Course “An Introduction to SAT and SMT”
Chapter 1: Propositional Satisfiability (SAT)

Roberto Sebastiani

DISI, Università di Trento, Italy – `roberto.sebastiani@unitn.it`
URL: `http://disi.unitn.it/rseba/DIDATTICA/SAT_SMT2020/`

Int. Graduate School on ICT, University of Trento,
Academic year 2019-2020

last update: Friday 22nd May, 2020

Outline

1. Basics on SAT
2. Basic SAT-Solving techniques
3. Modern CDCL SAT Solvers
   - Conflict-Driven Clause-Learning SAT solvers
   - Further Improvements
4. Tractable subclasses of SAT
5. Random k-SAT and Phase Transition
6. Advanced Functionalities: proofs, unsat cores, interpolants, optimization
7. Some Applications
   - Appl. #1: (Bounded) Planning
   - Appl. #2: Bounded Model Checking

Outline

1. Basics on SAT
2. Basic SAT-Solving techniques
3. Modern CDCL SAT Solvers
    - Conflict-Driven Clause-Learning SAT solvers
    - Further Improvements
4. Tractable subclasses of SAT
5. Random k-SAT and Phase Transition
6. Advanced Functionalities: proofs, unsat cores, interpolants, optimization
7. Some Applications
    - Appl. #1: (Bounded) Planning
    - Appl. #2: Bounded Model Checking

Boolean logic

Basic notation & definitions

- Boolean formula
  - $\top, \bot$ are formulas
  - A propositional atom $A_1, A_2, A_3, \ldots$ is a formula;
  - if φ_1 and φ_2 are formulas, then $\neg\varphi_1, \varphi_1 \wedge \varphi_2, \varphi_1 \vee \varphi_2, \varphi_1 \rightarrow \varphi_2, \varphi_1 \leftarrow \varphi_2, \varphi_1 \leftrightarrow \varphi_2$ are formulas.
- *Atoms*(φ): the set $\{A_1, \ldots, A_N\}$ of atoms occurring in φ.
- 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., $(A_1 \vee \neg A_2 \vee A_3 \vee \ldots)$)
- Cube: a conjunction of literals $\bigwedge_j l_j$ (e.g., $(A_1 \wedge \neg A_2 \wedge A_3 \wedge \ldots)$)

Semantics of Boolean operators

- Truth table:

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

Note

- $\wedge$, $\vee$ and $\leftrightarrow$ are commutative:

$$(\varphi_1 \wedge \varphi_2) \iff (\varphi_2 \wedge \varphi_1)$$
$$(\varphi_1 \vee \varphi_2) \iff (\varphi_2 \vee \varphi_1)$$
$$(\varphi_1 \leftrightarrow \varphi_2) \iff (\varphi_2 \leftrightarrow \varphi_1)$$

- $\wedge$ and $\vee$ are associative:

$$((\varphi_1 \wedge \varphi_2) \wedge \varphi_3) \iff (\varphi_1 \wedge (\varphi_2 \wedge \varphi_3)) \iff (\varphi_1 \wedge \varphi_2 \wedge \varphi_3)$$
$$((\varphi_1 \vee \varphi_2) \vee \varphi_3) \iff (\varphi_1 \vee (\varphi_2 \vee \varphi_3)) \iff (\varphi_1 \vee \varphi_2 \vee \varphi_3)$$

Semantics of Boolean operators

- Truth table:

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

Note

- $\wedge$, $\vee$ and $\leftrightarrow$ are commutative:

$$(\varphi_1 \wedge \varphi_2) \iff (\varphi_2 \wedge \varphi_1)$$
$$(\varphi_1 \vee \varphi_2) \iff (\varphi_2 \vee \varphi_1)$$
$$(\varphi_1 \leftrightarrow \varphi_2) \iff (\varphi_2 \leftrightarrow \varphi_1)$$

- $\wedge$ and $\vee$ are associative:

$$((\varphi_1 \wedge \varphi_2) \wedge \varphi_3) \iff (\varphi_1 \wedge (\varphi_2 \wedge \varphi_3)) \iff (\varphi_1 \wedge \varphi_2 \wedge \varphi_3)$$
$$((\varphi_1 \vee \varphi_2) \vee \varphi_3) \iff (\varphi_1 \vee (\varphi_2 \vee \varphi_3)) \iff (\varphi_1 \vee \varphi_2 \vee \varphi_3)$$

Syntactic Properties of Boolean Operators

$$
\begin{array}{rcl}
\neg\neg\varphi_1 & \Longleftrightarrow & \varphi_1 \\
(\varphi_1 \vee \varphi_2) & \Longleftrightarrow & \neg(\neg\varphi_1 \wedge \neg\varphi_2) \\
\neg(\varphi_1 \vee \varphi_2) & \Longleftrightarrow & (\neg\varphi_1 \wedge \neg\varphi_2) \\
(\varphi_1 \wedge \varphi_2) & \Longleftrightarrow & \neg(\neg\varphi_1 \vee \neg\varphi_2) \\
\neg(\varphi_1 \wedge \varphi_2) & \Longleftrightarrow & (\neg\varphi_1 \vee \neg\varphi_2) \\
(\varphi_1 \rightarrow \varphi_2) & \Longleftrightarrow & (\neg\varphi_1 \vee \varphi_2) \\
\neg(\varphi_1 \rightarrow \varphi_2) & \Longleftrightarrow & (\varphi_1 \wedge \neg\varphi_2) \\
(\varphi_1 \leftarrow \varphi_2) & \Longleftrightarrow & (\varphi_1 \vee \neg\varphi_2) \\
\neg(\varphi_1 \leftarrow \varphi_2) & \Longleftrightarrow & (\neg\varphi_1 \wedge \varphi_2) \\
(\varphi_1 \leftrightarrow \varphi_2) & \Longleftrightarrow & ((\varphi_1 \rightarrow \varphi_2) \wedge (\varphi_1 \leftarrow \varphi_2)) \\
 & \Longleftrightarrow & ((\neg\varphi_1 \vee \varphi_2) \wedge (\varphi_1 \vee \neg\varphi_2)) \\
\neg(\varphi_1 \leftrightarrow \varphi_2) & \Longleftrightarrow & (\neg\varphi_1 \leftrightarrow \varphi_2) \\
 & \Longleftrightarrow & (\varphi_1 \leftrightarrow \neg\varphi_2) \\
 & \Longleftrightarrow & ((\varphi_1 \vee \varphi_2) \wedge (\neg\varphi_1 \vee \neg\varphi_2))
\end{array}
$$

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

Syntactic Properties of Boolean Operators

$$
\begin{array}{rcl}
\neg\neg\varphi_1 & \Longleftrightarrow & \varphi_1 \\
(\varphi_1 \vee \varphi_2) & \Longleftrightarrow & \neg(\neg\varphi_1 \wedge \neg\varphi_2) \\
\neg(\varphi_1 \vee \varphi_2) & \Longleftrightarrow & (\neg\varphi_1 \wedge \neg\varphi_2) \\
(\varphi_1 \wedge \varphi_2) & \Longleftrightarrow & \neg(\neg\varphi_1 \vee \neg\varphi_2) \\
\neg(\varphi_1 \wedge \varphi_2) & \Longleftrightarrow & (\neg\varphi_1 \vee \neg\varphi_2) \\
(\varphi_1 \rightarrow \varphi_2) & \Longleftrightarrow & (\neg\varphi_1 \vee \varphi_2) \\
\neg(\varphi_1 \rightarrow \varphi_2) & \Longleftrightarrow & (\varphi_1 \wedge \neg\varphi_2) \\
(\varphi_1 \leftarrow \varphi_2) & \Longleftrightarrow & (\varphi_1 \vee \neg\varphi_2) \\
\neg(\varphi_1 \leftarrow \varphi_2) & \Longleftrightarrow & (\neg\varphi_1 \wedge \varphi_2) \\
(\varphi_1 \leftrightarrow \varphi_2) & \Longleftrightarrow & ((\varphi_1 \rightarrow \varphi_2) \wedge (\varphi_1 \leftarrow \varphi_2)) \\
 & \Longleftrightarrow & ((\neg\varphi_1 \vee \varphi_2) \wedge (\varphi_1 \vee \neg\varphi_2)) \\
\neg(\varphi_1 \leftrightarrow \varphi_2) & \Longleftrightarrow & (\neg\varphi_1 \leftrightarrow \varphi_2) \\
 & \Longleftrightarrow & (\varphi_1 \leftrightarrow \neg\varphi_2) \\
 & \Longleftrightarrow & ((\varphi_1 \vee \varphi_2) \wedge (\neg\varphi_1 \vee \neg\varphi_2))
\end{array}
$$

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

Tree and DAG representation of formulas: example

Formulas can be represented either as trees or as DAGS:

- DAG representation can be up to exponentially smaller

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

Tree and DAG representation of formulas: example

Formulas can be represented either as trees or as DAGS:

- DAG representation can be up to exponentially smaller

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

$$\Downarrow$$

$$\begin{array}{c}(((A_1 \leftrightarrow A_2) \rightarrow (A_3 \leftrightarrow A_4)) \wedge \\ ((A_3 \leftrightarrow A_4) \rightarrow (A_1 \leftrightarrow A_2)))\end{array}$$

Tree and DAG representation of formulas: example

Formulas can be represented either as trees or as DAGS:

- DAG representation can be up to exponentially smaller

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

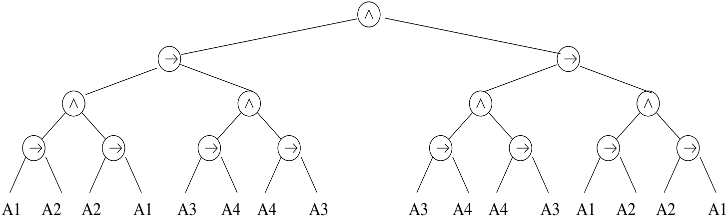
$$\Downarrow$$

$$\begin{gathered}(((A_1 \leftrightarrow A_2) \rightarrow (A_3 \leftrightarrow A_4))\wedge \\ ((A_3 \leftrightarrow A_4) \rightarrow (A_1 \leftrightarrow A_2)))\end{gathered}$$

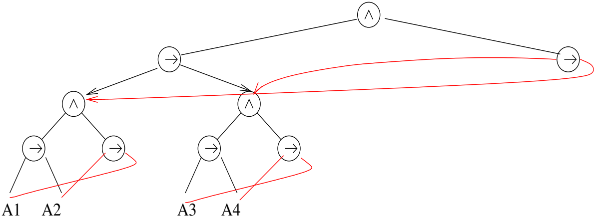
$$\Downarrow$$

$$\begin{gathered}(((A_1 \rightarrow A_2) \wedge (A_2 \rightarrow A_1)) \rightarrow ((A_3 \rightarrow A_4) \wedge (A_4 \rightarrow A_3)))\wedge \\ (((A_3 \rightarrow A_4) \wedge (A_4 \rightarrow A_3)) \rightarrow (((A_1 \rightarrow A_2) \wedge (A_2 \rightarrow A_1))))\end{gathered}$$

Tree and DAG repres. of formulas: example (cont)



Tree Representation



DAG Representation

Basic notation & definitions (cont)

- Total truth assignment μ for φ:
  $\mu : \mathit{Atoms}(\varphi) \longmapsto \{\top, \bot\}$.
- Partial Truth assignment μ for φ:
  $\mu : \mathcal{A} \longmapsto \{\top, \bot\}$, $\mathcal{A} \subset \mathit{Atoms}(\varphi)$.
- Set and formula representation 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: $\{\mu(A_1) := \top, \mu(A_2) := \bot\} \implies (A_1 \wedge \neg A_2)$

Basic notation & definitions (cont)

- a total truth assignment μ satisfies φ ($\mu \models \varphi$):
  - $\mu \models A_i \iff \mu(A_i) = \top$
  - $\mu \models \neg\varphi \iff$ *not* $\mu \models \varphi$
  - $\mu \models \varphi_1 \wedge \varphi_2 \iff \mu \models \varphi_1$ *and* $\mu \models \varphi_2$
  - $\mu \models \varphi_1 \vee \varphi_2 \iff \mu \models \varphi_1$ *or* $\mu \models \varphi_2$
  - $\mu \models \varphi_1 \rightarrow \varphi_2 \iff$ *if* $\mu \models \varphi_1$, *then* $\mu \models \varphi_2$
  - $\mu \models \varphi_1 \leftrightarrow \varphi_2 \iff \mu \models \varphi_1$ *iff* $\mu \models \varphi_2$
- a partial truth assignment μ satisfies φ iff it makes φ evaluate to true (Ex: $\{A_1\} \models (A_1 \vee A_2)$)
  $\Longrightarrow$ if μ satisfies φ, then all its total extensions satisfy φ
  (Ex: $\{A_1, A_2\} \models (A_1 \vee A_2)$ and $\{A_1, \neg A_2\} \models (A_1 \vee A_2)$)
- φ is satisfiable iff $\mu \models \varphi$ for some μ
- φ_1 entails φ_2 ($\varphi_1 \models \varphi_2$): $\varphi_1 \models \varphi_2$ iff $\mu \models \varphi_1 \Longrightarrow \mu \models \varphi_2$ for every μ
- φ is valid ($\models \varphi$): $\models \varphi$ iff $\mu \models \varphi$ for every μ

Property

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

Basic notation & definitions (cont)

- a total truth assignment μ satisfies φ ($\mu \models \varphi$):
  - $\mu \models A_i \iff \mu(A_i) = \top$
  - $\mu \models \neg\varphi \iff$ *not* $\mu \models \varphi$
  - $\mu \models \varphi_1 \wedge \varphi_2 \iff \mu \models \varphi_1$ *and* $\mu \models \varphi_2$
  - $\mu \models \varphi_1 \vee \varphi_2 \iff \mu \models \varphi_1$ *or* $\mu \models \varphi_2$
  - $\mu \models \varphi_1 \rightarrow \varphi_2 \iff$ *if* $\mu \models \varphi_1$, *then* $\mu \models \varphi_2$
  - $\mu \models \varphi_1 \leftrightarrow \varphi_2 \iff \mu \models \varphi_1$ *iff* $\mu \models \varphi_2$
- a partial truth assignment μ satisfies φ iff it makes φ evaluate to true (Ex: $\{A_1\} \models (A_1 \vee A_2)$)

  $\Longrightarrow$ if μ satisfies φ, then all its total extensions satisfy φ (Ex: $\{A_1, A_2\} \models (A_1 \vee A_2)$ and $\{A_1, \neg A_2\} \models (A_1 \vee A_2)$)
- φ is satisfiable iff $\mu \models \varphi$ for some μ
- φ_1 entails φ_2 ($\varphi_1 \models \varphi_2$): $\varphi_1 \models \varphi_2$ iff $\mu \models \varphi_1 \Longrightarrow \mu \models \varphi_2$ for every μ
- φ is valid ($\models \varphi$): $\models \varphi$ iff $\mu \models \varphi$ for every μ

Property

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

Basic notation & definitions (cont)

- a total truth assignment μ satisfies φ ($\mu \models \varphi$):
  - $\mu \models A_i \iff \mu(A_i) = \top$
  - $\mu \models \neg\varphi \iff \textit{not } \mu \models \varphi$
  - $\mu \models \varphi_1 \wedge \varphi_2 \iff \mu \models \varphi_1 \textit{ and } \mu \models \varphi_2$
  - $\mu \models \varphi_1 \vee \varphi_2 \iff \mu \models \varphi_1 \textit{ or } \mu \models \varphi_2$
  - $\mu \models \varphi_1 \rightarrow \varphi_2 \iff \textit{ if } \mu \models \varphi_1, \textit{ then } \mu \models \varphi_2$
  - $\mu \models \varphi_1 \leftrightarrow \varphi_2 \iff \mu \models \varphi_1 \textit{ iff } \mu \models \varphi_2$
- a partial truth assignment μ satisfies φ iff it makes φ evaluate to true (Ex: $\{A_1\} \models (A_1 \vee A_2)$)

  $\Longrightarrow$ if μ satisfies φ, then all its total extensions satisfy φ (Ex: $\{A_1, A_2\} \models (A_1 \vee A_2)$ and $\{A_1, \neg A_2\} \models (A_1 \vee A_2)$)
- φ is satisfiable iff $\mu \models \varphi$ for some μ
- φ_1 entails φ_2 ($\varphi_1 \models \varphi_2$): $\varphi_1 \models \varphi_2$ iff $\mu \models \varphi_1 \Longrightarrow \mu \models \varphi_2$ for every μ
- φ is valid ($\models \varphi$): $\models \varphi$ iff $\mu \models \varphi$ for every μ

Property

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

Basic notation & definitions (cont)

- a total truth assignment μ satisfies φ ($\mu \models \varphi$):
  - $\mu \models A_i \iff \mu(A_i) = \top$
  - $\mu \models \neg\varphi \iff \textit{not}\ \mu \models \varphi$
  - $\mu \models \varphi_1 \wedge \varphi_2 \iff \mu \models \varphi_1\ \textit{and}\ \mu \models \varphi_2$
  - $\mu \models \varphi_1 \vee \varphi_2 \iff \mu \models \varphi_1\ \textit{or}\ \mu \models \varphi_2$
  - $\mu \models \varphi_1 \rightarrow \varphi_2 \iff\ \textit{if}\ \mu \models \varphi_1,\ \textit{then}\ \mu \models \varphi_2$
  - $\mu \models \varphi_1 \leftrightarrow \varphi_2 \iff \mu \models \varphi_1\ \textit{iff}\ \mu \models \varphi_2$
- a partial truth assignment μ satisfies φ iff it makes φ evaluate to true (Ex: $\{A_1\} \models (A_1 \vee A_2)$)

  $\Longrightarrow$ if μ satisfies φ, then all its total extensions satisfy φ
  (Ex: $\{A_1, A_2\} \models (A_1 \vee A_2)$ and $\{A_1, \neg A_2\} \models (A_1 \vee A_2)$)
- φ is satisfiable iff $\mu \models \varphi$ for some μ
- φ_1 entails φ_2 ($\varphi_1 \models \varphi_2$): $\varphi_1 \models \varphi_2$ iff $\mu \models \varphi_1 \Longrightarrow \mu \models \varphi_2$ for every μ
- φ is valid ($\models \varphi$): $\models \varphi$ iff $\mu \models \varphi$ for every μ

Property

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

Basic notation & definitions (cont)

- a total truth assignment μ satisfies φ ($\mu \models \varphi$):
  - $\mu \models A_i \Longleftrightarrow \mu(A_i) = \top$
  - $\mu \models \neg\varphi \Longleftrightarrow$ *not* $\mu \models \varphi$
  - $\mu \models \varphi_1 \wedge \varphi_2 \Longleftrightarrow \mu \models \varphi_1$ *and* $\mu \models \varphi_2$
  - $\mu \models \varphi_1 \vee \varphi_2 \Longleftrightarrow \mu \models \varphi_1$ *or* $\mu \models \varphi_2$
  - $\mu \models \varphi_1 \rightarrow \varphi_2 \Longleftrightarrow$ *if* $\mu \models \varphi_1,$ *then* $\mu \models \varphi_2$
  - $\mu \models \varphi_1 \leftrightarrow \varphi_2 \Longleftrightarrow \mu \models \varphi_1$ *iff* $\mu \models \varphi_2$
- a partial truth assignment μ satisfies φ iff it makes φ evaluate to true (Ex: $\{A_1\} \models (A_1 \vee A_2)$)

  $\Longrightarrow$ if μ satisfies φ, then all its total extensions satisfy φ (Ex: $\{A_1, A_2\} \models (A_1 \vee A_2)$ and $\{A_1, \neg A_2\} \models (A_1 \vee A_2)$)
- φ is satisfiable iff $\mu \models \varphi$ for some μ
- φ_1 entails φ_2 ($\varphi_1 \models \varphi_2$): $\varphi_1 \models \varphi_2$ iff $\mu \models \varphi_1 \Longrightarrow \mu \models \varphi_2$ for every μ
- φ is valid ($\models \varphi$): $\models \varphi$ iff $\mu \models \varphi$ for every μ

Property

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

Basic notation & definitions (cont)

- a total truth assignment μ satisfies φ ($\mu \models \varphi$):
  - $\mu \models A_i \iff \mu(A_i) = \top$
  - $\mu \models \neg\varphi \iff \textit{not}\ \mu \models \varphi$
  - $\mu \models \varphi_1 \wedge \varphi_2 \iff \mu \models \varphi_1\ \textit{and}\ \mu \models \varphi_2$
  - $\mu \models \varphi_1 \vee \varphi_2 \iff \mu \models \varphi_1\ \textit{or}\ \mu \models \varphi_2$
  - $\mu \models \varphi_1 \rightarrow \varphi_2 \iff\ \textit{if}\ \mu \models \varphi_1,\ \textit{then}\ \mu \models \varphi_2$
  - $\mu \models \varphi_1 \leftrightarrow \varphi_2 \iff \mu \models \varphi_1\ \textit{iff}\ \mu \models \varphi_2$
- a partial truth assignment μ satisfies φ iff it makes φ evaluate to true (Ex: $\{A_1\} \models (A_1 \vee A_2)$)

  $\Longrightarrow$ if μ satisfies φ, then all its total extensions satisfy φ (Ex: $\{A_1, A_2\} \models (A_1 \vee A_2)$ and $\{A_1, \neg A_2\} \models (A_1 \vee A_2)$)
- φ is satisfiable iff $\mu \models \varphi$ for some μ
- φ_1 entails φ_2 ($\varphi_1 \models \varphi_2$): $\varphi_1 \models \varphi_2$ iff $\mu \models \varphi_1 \Longrightarrow \mu \models \varphi_2$ for every μ
- φ is valid ($\models \varphi$): $\models \varphi$ iff $\mu \models \varphi$ for every μ

Property

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

Equivalence and equi-satisfiability

- φ_1 and φ_2 are equivalent iff, for every μ, $\mu \models \varphi_1$ iff $\mu \models \varphi_2$
- φ_1 and φ_2 are equi-satisfiable iff exists μ_1 s.t. $\mu_1 \models \varphi_1$ iff exists μ_2 s.t. $\mu_2 \models \varphi_2$
- φ_1, φ_2 equivalent
  $\Downarrow \quad \not\Uparrow$
  φ_1, φ_2 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 φ_2 is the result of applying some transformation T to φ_1: $\varphi_2 \stackrel{\text{def}}{=} T(\varphi_1)$:
  we say that T is validity-preserving [satisfiability-preserving] iff $T(\varphi_1)$ and φ_1 are equivalent [equi-satisfiable]

Equivalence and equi-satisfiability

- φ_1 and φ_2 are equivalent iff, for every μ, $\mu \models \varphi_1$ iff $\mu \models \varphi_2$
- φ_1 and φ_2 are equi-satisfiable iff exists μ_1 s.t. $\mu_1 \models \varphi_1$ iff exists μ_2 s.t. $\mu_2 \models \varphi_2$
- φ_1, φ_2 equivalent
  $\Downarrow \quad \not\Uparrow$
  φ_1, φ_2 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 φ_2 is the result of applying some transformation T to φ_1: $\varphi_2 \stackrel{\text{def}}{=} T(\varphi_1)$:
  we say that T is validity-preserving [satisfiability-preserving] iff $T(\varphi_1)$ and φ_1 are equivalent [equi-satisfiable]

Equivalence and equi-satisfiability

- φ_1 and φ_2 are equivalent iff, for every μ,
  $\mu \models \varphi_1$ iff $\mu \models \varphi_2$
- φ_1 and φ_2 are equi-satisfiable iff
  exists μ_1 s.t. $\mu_1 \models \varphi_1$ iff exists μ_2 s.t. $\mu_2 \models \varphi_2$
- φ_1, φ_2 equivalent
  $\Downarrow \quad \not\Uparrow$
  φ_1, φ_2 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 φ_2 is the result of applying some transformation T to φ_1: $\varphi_2 \stackrel{\text{def}}{=} T(\varphi_1)$:
  we say that T is validity-preserving [satisfiability-preserving] iff $T(\varphi_1)$ and φ_1 are equivalent [equi-satisfiable]

