— 14th European School on Logic, Language and Computation —

Trento, August 2002

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

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], ...

Content

Part 1: PROPOSITIONAL SATISFIABILITY

Content (cont.)

Part 2: BEYOND SAT

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 φ_1 and φ_2 are formulas, then $\neg\varphi_1$, $\varphi_1 \wedge \varphi_2$, $\varphi_1 \vee \varphi_2$, $\varphi_1 \rightarrow \varphi_2$, $\varphi_1 \leftrightarrow \varphi_2$ 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 φ.
- a boolean formula can be represented as a tree or as a DAG

Basic notation & definitions (cont)

– Total truth assignment μ for φ:
$\mu : Atoms(\varphi) \longmapsto \{\top, \bot\}$.

– Partial Truth assignment μ for φ:
$\mu : \mathcal{A} \longmapsto \{\top, \bot\}$, $\mathcal{A} \subset Atoms(\varphi)$.

– Set and formula representation of an assignment:

- μ can be represented as a set of literals:
  EX: $\{\mu(A_1) := \top, \mu(A_2) := \bot\} \implies \{A_1, \neg A_2\}$
- μ can be represented as a formula:
  EX: $\{\mu(A_1) := \top, \mu(A_2) := \bot\} \implies A_1 \wedge \neg A_2$

Basic notation & definitions (cont)

– $\mu \models \varphi$ (μ satisfies φ):
  - $\mu \models A_i \iff \mu(A_i) = \top$
  - $\mu \models \neg\varphi \iff not\ \mu \models \varphi$
  - $\mu \models \varphi_1 \wedge \varphi_2 \iff \mu \models \varphi_1\ and\ \mu \models \varphi_2$
  - ...

– φ is satisfiable iff $\mu \models \varphi$ for some μ

– $\varphi_1 \models \varphi_2$ (φ_1 entails φ_2):
  $\varphi_1 \models \varphi_2$ iff for every μ $\mu \models \varphi_1 \Longrightarrow \mu \models \varphi_2$

– $\models \varphi$ (φ is valid):
  $\models \varphi$ iff for every μ $\mu \models \varphi$

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

Equivalence and equi-satisfiability

– φ_1 and φ_2 are equivalent iff, for every μ,
  $\mu \models \varphi_1$ iff $\mu \models \varphi_2$

– φ_1 and φ_2 are equi-satisfiable iff
  exists μ_1 s.t. $\mu_1 \models \varphi_1$ iff exists μ_2 s.t. $\mu_2 \models \varphi_2$

– φ_1, φ_2 equivalent

$$\Downarrow \quad \not\Uparrow$$

  φ_1, φ_2 equi-satisfiable

– EX: $\varphi_1 \vee \varphi_2$ and $(\varphi_1 \vee \neg A_3) \wedge (A_3 \vee \varphi_2)$, A_3 not in $\varphi_1 \vee \varphi_2$, are equi-satisfiable but not equivalent.

Complexity

- 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

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

Negative normal form (NNF)

– φ is in Negative normal form iff it is given only by applications of $\wedge, \vee$ to literals.

– every φ can be reduced into NNF:

1. substituting all $\rightarrow$'s and $\leftrightarrow$'s:

$$\varphi_1 \rightarrow \varphi_2 \quad \Longrightarrow \quad \neg\varphi_1 \vee \varphi_2$$

$$\varphi_1 \leftrightarrow \varphi_2 \quad \Longrightarrow \quad (\neg\varphi_1 \vee \varphi_2) \wedge (\varphi_1 \vee \neg\varphi_2)$$

2. pushing down negations recursively:

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

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

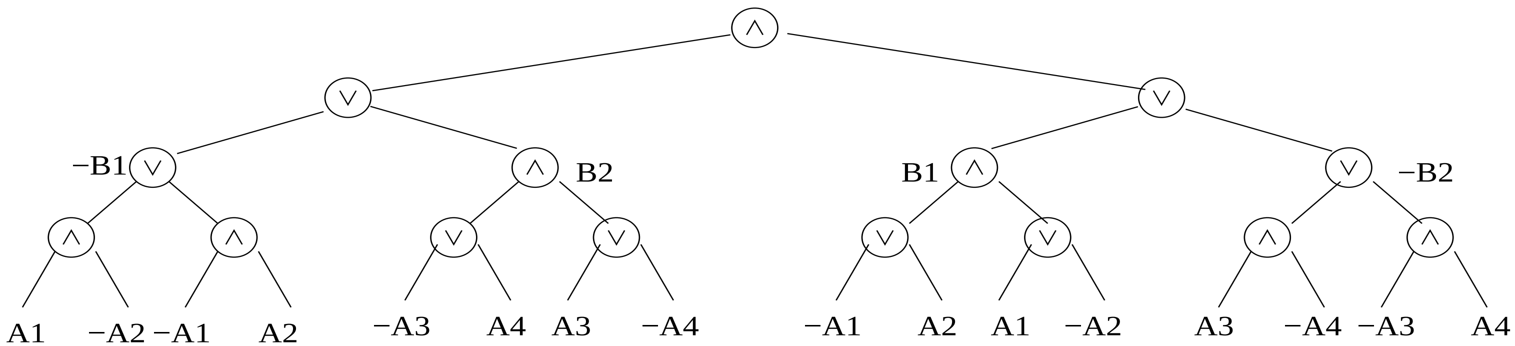
$$\neg\neg\varphi_1 \quad \Longrightarrow \quad \varphi_1$$

– The reduction is linear if a DAG representation is used.

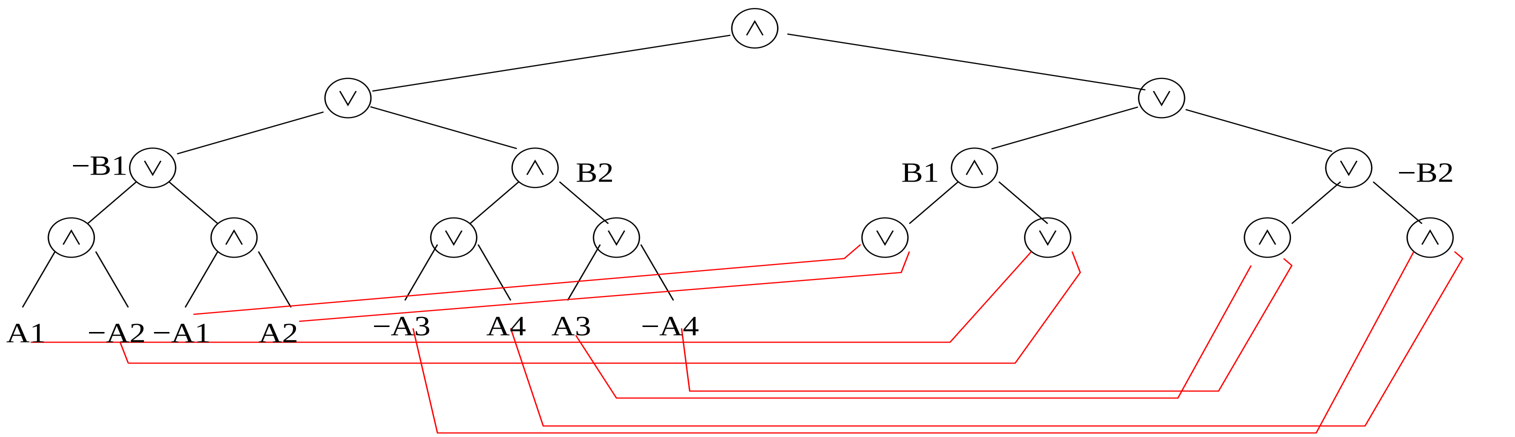
– Preserves the equivalence of formulas.

NNF: example

$$\underbrace{(A_1 \leftrightarrow A_2)}_{B_1} \leftrightarrow \underbrace{(A_3 \leftrightarrow A_4)}_{B_2}$$



Tree Representation



DAG Representation

Conjunctive Normal Form (CNF)

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

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

– the disjunctions of literals $\bigvee_{j_i=1}^{K_i} l_{j_i}$ are called clauses

– Easier to handle: list of lists of literals.

$\Longrightarrow$ no reasoning on the recursive structure of the formula

Classic CNF Conversion $CNF(\varphi)$

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

1. converting it into NNF;
2. applying recursively the DeMorgan's Rule:

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

– Worst-case exponential.

– $Atoms(CNF(\varphi)) = Atoms(\varphi)$.

– $CNF(\varphi)$ is equivalent to φ.

– Normal: if φ_1 equivalent to φ_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 φ can be reduced into CNF by, e.g., applying recursively bottom-up the rules:

$$\varphi \implies \varphi[(l_i \vee l_j)|B] \wedge CNF(B \leftrightarrow (l_i \vee l_j))$$

$$\varphi \implies \varphi[(l_i \wedge l_j)|B] \wedge CNF(B \leftrightarrow (l_i \wedge l_j))$$

$$\varphi \implies \varphi[(l_i \leftrightarrow l_j)|B] \wedge CNF(B \leftrightarrow (l_i \leftrightarrow l_j))$$

l_i, l_j being literals and B being a "new" variable.

– Worst-case linear.

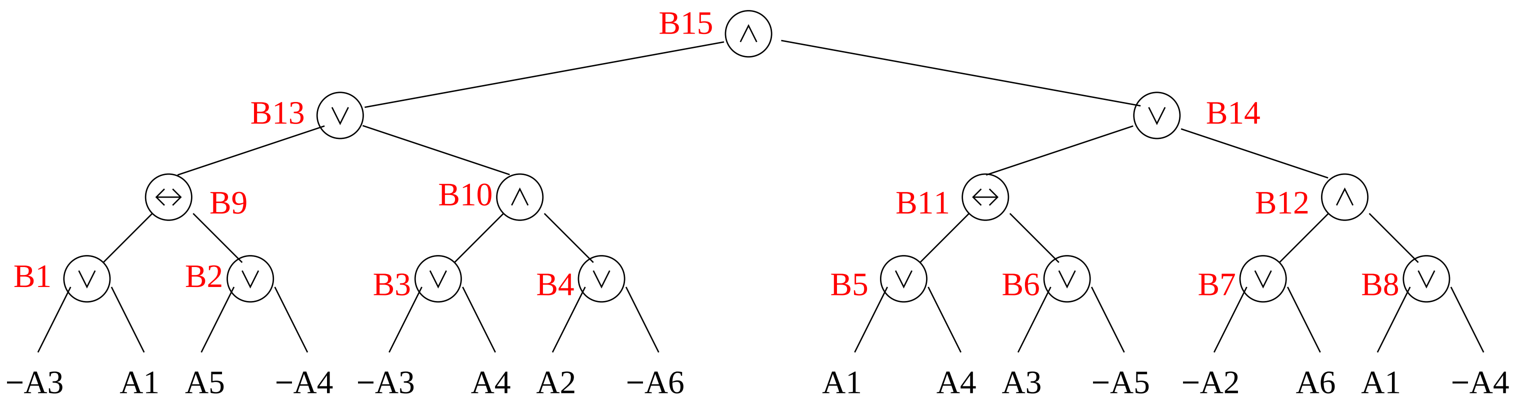
– $Atoms(CNF_{label}(\varphi)) \supseteq Atoms(\varphi)$.

