SAT: Propositional Satisfiability and Beyond

Roberto Sebastiani
Dept. of Information and Communication Technologies
University of Trento, Italy
rseba@dit.unitn.it  http://www.dit.unitn.it/~rseba

©Roberto Sebastiani
Motivations

▷ Last ten years: impressive advance in SAT techniques
  ● extremely efficient SAT solvers [56, 51, 8, 43]
  ● hard “real-world” problems encoded into SAT
    (e.g., planning [38, 37, 20, 26], model checking [9, 1, 50, 54, 14])

▷ Recent years: using SAT solvers as propositional reasoning kernels for more expressive solvers
  ● combine a SAT reasoner with a domain-specific solver
  ● Modal logics [29, 35, 31, 25], description logics [30, 35], temporal reasoning [2], resource planning [55], verification of timed systems [42, 4, 6], SW verification [13], ...
Part 1: PROPOSITIONAL SATISFIABILITY

- Basics on SAT ......................................................... 6
- NNF, CNF and conversions ................................. 12
- k-SAT and Phase Transition .............................. 22
- Basic SAT techniques ........................................ 31
- SAT for non-CNF formulas .............................. 52
- DPLL Heuristics & Optimizations .................... 61
- SOME APPLICATIONS ........................................ 83
- Appl. #1: (Bounded) Planning .......................... 85
- Appl. #2: Bounded Model Checking .................. 91
Content (cont.)

Part 2: BEYOND SAT

- Formal Framework .............................................. 104
- A Generalized Search Procedure ............................ 121
- Extending existing SAT procedures .......................... 131
- Optimizations ..................................................... 153
- Case study: Modal Logic(s) .................................... 183
- Case Study: Mathematical Reasoning ...................... 209
PART 1:
PROPOSITIONAL SATISFIABILITY
Basics on SAT
Basic notation & definitions

– **Boolean formula**
  
  * ● $\top, \bot$ are formulas
  * ● A **propositional atom** $A_1, A_2, A_3, ...$ is a formula;
  * ● if $\varphi_1$ and $\varphi_2$ are formulas, then $\neg\varphi_1, \varphi_1 \land \varphi_2, \varphi_1 \lor \varphi_2,$
  $\varphi_1 \rightarrow \varphi_2, \varphi_1 \iff \varphi_2$ are formulas.

– **Literal**: a propositional atom $A_i$ (positive literal) or its
  
  * negation $\neg A_i$ (negative literal)

– **Atoms($\varphi$)**: the set $\{A_1, ..., A_N\}$ of propositional atoms
  
  * occurring in $\varphi$.

– a boolean formula can be represented as a tree or as a **DAG**
Basic notation & definitions (cont)

- Total truth assignment $\mu$ for $\varphi$:
  $\mu : Atoms(\varphi) \rightarrow \{\top, \bot\}$.
- Partial Truth assignment $\mu$ for $\varphi$:
  $\mu : A \rightarrow \{\top, \bot\}$, $A \subseteq Atoms(\varphi)$.
- Set and formula representation of an assignment:
  - $\mu$ can be represented as a set of literals:
    EX: $\{\mu(A_1) := \top, \mu(A_2) := \bot\} \implies \{A_1, \lnot A_2\}$
  - $\mu$ can be represented as a formula:
    EX: $\{\mu(A_1) := \top, \mu(A_2) := \bot\} \implies A_1 \land \lnot A_2$
Basic notation & definitions (cont)

- \( \mu \models \varphi \) (\( \mu \) satisfies \( \varphi \)):
  - \( \mu \models A_i \iff \mu(A_i) = \top \)
  - \( \mu \models \neg \varphi \iff \text{not } \mu \models \varphi \)
  - \( \mu \models \varphi_1 \land \varphi_2 \iff \mu \models \varphi_1 \text{ and } \mu \models \varphi_2 \)
  - ... 

- \( \varphi \) is satisfiable iff \( \mu \models \varphi \) for some \( \mu \)

- \( \varphi_1 \models \varphi_2 \) (\( \varphi_1 \) entails \( \varphi_2 \)):
  - \( \varphi_1 \models \varphi_2 \) iff for every \( \mu \) \( \mu \models \varphi_1 \implies \mu \models \varphi_2 \)

- \( \models \varphi \) (\( \varphi \) is valid):
  - \( \models \varphi \) iff for every \( \mu \) \( \mu \models \varphi \)

- \( \varphi \) is valid \( \iff \neg \varphi \) is not satisfiable
Equivalence and equi-satisfiability

- \( \varphi_1 \) and \( \varphi_2 \) are **equivalent** iff, for every \( \mu \),
  \[
  \mu \models \varphi_1 \text{ iff } \mu \models \varphi_2
  \]

- \( \varphi_1 \) and \( \varphi_2 \) are **equi-satisfiable** iff
  \[
  \exists \mu_1 \text{ s.t. } \mu_1 \models \varphi_1 \text{ iff } \exists \mu_2 \text{ s.t. } \mu_2 \models \varphi_2
  \]

- \( \varphi_1, \varphi_2 \) equivalent
  \[
  \downarrow \quad \not\downarrow
  \]
  \( \varphi_1, \varphi_2 \) equi-satisfiable

- **EX:** \( \varphi_1 \lor \varphi_2 \) and \( (\varphi_1 \lor \neg A_3) \land (A_3 \lor \varphi_2) \), \( A_3 \) not in \( \varphi_1 \lor \varphi_2 \), are **equi-satisfiable** but **not equivalent**.
The problem of deciding the **satisfiability** of a propositional formula is **NP-complete** [15].

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.**
NNF, CNF and conversions
POLARITY of subformulas

Polarity: the number of nested negations modulo 2.

- Positive/negative occurrences
  - $\varphi$ occurs positively in $\varphi$;
  - if $\neg \varphi_1$ occurs positively [negatively] in $\varphi$, then $\varphi_1$ occurs negatively [positively] in $\varphi$;
  - if $\varphi_1 \land \varphi_2$ or $\varphi_1 \lor \varphi_2$ occur positively [negatively] in $\varphi$, then $\varphi_1$ and $\varphi_2$ occur positively [negatively] in $\varphi$;
  - if $\varphi_1 \rightarrow \varphi_2$ occurs positively [negatively] in $\varphi$, then $\varphi_1$ occurs negatively [positively] in $\varphi$ and $\varphi_2$ occurs positively [negatively] in $\varphi$;
  - if $\varphi_1 \leftrightarrow \varphi_2$ occurs in $\varphi$, then $\varphi_1$ and $\varphi_2$ occur positively and negatively in $\varphi$;
Negative normal form (NNF)

- \( \varphi \) is in Negative normal form iff it is given only by applications of \( \land, \lor \) to literals.

- Every \( \varphi \) can be reduced into NNF:
  1. Substituting all \( \rightarrow \)'s and \( \leftrightarrow \)'s:
     \[
     \varphi_1 \rightarrow \varphi_2 \implies \neg \varphi_1 \lor \varphi_2
     \]
     \[
     \varphi_1 \leftrightarrow \varphi_2 \implies (\neg \varphi_1 \lor \varphi_2) \land (\varphi_1 \lor \neg \varphi_2)
     \]
  2. Pushing down negations recursively:
     \[
     \neg (\varphi_1 \land \varphi_2) \implies \neg \varphi_1 \lor \neg \varphi_2
     \]
     \[
     \neg (\varphi_1 \lor \varphi_2) \implies \neg \varphi_1 \land \neg \varphi_2
     \]
     \[
     \neg \neg \varphi_1 \implies \varphi_1
     \]
- The reduction is linear if a DAG representation is used.
- Preserves the equivalence of formulas.
NNF: example

\[(A_1 \leftrightarrow A_2) \leftrightarrow (A_3 \leftrightarrow A_4)\]

**Tree Representation**

**DAG Representation**
Conjunctive Normal Form (CNF)

- $\varphi$ 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.
  $\implies$ no reasoning on the recursive structure of the formula
Classic CNF Conversion $CNF(\varphi)$

- Every $\varphi$ can be reduced into CNF by, e.g.,
  1. converting it into NNF;
  2. applying recursively the DeMorgan’s Rule:
     $$ (\varphi_1 \land \varphi_2) \lor \varphi_3 \implies (\varphi_1 \lor \varphi_3) \land (\varphi_2 \lor \varphi_3) $$
- Worst-case exponential.
- $Atoms(CNF(\varphi)) = Atoms(\varphi)$.
- $CNF(\varphi)$ is equivalent to $\varphi$.
- Normal: if $\varphi_1$ equivalent to $\varphi_2$, then $CNF(\varphi_1)$ identical to $CNF(\varphi_2)$ modulo reordering.
- Rarely used in practice.
Labeling CNF conversion $CNF_{label}(\varphi) \ [44, 19]$

- Every $\varphi$ can be reduced into CNF by, e.g., applying recursively bottom-up the rules:
  
  \[
  \begin{align*}
  \varphi & \implies \varphi[(l_i \lor l_j) \mid B] \land CNF(B \leftrightarrow (l_i \lor l_j)) \\
  \varphi & \implies \varphi[(l_i \land l_j) \mid B] \land CNF(B \leftrightarrow (l_i \land l_j)) \\
  \varphi & \implies \varphi[(l_i \leftrightarrow l_j) \mid B] \land CNF(B \leftrightarrow (l_i \leftrightarrow l_j))
  \end{align*}
  \]
  
  $l_i, l_j$ being literals and $B$ being a “new” variable.

- Worst-case linear.

- $Atoms(CNF_{label}(\varphi)) \supset Aoms(\varphi)$.

- $CNF_{label}(\varphi)$ is equi-satisfiable w.r.t. $\varphi$.

- Non-normal.

- More used in practice.
Labeling CNF conversion $CNF_{\text{label}}$ — example

$CNF(B_1 \leftrightarrow (\neg A_3 \lor A_1)) \land$

... \land

$CNF(B_8 \leftrightarrow (A_1 \lor \neg A_4)) \land$

$CNF(B_9 \leftrightarrow (B_1 \leftrightarrow B_2)) \land$

... \land

$CNF(B_{12} \leftrightarrow (B_7 \land B_8)) \land$

$CNF(B_{13} \leftrightarrow (B_9 \lor B_{10})) \land$

$CNF(B_{14} \leftrightarrow (B_{11} \lor B_{12})) \land$

$CNF(B_{15} \leftrightarrow (B_{13} \land B_{14})) \land$

$B_{15}$
Labeling CNF conversion $CNF_{label}$ (improved)

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

\[ \varphi \implies \varphi[(l_i \lor l_j) | B] \land CNF(B \rightarrow (l_i \lor l_j)) \text{ if } (l_i \lor l_j) \text{ positive} \]

\[ \varphi \implies \varphi[(l_i \lor l_j) | B] \land CNF((l_i \lor l_j) \rightarrow B) \text{ if } (l_i \lor l_j) \text{ negative} \]

\[ \varphi \implies \varphi[(l_i \land l_j) | B] \land CNF(B \rightarrow (l_i \land l_j)) \text{ if } (l_i \land l_j) \text{ positive} \]

\[ \varphi \implies \varphi[(l_i \land l_j) | B] \land CNF((l_i \land l_j) \rightarrow B) \text{ if } (l_i \land l_j) \text{ negative} \]

\[ \varphi \implies \varphi[(l_i \leftrightarrow l_j) | B] \land CNF(B \rightarrow (l_i \leftrightarrow l_j)) \text{ if } (l_i \leftrightarrow l_j) \text{ positive} \]

\[ \varphi \implies \varphi[(l_i \leftrightarrow l_j) | B] \land CNF((l_i \leftrightarrow l_j) \rightarrow B) \text{ if } (l_i \leftrightarrow l_j) \text{ negative} \]

- Smaller in size:

\[ CNF(B \rightarrow (l_i \lor l_j)) = (\neg B \lor l_i \lor l_j) \]

\[ CNF(((l_i \lor l_j) \rightarrow B)) = (\neg l_i \lor B) \land (\neg l_j \lor B) \]
Labeling CNF conversion $CNF_{label}$ – example

**Basic**

\[
CNF(B_1 \leftrightarrow (\neg A_3 \lor A_1)) \land \\
\ldots \land \\
CNF(B_8 \leftrightarrow (A_1 \lor \neg A_4)) \land \\
CNF(B_9 \leftrightarrow (B_1 \leftrightarrow B_2)) \land \\
\ldots \land \\
CNF(B_{12} \leftrightarrow (B_7 \land B_8)) \land \\
CNF(B_{13} \leftrightarrow (B_9 \lor B_{10})) \land \\
CNF(B_{14} \leftrightarrow (B_{11} \lor B_{12})) \land \\
CNF(B_{15} \leftrightarrow (B_{13} \land B_{14})) \land \\
B_{15}
\]

**Improved**

\[
CNF(B_1 \leftrightarrow (\neg A_3 \lor A_1)) \land \\
\ldots \land \\
CNF(B_8 \rightarrow (A_1 \lor \neg A_4)) \land \\
CNF(B_9 \rightarrow (B_1 \leftrightarrow B_2)) \land \\
\ldots \land \\
CNF(B_{12} \rightarrow (B_7 \land B_8)) \land \\
CNF(B_{13} \rightarrow (B_9 \lor B_{10})) \land \\
CNF(B_{14} \rightarrow (B_{11} \lor B_{12})) \land \\
CNF(B_{15} \rightarrow (B_{13} \land B_{14})) \land \\
B_{15}
\]
k-SAT and Phase Transition
The satisfiability of k-CNF (k-SAT) [22]

- **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:

\[
\begin{align*}
    &l_1 \lor l_2 \lor \ldots \lor l_{k-1} \lor l_k \\
    &\downarrow \\
    &\quad (l_1 \lor l_2 \lor B_1) \land \\
    &\quad (\neg B_1 \lor l_3 \lor B_2) \land \\
    &\quad \ldots \\
    &\quad (\neg B_{k-4} \lor l_{k-2} \lor B_{k-3}) \land \\
    &\quad (\neg B_{k-3} \lor l_{k-1} \lor l_k)
\end{align*}
\]
Random K-CNF formulas generation

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

**DO**

1. pick with uniform probability a set of $k$ atoms over $N$
2. randomly negate each atom with probability $0.5$
3. 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, 39]

- 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 [53, 23]
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 $N \rightarrow \infty$, 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.
Basic SAT techniques
Truth Tables

- **Exhaustive evaluation** of all subformulas:

<table>
<thead>
<tr>
<th>$\varphi_1$</th>
<th>$\varphi_2$</th>
<th>$\varphi_1 \land \varphi_2$</th>
<th>$\varphi_1 \lor \varphi_2$</th>
<th>$\varphi_1 \rightarrow \varphi_2$</th>
<th>$\varphi_1 \leftrightarrow \varphi_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\bot$</td>
<td>$\bot$</td>
<td>$\bot$</td>
<td>$\bot$</td>
<td>$T$</td>
<td>$T$</td>
</tr>
<tr>
<td>$\bot$</td>
<td>$T$</td>
<td>$\bot$</td>
<td>$T$</td>
<td>$T$</td>
<td>$T$</td>
</tr>
<tr>
<td>$T$</td>
<td>$\bot$</td>
<td>$\bot$</td>
<td>$T$</td>
<td>$\bot$</td>
<td>$\bot$</td>
</tr>
<tr>
<td>$T$</td>
<td>$T$</td>
<td>$T$</td>
<td>$T$</td>
<td>$T$</td>
<td>$T$</td>
</tr>
</tbody>
</table>

