{S_{i_1}, ..., S_{i_k}\} \subseteq \{S_1, ..., S_n\}$

$\Longrightarrow$ SAT($\bigwedge_{i=1}^{n}(\neg S_i \vee C_i), \mathcal{A}$): same as SAT($\bigwedge_{i=i_1}^{i_k}(C_i)$)

- if S_i is not assumed, then $\neg S_i \vee C_i$ does not contribute to search

$\Longrightarrow$ "Select" (activate) only a subset of the clauses in φ at each call.

Generalised Idea: select blocks of clauses

Let φ be $\bigwedge_{i=1}^{n}(\bigwedge_{j=1}^{n_i} C_{ij})$.

- let $S_1 \ldots S_n$ be fresh Boolean atoms (selection variables).
- let $\mathcal{A} \stackrel{\text{def}}{=} \{S_{i_1}, ..., S_{i_k}\} \subseteq \{S_1, ..., S_n\}$
- SAT($\bigwedge_{i=1}^{n}(\bigwedge_{j=1}^{n_i}(\neg S_i \vee C_{ij})), \mathcal{A}$): same as SAT($\bigwedge_{i=i_1}^{i_k}(\bigwedge_{j=1}^{n_i} C_{ij})$)

$\Longrightarrow$ Allows for "selecting" block of clauses at each call.

Selection of sub-formulas

Idea: select clauses [12, 23]

Let φ be $\bigwedge_{i=1}^{n} C_i$.

- let $S_1 ... S_n$ be fresh Boolean atoms (selection variables).
- let $\mathcal{A} \stackrel{\text{def}}{=} \{S_{i_1}, ..., S_{i_k}\} \subseteq \{S_1, ..., S_n\}$

$\Longrightarrow$ SAT($\bigwedge_{i=1}^{n}(\neg S_i \vee C_i), \mathcal{A}$): same as SAT($\bigwedge_{i=i_1}^{i_k}(C_i)$)

- if S_i is not assumed, then $\neg S_i \vee C_i$ does not contribute to search

$\Longrightarrow$ "Select" (activate) only a subset of the clauses in φ at each call.

Generalised Idea: select blocks of clauses

Let φ be $\bigwedge_{i=1}^{n}(\bigwedge_{j=1}^{n_i} C_{ij})$.

- let $S_1 ... S_n$ be fresh Boolean atoms (selection variables).
- let $\mathcal{A} \stackrel{\text{def}}{=} \{S_{i_1}, ..., S_{i_k}\} \subseteq \{S_1, ..., S_n\}$
- SAT($\bigwedge_{i=1}^{n}(\bigwedge_{j=1}^{n_i}(\neg S_i \vee C_{ij})), \mathcal{A}$): same as SAT($\bigwedge_{i=i_1}^{i_k}(\bigwedge_{j=1}^{n_i} C_{ij})$)

$\Longrightarrow$ Allows for "selecting" block of clauses at each call.

Selection of sub-formulas

Idea: select clauses [12, 23]

Let φ be $\bigwedge_{i=1}^{n} C_i$.

- let $S_1 ... S_n$ be fresh Boolean atoms (selection variables).
- let $\mathcal{A} \stackrel{\text{def}}{=} \{S_{i_1}, ..., S_{i_k}\} \subseteq \{S_1, ..., S_n\}$

$\Longrightarrow$ SAT($\bigwedge_{i=1}^{n}(\neg S_i \vee C_i), \mathcal{A}$): same as SAT($\bigwedge_{i=i_1}^{i_k}(C_i)$)

- if S_i is not assumed, then $\neg S_i \vee C_i$ does not contribute to search

$\Longrightarrow$ "Select" (activate) only a subset of the clauses in φ at each call.

Generalised Idea: select blocks of clauses

Let φ be $\bigwedge_{i=1}^{n}(\bigwedge_{j=1}^{n_i} C_{ij})$.

- let $S_1 ... S_n$ be fresh Boolean atoms (selection variables).
- let $\mathcal{A} \stackrel{\text{def}}{=} \{S_{i_1}, ..., S_{i_k}\} \subseteq \{S_1, ..., S_n\}$
- SAT($\bigwedge_{i=1}^{n}(\bigwedge_{j=1}^{n_i}(\neg S_i \vee C_{ij})), \mathcal{A}$): same as SAT($\bigwedge_{i=i_1}^{i_k}(\bigwedge_{j=1}^{n_i} C_{ij})$)

$\Longrightarrow$ Allows for "selecting" block of clauses at each call.

Selection of sub-formulas

Idea: select clauses [12, 23]

Let φ be $\bigwedge_{i=1}^{n} C_i$.

- let $S_1 ... S_n$ be fresh Boolean atoms (selection variables).
- let $\mathcal{A} \stackrel{\text{def}}{=} \{S_{i_1}, ..., S_{i_k}\} \subseteq \{S_1, ..., S_n\}$

$\Longrightarrow$ SAT($\bigwedge_{i=1}^{n}(\neg S_i \vee C_i), \mathcal{A}$): same as SAT($\bigwedge_{i=i_1}^{i_k}(C_i)$)

- if S_i is not assumed, then $\neg S_i \vee C_i$ does not contribute to search

$\Longrightarrow$ "Select" (activate) only a subset of the clauses in φ at each call.

Generalised Idea: select blocks of clauses

Let φ be $\bigwedge_{i=1}^{n}(\bigwedge_{j=1}^{n_i} C_{ij})$.

- let $S_1 ... S_n$ be fresh Boolean atoms (selection variables).
- let $\mathcal{A} \stackrel{\text{def}}{=} \{S_{i_1}, ..., S_{i_k}\} \subseteq \{S_1, ..., S_n\}$
- SAT($\bigwedge_{i=1}^{n}(\bigwedge_{j=1}^{n_i}(\neg S_i \vee C_{ij})), \mathcal{A}$): same as SAT($\bigwedge_{i=i_1}^{i_k}(\bigwedge_{j=1}^{n_i} C_{ij})$)

$\Longrightarrow$ Allows for "selecting" block of clauses at each call.

Selection of sub-formulas

Idea: select clauses [12, 23]

Let φ be $\bigwedge_{i=1}^{n} C_i$.

- let $S_1 ... S_n$ be fresh Boolean atoms (selection variables).
- let $\mathcal{A} \stackrel{\text{def}}{=} \{S_{i_1}, ..., S_{i_k}\} \subseteq \{S_1, ..., S_n\}$

$\Longrightarrow$ SAT($\bigwedge_{i=1}^{n}(\neg S_i \vee C_i), \mathcal{A}$): same as SAT($\bigwedge_{i=i_1}^{i_k}(C_i)$)

- if S_i is not assumed, then $\neg S_i \vee C_i$ does not contribute to search

$\Longrightarrow$ "Select" (activate) only a subset of the clauses in φ at each call.

Generalised Idea: select blocks of clauses

Let φ be $\bigwedge_{i=1}^{n}(\bigwedge_{j=1}^{n_i} C_{ij})$.

- let $S_1 ... S_n$ be fresh Boolean atoms (selection variables).
- let $\mathcal{A} \stackrel{\text{def}}{=} \{S_{i_1}, ..., S_{i_k}\} \subseteq \{S_1, ..., S_n\}$
- SAT($\bigwedge_{i=1}^{n}(\bigwedge_{j=1}^{n_i}(\neg S_i \vee C_{ij})), \mathcal{A}$): same as SAT($\bigwedge_{i=i_1}^{i_k}(\bigwedge_{j=1}^{n_i} C_{ij})$)

$\Longrightarrow$ Allows for "selecting" block of clauses at each call.

Selection of sub-formulas

Idea: select clauses [12, 23]

Let φ be $\bigwedge_{i=1}^{n} C_i$.

- let $S_1 ... S_n$ be fresh Boolean atoms (selection variables).
- let $\mathcal{A} \stackrel{\text{def}}{=} \{S_{i_1}, ..., S_{i_k}\} \subseteq \{S_1, ..., S_n\}$

$\Longrightarrow$ SAT($\bigwedge_{i=1}^{n}(\neg S_i \vee C_i), \mathcal{A}$): same as SAT($\bigwedge_{i=i_1}^{i_k}(C_i)$)

- if S_i is not assumed, then $\neg S_i \vee C_i$ does not contribute to search

$\Longrightarrow$ "Select" (activate) only a subset of the clauses in φ at each call.

Generalised Idea: select blocks of clauses

Let φ be $\bigwedge_{i=1}^{n}(\bigwedge_{j=1}^{n_i} C_{ij})$.

- let $S_1 ... S_n$ be fresh Boolean atoms (selection variables).
- let $\mathcal{A} \stackrel{\text{def}}{=} \{S_{i_1}, ..., S_{i_k}\} \subseteq \{S_1, ..., S_n\}$
- SAT($\bigwedge_{i=1}^{n}(\bigwedge_{j=1}^{n_i}(\neg S_i \vee C_{ij})), \mathcal{A}$): same as SAT($\bigwedge_{i=i_1}^{i_k}(\bigwedge_{j=1}^{n_i} C_{ij})$)

$\Longrightarrow$ Allows for "selecting" block of clauses at each call.