Equivalence and equi-satisfiability

- φ_1 and φ_2 are equivalent iff, for every μ, $\mu \models \varphi_1$ iff $\mu \models \varphi_2$
- φ_1 and φ_2 are equi-satisfiable iff exists μ_1 s.t. $\mu_1 \models \varphi_1$ iff exists μ_2 s.t. $\mu_2 \models \varphi_2$
- φ_1, φ_2 equivalent

  $\Downarrow \quad \not\Uparrow$

  φ_1, φ_2 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 φ_2 is the result of applying some transformation T to φ_1: $\varphi_2 \stackrel{\text{def}}{=} T(\varphi_1)$:
  we say that T is validity-preserving [satisfiability-preserving] iff $T(\varphi_1)$ and φ_1 are equivalent [equi-satisfiable]

Equivalence and equi-satisfiability

- φ_1 and φ_2 are equivalent iff, for every μ, $\mu \models \varphi_1$ iff $\mu \models \varphi_2$
- φ_1 and φ_2 are equi-satisfiable iff exists μ_1 s.t. $\mu_1 \models \varphi_1$ iff exists μ_2 s.t. $\mu_2 \models \varphi_2$
- φ_1, φ_2 equivalent
  $\Downarrow \quad \not\Uparrow$
  φ_1, φ_2 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 φ_2 is the result of applying some transformation T to φ_1: $\varphi_2 \stackrel{\text{def}}{=} T(\varphi_1)$:
  we say that T is validity-preserving [satisfiability-preserving] iff $T(\varphi_1)$ and φ_1 are equivalent [equi-satisfiable]

Complexity

- 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
- The most important logical problems (validity, inference, entailment, equivalence, ...) can be straightforwardly reduced to 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 $\varphi_1 \leftrightarrow \varphi_2$ 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:

$$\begin{array}{lcl} \varphi_1 \rightarrow \varphi_2 & \Longrightarrow & \neg\varphi_1 \vee \varphi_2 \\ \varphi_1 \leftrightarrow \varphi_2 & \Longrightarrow & (\neg\varphi_1 \vee \varphi_2) \wedge (\varphi_1 \vee \neg\varphi_2) \end{array}$$

  (ii) pushing down negations recursively:

$$\begin{array}{lcl} \neg(\varphi_1 \wedge \varphi_2) & \Longrightarrow & \neg\varphi_1 \vee \neg\varphi_2 \\ \neg(\varphi_1 \vee \varphi_2) & \Longrightarrow & \neg\varphi_1 \wedge \neg\varphi_2 \\ \neg\neg\varphi_1 & \Longrightarrow & \varphi_1 \end{array}$$

- 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

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

$$\Downarrow$$

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

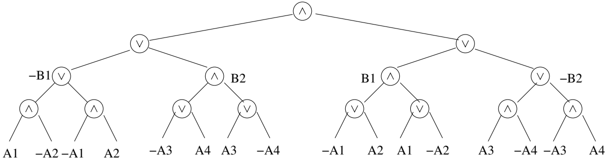
NNF: example

$$
\begin{gathered}
(A_1 \leftrightarrow A_2) \leftrightarrow (A_3 \leftrightarrow A_4) \\
\Downarrow \\
((((A_1 \rightarrow A_2) \wedge (A_1 \leftarrow A_2)) \rightarrow ((A_3 \rightarrow A_4) \wedge (A_3 \leftarrow A_4)))\wedge \\
(((A_1 \rightarrow A_2) \wedge (A_1 \leftarrow A_2)) \leftarrow ((A_3 \rightarrow A_4) \wedge (A_3 \leftarrow A_4)))) \\
\Downarrow \\
((\neg((\neg A_1 \vee A_2) \wedge (A_1 \vee \neg A_2)) \vee ((\neg A_3 \vee A_4) \wedge (A_3 \vee \neg A_4)))\wedge \\
(((\neg A_1 \vee A_2) \wedge (A_1 \vee \neg A_2)) \vee \neg((\neg A_3 \vee A_4) \wedge (A_3 \vee \neg A_4))))
\end{gathered}
$$

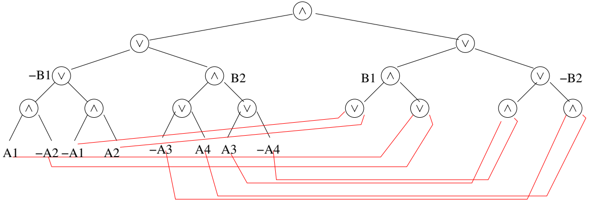
NNF: example

$$
\begin{array}{c}
(A_1 \leftrightarrow A_2) \leftrightarrow (A_3 \leftrightarrow A_4) \\
\Downarrow \\
((((A_1 \rightarrow A_2) \wedge (A_1 \leftarrow A_2)) \rightarrow ((A_3 \rightarrow A_4) \wedge (A_3 \leftarrow A_4)))\wedge \\
(((A_1 \rightarrow A_2) \wedge (A_1 \leftarrow A_2)) \leftarrow ((A_3 \rightarrow A_4) \wedge (A_3 \leftarrow A_4)))) \\
\Downarrow \\
((\neg((\neg A_1 \vee A_2) \wedge (A_1 \vee \neg A_2)) \vee ((\neg A_3 \vee A_4) \wedge (A_3 \vee \neg A_4)))\wedge \\
(((\neg A_1 \vee A_2) \wedge (A_1 \vee \neg A_2)) \vee \neg((\neg A_3 \vee A_4) \wedge (A_3 \vee \neg A_4)))) \\
\Downarrow \\
((((A_1 \wedge \neg A_2) \vee (\neg A_1 \wedge A_2)) \vee ((\neg A_3 \vee A_4) \wedge (A_3 \vee \neg A_4)))\wedge \\
(((\neg A_1 \vee A_2) \wedge (A_1 \vee \neg A_2)) \vee ((A_3 \wedge \neg A_4) \vee (\neg A_3 \wedge A_4))))
\end{array}
$$

NNF: example (cont)



Tree Representation

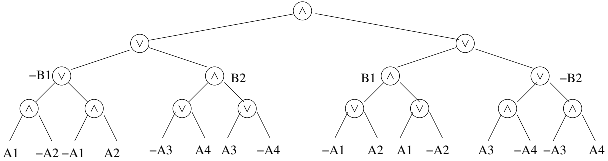


DAG Representation

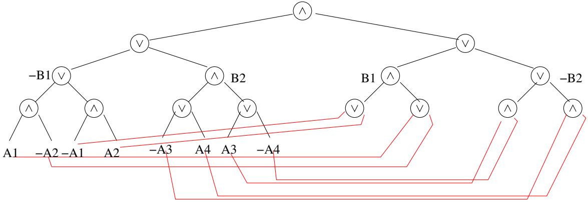
Note

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

NNF: example (cont)



Tree Representation



DAG Representation

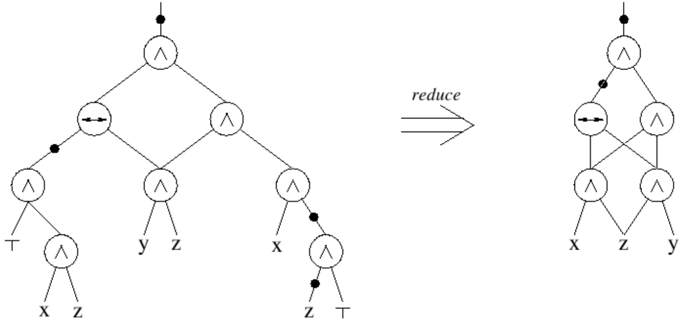
Note

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) converting it into NNF (not indispensible);
  (ii) applying recursively the DeMorgan's Rule:
  $$(\varphi_1 \wedge \varphi_2) \vee \varphi_3 \implies (\varphi_1 \vee \varphi_3) \wedge (\varphi_2 \vee \varphi_3)$$
- Worst-case exponential.
- $Atoms(CNF(\varphi)) = Atoms(\varphi)$.
- $CNF(\varphi)$ is equivalent to φ.
- Rarely used in practice.

Classic CNF Conversion $CNF(\varphi)$

- Every φ can be reduced into CNF by, e.g.,
  (i) converting it into NNF (not indispensible);
  (ii) applying recursively the DeMorgan's Rule:
  $$(\varphi_1 \wedge \varphi_2) \vee \varphi_3 \implies (\varphi_1 \vee \varphi_3) \wedge (\varphi_2 \vee \varphi_3)$$
- Worst-case exponential.
- $Atoms(CNF(\varphi)) = Atoms(\varphi)$.
- $CNF(\varphi)$ is equivalent to φ.
- Rarely used in practice.

Classic CNF Conversion $CNF(\varphi)$

- Every φ can be reduced into CNF by, e.g.,
  - (i) converting it into NNF (not indispensible);
  - (ii) applying recursively the DeMorgan's Rule:

$$(\varphi_1 \wedge \varphi_2) \vee \varphi_3 \quad \Longrightarrow \quad (\varphi_1 \vee \varphi_3) \wedge (\varphi_2 \vee \varphi_3)$$

- Worst-case exponential.
- $Atoms(CNF(\varphi)) = Atoms(\varphi)$.
- $CNF(\varphi)$ is equivalent to φ.
- Rarely used in practice.

Classic CNF Conversion $CNF(\varphi)$

- Every φ can be reduced into CNF by, e.g.,
  (i) converting it into NNF (not indispensible);
  (ii) applying recursively the DeMorgan's Rule:
  $$(\varphi_1 \wedge \varphi_2) \vee \varphi_3 \quad \Longrightarrow \quad (\varphi_1 \vee \varphi_3) \wedge (\varphi_2 \vee \varphi_3)$$
- Worst-case exponential.
- $Atoms(CNF(\varphi)) = Atoms(\varphi)$.
- $CNF(\varphi)$ is equivalent to φ.
- Rarely used in practice.

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

- 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 w.r.t. φ.
- More used in practice.

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

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

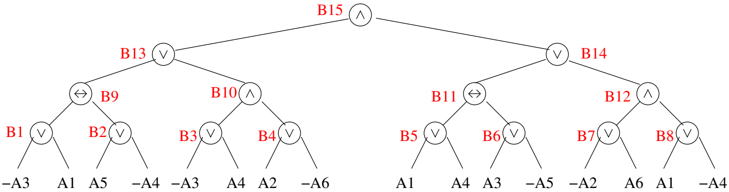
$$\begin{array}{lcl} \varphi & \Longrightarrow & \varphi[(l_i \vee l_j)|B] \wedge CNF(B \leftrightarrow (l_i \vee l_j)) \\ \varphi & \Longrightarrow & \varphi[(l_i \wedge l_j)|B] \wedge CNF(B \leftrightarrow (l_i \wedge l_j)) \\ \varphi & \Longrightarrow & \varphi[(l_i \leftrightarrow l_j)|B] \wedge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

  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 w.r.t. φ.
- More used in practice.

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

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

Labeling CNF conversion CNF_{label} – example



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

Labeling CNF conversion CNF_{label} (improved)

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

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

- Smaller in size:

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

Labeling CNF conversion CNF_{label} (improved)

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

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

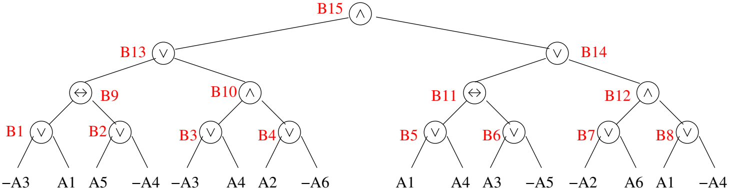
- Smaller in size:

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

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

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

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 optimizations

- 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)
- ...

Outline

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.

Resolution [49, 15]

- Search for a refutation of φ
- φ is represented as a set of clauses
- Applies iteratively the resolution rule to pairs of clauses containing a conflicting literal, until a false clause is generated or the resolution rule is no more applicable
- Many different strategies

Resolution Rule

- Resolution of a pair of clauses with exactly one incompatible variable:

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

- EXAMPLE:

$$\frac{(A \vee B \vee C \vee D \vee E) \qquad (A \vee B \vee \neg C \vee F)}{(A \vee B \vee D \vee E \vee F)}$$

- NOTE: many standard inference rules subcases of resolution:

$$\frac{A \rightarrow B \quad B \rightarrow C}{A \rightarrow C}\ (Transit.) \qquad \frac{A \quad A \rightarrow B}{B}\ (M.\ Ponens) \qquad \frac{\neg B \quad A \rightarrow B}{\neg A}$$

Resolution Rules [15, 14]: unit propagation

- 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!

Resolution: basic strategy [15]

```
function DP(Γ)
    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(Γ)
    A := select-variable(Γ);                     /* 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(Γ)
```

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

Resolution: Examples

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

$$\Downarrow$$

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

$$\Downarrow$$

$$\bot$$

$\Longrightarrow$ UNSAT

Resolution: Examples

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

$$\Downarrow$$

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

$$\Downarrow$$

$$\bot$$

$\Longrightarrow$ UNSAT

Resolution: Examples

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

$\Longrightarrow$ UNSAT

Resolution: Examples

$$
\begin{array}{c}
(A_1 \lor A_2)\ \ (A_1 \lor \neg A_2)\ \ (\neg A_1 \lor A_2)\ \ (\neg A_1 \lor \neg A_2) \\
\Downarrow \\
(A_2)\ \ (A_2 \lor \neg A_2)\ \ (\neg A_2 \lor A_2)\ \ (\neg A_2) \\
\Downarrow \\
\bot
\end{array}
$$

$\Longrightarrow$ 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.)

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

$$\Downarrow$$

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

$$\Downarrow$$

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

$\Longrightarrow$ SAT

Resolution: Examples (cont.)

$$
\begin{array}{c}
(A \vee B \vee C) \;\; (B \vee \neg C \vee \neg F) \;\; (\neg B \vee E) \\
\Downarrow \\
(A \vee C \vee E) \;\; (\neg C \vee \neg F \vee E) \\
\Downarrow \\
(A \vee E \vee \neg F)
\end{array}
$$

$\Longrightarrow$ SAT

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

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

$\Longrightarrow$ UNSAT

Resolution: Examples

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

$$\Downarrow$$

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

$$\Downarrow$$

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

$$\Downarrow$$

$$\bot$$

$\Longrightarrow$ UNSAT

Resolution: Examples

$$
\begin{array}{c}
(A \vee B)\ \ (A \vee \neg B)\ \ (\neg A \vee C)\ \ (\neg A \vee \neg C) \\
\Downarrow \\
(A)\ \ (\neg A \vee C)\ \ (\neg A \vee \neg C) \\
\Downarrow \\
(C)\ \ (\neg C) \\
\Downarrow \\
\bot
\end{array}
$$

$\Longrightarrow$ 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)

Semantic tableaux [55]

- 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

$$\frac{\varphi_1 \wedge \varphi_2}{\begin{array}{c}\varphi_1\\ \varphi_2\end{array}} \quad \frac{\neg(\varphi_1 \vee \varphi_2)}{\begin{array}{c}\neg\varphi_1\\ \neg\varphi_2\end{array}} \quad \frac{\neg(\varphi_1 \to \varphi_2)}{\begin{array}{c}\varphi_1\\ \neg\varphi_2\end{array}} \qquad (\wedge\text{-elimination})$$

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

$$\frac{\varphi_1 \vee \varphi_2}{\varphi_1 \qquad \varphi_2} \quad \frac{\neg(\varphi_1 \wedge \varphi_2)}{\neg\varphi_1 \qquad \neg\varphi_2} \quad \frac{\varphi_1 \to \varphi_2}{\neg\varphi_1 \qquad \varphi_2} \qquad (\vee\text{-elimination})$$

$$\frac{\varphi_1 \leftrightarrow \varphi_2}{\begin{array}{cc}\varphi_1 & \neg\varphi_1\\ \varphi_2 & \neg\varphi_2\end{array}} \quad \frac{\neg(\varphi_1 \leftrightarrow \varphi_2)}{\begin{array}{cc}\varphi_1 & \neg\varphi_1\\ \neg\varphi_2 & \varphi_2\end{array}} \qquad (\leftrightarrow\text{-elimination}).$$