- Requires **polynomial space**.
- Never used in practice.
Semantic tableaux [52]

- Search for an assignment satisfying $\varphi$
- applies recursively elimination rules to the connectives
- If a branch contains $A_i$ and $\neg A_i$, $(\psi_i$ and $\neg \psi_1)$ for some $i$, the branch is closed, otherwise it is open.
- if no rule can be applied to an open branch $\mu$, then $\mu \models \varphi$;
- if all branches are closed, the formula is not satisfiable;
Tableau elimination rules

\[ \varphi_1 \land \varphi_2 \quad \neg (\varphi_1 \lor \varphi_2) \quad \neg (\varphi_1 \rightarrow \varphi_2) \]

\[ \varphi_1 \quad \neg \varphi \quad \neg \neg \varphi \]

\[ \varphi_1 \lor \varphi_2 \quad \neg (\varphi_1 \land \varphi_2) \quad \varphi_1 \rightarrow \varphi_2 \]

\[ \varphi_1 \leftrightarrow \varphi_2 \quad \neg (\varphi_1 \leftrightarrow \varphi_2) \]

(\&\text{-elimination})

(\neg\neg\text{-elimination})

(\lor\text{-elimination})

(\leftrightarrow\text{-elimination}).
Semantic Tableaux – example

\[ \varphi = (A_1 \lor A_2) \land (A_1 \lor \neg A_2) \land (\neg A_1 \lor A_2) \land (\neg A_1 \lor \neg A_2) \]
function Tableau(\(\Gamma\))
    if \(A_i \in \Gamma\) and \(\neg A_i \in \Gamma\) /* branch closed */
        then return \(False;\)
    if \((\varphi_1 \land \varphi_2) \in \Gamma\) /* \(\land\)-elimination */
        then return Tableau(\(\Gamma \cup \{\varphi_1, \varphi_2\}\)\(\setminus\)\(\{\varphi_1 \land \varphi_2\}\));
    if \((\neg \neg \varphi_1) \in \Gamma\) /* \(\neg \neg\)-elimination */
        then return Tableau(\(\Gamma \cup \{\varphi_1\}\)\(\setminus\)\(\{\neg \neg \varphi_1\}\));
    if \((\varphi_1 \lor \varphi_2) \in \Gamma\) /* \(\lor\)-elimination */
        then return Tableau(\(\Gamma \cup \{\varphi_1\}\)\(\setminus\)\(\{\varphi_1 \lor \varphi_2\}\)) or Tableau(\(\Gamma \cup \{\varphi_2\}\)\(\setminus\)\(\{\varphi_1 \lor \varphi_2\}\));
    ...
    return \(True;\) /* branch expanded */
Semantic Tableaux – summary

- Handles all propositional formulas (CNF not required).
- Branches on disjunctions
- Intuitive, modular, easy to extend
  \[\implies\text{loved by logicians.}\]
- Rather inefficient
  \[\implies\text{avoided by computer scientists.}\]
- Requires polynomial space
DPLL [18, 17]

- Davis-Putnam-Longeman-Loveland procedure (DPLL)
- Tries to build recursively an assignment $\mu$ satisfying $\varphi$;
- At each recursive step assigns a truth value to (all instances of) one atom.
- Performs deterministic choices first.
DPLL rules

$$\varphi_1 \land (l) \quad \frac{\varphi_1 \land (l)}{\varphi_1[l\mid T]} \quad (\text{Unit})$$

$$\varphi \quad \frac{\varphi}{\varphi[l\mid T]} \quad (l \text{ Pure})$$

$$\varphi \quad \frac{\varphi}{\varphi[l\mid T] \quad \varphi[l\mid \bot]} \quad (\text{split})$$

(l is a pure literal in $\varphi$ iff it occurs only positively).

Split applied if and only if the others cannot be applied.
\[ \varphi = (A_1 \lor A_2) \land (A_1 \lor \neg A_2) \land (\neg A_1 \lor A_2) \land (\neg A_1 \lor \neg A_2) \]
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(\(\text{assign}(l, \varphi), \mu \land l\));
   if \{a literal \(l\) occurs pure in \(\varphi\}\} /* pure */
      then return DPLL(\(\text{assign}(l, \varphi), \mu \land l\));
   \(l := \text{choose-literal}(\varphi)\); /* split */
   return DPLL(\(\text{assign}(l, \varphi), \mu \land l\)) or
            DPLL(\(\text{assign}(\neg l, \varphi), \mu \land \neg l\));
DPLL – summary

- Handles **CNF formulas** (non-CNF variant known [3, 28]).
- **Branches on truth values**
  \[\Rightarrow\] all instances of an atom assigned simultaneously
- **Postpones branching as much as possible.**
- Mostly ignored by logicians.
- Probably **the most efficient SAT algorithm**
  \[\Rightarrow\] loved by computer scientists.
- Requires **polynomial space**
- **Choose\_literal()** critical!
- Many very efficient implementations [56, 51, 8, 43].
- A library: SIM [27]
Ordered Binary Decision Diagrams (OBDDs) [12]

- **Normal representation** of a boolean formula.
- “If-then-else” binary DAGs with two leaves: 1 and 0
- **Variable ordering** $A_1, A_2, \ldots, A_n$ imposed a priory.
- Paths leading to 1 represent **models**
  Paths leading to 0 represent **counter-models**
- Once built, logical operations (satisfiability, validity, equivalence) immediate.
- Finds **all** models.
(Implicit) OBDD structure

- \( OBDD(\top, \{\ldots\}) = 1, \)
- \( OBDD(\bot, \{\ldots\}) = 0, \)
- \( OBDD(\varphi, \{A_1, A_2, \ldots, A_n\}) = \)
  
  \( \text{if } A_1 \)
  \( \text{then } OBDD(\varphi[A_1 | \top], \{A_2, \ldots, A_n\}) \)
  \( \text{else } OBDD(\varphi[A_1 | \bot], \{A_2, \ldots, A_n\}) \)
OBDD - Examples

Figure 1: OBDDs of \((a_1 \leftrightarrow b_1) \land (a_2 \leftrightarrow b_2) \land (a_3 \leftrightarrow b_3)\) with different variable orderings
Incrementally building an OBDD

- \texttt{obdd\_build}(\top, \{\ldots\}) := 1,
- \texttt{obdd\_build}(\bot, \{\ldots\}) := 0,
- \texttt{obdd\_build}((\varphi_1 \texttt{ op } \varphi_2), \{A_1, \ldots, A_n\}) :=
  \texttt{obdd\_merge}( \texttt{ op },
    \texttt{obdd\_build}(\varphi_1, \{A_1, \ldots, A_n\}),
    \texttt{obdd\_build}(\varphi_2, \{A_1, \ldots, A_n\}),
    \{A_1, \ldots, A_n\}
  )
  \quad \text{\texttt{ op } }\in\{\land, \lor, \to, \leftrightarrow\}
OBBD incremental building – example

\[ \varphi = (A_1 \vee A_2) \land (A_1 \vee \neg A_2) \land (\neg A_1 \vee A_2) \land (\neg A_1 \vee \neg A_2) \]
OBDD – summary

- Handle all propositional formulas (CNF not required).
- (Implicitly) branch on truth values.
- Find all models.
- Factorize common parts of the search tree (DAG)
- Require setting a variable ordering a priori (critical!)
  \[\implies\text{cannot postpone branching}\]
- Very efficient for some problems (circuits, model checking).
- Require exponential space in worst-case
- Used by Hardware community, ignored by logicians, recently introduced in computer science.
Incomplete SAT techniques: GSAT [49]

- Hill-Climbing techniques: GSAT
- looks for a complete assignment;
- starts from a random assignment;
- Greedy search: looks for a better “neighbor” assignment
- Avoid local minima: restart & random walk
function $\text{GSAT}(\varphi)$

\[
\begin{align*}
\text{for } i & := 1 \text{ to Max-tries do} \\
\mu & := \text{rand-assign}(\varphi); \\
\text{for } j & := 1 \text{ to Max-flips do} \\
\text{if } (\text{score}(\varphi, \mu) = 0) \\
& \quad \text{then return True;} \\
\text{else } \text{Best-flips} & := \text{hill-climb}(\varphi, \mu); \\
A_i & := \text{rand-pick}(\text{Best-flips}); \\
\mu & := \text{flip}(A_i, \mu);
\end{align*}
\]

end

end

return “no satisfying assignment found”.
GSAT – summary

- Handle only CNF formulas.
- Incomplete
- Extremely efficient for some (satisfiable) problems.
- Require polynomial space
- Used in Artificial Intelligence (e.g., planning)
- Variants: GSAT+random walk, WSAT
- Non-CNF Variants: NC-GSAT [46], DAG-SAT [48]
SAT for non-CNF formulas
Non-CNF DPLL [3]

function $NC\text{-}DPLL(\varphi, \mu)$

if $\varphi = \top$

    then return $True$; /* base */

if $\varphi = \bot$

    then return $False$; /* backtrack */

if $\{ \exists l \text{ s.t. equivalent\_unit}(l, \varphi) \}$

    then return $NC\text{-}DPLL(assign(l, \varphi), \mu \land l)$; /* unit */

if $\{ \exists l \text{ s.t. equivalent\_pure}(l, \varphi) \}$

    then return $NC\text{-}DPLL(assign(l, \varphi), \mu \land l)$; /* pure */

$l := choose\text{-}literal(\varphi)$; /* split */

return $NC\text{-}DPLL(assign(l, \varphi), \mu \land l)$ or $NC\text{-}DPLL(assign(\neg l, \varphi), \mu \land \neg l)$;
Non-CNF DPLL (cont.)

\[ \text{equivalent\_unit}(l, \varphi) : \]
\[
\begin{align*}
\text{equivalent\_unit}(l, l_1) & := \top \text{ if } l = l_1 \\
& \quad \bot \text{ otherwise} \\
\text{equivalent\_unit}(l, \varphi_1 \land \varphi_2) & := \text{equivalent\_unit}(l, \varphi_1) \text{ or } \text{equivalent\_unit}(l, \varphi_2) \\
\text{equivalent\_unit}(l, \varphi_1 \lor \varphi_2) & := \text{equivalent\_unit}(l, \varphi_1) \text{ and } \text{equivalent\_unit}(l, \varphi_2)
\end{align*}
\]
Non-CNF DPLL (cont.)

\[ \text{equivalent\_pure}(l, \varphi) : \]

\[ \text{equivalent\_pure}(l, l_1) := \bot \quad \text{if } l = \neg l_1 \]
\[ \top \quad \text{otherwise} \]

\[ \text{equivalent\_pure}(l, \varphi_1 \land \varphi_2) := \text{equivalent\_pure}(l, \varphi_1) \land \text{equivalent\_pure}(l, \varphi_2) \]

\[ \text{equivalent\_pure}(l, \varphi_1 \lor \varphi_2) := \text{equivalent\_pure}(l, \varphi_1) \land \text{equivalent\_pure}(l, \varphi_2) \]
Applying DPLL to $CNF_{label}(\varphi)$ [28, 26]

- $CNF(\varphi) = O(2^{\mid \varphi \mid})$
  $\implies$ inapplicable in most cases.
- $CNF_{label}(\varphi)$ introduces $K = O(|\varphi|)$ new variables
  $\implies$ size of assignment space passes from $2^N$ to $2^{N+K}$
- Idea: values of new variables derive deterministically from those of original variables.
- Realization: restrict $Choose_{\text{literal}}(\varphi)$ to split first on original variables
  $\implies$ DPLL assigns the other variables deterministically.
Applying DPLL to $CNF_{\text{label}}(\varphi)$ (cont)

- If basic $CNF_{\text{label}}(\varphi)$ is used:

$$\varphi \iff \varphi[(l_i \lor l_j) \mid B] \land CNF(B \leftrightarrow (l_i \lor l_j))$$

... ... ...

then $B$ is deterministically assigned by unit propagation if $l_i$ and $l_j$ are assigned.
- If the improved $CNF_{\text{label}}(\varphi)$ is used:

\[ \varphi \iff \varphi[(l_i \lor l_j)|B] \land CNF(B \rightarrow (l_i \lor l_j)) \text{ if } (l_i \lor l_j) \text{ positive} \]

... ... ...

then $B$ is deterministically assigned:

- by unit propagation if $l_i$ and $l_j$ are assigned to $\bot$.
- by pure literal if one of $l_i$ and $l_j$ is assigned to $\top$. 
function \textit{NC-GSAT}(\varphi) \\
\hspace{0.5em} \text{for } i := 1 \text{ to Max-tries do} \\
\hspace{1em} \mu := \text{rand-assign}(\varphi) ; \\
\hspace{1em} \text{for } j := 1 \text{ to Max-flips do} \\
\hspace{2em} \text{if } (s(\mu, \varphi) = 0) \\
\hspace{3em} \text{then return True;} \\
\hspace{3em} \text{else Best-flips := hill-climb}(\varphi, \mu) ; \\
\hspace{4em} A_i := \text{rand-pick(Best-flips)} ; \\
\hspace{4em} \mu := \text{flip}(A_i, \mu) ; \\
\hspace{1em} \text{end} \\
\text{end} \\
\text{return } \text{“no satisfying assignment found”}. 
Non-CNF GSAT (cont.)

<table>
<thead>
<tr>
<th>( \varphi )</th>
<th>( s(\mu, \varphi) )</th>
<th>( s^-(\mu, \varphi) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \varphi ) literal</td>
<td>( \begin{cases} 0 &amp; \text{if } \mu \models \varphi \ 1 &amp; \text{otherwise} \end{cases} )</td>
<td>( \begin{cases} 1 &amp; \text{if } \mu \models \varphi \ 0 &amp; \text{otherwise} \end{cases} )</td>
</tr>
<tr>
<td>( \land_k \varphi_k )</td>
<td>( \sum_k s(\mu, \varphi_k) )</td>
<td>( \prod_k s^-(\mu, \varphi_k) )</td>
</tr>
<tr>
<td>( \lor_k \varphi_k )</td>
<td>( \prod_k s(\mu, \varphi_k) )</td>
<td>( \sum_k s^-(\mu, \varphi_k) )</td>
</tr>
<tr>
<td>( \varphi_1 \equiv \varphi_2 )</td>
<td>( \begin{cases} s^-(\mu, \varphi_1) \cdot s(\mu, \varphi_2) + \ s(\mu, \varphi_1) \cdot s^-(\mu, \varphi_2) \end{cases} )</td>
<td>( \begin{cases} (s(\mu, \varphi_1) + s^-(\mu, \varphi_2)) \cdot \ (s^-(\mu, \varphi_1) + s(\mu, \varphi_2)) \end{cases} )</td>
</tr>
</tbody>
</table>

\( s(\mu, \varphi) \) computes score(\( CNF(\mu, \varphi) \)) directly in linear time.
DPLL Heuristics & Optimizations
Techniques to achieve efficiency in DPLL

- **Preprocessing**: preprocess the input formula so that to make it easier to solve
- **Look-ahead**: exploit information about the remaining search space
  - unit propagation
  - pure literal
  - forward checking (splitting heuristics)
- **Look-back**: exploit information about search which has already taken place
  - Backjumping
  - Learning
DPLL is a family of algorithms.