Example

- Initial formula φ:

$$
\begin{array}{lllll}
(\ A_1 & \vee \neg A_2 & \vee \neg A_3 &)\wedge & // \textit{ group } 1 \\
(\neg A_3 & \vee \ A_2 & \vee \neg A_5 &)\wedge & // \textit{ group } 1 \\
(\neg A_2 & \vee \ A_5 & \vee \ A_7 &)\wedge & // \textit{ group } 2 \\
(\ A_3 & \vee \ A_5 & \vee \neg A_8 &)\wedge & // \textit{ group } 2 \\
(\neg A_1 & \vee \neg A_3 & \vee \ A_8 &)\wedge & // \textit{ group } 3
\end{array}
$$

- Augmented formula φ':

$$
\begin{array}{llllll}
(\neg S_1 & \vee \ A_1 & \vee \neg A_2 & \vee \neg A_3 &)\wedge & // \textit{ group } 1\textit{, inactive} \\
(\neg S_1 & \vee \neg A_3 & \vee \ A_2 & \vee \neg A_5 &)\wedge & // \textit{ group } 1\textit{, inactive} \\
(\neg S_2 & \vee \neg A_2 & \vee \ A_5 & \vee \ A_7 &)\wedge & // \textit{ group } 2\textit{, inactive} \\
(\neg S_2 & \vee \ A_2 & \vee \ A_5 & \vee \neg A_8 &)\wedge & // \textit{ group } 2\textit{, inactive} \\
(\neg S_3 & \vee \neg A_1 & \vee \neg A_3 & \vee \ A_8 &)\wedge & // \textit{ group } 3
\end{array}
$$

- $SAT(\varphi', \{S_2, S_3\})$: activates group 2,3
- $SAT(\varphi', \{S_1, S_3\})$: activates group 1,3

Example

- Initial formula φ:

$$
\begin{array}{llllll}
(& A_1 & \vee\neg A_2 & \vee\neg A_3 &)\wedge & // \textit{ group } 1 \\
(\neg A_3 & \vee & A_2 & \vee\neg A_5 &)\wedge & // \textit{ group } 1 \\
(\neg A_2 & \vee & A_5 & \vee & A_7 \;)\wedge & // \textit{ group } 2 \\
(& A_3 & \vee \; A_5 & \vee\neg A_8 &)\wedge & // \textit{ group } 2 \\
(\neg A_1 & \vee\neg A_3 & \vee & A_8 &)\wedge & // \textit{ group } 3
\end{array}
$$

- Augmented formula φ':

$$
\begin{array}{llllll}
(\neg S_1 & \vee \; A_1 & \vee\neg A_2 & \vee\neg A_3 &)\wedge & // \textit{ group } 1, \textit{ inactive} \\
(\neg S_1 & \vee\neg A_3 & \vee \; A_2 & \vee\neg A_5 &)\wedge & // \textit{ group } 1, \textit{ inactive} \\
(\neg S_2 & \vee\neg A_2 & \vee \; A_5 & \vee \; A_7 &)\wedge & // \textit{ group } 2, \textit{ inactive} \\
(\neg S_2 & \vee \; A_2 & \vee \; A_5 & \vee\neg A_8 &)\wedge & // \textit{ group } 2, \textit{ inactive} \\
(\neg S_3 & \vee\neg A_1 & \vee\neg A_3 & \vee \; A_8 &)\wedge & // \textit{ group } 3
\end{array}
$$

- $SAT(\varphi', \{S_2, S_3\})$: activates group 2,3
- $SAT(\varphi', \{S_1, S_3\})$: activates group 1,3

Example

- Initial formula φ:
  $$\begin{array}{llllll}(& A_1 & \vee\neg A_2 & \vee\neg A_3 &)\wedge & //\ group\ 1 \\ (\neg A_3 & \vee & A_2 & \vee\neg A_5 &)\wedge & //\ group\ 1 \\ (\neg A_2 & \vee & A_5 & \vee & A_7 &)\wedge & //\ group\ 2 \\ (& A_3 & \vee & A_5 & \vee\neg A_8 &)\wedge & //\ group\ 2 \\ (\neg A_1 & \vee\neg A_3 & \vee & A_8 &)\wedge & //\ group\ 3\end{array}$$
- Augmented formula φ':
  $$\begin{array}{lllll}(\neg S_1 & \vee\ A_1 & \vee\neg A_2 & \vee\neg A_3 &)\wedge\ //\ group\ 1,\ inactive \\ (\neg S_1 & \vee\neg A_3 & \vee\ A_2 & \vee\neg A_5 &)\wedge\ //\ group\ 1,\ inactive \\ (\neg S_2 & \vee\neg A_2 & \vee\ A_5 & \vee\ A_7 &)\wedge\ //\ group\ 2,\ inactive \\ (\neg S_2 & \vee\ A_2 & \vee\ A_5 & \vee\neg A_8 &)\wedge\ //\ group\ 2,\ inactive \\ (\neg S_3 & \vee\neg A_1 & \vee\neg A_3 & \vee\ A_8 &)\wedge\ //\ group\ 3\end{array}$$
- $SAT(\varphi', \{S_2, S_3\})$: activates group 2,3
- $SAT(\varphi', \{S_1, S_3\})$: activates group 1,3

Example

- Initial formula φ:

$$\begin{array}{llllll}
(& A_1 & \vee\neg A_2 & \vee\neg A_3 &)\wedge & // \textit{ group } 1 \\
(\neg A_3 & \vee & A_2 & \vee\neg A_5 &)\wedge & // \textit{ group } 1 \\
(\neg A_2 & \vee & A_5 & \vee & A_7)\wedge & // \textit{ group } 2 \\
(& A_3 & \vee & A_5 & \vee\neg A_8)\wedge & // \textit{ group } 2 \\
(\neg A_1 & \vee\neg A_3 & \vee & A_8 &)\wedge & // \textit{ group } 3
\end{array}$$

- Augmented formula φ':

$$\begin{array}{llllll}
(\neg S_1 & \vee A_1 & \vee\neg A_2 & \vee\neg A_3 &)\wedge & // \textit{ group } 1\textit{, inactive} \\
(\neg S_1 & \vee\neg A_3 & \vee A_2 & \vee\neg A_5 &)\wedge & // \textit{ group } 1\textit{, inactive} \\
(\neg S_2 & \vee\neg A_2 & \vee A_5 & \vee A_7 &)\wedge & // \textit{ group } 2\textit{, inactive} \\
(\neg S_2 & \vee A_2 & \vee A_5 & \vee\neg A_8 &)\wedge & // \textit{ group } 2\textit{, inactive} \\
(\neg S_3 & \vee\neg A_1 & \vee\neg A_3 & \vee A_8 &)\wedge & // \textit{ group } 3
\end{array}$$

- $SAT(\varphi', \{S_2, S_3\})$: activates group 2,3
- $SAT(\varphi', \{S_1, S_3\})$: activates group 1,3

Incremental SAT solving [12, 11]

- Many CDCL solvers provide a stack-based incremental interface
  - it is possible to push/pop ϕ_i into a stack of subformulas $\{\phi_1, ..., \phi_k\}$
  - check incrementally the satisfiability of $\varphi \stackrel{\text{def}}{=} \bigwedge_{i=1}^{k} \phi_i$.
- Maintains the status of the search from one call to the other
  - in particular it records the learned clauses (plus other information)

  $\Longrightarrow$ reuses search from one call to another
- Very useful in many applications (in particular in FV)

- Idea: incremental calls $SAT(\varphi', \mathcal{A}_1)$, $SAT(\varphi', \mathcal{A}_2)$,...
  - $\varphi' \stackrel{\text{def}}{=} \bigwedge_i (\neg S_i \vee \phi_i)$, $\mathcal{A}_j \subseteq \{S_1, ..., S_k\}$, $(\neg S_i \vee \bigwedge_j C_{ij}) \stackrel{\text{def}}{=} \bigwedge_j (\neg S_i \vee C_{ij})$
  - push/pop selection variables S_i
  - in practice, also subformulas ϕ_i can be pushed/popped
- Key efficiency issue: learned clauses safely reused from call to call (even if assumptions have been popped)

Incremental SAT solving [12, 11]

- Many CDCL solvers provide a stack-based incremental interface
  - it is possible to push/pop ϕ_i into a stack of subformulas $\{\phi_1, ..., \phi_k\}$
  - check incrementally the satisfiability of $\varphi \stackrel{\text{def}}{=} \bigwedge_{i=1}^{k} \phi_i$.
- Maintains the status of the search from one call to the other
  - in particular it records the learned clauses (plus other information)

  $\Longrightarrow$ reuses search from one call to another
- Very useful in many applications (in particular in FV)

- Idea: incremental calls $SAT(\varphi', \mathcal{A}_1)$, $SAT(\varphi', \mathcal{A}_2)$,...
  - $\varphi' \stackrel{\text{def}}{=} \bigwedge_i (\neg S_i \vee \phi_i)$, $\mathcal{A}_i \subseteq \{S_1, ..., S_k\}$, $(\neg S_i \vee \bigwedge_j C_{ij}) \stackrel{\text{def}}{=} \bigwedge_j (\neg S_i \vee C_{ij})$
  - push/pop selection variables S_i
  - in practice, also subformulas ϕ_i can be pushed/popped
- Key efficiency issue: learned clauses safely reused from call to call (even if assumptions have been popped)

Incremental SAT solving [12, 11]

- Many CDCL solvers provide a stack-based incremental interface
  - it is possible to push/pop ϕ_i into a stack of subformulas $\{\phi_1, ..., \phi_k\}$
  - check incrementally the satisfiability of $\varphi \stackrel{\text{def}}{=} \bigwedge_{i=1}^{k} \phi_i$.
- Maintains the status of the search from one call to the other
  - in particular it records the learned clauses (plus other information)

  $\Longrightarrow$ reuses search from one call to another
- Very useful in many applications (in particular in FV)

Incremental SAT solving [12, 11]

- Many CDCL solvers provide a stack-based incremental interface
  - it is possible to push/pop ϕ_i into a stack of subformulas $\{\phi_1, ..., \phi_k\}$
  - check incrementally the satisfiability of $\varphi \stackrel{\text{def}}{=} \bigwedge_{i=1}^{k} \phi_i$.
- Maintains the status of the search from one call to the other
  - in particular it records the learned clauses (plus other information)

  $\Longrightarrow$ reuses search from one call to another
- Very useful in many applications (in particular in FV)