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

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

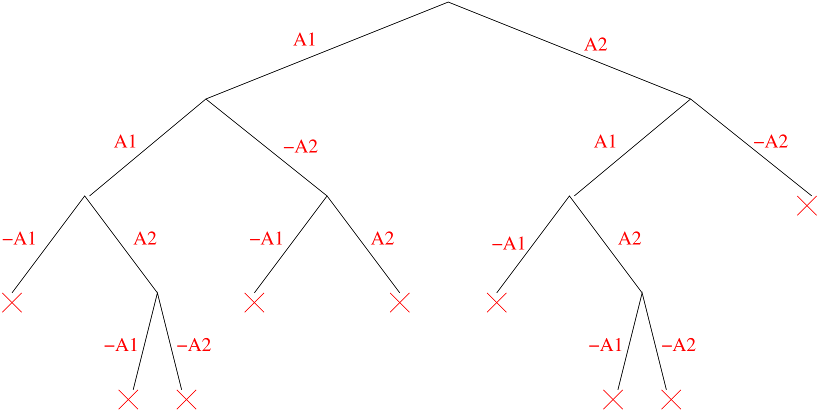


Tableau algorithm

```
function Tableau($\Gamma$)
    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 – 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

DPLL [15, 14]

- 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

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

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

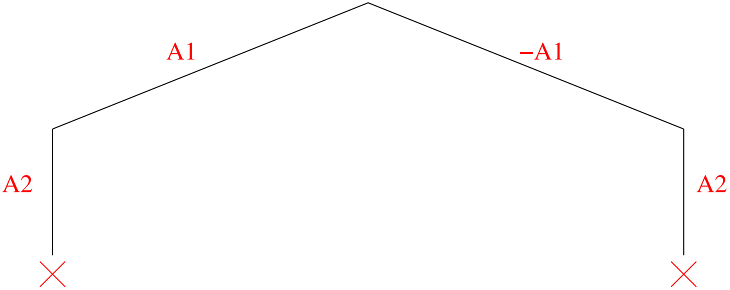
$$\frac{\varphi}{\varphi[l|\top] \quad \varphi[l|\bot]} \; (split)$$

(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 [57, 44, 45]

DPLL – example

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



DPLL Algorithm

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

DPLL – summary

- Handles CNF formulas (non-CNF variant known [2, 25]).
- 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 [61, 54, 4, 43].

Ordered Binary Decision Diagrams (OBDDs) [12]]

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)

Ordered Binary Decision Diagrams (OBDDs) [12]]

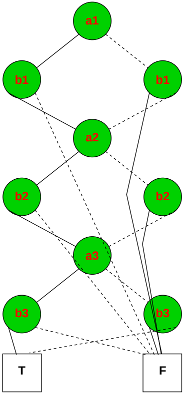
Canonical representation of Boolean formulas

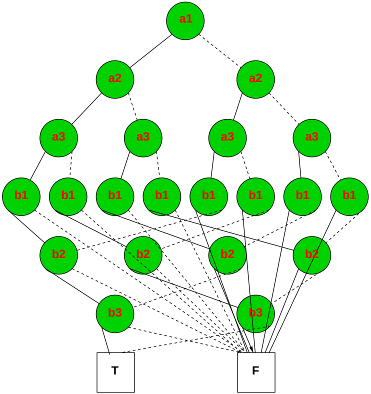
- “If-then-else” binary direct acyclic graphs (DAGs) with one root and two leaves: 1, 0 (or $\top$,$\bot$; or T, F)
- Variable ordering $A_1, A_2, ..., A_n$ imposed a priori.
- Paths leading to 1 represent models
  Paths leading to 0 represent counter-models

Note

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

OBDD - Examples

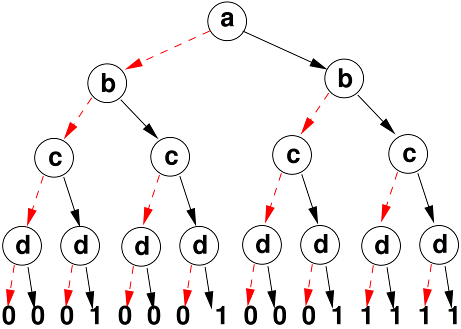




OBDDs of $(a_1 \leftrightarrow b_1) \wedge (a_2 \leftrightarrow b_2) \wedge (a_3 \leftrightarrow b_3)$ with different variable orderings

Ordered Decision Trees

- Ordered Decision Tree: from root to leaves, variables are encountered always in the same order
- Example: Ordered Decision tree for $\varphi = (a \land b) \lor (c \land d)$



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”)

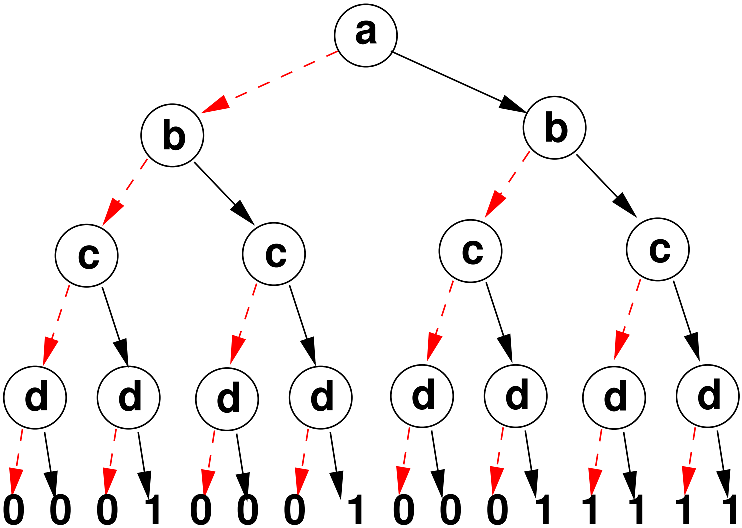
From Ordered Decision Trees to OBDD’s: reductions

- 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

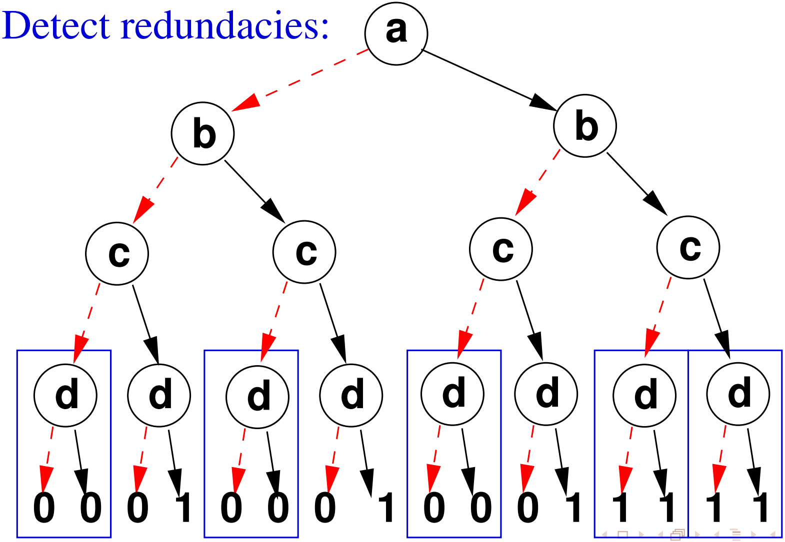
- Recursive applications of the following reductions:
    - share subnodes: point to the same occurrence of a subtree (via hash consing)
    - remove redundancies: nodes with same left and right children can be eliminated (“if A then B else B” $\Longrightarrow$ “B”)

Reduction: example



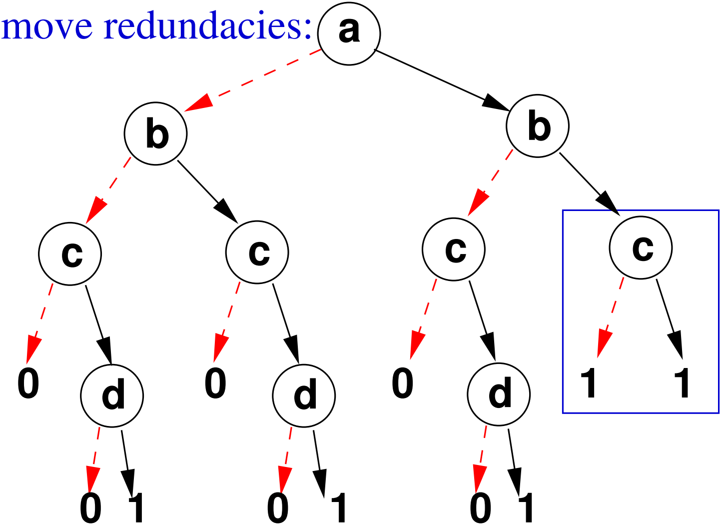
Reduction: example

Detect redundacies:



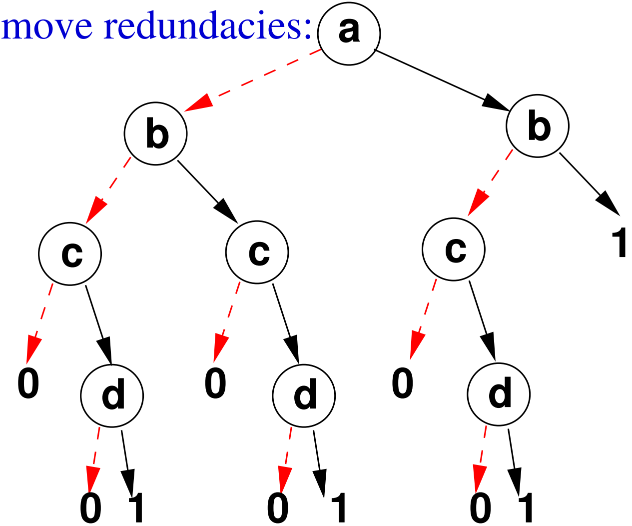
Reduction: example

Remove redundacies:



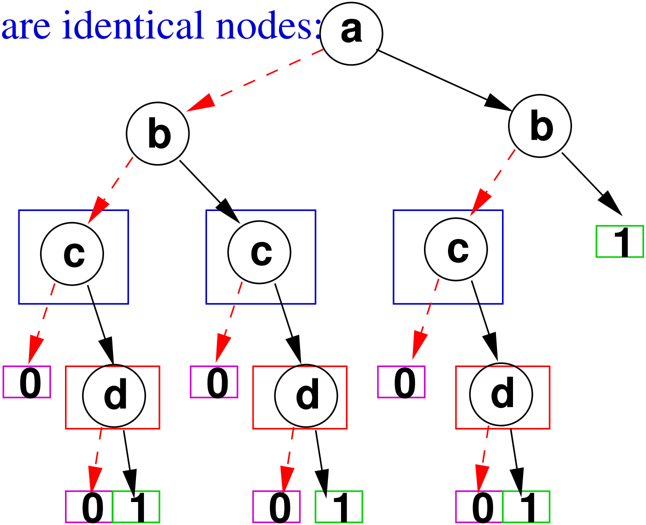
Reduction: example

Remove redundacies:



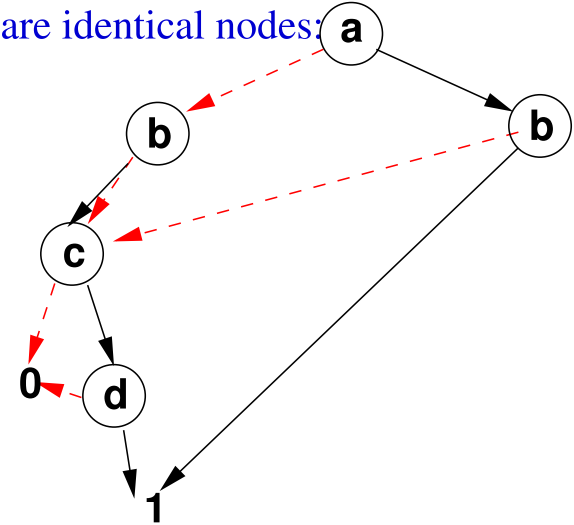
Reduction: example

Share identical nodes:



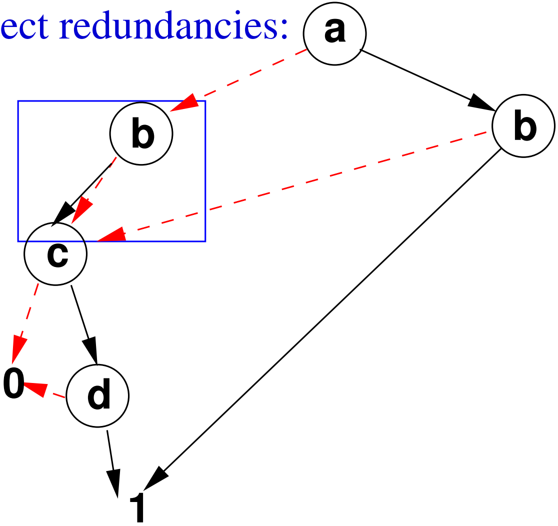
Reduction: example

Share identical nodes:



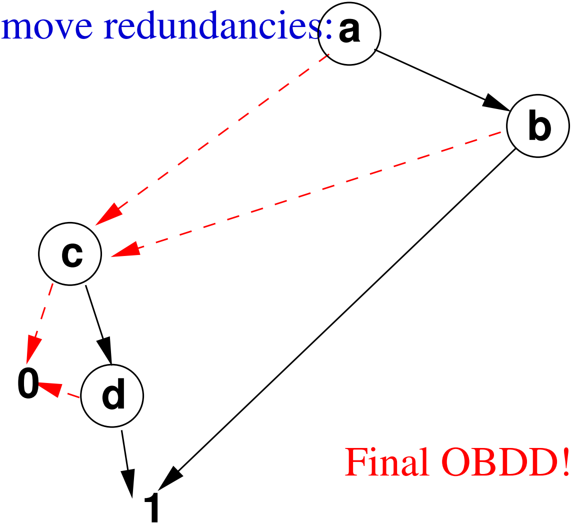
Reduction: example

Detect redundancies:



Reduction: example

Remove redundancies:



Final OBDD!

Recursive structure of an OBDD

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

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

Incrementally building an OBDD

- $obdd\_build(\top, \{...\}) := 1$,
- $obdd\_build(\bot, \{...\}) := 0$,
- $obdd\_build(A_i, \{...\}) := ite(A_i, 1, 0)$,
- $obdd\_build((\neg\varphi), \{A_1, ..., A_n\}) :=$
  $apply(\neg, obdd\_build(\varphi, \{A_1, ..., A_n\}))$
- $obdd\_build((\varphi_1\ op\ \varphi_2), \{A_1, ..., A_n\}) :=$
  $reduce($
  $apply(\quad op,$
  $obdd\_build(\varphi_1, \{A_1, ..., A_n\}),\quad op \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}$
  $obdd\_build(\varphi_2, \{A_1, ..., A_n\})$
  $))$

"$ite(A_i, \varphi_i^{\top}, \varphi_i^{\bot})$" is "If A_i Then $\varphi_i^{\top}$ Else $\varphi_i^{\bot}$"

Incrementally building an OBDD

- $obdd\_build(\top, \{...\}) := 1$,
- $obdd\_build(\bot, \{...\}) := 0$,
- $obdd\_build(A_i, \{...\}) := ite(A_i, 1, 0)$,
- $obdd\_build((\neg\varphi), \{A_1, ..., A_n\}) :=$ $apply(\neg, obdd\_build(\varphi, \{A_1, ..., A_n\}))$
- $obdd\_build((\varphi_1\ op\ \varphi_2), \{A_1, ..., A_n\}) :=$

```
reduce(
  apply(   op,
           obdd_build(φ1, {A1, ..., An}),   op ∈ {∧, ∨, →, ↔}
           obdd_build(φ2, {A1, ..., An})
))
```

"$ite(A_i, \varphi_i^\top, \varphi_i^\bot)$" is "If A_i Then $\varphi_i^\top$ Else $\varphi_i^\bot$"

Incrementally building an OBDD

- $obdd\_build(\top, \{...\}) := 1$,
- $obdd\_build(\bot, \{...\}) := 0$,
- $obdd\_build(A_i, \{...\}) := ite(A_i, 1, 0)$,
- $obdd\_build((\neg\varphi), \{A_1, ..., A_n\}) :=$
  $apply(\neg, obdd\_build(\varphi, \{A_1, ..., A_n\}))$
- $obdd\_build((\varphi_1\ op\ \varphi_2), \{A_1, ..., A_n\}) :=$
  $reduce($
  $apply(\quad op,$
  $obdd\_build(\varphi_1, \{A_1, ..., A_n\}), \quad op \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}$
  $obdd\_build(\varphi_2, \{A_1, ..., A_n\})$
  $))$

"$ite(A_i, \varphi_i^{\top}, \varphi_i^{\bot})$" is "If A_i Then $\varphi_i^{\top}$ Else $\varphi_i^{\bot}$"

Incrementally building an OBDD

- $obdd\_build(\top, \{...\}) := 1$,
- $obdd\_build(\bot, \{...\}) := 0$,
- $obdd\_build(A_i, \{...\}) := ite(A_i, 1, 0)$,
- $obdd\_build((\neg\varphi), \{A_1, ..., A_n\}) :=$
  $apply(\neg, obdd\_build(\varphi, \{A_1, ..., A_n\}))$
- $obdd\_build((\varphi_1\ op\ \varphi_2), \{A_1, ..., A_n\}) :=$

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

"$ite(A_i, \varphi_i^\top, \varphi_i^\bot)$" is "If A_i Then $\varphi_i^\top$ Else $\varphi_i^\bot$"

Incrementally building an OBDD

- $obdd\_build(\top, \{...\}) := 1$,
- $obdd\_build(\bot, \{...\}) := 0$,
- $obdd\_build(A_i, \{...\}) := ite(A_i, 1, 0)$,
- $obdd\_build((\neg\varphi), \{A_1, ..., A_n\}) :=$
  $apply(\neg, obdd\_build(\varphi, \{A_1, ..., A_n\}))$
- $obdd\_build((\varphi_1 \; op \; \varphi_2), \{A_1, ..., A_n\}) :=$

```
reduce(
  apply(   op,
           $obdd\_build(\varphi_1, \{A_1, ..., A_n\})$,   $op \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}$
           $obdd\_build(\varphi_2, \{A_1, ..., A_n\})$
) )
```

"$ite(A_i, \varphi_i^{\top}, \varphi_i^{\bot})$" is "If A_i Then $\varphi_i^{\top}$ Else $\varphi_i^{\bot}$"

Incrementally building an OBDD (cont.)

- $apply\ (op,\ O_i, O_j) := (O_i\ op\ O_j)$ **if** $(O_i, O_j \in \{1, 0\})$
- $apply\ (\neg,\ ite(A_i, \varphi_i^{\top}, \varphi_i^{\bot})) :=$
  $ite(A_i, apply(\neg, \varphi_i^{\top}), apply(\neg, \varphi_i^{\bot}))$
- $apply\ (op,\ ite(A_i, \varphi_i^{\top}, \varphi_i^{\bot}),\ ite(A_j, \varphi_j^{\top}, \varphi_j^{\bot})) :=$
  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** $ite(A_j,\ apply\ (op, ite(A_i, \varphi_i^{\top}, \varphi_i^{\bot}), \varphi_j^{\top}),$
  $apply\ (op, ite(A_i, \varphi_i^{\top}, \varphi_i^{\bot}), \varphi_j^{\bot}))$

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

Incrementally building an OBDD (cont.)

- $apply\ (op,\ O_i, O_j) := (O_i\ op\ O_j)$ **if** $(O_i, O_j \in \{1, 0\})$
- $apply\ (\neg,\ ite(A_i, \varphi_i^\top, \varphi_i^\bot)) :=$
  $ite(A_i, apply(\neg, \varphi_i^\top), apply(\neg, \varphi_i^\bot))$
- $apply\ (op,\ ite(A_i, \varphi_i^\top, \varphi_i^\bot),\ ite(A_j, \varphi_j^\top, \varphi_j^\bot)) :=$
  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** $ite(A_j,\ apply\ (op, ite(A_i, \varphi_i^\top, \varphi_i^\bot), \varphi_j^\top),$
  $apply\ (op, ite(A_i, \varphi_i^\top, \varphi_i^\bot), \varphi_j^\bot))$

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

Incrementally building an OBDD (cont.)

- $apply\ (op,\ O_i, O_j) := (O_i\ op\ O_j)$ **if** $(O_i, O_j \in \{1, 0\})$
- $apply\ (\neg,\ ite(A_i, \varphi_i^\top, \varphi_i^\bot)) :=$
  $ite(A_i, apply(\neg, \varphi_i^\top), apply(\neg, \varphi_i^\bot))$
- $apply\ (op,\ ite(A_i, \varphi_i^\top, \varphi_i^\bot),\ ite(A_j, \varphi_j^\top, \varphi_j^\bot)) :=$
  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** $ite(A_j,\ apply\ (op, ite(A_i, \varphi_i^\top, \varphi_i^\bot), \varphi_j^\top),\ apply\ (op, ite(A_i, \varphi_i^\top, \varphi_i^\bot), \varphi_j^\bot))$

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

Incrementally building an OBDD (cont.)

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

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

- 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):

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

Incrementally building an OBDD (cont.)

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

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

- 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):

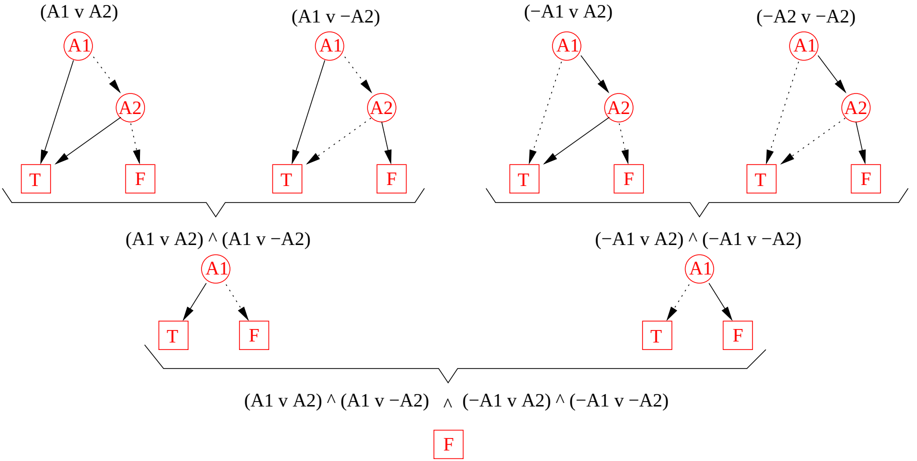
$$\begin{aligned} & apply(\wedge,\ \overbrace{ite(A_1, \top, ite(A_2, \top, \bot))}^{(A_1 \vee A_2)},\ \overbrace{ite(A_1, \top, ite(A_2, \bot, \top))}^{(A_1 \vee \neg A_2)}, \\ = \quad & ite(A_1,\ apply(\wedge, \top, \top),\ apply(\wedge,\ ite(A_2, \top, \bot),\ ite(A_2, \bot, \top)) \\ = \quad & ite(A_1,\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

OBBD incremental building – example

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

OBBD 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

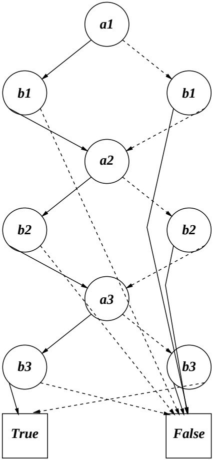
$$(a_1 \leftrightarrow b_1) \wedge (a_2 \leftrightarrow b_2) \wedge (a_3 \leftrightarrow b_3)$$

Linear size

Exponential size

Critical choice of variable Orderings in OBDD’s

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

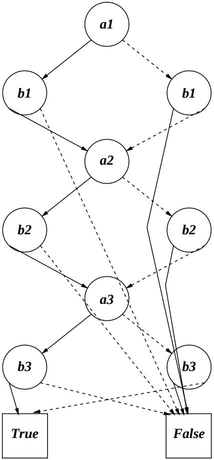


Linear size

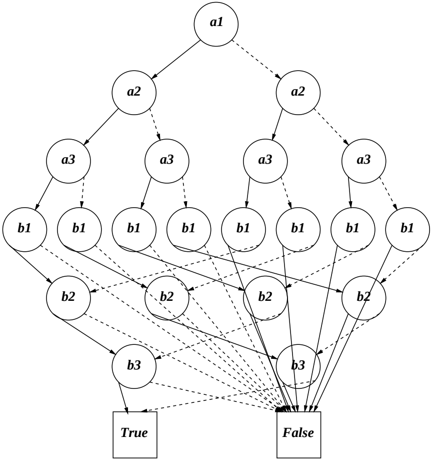
Exponential size

Critical choice of variable Orderings in OBDD's

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



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

OBD'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

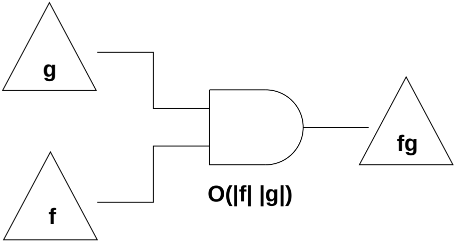
Useful Operations over OBDDs

- the equivalence check between two OBDDs is simple
    - are they the same OBDD? (⟹ 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).

Boolean quantification

Shannon's expansion:

- If v is a Boolean variable and f is a Boolean formula, then
  $$\begin{aligned}\exists v.f &:= f|_{v=0} \vee f|_{v=1}\\ \forall v.f &:= f|_{v=0} \wedge f|_{v=1}\end{aligned}$$
- v does no more occur in $\exists v.f$ and $\forall v.f$!!
- Multi-variable quantification: $\exists(w_1,\ldots,w_n).f \;:=\; \exists w_1 \ldots \exists w_n.f$

- Intuition:
  - $\mu \models \exists v.f$ iff exists $tvalue \in \{\top,\bot\}$ s.t. $\mu \cup \{v := tvalue\} \models f$
  - $\mu \models \forall v.f$ iff forall $tvalue \in \{\top,\bot\}$, $\mu \cup \{v := tvalue\} \models f$
- 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

$$\begin{array}{lcl} \exists v.f & := & f|_{v=0} \vee f|_{v=1} \\ \forall v.f & := & f|_{v=0} \wedge f|_{v=1} \end{array}$$

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

- Intuition:
  - $\mu \models \exists v.f$ iff exists $tvalue \in \{\top, \bot\}$ s.t. $\mu \cup \{v := tvalue\} \models f$
  - $\mu \models \forall v.f$ iff forall $tvalue \in \{\top, \bot\}$, $\mu \cup \{v := tvalue\} \models f$
- 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
  $$\begin{aligned} \exists v.f &:= f|_{v=0} \vee f|_{v=1} \\ \forall v.f &:= f|_{v=0} \wedge f|_{v=1} \end{aligned}$$
- v does no more occur in $\exists v.f$ and $\forall v.f$!!
- Multi-variable quantification: $\exists (w_1, \ldots, w_n).f \;:=\; \exists w_1 \ldots \exists w_n.f$

- Intuition:
  - $\mu \models \exists v.f$ iff exists $tvalue \in \{\top, \bot\}$ s.t. $\mu \cup \{v := tvalue\} \models f$
  - $\mu \models \forall v.f$ iff forall $tvalue \in \{\top, \bot\}$, $\mu \cup \{v := tvalue\} \models f$
- 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
  $$\begin{aligned} \exists v.f &:= f|_{v=0} \vee f|_{v=1} \\ \forall v.f &:= f|_{v=0} \wedge f|_{v=1} \end{aligned}$$
- v does no more occur in $\exists v.f$ and $\forall v.f$!!
- Multi-variable quantification: $\exists(w_1, \ldots, w_n).f \ := \ \exists w_1 \ldots \exists w_n.f$

- Intuition:
  - $\mu \models \exists v.f$ iff exists *tvalue* $\in \{\top, \bot\}$ s.t. $\mu \cup \{v := tvalue\} \models f$
  - $\mu \models \forall v.f$ iff forall *tvalue* $\in \{\top, \bot\}$, $\mu \cup \{v := tvalue\} \models f$
- 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
$$\begin{aligned} \exists v.f &:= f|_{v=0} \vee f|_{v=1} \\ \forall v.f &:= f|_{v=0} \wedge f|_{v=1} \end{aligned}$$
- v does no more occur in $\exists v.f$ and $\forall v.f$!!
- Multi-variable quantification: $\exists(w_1, \ldots, w_n).f \ := \ \exists w_1 \ldots \exists w_n.f$

- Intuition:
  - $\mu \models \exists v.f$ iff exists *tvalue* $\in \{\top, \bot\}$ s.t. $\mu \cup \{v := tvalue\} \models f$
  - $\mu \models \forall v.f$ iff forall *tvalue* $\in \{\top, \bot\}$, $\mu \cup \{v := tvalue\} \models f$
- 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

$$\begin{aligned} \exists v.f &:= f|_{v=0} \vee f|_{v=1} \\ \forall v.f &:= f|_{v=0} \wedge f|_{v=1} \end{aligned}$$

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

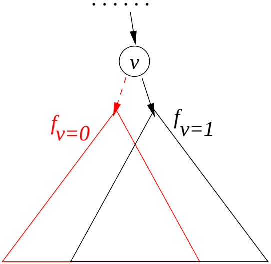
- Intuition:
  - $\mu \models \exists v.f$ iff exists $tvalue \in \{\top, \bot\}$ s.t. $\mu \cup \{v := tvalue\} \models f$
  - $\mu \models \forall v.f$ iff forall $tvalue \in \{\top, \bot\}$, $\mu \cup \{v := tvalue\} \models f$
- Example: $\exists(b, c).((a \wedge b) \vee (c \wedge d)) \;=\; a \vee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

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 $f|_{v=1}$ and $f|_{v=0}$ are the "then" and "else" branches of f



$\Longrightarrow$ lots of sharing of subformulae!

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).

Incomplete SAT techniques: GSAT, WSAT [53, 52]

- Hill-Climbing techniques: GSAT, WSAT
- looks for a complete assignment;
- starts from a random assignment;
- Greedy search: looks for a better “neighbor” assignment
- Avoid local minima: restart & random walk

The GSAT algorithm [53]

```
function GSAT($\varphi$)
    for $i := 1$ to Max-tries do
        $\mu$ := rand-assign($\varphi$);
        for $j := 1$ to Max-flips do
            if ($score(\varphi, \mu) = 0$)
                then return True;
                else Best-flips := hill-climb($\varphi, \mu$);
                    $A_i$ := rand-pick(Best-flips);
                    $\mu$ := flip($A_i, \mu$);
        end
    end
    return "no satisfying assignment found".
```

The WalkSAT algorithm(s) [52]