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

– Non-normal.

– More used in practice.

Labeling CNF conversion CNF_{label} – example



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

Labeling CNF conversion CNF_{label} (improved)

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

$$\varphi \implies \varphi[(l_i \vee l_j)|B] \wedge CNF(B \rightarrow (l_i \vee l_j)) \; if \; (l_i \vee l_j) \; positive$$

$$\varphi \implies \varphi[(l_i \vee l_j)|B] \wedge CNF((l_i \vee l_j) \rightarrow B) \; if \; (l_i \vee l_j) \; negative$$

$$\varphi \implies \varphi[(l_i \wedge l_j)|B] \wedge CNF(B \rightarrow (l_i \wedge l_j)) \; if \; (l_i \wedge l_j) \; positive$$

$$\varphi \implies \varphi[(l_i \wedge l_j)|B] \wedge CNF((l_i \wedge l_j) \rightarrow B) \; if \; (l_i \wedge l_j) \; negative$$

$$\varphi \implies \varphi[(l_i \leftrightarrow l_j)|B] \wedge CNF(B \rightarrow (l_i \leftrightarrow l_j)) \; if \; (l_i \leftrightarrow l_j) \; positiv$$

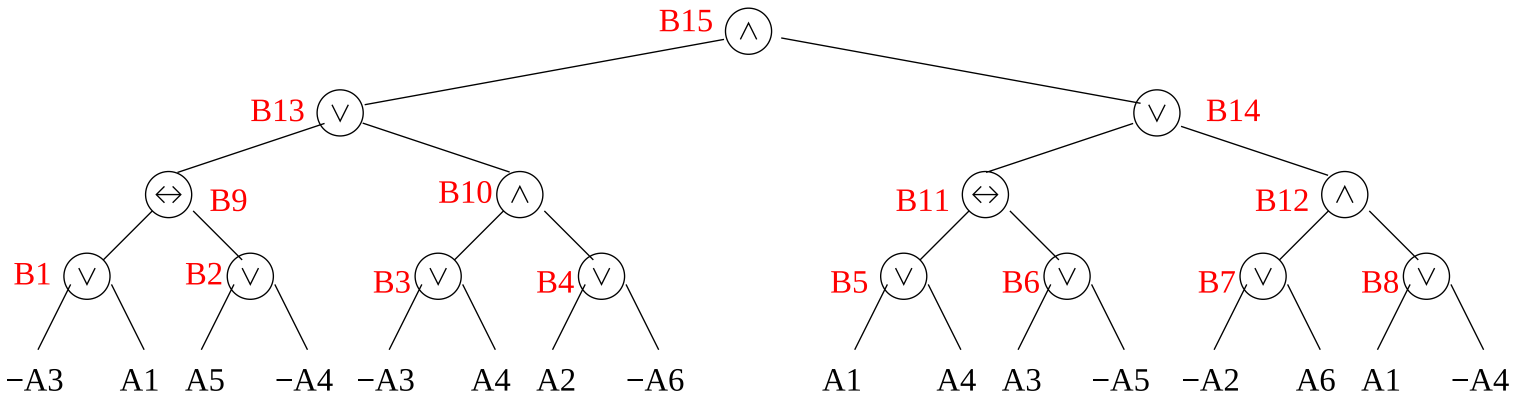
$$\varphi \implies \varphi[(l_i \leftrightarrow l_j)|B] \wedge CNF((l_i \leftrightarrow l_j) \rightarrow B) \; if \; (l_i \leftrightarrow l_j) \; negati$$

– Smaller in size:

$$CNF(B \rightarrow (l_i \vee l_j)) = (\neg B \vee l_i \vee l_j)$$

$$CNF(((l_i \vee l_j) \rightarrow B)) = (\neg l_i \vee B) \wedge (\neg l_j \vee B)$$

Labeling CNF conversion CNF_{label} – example



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

$$l_1 \vee l_2 \vee ... \vee l_{k-1} \vee l_k$$

$$\Downarrow$$

$$(l_1 \vee l_2 \vee B_1) \wedge$$
$$(\neg B_1 \vee l_3 \vee B_2) \wedge$$
$$...$$
$$(\neg B_{k-4} \vee l_{k-2} \vee B_{k-3}) \wedge$$
$$(\neg B_{k-3} \vee l_{k-1} \vee l_k)$$

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

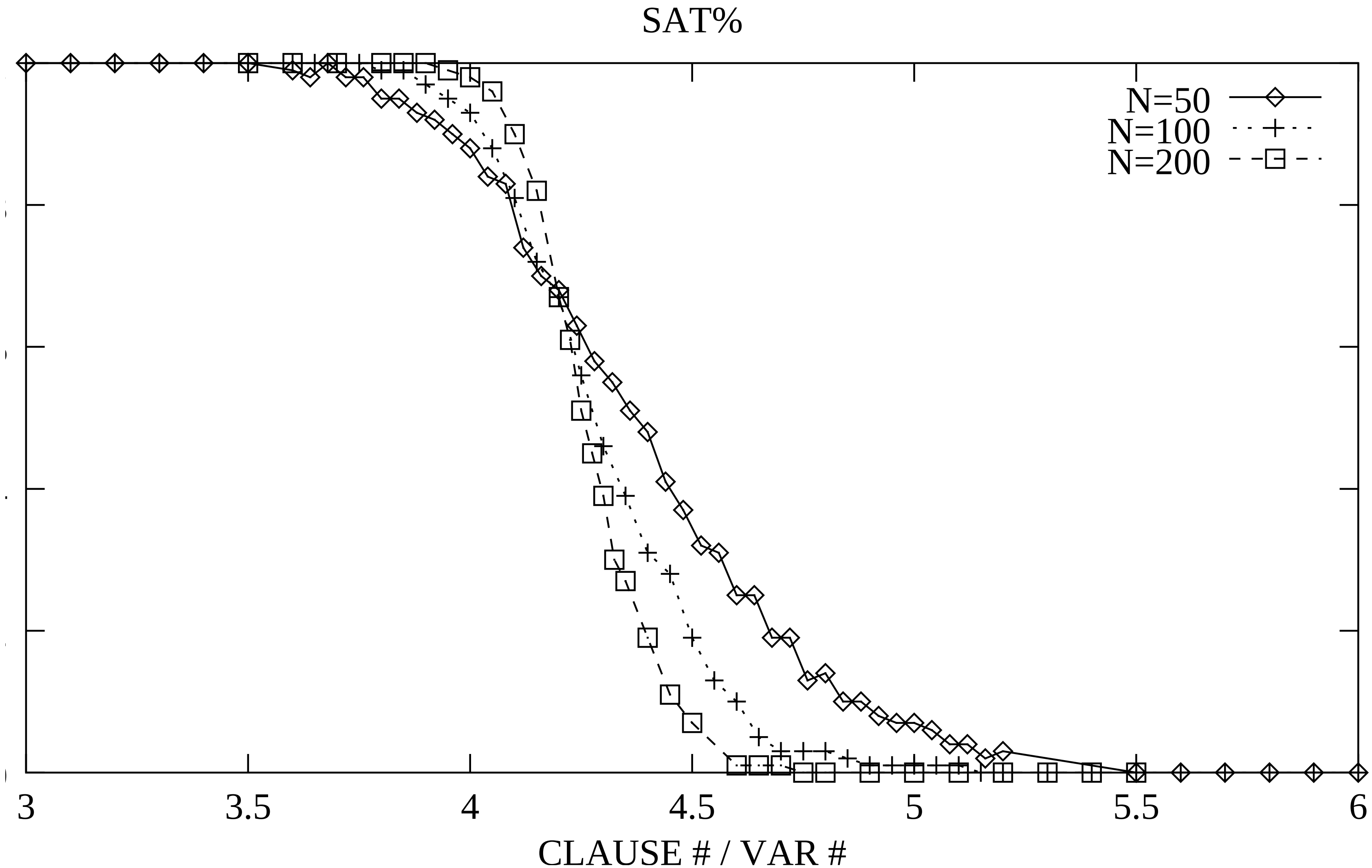
– plot

- 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]


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

The phase transition phenomenon: CPU times/step

Using search algorithms (DPLL):

– Increasing L/N we pass from easy problems, to very hard problems down to hard problems

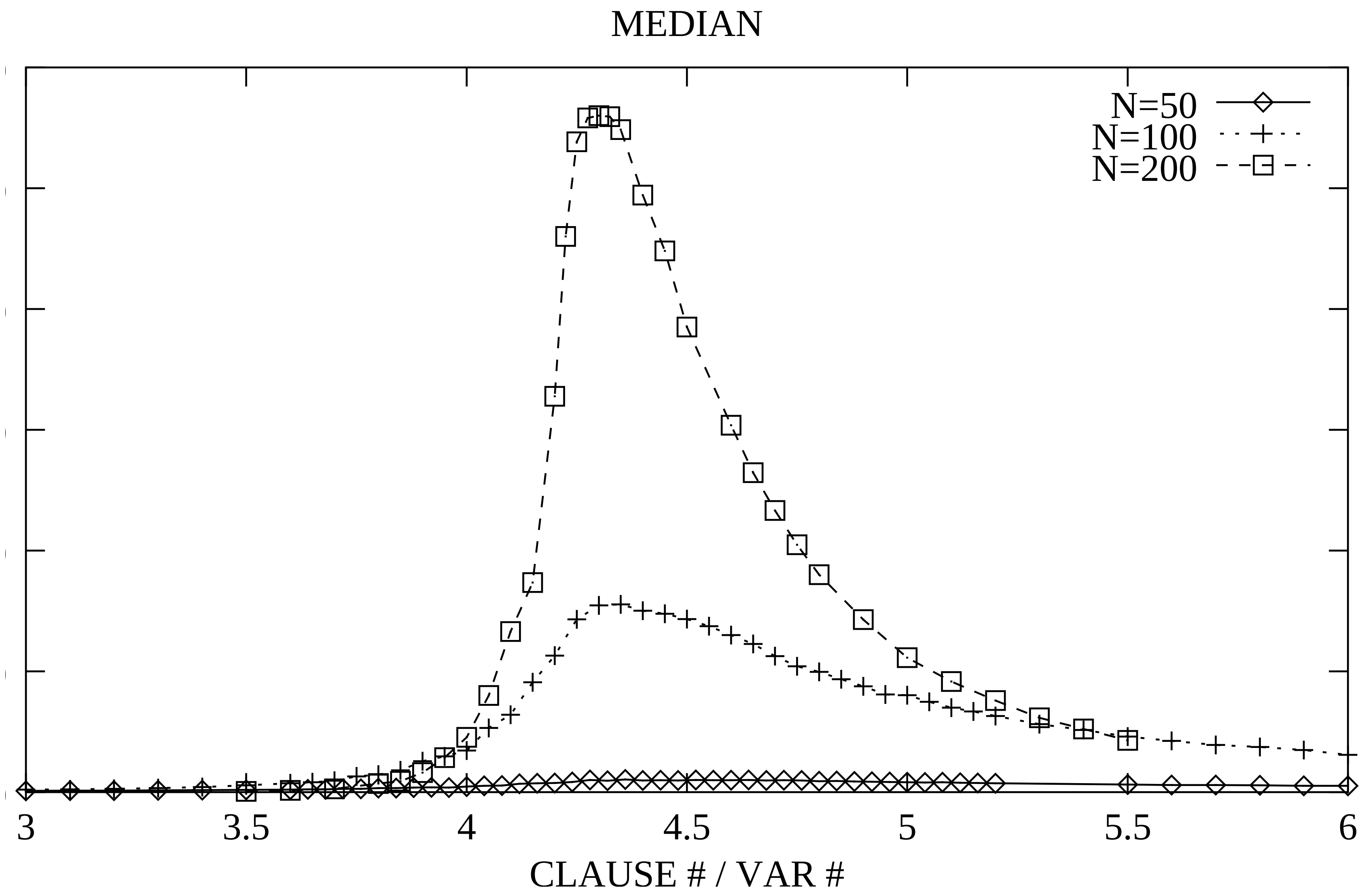
– the peak is centered in the 50% satisfiable point