- Idea: incremental calls $SAT(\varphi', \mathcal{A}_1)$, $SAT(\varphi', \mathcal{A}_2)$,...
  - $\varphi' \stackrel{\text{def}}{=} \bigwedge_i(\neg S_i \vee \phi_i)$, $\mathcal{A}_j \subseteq \{S_1, ..., S_k\}$, $(\neg S_i \vee \bigwedge_j C_{ij}) \stackrel{\text{def}}{=} \bigwedge_j(\neg S_i \vee C_{ij})$
  - push/pop selection variables S_i
  - in practice, also subformulas ϕ_i can be pushed/popped
- Key efficiency issue: learned clauses safely reused from call to call (even if assumptions have been popped)
  - a learned clause $C \stackrel{\text{def}}{=} \bigvee_j \neg S_j \vee C'$ is s.t. $\bigwedge_j(\neg S_j \vee \phi_j) \models C$

  $\Longrightarrow$ C contains the vars selecting the subformulas it is derived from

  $\Longrightarrow$ in $SAT(\varphi', \mathcal{A})$, if some $S_j \not\in \mathcal{A}$, then C is not active

Incremental SAT solving [12, 11]

- Many CDCL solvers provide a stack-based incremental interface
  - it is possible to push/pop ϕ_i into a stack of subformulas $\{\phi_1, ..., \phi_k\}$
  - check incrementally the satisfiability of $\varphi \stackrel{\text{def}}{=} \bigwedge_{i=1}^{k} \phi_i$.
- Maintains the status of the search from one call to the other
  - in particular it records the learned clauses (plus other information)

  $\Longrightarrow$ reuses search from one call to another
- Very useful in many applications (in particular in FV)

- Idea: incremental calls $SAT(\varphi', \mathcal{A}_1)$, $SAT(\varphi', \mathcal{A}_2)$,...
  - $\varphi' \stackrel{\text{def}}{=} \bigwedge_i (\neg S_i \vee \phi_i)$, $\mathcal{A}_j \subseteq \{S_1, ..., S_k\}$, $(\neg S_i \vee \bigwedge_j C_{ij}) \stackrel{\text{def}}{=} \bigwedge_j (\neg S_i \vee C_{ij})$
  - push/pop selection variables S_i
  - in practice, also subformulas ϕ_i can be pushed/popped
- Key efficiency issue: learned clauses safely reused from call to call (even if assumptions have been popped)
  - a learned clause $C \stackrel{\text{def}}{=} \bigvee_j \neg S_j \vee C'$ is s.t. $\bigwedge_j (\neg S_j \vee \phi_j) \models C$

  $\Longrightarrow$ C contains the vars selecting the subformulas it is derived from

  $\Longrightarrow$ in $SAT(\varphi', \mathcal{A})$, if some $S_j \notin \mathcal{A}$, then C is not active

Incremental SAT solving [12, 11]

- Many CDCL solvers provide a stack-based incremental interface
  - it is possible to push/pop ϕ_i into a stack of subformulas $\{\phi_1, ..., \phi_k\}$
  - check incrementally the satisfiability of $\varphi \stackrel{\text{def}}{=} \bigwedge_{i=1}^{k} \phi_i$.
- Maintains the status of the search from one call to the other
  - in particular it records the learned clauses (plus other information)

  $\Longrightarrow$ reuses search from one call to another
- Very useful in many applications (in particular in FV)

- Idea: incremental calls $SAT(\varphi', \mathcal{A}_1)$, $SAT(\varphi', \mathcal{A}_2)$,...
  - $\varphi' \stackrel{\text{def}}{=} \bigwedge_i (\neg S_i \vee \phi_i)$, $\mathcal{A}_j \subseteq \{S_1, ..., S_k\}$, $(\neg S_i \vee \bigwedge_j C_{ij}) \stackrel{\text{def}}{=} \bigwedge_j (\neg S_i \vee C_{ij})$
  - push/pop selection variables S_i
  - in practice, also subformulas ϕ_i can be pushed/popped
- Key efficiency issue: learned clauses safely reused from call to call (even if assumptions have been popped)
  - a learned clause $C \stackrel{\text{def}}{=} \bigvee_j \neg S_j \vee C'$ is s.t. $\bigwedge_j (\neg S_j \vee \phi_j) \models C$

  $\Longrightarrow$ C contains the vars selecting the subformulas it is derived from

  $\Longrightarrow$ in $SAT(\varphi', \mathcal{A})$, if some $S_j \not\in \mathcal{A}$, then C is not active

Incremental SAT solving [12, 11]

- Many CDCL solvers provide a stack-based incremental interface
  - it is possible to push/pop ϕ_i into a stack of subformulas $\{\phi_1, ..., \phi_k\}$
  - check incrementally the satisfiability of $\varphi \stackrel{\text{def}}{=} \bigwedge_{i=1}^{k} \phi_i$.
- Maintains the status of the search from one call to the other
  - in particular it records the learned clauses (plus other information)

  $\Longrightarrow$ reuses search from one call to another
- Very useful in many applications (in particular in FV)

- Idea: incremental calls $SAT(\varphi', \mathcal{A}_1)$, $SAT(\varphi', \mathcal{A}_2)$,...
  - $\varphi' \stackrel{\text{def}}{=} \bigwedge_i(\neg S_i \vee \phi_i)$, $\mathcal{A}_j \subseteq \{S_1, ..., S_k\}$, $(\neg S_i \vee \bigwedge_j C_{ij}) \stackrel{\text{def}}{=} \bigwedge_j(\neg S_i \vee C_{ij})$
  - push/pop selection variables S_i
  - in practice, also subformulas ϕ_i can be pushed/popped
- Key efficiency issue: learned clauses safely reused from call to call (even if assumptions have been popped)
  - a learned clause $C \stackrel{\text{def}}{=} \bigvee_j \neg S_j \vee C'$ is s.t. $\bigwedge_j(\neg S_j \vee \phi_j) \models C$

  $\Longrightarrow$ C contains the vars selecting the subformulas it is derived from

  $\Longrightarrow$ in $SAT(\varphi', \mathcal{A})$, if some $S_j \notin \mathcal{A}$, then C is not active

Incremental SAT solving [12, 11]

- Many CDCL solvers provide a stack-based incremental interface
  - it is possible to push/pop ϕ_i into a stack of subformulas $\{\phi_1, ..., \phi_k\}$
  - check incrementally the satisfiability of $\varphi \stackrel{\text{def}}{=} \bigwedge_{i=1}^{k} \phi_i$.
- Maintains the status of the search from one call to the other
  - in particular it records the learned clauses (plus other information)

  $\Longrightarrow$ reuses search from one call to another
- Very useful in many applications (in particular in FV)

- Idea: incremental calls $SAT(\varphi', \mathcal{A}_1)$, $SAT(\varphi', \mathcal{A}_2)$,...
  - $\varphi' \stackrel{\text{def}}{=} \bigwedge_i (\neg S_i \vee \phi_i)$, $\mathcal{A}_j \subseteq \{S_1, ..., S_k\}$, $(\neg S_i \vee \bigwedge_j C_{ij}) \stackrel{\text{def}}{=} \bigwedge_j (\neg S_i \vee C_{ij})$
  - push/pop selection variables S_i
  - in practice, also subformulas ϕ_i can be pushed/popped
- Key efficiency issue: learned clauses safely reused from call to call (even if assumptions have been popped)
  - a learned clause $C \stackrel{\text{def}}{=} \bigvee_j \neg S_j \vee C'$ is s.t. $\bigwedge_j (\neg S_j \vee \phi_j) \models C$

  $\Longrightarrow$ C contains the vars selecting the subformulas it is derived from

  $\Longrightarrow$ in $SAT(\varphi', \mathcal{A})$, if some $S_j \notin \mathcal{A}$, then C is not active

Incremental SAT solving [12, 11]

- Many CDCL solvers provide a stack-based incremental interface
  - it is possible to push/pop ϕ_i into a stack of subformulas $\{\phi_1, ..., \phi_k\}$
  - check incrementally the satisfiability of $\varphi \stackrel{\text{def}}{=} \bigwedge_{i=1}^{k} \phi_i$.
- Maintains the status of the search from one call to the other
  - in particular it records the learned clauses (plus other information)

  $\Longrightarrow$ reuses search from one call to another
- Very useful in many applications (in particular in FV)

- Idea: incremental calls $SAT(\varphi', \mathcal{A}_1)$, $SAT(\varphi', \mathcal{A}_2)$,...
  - $\varphi' \stackrel{\text{def}}{=} \bigwedge_i (\neg S_i \vee \phi_i)$, $\mathcal{A}_j \subseteq \{S_1, ..., S_k\}$, $(\neg S_i \vee \bigwedge_j C_{ij}) \stackrel{\text{def}}{=} \bigwedge_j (\neg S_i \vee C_{ij})$
  - push/pop selection variables S_i
  - in practice, also subformulas ϕ_i can be pushed/popped
- Key efficiency issue: learned clauses safely reused from call to call (even if assumptions have been popped)
  - a learned clause $C \stackrel{\text{def}}{=} \bigvee_j \neg S_j \vee C'$ is s.t. $\bigwedge_j (\neg S_j \vee \phi_j) \models C$

  $\Longrightarrow$ C contains the vars selecting the subformulas it is derived from

  $\Longrightarrow$ in $SAT(\varphi', \mathcal{A})$, if some $S_j \notin \mathcal{A}$, then C is not active

Example

- Initial formula φ:

$$
\begin{array}{llllll}
\ldots & & & & \wedge & \\
(\; A_1 & \vee\neg A_2 & \vee\neg A_3 & &)\wedge & //\ \phi_1 \\
(\neg A_3 & \vee\ A_2 & \vee\neg A_5 & &)\wedge & //\ \phi_1
\end{array}
$$