- different splitting heuristics
- preprocessing: (subsumption, 2-simplification)
- backjumping
- learning
- random restart
- horn relaxation
- ...
Splitting heuristics - Choose\_literal()

- **Split** is the source of non-determinism for DPLL
- **Choose\_literal()** critical for efficiency
- many split heuristics conceived in literature.
Some example heuristics

- **MOM** heuristics: pick the literal occurring most often in the minimal size clauses
  \(\implies\) fast and simple

- **Jeroslow-Wang**: choose the literal with maximum

\[
score(l) := \sum_{l \in c \land c \in \varphi} 2^{-|c|}
\]

\(\implies\) estimates \(l\)'s contribution to the satisfiability of \(\varphi\)

- **Satz**: selects a candidate set of literals, perform unit propagation, chooses the one leading to smaller clause set
  \(\implies\) maximizes the effects of unit propagation
Some preprocessing techniques

- Sorting+subsumption:

\[
\varphi_1 \land (l_2 \lor l_1) \land \varphi_2 \land (l_2 \lor l_3 \lor l_1) \land \varphi_3 \\
\downarrow \\
\varphi_1 \land (l_1 \lor l_2) \land \varphi_2 \land \varphi_3
\]
Some preprocessing techniques (cont.)

- 2-simplifying \[10\]: exploiting binary clauses.
- Repeat:
  1. build the implication graph induced by literals
  2. detect strongly connected cycles
     \[\Leftrightarrow\text{equivalence classes of literals}\]
  3. perform substitutions
  4. perform unit and pure.
     \textbf{Until} no more simplification possible.
- Very useful for many application domains.
Conflict-directed backtracking (backjumping) [8, 51]

- **Idea:** when a branch fails,
  1. reveal the sub-assignment causing the failure (*conflict set*)
  2. backtrack to the most recent branching point in the conflict set
- a *conflict set* is constructed from the conflict clause by tracking backwards the unit-implications causing it and by keeping the branching literals.
- when a branching point fails, a *conflict set* is obtained by resolving the two conflict sets of the two branches.
- may avoid lots of redundant search.
Conflict-directed backtracking – example

\[ \neg A_1 \lor A_2 \]
\[ \neg A_1 \lor A_3 \lor A_9 \]
\[ \neg A_2 \lor \neg A_3 \lor A_4 \]
\[ \neg A_4 \lor A_5 \lor A_{10} \]
\[ \neg A_4 \lor A_6 \lor A_{11} \]
\[ \neg A_5 \lor \neg A_6 \]
\[ A_1 \lor A_7 \lor \neg A_{12} \]
\[ A_1 \lor A_8 \]
\[ \neg A_7 \lor \neg A_8 \lor \neg A_{13} \]

...
Conflict-directed backtracking – example (cont.)

\[ \neg A_1 \lor A_2 \]
\[ \neg A_1 \lor A_3 \lor A_9 \]
\[ \neg A_2 \lor \neg A_3 \lor A_4 \]
\[ \neg A_4 \lor A_5 \lor A_{10} \]
\[ \neg A_4 \lor A_6 \lor A_{11} \]
\[ \neg A_5 \lor \neg A_6 \]
\[ A_1 \lor A_7 \lor \neg A_{12} \]
\[ A_1 \lor A_8 \]
\[ \neg A_7 \lor \neg A_8 \lor \neg A_{13} \]

\[ \{ \ldots, \neg A_9, \neg A_{10}, \neg A_{11}, A_{12}, A_{13}, \ldots \} \text{ (initial assignment)} \]
Conflict-directed backtracking – example (cont.)

\[ \neg A_1 \lor A_2 \]
\[ \neg A_1 \lor A_3 \lor A_9 \]
\[ \neg A_2 \lor \neg A_3 \lor A_4 \]
\[ \neg A_4 \lor A_5 \lor A_{10} \]
\[ \neg A_4 \lor A_6 \lor A_{11} \]
\[ \neg A_5 \lor \neg A_6 \]
\[ A_1 \lor A_7 \lor \neg A_{12} \quad \text{true} \Rightarrow \text{removed} \]
\[ A_1 \lor A_8 \quad \text{true} \Rightarrow \text{removed} \]
\[ \neg A_7 \lor \neg A_8 \lor \neg A_{13} \]

...\n
\{ \ldots, \neg A_9, \neg A_{10}, \neg A_{11}, A_{12}, A_{13}, \ldots, A_1 \} \quad \text{(branch on } A_1) \]
Conflict-directed backtracking – example (cont.)

\[ \neg A_1 \lor A_2 \quad true \implies removed \]
\[ \neg A_1 \lor A_3 \lor A_9 \quad true \implies removed \]
\[ \neg A_2 \lor \neg A_3 \lor A_4 \]
\[ \neg A_4 \lor A_5 \lor A_{10} \]
\[ \neg A_4 \lor A_6 \lor A_{11} \]
\[ \neg A_5 \lor \neg A_6 \]
\[ A_1 \lor A_7 \lor \neg A_{12} \quad true \implies removed \]
\[ A_1 \lor A_8 \quad true \implies removed \]
\[ \neg A_7 \lor \neg A_8 \lor \neg A_{13} \]

\[ \{ \ldots, \neg A_9, \neg A_{10}, \neg A_{11}, A_{12}, A_{13}, \ldots, A_1, A_2, A_3 \} \]

(unit \( A_2, A_3 \))
Conflict-directed backtracking – example (cont.)

\[\neg A_1 \lor A_2 \quad \text{true } \Rightarrow \text{ removed}\]
\[\neg A_1 \lor A_3 \lor A_9 \quad \text{true } \Rightarrow \text{ removed}\]
\[\neg A_2 \lor \neg A_3 \lor A_4 \quad \text{true } \Rightarrow \text{ removed}\]
\[\neg A_4 \lor A_5 \lor A_{10}\]
\[\neg A_4 \lor A_6 \lor A_{11}\]
\[\neg A_5 \lor \neg A_6\]
\[A_1 \lor A_7 \lor \neg A_{12} \quad \text{true } \Rightarrow \text{ removed}\]
\[A_1 \lor A_8 \quad \text{true } \Rightarrow \text{ removed}\]
\[\neg A_7 \lor \neg A_8 \lor \neg A_{13}\]

... 

\{\ldots, \neg A_9, \neg A_{10}, \neg A_{11}, A_{12}, A_{13}, \ldots, A_1, A_2, A_3, A_4\} \\
(\text{unit } A_4)\]
Conflict-directed backtracking – example (cont.)

\[ \neg A_1 \lor A_2 \quad \text{true} \implies \text{removed} \]

\[ \neg A_1 \lor A_3 \lor A_9 \quad \text{true} \implies \text{removed} \]

\[ \neg A_2 \lor \neg A_3 \lor A_4 \quad \text{true} \implies \text{removed} \]

\[ \neg A_4 \lor A_5 \lor A_{10} \quad \text{true} \implies \text{removed} \]

\[ \neg A_4 \lor A_6 \lor A_{11} \quad \text{true} \implies \text{removed} \]

\[ \neg A_5 \lor \neg A_6 \quad \text{false} \implies \text{conflict} \]

\[ A_1 \lor A_7 \lor \neg A_{12} \quad \text{true} \implies \text{removed} \]

\[ A_1 \lor A_8 \quad \text{true} \implies \text{removed} \]

\[ \neg A_7 \lor \neg A_8 \lor \neg A_{13} \quad \text{...} \]

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

(unit \( A_5, A_6 \))
Conflict-directed backtracking – example (cont.)

\[ \neg A_1 \lor A_2 \quad true \implies removed \]
\[ \neg A_1 \lor A_3 \lor A_9 \quad true \implies removed \]
\[ \neg A_2 \lor \neg A_3 \lor A_4 \quad true \implies removed \]
\[ \neg A_4 \lor A_5 \lor A_{10} \quad true \implies removed \]
\[ \neg A_4 \lor A_6 \lor A_{11} \quad true \implies removed \]
\[ \neg A_5 \lor \neg A_6 \quad false \implies conflict \]
\[ A_1 \lor A_7 \lor \neg A_{12} \quad true \implies removed \]
\[ A_1 \lor A_8 \quad true \implies removed \]
\[ \neg A_7 \lor \neg A_8 \lor \neg A_{13} \]

\[ \implies \text{Conflict set: } \{ \neg A_9, \neg A_{10}, \neg A_{11}, A_1 \} \implies \text{backtrack to } A_1 \]
Conflict-directed backtracking – example (cont.)

\[ \neg A_1 \lor A_2 \quad \text{true} \implies \text{removed} \]
\[ \neg A_1 \lor A_3 \lor A_9 \quad \text{true} \implies \text{removed} \]
\[ \neg A_2 \lor \neg A_3 \lor A_4 \]
\[ \neg A_4 \lor A_5 \lor A_{10} \]
\[ \neg A_4 \lor A_6 \lor A_{11} \]
\[ \neg A_5 \lor \neg A_6 \]
\[ A_1 \lor A_7 \lor \neg A_{12} \]
\[ A_1 \lor A_8 \]
\[ \neg A_7 \lor \neg A_8 \lor \neg A_{13} \]
\[ \ldots \]
\[ \{\ldots, \neg A_9, \neg A_{10}, \neg A_{11}, A_{12}, A_{13}, \ldots, \neg A_1\} \text{ (branch on } \neg A_1) \]
Conflict-directed backtracking – example (cont.)

\[ \neg A_1 \lor A_2 \quad true \implies removed \]
\[ \neg A_1 \lor A_3 \lor A_9 \quad true \implies removed \]
\[ \neg A_2 \lor \neg A_3 \lor A_4 \]
\[ \neg A_4 \lor A_5 \lor A_{10} \]
\[ \neg A_4 \lor A_6 \lor A_{11} \]
\[ \neg A_5 \lor \neg A_6 \]
\[ A_1 \lor A_7 \lor \neg A_{12} \quad true \implies removed \]
\[ A_1 \lor A_8 \quad true \implies removed \]
\[ \neg A_7 \lor \neg A_8 \lor \neg A_{13} \quad false \implies conflict \]

...
Conflict-directed backtracking – example (cont.)

\[ \neg A_1 \lor A_2 \quad \text{true } \implies \text{removed} \]

\[ \neg A_1 \lor A_3 \lor A_9 \quad \text{true } \implies \text{removed} \]

\[ \neg A_2 \lor \neg A_3 \lor A_4 \]

\[ \neg A_4 \lor A_5 \lor A_{10} \]

\[ \neg A_4 \lor A_6 \lor A_{11} \]

\[ \neg A_5 \lor \neg A_6 \]

\[ A_1 \lor A_7 \lor \neg A_{12} \quad \text{true } \implies \text{removed} \]

\[ A_1 \lor A_8 \quad \text{true } \implies \text{removed} \]

\[ \neg A_7 \lor \neg A_8 \lor \neg A_{13} \quad \text{false } \implies \text{conflict} \]

\[ \implies \text{conflict set: } \{ A_{12}, A_{13}, \neg A_1 \} \]
Conflict-directed backtracking – example (cont.)

\[ \neg A_1 \lor A_2 \quad \text{true } \implies \text{removed} \]
\[ \neg A_1 \lor A_3 \lor A_9 \quad \text{true } \implies \text{removed} \]
\[ \neg A_2 \lor \neg A_3 \lor A_4 \]
\[ \neg A_4 \lor A_5 \lor A_{10} \]
\[ \neg A_4 \lor A_6 \lor A_{11} \]
\[ \neg A_5 \lor \neg A_6 \]
\[ A_1 \lor A_7 \lor \neg A_{12} \quad \text{true } \implies \text{removed} \]
\[ A_1 \lor A_8 \quad \text{true } \implies \text{removed} \]
\[ \neg A_7 \lor \neg A_8 \lor \neg A_{13} \quad \text{false } \implies \text{conflict} \]

...  

\[ \implies \text{conflict set: } \{ A_{12}, A_{13}, \neg A_1 \} \ldots \lor \{ \neg A_9, \neg A_{10}, \neg A_{11}, A_1 \} \]
\[ \implies \{ \neg A_9, \neg A_{10}, \neg A_{11}, A_{12}, A_{13} \} \implies \text{backtrack to } A_{13}. \]
\{-A_9, -A_{10}, -A_{11}, A_{12}, A_{13}\}
Learning [8, 51]

– **Idea:** When a conflict set $C$ is revealed, then $\neg C$ can be added to the clause set
  $\Rightarrow$ DPLL will never again generate an assignment containing $C$.
– **May avoid a lot of redundant search.**
– **Problem:** may cause a blowup in space
  $\Rightarrow$ techniques to control learning and to drop learned clauses when necessary
Learning – example (cont.)

\[
\neg A_1 \lor A_2 \quad \text{true} \implies \text{removed}
\]

\[
\neg A_1 \lor A_3 \lor A_9 \quad \text{true} \implies \text{removed}
\]

\[
\neg A_2 \lor \neg A_3 \lor A_4 \quad \text{true} \implies \text{removed}
\]

\[
\neg A_4 \lor A_5 \lor A_{10} \quad \text{true} \implies \text{removed}
\]

\[
\neg A_4 \lor A_6 \lor A_{11} \quad \text{true} \implies \text{removed}
\]

\[
\neg A_5 \lor \neg A_6 \quad \text{false} \implies \text{conflict}
\]

\[
A_1 \lor A_7 \lor \neg A_{12} \quad \text{true} \implies \text{removed}
\]

\[
A_1 \lor A_8 \quad \text{true} \implies \text{removed}
\]

\[
\neg A_7 \lor \neg A_8 \lor \neg A_{13}
\]

\[
\ldots
\]

\[
A_9 \lor A_{10} \lor A_{11} \lor \neg A_1 \quad \text{learned clause}
\]

\[
\implies \text{Conflict set: } \{ \neg A_9, \neg A_{10}, \neg A_{11}, A_1 \}
\]

\[
\implies \text{learn } A_9 \lor A_{10} \lor A_{11} \lor \neg A_1
\]
SOME APPLICATIONS
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.
Appl. #1: (Bounded) Planning
The problem [38, 37]

- **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 $o_1, \ldots, o_n$, leading from the initial state to the goal state.