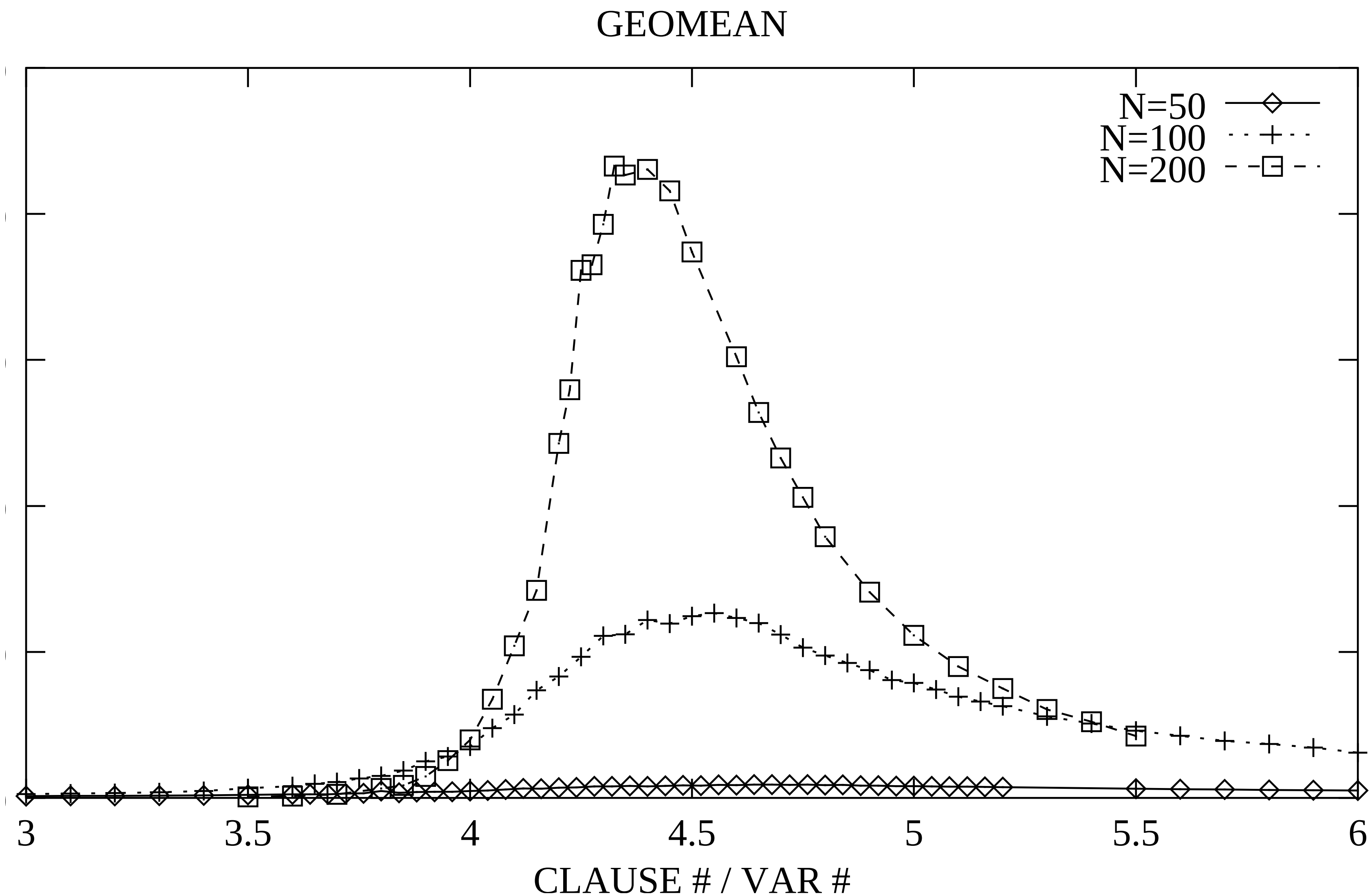
– the decay becomes steeper with N

– for $N \to \infty$, the plot converges to an impulse in the cross-over point ($L/N \approx 4.28$ for k=3)

– easy problems ($L/N \leq\approx 3.8$) increase polynomially with N, hard problems increase exponentially with N

– Increasing L/N, satisfiable problems get harder, unsatisfiable problems get easier.


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


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

Basic SAT techniques

Truth Tables

– Exhaustive evaluation of all subformulas:

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

– Requires polynomial space.

– Never used in practice.

Semantic tableaux [52]

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

Tableau elimination rules

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

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

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

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

Semantic Tableaux – example

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

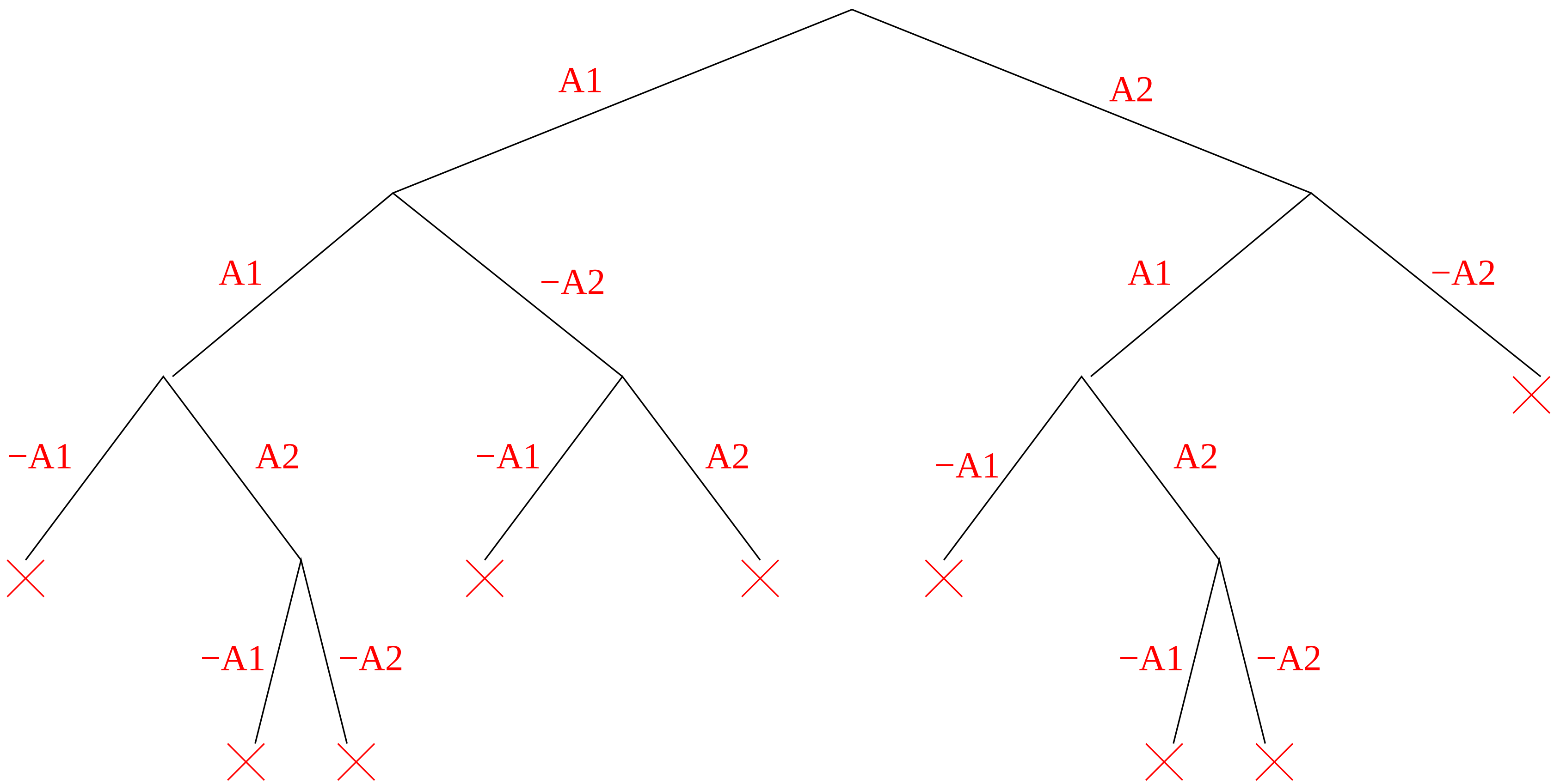


Tableau algorithm

function *Tableau*(Γ)

    if $A_i \in \Gamma$ **and** $\neg A_i \in \Gamma$ /* branch closed */

        then return *False*;

    if $(\varphi_1 \wedge \varphi_2) \in \Gamma$ /* $\wedge$-elimination */

        then return *Tableau*$(\Gamma \cup \{\varphi_1, \varphi_2\} \backslash \{(\varphi_1 \wedge \varphi_2)\})$;

    if $(\neg\neg\varphi_1) \in \Gamma$ /* $\neg\neg$-elimination */

        then return *Tableau*$(\Gamma \cup \{\varphi_1\} \backslash \{(\neg\neg\varphi_1)\})$;

    if $(\varphi_1 \vee \varphi_2) \in \Gamma$ /* $\vee$-elimination */

        then return *Tableau*$(\Gamma \cup \{\varphi_1\} \backslash \{(\varphi_1 \vee \varphi_2)\})$ **or**

                Tableau$(\Gamma \cup \{\varphi_2\} \backslash \{(\varphi_1 \vee \varphi_2)\})$;

    ...

    return *True*; /* branch expanded */

Semantic Tableaux – summary

– Handles all propositional formulas (CNF not required).