- Augmented formula φ':

$$
\begin{array}{llllll}
\ldots & & & & \wedge & \\
(\neg S_1 & \vee\ A_1 & \vee\neg A_2 & \vee\neg A_3 &)\wedge & //\ \phi_1 \\
(\neg S_1 & \vee\neg A_3 & \vee\ A_2 & \vee\neg A_5 &)\wedge & //\ \phi_1
\end{array}
$$

[push(S_1)]: $SAT(\varphi', \{..., S_1\})$: ϕ_1 active $\Longrightarrow$ learn C_1 from ϕ_1

- C_1 derived from $\phi_1 \Longrightarrow C_1$ active only when ϕ_1 is active
- C_2 derived from $\phi_1, \phi_2 \Longrightarrow C_2$ active only when both ϕ_1, ϕ_2 are active

Example

- Initial formula φ:

$$
\begin{array}{llllll}
\ldots & & & & \wedge & \\
(\;A_1 & \vee\neg A_2 & \vee\neg A_3 &) & \wedge & // \; \phi_1 \\
(\neg A_3 & \vee \; A_2 & \vee\neg A_5 &) & \wedge & // \; \phi_1
\end{array}
$$

- Augmented formula φ':

$$
\begin{array}{lllllll}
\ldots & & & & & \wedge & \\
(\neg S_1 & \vee \; A_1 & \vee\neg A_2 & \vee\neg A_3 &) & \wedge & // \; \phi_1 \\
(\neg S_1 & \vee\neg A_3 & \vee \; A_2 & \vee\neg A_5 &) & \wedge & // \; \phi_1
\end{array}
$$

$$
\begin{array}{lllllll}
(\neg S_1 & \vee \; A_1 & \vee\neg A_3 & \vee\neg A_5 &) & \wedge & // \; \textit{learned } C_1
\end{array}
$$

[push(S_1)]: $SAT(\varphi', \{..., S_1\})$: ϕ_1 active $\Longrightarrow$ learn C_1 from ϕ_1

- C_1 derived from $\phi_1 \Longrightarrow C_1$ active only when ϕ_1 is active
- C_2 derived from $\phi_1, \phi_2 \Longrightarrow C_2$ active only when both ϕ_1, ϕ_2 are active

Example

- Initial formula φ:

$$\begin{array}{llllll}
\dots & & & & \wedge & \\
(\ A_1 & \vee\neg A_2 & \vee\neg A_3 &) & \wedge & // \ \phi_1 \\
(\neg A_3 & \vee \ A_2 & \vee\neg A_5 &) & \wedge & // \ \phi_1 \\
(\neg A_2 & \vee \ A_5 & \vee \ A_7 &) & \wedge & // \ \phi_2 \\
(\neg A_1 & \vee\neg A_3 & \vee\neg A_5 &) & \wedge & // \ \phi_2
\end{array}$$

- Augmented formula φ':

$$\begin{array}{lllllll}
\dots & & & & & \wedge & \\
(\neg S_1 & \vee \ A_1 & \vee\neg A_2 & \vee\neg A_3 &) & \wedge & // \ \phi_1 \\
(\neg S_1 & \vee\neg A_3 & \vee \ A_2 & \vee\neg A_5 &) & \wedge & // \ \phi_1 \\
(\neg S_2 & \vee\neg A_2 & \vee \ A_5 & \vee \ A_7 &) & \wedge & // \ \phi_2, \textit{inactive} \\
(\neg S_2 & \vee\neg A_1 & \vee\neg A_3 & \vee\neg A_5 &) & \wedge & // \ \phi_2, \textit{inactive} \\
 & & & & & & \\
(\neg S_1 & \vee \ A_1 & \vee\neg A_3 & \vee\neg A_5 &) & \wedge & // \ \textit{learned } C_1
\end{array}$$

[push(S_2)]: $SAT(\varphi', \{..., S_1, S_2\})$: ϕ_1, ϕ_2 active $\Longrightarrow$ learn C_2 from ϕ_1, ϕ_2

- C_1 derived from $\phi_1 \Longrightarrow C_1$ active only when ϕ_1 is active
- C_2 derived from $\phi_1, \phi_2 \Longrightarrow C_2$ active only when both ϕ_1, ϕ_2 are active

Example

- Initial formula φ:

$$\begin{array}{llllll} \ldots & & & & \wedge & \\ (\; A_1 & \vee\neg A_2 & \vee\neg A_3 & &)\wedge & // \; \phi_1 \\ (\neg A_3 & \vee \; A_2 & \vee\neg A_5 & &)\wedge & // \; \phi_1 \\ (\neg A_2 & \vee \; A_5 & \vee \; A_7 & &)\wedge & // \; \phi_2 \\ (\neg A_1 & \vee\neg A_3 & \vee\neg A_5 & &)\wedge & // \; \phi_2 \end{array}$$

- Augmented formula φ':

$$\begin{array}{llllll} \ldots & & & & \wedge & \\ (\neg S_1 & \vee \; A_1 & \vee\neg A_2 & \vee\neg A_3 &)\wedge & // \; \phi_1 \\ (\neg S_1 & \vee\neg A_3 & \vee \; A_2 & \vee\neg A_5 &)\wedge & // \; \phi_1 \\ (\neg S_2 & \vee\neg A_2 & \vee \; A_5 & \vee \; A_7 &)\wedge & // \; \phi_2, \text{ inactive} \\ (\neg S_2 & \vee\neg A_1 & \vee\neg A_3 & \vee\neg A_5 &)\wedge & // \; \phi_2, \text{ inactive} \\ & & & & & \\ (\neg S_1 & \vee \; A_1 & \vee\neg A_3 & \vee\neg A_5 &)\wedge & // \; \textit{learned } C_1 \\ (\neg S_1 & \vee\neg S_2 & \vee\neg A_3 & \vee\neg A_5 &)\wedge & // \; \textit{learned } C_2, \text{ inactive} \end{array}$$

[push(S_2)]: $SAT(\varphi', \{..., S_1, S_2\})$: ϕ_1, ϕ_2 active $\Longrightarrow$ learn C_2 from ϕ_1, ϕ_2

- C_1 derived from $\phi_1 \Longrightarrow C_1$ active only when ϕ_1 is active
- C_2 derived from $\phi_1, \phi_2 \Longrightarrow C_2$ active only when both ϕ_1, ϕ_2 are active

Example

- Initial formula φ:

$$\begin{array}{llllll}
\ldots & & & & \wedge & \\
(\; A_1 & \vee\neg A_2 & \vee\neg A_3 &) & \wedge & // \; \phi_1 \\
(\neg A_3 & \vee \; A_2 & \vee\neg A_5 &) & \wedge & // \; \phi_1 \\
(\neg A_1 & \vee\neg A_3 & \vee \; A_8 &) & \wedge & // \; \phi_3
\end{array}$$

- Augmented formula φ':

$$\begin{array}{lllllll}
\ldots & & & & & \wedge & \\
(\neg S_1 & \vee \; A_1 & \vee\neg A_2 & \vee\neg A_3 &) & \wedge & // \; \phi_1 \\
(\neg S_1 & \vee\neg A_3 & \vee \; A_2 & \vee\neg A_5 &) & \wedge & // \; \phi_1 \\
(\neg S_2 & \vee\neg A_2 & \vee \; A_5 & \vee \; A_7 &) & \wedge & // \; \phi_2, \textit{inactive} \\
(\neg S_2 & \vee\neg A_1 & \vee\neg A_3 & \vee\neg A_5 &) & \wedge & // \; \phi_2, \textit{inactive} \\
(\neg S_3 & \vee\neg A_1 & \vee\neg A_3 & \vee \; A_8 &) & \wedge & // \; \phi_3 \\
(\neg S_1 & \vee \; A_1 & \vee\neg A_3 & \vee\neg A_5 &) & \wedge & // \; \textit{learned } C_1 \\
(\neg S_1 & \vee\neg S_2 & \vee\neg A_3 & \vee\neg A_5 &) & \wedge & // \; \textit{learned } C_2, \textit{inactive}
\end{array}$$

[pop(S_2);push(S_3)]: $SAT(\varphi', \{..., S_1, S_3\})$: ϕ_1, ϕ_3 active $\Longrightarrow$...

- C_1 derived from $\phi_1 \Longrightarrow C_1$ active only when ϕ_1 is active
- C_2 derived from $\phi_1, \phi_2 \Longrightarrow C_2$ active only when both ϕ_1, ϕ_2 are active

Example

- Initial formula φ:

$$
\begin{array}{llllll}
\ldots & & & \wedge & \\
(\ A_1 & \vee\neg A_2 & \vee\neg A_3 &)\wedge & //\ \phi_1 \\
(\neg A_3 & \vee\ A_2 & \vee\neg A_5 &)\wedge & //\ \phi_1 \\
 & & & & \\
(\neg A_1 & \vee\neg A_3 & \vee\ A_8 &)\wedge & //\ \phi_3
\end{array}
$$

- Augmented formula φ':

$$
\begin{array}{lllllll}
\ldots & & & & \wedge & \\
(\neg S_1 & \vee\ A_1 & \vee\neg A_2 & \vee\neg A_3 &)\wedge & //\ \phi_1 \\
(\neg S_1 & \vee\neg A_3 & \vee\ A_2 & \vee\neg A_5 &)\wedge & //\ \phi_1 \\
(\neg S_2 & \vee\neg A_2 & \vee\ A_5 & \vee\ A_7 &)\wedge & //\ \phi_2, \mathit{inactive} \\
(\neg S_2 & \vee\neg A_1 & \vee\neg A_3 & \vee\neg A_5 &)\wedge & //\ \phi_2, \mathit{inactive} \\
(\neg S_3 & \vee\neg A_1 & \vee\neg A_3 & \vee\ A_8 &)\wedge & //\ \phi_3 \\
(\neg S_1 & \vee\ A_1 & \vee\neg A_3 & \vee\neg A_5 &)\wedge & //\ \mathit{learned}\ C_1 \\
(\neg S_1 & \vee\neg S_2 & \vee\neg A_3 & \vee\neg A_5 &)\wedge & //\ \mathit{learned}\ C_2, \mathit{inactive}
\end{array}
$$

[pop(S_2);push(S_3)]: $SAT(\varphi', \{..., S_1, S_3\})$: ϕ_1, ϕ_3 active $\Longrightarrow$...