```
function WalkSAT($\varphi$)
    for $i := 1$ to Max-tries do
        $\mu$ := rand-assign($\varphi$);
        for $j := 1$ to Max-flips do
            if ($score(\varphi, \mu) = 0$)
                then return True;
                else C := randomly-pick-clause(unsat-clauses($\varphi, \mu$));
                    $A_i$ := heuristically-select-variable(C);
                    $\mu$ := flip($A_i, \mu$);
        end
    end
    return "no satisfying assignment found".
```

- many variants available [27, 58, 5]

SLS SAT solvers – summary

- Handle only CNF formulas.
- Incomplete
- Extremely efficient for some (satisfiable) problems.
- Require polynomial space
- Used in Artificial Intelligence (e.g., planning)
- Lots of variants (see e.g. [31])
- Non-CNF Variants: [50, 51, 6]

Outline

1. Basics on SAT
2. Basic SAT-Solving techniques
3. Modern CDCL SAT Solvers
    - Conflict-Driven Clause-Learning SAT solvers
    - Further Improvements
4. Tractable subclasses of SAT
5. Random k-SAT and Phase Transition
6. Advanced Functionalities: proofs, unsat cores, interpolants, optimization
7. Some Applications
    - Appl. #1: (Bounded) Planning
    - Appl. #2: Bounded Model Checking

Variants of DPLL

DPLL is a family of algorithms.

- backjumping & learning
- preprocessing: (subsumption, 2-simplification, resolution)
- different branching heuristics
- restarts
- (horn relaxation)
- ...

“Classic” chronological backtracking

DPLL implements “classic” chronological backtracking:

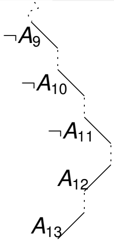
- variable assignments (literals) stored in a stack
- each variable assignments labeled as “unit”, “open”, “closed”
- when a conflict is encountered, the stack is popped up to the most recent open assignment l
- l is toggled, is labeled as “closed”, and the search proceeds.

Classic chronological backtracking – example

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

Classic chronological backtracking – example

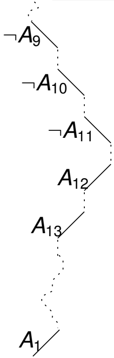
$c_1 : \neg A_1 \vee A_2$
$c_2 : \neg A_1 \vee A_3 \vee A_9$
$c_3 : \neg A_2 \vee \neg A_3 \vee A_4$
$c_4 : \neg A_4 \vee A_5 \vee A_{10}$
$c_5 : \neg A_4 \vee A_6 \vee A_{11}$
$c_6 : \neg A_5 \vee \neg A_6$
$c_7 : A_1 \vee A_7 \vee \neg A_{12}$
$c_8 : A_1 \vee A_8$
$c_9 : \neg A_7 \vee \neg A_8 \vee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



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

Classic chronological backtracking – example

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



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

Classic chronological backtracking – example

$c_1 : \neg A_1 \lor A_2$ ✓

$c_2 : \neg A_1 \lor A_3 \lor A_9$ ✓

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

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

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

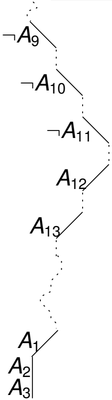
$c_6 : \neg A_5 \lor \neg A_6$

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

$c_8 : A_1 \lor A_8$ ✓

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

...

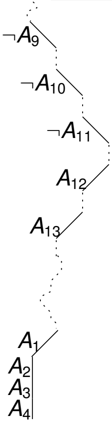


$\{..., \neg A_9, \neg A_{10}, \neg A_{11}, A_{12}, A_{13}, ..., A_1, A_2, A_3\}$

(unit A_2, A_3)

Classic chronological backtracking – example

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



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

Classic chronological backtracking – example

$c_1 : \neg A_1 \lor A_2$ ✓

$c_2 : \neg A_1 \lor A_3 \lor A_9$ ✓

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

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

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

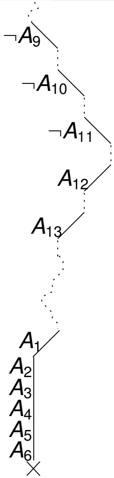
$c_6 : \neg A_5 \lor \neg A_6$ ×

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

$c_8 : A_1 \lor A_8$ ✓

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

...

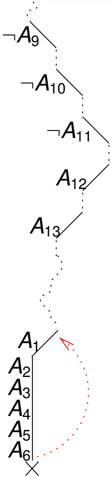


$\{..., \neg A_9, \neg A_{10}, \neg A_{1\neg A_4 1}, A_{12}, A_{13}, ..., A_1, A_2, A_3, A_4, A_5, A_6\}$

(unit A_5, A_6)$\Longrightarrow$ conflict

Classic chronological backtracking – example

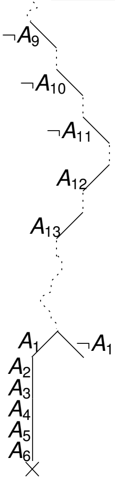
$c_1 : \neg A_1 \vee A_2$
$c_2 : \neg A_1 \vee A_3 \vee A_9$
$c_3 : \neg A_2 \vee \neg A_3 \vee A_4$
$c_4 : \neg A_4 \vee A_5 \vee A_{10}$
$c_5 : \neg A_4 \vee A_6 \vee A_{11}$
$c_6 : \neg A_5 \vee \neg A_6$
$c_7 : A_1 \vee A_7 \vee \neg A_{12}$
$c_8 : A_1 \vee A_8$
$c_9 : \neg A_7 \vee \neg A_8 \vee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



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

Classic chronological backtracking – example

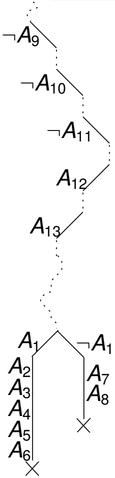
$c_1 : \neg A_1 \vee A_2$ ✓
$c_2 : \neg A_1 \vee A_3 \vee A_9$ ✓
$c_3 : \neg A_2 \vee \neg A_3 \vee A_4$
$c_4 : \neg A_4 \vee A_5 \vee A_{10}$
$c_5 : \neg A_4 \vee A_6 \vee A_{11}$
$c_6 : \neg A_5 \vee \neg A_6$
$c_7 : A_1 \vee A_7 \vee \neg A_{12}$
$c_8 : A_1 \vee A_8$
$c_9 : \neg A_7 \vee \neg A_8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



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

Classic chronological backtracking – example

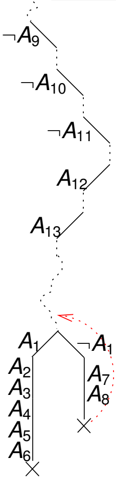
$c_1 : \neg A_1 \vee A_2$ ✓
$c_2 : \neg A_1 \vee A_3 \vee A_9$ ✓
$c_3 : \neg A_2 \vee \neg A_3 \vee A_4$
$c_4 : \neg A_4 \vee A_5 \vee A_{10}$
$c_5 : \neg A_4 \vee A_6 \vee A_{11}$
$c_6 : \neg A_5 \vee \neg A_6$
$c_7 : A_1 \vee A_7 \vee \neg A_{12}$ ✓
$c_8 : A_1 \vee A_8$ ✓
$c_9 : \neg A_7 \vee \neg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



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

Classic chronological backtracking – example

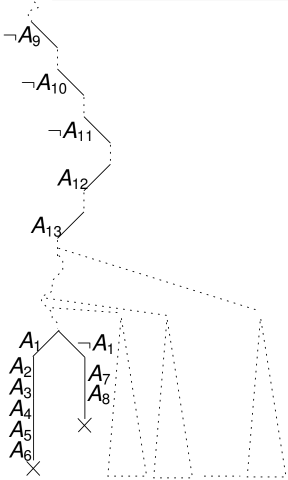
$c_1 : \neg A_1 \vee A_2$
$c_2 : \neg A_1 \vee A_3 \vee A_9$
$c_3 : \neg A_2 \vee \neg A_3 \vee A_4$
$c_4 : \neg A_4 \vee A_5 \vee A_{10}$
$c_5 : \neg A_4 \vee A_6 \vee A_{11}$
$c_6 : \neg A_5 \vee \neg A_6$
$c_7 : A_1 \vee A_7 \vee \neg A_{12}$
$c_8 : A_1 \vee A_8$
$c_9 : \neg A_7 \vee \neg A_8 \vee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



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

Classic chronological backtracking – example

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



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

Classic chronological backtracking: drawbacks

- often the branch heuristic delays the “right” choice
- chronological backtracking always backtracks to the most recent branching point, even though a higher backtrack could be possible $\Longrightarrow$ lots of useless search!

Outline

1. Basics on SAT
2. Basic SAT-Solving techniques
3. Modern CDCL SAT Solvers
   - Conflict-Driven Clause-Learning SAT solvers
   - Further Improvements
4. Tractable subclasses of SAT
5. Random k-SAT and Phase Transition
6. Advanced Functionalities: proofs, unsat cores, interpolants, optimization
7. Some Applications
   - Appl. #1: (Bounded) Planning
   - Appl. #2: Bounded Model Checking

Conflict-Driven Clause-Learning (CDCL) SAT solvers

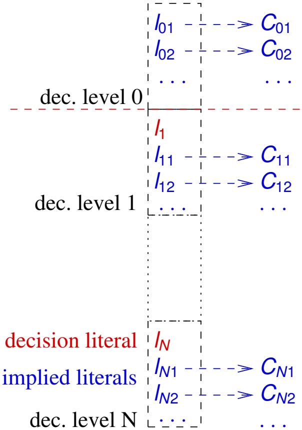
Conflict-Driven Clause-Learning (CDCL) SAT solvers [54, 43, 18, 37]

- Evolution of Davis-Putnam-Longeman-Loveland (DPLL) [15, 14]
- non-recursive: stack-based representation of data structures
- Perform conflict-directed backtracking (backjumping) and learning
- efficient data structures for doing and undoing assignments (e.g., two-watched-literal scheme)
- perform search restarts
- ...

Dramatically efficient: solve industrial-derived problems with $\approx 10^7$ Boolean variables and $\approx 10^7 - 10^8$ 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



l_{01} → C_{01}
l_{02} → C_{02}
...   ...
dec. level 0

l_1
l_{11} → C_{11}
l_{12} → C_{12}
dec. level 1 ...   ...

decision literal l_N
implied literals l_{N1} → C_{N1}
l_{N2} → C_{N2}
dec. level N ...   ...

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

Implication graph - example

$c_1 : \neg A_1 \vee A_2$ ✓

$c_2 : \neg A_1 \vee A_3 \vee A_9$ ✓

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

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

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

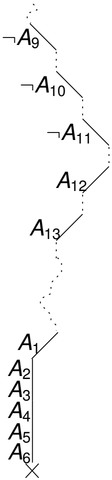
$c_6 : \neg A_5 \vee \neg A_6$ ×

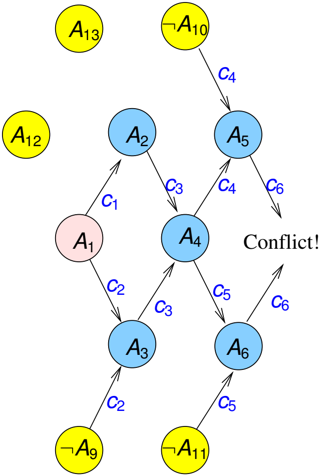
$c_7 : A_1 \vee A_7 \vee \neg A_{12}$ ✓

$c_8 : A_1 \vee A_8$ ✓

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

...





Unique implication point - UIP [63]

- 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

Unique implication point - UIP - example

$c_1 : \neg A_1 \lor A_2$ ✓

$c_2 : \neg A_1 \lor A_3 \lor A_9$ ✓

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

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

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

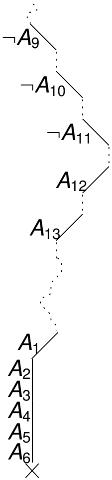
$c_6 : \neg A_5 \lor \neg A_6$ ×

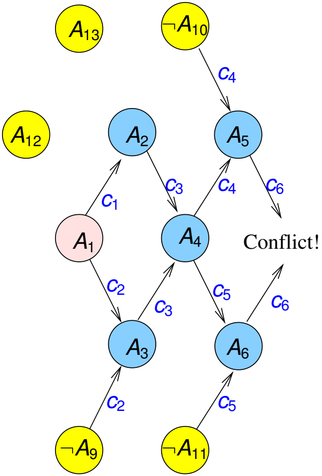
$c_7 : A_1 \lor A_7 \lor \neg A_{12}$ ✓

$c_8 : A_1 \lor A_8$ ✓

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

...





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

Schema of a CDCL DPLL solver [54, 64]