– Branches on disjunctions

– Intuitive, modular, easy to extend
$\Longrightarrow$ loved by logicians.

– Rather inefficient
$\Longrightarrow$ avoided by computer scientists.

– Requires polynomial space

DPLL [18, 17]

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

DPLL rules

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

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

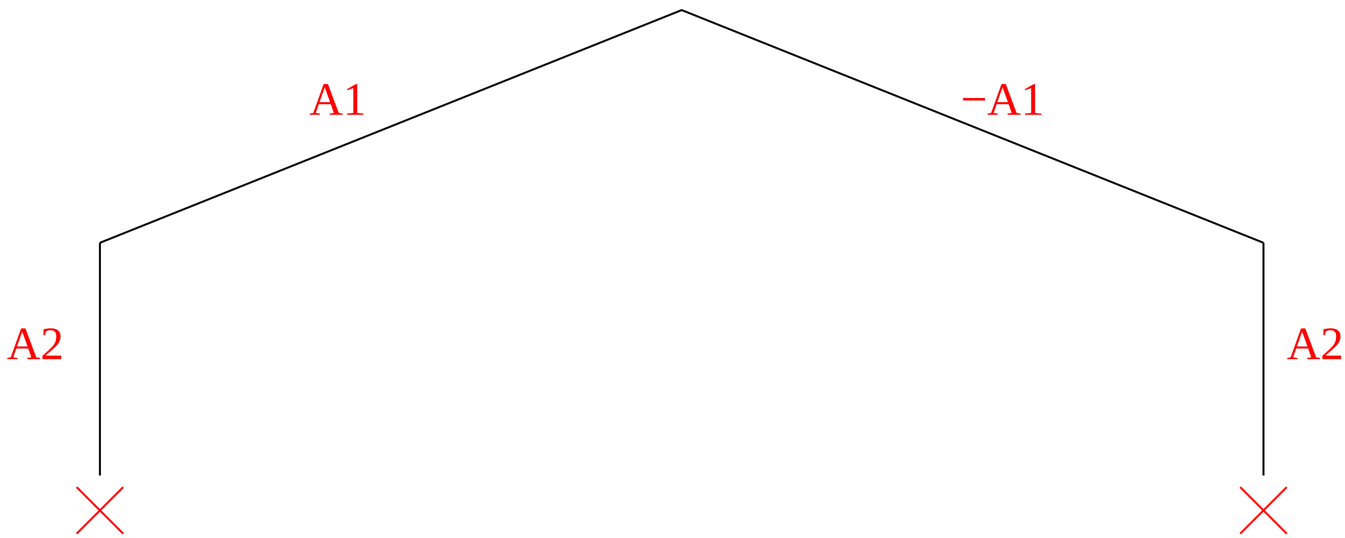
$$\frac{\varphi}{\varphi[l|\top] \quad \varphi[l|\bot]} \ (split)$$

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

Split applied if and only if the others cannot be applied.

DPLL – example

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



DPLL Algorithm

```
function DPLL(φ, μ)
    if φ = ⊤                                              /* base      */
        then return True;
    if φ = ⊥                                              /* backtrack */
        then return False;
    if {a unit clause (l) occurs in φ}                    /* unit      */
        then return DPLL(assign(l, φ), μ ∧ l);
    if {a literal l occurs pure in φ}                     /* pure      */
        then return DPLL(assign(l, φ), μ ∧ l);
    l := choose-literal(φ);                               /* split     */
    return  DPLL(assign(l, φ), μ ∧ l)  or
            DPLL(assign(¬l, φ), μ ∧ ¬l);
```

DPLL – summary

- Handles CNF formulas (non-CNF variant known [3, 28]).
- Branches on truth values
  $\Longrightarrow$all instances of an atom assigned simultaneously
- Postpones branching as much as possible.
- Mostly ignored by logicians.
- Probably the most efficient SAT algorithm
  $\Longrightarrow$ 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 $A_1, A_2, ..., A_n$ 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

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

OBDD - Examples

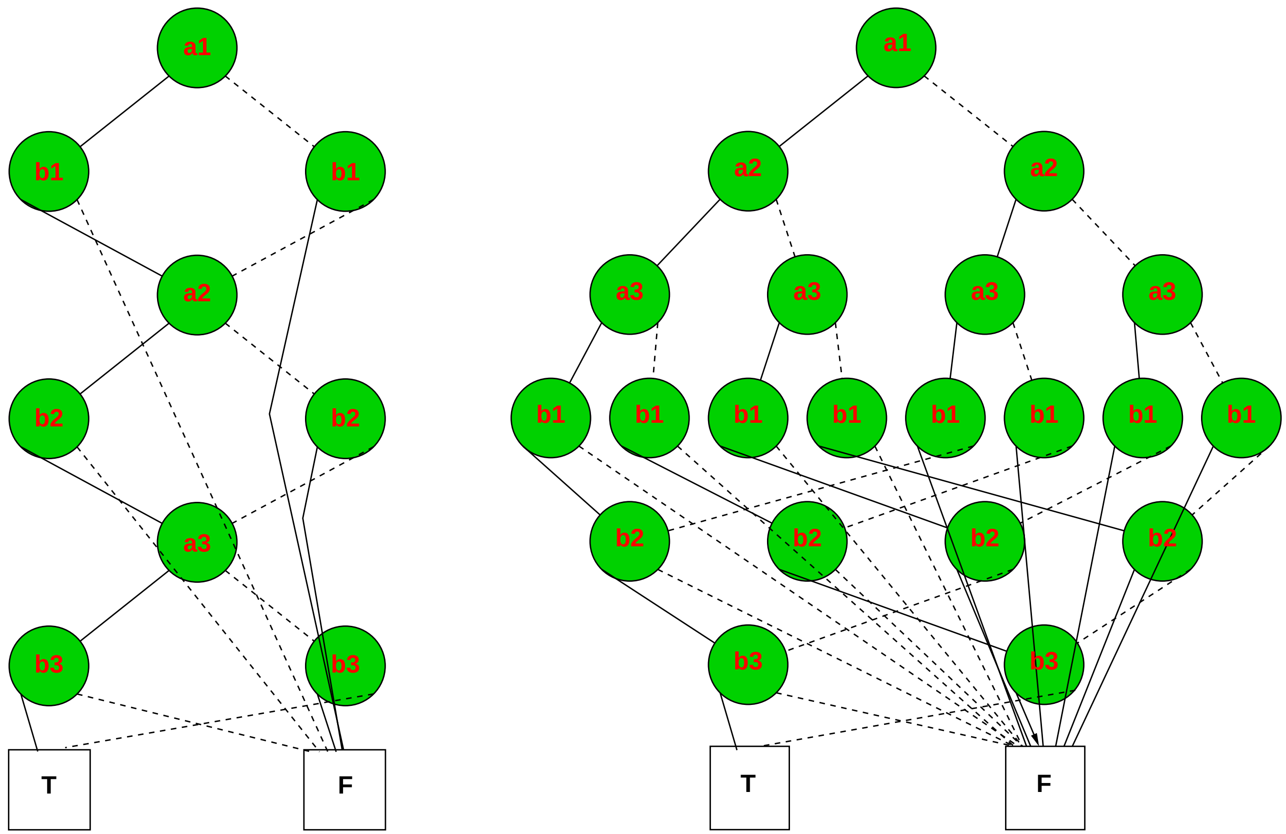


Figure 1: OBDDs of $(a_1 \leftrightarrow b_1) \wedge (a_2 \leftrightarrow b_2) \wedge (a_3 \leftrightarrow b_3)$ with different variable orderings

Incrementally building an OBDD

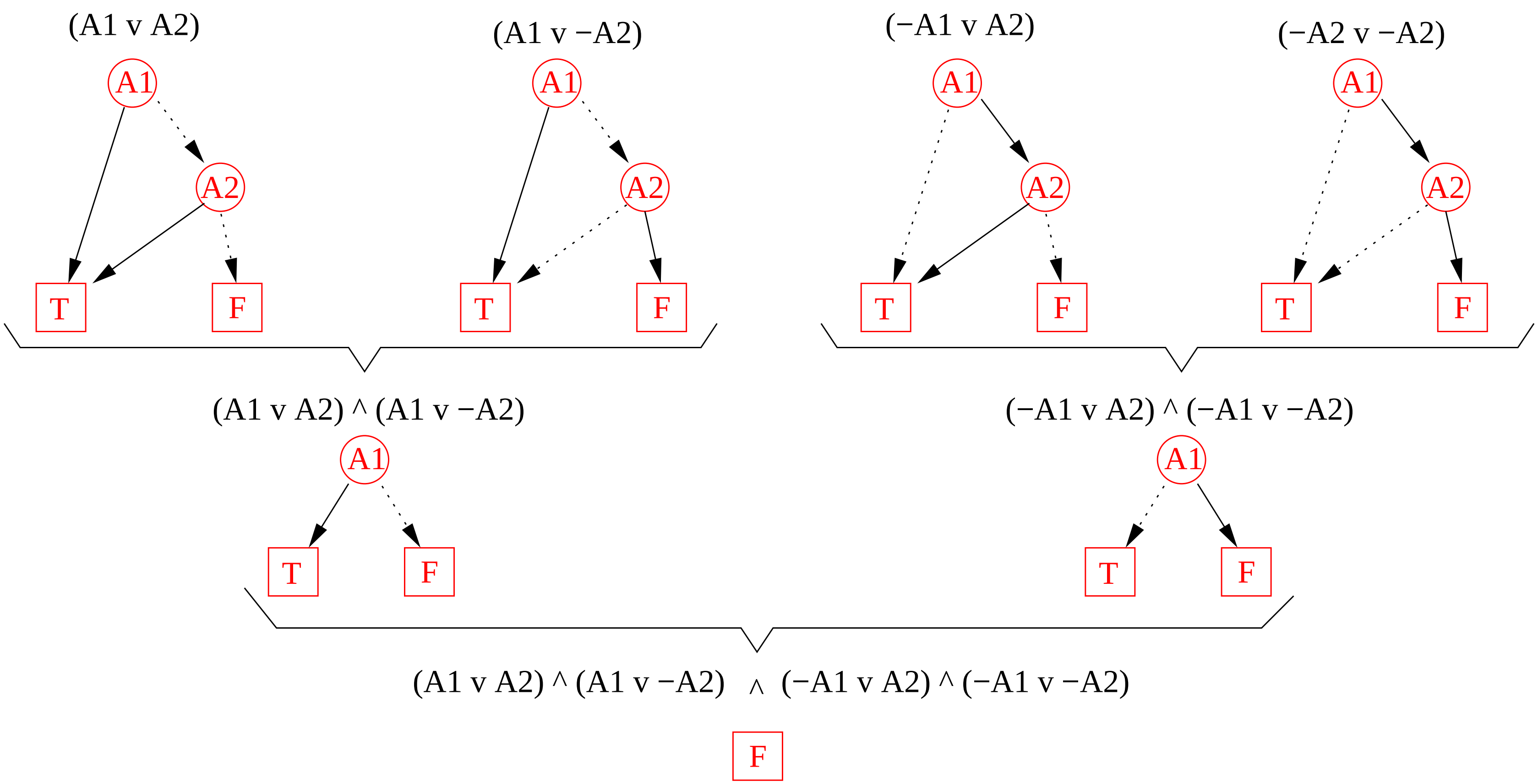
- $obdd\_build(\top, \{...\}) := 1$,
- $obdd\_build(\bot, \{...\}) := 0$,
- $obdd\_build((\varphi_1 \; op \; \varphi_2), \{A_1, ..., A_n\}) :=$

$$obdd\_merge(\quad op, \; obdd\_build(\varphi_1, \{A_1, ..., A_n\}), \; obdd\_build(\varphi_2, \{A_1, ..., A_n\}), \; \{A_1, ..., A_n\} \;)$$