- C_1 derived from $\phi_1 \Longrightarrow C_1$ active only when ϕ_1 is active
- C_2 derived from $\phi_1, \phi_2 \Longrightarrow C_2$ active only when both ϕ_1, ϕ_2 are active

Outline

Advanced functionalities

Advanced SAT functionalities (very important in formal verification):

- Building proofs of unsatisfiability
- Extracting unsatisfiable Cores
- Enumeration in SAT: AllSAT (hints)
- Optimization in SAT: MaxSAT (hints)

Building Proofs of Unsatisfiability

- When φ is unsat, it is very important to build a (resolution) proof of unsatisfiability:
  - to verify the result of the solver
  - to understand a "reason" for unsatisfiability
  - to build unsatisfiable cores and interpolants
- Can be built by keeping track of the resolution steps performed when constructing the conflict clauses.

Building Proofs of Unsatisfiability

- When φ is unsat, it is very important to build a (resolution) proof of unsatisfiability:
  - to verify the result of the solver
  - to understand a “reason” for unsatisfiability
  - to build unsatisfiable cores and interpolants
- Can be built by keeping track of the resolution steps performed when constructing the conflict clauses.

Building Proofs of Unsatisfiability

- Recall: each conflict clause C_i learned is computed from the conflicting clause C_{i-k} by backward resolving with the antecedent clause of one literal

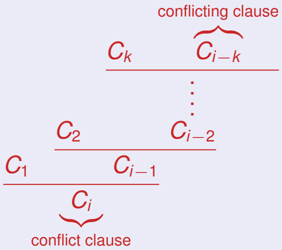
$$
\cfrac{C_1 \qquad \cfrac{C_2 \qquad \cfrac{\vdots}{C_{i-2}}}{C_{i-1}}}{\underbrace{C_i}_{\text{conflict clause}}}
\qquad\qquad
\cfrac{C_k \qquad \overbrace{C_{i-k}}^{\text{conflicting clause}}}{\vdots}
$$

- $C_1, \ldots, C_k$, and C_{i-k} can be either original or learned clauses
- each resolution (sub)proof can be easily tracked:

```
k i-k -> i-k-1
...
2 i-2 -> i-1
1 i-1 -> i
```

Building Proofs of Unsatisfiability

- Recall: each conflict clause C_i learned is computed from the conflicting clause C_{i-k} by backward resolving with the antecedent clause of one literal

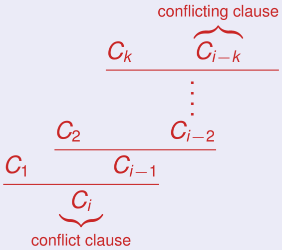


- $C_1, \ldots, C_k$, and C_{i-k} can be either original or learned clauses
- each resolution (sub)proof can be easily tracked:

```
k i-k -> i-k-1
...
2 i-2 -> i-1
1 i-1 -> i
```

Building Proofs of Unsatisfiability

- Recall: each conflict clause C_i learned is computed from the conflicting clause C_{i-k} by backward resolving with the antecedent clause of one literal



- $C_1, \ldots, C_k$, and C_{i-k} can be either original or learned clauses
- each resolution (sub)proof can be easily tracked:

```
k i-k -> i-k-1
...
2 i-2 -> i-1
1 i-1 -> i
```

Building Proofs of Unsatisfiability

- ... in particular, if φ is unsatisfiable, the last step produces "false" as conflict clause:

$$\dfrac{C_1 \qquad \dfrac{C_2 \qquad \begin{array}{c} \dfrac{C_k \qquad \overbrace{C_{i-k}}^{\text{conflicting clause}}}{\vdots} \\ C_{i-2} \end{array}}{C_{i-1}}}{\perp}$$

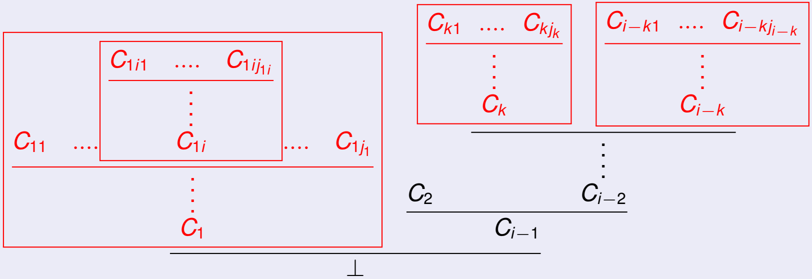
- note: $C_1 = l$, $C_{i-1} = \neg l$ for some literal l
- $C_1, ..., C_k$, and C_{i-k} can be original or learned clauses...

Building Proofs of Unsatisfiability

Starting from the previous proof of unsatisfiability, repeat recursively:

- for every learned leaf clause C_i, substitute C_i with the resolution proof generating it

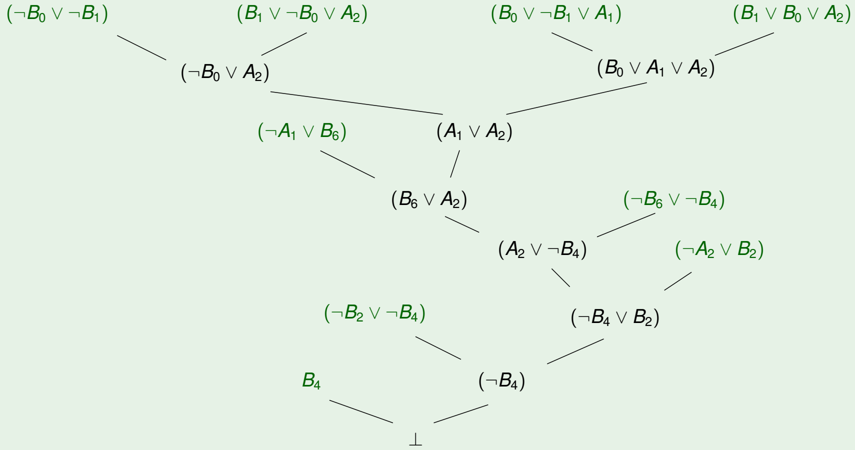
until all leaf clauses are original clauses



$\Longrightarrow$ We obtain a resolution proof of unsatisfiability for (a subset of) the clauses in φ

Building Proofs of Unsatisfiability: example

$$(B_0 \vee \neg B_1 \vee A_1) \wedge (B_0 \vee B_1 \vee A_2) \wedge (\neg B_0 \vee B_1 \vee A_2) \wedge (\neg B_0 \vee \neg B_1) \wedge (\neg B_2 \vee \neg B_4) \wedge$$
$$(\neg A_2 \vee B_2) \wedge (\neg A_1 \vee B_3) \wedge B_4 \wedge (A_2 \vee B_5) \wedge (\neg B_6 \vee \neg B_4) \wedge (B_6 \vee \neg A_1) \wedge B_7$$



Extraction of unsatisfiable cores

- Problem: given an unsatisfiable set of clauses, extract from it a (possibly small/minimal/minimum) unsatisfiable subset

  $\Longrightarrow$ unsatisfiable cores (aka (Minimal) Unsatisfiable Subsets, (M)US)
- Lots of literature on the topic [46, 24, 26, 31, 43, 19, 13, 6]
- We recognize two main approaches:
  - Proof-based approach [46]: byproduct of finding a resolution proof
  - Assumption-based approach [24]: use extra variables labeling clauses
- Many optimizations for further reducing the size of the core:
  - repeat the process up to fixpoit
  - remove clauses one-by one, until satisfiability is obtained
  - combinations of the two processed above
  - ...

The proof-based approach to core extraction [46]

Unsat core: the set of leaf clauses of a resolution proof

$$(B_0 \vee \neg B_1 \vee A_1) \wedge (B_0 \vee B_1 \vee A_2) \wedge (\neg B_0 \vee B_1 \vee A_2) \wedge (\neg B_0 \vee \neg B_1) \wedge (\neg B_2 \vee \neg B_4) \wedge$$
$$(\neg A_2 \vee B_2) \wedge (\neg A_1 \vee B_3) \wedge B_4 \wedge (A_2 \vee B_5) \wedge (\neg B_6 \vee \neg B_4) \wedge (B_6 \vee \neg A_1) \wedge B_7$$

$$
\begin{prooftree}
\AxiomC{B_4}
\AxiomC{$(\neg B_2 \vee \neg B_4)$}
\AxiomC{$(\neg A_1 \vee B_6)$}
\AxiomC{$(\neg B_0 \vee \neg B_1)$}
\AxiomC{$(B_1 \vee \neg B_0 \vee A_2)$}
\BinaryInfC{$(\neg B_0 \vee A_2)$}
\AxiomC{$(B_0 \vee \neg B_1 \vee A_1)$}
\AxiomC{$(B_1 \vee B_0 \vee A_2)$}
\BinaryInfC{$(B_0 \vee A_1 \vee A_2)$}
\BinaryInfC{$(A_1 \vee A_2)$}
\BinaryInfC{$(B_6 \vee A_2)$}
\AxiomC{$(\neg B_6 \vee \neg B_4)$}
\BinaryInfC{$(A_2 \vee \neg B_4)$}
\AxiomC{$(\neg A_2 \vee B_2)$}
\BinaryInfC{$(\neg B_4 \vee B_2)$}
\BinaryInfC{$(\neg B_4)$}
\BinaryInfC{$\perp$}
\end{prooftree}
$$

The assumption-based approach to core extraction [24]

Based on the following process:

(i) each clause C_i is substituted by $\neg S_i \vee C_i$, s.t. S_i fresh "selector" variable

(ii) before starting the search each S_i is forced to true.