```
Function CDCL-SAT (formula: $\varphi$, assignment & $\mu$) {
        status := preprocess($\varphi,\mu$);
        while (1) {
            while (1) {
                status := deduce($\varphi,\mu$);
                if (status == Sat)
                    return Sat;
                if (status == Conflict) {
                    $\langle$blevel,$\eta\rangle$ := analyze_conflict($\varphi,\mu$);
                    //$\eta$ is a conflict set
                    if (blevel == 0)
                        return Unsat;
                    else backtrack(blevel,$\varphi,\mu$);
                }
                else break;
            }
            decide_next_branch($\varphi,\mu$);
}       }
```

Schema of a CDCL DPLL solver [54, 64]

- `preprocess(`φ, μ`)` simplifies φ into an easier equisatisfiable formula (and updates μ if it is the case)
- `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), 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 (“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

Example

$c_1 : \neg A_1 \vee A_2$

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

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

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

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

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

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

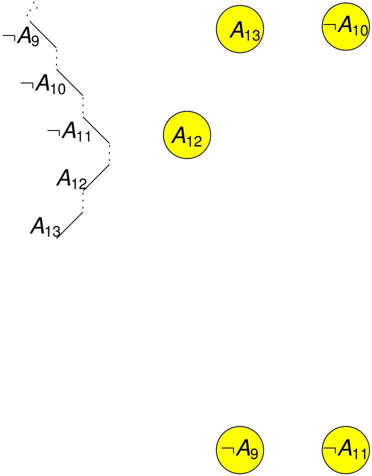
$c_8 : A_1 \vee A_8$

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

...

Example

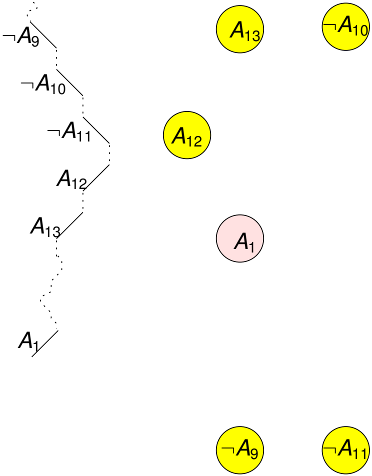
$c_1 : \neg A_1 \vee A_2$
$c_2 : \neg A_1 \vee A_3 \vee A_9$
$c_3 : \neg A_2 \vee \neg A_3 \vee A_4$
$c_4 : \neg A_4 \vee A_5 \vee A_{10}$
$c_5 : \neg A_4 \vee A_6 \vee A_{11}$
$c_6 : \neg A_5 \vee \neg A_6$
$c_7 : A_1 \vee A_7 \vee \neg A_{12}$
$c_8 : A_1 \vee A_8$
$c_9 : \neg A_7 \vee \neg A_8 \vee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



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

Example

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



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

Example

$c_1 : \neg A_1 \vee A_2$ ✓

$c_2 : \neg A_1 \vee A_3 \vee A_9$ ✓

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

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

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

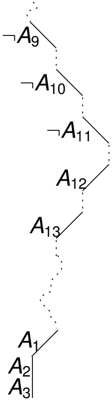
$c_6 : \neg A_5 \vee \neg A_6$

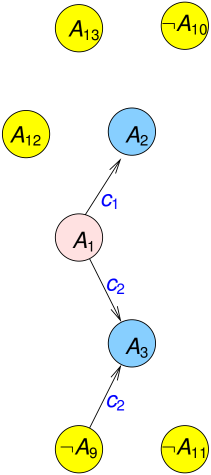
$c_7 : A_1 \vee A_7 \vee \neg A_{12}$ ✓

$c_8 : A_1 \vee A_8$ ✓

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

...





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

Example

$c_1 : \neg A_1 \lor A_2$ ✓

$c_2 : \neg A_1 \lor A_3 \lor A_9$ ✓

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

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

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

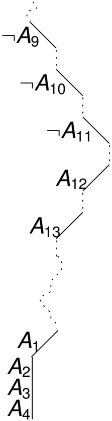
$c_6 : \neg A_5 \lor \neg A_6$

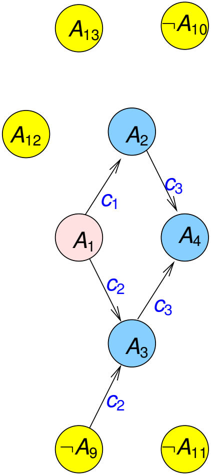
$c_7 : A_1 \lor A_7 \lor \neg A_{12}$ ✓

$c_8 : A_1 \lor A_8$ ✓

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

...





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

(unit A_4)

Example

$c_1 : \neg A_1 \vee A_2$ ✓

$c_2 : \neg A_1 \vee A_3 \vee A_9$ ✓

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

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

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

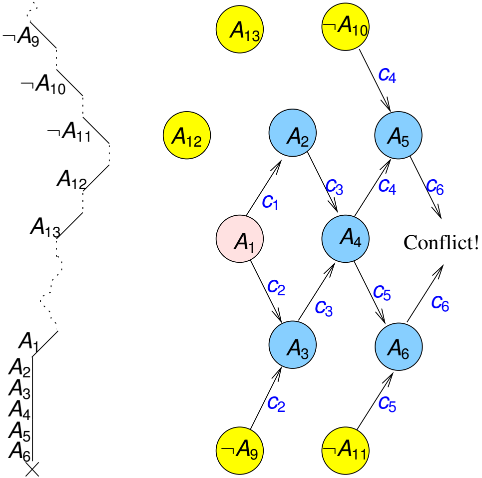
$c_6 : \neg A_5 \vee \neg A_6$ ×

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

$c_8 : A_1 \vee A_8$ ✓

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

...



$\{..., \neg A_9, \neg A_{10}, \neg A_{1}{}_{\neg A_4}1, A_{12}, A_{13}, ..., A_1, A_2, A_3, A_4, A_5, A_6\}$

(unit A_5, A_6) $\Longrightarrow$ conflict

Backjumping and learning: general ideas [4, 54]

- 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 $C \overset{\text{def}}{=} \neg\eta$ to the clause set
  (iii) backjumping: use η to decide the point where to backtrack
- may 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 [54]
  - the state-of-the-art 1^{st}UIP strategy of [63]

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

criterium: 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

criterium: last UIP

... until C contains only one literal assigned at current decision level: the decision literal (last UIP)

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

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

criterium: 1st UIP

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

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

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 C.

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 C.

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

$c_1 : \neg A_1 \lor A_2$ ✓

$c_2 : \neg A_1 \lor A_3 \lor A_9$ ✓

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

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

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

$c_6 : \neg A_5 \lor \neg A_6$ ×

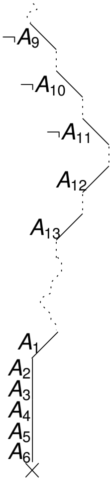
$c_7 : A_1 \lor A_7 \lor \neg A_{12}$ ✓

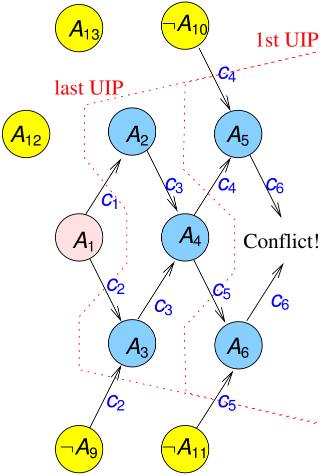
$c_8 : A_1 \lor A_8$ ✓

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

...

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



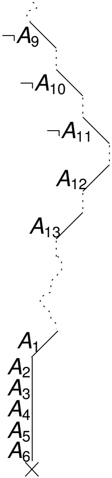


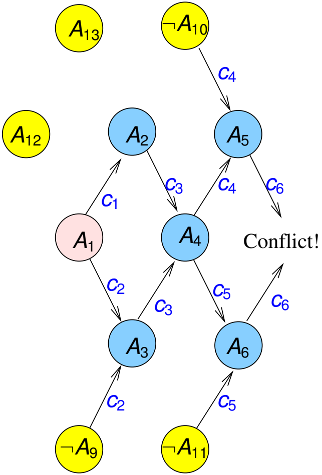
The original backjumping and learning strategy of [54]

- Idea: when a branch μ fails,
  - (i) conflict analysis: find the conflict set $\eta \subseteq \mu$ by generating the conflict clause $C \overset{\text{def}}{=} \neg\eta$ 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

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





$\{\ldots, \neg A_9, \neg A_{10}, \neg A_{1_{\neg A_4}1}, A_{12}, A_{13}, \ldots, A_1, A_2, A_3, A_4, A_5, A_6\}$
(unit A_5, A_6) $\Longrightarrow$ conflict

The original backjumping strategy – example

$c_1 : \neg A_1 \vee A_2$ ✓

$c_2 : \neg A_1 \vee A_3 \vee A_9$ ✓

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

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

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

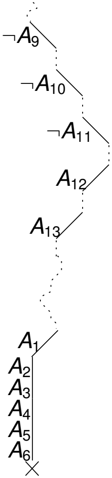
$c_6 : \neg A_5 \vee \neg A_6$ ×

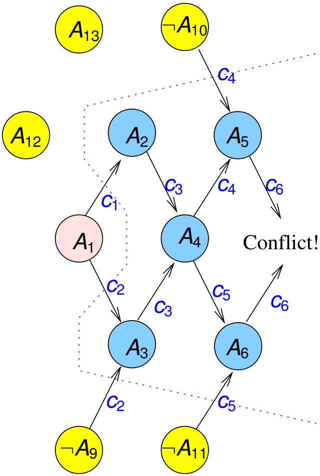
$c_7 : A_1 \vee A_7 \vee \neg A_{12}$ ✓

$c_8 : A_1 \vee A_8$ ✓

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

...



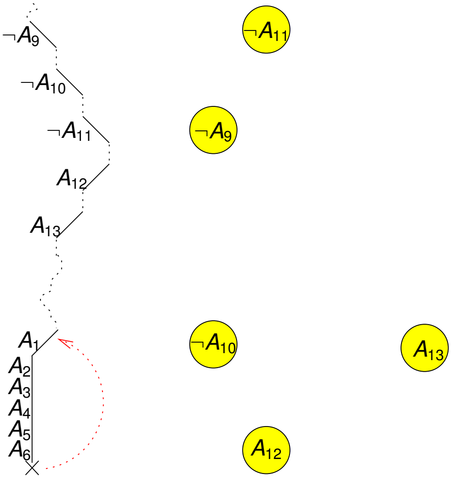


$\Longrightarrow$ Conflict set: $\{\neg A_9, \neg A_{10}, \neg A_{11}, A_1\}$ (last-UIP schema)

$\Longrightarrow$ learn the conflict clause $c_{10} := A_9 \vee A_{10} \vee A_{11} \vee \neg A_1$

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

$c_1 : \neg A_1 \vee A_2$ √

$c_2 : \neg A_1 \vee A_3 \vee A_9$ √

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

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

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

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

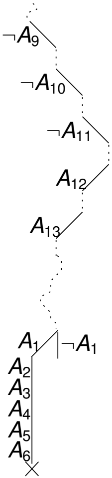
$c_7 : A_1 \vee A_7 \vee \neg A_{12}$

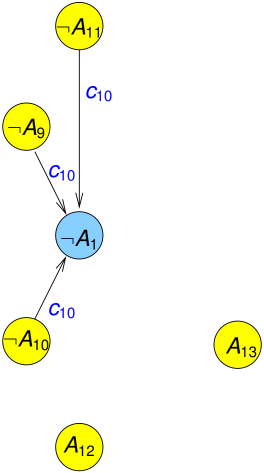
$c_8 : A_1 \vee A_8$

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

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

...

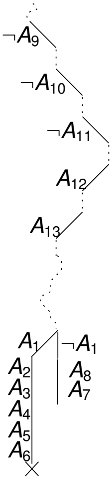


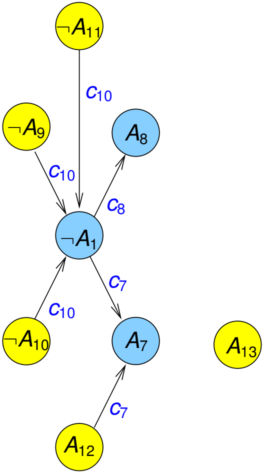


$\{\ldots, \neg A_9, \neg A_{10}, \neg A_{11}, A_{12}, A_{13}, \ldots, \neg A_1\}$
(unit $\neg A_1$)

The original backjumping strategy – example

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





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

The original backjumping strategy – example

$c_1 : \neg A_1 \vee A_2$ ✓

$c_2 : \neg A_1 \vee A_3 \vee A_9$ ✓

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

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

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

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

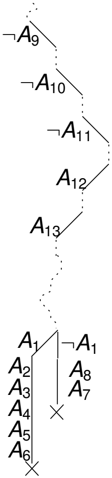
$c_7 : A_1 \vee A_7 \vee \neg A_{12}$ ✓

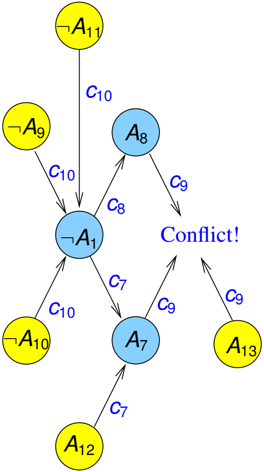
$c_8 : A_1 \vee A_8$ ✓

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

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

...





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

Conflict!

The original backjumping strategy – example

$c_1 : \neg A_1 \vee A_2$ ✓

$c_2 : \neg A_1 \vee A_3 \vee A_9$ ✓

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

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

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

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

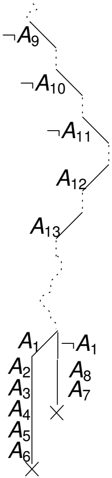
$c_7 : A_1 \vee A_7 \vee \neg A_{12}$ ✓

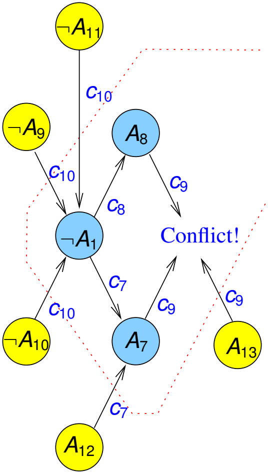
$c_8 : A_1 \vee A_8$ ✓

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

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

...





$\Longrightarrow$ 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

$c_1 : \neg A_1 \vee A_2$

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

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

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

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

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

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

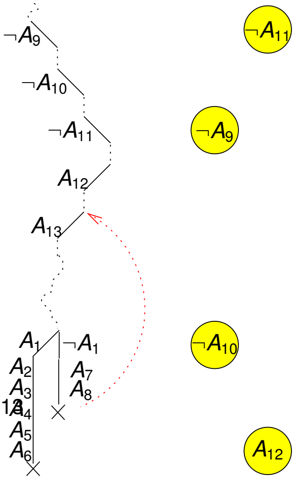
$c_8 : A_1 \vee A_8$

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

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

$c_{11} : A_9 \vee A_{10} \vee A_{11} \vee \neg A_{12} \vee \neg A_{13}$

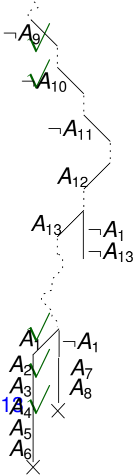
...

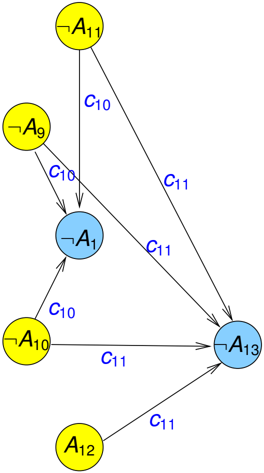


$\Longrightarrow$ backtrack to A_{13} $\Longrightarrow$ Lots of search saved!

The original backjumping strategy – example

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





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

State-of-the-art backjumping and learning [63]

- 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)

$c_1 : \neg A_1 \vee A_2$ ✓

$c_2 : \neg A_1 \vee A_3 \vee A_9$ ✓

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

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

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

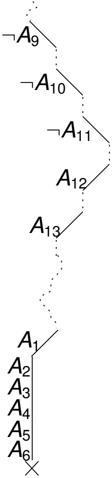
$c_6 : \neg A_5 \vee \neg A_6$ ×

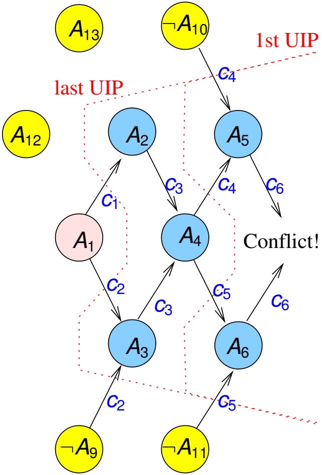
$c_7 : A_1 \vee A_7 \vee \neg A_{12}$ ✓

$c_8 : A_1 \vee A_8$ ✓

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

...





$\Longrightarrow$ 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 [63]

- 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)

$c_1 : \neg A_1 \lor A_2$ ✓

$c_2 : \neg A_1 \lor A_3 \lor A_9$ ✓

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

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

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

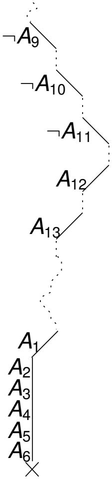
$c_6 : \neg A_5 \lor \neg A_6$ ×

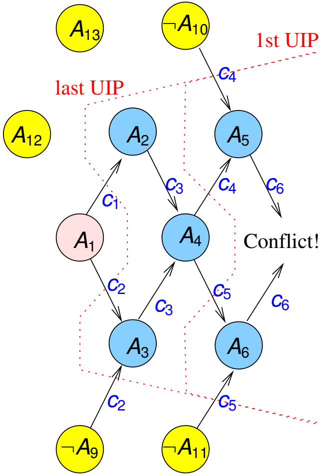
$c_7 : A_1 \lor A_7 \lor \neg A_{12}$ ✓

$c_8 : A_1 \lor A_8$ ✓

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

...

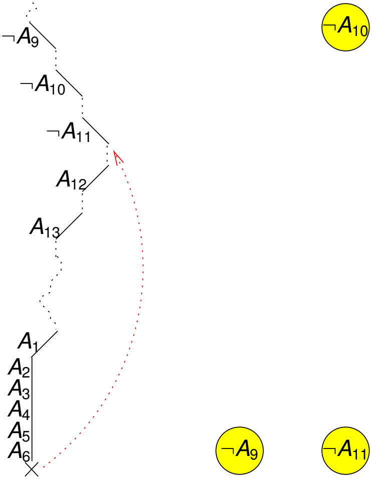




$\Longrightarrow$ Conflict set: $\{\neg A_{10}, \neg A_{11}, A_4\}$, learn $c_{10} := A_{10} \lor A_{11} \lor \neg A_4$

1st UIP strategy – example (8)

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



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

1st UIP strategy – example (9)

$c_1 : \neg A_1 \vee A_2$

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

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

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

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

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

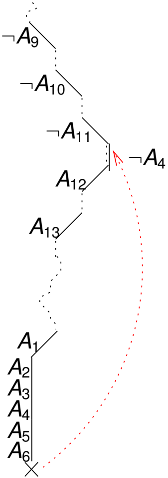
$c_7 : A_1 \vee A_7 \vee \neg A_{12}$

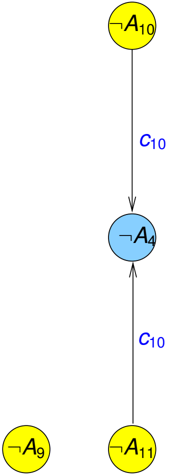
$c_8 : A_1 \vee A_8$

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

$c_{10} : A_{10} \vee A_{11} \vee \neg A_4$ ✓

...





$\Longrightarrow$ 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 [4, 54]

Idea: When a conflict set η is revealed, then $C \overset{\text{def}}{=} \neg\eta$ 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

$c_1 : \neg A_1 \vee A_2$

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

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

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

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

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

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

$c_8 : A_1 \vee A_8$

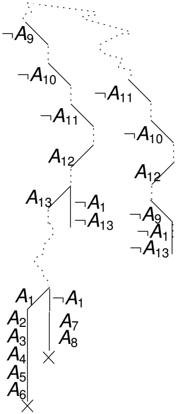
$c_9 : \neg A_7 \vee \neg A_8 \vee \neg A_{13}$ ✓

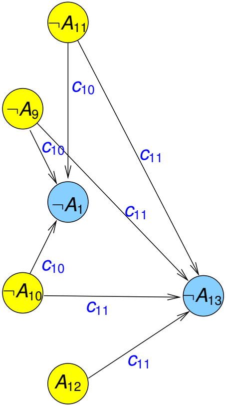
$c_{10} : A_9 \vee A_{10} \vee A_{11} \vee \neg A_1$ ✓

$c_{11} : A_9 \vee A_{10} \vee A_{11} \vee \neg A_{12} \vee \neg A_{13}$ ✓

...

$\Longrightarrow$ 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., [45])

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., [45])

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., [45])

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., [4, 54, 63])

Outline

1. Basics on SAT
2. Basic SAT-Solving techniques
3. Modern CDCL SAT Solvers
    - Conflict-Driven Clause-Learning SAT solvers
    - Further Improvements
4. Tractable subclasses of SAT
5. Random k-SAT and Phase Transition
6. Advanced Functionalities: proofs, unsat cores, interpolants, optimization
7. Some Applications
    - Appl. #1: (Bounded) Planning
    - Appl. #2: Bounded Model Checking

Preprocessing: (sorting plus) subsumption

- Detect and remove subsumed clauses:

$$\begin{array}{c}\varphi_1 \wedge (l_2 \vee l_1) \wedge \varphi_2 \wedge (l_2 \vee l_3 \vee l_1) \wedge \varphi_3 \\ \Downarrow \\ \varphi_1 \wedge (l_1 \vee l_2) \wedge \varphi_2 \wedge \varphi_3\end{array}$$

Preprocessing: detect & collapse equivalent literals [11]

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: detect & collapse equivalent literals [11]

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: detect & collapse equivalent literals [11]

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: detect & collapse equivalent literals [11]

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: detect & collapse equivalent literals [11]

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: detect & collapse equivalent literals [11]

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: detect & collapse equivalent literals [11]

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: resolution (and subsumption) [3]

- Apply some basic steps of resolution (and simplify):

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

Branching heuristics

- Branch is the source of non-determinism for DPLL
  $\Longrightarrow$ critical for efficiency
- many branch heuristics conceived in literature.

Some example heuristics

- MOMS heuristics: pick the literal occurring most often in the minimal size clauses
  $\Longrightarrow$ fast and simple, many variants
- Jeroslow-Wang: 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 [33]: 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 [43]: variable state independent decaying sum
  - “static”: scores updated only at the end of a branch
  - “local”: privileges variable in recently learned clauses

Restarts [26]

(according to some strategy) restart DPLL

- 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
- may significantly reduce the overall search space

Outline

1. Basics on SAT
2. Basic SAT-Solving techniques
3. Modern CDCL SAT Solvers
    - Conflict-Driven Clause-Learning SAT solvers
    - Further Improvements
4. Tractable subclasses of SAT
5. Random k-SAT and Phase Transition
6. Advanced Functionalities: proofs, unsat cores, interpolants, optimization
7. Some Applications
    - Appl. #1: (Bounded) Planning
    - Appl. #2: Bounded Model Checking

Tractable subclasses of SAT

- SAT in general is an NP-complete problem
- Some subclasses of SAT are tractable
- Two noteworthy tractable subclasses of SAT:
  - Horn Formulas (Horn-SAT)
  - 2-CNF formulas (2-SAT)

Horn Formulas

- A Horn formula is a CNF Boolean formula s.t. each clause contains at most one positive literal.

  $A_1 \vee \neg A_2$
  $A_2 \vee \neg A_3 \vee \neg A_4$
  $\neg A_5 \vee \neg A_3 \vee \neg A_4$
  A_3
- Intuition: implications between positive Boolean variables:

$$\begin{array}{rl} A_2 \rightarrow & A_1 \\ (A_3 \wedge A_4) \rightarrow & A_2 \\ (A_5 \wedge A_3 \wedge A_4) \rightarrow & \bot \\ & A_3 \end{array}$$

Formulas reducible to Horn

- Remark: Some non-Horn formulas can be reduced to Horn by simply renaming literals

$$\begin{array}{l} A_1 \vee A_2 \\ \neg A_2 \vee \neg A_3 \vee \neg A_4 \\ \neg A_5 \vee \neg A_3 \vee \neg A_4 \\ A_3 \end{array} \Longrightarrow_{B:=\neg A_2} \begin{array}{l} A_1 \vee \neg B \\ B \vee \neg A_3 \vee \neg A_4 \\ \neg A_5 \vee \neg A_3 \vee \neg A_4 \\ A_3 \end{array}$$

Tractability of Horn Formulas

Property

Checking the satisfiability of Horn formulas requires polynomial time

Hint:

(i) Eliminate unit clauses by propagating their value;
$\Longrightarrow$ Every clause contains at least one negative literal.

(ii) Assign all variables to $\bot$;

Tractability of Horn Formulas

Property

Checking the satisfiability of Horn formulas requires polynomial time

Hint:

(i) Eliminate unit clauses by propagating their value;
$\Longrightarrow$ Every clause contains at least one negative literal.

(ii) Assign all variables to $\perp$;

Tractability of Horn Formulas

Property

Checking the satisfiability of Horn formulas requires polynomial time

Hint:

(i) Eliminate unit clauses by propagating their value;
$\Longrightarrow$ Every clause contains at least one negative literal.

(ii) Assign all variables to $\bot$;

A simple polynomial procedure for Horn-SAT

```
function Horn_SAT(formula $\varphi$, assignment & $\mu$) {
    Unit_Propagate($\varphi$, $\mu$);
    if ($\varphi == \bot$)
        then return UNSAT;
    else {
        $\mu := \mu \cup \bigcup_{A_i \not\in \mu} \{\neg A_i\}$;
        return SAT;
} }

function Unit_Propagate(formula & $\varphi$, assignment & $\mu$)
    while ($\varphi \neq \top$ and $\varphi \neq \bot$ and {a unit clause ($l$) occurs in $\varphi$}) do {
        $\varphi = assign(\varphi, l)$;
        $\mu := \mu \cup \{l\}$;
} }
```

Example

$$\begin{array}{llll} \neg A_1 & \vee\ A_2 & \vee \neg A_3 \\ A_1 & \vee \neg A_3 & \vee \neg A_4 \\ \neg A_2 & \vee \neg A_4 & \\ A_3 & \vee \neg A_4 & \\ A_4 & & \end{array}$$

Example

$$\begin{array}{lll} \neg A_1 & \vee\ A_2 & \vee\neg A_3 \\ \ \ A_1 & \vee\neg A_3 & \vee\neg A_4 \\ \neg A_2 & \vee\neg A_4 & \\ \ \ A_3 & \vee\neg A_4 & \\ \ \ A_4 & & \end{array}$$

$\mu := \{A_4 := \top\}$

Example

$$
\begin{array}{rlll}
\neg A_1 & \vee & A_2 & \vee \neg A_3 \\
A_1 & \vee \neg A_3 & & \vee \neg A_4 \\
\neg A_2 & \vee \neg A_4 & & \\
A_3 & \vee \neg A_4 & & \\
A_4 & & &
\end{array}
$$

$\mu := \{A_4 := \top, A_3 := \top\}$

Example

$$\begin{array}{lll} \neg A_1 & \vee\ A_2 & \vee \neg A_3 \\ A_1 & \vee \neg A_3 & \vee \neg A_4 \\ \neg A_2 & \vee \neg A_4 & \\ A_3 & \vee \neg A_4 & \\ A_4 & & \end{array}$$

$\mu := \{A_4 := \top, A_3 := \top, A_2 := \bot\}$

Example

$$
\begin{array}{rlll}
\neg A_1 & \vee \; A_2 & \vee \neg A_3 & \times \\
A_1 & \vee \neg A_3 & \vee \neg A_4 & \\
\neg A_2 & \vee \neg A_4 & & \\
A_3 & \vee \neg A_4 & & \\
A_4 & & &
\end{array}
$$

$$\mu := \{A_4 := \top, A_3 := \top, A_2 := \bot, A_1 := \top\} \Longrightarrow \text{UNSAT}$$

Example 2

$$\begin{array}{lll} A_1 & \vee \neg A_2 & \\ A_2 & \vee \neg A_5 & \vee \neg A_4 \\ A_4 & \vee \neg A_3 & \\ A_3 & & \end{array}$$

Example 2

$$\begin{array}{lll} A_1 & \vee \neg A_2 & \\ A_2 & \vee \neg A_5 & \vee \neg A_4 \\ A_4 & \vee \neg A_3 & \\ A_3 & & \end{array}$$

$\mu := \{A_3 := \top\}$

Example 2

$$\begin{array}{lll} A_1 & \vee \neg A_2 & \\ A_2 & \vee \neg A_5 & \vee \neg A_4 \\ A_4 & \vee \neg A_3 & \\ A_3 & & \end{array}$$

$\mu := \{A_3 := \top, A_4 := \top\}$

Example 2

$$\begin{array}{lll} A_1 & \vee \neg A_2 & \\ A_2 & \vee \neg A_5 & \vee \neg A_4 \\ A_4 & \vee \neg A_3 & \\ A_3 & & \end{array}$$

$\mu := \{A_3 := \top, A_4 := \top\} \Longrightarrow$ SAT

2-CNF Formulas

- A 2-CNF formula is a CNF formula in which each clause has (at most) two literals.

  $A_1 \vee \neg A_2$
  $A_2 \vee \neg A_3$
  $\neg A_5 \vee \neg A_3$
  $A_3 \vee \neg A_1$
  A_5
- Checking the satisfiability of 2-CNF formulas requires polynomial time

Tractability of 2-CNF Formulas

Graph-based approach:

(i) Build the implication graph of the formula

(ii) check if it has a cycle containing both A_i and $\neg A_i$ for some i (e.g., by Tarjan's algorithm)
$\Longrightarrow$ the formula is unsatisfiable iff such cycle exists

- requires linear time

Tractability of 2-CNF Formulas

Graph-based approach:

(i) Build the implication graph of the formula

(ii) check if it has a cycle containing both A_i and $\neg A_i$ for some i (e.g., by Tarjan's algorithm)
$\Longrightarrow$ the formula is unsatisfiable iff such cycle exists

- requires linear time

Tractability of 2-CNF Formulas

Graph-based approach:

(i) Build the implication graph of the formula

(ii) check if it has a cycle containing both A_i and $\neg A_i$ for some i (e.g., by Tarjan's algorithm)
$\Longrightarrow$ the formula is unsatisfiable iff such cycle exists

- requires linear time

Tractability of 2-CNF Formulas

Graph-based approach:

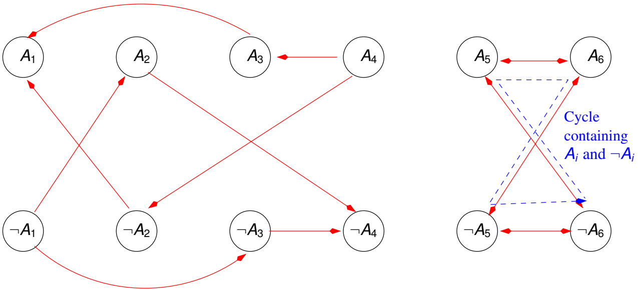
(i) Build the implication graph of the formula

(ii) check if it has a cycle containing both A_i and $\neg A_i$ for some i (e.g., by Tarjan's algorithm)
$\Longrightarrow$ the formula is unsatisfiable iff such cycle exists

- requires linear time

Example:

$A_1 \vee A_2$
$A_1 \vee \neg A_3$
$\neg A_2 \vee \neg A_4$
$A_3 \vee \neg A_4$
A_4
$\neg A_5 \vee A_6$
$A_5 \vee A_6$
$A_5 \vee \neg A_6$
$\neg A_5 \vee \neg A_6$



Cycle containing A_i and $\neg A_i$

Tractability of 2-CNF Formulas

Idea

Let φ, l s.t. $var(l) \in \varphi$ and $(\varphi \wedge l) \not\models_{BCP} \bot$.

- φ': clauses remained after BCP
- φ'': clauses removed by BCP

Suppose φ' is UNSAT. Can we conclude anything about φ?

- Case φ is $>$2-CNF: No!
  - there may be (non-unit) clauses $C \in \varphi'$ s.t. $(\neg l \vee C) \in \varphi$

  $\Longrightarrow$ $\varphi \neq \varphi' \wedge \varphi''$ and $\varphi' \models \bot \not\Longrightarrow \varphi \models \bot$

  $\Longrightarrow$ we must check also $\varphi \wedge \neg l$
- Case φ is 2-CNF: Yes!
  - there cannot be clause $C \in \varphi'$ s.t. $(\neg l \vee C) \in \varphi$

  $\Longrightarrow$ $\varphi = \varphi' \wedge \varphi''$ and $\varphi' \models \bot \Longrightarrow \varphi \models \bot$

  $\Longrightarrow$ φ is UNSAT

Note: we need to check first that $(\varphi \wedge l) \not\models_{BCP} \bot$:
If $(\varphi \wedge l) \models_{BCP} \bot$, then $\varphi' \models \bot \not\Longrightarrow \varphi \models \bot$ (see later Example 2).

Tractability of 2-CNF Formulas

Idea

Let φ, l s.t. $var(l) \in \varphi$ and $(\varphi \wedge l) \not\models_{BCP} \bot$.

- φ': clauses remained after BCP
- φ'': clauses removed by BCP

Suppose φ' is UNSAT. Can we conclude anything about φ?

- Case φ is $>$2-CNF: No!
  - there may be (non-unit) clauses $C \in \varphi'$ s.t. $(\neg l \vee C) \in \varphi$

$\Longrightarrow$ $\varphi \neq \varphi' \wedge \varphi''$ and $\varphi' \models \bot \not\Longrightarrow \varphi \models \bot$
$\Longrightarrow$ we must check also $\varphi \wedge \neg l$

- Case φ is 2-CNF: Yes!
  - there cannot be clause $C \in \varphi'$ s.t. $(\neg l \vee C) \in \varphi$

$\Longrightarrow$ $\varphi = \varphi' \wedge \varphi''$ and $\varphi' \models \bot \Longrightarrow \varphi \models \bot$
$\Longrightarrow$ φ is UNSAT

Note: we need to check first that $(\varphi \wedge l) \not\models_{BCP} \bot$:
If $(\varphi \wedge l) \models_{BCP} \bot$, then $\varphi' \models \bot \not\Longrightarrow \varphi \models \bot$ (see later Example 2).

Tractability of 2-CNF Formulas

Idea

Let φ, l s.t. $var(l) \in \varphi$ and $(\varphi \wedge l) \not\models_{BCP} \bot$.

- φ': clauses remained after BCP
- φ'': clauses removed by BCP

Suppose φ' is UNSAT. Can we conclude anything about φ?

- Case φ is >2-CNF: No!
  - there may be (non-unit) clauses $C \in \varphi'$ s.t. $(\neg l \vee C) \in \varphi$

  $\Longrightarrow$ $\varphi \neq \varphi' \wedge \varphi''$ and $\varphi' \models \bot \not\Longrightarrow \varphi \models \bot$

  $\Longrightarrow$ we must check also $\varphi \wedge \neg l$
- Case φ is 2-CNF: Yes!
  - there cannot be clause $C \in \varphi'$ s.t. $(\neg l \vee C) \in \varphi$

  $\Longrightarrow$ $\varphi = \varphi' \wedge \varphi''$ and $\varphi' \models \bot \Longrightarrow \varphi \models \bot$

  $\Longrightarrow$ φ is UNSAT

Note: we need to check first that $(\varphi \wedge l) \not\models_{BCP} \bot$:
If $(\varphi \wedge l) \models_{BCP} \bot$, then $\varphi' \models \bot \not\Longrightarrow \varphi \models \bot$ (see later Example 2).

Tractability of 2-CNF Formulas

Idea

Let φ, l s.t. $var(l) \in \varphi$ and $(\varphi \wedge l) \not\models_{BCP} \bot$.

- φ': clauses remained after BCP
- φ'': clauses removed by BCP

Suppose φ' is UNSAT. Can we conclude anything about φ?

- Case φ is $>$2-CNF: No!
  - there may be (non-unit) clauses $C \in \varphi'$ s.t. $(\neg l \vee C) \in \varphi$

  $\Longrightarrow$ $\varphi \neq \varphi' \wedge \varphi''$ and $\varphi' \models \bot \not\Longrightarrow \varphi \models \bot$

  $\Longrightarrow$ we must check also $\varphi \wedge \neg l$
- Case φ is 2-CNF: Yes!
  - there cannot be clause $C \in \varphi'$ s.t. $(\neg l \vee C) \in \varphi$

  $\Longrightarrow$ $\varphi = \varphi' \wedge \varphi''$ and $\varphi' \models \bot \Longrightarrow \varphi \models \bot$

  $\Longrightarrow$ φ is UNSAT

Note: we need to check first that $(\varphi \wedge l) \not\models_{BCP} \bot$:
If $(\varphi \wedge l) \models_{BCP} \bot$, then $\varphi' \models \bot \not\Longrightarrow \varphi \models \bot$ (see later Example 2).

A simple polynomial procedure for 2-SAT