- **Idea:** Encode it into satisfiability problem of a boolean formula $\varphi$
Example

\[ Move(b, s, d) \]

**Precond:** \( Block(b) \land Clear(b) \land On(b, s) \land \)

\( (Clear(d) \lor Table(d)) \land \)

\( b \neq s \land b \neq d \land s \neq d \)

**Effect:** \( Clear(s) \land \neg On(b, s) \land \)

\( On(b, d) \land \neg Clear(d) \)
Encoding

- Initial states:

\[ \text{On}_0(A, B), \text{On}_0(B, C), \text{On}_0(C, T), \text{Clear}_0(A). \]

- Goal states:

\[ \text{On}_{2n}(C, B) \land \text{On}_{2n}(B, A) \land \text{On}_{2n}(A, T). \]

- Action preconditions and effects:

\[ \text{Move}_t(A, B, C) \rightarrow \]
\[ \text{Clear}_{t-1}(A) \land \text{On}_{t-1}(A, B) \land \text{Clear}_{t-1}(C) \land \]
\[ \text{Clear}_{t+1}(B) \land \lnot \text{On}_{t+1}(A, B) \land \]
\[ \text{On}_{t+1}(A, C) \land \lnot \text{Clear}_{t+1}(C). \]
Encoding: Frame axioms

- Classic

\[ Move_t(A, B, T) \land Clear_{t-1}(C) \rightarrow Clear_{t+1}(C), \]
\[ Move_t(A, B, T) \land \neg Clear_{t-1}(C) \rightarrow \neg Clear_{t+1}(C). \]

"At least one action" axiom:

\[ \bigvee Move_t(b, s, d). \]
\[ b, s, d \in \{A, B, C, T\} \]
\[ b \neq s, b \neq d, s \neq d, b \neq T \]

- Explanatory

\[ \neg Clear_{t+1}(C') \land Clear_{t-1}(C') \rightarrow \]
\[ Move_t(A, B, C') \lor Move_t(A, T, C') \lor Move_t(B, A, C) \lor Move_t(B, T, C'). \]
Planning strategy

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

  \[
  \neg\text{Move}_t(A, B, C) \lor \neg\text{Move}_t(A, B, T). 
  \]

- **Parallel** for each pair of actions $\alpha$ and $\beta$, add axioms of the form $\neg\alpha_t \lor \neg\beta_t$ for each odd time step if $\alpha$ effects contradict $\beta$ preconditions. For example, we will have

  \[
  \neg\text{Move}_t(B, T, A) \lor \neg\text{Move}_t(A, B, C). 
  \]
Appl. #2: Bounded Model Checking
Bounded Planning

- Incomplete technique
- very efficient: competitive with state-of-the-art planners
- lots of enhancements [38, 37, 20, 26]
The problem [9]

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 $p$
  • $h \land g$, $h \lor g$, $Xg$, $Gg$, $Fg$, $hUg$ and $hRg$, $X$, $G$, $F$, $U$, $R$ “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 $f$?

$M \models^k E f$
The encoding

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

\[
[[M, f]]_k := [[M]]_k \land [[f]]_k
\] (1)

\[
[[M]]_k := I(s_0) \land \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}),
\] (2)

\[
[[f]]_k := \left( \neg \bigvee_{l=0}^{k} T(s_k, s_l) \land [[f]]_k^0 \right) \lor \bigvee_{l=0}^{k} (T(s_k, s_l) \land l[[f]]_k^0)
\] (3)
The encoding of \([[[f]]_k^i]\) and \(l[[f]]_k^i\)

<table>
<thead>
<tr>
<th></th>
<th>([[[f]]_k^i])</th>
<th>(l[[f]]_k^i)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(p)</td>
<td>(p_i)</td>
<td>(p_i)</td>
</tr>
<tr>
<td>(\neg p)</td>
<td>(\neg p_i)</td>
<td>(\neg p_i)</td>
</tr>
<tr>
<td>(h \land g)</td>
<td>([[h]]_k^i \land [[g]]_k^i)</td>
<td>(i[[h]]_k^i \land l[[g]]_k^i)</td>
</tr>
<tr>
<td>(h \lor g)</td>
<td>([[h]]_k^i \lor [[g]]_k^i)</td>
<td>(i[[h]]_k^i \lor l[[g]]_k^i)</td>
</tr>
<tr>
<td>(X_g)</td>
<td>(i[[g]]_k^{i+1}) if (i &lt; k) (\perp) otherwise.</td>
<td>(i[[g]]_k^{i+1}) if (i &lt; k) (i[[g]]_k^l) otherwise.</td>
</tr>
<tr>
<td>(G_g)</td>
<td>(\perp)</td>
<td>(\land_{j=\min(i,l)}^k l[[g]]_k^j)</td>
</tr>
<tr>
<td>(F_g)</td>
<td>(\lor_{j=i}^k [[g]]_k^j)</td>
<td>(\lor_{j=\min(i,l)}^k l[[g]]_k^j)</td>
</tr>
<tr>
<td>(h \mathcal{U} g)</td>
<td>(\lor_{j=i}^k \left( [[g]]<em>k^j \land \land</em>{n=i}^{j-1} [[h]]_k^n \right))</td>
<td>(\lor_{j=i}^k \left( l[[g]]<em>k^j \land \land</em>{n=i}^{j-1} l[[h]]<em>k^n \right) \lor \land</em>{n=i}^k l[[h]]<em>k^n \land \land</em>{n=l}^{j-1} l[[h]]_k^n)</td>
</tr>
<tr>
<td>(h \mathcal{R} g)</td>
<td>(\lor_{j=i}^k \left( [[h]]<em>k^j \land \land</em>{n=i}^{j-1} [[g]]_k^n \right))</td>
<td>(\land_{j=\min(i,l)}^k l[[g]]<em>k^j \lor \land</em>{j=i}^k \left( l[[h]]<em>k^j \land \land</em>{n=i}^{j-1} l[[g]]<em>k^n \right) \lor \land</em>{n=i}^k l[[h]]<em>k^n \land \land</em>{n=l}^{j-1} l[[g]]_k^n)</td>
</tr>
</tbody>
</table>
Example: $F_p$ (reachability)

- $f := F_p$: is there a reachable state in which $p$ holds?
- $[[M, f]]_k$ is:

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

- $f := G_p$: is there a path where $p$ holds forever?
- $[[M, f]]_k$ is:

$$I(s_0) \land \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \land \bigvee_{l=0}^{k} T(s_k, s_l) \land \bigwedge_{j=0}^{k} p_j$$
Example: $\mathbf{GF}_q \land \mathbf{F}_p$ (fair reachability)

- $f := \mathbf{GF}_q \land \mathbf{F}_p$: is there a reachable state in which $p$ holds provided that $q$ holds infinitely often?

- $[[M, f]]_k$ is:

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

- incomplete technique
- very efficient for some problems
- lots of enhancements [9, 1, 50, 54, 14]
PART 2:
BEYOND
PROPOSITIONAL
SATISFIABILITY
Goal

Integrate SAT procedures with domain-specific solvers in an efficient way

Different viewpoints:

- **(Computer scientists)** Extending SAT techniques to more expressive domains (preserving efficiency)
- **(Logicians)** Provide a new “SAT based” general framework from which to build efficient decision procedures (alternative, e.g., to semantic tableaux)
- ...
Key issues

- **Correctness, completeness & termination**
  - A general logic framework
  - A general integration schema

- **Efficiency**
  - Efficiency issues of the SAT procedure
  - Efficiency issues of the domain-specific solver
  - Efficiency of the integration
Formal Framework
Ingredients

- A **logic language** $\mathcal{L}$ extending boolean logic:
  - Language-specific **atomic expression** are formulas (e.g., $P(x)$, $\Box (A_1 \lor \Box A_2)$, $(x - y \geq 6)$, $\exists \text{ CHILDREN (male} \land \text{ teen})$)
  - if $\varphi_1$ and $\varphi_2$ formulas, then $\neg \varphi_1$, $\varphi_1 \land \varphi_2$, $\varphi_1 \lor \varphi_2$, $\varphi_1 \rightarrow \varphi_2$, $\varphi_1 \leftrightarrow \varphi_2$ are formulas.
  - Nothing else is a formula (e.g., no external quantifiers!)
Ingredients (cont.)

- A **semantic** for $\mathcal{L}$ extending standard boolean one:

  \[ M \models \psi, \quad (\psi \text{ atomic}) \iff [\text{definition specific for } \mathcal{L}] \]

  \[ M \models \neg \phi \iff M \not\models \phi \]

  \[ M \models \varphi_1 \land \varphi_2 \iff M \models \varphi_1 \text{ and } M \models \varphi_2 \]

  \[ M \models \varphi_1 \lor \varphi_2 \iff M \models \varphi_1 \text{ or } M \models \varphi_2 \]

  \[ M \models \varphi_1 \rightarrow \varphi_2 \iff \text{if } M \models \varphi_1 \text{ then } M \models \varphi_2 \]

  \[ M \models \varphi_1 \leftrightarrow \varphi_2 \iff M \models \varphi_1 \text{ iff } M \models \varphi_2 \]
Ingredients (cont.)

- A language-specific procedure $\mathcal{L}$-SOLVE able to decide the satisfiability of lists of atomic expressions and their negations

E.g.:

- $\text{FO-SOLVE}(\{P(x, a), P(b, y)\}) \rightarrow \text{Sat}$
- $\text{K-SOLVE}(\{\Box(A_1 \rightarrow A_2), \Box(A_1), \neg\Box(A_2)\}) \rightarrow \text{Unsat}$
- $\text{MATH-SOLVE}(\{(x - y \leq 3), (y - z \leq 4), \neg(x - z \leq 8)\}) \rightarrow \text{Unsat}$
- $\text{ALC-SOLVE} \left( \begin{array}{l}
  \forall \text{CHILDREN} \ (\neg \text{MALE} \lor \text{TEEN}), \\
  \forall \text{CHILDREN} \ (\text{MALE}), \\
  \exists \text{CHILDREN} \ (\neg \text{TEEN})
\end{array} \right) \rightarrow \text{Unsat}$
Definitions: atoms, literals

- An **atom** is every formula in $\mathcal{L}$ whose main connective is not a boolean operator.
- A **literal** is either an atom (a **positive** literal) or its negation (a **negative** literal).
- **Examples:**
  
  - $P(x), \neg \forall x. Q(x, f(a))$
  - $\Box(A_1 \lor \Box A_2), \neg \Box(A_2 \rightarrow \Box(A_3 \lor A_4))$
  - $(x - y \geq 6), \neg(z - y < 2), \exists \text{ CHILDREN (male} \land \text{ teen}), \neg \forall \text{ PARENT (old)}$
- **Atoms($\varphi$)**: the set of top-level atoms in $\varphi$. 
Definitions: total truth assignment

- We call a **total truth assignment** $\mu$ for $\varphi$ a **total function**

$$\mu : Atoms(\varphi) \rightarrow \{\top, \bot\}$$

- We represent a total truth assignment $\mu$ either as a **set** of literals

$$\mu = \{\alpha_1, \ldots, \alpha_N, \neg\beta_1, \ldots, \neg\beta_M, A_1, \ldots, A_R, \neg A_{R+1}, \ldots, \neg A_S\},$$

or as a **boolean formula**

$$\mu = \bigwedge_{i} \alpha_i \land \bigwedge_{j} \neg\beta_j \land \bigwedge_{k=1}^{R} A_k \land \bigwedge_{h=R+1}^{S} \neg A_h$$
Definitions: partial truth assignment

- We call a **partial truth assignment** \( \mu \) for \( \varphi \) a **partial function**

\[
\mu : \text{Atoms}(\varphi) \rightarrow \{\top, \bot\}
\]

- Partial truth assignments can be represented as sets of literals or as boolean functions, as before.

- A partial truth assignment \( \mu \) for \( \varphi \) is a subset of a total truth assignment for \( \varphi \).

- If \( \mu_2 \subseteq \mu_1 \), then we say that \( \mu_1 \) **extends** \( \mu_2 \) and that \( \mu_2 \) **subsumes** \( \mu_1 \).

- A **conflict set** for \( \mu_1 \) is an inconsistent subset \( \mu_2 \subseteq \mu_1 \) s.t. no strict subset of \( \mu_2 \) is inconsistent.
Definitions: total and partial truth assignment (cont.)

Remark:

– **Syntactically identical instances of the same atom** in $\varphi$ are always assigned identical truth values.

E.g., $\ldots \land ((t_1 \geq t_2) \lor A_1) \land ((t_1 \geq t_2) \lor A_2) \land \ldots$

– **Equivalent but syntactically different atoms** in $\varphi$ may (in principle) be assigned different truth values.

E.g., $\ldots \land ((t_1 \geq t_2) \lor A_1) \land ((t_2 \leq t_1) \lor A_2) \land \ldots$
Definition: propositional satisfiability in $\mathcal{L}$

A truth assignment $\mu$ for $\varphi$ propositionally satisfies $\varphi$ in $\mathcal{L}$, written $\mu \models_p \varphi$, iff it makes $\varphi$ evaluate to $\top$:

\[
\begin{align*}
\mu \models_p \varphi_1, \quad \varphi_1 & \in \text{Atoms}(\varphi) \iff \varphi_1 \in \mu; \\
\mu \models_p \neg \varphi_1 & \iff \mu \not\models_p \varphi_1; \\
\mu \models_p \varphi_1 \land \varphi_2 & \iff \mu \models_p \varphi_1 \text{ and } \mu \models_p \varphi_2.
\end{align*}
\]

- A partial assignment $\mu$ propositionally satisfies $\varphi$ iff all total assignments extending $\mu$ propositionally satisfy $\varphi$. 

Definition: propositional satisfiability in $\mathcal{L}$ (cont)

- **Intuition:** If $\varphi$ is seen as a boolean combination of its atoms, $\models_p$ is standard propositional satisfiability.
- Atoms seen as (recognizable) blackboxes
- The definitions of $\models_p \varphi_1 \models_p \varphi_2$, $\models_p \varphi$ is straightforward.
- $\models_p$ stronger than $\models$: if $\varphi_1 \models_p \varphi_2$, then $\varphi_1 \models \varphi_2$, but not vice versa.

E.g., $(v_1 \leq v_2) \land (v_2 \leq v_3) \models (v_1 \leq v_3)$, but
$(v_1 \leq v_2) \land (v_2 \leq v_3) \not\models_p (v_1 \leq v_3)$. 
Satisfiability and propositional satisfiability in $\mathcal{L}$

**Proposition**: $\varphi$ is satisfiable in $\mathcal{L}$ iff there exists a truth assignment $\mu$ for $\varphi$ s.t.

- $\mu \models_p \varphi$, and
- $\mu$ is satisfiable in $\mathcal{L}$.