(iii) final conflict clause at dec. level 0: $\bigvee_j \neg S_j$

$\Longrightarrow$ $\{C_j\}_j$ is the unsat core!

The assumption-based approach to core extraction [24]

Based on the following process:

(i) each clause C_i is substituted by $\neg S_i \vee C_i$, s.t. S_i fresh "selector" variable

(ii) before starting the search each S_i is forced to true.

(iii) final conflict clause at dec. level 0: $\bigvee_j \neg S_j$

$\Longrightarrow$ $\{C_j\}_j$ is the unsat core!

The assumption-based approach to core extraction [24]

Based on the following process:

(i) each clause C_i is substituted by $\neg S_i \vee C_i$, s.t. S_i fresh “selector” variable

(ii) before starting the search each S_i is forced to true.

(iii) final conflict clause at dec. level 0: $\bigvee_j \neg S_j$

$\Longrightarrow$ $\{C_j\}_j$ is the unsat core!

The assumption-based approach to core extraction [24]

Based on the following process:

(i) each clause C_i is substituted by $\neg S_i \vee C_i$, s.t. S_i fresh "selector" variable

(ii) before starting the search each S_i is forced to true.

(iii) final conflict clause at dec. level 0: $\bigvee_j \neg S_j$

$\Longrightarrow \{C_j\}_j$ is the unsat core!

The assumption-based approach to core extraction [24]

Based on the following process:

(i) each clause C_i is substituted by $\neg S_i \vee C_i$, s.t. S_i fresh "selector" variable

(ii) before starting the search each S_i is forced to true.

(iii) final conflict clause at dec. level 0: $\bigvee_j \neg S_j$

$\Longrightarrow$ $\{C_j\}_j$ is the unsat core!

The assumption-based approach to core extraction

Example

$(B_0 \vee \neg B_1 \vee A_1) \wedge (B_0 \vee B_1 \vee A_2) \wedge (\neg B_0 \vee B_1 \vee A_2) \wedge$
$(\neg B_0 \vee \neg B_1) \wedge (\neg B_2 \vee \neg B_4) \wedge (\neg A_2 \vee B_2) \wedge (\neg A_1 \vee B_3) \wedge$
$B_4 \wedge (A_2 \vee B_5) \wedge (\neg B_6 \vee \neg B_4) \wedge (B_6 \vee \neg A_1) \wedge B_7$

(i) add selector variables: $(\neg S_1 \vee B_0 \vee \neg B_1 \vee A_1) \wedge (\neg S_2 \vee B_0 \vee B_1 \vee A_2) \wedge (\neg S_3 \vee \neg B_0 \vee B_1 \vee A_2) \wedge$
$(\neg S_4 \vee \neg B_0 \vee \neg B_1) \wedge (\neg S_5 \vee \neg B_2 \vee \neg B_4) \wedge (\neg S_6 \vee \neg A_2 \vee B_2) \wedge$
$(\neg S_7 \vee \neg A_1 \vee B_3) \wedge (\neg S_8 \vee B_4) \wedge (\neg S_9 \vee A_2 \vee B_5) \wedge (\neg S_{10} \vee \neg B_6 \vee \neg B_4) \wedge$
$(\neg S_{11} \vee B_6 \vee \neg A_1) \wedge (\neg S_{12} \vee B_7)$

(ii) The conflict analysis returns: $\neg S_1 \vee \neg S_2 \vee \neg S_3 \vee \neg S_4 \vee \neg S_5 \vee \neg S_6 \vee \neg S_8 \vee \neg S_{10} \vee \neg S_{11}$,

(iii) corresponding to the unsat core:

$(B_0 \vee \neg B_1 \vee A_1) \wedge (B_0 \vee B_1 \vee A_2) \wedge (\neg B_0 \vee B_1 \vee A_2) \wedge$
$(\neg B_0 \vee \neg B_1) \wedge (\neg B_2 \vee \neg B_4) \wedge (\neg A_2 \vee B_2) \wedge$
$B_4 \wedge (\neg B_6 \vee \neg B_4) \wedge (B_6 \vee \neg A_1)$

The assumption-based approach to core extraction

Example

$$(B_0 \vee \neg B_1 \vee A_1) \wedge (B_0 \vee B_1 \vee A_2) \wedge (\neg B_0 \vee B_1 \vee A_2) \wedge$$
$$(\neg B_0 \vee \neg B_1) \wedge (\neg B_2 \vee \neg B_4) \wedge (\neg A_2 \vee B_2) \wedge (\neg A_1 \vee B_3) \wedge$$
$$B_4 \wedge (A_2 \vee B_5) \wedge (\neg B_6 \vee \neg B_4) \wedge (B_6 \vee \neg A_1) \wedge B_7$$

(i) add selector variables:

$$(\neg S_1 \vee B_0 \vee \neg B_1 \vee A_1) \wedge (\neg S_2 \vee B_0 \vee B_1 \vee A_2) \wedge (\neg S_3 \vee \neg B_0 \vee B_1 \vee A_2) \wedge$$
$$(\neg S_4 \vee \neg B_0 \vee \neg B_1) \wedge (\neg S_5 \vee \neg B_2 \vee \neg B_4) \wedge (\neg S_6 \vee \neg A_2 \vee B_2) \wedge$$
$$(\neg S_7 \vee \neg A_1 \vee B_3) \wedge (\neg S_8 \vee B_4) \wedge (\neg S_9 \vee A_2 \vee B_5) \wedge (\neg S_{10} \vee \neg B_6 \vee \neg B_4) \wedge$$
$$(\neg S_{11} \vee B_6 \vee \neg A_1) \wedge (\neg S_{12} \vee B_7)$$

(ii) The conflict analysis returns: $\neg S_1 \vee \neg S_2 \vee \neg S_3 \vee \neg S_4 \vee \neg S_5 \vee \neg S_6 \vee \neg S_8 \vee \neg S_{10} \vee \neg S_{11}$,

(iii) corresponding to the unsat core:

$$(B_0 \vee \neg B_1 \vee A_1) \wedge (B_0 \vee B_1 \vee A_2) \wedge (\neg B_0 \vee B_1 \vee A_2) \wedge$$
$$(\neg B_0 \vee \neg B_1) \wedge (\neg B_2 \vee \neg B_4) \wedge (\neg A_2 \vee B_2) \wedge$$
$$B_4 \wedge (\neg B_6 \vee \neg B_4) \wedge (B_6 \vee \neg A_1)$$

The assumption-based approach to core extraction

Example

$(B_0 \lor \neg B_1 \lor A_1) \land (B_0 \lor B_1 \lor A_2) \land (\neg B_0 \lor B_1 \lor A_2) \land$
$(\neg B_0 \lor \neg B_1) \land (\neg B_2 \lor \neg B_4) \land (\neg A_2 \lor B_2) \land (\neg A_1 \lor B_3) \land$
$B_4 \land (A_2 \lor B_5) \land (\neg B_6 \lor \neg B_4) \land (B_6 \lor \neg A_1) \land B_7$

(i) add selector variables:
$(\neg S_1 \lor B_0 \lor \neg B_1 \lor A_1) \land (\neg S_2 \lor B_0 \lor B_1 \lor A_2) \land (\neg S_3 \lor \neg B_0 \lor B_1 \lor A_2) \land$
$(\neg S_4 \lor \neg B_0 \lor \neg B_1) \land (\neg S_5 \lor \neg B_2 \lor \neg B_4) \land (\neg S_6 \lor \neg A_2 \lor B_2) \land$
$(\neg S_7 \lor \neg A_1 \lor B_3) \land (\neg S_8 \lor B_4) \land (\neg S_9 \lor A_2 \lor B_5) \land (\neg S_{10} \lor \neg B_6 \lor \neg B_4) \land$
$(\neg S_{11} \lor B_6 \lor \neg A_1) \land (\neg S_{12} \lor B_7)$

(ii) The conflict analysis returns: $\neg S_1 \lor \neg S_2 \lor \neg S_3 \lor \neg S_4 \lor \neg S_5 \lor \neg S_6 \lor \neg S_8 \lor \neg S_{10} \lor \neg S_{11}$,

(iii) corresponding to the unsat core:

$(B_0 \lor \neg B_1 \lor A_1) \land (B_0 \lor B_1 \lor A_2) \land (\neg B_0 \lor B_1 \lor A_2) \land$
$(\neg B_0 \lor \neg B_1) \land (\neg B_2 \lor \neg B_4) \land (\neg A_2 \lor B_2) \land$
$B_4 \land (\neg B_6 \lor \neg B_4) \land (B_6 \lor \neg A_1)$

The assumption-based approach to core extraction

Example

$(B_0 \vee \neg B_1 \vee A_1) \wedge (B_0 \vee B_1 \vee A_2) \wedge (\neg B_0 \vee B_1 \vee A_2) \wedge$
$(\neg B_0 \vee \neg B_1) \wedge (\neg B_2 \vee \neg B_4) \wedge (\neg A_2 \vee B_2) \wedge (\neg A_1 \vee B_3) \wedge$
$B_4 \wedge (A_2 \vee B_5) \wedge (\neg B_6 \vee \neg B_4) \wedge (B_6 \vee \neg A_1) \wedge B_7$