```
function 2_SAT(formula $\varphi$, assignment & $\mu$) {
    Unit_Propagate($\varphi, \mu$);
    if ($\varphi == \bot$) then return UNSAT;
    if ($\varphi == \top$) then return SAT;
    while True do {
        {choose some literal l occurring in $\varphi$};
        save($\varphi, \mu$);
        $\varphi := \varphi \wedge l$;
        Unit_Propagate($\varphi, \mu$);
        if ($\varphi == \bot$) then {
            retrieve($\varphi, \mu$);
            $\varphi = \varphi \wedge \neg l$;
            Unit_Propagate($\varphi, \mu$); }
        if ($\varphi == \bot$) then return UNSAT;
        if ($\varphi == \top$) then return SAT;
} };
```

Example

$$\begin{array}{rl}
A_1 & \vee\ \ A_2 \\
A_1 & \vee \neg A_3 \\
\neg A_2 & \vee \neg A_4 \\
A_3 & \vee \neg A_4 \\
A_4 & \\
\neg A_5 & \vee\ \ A_6 \\
A_5 & \vee\ \ A_6 \\
A_5 & \vee \neg A_6 \\
\neg A_5 & \vee \neg A_6
\end{array}$$

Example

$$\begin{array}{rcl}
A_1 & \vee & A_2 \\
A_1 & \vee & \neg A_3 \\
\neg A_2 & \vee & \neg A_4 \\
A_3 & \vee & \neg A_4 \\
A_4 & & \\
\neg A_5 & \vee & A_6 \\
A_5 & \vee & A_6 \\
A_5 & \vee & \neg A_6 \\
\neg A_5 & \vee & \neg A_6
\end{array}$$

$\mu := \{A_4 := \top\}$

Example

$$\begin{array}{rcl} A_1 & \vee & A_2 \\ A_1 & \vee & \neg A_3 \\ \neg A_2 & \vee & \neg A_4 \\ A_3 & \vee & \neg A_4 \\ A_4 & & \\ \neg A_5 & \vee & A_6 \\ A_5 & \vee & A_6 \\ A_5 & \vee & \neg A_6 \\ \neg A_5 & \vee & \neg A_6 \end{array}$$

$\mu := \{A_4 := \top, A_3 := \top\}$

Example

$$
\begin{array}{rl}
A_1 & \vee\ \ A_2 \\
A_1 & \vee \neg A_3 \\
\neg A_2 & \vee \neg A_4 \\
A_3 & \vee \neg A_4 \\
A_4 & \\
\neg A_5 & \vee\ \ A_6 \\
A_5 & \vee\ \ A_6 \\
A_5 & \vee \neg A_6 \\
\neg A_5 & \vee \neg A_6
\end{array}
$$

$\mu := \{A_4 := \top, A_3 := \top, A_2 := \bot\}$

Example

$$
\begin{array}{rl}
A_1 & \vee \ A_2 \\
A_1 & \vee \neg A_3 \\
\neg A_2 & \vee \neg A_4 \\
A_3 & \vee \neg A_4 \\
A_4 & \\
\neg A_5 & \vee \ A_6 \\
A_5 & \vee \ A_6 \\
A_5 & \vee \neg A_6 \\
\neg A_5 & \vee \neg A_6
\end{array}
$$

$\mu := \{A_4 := \top, A_3 := \top, A_2 := \bot, A_6 := \bot\}$ (Select $\neg A_6$)

Example

$$\begin{array}{rl} A_1 & \vee\ A_2 \\ A_1 & \vee \neg A_3 \\ \neg A_2 & \vee \neg A_4 \\ A_3 & \vee \neg A_4 \\ A_4 & \\ \neg A_5 & \vee\ A_6 \\ A_5 & \vee\ A_6 \quad \times \\ A_5 & \vee \neg A_6 \\ \neg A_5 & \vee \neg A_6 \end{array}$$

$\mu := \{A_4 := \top, A_3 := \top, A_2 := \bot, A_6 := \bot, A_5 := \bot\} \Longrightarrow$ backtrack

Example

$$
\begin{array}{rl}
A_1 & \vee \ A_2 \\
A_1 & \vee \neg A_3 \\
\neg A_2 & \vee \neg A_4 \\
A_3 & \vee \neg A_4 \\
A_4 & \\
\neg A_5 & \vee \ A_6 \\
A_5 & \vee \ A_6 \\
A_5 & \vee \neg A_6 \\
\neg A_5 & \vee \neg A_6
\end{array}
$$

$\mu := \{A_4 := \top, A_3 := \top, A_2 := \bot, A_6 := \top\}$ (Select A_6)

Example

$$\begin{array}{rcll}
A_1 & \vee & A_2 & \\
A_1 & \vee & \neg A_3 & \\
\neg A_2 & \vee & \neg A_4 & \\
A_3 & \vee & \neg A_4 & \\
A_4 & & & \\
\neg A_5 & \vee & A_6 & \\
A_5 & \vee & A_6 & \\
A_5 & \vee & \neg A_6 & \times \\
\neg A_5 & \vee & \neg A_6 &
\end{array}$$

$$\mu := \{A_4 := \top, A_3 := \top, A_2 := \bot, A_6 := \top, A_5 := \top\} \Longrightarrow \text{UNSAT}$$

Example 2

$$\begin{array}{rl}
A_1 & \vee A_2 \\
A_1 & \vee \neg A_3 \\
\neg A_2 & \vee \neg A_4 \\
A_3 & \vee \neg A_4 \\
A_4 & \\
\neg A_5 & \vee A_6 \\
A_5 & \vee A_6 \\
A_5 & \vee \neg A_6
\end{array}$$

Example 2

$$\begin{array}{rl}
A_1 & \vee \ A_2 \\
A_1 & \vee \neg A_3 \\
\neg A_2 & \vee \neg A_4 \\
A_3 & \vee \neg A_4 \\
A_4 & \\
\neg A_5 & \vee \ A_6 \\
A_5 & \vee \ A_6 \\
A_5 & \vee \neg A_6
\end{array}$$

$\mu := \{A_4 := \top\}$

Example 2

$$\begin{array}{rl} A_1 & \vee \ A_2 \\ A_1 & \vee \neg A_3 \\ \neg A_2 & \vee \neg A_4 \\ A_3 & \vee \neg A_4 \\ A_4 & \\ \neg A_5 & \vee \ A_6 \\ A_5 & \vee \ A_6 \\ A_5 & \vee \neg A_6 \end{array}$$

$\mu := \{A_4 := \top, A_3 := \top\}$

Example 2

$$\begin{array}{rl} A_1 & \vee \quad A_2 \\ A_1 & \vee \neg A_3 \\ \neg A_2 & \vee \neg A_4 \\ A_3 & \vee \neg A_4 \\ A_4 & \\ \neg A_5 & \vee \quad A_6 \\ A_5 & \vee \quad A_6 \\ A_5 & \vee \neg A_6 \end{array}$$

$\mu := \{A_4 := \top, A_3 := \top, A_2 := \bot\}$

Example 2

$$\begin{array}{rl} A_1 & \vee\ A_2 \\ A_1 & \vee \neg A_3 \\ \neg A_2 & \vee \neg A_4 \\ A_3 & \vee \neg A_4 \\ A_4 & \\ \neg A_5 & \vee\ A_6 \\ A_5 & \vee\ A_6 \\ A_5 & \vee \neg A_6 \end{array}$$

$\mu := \{A_4 := \top, A_3 := \top, A_2 := \bot, A_6 := \bot\}$ (Select $\neg A_6$)

Example 2

$$\begin{array}{rcl} A_1 & \vee & A_2 \\ A_1 & \vee & \neg A_3 \\ \neg A_2 & \vee & \neg A_4 \\ A_3 & \vee & \neg A_4 \\ A_4 & & \\ \neg A_5 & \vee & A_6 \\ A_5 & \vee & A_6 \quad \times \\ A_5 & \vee & \neg A_6 \end{array}$$

$\mu := \{A_4 := \top, A_3 := \top, A_2 := \bot, A_6 := \bot, A_5 := \bot\} \Longrightarrow$ backtrack

Example 2

$$\begin{array}{rl}
A_1 & \vee \ A_2 \\
A_1 & \vee \neg A_3 \\
\neg A_2 & \vee \neg A_4 \\
A_3 & \vee \neg A_4 \\
A_4 & \\
\neg A_5 & \vee \ A_6 \\
A_5 & \vee \ A_6 \\
A_5 & \vee \neg A_6
\end{array}$$

$\mu := \{A_4 := \top, A_3 := \top, A_2 := \bot, A_6 := \top\}$ (Select A_6)

Example 2

$$\begin{array}{rl}
A_1 & \vee A_2 \\
A_1 & \vee \neg A_3 \\
\neg A_2 & \vee \neg A_4 \\
A_3 & \vee \neg A_4 \\
A_4 & \\
\neg A_5 & \vee A_6 \\
A_5 & \vee A_6 \\
A_5 & \vee \neg A_6
\end{array}$$

$\mu := \{A_4 := \top, A_3 := \top, A_2 := \bot, A_6 := \top, A_5 := \top\} \Longrightarrow$ SAT

Outline

The satisfiability of k-CNF (k-SAT) [20]

- k-CNF: CNF s.t. all clauses have *k* literals
- the satisfiability of 2-CNF is polynomial
- the satisfiability of k-CNF is NP-complete for $k \geq 3$
- every k-CNF formula can be converted into 3-CNF:

$$l_1 \lor l_2 \lor \ldots \lor l_{k-1} \lor l_k$$

$$\Downarrow$$

$$\begin{gathered}(l_1 \lor l_2 \lor B_1) \land \\ (\neg B_1 \lor l_3 \lor B_2) \land \\ \ldots \\ (\neg B_{k-4} \lor l_{k-2} \lor B_{k-3}) \land \\ (\neg B_{k-3} \lor l_{k-1} \lor l_k)\end{gathered}$$

The satisfiability of k-CNF (k-SAT) [20]

- k-CNF: CNF s.t. all clauses have k literals
- the satisfiability of 2-CNF is polynomial
- the satisfiability of k-CNF is NP-complete for $k \geq 3$
- every k-CNF formula can be converted into 3-CNF:

$$\begin{gathered} l_1 \vee l_2 \vee \ldots \vee l_{k-1} \vee l_k \\ \Downarrow \\ (l_1 \vee l_2 \vee B_1) \wedge \\ (\neg B_1 \vee l_3 \vee B_2) \wedge \\ \ldots \\ (\neg B_{k-4} \vee l_{k-2} \vee B_{k-3}) \wedge \\ (\neg B_{k-3} \vee l_{k-1} \vee l_k) \end{gathered}$$

Random K-CNF formulas generation

Random k-CNF formulas with N variables and L clauses:

```
DO
    (i) pick with uniform probability a set of $k$ atoms over $N$
    (ii) randomly negate each atom with probability 0.5
    (iii) create a disjunction of the resulting literals
UNTIL $L$ different clauses have been generated;
```

Random K-CNF formulas generation

Random k-CNF formulas with N variables and L clauses:
DO

(i) pick with uniform probability a set of k atoms over N
(ii) randomly negate each atom with probability 0.5
(iii) create a disjunction of the resulting literals

UNTIL L different clauses have been generated;

Random K-CNF formulas generation

```
Random k-CNF formulas with $N$ variables and $L$ clauses:
DO
    (i) pick with uniform probability a set of $k$ atoms over $N$
    (ii) randomly negate each atom with probability 0.5
    (iii) create a disjunction of the resulting literals
UNTIL $L$ different clauses have been generated;
```

Random K-CNF formulas generation

Random k-CNF formulas with N variables and L clauses:

```
DO
  (i) pick with uniform probability a set of $k$ atoms over $N$
  (ii) randomly negate each atom with probability 0.5
  (iii) create a disjunction of the resulting literals
UNTIL $L$ different clauses have been generated;
```

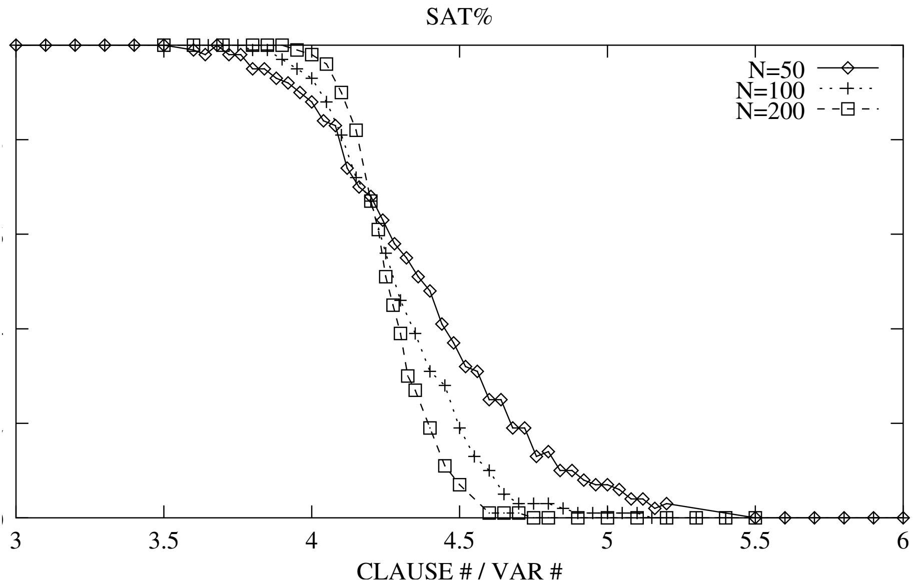
Random k-SAT plots

- fix k and N
- for increasing L, randomly generate and solve (500,1000,10000,...) problems with k, L, N
- plot
  - satisfiability percentages
  - median/geometrical mean CPU time/# of steps

  against L/N

The phase transition phenomenon: SAT % Plots [41, 32]

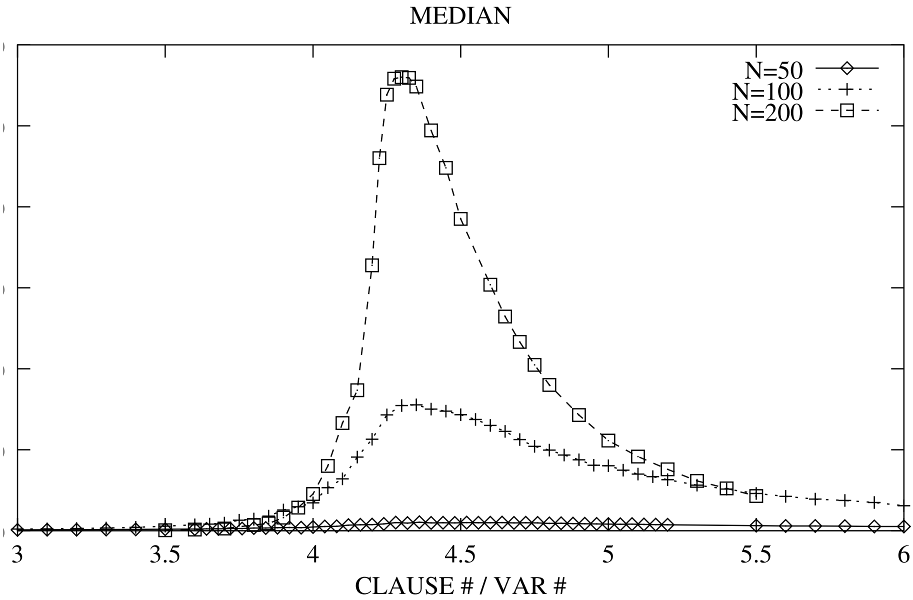
- Increasing L/N we pass from 100% satisfiable to 100% unsatisfiable formulas
- the decay becomes steeper with N
- for $N \to \infty$, the plot converges to a step in the cross-over point ($L/N \approx 4.28$ for k=3)
- Revealed for many other NP-complete problems
- Many theoretical models [59, 21, 32, 16, 42]
- Strong relation with Thermodynamics

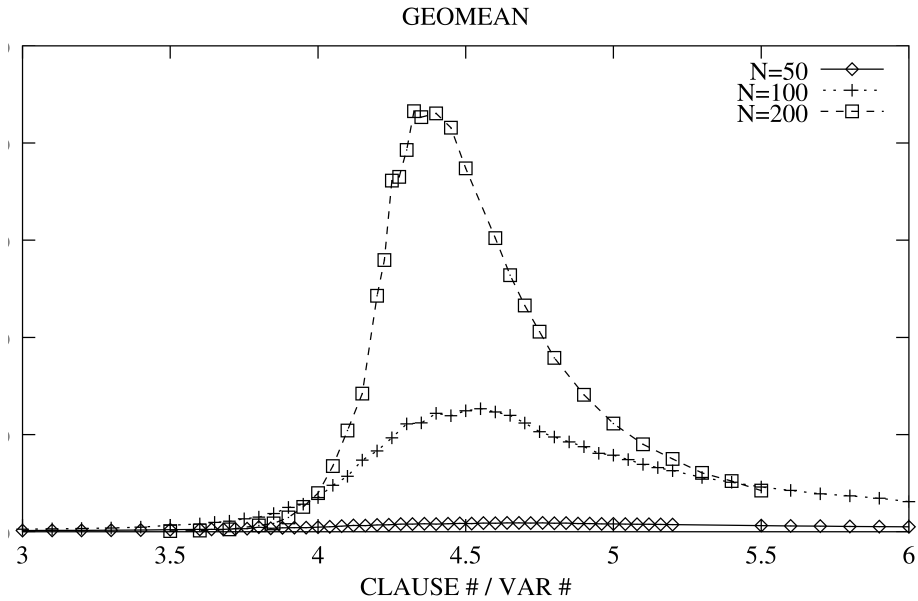

SAT%
N=50
N=100
N=200
3
3.5
4
4.5
5
5.5
6
CLAUSE # / VAR #

The phase transition phenomenon: CPU times/step

Using search algorithms (DPLL):

- Increasing L/N we pass from easy problems, to very hard problems down to hard problems
- the peak is centered in the 50% satisfiable point
- the decay becomes steeper with N
- for $N \to \infty$, the plot converges to an impulse in the cross-over point ($L/N \approx 4.28$ for k=3)
- easy problems ($L/N \leq\approx 3.8$) increase polynomially with N, hard problems increase exponentially with N
- Increasing L/N, satisfiable problems get harder, unsatisfiable problems get easier.


MEDIAN
N=50
N=100
N=200
3
3.5
4
4.5
5
5.5
6
CLAUSE # / VAR #


GEOMEAN
N=50
N=100
N=200
3
3.5
4
4.5
5
5.5
6
CLAUSE # / VAR #

Outline

1. Basics on SAT
2. Basic SAT-Solving techniques
3. Modern CDCL SAT Solvers
    - Conflict-Driven Clause-Learning SAT solvers
    - Further Improvements
4. Tractable subclasses of SAT
5. Random k-SAT and Phase Transition
6. Advanced Functionalities: proofs, unsat cores, interpolants, optimization
7. Some Applications
    - Appl. #1: (Bounded) Planning
    - Appl. #2: Bounded Model Checking

Advanced functionalities

Advanced SAT functionalities (very important in formal verification):

- Computing SAT under assumptions & Incremental SAT solving
- Building proofs of unsatisfiability
- Extracting unsatisfiable Cores
- Computing Craig Interpolants

SAT under assumptions: $SAT(\varphi, \{l_1, ..., l_n\})$ [18]

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

SAT under assumptions: $SAT(\varphi, \{l_1, ..., l_n\})$ [18]

- Many SAT solvers allow for solving a CNF formula φ under a set of assumption literals $\mathcal{A} \overset{\text{def}}{=} \{l_1, ..., l_n\}$: $SAT(\varphi, \{l_1, ..., l_n\})$
  - $SAT(\varphi, \{l_1, ..., l_n\})$: 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 $C \overset{\text{def}}{=} \neg\eta$ s.t. $\eta \subseteq \mathcal{A}$, then return UNSAT
  - if the "decision" strategy for conflict analysis is used, then η is the subset of assumptions causing the inconsistency

Selection of sub-formulas

Let φ be $\bigwedge_{i=1}^{n} C_i$.

Idea [18, 35]

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

$\Longrightarrow$ allows for "selecting" (activating) only a subset of the clauses in φ at each call.

Incremental SAT solving [18, 17]

- Many CDCL solvers provide a stack-based incremental interface
  - it is possible to push/pop ϕ_i into a stack of formulas $\Phi \stackrel{\text{def}}{=} \{\phi_1, ..., \phi_k\}$
  - check incrementally the satisfiability of $\bigwedge_{i=1}^{k} \phi_i$.
- 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)
- Simple idea [18, 17]: incremental calls $SAT(\varphi, \mathcal{A}_1), SAT(\varphi, \mathcal{A}_2)$,...
  - $\varphi \stackrel{\text{def}}{=} \bigwedge_i (\neg A_i \vee \phi_i)$, $\mathcal{A}_i \subseteq \{A_1, ..., A_k\}\ \forall\, i$,
  - stack-based interface for $\mathcal{A} \stackrel{\text{def}}{=} \{A_1, A_2, ...\}$

learned clauses safely reused from call to call even if assumptions have been removed

- learned clauses C_j s.t. $\varphi \models C_j$
- C_j may be in the form $\neg A_j \vee C_j'$ s.t. $A_i \not\in \mathcal{A}_i \Longrightarrow C_j$ not reused

Incremental SAT solving [18, 17]

- Many CDCL solvers provide a stack-based incremental interface
  - it is possible to push/pop ϕ_i into a stack of formulas $\Phi \stackrel{\text{def}}{=} \{\phi_1, ..., \phi_k\}$
  - check incrementally the satisfiability of $\bigwedge_{i=1}^{k} \phi_i$.
- 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)
- Simple idea [18, 17]: incremental calls $SAT(\varphi, \mathcal{A}_1)$, $SAT(\varphi, \mathcal{A}_2)$,...
  - $\varphi \stackrel{\text{def}}{=} \bigwedge_i (\neg A_i \vee \phi_i)$, $\mathcal{A}_i \subseteq \{A_1, ..., A_k\}\ \forall\ i$,
  - stack-based interface for $\mathcal{A} \stackrel{\text{def}}{=} \{A_1, A_2, ...\}$

learned clauses safely reused from call to call even if assumptions have been removed

- learned clauses C_j s.t. $\varphi \models C_j$
- C_j may be in the form $\neg A_j \vee C_j'$ s.t. $A_i \not\in \mathcal{A}_i \Longrightarrow C_j$ not reused

Building Proofs of Unsatisfiability

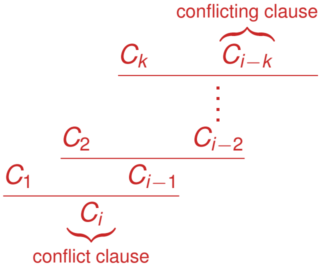
- When φ is unsat, it is very important to build a (resolution) proof of unsatisfiability:
  - to verify the result of the solver
  - to understand a “reason” for unsatisfiability
  - to build unsatisfiable cores and interpolants
- can be built by keeping track of the resolution steps performed when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

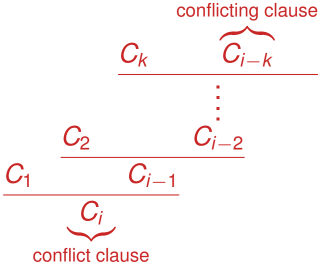


- $C_1, ..., C_k$, and C_{i-k} can be original or learned clauses
- each resolution (sub)proof can be easily tracked:

```
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

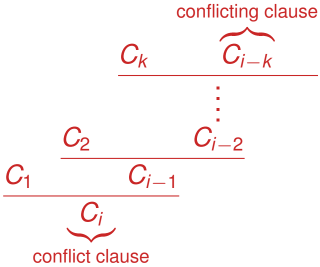


- $C_1, ..., C_k$, and C_{i-k} can be original or learned clauses
- each resolution (sub)proof can be easily tracked:

```
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

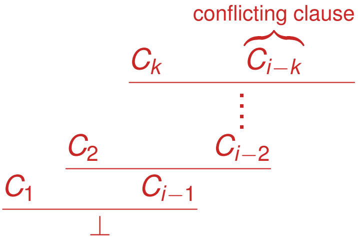


- $C_1, ..., C_k$, and C_{i-k} can be original or learned clauses
- each resolution (sub)proof can be easily tracked:

```
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:



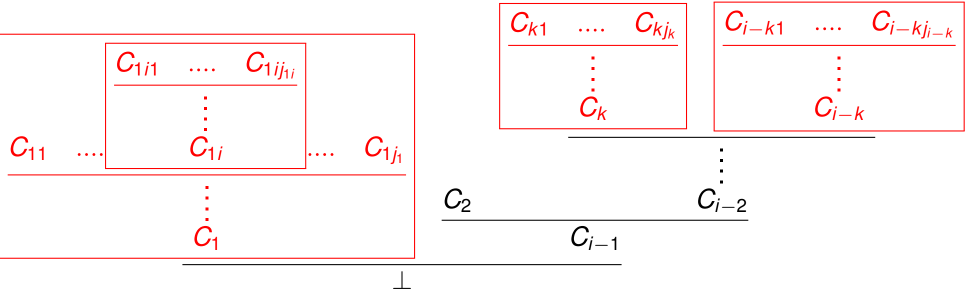
- note: $C_1 = l$, $C_{i-1} = \neg l$ for some literal l
- $C_1, ..., C_k$, and C_{i-k} can be original or learned clauses...