---

Search decomposed into two orthogonal components:

- **Purely propositional**: search for a truth assignments $\mu$ propositionally satisfying $\varphi$
- **Purely domain-dependent**: verify the satisfiability in $\mathcal{L}$ of $\mu$. 
Example

\[ \varphi = \{ \neg(2v_2 - v_3 > 2) \lor A_1 \} \land \\
\{ \neg A_2 \lor (2v_1 - 4v_5 \geq 3) \} \land \\
\{ (3v_1 - 2v_2 \leq 3) \lor A_2 \} \land \\
\{ \neg (2v_3 + v_4 \geq 5) \lor \neg (3v_1 - v_3 \leq 6) \lor \neg A_1 \} \land \\
\{ A_1 \lor (3v_1 - 2v_2 \leq 3) \} \land \\
\{ (v_1 - v_5 \leq 1) \lor (v_5 = 5 - 3v_4) \lor \neg A_1 \} \land \\
\{ A_1 \lor (v_3 = 3v_5 + 4) \lor A_2 \}. \]

\[ \mu = \{ \neg(2v_2 - v_3 > 2), \neg A_2, (3v_1 - 2v_2 \leq 3), (v_1 - v_5 \leq 1), \neg (3v_1 - v_3 \leq 6), (v_3 = 3v_5 + 4) \}. \]

\[ \mu' = \{ \neg(2v_2 - v_3 > 2), \neg A_2, \neg A_1, (3v_1 - 2v_2 \leq 3), (v_3 = 3v_5 + 4) \}. \]

- \( \mu \models_p \varphi \), but is unsatisfiable, as contains conflict sets:

\[ \{ (3v_1 - 2v_2 \leq 3), \neg (2v_2 - v_3 > 2), \neg (3v_1 - v_3 \leq 6) \} \]

\[ \{ (v_1 - v_5 \leq 1), (v_3 = 3v_5 + 4), \neg (3v_1 - v_3 \leq 6) \} \]

- \( \mu' \models_p \varphi \), and is satisfiable \((v_1, v_2, v_3 := 0, v_5 := -4/3)\).
Complete collection of assignments

A collection \( \mathcal{M} = \{\mu_1, \ldots, \mu_n\} \) of (possibly partial) assignments propositionally satisfying \( \varphi \) is complete iff

\[
\models_p \varphi \iff \bigvee_j \mu_j. \tag{4}
\]

- for every total assignment \( \eta \) s.t. \( \eta \models_p \varphi \), there is \( \mu_i \in \mathcal{M} \) s.t. \( \mu_i \subseteq \eta \).
- \( \mathcal{M} \) "compact" representation of the whole set of total assignments propositionally satisfying \( \varphi \).
Complete collection of assignments and satisfiability in \( \mathcal{L} \)

Proposition. Let \( \mathcal{M} = \{\mu_1, \ldots, \mu_n\} \) be a complete collection of truth assignments propositionally satisfying \( \varphi \). Then \( \varphi \) is satisfiable if and only if \( \mu_j \) is satisfiable for some \( \mu_j \in \mathcal{M} \).

– Search decomposed into two orthogonal components:
  
  ● **Purely propositional:** generate (in a lazy way) a complete collection \( \mathcal{M} = \{\mu_1, \ldots, \mu_n\} \) of truth assignments propositionally satisfying \( \varphi \);
  
  ● **Purely domain-dependent:** check one by one the satisfiability in \( \mathcal{L} \) of the \( \mu_i \)'s.
Redundancy of complete collection of assignments

A complete collection $\mathcal{M} = \{\mu_1, \ldots, \mu_n\}$ of assignments propositionally satisfying $\varphi$ is

- strongly non redundant iff, for every $\mu_i, \mu_j \in \mathcal{M}$, $(\mu_i \land \mu_j)$ is propositionally unsatisfiable,
- non redundant iff, for every $\mu_j \in \mathcal{M}$, $\mathcal{M} \setminus \{\mu_j\}$ is no more complete,
- redundant otherwise.
- If \( \mathcal{M} \) is redundant, then \( \mu_j \supseteq \mu_i \) for some \( \mu_i, \mu_j \in \mathcal{M} \):

\[
\models_p \varphi \iff \bigvee_{i \neq j} \mu_i \quad \implies \quad \models_p \bigvee_i \mu_i \iff \bigvee_{i \neq j} \mu_i \\
\bigvee_i \mu_i \models_p \bigvee_{i \neq j} \mu_i \quad \implies \quad \mu_j \models_p \bigvee_{i \neq j} \mu_i \\
\mu_j \models_p \mu_i \text{ for some } i \quad \implies \quad \mu_j \supseteq \mu_i
\]

- If \( \mathcal{M} \) is strongly non redundant, then \( \mathcal{M} \) is non redundant:

\[
\mu_j \land \mu_i \text{ propositionally inconsistent} \implies
\mu_j \models_p \neg \mu_i
\]

\( \mathcal{M} \) non redundant
Redundancy: example

Let $\varphi := (\alpha \lor \beta \lor \gamma) \land (\alpha \lor \beta \lor \lnot \gamma)$, $\alpha$, $\beta$, $\gamma$ atoms. Then

1. $\{\{\alpha, \beta, \gamma\}, \{\alpha, \beta, \lnot \gamma\}, \{\alpha, \lnot \beta, \gamma\}, \{\alpha, \lnot \beta, \lnot \gamma\},$
   $\{|\lnot \alpha, \beta, \gamma\}, \{|\lnot \alpha, \beta, \lnot \gamma\}\}$ is the set of all total assignments propositionally satisfying $\varphi$;

2. $\{\{\alpha\}, \{\alpha, \beta\}, \{\alpha, \lnot \gamma\}, \{\alpha, \beta\}, \{\beta\}, \{\beta, \lnot \gamma\},$
   $\{\alpha, \gamma\}, \{\beta, \gamma\}\}$ is complete but redundant;

3. $\{\{\alpha\}, \{\beta\}\}$ is complete, non redundant but not strongly non redundant;

4. $\{\{\alpha\}, \{|\lnot \alpha, \beta\}\}$ is complete and strongly non redundant.
A Generalized Search Procedure
Truth assignment enumerator

A truth assignment enumerator is a total function $\text{Assign Enumerator}()$ which takes as input a formula $\varphi$ in $\mathcal{L}$ and returns a complete collection $\{\mu_1, \ldots, \mu_n\}$ of assignments propositionally satisfying $\varphi$.

- A truth assignment enumerator is
  - strongly non-redundant if $\text{Assign Enumerator}(\varphi)$ is strongly non-redundant, for every $\varphi$,
  - non-redundant if $\text{Assign Enumerator}(\varphi)$ is non-redundant, for every $\varphi$,
  - redundant otherwise.
Remark. Notice the difference:

- A SAT solver has to find only one satisfying assignment — or to decide there is none;
- A Truth assignment enumerator has to find a complete collection of satisfying assignments.
A generalized procedure

boolean \textsc{L-SAT}(\textit{formula} \(\varphi\), \textit{assignment} & \(\mu\), \textit{model} & \(M\))

\begin{verbatim}
do
  \(\mu := \text{Next\_Assignment}(\varphi)\)  /* next in \(\{\mu_1, \ldots, \mu_n\}\) */
  if (\(\mu \neq \text{Null}\))
    satifiable := \textsc{L-SOLVE}(\mu, M);
while ((satifiable = \text{False}) and (\(\mu \neq \text{Null}\)))
  if (satifiable \(\neq \text{False}\))
    then return \textit{True};  /* a satisf. assignment found */
else return \textit{False};  /* no satisf. assignment found */
\end{verbatim}
\(\mathcal{L}\)-SAT

- \(\mathcal{L}\)-SAT(\(\varphi\)) terminating, correct and complete \iff \(\mathcal{L}\)-SOLVE(\(\mu\)) terminating, correct and complete.

- \(\mathcal{L}\)-SAT depends on \(\mathcal{L}\) only for \(\mathcal{L}\)-SOLVE

- \(\mathcal{L}\)-SAT requires polynomial space iff
  - \(\mathcal{L}\)-SOLVE requires polynomial space and
  - ASSIGN_ENUMERATOR is lazy
Mandatory requirements for an assignment enumerator

An assignment enumerator must always:

- *(Termination)* terminate
- *(Correctness)* generate assignments propositionally satisfying $\varphi$
- *(Completeness)* generate complete set of assignments
Mandatory requirements for $\mathcal{L}$-SOLVE()

$\mathcal{L}$-SOLVE() must always:

- (Termination) terminate
- (Correctness & completeness) return $True$ if $\mu$ is satisfiable in $\mathcal{L}$, $False$ otherwise
Efficiency requirements for an assignent enumerator

To achieve the maximum efficiency, an assignent enumerator should:

- (Laziness) generate the assignments one-at-a-time.
- (Polynomial Space) require only polynomial space
- (Strong Non-redundancy) be strongly non-redundant
- (Time efficiency) be fast
- [(Symbiosis with $\mathcal{L}$-SOLVE) be able to take benefit from failure & success information provided by $\mathcal{L}$-SOLVE (e.g., conflict sets, inferred assignments)]]
Benefits of (strongly) non-redundant generators

- **Non-redundant enumerators** avoid generating partial assignments whose unsatisfiability is a propositional consequence of those already generated.

- **Strongly non-redundant enumerators** avoid generating partial assignments covering areas of the search space which are covered by already-generated ones.

- **Strong non-redundancy** provides a logical warrant that an already generated assignment will never be generated again.
  \[\implies\] no extra control required to avoid redundancy.
Efficiency requirements for $\mathcal{L}$-SOLVE()

To achieve the maximum efficiency, $\mathcal{L}$-SOLVE() should:

- (Time efficiency) be fast
- (Polynomial Space) require only polynomial space
- [(Symbiosis with ASSIGN_ENUMERATOR) be able to produce failure & success information (e.g., conflict sets, inferred assignments)]
- [(Incrementality) be incremental: $\mathcal{L}$-SOLVE($\mu_1 \cup \mu_2$) reuses computation of $\mathcal{L}$-SOLVE($\mu_1$)]
Extending existing SAT procedures
General ideas

Existing SAT procedures are natural candidates to be used as assignment enumerators.

- Atoms labelled by propositional atoms
- Slight modifications
  (backtrack when assignment found)
- Completeness to be verified!
  (E.g., DPLL with Pure literal)
- Candidates: OBDDs, Semantic Tableaux, DPLL
OBDDs

- In an OBDD, the set of paths from the root to (1) represent a complete collection of assignments
- Some may be inconsistent in $\mathcal{L}$
- Reduction: [13, 42]
  1. inconsistent paths from the root to internal nodes are detected
  2. they are redirected to the (0) node
  3. the resulting OBDD is simplified.
OBDD: example

OBDD

OBDD of \((\alpha \lor \beta \lor \gamma) \land (\alpha \lor \beta \lor \neg \gamma)\).
OBDD reduction: example

Reduced OBDD of $(\alpha \lor \beta \lor \gamma) \land (\alpha \lor \beta \lor \neg \gamma)$, 
\(\alpha := (x - y \leq 4), \beta := (x - y \leq 2)\).
OBDD: summary

- strongly non-redundant
- time-efficient
- factor sub-graphs
- require exponential memory
- non lazy
- [allow for early pruning]
- [do not allow for backjumping or learning]
Generalized semantic tableaux

- General rules = propositional rules + $\mathcal{L}$-specific rules

\[
\begin{align*}
\frac{\varphi_1 \land \varphi_2}{\varphi_1} & \quad \frac{\neg (\varphi_1 \lor \varphi_2)}{\neg \varphi_1} & \quad \frac{\neg (\varphi_1 \rightarrow \varphi_2)}{\varphi_1} \\
\frac{\varphi_2}{\varphi_1} & \quad \frac{\neg \varphi_1}{\neg \varphi_2} & \quad \frac{\neg \varphi_2}{\varphi}
\end{align*}
\]

\[
\begin{align*}
\frac{\varphi_1 \lor \varphi_2}{\varphi_1} & \quad \frac{\neg (\varphi_1 \land \varphi_2)}{\neg \varphi_1} & \quad \frac{\varphi_1 \rightarrow \varphi_2}{\neg \varphi_1} \\
\frac{\varphi_2}{\varphi} & \quad \frac{\neg \varphi_1}{\neg \varphi_2} & \quad \frac{\neg \varphi_2}{\varphi_1}
\end{align*}
\]

\[
\left\{ \text{$\mathcal{L}$-specific Rules} \right\}
\]

- Widely used by logicians
Generalized tableau algorithm

function \( \mathcal{L}\text{-Tableau}(\Gamma) \)

if \( A_i \in \Gamma \) and \( \lnot A_i \in \Gamma \)  /* branch closed */
then return \( \text{False} \);
if \( (\varphi_1 \land \varphi_2) \in \Gamma \)  /* \( \land \)-elimination */
then return \( \mathcal{L}\text{-Tableau}(\Gamma \cup \{\varphi_1, \varphi_2\}\setminus\{(\varphi_1 \land \varphi_2)\}) \);
if \( (\lnot \lnot \varphi_1) \in \Gamma \)  /* \( \lnot \lnot \)-elimination */
then return \( \mathcal{L}\text{-Tableau}(\Gamma \cup \{\varphi_1\}\setminus\{(\lnot \lnot \varphi_1)\}) \);
if \( (\varphi_1 \lor \varphi_2) \in \Gamma \)  /* \( \lor \)-elimination */
then return \( \mathcal{L}\text{-Tableau}(\Gamma \cup \{\varphi_1\}\setminus\{(\varphi_1 \lor \varphi_2)\}) \)  or  \\
\( \mathcal{L}\text{-Tableau}(\Gamma \cup \{\varphi_2\}\setminus\{(\varphi_1 \lor \varphi_2)\}) \);

...  

return \( (\mathcal{L}\text{-SOLVE}(\Gamma)= \text{satisfiable}) \);  /* branch expanded */
General tableaux: example

Tableau Search Graph

Tableau search graph for \((\alpha \lor \beta \lor \gamma) \land (\alpha \lor \beta \lor \neg \gamma)\).
Generalized tableaux: problems

Two main problems [16, 30, 31]

- syntactic branching
  - branch on disjunctions
  - possible many duplicate or subsumed branches
    \[\text{\longrightarrow redundant}\]
  - duplicates search (both propositional and domain-dependent)
- no constraint violation detection
  - incapable to detect when current branches violate a constraint
    \[\text{\longrightarrow lots of redundant propositional search.}\]
Syntactic branching: example

Tableau search graph for \((\alpha \lor \neg \beta) \land (\alpha \lor \beta) \land (\neg \alpha \lor \neg \beta)\).
Detecting constraints violations: example

Tableau search graph for
\[(\alpha \lor \phi_1) \land (\beta \lor \phi_2) \land \phi_3 \land (\neg \alpha \lor \neg \beta)\]
Generalized tableaux: summary

- lazy
- require polynomial memory
- redundant
- time-inefficient
- [allow backjumping]
- [do not allow learning]
Remark.
The word “Tableau” is a bit overloaded in literature. Some existing (and rather efficient) systems, like FacT and DLP [35], call themselves “Tableau” procedures, although they use a DPLL-like technique to perform boolean reasoning. Same discourse holds for the boolean system KE [16] and its derived systems.
Generalized DPLL

- **General rules** = propositional rules + $\mathcal{L}$-specific rules

\[
\begin{align*}
\frac{\varphi_1 \land (l) \land \varphi_2}{(\varphi_1 \land \varphi_2)[l|\top]} & & (Unit) \\
\frac{\varphi}{\varphi[l|\top] \varphi[l|\bot]} & & (split)
\end{align*}
\]

- **No Pure Literal Rule:** Pure literal causes incomplete assignment sets!
Pure literal and Generalized DPLL: Example

\[ \varphi = ((x - y \leq 1) \lor A_1) \land \\
(((y - z \leq 2) \lor A_2) \land \\
(\neg(x - z \leq 4) \lor A_2) \land \\
(\neg A_2 \lor A_3) \land \\
(\neg A_2 \lor \neg A_3) \]

- A satisfiable assignment propositionally satisfying \( \varphi \) is:
  \[ \mu = \{ A_1, \neg A_2, (y - z \leq 2), \neg(x - z \leq 4) \} \]

- No satisfiable assignment propositionally satisfying \( \varphi \) contains \( (x - y \leq 1) \)