(i) add selector variables:
$(\neg S_1 \vee B_0 \vee \neg B_1 \vee A_1) \wedge (\neg S_2 \vee B_0 \vee B_1 \vee A_2) \wedge (\neg S_3 \vee \neg B_0 \vee B_1 \vee A_2) \wedge$
$(\neg S_4 \vee \neg B_0 \vee \neg B_1) \wedge (\neg S_5 \vee \neg B_2 \vee \neg B_4) \wedge (\neg S_6 \vee \neg A_2 \vee B_2) \wedge$
$(\neg S_7 \vee \neg A_1 \vee B_3) \wedge (\neg S_8 \vee B_4) \wedge (\neg S_9 \vee A_2 \vee B_5) \wedge (\neg S_{10} \vee \neg B_6 \vee \neg B_4) \wedge$
$(\neg S_{11} \vee B_6 \vee \neg A_1) \wedge (\neg S_{12} \vee B_7)$

(ii) The conflict analysis returns: $\neg S_1 \vee \neg S_2 \vee \neg S_3 \vee \neg S_4 \vee \neg S_5 \vee \neg S_6 \vee \neg S_8 \vee \neg S_{10} \vee \neg S_{11}$,

(iii) corresponding to the unsat core:

$(B_0 \vee \neg B_1 \vee A_1) \wedge (B_0 \vee B_1 \vee A_2) \wedge (\neg B_0 \vee B_1 \vee A_2) \wedge$
$(\neg B_0 \vee \neg B_1) \wedge (\neg B_2 \vee \neg B_4) \wedge (\neg A_2 \vee B_2) \wedge$
$B_4 \wedge (\neg B_6 \vee \neg B_4) \wedge (B_6 \vee \neg A_1)$

All-SAT (hints)

All-SAT & Projected All-SAT

- All-SAT: enumerate all truth assignments satisfying φ
  - Ex: $\varphi \stackrel{\text{def}}{=} (A_1 \vee \neg A_2) \wedge (\neg A_1 \vee A_2)$
    $\Longrightarrow AllSAT(\varphi) \; = \{\{A_1, A_2\}, \{\neg A_1, \neg A_2\}\}$
- Projected All-SAT: given an "important" subset $\mathbf{A} \stackrel{\text{def}}{=} \{A_i\}_i$ of $Atoms(\varphi) \stackrel{\text{def}}{=} \mathbf{A} \cup \mathbf{B}$, enumerate all assignments over $\mathbf{A}$ which can be extended to truth assignments on $\mathbf{B}$ satisfying φ
  - Equivalent to compute $AllSAT(\exists \mathbf{B}.\varphi)$
  - Ex: $\varphi \stackrel{\text{def}}{=} B_1 \wedge B_2 \wedge (B_1 \leftrightarrow (A_1 \vee \neg A_2)) \wedge (B_2 \leftrightarrow (\neg A_1 \vee A_2))$
    $\Longrightarrow AllSAT(\exists \mathbf{B}.\varphi) \; = \{\{A_1, A_2\}, \{\neg A_1, \neg A_2\}\}$

All-SAT (hints)

All-SAT & Projected All-SAT

- All-SAT: enumerate all truth assignments satisfying φ
  - Ex: $\varphi \stackrel{\text{def}}{=} (A_1 \vee \neg A_2) \wedge (\neg A_1 \vee A_2)$
    $\Longrightarrow AllSAT(\varphi) = \{\{A_1, A_2\}, \{\neg A_1, \neg A_2\}\}$
- Projected All-SAT: given an "important" subset $\mathbf{A} \stackrel{\text{def}}{=} \{A_i\}_i$ of $Atoms(\varphi) \stackrel{\text{def}}{=} \mathbf{A} \cup \mathbf{B}$, enumerate all assignments over $\mathbf{A}$ which can be extended to truth assignments on $\mathbf{B}$ satisfying φ
  - Equivalent to compute $AllSAT(\exists \mathbf{B}.\varphi)$
  - Ex: $\varphi \stackrel{\text{def}}{=} B_1 \wedge B_2 \wedge (B_1 \leftrightarrow (A_1 \vee \neg A_2)) \wedge (B_2 \leftrightarrow (\neg A_1 \vee A_2))$
    $\Longrightarrow AllSAT(\exists \mathbf{B}.\varphi) = \{\{A_1, A_2\}, \{\neg A_1, \neg A_2\}\}$

Model Counting, aka #SAT (hints)

Model Counting & Projected Model Counting

- #SAT: count all truth assignments on $\mathbf{A} \supseteq Atoms(\varphi)$ satisfying φ
  - Ex: $\varphi \stackrel{\text{def}}{=} (A_1 \vee \neg A_2) \wedge (\neg A_1 \vee A_2)$
    $\Longrightarrow \#SAT(\varphi) = 2$
- Projected #SAT: given an "important" subset $\mathbf{A} \stackrel{\text{def}}{=} \{A_i\}_i$ of $\mathbf{A} \cup \mathbf{B} \supseteq Atoms(\varphi)$,
  count all assignments over $\mathbf{A}$ which can be extended to truth assignments on $\mathbf{B}$ satisfying φ
  - Equivalent to compute $\#SAT(\exists \mathbf{B}.\varphi)$
  - Ex: $\varphi \stackrel{\text{def}}{=} B_1 \wedge B_2 \wedge (B_1 \leftrightarrow (A_1 \vee \neg A_2)) \wedge (B_2 \leftrightarrow (\neg A_1 \vee A_2))$
    $\Longrightarrow \#SAT(\exists \mathbf{B}.\varphi) = 2$

Model Counting, aka #SAT (hints)

Model Counting & Projected Model Counting

- #SAT: count all truth assignments on $\mathbf{A} \supseteq Atoms(\varphi)$ satisfying φ
  - Ex: $\varphi \stackrel{\text{def}}{=} (A_1 \vee \neg A_2) \wedge (\neg A_1 \vee A_2)$
    $\Longrightarrow \#SAT(\varphi) = 2$
- Projected #SAT: given an "important" subset $\mathbf{A} \stackrel{\text{def}}{=} \{A_i\}_i$ of $\mathbf{A} \cup \mathbf{B} \supseteq Atoms(\varphi)$, count all assignments over $\mathbf{A}$ which can be extended to truth assignments on $\mathbf{B}$ satisfying φ
  - Equivalent to compute $\#SAT(\exists \mathbf{B}.\varphi)$
  - Ex: $\varphi \stackrel{\text{def}}{=} B_1 \wedge B_2 \wedge (B_1 \leftrightarrow (A_1 \vee \neg A_2)) \wedge (B_2 \leftrightarrow (\neg A_1 \vee A_2))$
    $\Longrightarrow \#SAT(\exists \mathbf{B}.\varphi) = 2$

MaxSAT (hints)

- MaxSAT: given a pair of CNF formulas $\langle \varphi_h, \varphi_s \rangle$ s.t. $\varphi_h \wedge \varphi_s \models \bot$, $\varphi_s \stackrel{\text{def}}{=} \{C_1, ..., C_k\}$, find a truth assignment μ satisfying φ_h and maximizing the amount of the satisfied clauses in φ_s.
- Weighted MaxSAT: given also the positive integer penalties $\{w_1, ..., w_k\}$, μ must satisfy φ_h and maximize the sum of penalties of the satisfied clauses in φ_s
- Generalization of SAT to optimization

  $\Longrightarrow$ much harder than SAT
- Many different approaches (see e.g. [22])
- EX:

$$\varphi_h \stackrel{\text{def}}{=} (A_1 \vee A_2) \qquad \varphi_s \stackrel{\text{def}}{=} \left(\begin{array}{rcl} (A_1 \vee \neg A_2) & \wedge & [4] \\ (\neg A_1 \vee A_2) & \wedge & [3] \\ (\neg A_1 \vee \neg A_2) & \wedge & [2] \end{array} \right)$$

$\Longrightarrow \mu = \{A_1, A_2\}$ (penalty = 2)

MaxSAT (hints)

- MaxSAT: given a pair of CNF formulas $\langle \varphi_h, \varphi_s \rangle$ s.t. $\varphi_h \wedge \varphi_s \models \bot$, $\varphi_s \stackrel{\text{def}}{=} \{C_1, ..., C_k\}$, find a truth assignment μ satisfying φ_h and maximizing the amount of the satisfied clauses in φ_s.
- Weighted MaxSAT: given also the positive integer penalties $\{w_1, ..., w_k\}$, μ must satisfy φ_h and maximize the sum of penalties of the satisfied clauses in φ_s
- Generalization of SAT to optimization

  $\Longrightarrow$ much harder than SAT
- Many different approaches (see e.g. [22])
- EX:

$$\varphi_h \stackrel{\text{def}}{=} (A_1 \vee A_2) \qquad \varphi_s \stackrel{\text{def}}{=} \left(\begin{array}{cccc} (A_1 \vee \neg A_2) & \wedge & [4] \\ (\neg A_1 \vee A_2) & \wedge & [3] \\ (\neg A_1 \vee \neg A_2) & \wedge & [2] \end{array} \right)$$

$\Longrightarrow \mu = \{A_1, A_2\}$ (penalty = 2)

MaxSAT (hints)

- MaxSAT: given a pair of CNF formulas $\langle \varphi_h, \varphi_s \rangle$ s.t. $\varphi_h \wedge \varphi_s \models \perp$, $\varphi_s \stackrel{\text{def}}{=} \{C_1, ..., C_k\}$, find a truth assignment μ satisfying φ_h and maximizing the amount of the satisfied clauses in φ_s.
- Weighted MaxSAT: given also the positive integer penalties $\{w_1, ..., w_k\}$, μ must satisfy φ_h and maximize the sum of penalties of the satisfied clauses in φ_s
- Generalization of SAT to optimization

  $\Longrightarrow$ much harder than SAT
- Many different approaches (see e.g. [22])
- EX:

$$\varphi_h \stackrel{\text{def}}{=} (A_1 \vee A_2) \qquad \varphi_s \stackrel{\text{def}}{=} \left(\begin{array}{cccc} (A_1 \vee \neg A_2) & \wedge & [4] \\ (\neg A_1 \vee A_2) & \wedge & [3] \\ (\neg A_1 \vee \neg A_2) & \wedge & [2] \end{array} \right)$$