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

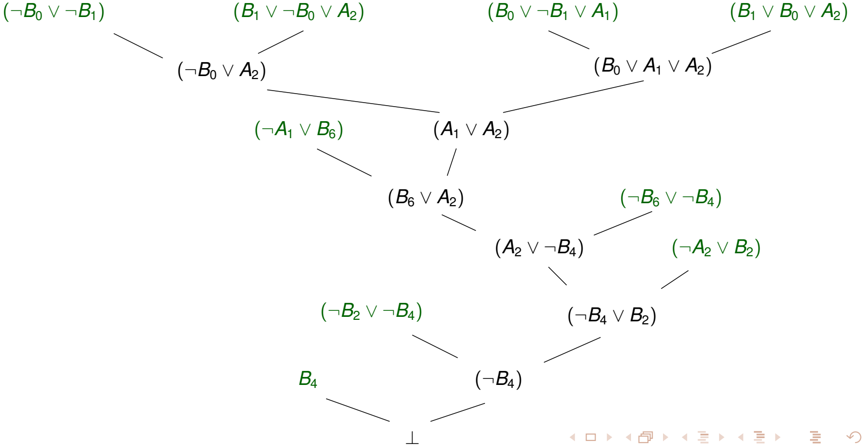
until all leaf clauses are original clauses



$\Longrightarrow$ we obtain a resolution proof of unsatisfiability for (a subset of) the clauses in φ

Building Proofs of Unsatisfiability: example

$$(B_0 \vee \neg B_1 \vee A_1) \wedge (B_0 \vee B_1 \vee A_2) \wedge (\neg B_0 \vee B_1 \vee A_2) \wedge (\neg B_0 \vee \neg B_1) \wedge (\neg B_2 \vee \neg B_4) \wedge$$
$$(\neg A_2 \vee B_2) \wedge (\neg A_1 \vee B_3) \wedge B_4 \wedge (A_2 \vee B_5) \wedge (\neg B_6 \vee \neg B_4) \wedge (B_6 \vee \neg A_1) \wedge B_7$$



$(\neg B_0 \vee \neg B_1)$ $(B_1 \vee \neg B_0 \vee A_2)$ $(B_0 \vee \neg B_1 \vee A_1)$ $(B_1 \vee B_0 \vee A_2)$

$(\neg B_0 \vee A_2)$ $(B_0 \vee A_1 \vee A_2)$

$(\neg A_1 \vee B_6)$ $(A_1 \vee A_2)$

$(B_6 \vee A_2)$ $(\neg B_6 \vee \neg B_4)$

$(A_2 \vee \neg B_4)$ $(\neg A_2 \vee B_2)$

$(\neg B_2 \vee \neg B_4)$ $(\neg B_4 \vee B_2)$

B_4 $(\neg B_4)$

$\perp$

Extraction of unsatisfiable cores

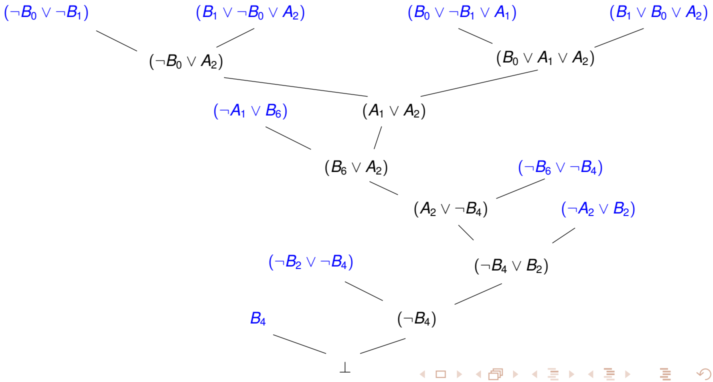
- Problem: given an unsatisfiable set of clauses, extract from it a (possibly small/minimal/minimum) unsatisfiable subset

  $\Longrightarrow$ unsatisfiable cores (aka (Minimal) Unsatisfiable Subsets, (M)US)
- Lots of literature on the topic [65, 36, 39, 46, 62, 28, 22, 10]
- We recognize two main approaches:
  - Proof-based approach [65]: byproduct of finding a resolution proof
  - Assumption-based approach [36]: 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 unsat-core extraction [65]

Unsat core: the set of leaf clauses of a resolution proof

$(B_0 \vee \neg B_1 \vee A_1) \wedge (B_0 \vee B_1 \vee A_2) \wedge (\neg B_0 \vee B_1 \vee A_2) \wedge (\neg B_0 \vee \neg B_1) \wedge (\neg B_2 \vee \neg B_4) \wedge$
$(\neg A_2 \vee B_2) \wedge (\neg A_1 \vee B_3) \wedge B_4 \wedge (A_2 \vee B_5) \wedge (\neg B_6 \vee \neg B_4) \wedge (B_6 \vee \neg A_1) \wedge B_7$



The assumption-based approach to unsat-core extraction [36]

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 unsat-core extraction [36]

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 unsat-core extraction [36]

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 unsat-core extraction [36]

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 unsat-core extraction [36]

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 unsat-core extraction

$(B_0 \vee \neg B_1 \vee A_1) \wedge (B_0 \vee B_1 \vee A_2) \wedge (\neg B_0 \vee B_1 \vee A_2) \wedge$
$(\neg B_0 \vee \neg B_1) \wedge (\neg B_2 \vee \neg B_4) \wedge (\neg A_2 \vee B_2) \wedge (\neg A_1 \vee B_3) \wedge$
$B_4 \wedge (A_2 \vee B_5) \wedge (\neg B_6 \vee \neg B_4) \wedge (B_6 \vee \neg A_1) \wedge B_7$

(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 unsat-core extraction

$(B_0 \vee \neg B_1 \vee A_1) \wedge (B_0 \vee B_1 \vee A_2) \wedge (\neg B_0 \vee B_1 \vee A_2) \wedge$
$(\neg B_0 \vee \neg B_1) \wedge (\neg B_2 \vee \neg B_4) \wedge (\neg A_2 \vee B_2) \wedge (\neg A_1 \vee B_3) \wedge$
$B_4 \wedge (A_2 \vee B_5) \wedge (\neg B_6 \vee \neg B_4) \wedge (B_6 \vee \neg A_1) \wedge B_7$

(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 unsat-core extraction

$(B_0 \vee \neg B_1 \vee A_1) \wedge (B_0 \vee B_1 \vee A_2) \wedge (\neg B_0 \vee B_1 \vee A_2) \wedge$
$(\neg B_0 \vee \neg B_1) \wedge (\neg B_2 \vee \neg B_4) \wedge (\neg A_2 \vee B_2) \wedge (\neg A_1 \vee B_3) \wedge$
$B_4 \wedge (A_2 \vee B_5) \wedge (\neg B_6 \vee \neg B_4) \wedge (B_6 \vee \neg A_1) \wedge B_7$

(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 unsat-core extraction

$$(B_0 \vee \neg B_1 \vee A_1) \wedge (B_0 \vee B_1 \vee A_2) \wedge (\neg B_0 \vee B_1 \vee A_2) \wedge$$
$$(\neg B_0 \vee \neg B_1) \wedge (\neg B_2 \vee \neg B_4) \wedge (\neg A_2 \vee B_2) \wedge (\neg A_1 \vee B_3) \wedge$$
$$B_4 \wedge (A_2 \vee B_5) \wedge (\neg B_6 \vee \neg B_4) \wedge (B_6 \vee \neg A_1) \wedge B_7$$

(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:

$$(B_0 \vee \neg B_1 \vee A_1) \wedge (B_0 \vee B_1 \vee A_2) \wedge (\neg B_0 \vee B_1 \vee A_2) \wedge$$
$$(\neg B_0 \vee \neg B_1) \wedge (\neg B_2 \vee \neg B_4) \wedge (\neg A_2 \vee B_2) \wedge$$
$$B_4 \wedge (\neg B_6 \vee \neg B_4) \wedge (B_6 \vee \neg A_1)$$

Computing Craig Interpolants in SAT

Let "$X \preceq Y$", X, Y being Boolean formulas, denote the fact that all Boolean atoms in X occur also in Y.

Definition: Craig Interpolant

Given an ordered pair (A, B) of formulas such that $A \wedge B \models \perp$, a *Craig interpolant* is a formula I s.t.:

a) $A \models I$,

b) $I \wedge B \models \perp$,

c) $I \preceq A$ and $I \preceq B$.

- Very important in many Formal Verification applications
- A few works presented [47, 38, 40]

Computing Craig Interpolants in SAT

Let "$X \preceq Y$", X, Y being Boolean formulas, denote the fact that all Boolean atoms in X occur also in Y.

Definition: Craig Interpolant

Given an ordered pair (A, B) of formulas such that $A \wedge B \models \bot$, a *Craig interpolant* is a formula I s.t.:

a) $A \models I$,

b) $I \wedge B \models \bot$,

c) $I \preceq A$ and $I \preceq B$.

- Very important in many Formal Verification applications
- A few works presented [47, 38, 40]

Computing Craig Interpolants in SAT

Let "$X \preceq Y$", X, Y being Boolean formulas, denote the fact that all Boolean atoms in X occur also in Y.

Definition: Craig Interpolant

Given an ordered pair (A, B) of formulas such that $A \wedge B \models \bot$, a *Craig interpolant* is a formula I s.t.:

a) $A \models I$,

b) $I \wedge B \models \bot$,

c) $I \preceq A$ and $I \preceq B$.

- Very important in many Formal Verification applications
- A few works presented [47, 38, 40]

Computing Craig Interpolants in SAT

Let "$X \preceq Y$", X, Y being Boolean formulas, denote the fact that all Boolean atoms in X occur also in Y.

Definition: Craig Interpolant

Given an ordered pair (A, B) of formulas such that $A \wedge B \models \bot$, a *Craig interpolant* is a formula I s.t.:

a) $A \models I$,

b) $I \wedge B \models \bot$,

c) $I \preceq A$ and $I \preceq B$.

- Very important in many Formal Verification applications
- A few works presented [47, 38, 40]

Computing Craig Interpolants in SAT

Let “$X \preceq Y$”, X, Y being Boolean formulas, denote the fact that all Boolean atoms in X occur also in Y.

Definition: Craig Interpolant

Given an ordered pair (A, B) of formulas such that $A \wedge B \models \bot$, a *Craig interpolant* is a formula I s.t.:

a) $A \models I$,

b) $I \wedge B \models \bot$,

c) $I \preceq A$ and $I \preceq B$.

- Very important in many Formal Verification applications
- A few works presented [47, 38, 40]

Computing Craig Interpolants in SAT

Let "$X \preceq Y$", X, Y being Boolean formulas, denote the fact that all Boolean atoms in X occur also in Y.

Definition: Craig Interpolant

Given an ordered pair (A, B) of formulas such that $A \wedge B \models \perp$, a *Craig interpolant* is a formula I s.t.:

a) $A \models I$,

b) $I \wedge B \models \perp$,

c) $I \preceq A$ and $I \preceq B$.

- Very important in many Formal Verification applications
- A few works presented [47, 38, 40]

Computing Craig Interpolants in SAT

Let "$X \preceq Y$", X, Y being Boolean formulas, denote the fact that all Boolean atoms in X occur also in Y.

Definition: Craig Interpolant

Given an ordered pair (A, B) of formulas such that $A \wedge B \models \bot$, a *Craig interpolant* is a formula I s.t.:

a) $A \models I$,

b) $I \wedge B \models \bot$,

c) $I \preceq A$ and $I \preceq B$.

- Very important in many Formal Verification applications
- A few works presented [47, 38, 40]

Computing Craig Interpolants in SAT: a General Algorithm [47]

Algorithm: Interpolant generation (for SAT)

(i) Generate a resolution proof of unsatisfiability $\mathcal{P}$ for $A \wedge B$.

(ii) ...

(iii) For every leaf clause C in $\mathcal{P}$, set $I_C \stackrel{\text{def}}{=} C \downarrow B$ if $C \in A$, and $I_C \stackrel{\text{def}}{=} \top$ if $C \in B$.

(iv) For every inner node C of $\mathcal{P}$ obtained by resolution from $C_1 \stackrel{\text{def}}{=} p \vee \phi_1$ and $C_2 \stackrel{\text{def}}{=} \neg p \vee \phi_2$, set $I_C \stackrel{\text{def}}{=} I_{C_1} \vee I_{C_2}$ if p does not occur in B, and $I_C \stackrel{\text{def}}{=} I_{C_1} \wedge I_{C_2}$ otherwise.

(v) Output $I_\perp$ as an interpolant for (A, B).

"$\eta \setminus B$" [resp. "$\eta \downarrow B$"] is the set of literals in η whose atoms do not [resp. do] occur in B.

- optimized versions for the purely-propositional case [38, 40]

Computing Craig Interpolants in SAT: a General Algorithm [47]

Algorithm: Interpolant generation (for SAT)

(i) Generate a resolution proof of unsatisfiability $\mathcal{P}$ for $A \wedge B$.

(ii) ...

(iii) For every leaf clause C in $\mathcal{P}$, set $I_C \stackrel{\text{def}}{=} C \downarrow B$ if $C \in A$, and $I_C \stackrel{\text{def}}{=} \top$ if $C \in B$.

(iv) For every inner node C of $\mathcal{P}$ obtained by resolution from $C_1 \stackrel{\text{def}}{=} p \vee \phi_1$ and $C_2 \stackrel{\text{def}}{=} \neg p \vee \phi_2$, set $I_C \stackrel{\text{def}}{=} I_{C_1} \vee I_{C_2}$ if p does not occur in B, and $I_C \stackrel{\text{def}}{=} I_{C_1} \wedge I_{C_2}$ otherwise.

(v) Output $I_\perp$ as an interpolant for (A, B).

"$\eta \setminus B$" [resp. "$\eta \downarrow B$"] is the set of literals in η whose atoms do not [resp. do] occur in B.

- optimized versions for the purely-propositional case [38, 40]

Computing Craig Interpolants in SAT: a General Algorithm [47]

Algorithm: Interpolant generation (for SAT)

(i) Generate a resolution proof of unsatisfiability $\mathcal{P}$ for $A \wedge B$.

(ii) ...

(iii) For every leaf clause C in $\mathcal{P}$, set $I_C \stackrel{\text{def}}{=} C \downarrow B$ if $C \in A$, and $I_C \stackrel{\text{def}}{=} \top$ if $C \in B$.

(iv) For every inner node C of $\mathcal{P}$ obtained by resolution from $C_1 \stackrel{\text{def}}{=} p \vee \phi_1$ and $C_2 \stackrel{\text{def}}{=} \neg p \vee \phi_2$, set $I_C \stackrel{\text{def}}{=} I_{C_1} \vee I_{C_2}$ if p does not occur in B, and $I_C \stackrel{\text{def}}{=} I_{C_1} \wedge I_{C_2}$ otherwise.

(v) Output $I_\perp$ as an interpolant for (A, B).

"$\eta \setminus B$" [resp. "$\eta \downarrow B$"] is the set of literals in η whose atoms do not [resp. do] occur in B.

- optimized versions for the purely-propositional case [38, 40]

Computing Craig Interpolants in SAT: a General Algorithm [47]

Algorithm: Interpolant generation (for SAT)

(i) Generate a resolution proof of unsatisfiability $\mathcal{P}$ for $A \wedge B$.

(ii) ...

(iii) For every leaf clause C in $\mathcal{P}$, set $I_C \stackrel{\text{def}}{=} C \downarrow B$ if $C \in A$, and $I_C \stackrel{\text{def}}{=} \top$ if $C \in B$.

(iv) For every inner node C of $\mathcal{P}$ obtained by resolution from $C_1 \stackrel{\text{def}}{=} p \vee \phi_1$ and $C_2 \stackrel{\text{def}}{=} \neg p \vee \phi_2$, set $I_C \stackrel{\text{def}}{=} I_{C_1} \vee I_{C_2}$ if p does not occur in B, and $I_C \stackrel{\text{def}}{=} I_{C_1} \wedge I_{C_2}$ otherwise.

(v) Output $I_\perp$ as an interpolant for (A, B).

"$\eta \setminus B$" [resp. "$\eta \downarrow B$"] is the set of literals in η whose atoms do not [resp. do] occur in B.

- optimized versions for the purely-propositional case [38, 40]

Computing Craig Interpolants in SAT: a General Algorithm [47]

Algorithm: Interpolant generation (for SAT)

(i) Generate a resolution proof of unsatisfiability $\mathcal{P}$ for $A \wedge B$.

(ii) ...

(iii) For every leaf clause C in $\mathcal{P}$, set $I_C \overset{\text{def}}{=} C \downarrow B$ if $C \in A$, and $I_C \overset{\text{def}}{=} \top$ if $C \in B$.

(iv) For every inner node C of $\mathcal{P}$ obtained by resolution from $C_1 \overset{\text{def}}{=} p \vee \phi_1$ and $C_2 \overset{\text{def}}{=} \neg p \vee \phi_2$, set $I_C \overset{\text{def}}{=} I_{C_1} \vee I_{C_2}$ if p does not occur in B, and $I_C \overset{\text{def}}{=} I_{C_1} \wedge I_{C_2}$ otherwise.

(v) Output $I_\perp$ as an interpolant for (A, B).

"$\eta \setminus B$" [resp. "$\eta \downarrow B$"] is the set of literals in η whose atoms do not [resp. do] occur in B.

- optimized versions for the purely-propositional case [38, 40]

Computing Craig Interpolants in SAT: a General Algorithm [47]

Algorithm: Interpolant generation (for SAT)

(i) Generate a resolution proof of unsatisfiability $\mathcal{P}$ for $A \wedge B$.

(ii) ...

(iii) For every leaf clause C in $\mathcal{P}$, set $I_C \stackrel{\text{def}}{=} C \downarrow B$ if $C \in A$, and $I_C \stackrel{\text{def}}{=} \top$ if $C \in B$.

(iv) For every inner node C of $\mathcal{P}$ obtained by resolution from $C_1 \stackrel{\text{def}}{=} p \vee \phi_1$ and $C_2 \stackrel{\text{def}}{=} \neg p \vee \phi_2$, set $I_C \stackrel{\text{def}}{=} I_{C_1} \vee I_{C_2}$ if p does not occur in B, and $I_C \stackrel{\text{def}}{=} I_{C_1} \wedge I_{C_2}$ otherwise.

(v) Output $I_\perp$ as an interpolant for (A, B).

"$\eta \setminus B$" [resp. "$\eta \downarrow B$"] is the set of literals in η whose atoms do not [resp. do] occur in B.

- optimized versions for the purely-propositional case [38, 40]

Computing Craig Interpolants in SAT: a General Algorithm [47]

Algorithm: Interpolant generation (for SAT)

(i) Generate a resolution proof of unsatisfiability $\mathcal{P}$ for $A \wedge B$.

(ii) ...

(iii) For every leaf clause C in $\mathcal{P}$, set $I_C \stackrel{\text{def}}{=} C \downarrow B$ if $C \in A$, and $I_C \stackrel{\text{def}}{=} \top$ if $C \in B$.

(iv) For every inner node C of $\mathcal{P}$ obtained by resolution from $C_1 \stackrel{\text{def}}{=} p \vee \phi_1$ and $C_2 \stackrel{\text{def}}{=} \neg p \vee \phi_2$, set $I_C \stackrel{\text{def}}{=} I_{C_1} \vee I_{C_2}$ if p does not occur in B, and $I_C \stackrel{\text{def}}{=} I_{C_1} \wedge I_{C_2}$ otherwise.

(v) Output $I_\perp$ as an interpolant for (A, B).

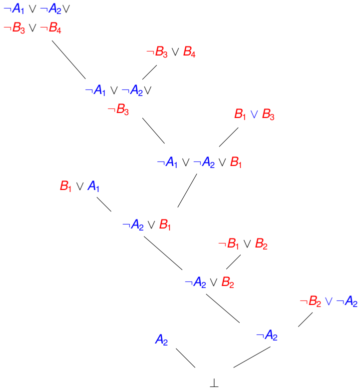
"$\eta \setminus B$" [resp. "$\eta \downarrow B$"] is the set of literals in η whose atoms do not [resp. do] occur in B.