- Pure literal may assign \( (x - y \leq 1) := \top \) as first step
  \[ \implies \text{return unsatisfiable.} \]
Generalized DPLL algorithm

function \( \mathcal{L}\text{-}DPLL(\varphi,\mu) \)

if \( \varphi = \top \)

then return \( (\mathcal{L}\text{-}\text{SOLVE}(\mu) = \text{satisfiable}) \);

if \( \varphi = \bot \)

then return False;

if \{ \text{a unit clause} \( l \) \text{ occurs in } \varphi \} 

then return \( \mathcal{L}\text{-}DPLL(\text{assign}(l,\varphi),\mu \land l) \);

\( l := \text{choose-literal}(\varphi) \);

return \( \mathcal{L}\text{-}DPLL(\text{assign}(l,\varphi),\mu \land l) \text{ or } \mathcal{L}\text{-}DPLL(\text{assign}(\neg l,\varphi),\mu \land \neg l) \);
General DPLL: example

DPLL search graph

\begin{itemize}
    \item $\alpha$
    \item $\bar{\alpha}$
    \item $\{\alpha\}$
    \item $\beta$
    \item $\bar{\beta}$
    \item $\{-\alpha,\beta\}$
    \item $\gamma$
    \item $\times$
\end{itemize}

DPLL search graph for \((\alpha \lor \beta \lor \gamma) \land (\alpha \lor \beta \lor \neg \gamma)\).
Generalized DPLL vs. generalized tableau

Two big advantages: [16, 30, 31]

- **semantic vs. syntactic branching**
  - branch on **truth values**
  - no duplicate or subsumed branches
    \[\implies\text{strongly non redundant}\]
  - no search duplicates

- **constraint violation detection**
  - backtracks as soon as the current branch violates a constraint
    \[\implies\text{no redundant propositional search.}\]
Semantic branching: example

Tableau search graph for \((\alpha \lor \neg \beta) \land (\alpha \lor \beta) \land (\neg \alpha \lor \neg \beta)\).
Detecting constraints violations: example

DPLL search graph for \((\alpha \lor \phi_1) \land (\beta \lor \phi_2) \land \phi_3 \land (\neg \alpha \lor \neg \beta)\)
Generalized DPLL: summary

- lazy
- require polynomial memory
- strongly non redundant
- time-efficient
- [allow backjumping and learning]
Optimizations
Possible Improvements

- Preprocessing atoms [29, 35, 5]
- Static learning [2]
- Early pruning [29, 13, 4]
- Enhanced Early pruning [4]
- Backjumping [35, 55]
- Memoizing [35, 25]
- Learning [35, 55]
- Forward Checking [2]
- Triggering [55, 4]
Preprocessing atoms [29, 35, 5]

Source of inefficiency: semantically equivalent but syntactically different atoms are not recognized to be identical [resp. one the negation of the other] $\iff$ they may be assigned different [resp. identical] truth values.

Solution: rewrite trivially equivalent atoms into one.
Preprocessing atoms (cont.)

- **Sorting:** \((v_1 + v_2 \leq v_3 + 1), (v_2 + v_1 \leq v_3 + 1), (v_1 + v_2 - 1 \leq v_3) \implies (v_1 + v_2 - v_3 \leq 1)\);

- **Rewriting dual operators:**
  \((v_1 < v_2), (v_1 \geq v_2) \implies (v_1 < v_2), \neg(v_1 < v_2)\)

- **Exploiting associativity:**
  \((v_1 + (v_2 + v_3) = 1), ((v_1 + v_2) + v_3) = 1) \implies (v_1 + v_2 + v_3 = 1)\);

- **Factoring** \((v_1 + 2.0v_2 \leq 4.0), (-2.0v_1 - 4.0v_2 \geq -8.0), \implies (0.25v_1 + 0.5v_2 \leq 1.0)\);

- **Exploiting properties of \(L\):**
  \((v_1 \leq 3), (v_1 < 4) \implies (v_1 \leq 3) \text{ if } v_1 \in \mathbb{Z};\)

- ...
Preprocessing atoms: summary

- Very efficient with DPLL
- Presumably very efficient with OBDDs
- Scarcely efficient with semantic tableaux
Static learning [2]

– **Rationale:** Many literals are mutually exclusive (e.g., \((x - y < 3), \neg(x - y < 5)\))

– **Preprocessing step:** detect these literals and add binary clauses to the input formula: (e.g., \(\neg(x - y < 3) \lor (x - y < 5)\))

– (with DPLL) assignments including both literals are never generated.

– requires \(O(|\varphi|^2)\) steps.
Static learning (cont.)

- Very efficient with DPLL
- Possibly very efficient with OBDDs (?)
- Completely ineffective with semantic tableaux
Early pruning [29, 13, 4]

– **rationale:** if an assignment $\mu'$ is unsatisfiable, then all its extensions are unsatisfiable.

– the unsatisfiability of $\mu'$ detected during its construction, avoids checking the satisfiability of all the up to $2^{|Atoms(\varphi)| - |\mu'|}$ assignments extending $\mu'$.

– Introduce a satisfiability test on incomplete assignments just **before every branching step**:

\[
\text{if Likely-Unsatisfiable}(\mu) \quad /* \text{early pruning} */ \\
\quad \text{if } (L-SOLVE(\mu) = False) \quad \text{then return } False;
\]
function $\mathcal{L}$-DPLL($\varphi$, $\mu$)

if $\varphi = T$ /* base */
  then return ($\mathcal{L}$-SOLVE($\mu$)=satisfiable);
if $\varphi = \bot$ /* backtrack */
  then return False;
if {a unit clause ($l$) occurs in $\varphi$} /* unit */
  then return $\mathcal{L}$-DPLL(assign($l$, $\varphi$), $\mu \land l$);
if Likely-Unsatisfiable($\mu$) /* early pruning */
  if ($\mathcal{L}$-SOLVE($\mu$) = False)
    then return False;

$l :=$ choose-literal($\varphi$); /* split */

return $\mathcal{L}$-DPLL(assign($l$, $\varphi$), $\mu \land l$) or
         $\mathcal{L}$-DPLL(assign($\neg l$, $\varphi$), $\mu \land \neg l$);
Early pruning: example

\[ \varphi = \:\{ \neg(2v_2 - v_3 > 2) \lor A_1 \} \land \\
\{ \neg A_2 \lor (2v_1 - 4v_5 > 3) \} \land \\
\{ (3v_1 - 2v_2 \leq 3) \lor A_2 \} \land \\
\{ \neg(2v_3 + v_4 \geq 5) \lor \neg(3v_1 - v_3 \leq 6) \lor \neg A_1 \} \land \\
\{ A_1 \lor (3v_1 - 2v_2 \leq 3) \} \land \\
\{ (v_1 - v_5 \leq 1) \lor (v_5 = 5 - 3v_4) \lor \neg A_1 \} \land \\
\{ A_1 \lor (v_3 = 3v_5 + 4) \lor A_2 \}. \]

- Suppose it is built the intermediate assignment:

\[ \mu' = \neg(2v_2 - v_3 > 2) \land \neg A_2 \land (3v_1 - 2v_2 \leq 3) \land \neg(3v_1 - v_3 \leq 6). \]

- If \( \mathcal{L}\text{-SOLVE} \) is invoked on \( \mu' \), it returns \texttt{False}, and \( \mathcal{L}\text{-DPLL} \) backtracks without exploring any extension of \( \mu' \).
Early pruning: drawback

- Reduces drastically the search
- **Drawback:** possibly lots of useless calls to $\mathcal{L}$-SOLVE
  \(\iff\) to be used with care when $\mathcal{L}$-SOLVE calls recursively $\mathcal{L}$-SAT (e.g., with modal logics)
- Roughly speaking, worth doing when each branch saves at least one branching
- **Possible solutions:**
  - introduce a **selective heuristic** Likely-unsatisfiable
  - use **incremental versions** of $\mathcal{L}$-SOLVE
  one split.
Early pruning: Likely-unsatisfiable

- **Rationale:** if no literal which may likely cause conflict with the previous assignment has been added since last call, return false.
- **Examples:** return false if they are added only
  - boolean literals
  - disequalities $(x - y \neq 3)$
  - atoms introducing new variables $(x - z \neq 3)$
  - ...
Early pruning: incrementality of $\mathcal{L}$-SOLVE

- With early pruning, lots of incremental calls to $\mathcal{L}$-SOLVE:
  
  \[ \mathcal{L}\text{-SOLVE}(\mu) \implies \text{satisfiable} \]
  \[ \mathcal{L}\text{-SOLVE}(\mu \cup \mu') \implies \text{satisfiable} \]
  \[ \mathcal{L}\text{-SOLVE}(\mu \cup \mu' \cup \mu'') \implies \text{satisfiable} \]
  ...

- $\mathcal{L}$-SOLVE incremental: $\mathcal{L}$-SOLVE($\mu_1 \cup \mu_2$) reuses computation of $\mathcal{L}$-SOLVE($\mu_1$) without restarting from scratch $\implies$ lots of computation saved

- requires saving the status of $\mathcal{L}$-SOLVE
Early pruning: summary

- Very efficient with DPLL & OBDDs
- Possibly very efficient with semantic tableaux (?)
- In some cases may introduce big overhead (e.g., modal logics)
- Benefits if $\mathcal{L}$-SOLVE is incremental
Enhanced Early Pruning [4]

- In early pruning, \( \mathcal{L}\text{-SOLVE} \) is not effective if it returns “satisfiable”.
- \( \mathcal{L}\text{-SOLVE}(\mu) \) may be able to derive deterministically a sub-assignment \( \eta \) s.t. \( \mu \models \eta \), and return it.
- The literals in \( \eta \) are then unit-propagated away.
Enhanced Early Pruning: Examples

(We assume that all the following literals occur in \( \varphi \).)

- If \((v_1 - v_2 \leq 4) \in \mu\) and \((v_1 - v_2 \leq 6) \notin \mu\), then \(\mathcal{L}\text{-SOLVE}\) can derive \((v_1 - v_2 \leq 6)\) from \(\mu\).

- If \((v_1 - v_3 > 2), (v_2 = v_3) \in \mu\) and \((v_1 - v_2 > 2) \notin \mu\), then \(\mathcal{L}\text{-SOLVE}\) can derive \((v_1 - v_2 > 2)\) from \(\mu\).
Enhanced Early Pruning: summary

- Further improves efficiency with DPLL
- Presumably scarcely effective with semantic tableaux
- Effective with OBDDs?
- Requires a sophisticated $\mathcal{L}$-SOLVE
Backjumping (driven by $\mathcal{L}$-SOLVE) [35, 55]

- Similar to SAT backjumping
- **Rationale:** same as for early pruning
- **Idea:** when a branch is found unsatisfiable in $\mathcal{L}$,
  1. $\mathcal{L}$-SOLVE returns the **conflict set** causing the failure
  2. $\mathcal{L}$-SAT backtracks to the **most recent branching point** in the conflict set
Backjumping: Example

\[
\varphi = \{ \neg(2v_2 - v_3 > 2) \lor A_1 \} \land \\
\{ \neg A_2 \lor (2v_1 - 4v_5 > 3) \} \land \\
\{ (3v_1 - 2v_2 \leq 3) \lor A_2 \} \land \\
\{ \neg(2v_3 + v_4 \geq 5) \lor \neg(3v_1 - v_3 \leq 6) \lor \neg A_1 \} \land \\
\{ A_1 \lor (3v_1 - 2v_2 \leq 3) \} \land \\
\{ (v_1 - v_5 \leq 1) \lor (v_5 = 5 - 3v_4) \lor \neg A_1 \} \land \\
\{ A_1 \lor (v_3 = 3v_5 + 4) \lor A_2 \}.
\]

\[
\mu = \{ \neg(2v_2 - v_3 > 2), \neg A_2, (3v_1 - 2v_2 \leq 3), (v_1 - v_5 \leq 1), \neg(3v_1 - v_3 \leq 6), (v_3 = 3v_5 + 4) \}.
\]

- \( \mathcal{L}\text{-SOLVE}(\mu) \) returns \textit{false} with the \textit{conflict set}:
  \[
  \{ (3v_1 - 2v_2 \leq 3), \neg(2v_2 - v_3 > 2), \neg(3v_1 - v_3 \leq 6) \}
  \]

- \( \mathcal{L}\text{-SAT} \) can jump back directly to the branching point \( \neg(3v_1 - v_3 \leq 6) \), without branching on \( (v_3 = 3v_5 + 4) \).
Backjumping vs. Early Pruning

- Backjumping requires no extra calls to $L$-SOLVE.
- Effectiveness depends on the conflict set $C$, i.e., on how recent the most recent branching point in $C$ is.
- Example: no pruning effect with the conflict set:
  \[
  \{(v_1 - v_5 \leq 1), (v_3 = 3v_5 + 4), \neg(3v_1 - v_3 \leq 6)\}
  \]
- Same pruning effect as with Early Pruning only with the best conflict set.
- More effective than Early Pruning only when the overhead compensates the pruning effect (e.g., modal logics with high depths).
Backjumping: summary

- Very efficient with DPLL
- Never applied to OBDDs
- Very efficient with semantic tableaux
- Alternative to but less effective than early pruning.
- No significant overhead
- $\mathcal{L}$-SOLVE must be able to detect conflict sets.
Memoizing [35, 25]

– Idea 1:
  ● When a conflict set $C$ is revealed, then $C$ can be cached into an ad hoc data structure
  ● $\mathcal{L}$-SOLVE($\mu$) checks first if (any subset of) $\mu$ is cached. If yes, returns unsatisfiable.
– Idea 2:
  ● When a satisfying (sub)-assignment $\mu'$ is found, then $\mu'$ can be cached into an ad hoc data structure
  ● $\mathcal{L}$-SOLVE($\mu$) checks first if (any superset of) $\mu$ is cached. If yes, returns satisfiable.
Memoizing (cont.)

- Can dramatically prune search.
- May cause a blowup in memory.
- Applicable also to semantic tableaux.
- Idea 1 subsumed by learning.
Learning (driven by $\mathcal{L}$-SOLVE) [35, 55]

- Similar to SAT learning
- **Idea:** When a conflict set $C'$ is revealed, then $\neg C'$ can be added to the clause set
  $\implies$ DPLL will never again generate an assignment containing $C'$.
- **May avoid a lot of redundant search.**
- **Problem:** may cause a blowup in space
  $\implies$ techniques to control learning and to drop learned clauses when necessary
Learning: example

- \( \mathcal{L}\text{-SOLVE} \) returns the conflict set:
  \[
  \{(3v_1 - 2v_2 \leq 3), \neg(2v_2 - v_3 > 2), \neg(3v_1 - v_3 \leq 6)\}
  \]
- it is added the clause
  \[
  \neg(3v_1 - 2v_2 \leq 3) \lor (2v_2 - v_3 > 2) \lor (3v_1 - v_3 \leq 6)
  \]
- Prunes up to \(2^{N-3}\) assignments
  \(\iff\) the smaller the conflict set, the better.
Learning: summary

- Very efficient with DPLL
- Never applied to OBDDs
- Completely ineffective with semantic tableaux
- May cause memory blowup
- $\mathcal{L}$-SOLVE must be able to detect conflict sets.
Forward Checking [2]

- **Idea:** if $\mu \land l \land l'$ inconsistent, then $\mu \land l \models \neg l'$
- $\text{assign}(\varphi, l)$ substituted with $\text{fc_assign}(\varphi, \mu \land l)$:
  - $\text{fc_assign}(\varphi, \mu \land l)$ replaces $cl \lor l'$ with $cl$ if $\mathcal{L}\text{-SOLVE}(\mu \land l \land l')$ returns false, for every $l'$
- can significantly prune search
- significant overhead: many possibly redundant calls to $\mathcal{L}\text{-SOLVE}$
Proposition Let \( C \) be a non-boolean atom occurring only positively [resp. negatively] in \( \varphi \). Let \( \mathcal{M} \) be a complete set of assignments for \( \varphi \), and let

\[
\mathcal{M}' := \{ \mu_j / \neg C \mid \mu_j \in \mathcal{M} \} \quad [\text{resp.} \quad \{ \mu_j / C \mid \mu_j \in \mathcal{M} \}].
\]

Then \( \varphi \) is satisfiable if and only if there exist a satisfiable \( \eta' \in \mathcal{M}' \) s.t. \( \eta' \models_p \varphi \).
Triggers (cont.)