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

OBBD incremental building – example

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



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!)
$\Longrightarrow$ 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

GSAT algorithm

function *GSAT(φ)*

  for $i := 1$ **to** Max-tries **do**

    μ := rand-assign(φ);

    for $j := 1$ **to** Max-flips **do**

      if ($score(\varphi, \mu) = 0$)

        then return True;

        else Best-flips := hill-climb(φ, μ);

          A_i := rand-pick(Best-flips);

          μ := flip(A_i, μ);

    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_DPLL(φ, μ)
    if φ = ⊤                                              /* base      */
        then return True;
    if φ = ⊥                                              /* backtrack */
        then return False;
    if {∃l s.t. equivalent_unit (l, φ)}                   /* unit      */
        then return NC_DPLL(assign(l, φ), μ ∧ l);
    if {∃l s.t. equivalent_pure(l, φ)}                    /* pure      */
        then return NC_DPLL(assign(l, φ), μ ∧ l);
    l := choose-literal(φ);                               /* split     */
    return  NC_DPLL(assign(l, φ), μ ∧ l)  or
            NC_DPLL(assign(¬l, φ), μ ∧ ¬l);
```

Non-CNF DPLL (cont.)

– $equivalent\_unit(l, \varphi)$:

$$
\begin{array}{lcl}
equivalent\_unit(l, l_1) & := & \top \quad if\ l = l_1 \\
 & & \bot \quad otherwise \\
equivalent\_unit(l, \varphi_1 \wedge \varphi_2) & := & equivalent\_unit(l, \varphi_1)\ or \\
 & & equivalent\_unit(l, \varphi_2) \\
equivalent\_unit(l, \varphi_1 \vee \varphi_2) & := & equivalent\_unit(l, \varphi_1)\ and \\
 & & equivalent\_unit(l, \varphi_2)
\end{array}
$$

Non-CNF DPLL (cont.)

– $equivalent\_pure(l, \varphi)$:

$$
\begin{array}{lcl}
equivalent\_pure(l, l_1) & := & \bot \quad if\ l = \neg l_1 \\
 & & \top \quad otherwise \\
equivalent\_pure(l, \varphi_1 \wedge \varphi_2) & := & equivalent\_pure(l, \varphi_1)\ and \\
 & & equivalent\_pure(l, \varphi_2) \\
equivalent\_pure(l, \varphi_1 \vee \varphi_2) & := & equivalent\_pure(l, \varphi_1)\ and \\
 & & equivalent\_pure(l, \varphi_2)
\end{array}
$$

Applying DPLL to $CNF_{label}(\varphi)$ [28, 26]

– $CNF(\varphi) = O(2^{|\varphi|})$
  $\Longrightarrow$inapplicable in most cases.
– $CNF_{label}(\varphi)$ introduces $K = O(|\varphi|)$ new variables
  $\Longrightarrow$ 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 $Choose\_literal(\varphi)$ to split first on original variables
  $\Longrightarrow$DPLL assigns the other variables deterministically.

Applying DPLL to $CNF_{label}(\varphi)$ (cont)

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

$$\varphi \implies \varphi[(l_i \vee l_j)|B] \wedge CNF(B \leftrightarrow (l_i \vee l_j))$$

... ...

then B is deterministicaly assigned by unit propagation if l_i and l_j are assigned.

– If the improved $CNF_{label}(\varphi)$ is used:

$$\varphi \quad \Longrightarrow \quad \varphi[(l_i \vee l_j)|B] \wedge CNF(B \rightarrow (l_i \vee l_j)) \; if \; (l_i \vee l_j) \; positive$$

$$\ldots \quad \ldots \quad \ldots$$

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 $\top$.

Non-CNF GSAT [46]