- optimized versions for the purely-propositional case [38, 40]

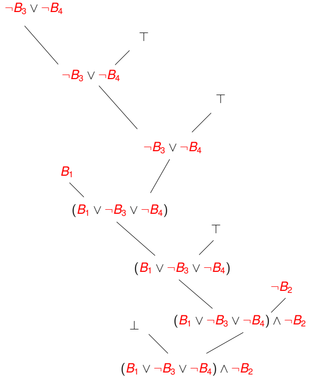
Computing Craig Interpolants in SAT: example

$$A \overset{\text{def}}{=} (B_1 \vee A_1) \wedge A_2 \wedge (\neg B_2 \vee \neg A_2) \wedge (\neg A_1 \vee \neg A_2 \vee \neg B_3 \vee \neg B_4)$$

$$B \overset{\text{def}}{=} (\neg B_3 \vee B_4) \wedge (\neg B_1 \vee B_2) \wedge (B_1 \vee B_3)$$



original proof



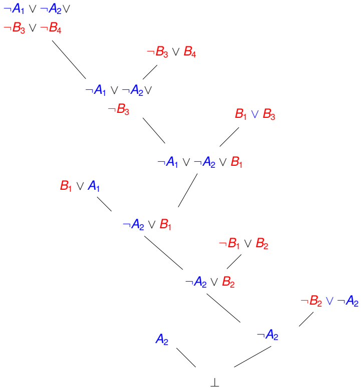
interpolant proof

$\Longrightarrow$ $(B_1 \vee \neg B_3 \vee \neg B_4) \wedge \neg B_2$ is an interpolant

Computing Craig Interpolants in SAT: example

$$A \overset{\text{def}}{=} (B_1 \vee A_1) \wedge A_2 \wedge (\neg B_2 \vee \neg A_2) \wedge (\neg A_1 \vee \neg A_2 \vee \neg B_3 \vee \neg B_4)$$

$$B \overset{\text{def}}{=} (\neg B_3 \vee B_4) \wedge (\neg B_1 \vee B_2) \wedge (B_1 \vee B_3)$$



$\neg A_1 \vee \neg A_2 \vee \neg B_3 \vee \neg B_4$

$\neg B_3 \vee B_4$

$\neg A_1 \vee \neg A_2 \vee \neg B_3$

$B_1 \vee B_3$

$\neg A_1 \vee \neg A_2 \vee B_1$

$B_1 \vee A_1$

$\neg A_2 \vee B_1$

$\neg B_1 \vee B_2$

$\neg A_2 \vee B_2$

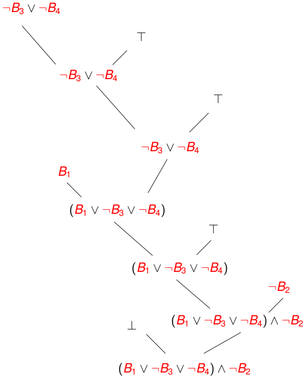
$\neg B_2 \vee \neg A_2$

A_2

$\neg A_2$

$\perp$

original proof



$\neg B_3 \vee \neg B_4$

$\top$

$\neg B_3 \vee \neg B_4$

$\top$

$\neg B_3 \vee \neg B_4$

B_1

$(B_1 \vee \neg B_3 \vee \neg B_4)$

$\top$

$(B_1 \vee \neg B_3 \vee \neg B_4)$

$\neg B_2$

$\perp$

$(B_1 \vee \neg B_3 \vee \neg B_4) \wedge \neg B_2$

$(B_1 \vee \neg B_3 \vee \neg B_4) \wedge \neg B_2$

interpolant proof

$\Longrightarrow$ $(B_1 \vee \neg B_3 \vee \neg B_4) \wedge \neg B_2$ is an interpolant

MaxSAT (hints)

- **MaxSAT:** given a pair of CNF formulas $\langle \varphi_h, \varphi_s \rangle$ s.t. $\varphi_h \wedge \varphi_s \models \bot$, $\varphi_s \overset{\text{def}}{=} \{C_1, ..., C_k\}$, 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. [34])
- EX:

$$\varphi_h \overset{\text{def}}{=} (A_1 \vee A_2) \qquad \varphi_s \overset{\text{def}}{=} \left(\begin{array}{ccc} (\ \ A_1 \vee \neg A_2) & \wedge & [4] \\ (\neg A_1 \vee \ \ A_2) & \wedge & [3] \\ (\neg A_1 \vee \neg A_2) & \wedge & [2] \end{array} \right)$$

$\Longrightarrow \mu = \{A_1, A_2\}$ (penalty = 2)

MaxSAT (hints)

- MaxSAT: given a pair of CNF formulas $\langle \varphi_h, \varphi_s \rangle$ s.t. $\varphi_h \wedge \varphi_s \models \perp$, $\varphi_s \overset{\text{def}}{=} \{C_1, ..., C_k\}$, 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. [34])
- EX:

$$\varphi_h \overset{\text{def}}{=} (A_1 \vee A_2) \qquad \varphi_s \overset{\text{def}}{=} \left(\begin{array}{ccc} (\ \ A_1 \vee \neg A_2) & \wedge & [4] \\ (\neg A_1 \vee \ \ A_2) & \wedge & [3] \\ (\neg A_1 \vee \neg A_2) & \wedge & [2] \end{array} \right)$$

$\Longrightarrow \mu = \{A_1, A_2\}$ (penalty = 2)

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. [34])
- 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. $\varphi_h \wedge \varphi_s \models \bot$, $\varphi_s \overset{\text{def}}{=} \{C_1, ..., C_k\}$, 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. [34])
- EX:

$$\varphi_h \overset{\text{def}}{=} (A_1 \vee A_2) \qquad \varphi_s \overset{\text{def}}{=} \left(\begin{array}{lll} (\ \ A_1 \vee \neg A_2) & \wedge & [4] \\ (\neg A_1 \vee \ \ A_2) & \wedge & [3] \\ (\neg A_1 \vee \neg A_2) & \wedge & [2] \end{array}\right)$$

$\Longrightarrow \mu = \{A_1, A_2\}$ (penalty = 2)

Outline

1. Basics on SAT
2. Basic SAT-Solving techniques
3. Modern CDCL SAT Solvers
    - Conflict-Driven Clause-Learning SAT solvers
    - Further Improvements
4. Tractable subclasses of SAT
5. Random k-SAT and Phase Transition
6. Advanced Functionalities: proofs, unsat cores, interpolants, optimization
7. **Some Applications**
    - Appl. #1: (Bounded) Planning
    - Appl. #2: Bounded Model Checking

Many applications of SAT

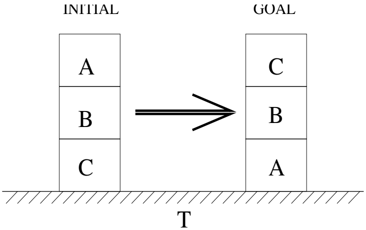
- Many successful applications of SAT:
    - Boolean circuits
    - (Bounded) Planning
    - (Bounded) Model Checking
    - Cryptography
    - Scheduling
    - ...
- All NP-complete problem can be (polynomially) converted to SAT.
- Key issue: find an efficient encoding.

Outline

The problem [30, 29, 48]

- Problem Given a set of action operators OP, (a representation of) an initial state I and goal state G, and a bound n, find a sequence of operator applications $o_1, .., o_n$, leading from the initial state to the goal state.
- Idea: Encode it into satisfiability problem of a Boolean formula φ

Example



$$\begin{array}{lll} \mathit{Move}(b,s,d) & & \\ & \mathit{Precond}: & \mathit{Block}(b) \wedge \mathit{Clear}(b) \wedge \mathit{On}(b,s) \wedge \\ & & (\mathit{Clear}(d) \vee \mathit{Table}(d)) \wedge \\ & & b \neq s \wedge b \neq d \wedge s \neq d \\ & \mathit{Effect}: & \mathit{Clear}(s) \wedge \neg \mathit{On}(b,s) \wedge \\ & & \mathit{On}(b,d) \wedge \neg \mathit{Clear}(d) \end{array}$$

Encoding

- Initial states:

$$On_0(A,B), On_0(B,C), On_0(C,T), Clear_0(A).$$

- Goal states:

$$On_{2n}(C,B) \wedge On_{2n}(B,A) \wedge On_{2n}(A,T).$$

- Action preconditions and effects:

$$\begin{array}{l} Move_t(A,B,C) \rightarrow \\ \qquad Clear_{t-1}(A) \wedge On_{t-1}(A,B) \wedge Clear_{t-1}(C) \wedge \\ \qquad Clear_{t+1}(B) \wedge \neg On_{t+1}(A,B) \wedge \\ \qquad On_{t+1}(A,C) \wedge \neg Clear_{t+1}(C). \end{array}$$

Encoding: Frame axioms

- Classic

$$Move_t(A,B,T) \wedge Clear_{t-1}(C) \rightarrow Clear_{t+1}(C),$$
$$Move_t(A,B,T) \wedge \neg Clear_{t-1}(C) \rightarrow \neg Clear_{t+1}(C).$$

“At least one action” axiom:

$$\bigvee_{\substack{b,s,d \in \{A,B,C,T\} \\ b \neq s, b \neq d, s \neq d, b \neq T}} Move_t(b,s,d).$$

- Explanatory

$$\neg Clear_{t+1}(C) \wedge Clear_{t-1}(C) \rightarrow$$
$$Move_t(A,B,C) \vee Move_t(A,T,C) \vee Move_t(B,A,C) \vee Move_t(B,$$

Planning strategy

- Sequential for each pair of actions α and β, add axioms of the form $\neg\alpha_t \vee \neg\beta_t$ for each odd time step. For example, we will have:

$$\neg Move_t(A, B, C) \vee \neg Move_t(A, B, T).$$

- parallel for each pair of actions α and β, add axioms of the form $\neg\alpha_t \vee \neg\beta_t$ for each odd time step if α effects contradict β preconditions. For example, we will have

$$\neg Move_t(B, T, A) \vee \neg Move_t(A, B, C).$$

Encoding into SAT

- Assumption: the possible values of all the variables are bounded.
- Naive idea: Encode all possible ground predicates as Boolean variables.
  E.g.: $Move_1(B, T, A) \Longrightarrow$ `Move1_B_T_A`
- much more efficient encodings have been presented [29, 19]
- customizations of SAT solvers [23].

Outline

1. Basics on SAT
2. Basic SAT-Solving techniques
3. Modern CDCL SAT Solvers
   - Conflict-Driven Clause-Learning SAT solvers
   - Further Improvements
4. Tractable subclasses of SAT
5. Random k-SAT and Phase Transition
6. Advanced Functionalities: proofs, unsat cores, interpolants, optimization
7. **Some Applications**
   - Appl. #1: (Bounded) Planning
   - **Appl. #2: Bounded Model Checking**

The problem [8, 7]

Ingredients:

- A system written as a Kripke structure $M := \langle S, I, T, \mathcal{L} \rangle$
  - S: set of states
  - I: set of initial states
  - T: transition relation
  - $\mathcal{L}$: labeling function
- A property f written as a LTL formula:
  - a propositional literal p
  - $h \wedge g$, $h \vee g$, $\mathbf{X}g$, $\mathbf{G}g$, $\mathbf{F}g$,$h\mathbf{U}g$ and $h\mathbf{R}g$,
    X, **G**, **F**, **U**, **R** “next”, “globally”, “eventually”, “until” and “releases”
- an integer k (bound)

The problem (cont.)

Problem:
Is there an execution path of M of length k satisfying the temporal property f?:

$$M \models_k \mathbf{f}$$

The encoding

Equivalent to the satisfiability problem of a Boolean formula $[[M,f]]_k$ defined as follows:

$$
\begin{aligned}
[[M,f]]_k &:= [[M]]_k \wedge [[f]]_k \quad (1)\\
[[M]]_k &:= I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}), \quad (2)\\
[[f]]_k &:= (\neg \bigvee_{l=0}^{k} T(s_k, s_l) \wedge [[f]]_k^0) \vee \bigvee_{l=0}^{k} (T(s_k, s_l) \wedge {}_l[[f]]_k^0), \quad (3)
\end{aligned}
$$

The encoding of $[[f]]_k^i$ and ${}_l[[f]]_k^i$

f	$[[f]]_k^i$	${}_l[[f]]_k^i$
p	p_i	p_i
$\neg p$	$\neg p_i$	$\neg p_i$
$h \wedge g$	$[[h]]_k^i \wedge [[g]]_k^i$	${}_l[[h]]_k^i \wedge {}_l[[g]]_k^i$
$h \vee g$	$[[h]]_k^i \vee [[g]]_k^i$	${}_l[[h]]_k^i \vee {}_l[[g]]_k^i$
$\mathbf{X}g$	$[[g]]_k^{i+1}$ $\quad$ $if\ i < k$ $\bot$ $\quad$ $otherwise.$	${}_l[[g]]_k^{i+1}$ $\quad$ $if\ i < k$ ${}_l[[g]]_k^{l}$ $\quad$ $otherwise.$
$\mathbf{G}g$	$\bot$	$\bigwedge_{j=min(i,l)}^{k} {}_l[[g]]_k^j$
$\mathbf{F}g$	$\bigvee_{j=i}^{k} [[g]]_k^j$	$\bigvee_{j=min(i,l)}^{k} {}_l[[g]]_k^j$
$h\mathbf{U}g$	$\bigvee_{j=i}^{k} \left([[g]]_k^j \wedge \bigwedge_{n=i}^{j-1} [[h]]_k^n \right)$	$\bigvee_{j=i}^{k} \left({}_l[[g]]_k^j \wedge \bigwedge_{n=i}^{j-1} {}_l[[h]]_k^n \right) \vee$ $\bigvee_{j=l}^{i-1} \left({}_l[[g]]_k^j \wedge \bigwedge_{n=i}^{k} {}_l[[h]]_k^n \wedge \bigwedge_{n=l}^{j-1} {}_l[[h]]_k^n \right)$
$h\mathbf{R}g$	$\bigvee_{j=i}^{k} \left([[h]]_k^j \wedge \bigwedge_{n=i}^{j} [[g]]_k^n \right)$	$\bigwedge_{j=min(i,l)}^{k} {}_l[[g]]_k^j \vee$ $\bigvee_{j=i}^{k} \left({}_l[[h]]_k^j \wedge \bigwedge_{n=i}^{j} {}_l[[g]]_k^n \right) \vee$ $\bigvee_{j=l}^{i-1} \left({}_l[[h]]_k^j \wedge \bigwedge_{n=i}^{k} {}_l[[g]]_k^n \wedge \bigwedge_{n=l}^{j} {}_l[[g]]_k^n \right)$

Example: **F**p (reachability)

- $f := \mathbf{F}p$: is there a reachable state in which p holds?
- $[[M, f]]_k$ is:

$$I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge \bigvee_{j=0}^{k} p_j$$

Example: **G**p

- $f := \mathbf{G}p$: is there a path where p holds forever?
- $[[M, f]]_k$ is:

$$I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge \bigvee_{l=0}^{k} T(s_k, s_l) \wedge \bigwedge_{j=0}^{k} p_j$$

Example: **GF**$q \wedge$ **F**p (fair reachability)

- $f :=$ **GF**$q \wedge$ **F**p: is there a reachable state in which p holds provided that q holds infinitely often?
- $[[M, f]]_k$ is:

$$I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge \bigvee_{j=0}^{k} p_j \wedge \bigvee_{l=0}^{k} \left(T(s_k, s_l) \wedge \bigvee_{j=l}^{k} q \right)$$

Bounded Model Checking

- very efficient for some problems
- lots of enhancements [8, 1, 56, 60, 13]

References I

[1] P. A. Abdullah, P. Bjesse, and N. Een.
Symbolic Reachability Analysis based on SAT-Solvers.
In *Sixth Int.nl Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'00)*, 2000.

[2] 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.

[3] F. Bacchus and J. Winter.
Effective Preprocessing with Hyper-Resolution and Equality Reduction.
In *Proc. Sixth International Symposium on Theory and Applications of Satisfiability Testing*, 2003.

[4] 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.

[5] A. Belov and Z. Stachniak.
Improving variable selection process in stochastic local search for propositional satisfiability.
In *SAT'09*, LNCS. Springer, 2009.

[6] A. Belov and Z. Stachniak.
Improved local search for circuit satisfiability.
In *SAT*, volume 6175 of *LNCS*, pages 293–299. Springer, 2010.

[7] A. Biere.
Bounded Model Checking, chapter 14, pages 455–481.
In Biere et al. [9], February 2009.

[8] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu.
Symbolic Model Checking without BDDs.
In *Proc. TACAS'99*, pages 193–207, 1999.

References II

[9] A. Biere, M. J. H. Heule, H. van Maaren, and T. Walsh, editors.
Handbook of Satisfiability.
IOS Press, February 2009.

[10] Booleforce, `http://fmv.jku.at/booleforce/`.

[11] R. Brafman.
A simplifier for propositional formulas with many binary clauses.
In *Proc. IJCAI01*, 2001.

[12] R. E. Bryant.
Graph-Based Algorithms for Boolean Function Manipulation.
IEEE Transactions on Computers, C-35(8):677–691, Aug. 1986.

[13] A. Cimatti, M. Pistore, M. Roveri, and R. Sebastiani.
Improving the Encoding of LTL Model Checking into SAT.
In *Proc. VMCAI'02*, volume 2294 of *LNCS*. Springer, January 2002.

[14] M. Davis, G. Longemann, and D. Loveland.
A machine program for theorem proving.
Journal of the ACM, 5(7), 1962.

[15] M. Davis and H. Putnam.
A computing procedure for quantification theory.
Journal of the ACM, 7:201–215, 1960.

[16] E. Friedgut.
Sharp thresholds of graph properties, and the k-sat problem.
Journal of the American Mathematical Society, 12(4), 1998.

References III

[17] N. Eén and N. Sörensson.
Temporal induction by incremental sat solving.
Electr. Notes Theor. Comput. Sci., 89(4):543–560, 2003.

[18] 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.

[19] M. Ernst, T. Millstein, and D. Weld.
Automatic SAT-compilation of planning problems.
In *Proc. IJCAI-97*, 1997.

[20] M. R. Garey and D. S. Johnson.
Computers and Intractability.
Freeman and Company, New York, 1979.

[21] I. P. Gent, E. MacIntyre, P. Prosser, and T. Walsh.
The constrainedness of search.
In *Proceedings of AAAI-96*, pages 246–252, Menlo Park, 1996. AAAI Press / MIT Press.

[22] R. Gershman, M. Koifman, and O. Strichman.
Deriving Small Unsatisfiable Cores with Dominators.
In *Proc. CAV'06*, volume 4144 of *LNCS*. Springer, 2006.

[23] E. Giunchiglia, A. Massarotto, and R. Sebastiani.
Act, and the Rest Will Follow: Exploiting Determinism in Planning as Satisfiability.
In *Proc. AAAI'98*, pages 948–953, 1998.

[24] 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.

References IV

[25] 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.

[26] C. Gomes, B. Selman, and H. Kautz.
Boosting Combinatorial Search Through Randomization.
In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, 1998.

[27] H. H. Hoos and T. Stutzle.
Stochastic Local Search Foundation And Application.
Morgan Kaufmann, 2005.

[28] J. Huang.
MUP: a minimal unsatisfiability prover.
In *Proc. ASP-DAC '05*. ACM Press, 2005.

[29] H. Kautz, D. McAllester, and B. Selman.
Encoding Plans in Propositional Logic.
In *Proceedings International Conference on Knowledge Representation and Reasoning*. AAAI Press, 1996.

[30] H. Kautz and B. Selman.
Planning as Satisfiability.
In *Proc. ECAI-92*, pages 359–363, 1992.

[31] H. A. Kautz, A. Sabharwal, and B. Selman.
Incomplete Algorithms, chapter 6, pages 185–203.
In Biere et al. [9], February 2009.

[32] S. Kirkpatrick and B. Selman.
Critical behaviour in the satisfiability of random boolean expressions.
Science, 264:1297–1301, 1994.

References V

[33] 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.

[34] C. M. Li and F. Manyà.
MaxSAT, Hard and Soft Constraints, chapter 19, pages 613–631.
In Biere et al. [9], February 2009.

[35] I. Lynce and J. Marques-Silva.
On Computing Minimum Unsatisfiable Cores.
In *7th International Conference on Theory and Applications of Satisfiability Testing*, 2004.

[36] I. Lynce and J. P. Marques-Silva.
On computing minimum unsatisfiable cores.
In *SAT*, 2004.

[37] J. P. Marques-Silva, I. Lynce, and S. Malik.
Conflict-Driven Clause Learning SAT Solvers, chapter 4, pages 131–153.
In Biere et al. [9], February 2009.

[38] K. McMillan.
Interpolation and SAT-based model checking.
In *Proc. CAV*, 2003.

[39] K. McMillan and N. Amla.
Automatic abstraction without counterexamples.
In *Proc. of TACAS*, 2003.

[40] K. L. McMillan.
An interpolating theorem prover.
Theor. Comput. Sci., 345(1):101–121, 2005.

References VI

[41] D. Mitchell, B. Selman, and H. Levesque.
Hard and Easy Distributions of SAT Problems.
In *Proc. of the 10th National Conference on Artificial Intelligence*, pages 459–465, 1992.

[42] M.Mezard, G.Parisi, and R. Zecchina.
Analytic and Algorithmic Solution of Random Satisfiability Problems.
Science, 297(812), 2002.

[43] M. W. Moskewicz, C. F. Madigan, Y. Z., L. Zhang, and S. Malik.
Chaff: Engineering an efficient SAT solver.
In *Design Automation Conference*, 2001.

[44] 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.

[45] 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.

[46] 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.

[47] P. Pudlák.
Lower bounds for resolution and cutting planes proofs and monotone computations.
J. of Symb. Logic, 62(3), 1997.

References VII

[48] J. Rintanen.
Planning and SAT, chapter 15, pages 483–504.
In Biere et al. [9], February 2009.

[49] A. Robinson.
A machine-oriented logic based on the resolution principle.
Journal of the ACM, 12:23–41, 1965.

[50] R. Sebastiani.
Applying GSAT to Non-Clausal Formulas.
Journal of Artificial Intelligence Research, 1:309–314, 1994.

[51] 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.

[52] 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.

[53] 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.

[54] J. P. M. Silva and K. A. Sakallah.
GRASP - A new Search Algorithm for Satisfiability.
In *Proc. ICCAD'96*, 1996.

[55] R. M. Smullyan.
First-Order Logic.
Springer-Verlag, NY, 1968.

References VIII

[56] O. Strichmann.
Tuning SAT checkers for Bounded Model Checking.
In *Proc. CAV00*, volume 1855 of *LNCS*, pages 480–494. Springer, 2000.

[57] C. Tinelli.
A DPLL-based Calculus for Ground Satisfiability Modulo Theories.
In *Proc. JELIA-02*, volume 2424 of *LNAI*, pages 308–319. Springer, 2002.

[58] 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.

[59] C. P. Williams and T. Hogg.
Exploiting the deep structure of constraint problems.
Artificial Intelligence, 70:73–117, 1994.

[60] P. F. Williams, A. Biere, E. M. Clarke, and A. Gupta.
Combining Decision Diagrams and SAT Procedures for Efficient Symbolic Model Checking.
In *Proc. CAV2000*, volume 1855 of *LNCS*, pages 124–138, Berlin, 2000. Springer.

[61] H. Zhang and M. Stickel.
Implementing the Davis-Putnam algorithm by tries.
Technical report, University of Iowa, August 1994.

[62] 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 IX

[63] 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.

[64] 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.

[65] 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:
`http://disi.unitn.it/rseba/publist.html`.

Related web sites:

- Combination Methods in Automated Reasoning
  `http://combination.cs.uiowa.edu/`
- The SAT Association
  `http://satassociation.org/`
- SATLive! - Up-to-date links for SAT
  `http://www.satlive.org/index.jsp`
- SATLIB - The Satisfiability Library
  `http://www.intellektik.informatik.tu-darmstadt.de/SATLIB/`