- If we have non-boolean atoms occurring only positively [negatively] in $\varphi$, we can drop any negative [positive] occurrence of them from the assignment to be checked by $\mathcal{L}$-SOLVE.

- Particularly useful when we deal with equality atoms (e.g., $(v_1 - v_2 = 3.2)$), as handling negative equalities like $(v_1 - v_2 \neq 3.2)$ forces splitting: $(v_1 - v_2 > 3.2) \lor (v_1 - v_2 < 3.2)$. 
Application Fields

- Modal Logics
- Description Logics
- Temporal Logics
- **Boolean+Mathematical reasoning** (Temporal reasoning, Resource Planning, Verification of Timed Systems, Verification of systems with arithmetical operators, verification of hybrid systems)
- QBF
- ...

QBF
Case study: Modal Logic(s)
Satisfiability in Modal logics

– Propositional logics enhanced with modal operators $\square_i$, $K_i$, etc.
– Used to represent complex concepts like knowledge, necessity/possibility, etc.
– Based on Kripke’s possible worlds semantics [40]
– Very hard to decide [33, 32] (typically PSPACE-complete or worse)
– Strictly related to Description Logics [45] (ex: $K(m) \iff \mathcal{ALC}$)
– Various fields of application: AI, formal verification, knowledge bases, etc.
Syntax

Given a non-empty set of primitive propositions 
\( \mathcal{A} = \{ A_1, A_2, \ldots \} \) and a set of \( m \) modal operators 
\( \mathcal{B} = \{ \Box_1, \ldots, \Box_m \} \), the modal language \( \mathcal{L} \) is the least set
of formulas containing \( \mathcal{A} \), closed under the set of
propositional connectives \( \{ \neg, \land, \lor, \rightarrow, \leftrightarrow \} \) and the set of
modal operators in \( \mathcal{B} \).

- \( \text{depth}(\varphi) \) is the maximum number of nested modal
  operators in \( \varphi \).
- “\( \Box_i \varphi \)” can be interpreted as “Agent \( i \) knows \( \varphi \)”
A Kripke structure for $\mathcal{L}$ is a tuple $M = \langle \mathcal{U}, \pi, \mathcal{R}_1, \ldots, \mathcal{R}_m \rangle$, where

- $\mathcal{U}$ is a set of states $u_1, u_2, \ldots$.
- $\pi$ is a function $\pi : \mathcal{A} \times \mathcal{U} \rightarrow \{\top, \bot\}$,
- each $\mathcal{R}_r$ is a binary relation on the states of $\mathcal{U}$. 
Given $M, u$ s.t. $u \in \mathcal{U}$, $M, u \models \varphi$ is defined as follows:

\[
\begin{align*}
M, u \models A_i, & \quad A_i \in \mathcal{A} \iff \pi(A_i, u) = \top; \\
M, u \models \neg \varphi_1 \iff M, u \not\models \varphi_1; \\
M, u \models \varphi_1 \land \varphi_2 \iff M, u \models \varphi_1 \text{ and } M, u \models \varphi_2; \\
M, u \models \varphi_1 \lor \varphi_2 \iff M, u \models \varphi_1 \text{ or } M, u \models \varphi_2. \\
\ldots \\
M, u \models \Box_r \varphi_1, & \quad \Box_r \in \mathcal{B} \iff M, v \models \varphi_1 \text{ for every } v \in \mathcal{U} \text{ s.t. } \mathcal{R}_r(u, v) \text{ holds in } M. \\
M, u \models \neg \Box_r \varphi_1, & \quad \Box_r \in \mathcal{B} \iff M, v \models \neg \varphi_1 \text{ for some } v \in \mathcal{U} \text{ s.t. } \mathcal{R}_r(u, v) \text{ holds in } M.
\end{align*}
\]
The (normal) modal logics vary with the properties of $\mathcal{R}_r$:

<table>
<thead>
<tr>
<th>Axiom</th>
<th>Property of $\mathcal{R}$</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>symmetric</td>
<td>$\forall u v \mathcal{R}(u, v) \implies \mathcal{R}(v, u)$</td>
</tr>
<tr>
<td>D</td>
<td>serial</td>
<td>$\forall u \exists v \mathcal{R}(u, v)$</td>
</tr>
<tr>
<td>T</td>
<td>reflexive</td>
<td>$\forall u \mathcal{R}(u, u)$</td>
</tr>
<tr>
<td>4</td>
<td>transitive</td>
<td>$\forall u v w \mathcal{R}(u, v) \land \mathcal{R}(v, w) \implies \mathcal{R}(u, w)$</td>
</tr>
<tr>
<td>5</td>
<td>euclidean</td>
<td>$\forall u v w \mathcal{R}(u, v) \land \mathcal{R}(u, w) \implies \mathcal{R}(v, w)$</td>
</tr>
<tr>
<td>Normal Modal Logic</td>
<td>Properties of $\mathcal{R}_r$</td>
<td></td>
</tr>
<tr>
<td>-------------------</td>
<td>--------------------------------</td>
<td></td>
</tr>
<tr>
<td>K</td>
<td>—</td>
<td></td>
</tr>
<tr>
<td>KB</td>
<td>symmetric</td>
<td></td>
</tr>
<tr>
<td>KD</td>
<td>serial</td>
<td></td>
</tr>
<tr>
<td>KT = KDT (T)</td>
<td>reflexive</td>
<td></td>
</tr>
<tr>
<td>K4</td>
<td>transitive</td>
<td></td>
</tr>
<tr>
<td>K5</td>
<td>euclidean</td>
<td></td>
</tr>
<tr>
<td>KBD</td>
<td>symmetric and serial</td>
<td></td>
</tr>
<tr>
<td>KBT = KBDT (B)</td>
<td>symmetric and reflexive</td>
<td></td>
</tr>
<tr>
<td>KB4 = KB5 = KB45</td>
<td>symmetric and transitive</td>
<td></td>
</tr>
<tr>
<td>KD4</td>
<td>serial and transitive</td>
<td></td>
</tr>
<tr>
<td>KD5</td>
<td>serial and euclidean</td>
<td></td>
</tr>
<tr>
<td>KT4 = KDT4 (S4)</td>
<td>reflexive and transitive</td>
<td></td>
</tr>
<tr>
<td>KT5 = KBD4 = KBD5 = KBT4 = KBT5 = KDT5 = KT45 = KBD45 = KBT45 = KDT45 = KBDT4 = KBDT5 = KBDT45 (S5)</td>
<td>reflexive, transitive and symmetric (equivalence)</td>
<td></td>
</tr>
<tr>
<td>K45</td>
<td>transitive and euclidean</td>
<td></td>
</tr>
<tr>
<td>KD45</td>
<td>serial, transitive and euclidean</td>
<td></td>
</tr>
</tbody>
</table>
Axiomatic framework

– Basic Axioms:

1. \( \alpha \rightarrow (\beta \rightarrow \alpha) \),

2. \( (\alpha \rightarrow (\beta \rightarrow \gamma)) \rightarrow ((\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \gamma)) \),

3. \( (\neg \alpha \rightarrow \beta) \rightarrow ((\neg \alpha \rightarrow \neg \beta) \rightarrow \alpha) \),

4. \( \Box_r \alpha \rightarrow (\Box_r (\alpha \rightarrow \beta) \rightarrow \Box_r \beta) \)

– Specific Axioms:

5. \( \alpha \rightarrow \Box_r \neg \Box_r \neg \alpha \),

6. \( \Box_r \alpha \rightarrow \neg \Box_r \neg \alpha \),

7. \( \Box_r \alpha \rightarrow \alpha \),

8. \( \Box_r \alpha \rightarrow \Box_r \Box_r \alpha \),

9. \( \neg \Box_r \alpha \rightarrow \Box_r \neg \Box_r \alpha \).
Axiomatic framework (cont.)

– Inference rules:
  \[
  \frac{\alpha \quad \alpha \rightarrow \beta}{\beta} \quad \text{(modus ponens)} \\
  \frac{\alpha}{\Box \alpha} \quad \text{(necessitation)}.
  \]

– Correctness & completeness:
  \[ \varphi \text{ is valid} \iff \varphi \text{ can be deduced} \]
Tableaux for modal $K(m) / \mathcal{ACL}$ [21]

- Rules = tableau rules + $K(m)$-specific rules

\[
\begin{align*}
\varphi_1 \land \varphi_2 & \quad \neg(\varphi_1 \lor \varphi_2) & \quad \neg(\varphi_1 \rightarrow \varphi_2) \\
\varphi_1 & \quad \neg \varphi_1 & \quad \neg \varphi \quad \neg \varphi \\
\varphi_2 & \quad \neg \varphi_2 & \quad \\neg \varphi_2 \quad \neg \varphi_2 \\
\varphi_1 \lor \varphi_2 & \quad \neg(\varphi_1 \land \varphi_2) & \quad \varphi_1 \rightarrow \varphi_2 \\
\varphi_1 & \quad \neg \varphi_1 \quad \neg \varphi_2 & \quad \varphi_1 \rightarrow \varphi_2 \\
\varphi_2 & \quad \neg \varphi_1 \quad \neg \varphi_2 & \quad \varphi_1 \rightarrow \varphi_2 \\
\varphi_1 \leftrightarrow \varphi_2 & \quad \neg(\varphi_1 \leftrightarrow \varphi_2) & \quad \neg(\varphi_1 \leftrightarrow \varphi_2) \\
\varphi_1 & \quad \neg \varphi_1 \quad \neg \varphi_1 & \quad \varphi_1 \rightarrow \varphi_2 \\
\varphi_2 & \quad \neg \varphi_2 \quad \neg \varphi_2 & \quad \varphi_1 \rightarrow \varphi_2
\end{align*}
\]
**DPLL for $K(m)/\mathcal{ALC}$: K-SAT [29, 30]**

- **Rules** = DPLL rules + $K(m)$-specific rules

\[
\begin{align*}
\frac{\varphi_1 \land (l) \land \varphi_2}{(\varphi_1 \land \varphi_2)[l|\top]} \quad (Unit) \\
\frac{\varphi}{\varphi[l|\top] \quad \varphi[l|\bot]} \quad (split)
\end{align*}
\]

\[
\cup \left\{ \frac{\lozenge_r \alpha_1, \ldots, \lozenge_r \alpha_N, \neg \lozenge_r \beta_j}{\alpha_1, \ldots, \alpha_N, \neg \beta_j} \right\}
\]
The K-SAT algorithm [29, 30]

function K-SAT(\( \varphi \))
    return K-DPLL(\( \varphi, \top \));

function K-DPLL(\( \varphi, \mu \))
    if \( \varphi = \top \) /* base */
        then return K-SOLVE(\( \mu \));
    if \( \varphi = \bot \) /* backtrack */
        then return False;
    if \{a unit clause (l) occurs in \( \varphi \}\} /* unit */
        then return K-DPLL(assign(l, \( \varphi \), \( \mu \wedge l \)));
    if Likely-Unsatisfiable(\( \mu \)) /* early pruning */
        if not K-SOLVE(\( \mu \))
            then return False;
    \( l := \text{choose-literal}(\varphi) \); /* split */
    return K-DPLL(assign(l, \( \varphi \), \( \mu \wedge l \)) or
             K-DPLL(assign(\( \neg l \), \( \varphi \), \( \mu \wedge \neg l \));
The K-SAT algorithm (cont.)

function K-SOLVE\((\bigwedge_i \square_1 \alpha_{1i} \land \bigwedge_j \neg \square_1 \beta_{1j} \land \ldots \land \bigwedge_i \square_m \alpha_{mi} \land \bigwedge_j \neg \square_m \beta_{mj} \land \gamma)\)
  for each box index \(r\) do
    if not \(K\text{-SOLVE}_{\text{restr}}(\bigwedge_i \square_r \alpha_{ri} \land \bigwedge_j \neg \square_r \beta_{rj})\)
      then return \(False\);
  return \(True\);

function K-SOLVE\(_{\text{restr}}(\bigwedge_i \square_r \alpha_{ri} \land \bigwedge_j \neg \square_r \beta_{rj})\)
  for each conjunct \(\neg \square_r \beta_{rj}\) do
    if not \(K\text{-SAT}(\bigwedge_i \alpha_{ri} \land \neg \beta_{rj})\)
      then return \(False\);
  return \(True\);
K-SAT: Example

\[ \varphi = \{ \neg \Box_1 (\neg A_3 \lor \neg A_1 \lor A_2) \lor A_1 \lor A_5 \} \land \\
\{ \neg A_2 \lor \neg A_5 \lor \Box_2 (\neg A_2 \lor \neg A_4 \lor \neg A_3) \} \land \\
\{ A_1 \lor \Box_2 (\neg A_4 \lor A_5 \lor A_2) \lor A_2 \} \land \\
\{ \neg \Box_2 (A_4 \lor \neg A_3 \lor A_1) \lor \neg \Box_1 (A_4 \lor \neg A_2 \lor A_3) \lor \neg A_5 \} \land \\
\{ \neg A_3 \lor A_1 \lor \Box_2 (\neg A_4 \lor A_5 \lor A_2) \} \land \\
\{ \Box_1 (\neg A_5 \lor A_4 \lor A_3) \lor \Box_1 (\neg A_1 \lor A_4 \lor A_3) \lor \neg A_1 \} \land \\
\{ A_1 \lor \Box_1 (\neg A_2 \lor A_1 \lor A_4) \lor A_2 \} \]

\[ \Downarrow \quad \text{K-SOLVE()} \]

\[ \mu = \Box_1 (\neg A_5 \lor A_4 \lor A_3) \land \Box_1 (\neg A_2 \lor A_1 \lor A_4) \land \Box_2 (\neg A_4 \lor A_5 \lor A_2) \land \neg A_2. \]
\[ \mu = \square_1 (\neg A_5 \lor A_4 \lor A_3) \land \square_1 (\neg A_2 \lor A_1 \lor A_4) \land \neg \square_1 (\neg A_3 \lor \neg A_1 \lor A_2) \land \neg \square_1 (A_4 \lor \neg A_2 \lor A_3) \land \square_2 (\neg A_4 \lor A_5 \lor A_2) \land \neg A_2. \]

\[ \Downarrow \quad \text{K-SOLVE}_{\text{restr}}() \]

\[ \mu^1 = \square_1 (\neg A_5 \lor A_4 \lor A_3) \land \square_1 (\neg A_2 \lor A_1 \lor A_4) \land \neg \square_1 (\neg A_3 \lor \neg A_1 \lor A_2) \land \neg \square_1 (A_4 \lor \neg A_2 \lor A_3) \land \square_2 (\neg A_4 \lor A_5 \lor A_2) \land \neg A_2. \]

\[ \Downarrow \quad \text{K-SAT}() \]

\[ \varphi^{11} = (\neg A_5 \lor A_4 \lor A_3) \land (\neg A_2 \lor A_1 \lor A_4) \land A_3 \land A_1 \land \neg A_2, \]

\[ \varphi^{12} = (\neg A_5 \lor A_4 \lor A_3) \land (\neg A_2 \lor A_1 \lor A_4) \land \neg A_4 \land A_2 \land \neg A_3. \]
K-SAT: Example (cont.)