```
function NC_GSAT(φ)
    for i := 1 to Max-tries do
        μ := rand-assign(φ);
        for j := 1 to Max-flips do
            if (s(μ, φ) = 0)
                then return True;
                else Best-flips := hill-climb(φ, μ);
                    A_i := rand-pick(Best-flips);
                    μ := flip(A_i, μ);
        end
    end
    return "no satisfying assignment found".
```

Non-CNF GSAT (cont.)

φ	$s(\mu,\varphi)$	$s^-(\mu,\varphi)$
$\varphi\ literal$	$\begin{cases} 0 & if\ \mu \models \varphi \\ 1 & otherwise \end{cases}$	$\begin{cases} 1 & if\ \mu \models \varphi \\ 0 & otherwise \end{cases}$
$\bigwedge_k \varphi_k$	$\sum_k s(\mu,\varphi_k)$	$\prod_k s^-(\mu,\varphi_k)$
$\bigvee_k \varphi_k$	$\prod_k s(\mu,\varphi_k)$	$\sum_k s^-(\mu,\varphi_k)$
$\varphi_1 \equiv \varphi_2$	$\begin{cases} s^-(\mu,\varphi_1)\cdot s(\mu,\varphi_2)+ \\ s(\mu,\varphi_1)\cdot s^-(\mu,\varphi_2) \end{cases}$	$\begin{cases} (s(\mu,\varphi_1)+s^-(\mu,\varphi_2))\cdot \\ (s^-(\mu,\varphi_1)+s(\mu,\varphi_2)) \end{cases}$

$s(\mu,\varphi)$ 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

Variants of DPLL

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
  $\Longrightarrow$fast and simple

– Jeroslow-Wang: choose the literal with maximum

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

  $\Longrightarrow$estimates l's contribution to the satisfiability of φ

– Satz: selects a candidate set of literals, perform unit propagation, chooses the one leading to smaller clause set
  $\Longrightarrow$maximizes the effects of unit propagation

Some preprocessing techniques

– Sorting+subsumption:

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

$$\Downarrow$$

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

Some preprocessing techniques (cont.)

– 2-simplifying [10]: exploiting binary clauses.

– **Repeat:**

1. build the implication graph induced by literals
2. detect strongly connected cycles
   $\Longrightarrow$equivalence classes of literals
3. perform substitutions
4. perform unit and pure.

Until no more simplification possible.

– Very suseful 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

$\neg A_1 \vee A_2$

$\neg A_1 \vee A_3 \vee A_9$

$\neg A_2 \vee \neg A_3 \vee A_4$

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

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

$\neg A_5 \vee \neg A_6$

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

$A_1 \vee A_8$

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

$\ldots$

Conflict-directed backtracking – example (cont.)

$\neg A_1 \vee A_2$

$\neg A_1 \vee A_3 \vee A_9$

$\neg A_2 \vee \neg A_3 \vee A_4$

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

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

$\neg A_5 \vee \neg A_6$

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

$A_1 \vee A_8$

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

$\ldots$

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

Conflict-directed backtracking – example (cont.)

$\neg A_1 \vee A_2$

$\neg A_1 \vee A_3 \vee A_9$

$\neg A_2 \vee \neg A_3 \vee A_4$

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

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

$\neg A_5 \vee \neg A_6$

$A_1 \vee A_7 \vee \neg A_{12}$ $\quad true \Longrightarrow removed$

$A_1 \vee A_8$ $\quad true \Longrightarrow removed$

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

$\ldots$

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

Conflict-directed backtracking – example (cont.)

$\neg A_1 \vee A_2$ $\quad$ $true \Longrightarrow removed$

$\neg A_1 \vee A_3 \vee A_9$ $\quad$ $true \Longrightarrow removed$

$\neg A_2 \vee \neg A_3 \vee A_4$

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

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

$\neg A_5 \vee \neg A_6$

$A_1 \vee A_7 \vee \neg A_{12}$ $\quad$ $true \Longrightarrow removed$

$A_1 \vee A_8$ $\quad$ $true \Longrightarrow removed$

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

$\ldots$

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

Conflict-directed backtracking – example (cont.)

$\neg A_1 \vee A_2$ $\quad$ $true \Longrightarrow removed$

$\neg A_1 \vee A_3 \vee A_9$ $\quad$ $true \Longrightarrow removed$

$\neg A_2 \vee \neg A_3 \vee A_4$ $\quad$ $true \Longrightarrow removed$

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

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

$\neg A_5 \vee \neg A_6$

$A_1 \vee A_7 \vee \neg A_{12}$ $\quad$ $true \Longrightarrow removed$

$A_1 \vee A_8$ $\quad$ $true \Longrightarrow removed$

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

$\ldots$

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

Conflict-directed backtracking – example (cont.)

$\neg A_1 \vee A_2$ — $true \implies removed$

$\neg A_1 \vee A_3 \vee A_9$ — $true \implies removed$

$\neg A_2 \vee \neg A_3 \vee A_4$ — $true \implies removed$

$\neg A_4 \vee A_5 \vee A_{10}$ — $true \implies removed$

$\neg A_4 \vee A_6 \vee A_{11}$ — $true \implies removed$

$\neg A_5 \vee \neg A_6$ — $false \implies conflict$

$A_1 \vee A_7 \vee \neg A_{12}$ — $true \implies removed$

$A_1 \vee A_8$ — $true \implies removed$

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

$\ldots$

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

Conflict-directed backtracking – example (cont.)

$\neg A_1 \vee A_2$	$true$	$\Longrightarrow$ $removed$
$\neg A_1 \vee A_3 \vee A_9$	$true$	$\Longrightarrow$ $removed$
$\neg A_2 \vee \neg A_3 \vee A_4$	$true$	$\Longrightarrow$ $removed$
$\neg A_4 \vee A_5 \vee A_{10}$	$true$	$\Longrightarrow$ $removed$
$\neg A_4 \vee A_6 \vee A_{11}$	$true$	$\Longrightarrow$ $removed$
$\neg A_5 \vee \neg A_6$	$false$	$\Longrightarrow$ $conflict$
$A_1 \vee A_7 \vee \neg A_{12}$	$true$	$\Longrightarrow$ $removed$
$A_1 \vee A_8$	$true$	$\Longrightarrow$ $removed$
$\neg A_7 \vee \neg A_8 \vee \neg A_{13}$		
$\ldots$		

$\Longrightarrow$ Conflict set: $\{\neg A_9, \neg A_{10}, \neg A_{11}, A_1\}$ $\Longrightarrow$ backtrack to A_1

Conflict-directed backtracking – example (cont.)

$\neg A_1 \vee A_2 \qquad true \implies removed$

$\neg A_1 \vee A_3 \vee A_9 \qquad true \implies removed$

$\neg A_2 \vee \neg A_3 \vee A_4$

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

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

$\neg A_5 \vee \neg A_6$

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

$A_1 \vee A_8$

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

$...$

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

Conflict-directed backtracking – example (cont.)

$\neg A_1 \vee A_2 \quad true \Longrightarrow removed$

$\neg A_1 \vee A_3 \vee A_9 \quad true \Longrightarrow removed$

$\neg A_2 \vee \neg A_3 \vee A_4$

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

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

$\neg A_5 \vee \neg A_6$

$A_1 \vee A_7 \vee \neg A_{12} \quad true \Longrightarrow removed$

$A_1 \vee A_8 \quad true \Longrightarrow removed$

$\neg A_7 \vee \neg A_8 \vee \neg A_{13} \quad false \Longrightarrow conflict$

$...$

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

(unit A_7, A_8)

Conflict-directed backtracking – example (cont.)

$\neg A_1 \vee A_2$	$true \implies removed$
$\neg A_1 \vee A_3 \vee A_9$	$true \implies removed$
$\neg A_2 \vee \neg A_3 \vee A_4$	
$\neg A_4 \vee A_5 \vee A_{10}$	
$\neg A_4 \vee A_6 \vee A_{11}$	
$\neg A_5 \vee \neg A_6$	
$A_1 \vee A_7 \vee \neg A_{12}$	$true \implies removed$
$A_1 \vee A_8$	$true \implies removed$
$\neg A_7 \vee \neg A_8 \vee \neg A_{13}$	$false \implies conflict$
...	

$\Longrightarrow$conflict set: $\{A_{12}, A_{13}, \neg A_1\}$.

Conflict-directed backtracking – example (cont.)

$\neg A_1 \vee A_2$ $\quad$ $true \implies removed$

$\neg A_1 \vee A_3 \vee A_9$ $\quad$ $true \implies removed$

$\neg A_2 \vee \neg A_3 \vee A_4$

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

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

$\neg A_5 \vee \neg A_6$

$A_1 \vee A_7 \vee \neg A_{12}$ $\quad$ $true \implies removed$

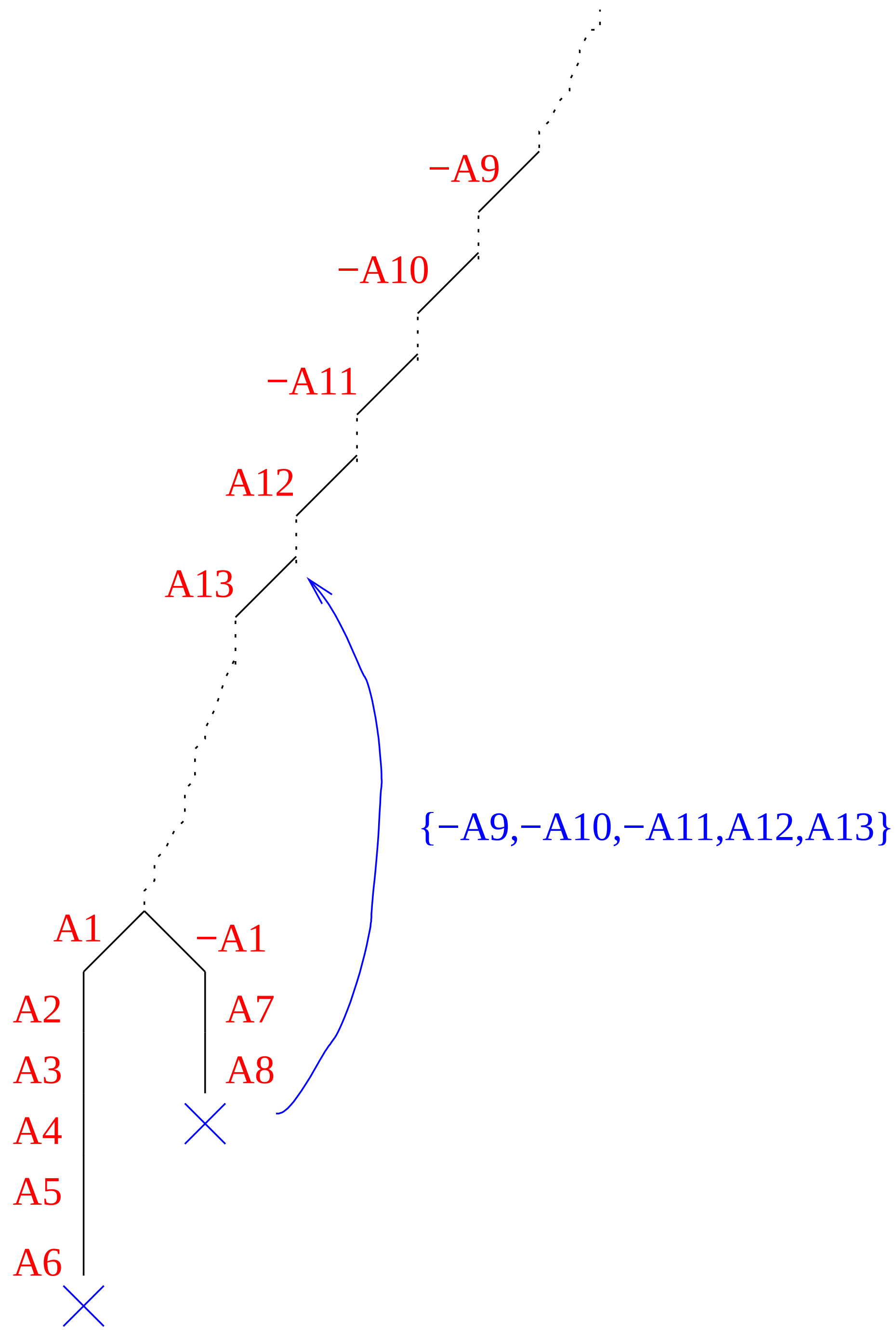
$A_1 \vee A_8$ $\quad$ $true \implies removed$

$\neg A_7 \vee \neg A_8 \vee \neg A_{13}$ $\quad$ $false \implies conflict$

...

$\implies$conflict set: $\{A_{12}, A_{13}, \neg A_1\}$ ...$\vee$ $\{\neg A_9, \neg A_{10}, \neg A_{11}, A_1\}$

$\implies\{\neg A_9, \neg A_{10}, \neg A_{11}, A_{12}, A_{13}\} \implies$ backtrack to A_{13}.


−A9
−A10
−A11
A12
A13
{−A9,−A10,−A11,A12,A13}
A1
−A1
A2
A3
A4
A5
A6
A7
A8

Learning [8, 51]

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

Learning – example (cont.)

$\neg A_1 \vee A_2$ $\quad true \Longrightarrow removed$

$\neg A_1 \vee A_3 \vee A_9$ $\quad true \Longrightarrow removed$

$\neg A_2 \vee \neg A_3 \vee A_4$ $\quad true \Longrightarrow removed$

$\neg A_4 \vee A_5 \vee A_{10}$ $\quad true \Longrightarrow removed$

$\neg A_4 \vee A_6 \vee A_{11}$ $\quad true \Longrightarrow removed$

$\neg A_5 \vee \neg A_6$ $\quad false \Longrightarrow conflict$

$A_1 \vee A_7 \vee \neg A_{12}$ $\quad true \Longrightarrow removed$

$A_1 \vee A_8$ $\quad true \Longrightarrow removed$

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

...

$A_9 \vee A_{10} \vee A_{11} \vee \neg A_1$ $\quad learned\ clause$

$\Longrightarrow$Conflict set: $\{\neg A_9, \neg A_{10}, \neg A_{11}, A_1\}$

$\Longrightarrow$learn $A_9 \vee A_{10} \vee A_{11} \vee \neg A_1$

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.

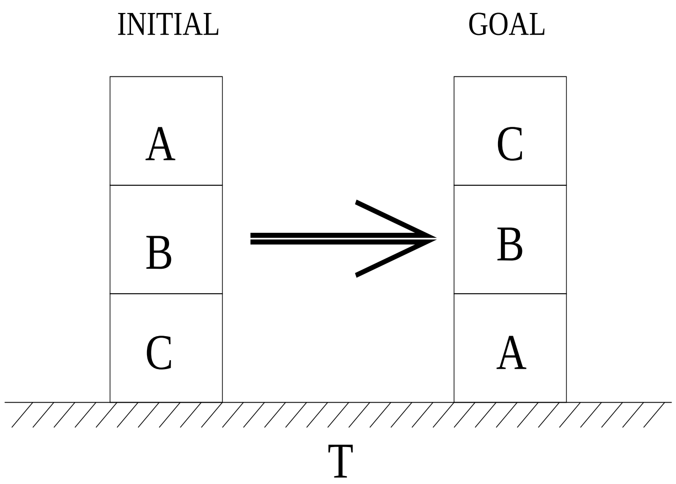
– Key issue: find an efficient encoding.

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 $o_1, .., o_n$, leading from the initial state to the goal state.
- Idea: Encode it into satisfiability problem of a boolean formula φ

Example



$$Move(b, s, d)$$

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

Encoding

– Initial states:

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

– Goal states:

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

– Action preconditions and effects:

$$\begin{aligned}
&Move_t(A,B,C) \rightarrow \\
&\quad Clear_{t-1}(A) \wedge On_{t-1}(A,B) \wedge Clear_{t-1}(C) \wedge \\
&\quad Clear_{t+1}(B) \wedge \neg On_{t+1}(A,B) \wedge \\
&\quad On_{t+1}(A,C) \wedge \neg Clear_{t+1}(C).
\end{aligned}$$