$\Longrightarrow \mu = \{A_1, A_2\}$ (penalty = 2)

MaxSAT (hints)

- MaxSAT: given a pair of CNF formulas $\langle\varphi_h, \varphi_s\rangle$ s.t. $\varphi_h \wedge \varphi_s \models \bot$, $\varphi_s \stackrel{\text{def}}{=} \{C_1, ..., C_k\}$, find a truth assignment μ satisfying φ_h and maximizing the amount of the satisfied clauses in φ_s.
- Weighted MaxSAT: given also the positive integer penalties $\{w_1, ..., w_k\}$, μ must satisfy φ_h and maximize the sum of penalties of the satisfied clauses in φ_s
- Generalization of SAT to optimization

  $\Longrightarrow$ much harder than SAT
- Many different approaches (see e.g. [22])
- EX:

$$\varphi_h \stackrel{\text{def}}{=} (A_1 \vee A_2) \qquad \varphi_s \stackrel{\text{def}}{=} \left(\begin{array}{ccc} (\ A_1 \vee \neg A_2) & \wedge & [4] \\ (\neg A_1 \vee \ A_2) & \wedge & [3] \\ (\neg A_1 \vee \neg A_2) & \wedge & [2] \end{array}\right)$$

$\Longrightarrow \mu = \{A_1, A_2\}$ (penalty = 2)

References I

A. Armando and E. Giunchiglia.
Embedding Complex Decision Procedures inside an Interactive Theorem Prover.
Annals of Mathematics and Artificial Intelligence, 8(3–4):475–502, 1993.

R. J. Bayardo, Jr. and R. C. Schrag.
Using CSP Look-Back Techniques to Solve Real-World SAT instances.
In *Proc. AAAI'97*, pages 203–208. AAAI Press, 1997.

A. Belov and Z. Stachniak.
Improving variable selection process in stochastic local search for propositional satisfiability.
In *SAT'09*, LNCS. Springer, 2009.

A. Belov and Z. Stachniak.
Improved local search for circuit satisfiability.
In *SAT*, volume 6175 of *LNCS*, pages 293–299. Springer, 2010.

A. Biere, M. J. H. Heule, H. van Maaren, and T. Walsh, editors.
Handbook of Satisfiability.
IOS Press, February 2009.

Booleforce, `http://fmv.jku.at/booleforce/`.

R. Brafman.
A simplifier for propositional formulas with many binary clauses.
In *Proc. IJCAI01*, 2001.

R. E. Bryant.
Graph-Based Algorithms for Boolean Function Manipulation.
IEEE Transactions on Computers, C-35(8):677–691, Aug. 1986.

References II

M. Davis, G. Longemann, and D. Loveland.
A machine program for theorem proving.
Journal of the ACM, 5(7), 1962.

M. Davis and H. Putnam.
A computing procedure for quantification theory.
Journal of the ACM, 7:201–215, 1960.

N. Eén and N. Sörensson.
Temporal induction by incremental sat solving.
Electr. Notes Theor. Comput. Sci., 89(4):543–560, 2003.

N. Eén and N. Sörensson.
An extensible SAT-solver.
In *Theory and Applications of Satisfiability Testing (SAT 2003)*, volume 2919 of *LNCS*, pages 502–518. Springer, 2004.

R. Gershman, M. Koifman, and O. Strichman.
Deriving Small Unsatisfiable Cores with Dominators.
In *Proc. CAV'06*, volume 4144 of *LNCS*. Springer, 2006.

E. Giunchiglia, M. Narizzano, A. Tacchella, and M. Vardi.
Towards an Efficient Library for SAT: a Manifesto.
In *Proc. SAT 2001*, Electronics Notes in Discrete Mathematics. Elsevier Science., 2001.

E. Giunchiglia and R. Sebastiani.
Applying the Davis-Putnam procedure to non-clausal formulas.
In *Proc. AI*IA'99*, volume 1792 of *LNAI*. Springer, 1999.

References III

C. Gomes, B. Selman, and H. Kautz.
Boosting Combinatorial Search Through Randomization.
In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, 1998.

C. P. Gomes, A. Sabharwal, and B. Selman.
Model Counting, chapter 20, pages 633–654.
In Biere et al. [5], February 2009.

H. H. Hoos and T. Stutzle.
Stochastic Local Search Foundation And Application.
Morgan Kaufmann, 2005.

J. Huang.
MUP: a minimal unsatisfiability prover.
In *Proc. ASP-DAC '05*. ACM Press, 2005.

H. A. Kautz, A. Sabharwal, and B. Selman.
Incomplete Algorithms, chapter 6, pages 185–203.
In Biere et al. [5], February 2009.

C. M. Li and Anbulagan.
Heuristics based on unit propagation for satisfiability problems.
In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 366–371, 1997.

C. M. Li and F. Manyà.
MaxSAT, Hard and Soft Constraints, chapter 19, pages 613–631.
In Biere et al. [5], February 2009.

References IV

I. Lynce and J. Marques-Silva.
On Computing Minimum Unsatisfiable Cores.
In *7th International Conference on Theory and Applications of Satisfiability Testing*, 2004.

I. Lynce and J. P. Marques-Silva.
On computing minimum unsatisfiable cores.
In *SAT*, 2004.

K. McMillan.
Interpolation and SAT-based model checking.
In *Proc. CAV*, 2003.

K. McMillan and N. Amla.
Automatic abstraction without counterexamples.
In *Proc. of TACAS*, 2003.

K. L. McMillan.
An interpolating theorem prover.
Theor. Comput. Sci., 345(1):101–121, 2005.

M. W. Moskewicz, C. F. Madigan, Y. Z., L. Zhang, and S. Malik.
Chaff: Engineering an efficient SAT solver.
In *Design Automation Conference*, 2001.

R. Nieuwenhuis, A. Oliveras, and C. Tinelli.
Abstract DPLL and abstract DPLL modulo theories.
In F. Baader and A. Voronkov, editors, *Proceedings of the 11th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR'04), Montevideo, Uruguay*, volume 3452 of *LNCS*, pages 36–50. Springer, 2005.

References V

R. Nieuwenhuis, A. Oliveras, and C. Tinelli.
Solving SAT and SAT Modulo Theories: from an Abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T).
Journal of the ACM, 53(6):937–977, November 2006.

Y. Oh, M. N. Mneimneh, Z. S. Andraus, K. A. Sakallah, and I. L. Markov.
Amuse: A Minimally-Unsatisfiable Subformula Extractor.
In *Proc. DAC'04*. ACM/IEEE, 2004.

P. Pudlák.
Lower bounds for resolution and cutting planes proofs and monotone computations.
J. of Symb. Logic, 62(3), 1997.

A. Robinson.
A machine-oriented logic based on the resolution principle.
Journal of the ACM, 12:23–41, 1965.

R. Sebastiani.
Applying GSAT to Non-Clausal Formulas.
Journal of Artificial Intelligence Research, 1:309–314, 1994.

B. Selman and H. Kautz.
Domain-Independent Extension to GSAT: Solving Large Structured Satisfiability Problems.
In *Proc. of the 13th International Joint Conference on Artificial Intelligence*, pages 290–295, 1993.

B. Selman, H. Kautz, and B. Cohen.
Local Search Strategies for Satisfiability Testing.
In *Cliques, Coloring, and Satisfiability*, volume 26 of *DIMACS*, pages 521–532, 1996.

References VI

B. Selman, H. Levesque., and D. Mitchell.
A New Method for Solving Hard Satisfiability Problems.
In *Proc. of the 10th National Conference on Artificial Intelligence*, pages 440–446, 1992.

J. P. M. Silva and K. A. Sakallah.
GRASP - A new Search Algorithm for Satisfiability.
In *Proc. ICCAD'96*, 1996.

R. M. Smullyan.
First-Order Logic.
Springer-Verlag, NY, 1968.

C. Tinelli.
A DPLL-based Calculus for Ground Satisfiability Modulo Theories.
In *Proc. JELIA-02*, volume 2424 of *LNAI*, pages 308–319. Springer, 2002.

D. Tompkins and H. Hoos.
UBCSAT: An Implementation and Experimentation Environment for SLS Algorithms for SAT and MAX-SAT.
In *SAT*, volume 3542 of *LNCS*. Springer, 2004.

H. Zhang and M. Stickel.
Implementing the Davis-Putnam algorithm by tries.
Technical report, University of Iowa, August 1994.

J. Zhang, S. Li, and S. Shen.
Extracting Minimum Unsatisfiable Cores with a Greedy Genetic Algorithm.
In *Proc. ACAI*, volume 4304 of *LNCS*. Springer, 2006.

References VII

L. Zhang, C. F. Madigan, M. H. Moskewicz, and S. Malik.
Efficient conflict driven learning in a boolean satisfiability solver.
In *ICCAD '01: Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, pages 279–285, Piscataway, NJ, USA, 2001. IEEE Press.

L. Zhang and S. Malik.
The quest for efficient boolean satisfiability solvers.
In *Proc. CAV'02*, number 2404 in LNCS, pages 17–36. Springer, 2002.

L. Zhang and S. Malik.
Extracting small unsatisfiable cores from unsatisfiable boolean formula.
In *Proc. of SAT*, 2003.

Disclaimer

The list of references above is by no means intended to be all-inclusive. The author of these slides apologizes both with the authors and with the readers for all the relevant works which are not cited here.

The papers (co)authored by the author of these slides are availlable at:
`https://disi.unitn.it/rseba/publist.html`.

Related web sites:

- Combination Methods in Automated Reasoning
  `https://combination.cs.uiowa.edu/`
- The SAT Association
  `https://satassociation.org/`
- SATLive! - Up-to-date links for SAT
  `https://www.satlive.org/index.jsp`
- SATLIB - The Satisfiability Library
  `https://www.intellektik.informatik.tu-darmstadt.de/SATLIB/`