\[
\varphi^{11} = (\neg A_5 \lor A_4 \lor A_3) \land (\neg A_2 \lor A_1 \lor A_4) \land A_3 \land A_1 \land \neg A_2,
\]

\[
\varphi^{12} = (\neg A_5 \lor A_4 \lor A_3) \land (\neg A_2 \lor A_1 \lor A_4) \land \neg A_4 \land A_2 \land \neg A_3
\]

\[\downarrow \quad \text{K-SOLVE()}
\]

\[
\mu^{11} = A_3 \land A_1 \land \neg A_2
\]

\[
\mu^{12} = \neg A_4 \land A_2 \land \neg A_3 \land \neg A_5 \land A_1
\]

\[\downarrow
\]

Satisfiable
Example

Resulting Kripke Model:

\[ \square_2 (\neg A_4 \lor A_5 \lor A_2 ) \]
\[ \neg A_2 \]
\[ R_1 \]

\[ \square_1 (\neg A_5 \lor A_4 \lor A_3 ) \]
\[ \square_1 (\neg A_2 \lor A_1 \lor A_4 ) \]
\[ \neg \square_1 (\neg A_3 \lor A_1 \lor A_2 ) \]
\[ \neg \square_1 (A_4 \lor \neg A_2 \lor A_3 ) \]

A_3, A_1, \neg A_2

\[ \neg A_4, A_2, \neg A_3, \neg A_5, A_1 \]
Search in modal logic:

Two alternating orthogonal components of search:

- **Modal search: model spanning**
  - jumping among states
  - conjunctive branching
  - up to linearly many successors

- **Propositional search: local search**
  - reasoning within the single states
  - disjunctive branching
  - up to exponentially many successors
Some Systems

- **Kris** [7], **CRACK** [11],
  - Logics: $\mathcal{ALC}$ & many description logics
  - Boolean reasoning technique: semantic tableau
  - Optimizations: preprocessing

- **K-SAT** [29, 24]
  - Logics: $K(m), \mathcal{ALC}$
  - Boolean reasoning technique: DPLL
  - Optimizations: preprocessing, early pruning
Some Systems (cont.)

- **FaCT & DLP** [35]
  - Logics: $\mathcal{ALC}$ & many description logics
  - Boolean reasoning technique: DPLL-like
  - Optimizations: preprocessing, memoizing, backjumping + optimizations for description logics

- **ESAT & *SAT** [25]
  - Logics: non-normal modal logics, $K(m)$, $\mathcal{ALC}$
  - Boolean reasoning technique: DPLL
  - Optimizations: preprocessing, early pruning, memoizing, backjumping, learning
Some empirical results [24]

Left: KRIS, TA, K-SAT (LISP), K-SAT (C) median CPU time, 100 samples/point.
Right: K-SAT (LISP), K-SAT (C) median number of consistency checks, 100 samples/point.
Background: satisfiability percentage.
Some empirical results (cont.)

K-SAT (up) versus TA (down) CPU times.
### Some empirical results [36]

#### Formulas of Tableau’98 competition [34]

<table>
<thead>
<tr>
<th></th>
<th>branch</th>
<th>d4</th>
<th>dum</th>
<th>grz</th>
<th>lin</th>
<th>path</th>
<th>ph</th>
<th>poly</th>
<th>t4p</th>
</tr>
</thead>
<tbody>
<tr>
<td>K</td>
<td>p</td>
<td>n</td>
<td>p</td>
<td>n</td>
<td>p</td>
<td>n</td>
<td>p</td>
<td>n</td>
<td>p</td>
</tr>
<tr>
<td>leanK 2.0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>≥21</td>
<td>≥21</td>
</tr>
<tr>
<td>□KE</td>
<td>13</td>
<td>3</td>
<td>13</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>≥21</td>
</tr>
<tr>
<td>LWB 1.0</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>6</td>
<td>13</td>
<td>19</td>
<td>7</td>
<td>13</td>
<td>11</td>
</tr>
<tr>
<td>TA</td>
<td>9</td>
<td>9</td>
<td>≥21</td>
<td>18</td>
<td>≥21</td>
<td>≥21</td>
<td>≥21</td>
<td>≥21</td>
<td>≥21</td>
</tr>
<tr>
<td>*SAT 1.2</td>
<td>≥21</td>
<td>12</td>
<td>≥21</td>
<td>≥21</td>
<td>≥21</td>
<td>≥21</td>
<td>≥21</td>
<td>≥21</td>
<td>≥21</td>
</tr>
<tr>
<td>Crack 1.0</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>≥21</td>
<td>1</td>
<td>≥21</td>
<td>5</td>
</tr>
<tr>
<td>Kris</td>
<td>3</td>
<td>3</td>
<td>8</td>
<td>6</td>
<td>15</td>
<td>≥21</td>
<td>13</td>
<td>≥21</td>
<td>6</td>
</tr>
<tr>
<td>Fact 1.2</td>
<td>6</td>
<td>4</td>
<td>≥21</td>
<td>8</td>
<td>≥21</td>
<td>≥21</td>
<td>≥21</td>
<td>≥21</td>
<td>7</td>
</tr>
<tr>
<td>DLP 3.1</td>
<td>19</td>
<td>13</td>
<td>≥21</td>
<td>≥21</td>
<td>≥21</td>
<td>≥21</td>
<td>≥21</td>
<td>≥21</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>45</td>
<td>branch</td>
<td>dum</td>
<td>grz</td>
<td>md</td>
<td>path</td>
<td>ph</td>
<td>poly</td>
<td>t4p</td>
</tr>
<tr>
<td>------</td>
<td>--------</td>
<td>--------</td>
<td>-------</td>
<td>-------</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>------</td>
</tr>
<tr>
<td>KT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TA</td>
<td>17</td>
<td>6</td>
<td>13</td>
<td>9</td>
<td>17</td>
<td>9</td>
<td>21</td>
<td>21</td>
<td>16</td>
</tr>
<tr>
<td>Kris</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>14</td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>FaCT 1.2</td>
<td>≥21 ≥21</td>
<td>6</td>
<td>4</td>
<td>≥21</td>
<td>≥21</td>
<td>≥21</td>
<td>≥21</td>
<td>≥21</td>
<td>7</td>
</tr>
<tr>
<td>DLP 3.1</td>
<td>≥21 ≥21</td>
<td>19</td>
<td>12</td>
<td>≥21</td>
<td>≥21</td>
<td>≥21</td>
<td>≥21</td>
<td>≥21</td>
<td>12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>45</th>
<th>branch</th>
<th>dum</th>
<th>grz</th>
<th>md</th>
<th>path</th>
<th>ph</th>
<th>poly</th>
<th>t4p</th>
</tr>
</thead>
<tbody>
<tr>
<td>S4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KT4</td>
<td>1</td>
<td>6</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>17</td>
<td>5</td>
<td>8</td>
<td>≥21</td>
</tr>
<tr>
<td>leanS4 2.0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>□KE</td>
<td>8</td>
<td>0</td>
<td>≥21</td>
<td>≥21</td>
<td>0</td>
<td>≥21</td>
<td>6</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>LWB 1.0</td>
<td>3</td>
<td>5</td>
<td>11</td>
<td>7</td>
<td>9</td>
<td>≥21</td>
<td>8</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>TA</td>
<td>9</td>
<td>0</td>
<td>21</td>
<td>4</td>
<td>14</td>
<td>0</td>
<td>6</td>
<td>21</td>
<td>9</td>
</tr>
<tr>
<td>FaCT 1.2</td>
<td>≥21 ≥21</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>≥21</td>
<td>5</td>
<td>4</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>DLP 3.1</td>
<td>≥21 ≥21</td>
<td>18</td>
<td>12</td>
<td>≥21</td>
<td>≥21</td>
<td>10</td>
<td>≥21</td>
<td>3</td>
<td>21</td>
</tr>
</tbody>
</table>
SAT techniques for modal logics: summary

- SAT techniques have been successfully applied to modal/description logics
- Many optimizations applicable.
- Currently at the State-of-the-art.
Case Study: Mathematical Reasoning
MATH-SAT

– Boolean combinations of mathematical propositions on the reals or integers.
– Typically NP-complete
– Various fields of application: temporal reasoning, scheduling, formal verification, resource planning, etc.
Syntax

Let $\mathcal{D}$ be the domain of either reals $\mathbb{R}$ or integers $\mathbb{Z}$ with its set $\mathcal{OP}_D$ of arithmetical operators.

Given a non-empty set of primitive propositions $\mathcal{A} = \{A_1, A_2, \ldots\}$ and a set $\mathcal{E}_D$ of (linear) mathematical expressions over $\mathcal{D}$, the mathematical language $\mathcal{L}$ is the least set of formulas containing $\mathcal{A}$ and $\mathcal{E}_D$ closed under the set of propositional connectives $\{\neg, \land, \lor, \rightarrow, \leftrightarrow\}$. 
Syntax: math-terms and math-formulas

- a constant $c_i \in \mathbb{R}[Z]$ is a math-term;
- a variable $v_i$ over $\mathbb{R}[Z]$ is a math-term;
- $c_i \cdot v_j$ is a math-term, $c_i \in \mathbb{R}$ and $v_j$ being a constant and a variable over $\mathbb{R}[Z]$;
- if $t_1$ and $t_2$ are math-terms, then $-t_1$ and $(t_1 \otimes t_2)$ are math-terms, $\otimes \in \{+, -, \}$.
- a boolean proposition $A_i$ over $\mathbb{B} := \{\bot, \top\}$ is a math-formula;
- if $t_1, t_2$ are math-terms, then $(t_1 \oslash t_2)$ is a math-formula, $\oslash \in \{=, \neq, >, <, \geq, \leq\}$;
- if $\varphi_1, \varphi_2$ are math-formulas, then $\neg \varphi_1$, $(\varphi_1 \land \varphi_2)$, $(\varphi_1 \lor \varphi_2)$, $(\varphi_1 \rightarrow \varphi_2)$ and $(\varphi_1 \leftrightarrow \varphi_2)$, are math-formulas.
Interpretations

**Interpretation:** a map $\mathcal{I}$ assigning real [integer] and boolean values to math-terms and math-formulas respectively and preserving constants and operators:

- $\mathcal{I}(A_i) \in \{\top, \bot\}$, for every $A_i \in \mathcal{A}$;
- $\mathcal{I}(c_i) = c_i$, for every constant $c_i \in \mathbb{R}$;
- $\mathcal{I}(v_i) \in \mathbb{R}$, for every variable $v_i$ over $\mathbb{R}$;
- $\mathcal{I}(t_1 \otimes t_2) = \mathcal{I}(t_1) \otimes \mathcal{I}(t_2)$, for all math-terms $t_1, t_2$ and $\otimes \in \{+, -, \cdot\}$;
- $\mathcal{I}(t_1 \boxdot t_2) = \mathcal{I}(t_1) \boxdot \mathcal{I}(t_2)$, for all math-terms $t_1, t_2$ and $\boxdot \in \{=, \neq, >, <, \geq, \leq\}$;
- $\mathcal{I}(\neg \varphi_1) = \neg \mathcal{I}(\varphi_1)$, for every math-formula $\varphi_1$;
- $\mathcal{I}(\varphi_1 \land \varphi_2) = \mathcal{I}(\varphi_1) \land \mathcal{I}(\varphi_2)$, for all math-formulas $\varphi_1, \varphi_2$. 
DPLL for math-formulas [55, 2, 4, 5]

```plaintext
function MATH-SAT(φ)  
    return MATH-DPLL(φ, ⊤);

function MATH-DPLL(φ, µ)  
    if φ = ⊤                          /* base */
    then return MATH-SOLVE(µ);
    if φ = ⊥                          /* backtrack */
    then return False;
    if {a unit clause (l) occurs in φ} /* unit */
    then return MATH-DPLL(assign(l, φ), µ ∧ l);
    if Likely- Unsatisfiable(µ)        /* early pruning */
    if not MATH-SOLVE(µ)
    then return False;
    l := choose-literal(φ);            /* split */
    return MATH-DPLL(assign(l, φ), µ ∧ l) or
        MATH-DPLL(assign(¬l, φ), µ ∧ ¬l);
```
**Math-SOLVE**: different algorithms for different kinds of math-atoms:

- **Difference expressions** \((x - y \leq 3)\): Belman-Ford minimal path algorithm with negative cycle detection
- **Equalities** \((x = y)\): equivalent class building and rewriting.
- **General linear expressions** \((3x - 4y + 2z \leq 5)\): linear programming techniques (Symplex, etc.)
- **Disequalities** \((x \neq y)\): postpone at the end. Expand \(((x < y) \lor (y < x))\) only if indispensable!
Some Systems

- **Tsat [2]**
  - Logics: disjunctions of difference expressions (positive math-atoms only)
  - Applications: temporal reasoning
  - Boolean reasoning technique: DPLL
  - Optimizations: preprocessing, static learning, forward checking

- **LPsat [55]**
  - Logics: MATH-SAT (positive math-atoms only)
  - Applications: resource planning
  - Boolean reasoning technique: DPLL
  - Optimizations: preprocessing, backjumping, learning, triggering
Some systems (cont.)

- **DDD** [42]
  - Logics: boolean + difference expressions
  - Applications: formal verification of timed systems
  - Boolean reasoning technique: OBDD
  - Optimizations: preprocessing, early pruning

- **Math-SAT** [4]
  - Logics: MATH-SAT
  - Applications: resource planning, formal verification of timed systems
  - Boolean reasoning technique: DPLL
  - Optimizations: preprocessing, enhanced early pruning, backjumping, learning, triggering
SAT techniques for modal logics: summary

- SAT techniques have been successfully applied to MATH-SAT
- Many optimizations applicable.
- Currently competitive with state-of-the-art applications for temporal reasoning, resource planning, formal verification of timed systems.
References


[14] 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 Verlag, january 2002.


[23] I. P. Gent, E. MacIntyre, P. Prosser, and T. Walsh. The constrainedness of


The papers (co)authored by Roberto Sebastiani are available at:

http://www.dit.unitn.it/~rseba/publist.html.