Encoding: Frame axioms

– Classic

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

"At least one action" axiom:

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

– Explanatory

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

Planning strategy

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

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

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

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

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
  - $h \wedge g$, $h \vee g$, $\mathbf{X}g$, $\mathbf{G}g$, $\mathbf{F}g$,$h\mathbf{U}g$ and $h\mathbf{R}g$,
    $\mathbf{X}$, $\mathbf{G}$, $\mathbf{F}$, $\mathbf{U}$, $\mathbf{R}$ "next", "globally", "eventually", "until" and "releases"

– an integer k (bound)

The problem (cont.)

Problem:

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

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

The encoding

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

$$[[M,f]]_k \quad := \quad [[M]]_k \wedge [[f]]_k \tag{1}$$

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

$$[[f]]_k \quad := \quad (\neg \bigvee_{l=0}^{k} T(s_k, s_l) \wedge [[f]]_k^0) \vee \bigvee_{l=0}^{k} (T(s_k, s_l) \wedge {}_l[[f]]_k^0), \tag{3}$$

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

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

Example: $\mathbf{F}p$ (reachability)

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

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

Example: $\mathrm{G}p$

– $f := \mathbf{G}p$: is there a path where p holds forever?

– $[[M, f]]_k$ is:

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

Example: $\mathbf{GF}q \wedge \mathbf{F}p$ (fair reachability)

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

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

Bounded Model Checking

- 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., $P(x)$, $\Box(A_1 \vee \Box A_2)$, $(x - y \geq 6)$, $\exists$ $\textsc{Children}$ $(\textsc{male} \wedge \textsc{teen})$)
- if φ_1 and φ_2 formulas, then $\neg\varphi_1$, $\varphi_1 \wedge \varphi_2$, $\varphi_1 \vee \varphi_2$, $\varphi_1 \rightarrow \varphi_2$, $\varphi_1 \leftrightarrow \varphi_2$ are formulas.
- Nothing else is a formula (e.g., no external quantifiers!)

Ingredients (cont.)

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

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

$$M \models \neg\phi \iff M \not\models \phi$$

$$M \models \varphi_1 \wedge \varphi_2 \iff M \models \varphi_1 \textbf{ and } M \models \varphi_2$$

$$M \models \varphi_1 \vee \varphi_2 \iff M \models \varphi_1 \textbf{ or } M \models \varphi_2$$

$$M \models \varphi_1 \rightarrow \varphi_2 \iff \textbf{if} M \models \varphi_1 \textbf{ then } M \models \varphi_2$$

$$M \models \varphi_1 \leftrightarrow \varphi_2 \iff M \models \varphi_1 \textbf{ iff } M \models \varphi_2$$

Ingredients (cont.)

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

E.g.:

- $\text{FO-Solve}(\{P(x,a), P(b,y)\}) \Longrightarrow$ Sat
- $\text{K-Solve}(\{\Box(A_1 \rightarrow A_2), \Box(A_1), \neg\Box(A_2)\}) \Longrightarrow$ Unsat
- $\text{Math-Solve}(\{(x - y \leq 3), (y - z \leq 4), \neg(x - z \leq 8)\})$ $\Longrightarrow$ Unsat
- $\mathcal{ALC}\text{-Solve}\left(\left\{\begin{array}{l} \forall\, \text{Children}\ (\neg \text{male} \vee \text{teen}), \\ \forall\, \text{Children}\ (\text{male}), \\ \exists\, \text{Children}\ (\neg\ \text{teen}) \end{array}\right\}\right)$ $\Longrightarrow$ 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:
  $P(x)$, $\neg\forall x.Q(x, f(a))$
  $\Box(A_1 \vee \Box A_2)$, $\neg\Box(A_2 \rightarrow \Box(A_3 \vee A_4))$
  $(x - y \geq 6)$, $\neg(z - y < 2)$,
  $\exists$ $\textsc{Children}$ $(\textsc{male} \wedge \textsc{teen})$,$\neg\forall$ $\textsc{Parent}$ $(\textsc{old})$
- $Atoms(\varphi)$: the set of top-level atoms in φ.

Definitions: total truth assignment

– We call a total truth assignment μ for φ a total function

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

– We represent a total truth assignment μ 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

$$\mu = \bigwedge_i \alpha_i \wedge \bigwedge_j \neg\beta_j \wedge \bigwedge_{k=1}^{R} A_k \ \wedge \bigwedge_{h=R+1}^{S} \neg A_h$$

Definitions: partial truth assignment

– We call a partial truth assignment μ for φ a partial function

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

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

– A partial truth assignment μ for φ is a subset of a total truth assignment for φ.

– If $\mu_2 \subseteq \mu_1$, then we say that μ_1 extends μ_2 and that μ_2 subsumes μ_1.

– a conflict set for μ_1 is an inconsistent subset $\mu_2 \subseteq \mu_1$ s.t. no strict subset of μ_2 is inconsistent.

Definitions: total and partial truth assignment (cont.)

Remark:

- Syntactically identical instances of the same atom in φ are always assigned identical truth values.
  E.g., $... \wedge ((t_1 \geq t_2) \vee A_1) \wedge ((t_1 \geq t_2) \vee A_2) \wedge ...$
- Equivalent but syntactically different atoms in φ may (in principle) be assigned different truth values.
  E.g., $... \wedge ((t_1 \geq t_2) \vee A_1) \wedge ((t_2 \leq t_1) \vee A_2) \wedge ...$

Definition: propositional satisfiability in $\mathcal{L}$

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

$$
\begin{array}{lcl}
\mu \models_p \varphi_1,\ \varphi_1 \in Atoms(\varphi) & \Longleftrightarrow & \varphi_1 \in \mu; \\
\mu \models_p \neg\varphi_1 & \Longleftrightarrow & \mu \not\models_p \varphi_1; \\
\mu \models_p \varphi_1 \wedge \varphi_2 & \Longleftrightarrow & \mu \models_p \varphi_1 \ and\ \mu \models_p \varphi_2. \\
\ldots & \ldots & \ldots
\end{array}
$$

– A partial assignment μ propositionally satisfies φ iff all total assignments extending μ propositionally satisfy φ.

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

- Intuition: If φ is seen as a boolean combination of its atoms, $\models_p$ is standard propositional satisfiability.
- Atoms seen as (recognizable) blackboxes
- The definitions of $\varphi_1 \models_p \varphi_2$, $\models_p \varphi$ 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., $(v_1 \leq v_2) \wedge (v_2 \leq v_3) \models (v_1 \leq v_3)$, but $(v_1 \leq v_2) \wedge (v_2 \leq v_3) \not\models_p (v_1 \leq v_3)$.

Satisfiability and propositional satisfiability in $\mathcal{L}$

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

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

– Search decomposed into two orthogonal components:
  - Purely propositional: search for a truth assignments μ propositionally satisfying φ
  - Purely domain-dependent: verify the satisfiability in $\mathcal{L}$ of μ.

Example

$$\varphi = \begin{array}{l} \{\neg(2v_2 - v_3 > 2) \vee A_1\} \wedge \\ \{\neg A_2 \vee (2v_1 - 4v_5 > 3)\} \wedge \\ \{(3v_1 - 2v_2 \leq 3) \vee A_2\} \wedge \\ \{\neg(2v_3 + v_4 \geq 5) \vee \neg(3v_1 - v_3 \leq 6) \vee \neg A_1\} \wedge \\ \{A_1 \vee (3v_1 - 2v_2 \leq 3)\} \wedge \\ \{(v_1 - v_5 \leq 1) \vee (v_5 = 5 - 3v_4) \vee \neg A_1\} \wedge \\ \{A_1 \vee (v_3 = 3v_5 + 4) \vee A_2\}. \end{array}$$

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

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

– $\mu \models_p \varphi$, 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 φ is complete iff

$$\models_p \varphi \leftrightarrow \bigvee_j \mu_j. \quad (4)$$

– for every total assignment η s.t. $\eta \models_p \varphi$, there is $\mu_i \in \mathcal{M}$ s.t. $\mu_i \subseteq \eta$.
  $\Longrightarrow \mathcal{M}$ represents all assignments.

– $\mathcal{M}$ "compact" representation of the whole set of total assignments propositionally satisfying φ.

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 φ. Then φ is satisfiable if and only if μ_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 φ;
  - Purely domain-dependent: check one by one the satisfiability in $\mathcal{L}$ of the μ_i's.

Redundancy of complete collection of assignments

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

– strongly non redundant iff, for every $\mu_i, \mu_j \in \mathcal{M}$, $(\mu_i \wedge \mu_j)$ 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 $\mu_i, \mu_j \in \mathcal{M}$:

$$
\begin{array}{lcll}
\models_p \varphi \leftrightarrow \bigvee_{i \neq j} \mu_i & \Longrightarrow & \models_p \bigvee_i \mu_i \leftrightarrow \bigvee_{i \neq j} \mu_i & \Longrightarrow \\
\bigvee_i \mu_i \models_p \bigvee_{i \neq j} \mu_i & \Longrightarrow & \mu_j \models_p \bigvee_{i \neq j} \mu_i & \Longrightarrow \\
\mu_j \models_p \mu_i \;\; for\ some\ i & \Longrightarrow & \mu_j \supseteq \mu_i &
\end{array}
$$

– If $\mathcal{M}$ is strongly non redundant, then $\mathcal{M}$ is non redundant:

$$
\begin{array}{ll}
\mu_j \wedge \mu_i \;\; propositionally\ inconsistent & \Longrightarrow \\
\mu_j \models_p \neg \mu_i & \Longrightarrow \\
\mathcal{M}\ non\ redundant &
\end{array}
$$

Redundancy: example

Let $\varphi := (\alpha \vee \beta \vee \gamma) \wedge (\alpha \vee \beta \vee \neg\gamma)$, α, β, γ atoms. Then

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

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

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

4. $\{\{\alpha\}, \{\neg\alpha, \beta\}\}$ is complete and strongly non redundant.

A Generalized Search Procedure

Truth assignment enumerator

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

– A truth assignment enumerator is
  - strongly non-redundant if $\textsc{Assign\_Enumerator}(\varphi)$ is strongly non-redundant, for every φ,
  - non-redundant if $\textsc{Assign\_Enumerator}(\varphi)$ is non-redundant, for every φ,
  - redundant otherwise.

Truth assignment enumerator w.r.t. SAT solver

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 L-SAT(formula φ, assignment & μ, model & M)
    do
        μ := Next_Assignment(φ)                  /* next in {μ_1, ..., μ_n} */
        if (μ ≠ Null)
            satifiable := L-SOLVE(μ, M);
    while ((satifiable = False) and (μ ≠ Null))
    if (satifiable ≠ False)
    then return True;                            /* a satisf. assignment found */
    else return False;                           /* no satisf. assignment found */
```

$\mathcal{L}$-SAT

- $\mathcal{L}$-SAT(φ) terminating, correct and complete $\Longleftrightarrow$ $\mathcal{L}$-SOLVE(μ) 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 φ
- (Completeness) generate complete set of assignments

Mandatory requirements for $\mathcal{L}$-SOLVE()

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

- (Termination) terminate
- (Correctness & completeness) return $True$ if μ 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 tale 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.

  $\Longrightarrow$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(μ_1)]

Extending existing SAT procedures

General ideas

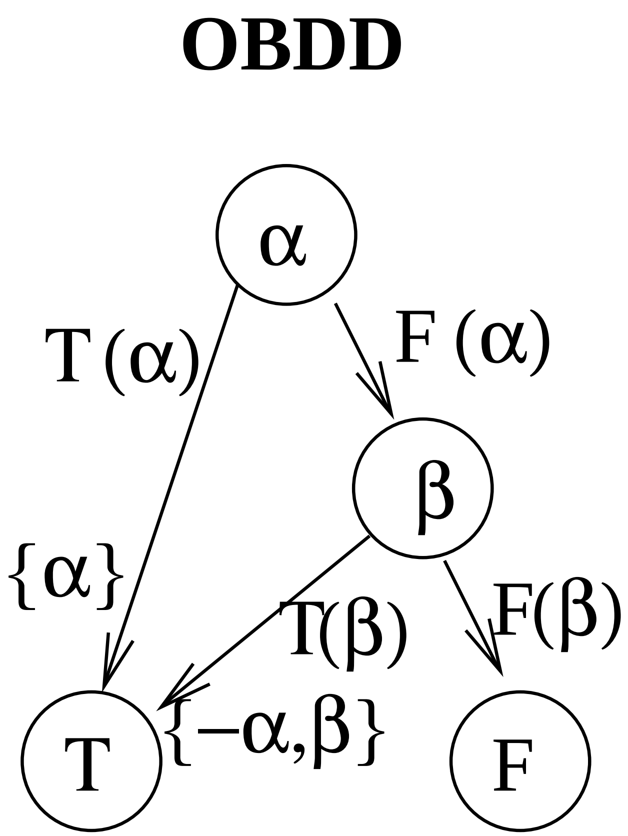
Existing SAT procedures are natural candidates to be used as assignment enumerators.

- 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

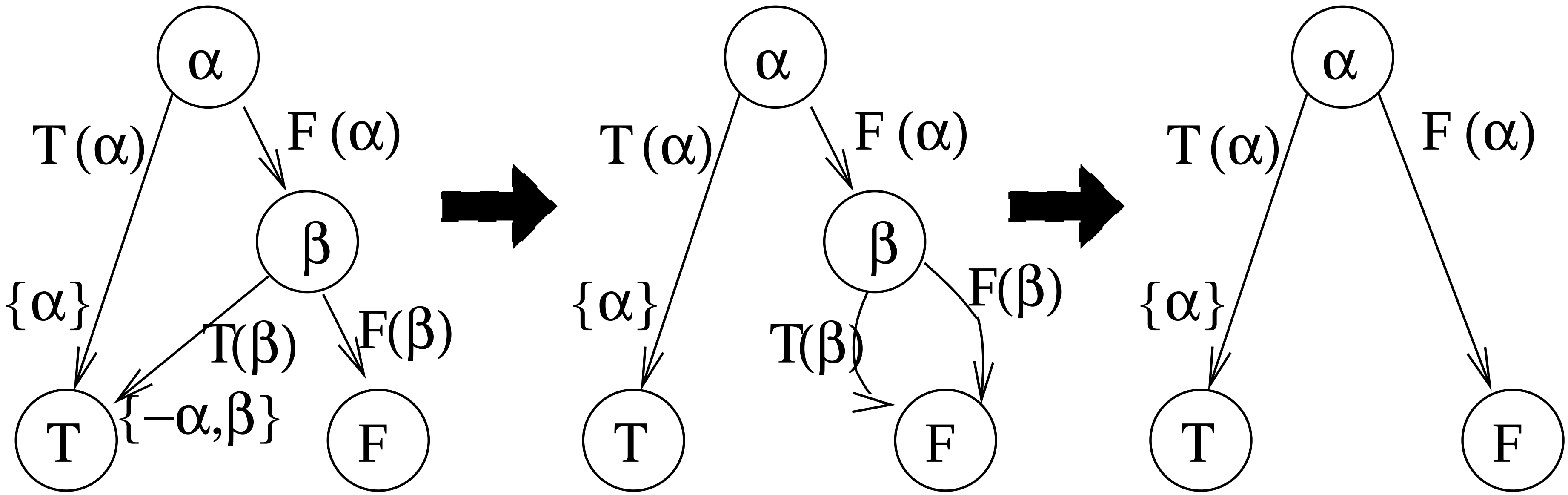
– In an OBDD, the set of paths from the root to (1) represent a complete collection of assignments
– Some may be inconsistent in $\mathcal{L}$
– Reduction: [13, 42]
   1. inconsistent paths from the root to internal nodes are detected
   2. they are redirected to the (0) node
   3. the resulting OBDD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

OBDD: example



OBDD of $(\alpha \vee \beta \vee \gamma) \wedge (\alpha \vee \beta \vee \neg\gamma)$.

OBDD reduction: example



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

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

$$\left\{ \begin{array}{ccc} \dfrac{\varphi_1 \wedge \varphi_2}{\begin{array}{c}\varphi_1\\ \varphi_2\end{array}} & \dfrac{\neg(\varphi_1 \vee \varphi_2)}{\begin{array}{c}\neg\varphi_1\\ \neg\varphi_2\end{array}} & \dfrac{\neg(\varphi_1 \rightarrow \varphi_2)}{\begin{array}{c}\varphi_1\\ \neg\varphi_2\end{array}} \\ & \dfrac{\neg\neg\varphi}{\varphi} & \\ \dfrac{\varphi_1 \vee \varphi_2}{\varphi_1 \quad\quad \varphi_2} & \dfrac{\neg(\varphi_1 \wedge \varphi_2)}{\neg\varphi_1 \quad\quad \neg\varphi_2} & \dfrac{\varphi_1 \rightarrow \varphi_2}{\neg\varphi_1 \quad\quad \varphi_2} \\ \dfrac{\varphi_1 \leftrightarrow \varphi_2}{\begin{array}{cc}\varphi_1 & \neg\varphi_1\\ \varphi_2 & \neg\varphi_2\end{array}} & \dfrac{\neg(\varphi_1 \leftrightarrow \varphi_2)}{\begin{array}{cc}\varphi_1 & \neg\varphi_1\\ \neg\varphi_2 & \varphi_2\end{array}} & \end{array} \right\} \cup \left\{ \begin{array}{c}\mathcal{L}\text{-specific}\\ \text{Rules}\end{array} \right\}$$

– Widely used by logicians

Generalized tableau algorithm

function $\mathcal{L}$-*Tableau*(Γ)

if $A_i \in \Gamma$ **and** $\neg A_i \in \Gamma$ /* branch closed */

then return *False*;

if $(\varphi_1 \wedge \varphi_2) \in \Gamma$ /* $\wedge$-elimination */

then return $\mathcal{L}$-*Tableau*$(\Gamma \cup \{\varphi_1, \varphi_2\} \backslash \{(\varphi_1 \wedge \varphi_2)\})$;

if $(\neg\neg\varphi_1) \in \Gamma$ /* $\neg\neg$-elimination */

then return $\mathcal{L}$-*Tableau*$(\Gamma \cup \{\varphi_1\} \backslash \{(\neg\neg\varphi_1)\})$;

if $(\varphi_1 \vee \varphi_2) \in \Gamma$ /* $\vee$-elimination */

then return $\mathcal{L}$-*Tableau*$(\Gamma \cup \{\varphi_1\} \backslash \{(\varphi_1 \vee \varphi_2)\})$ **or**

$\mathcal{L}$-*Tableau*$(\Gamma \cup \{\varphi_2\} \backslash \{(\varphi_1 \vee \varphi_2)\})$;

...

return ($\mathcal{L}$-SOLVE(Γ)= satisfiable); /* branch expanded */

General tableaux: example

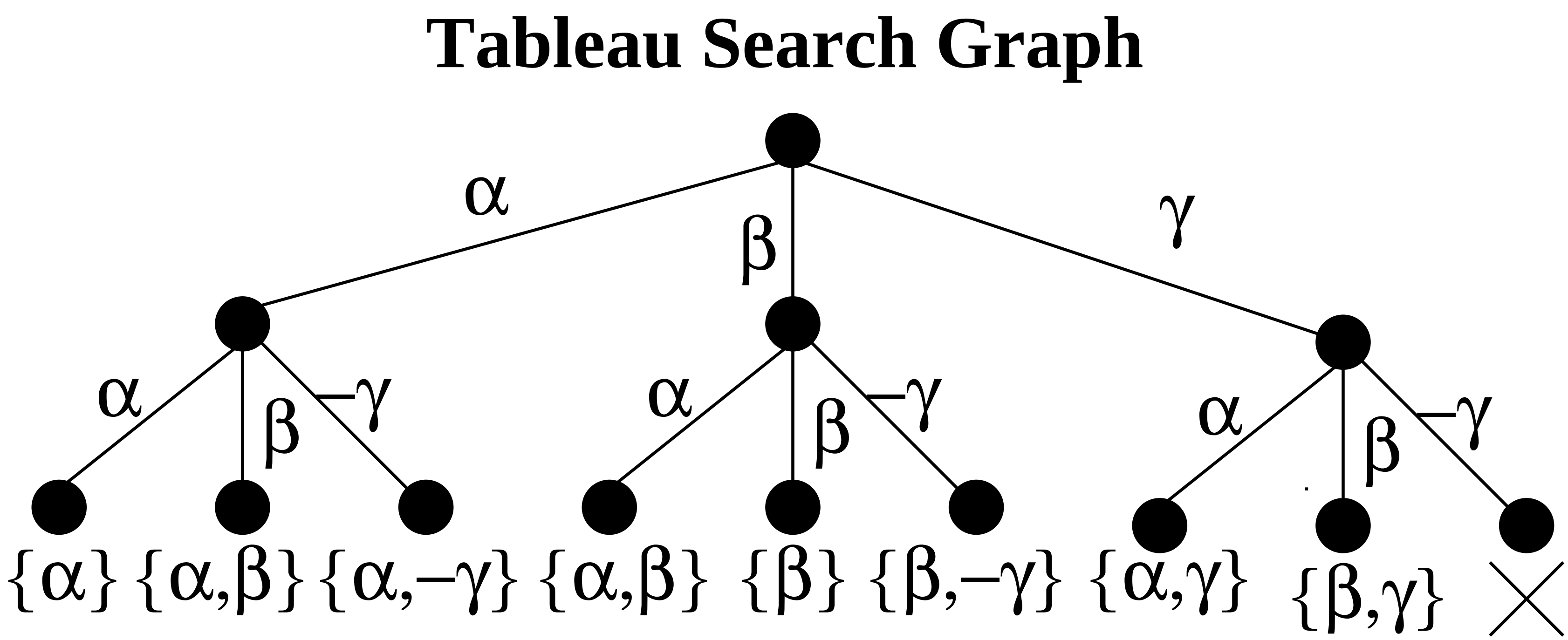


Tableau search graph for $(\alpha \vee \beta \vee \gamma) \wedge (\alpha \vee \beta \vee \neg\gamma)$.

Generalized tableaux: problems

Two main problems [16, 30, 31]

- syntactic branching
  - branch on disjunctions
  - possible many duplicate or subsumed branches
    $\Longrightarrow$redundant
  - duplicates search (both propositional and domain-dependent)
- no constraint violation detection
  - incapable to detect when current branches violate a constraint
    $\Longrightarrow$lots of redundant propositional search.

Syntactic branching: example

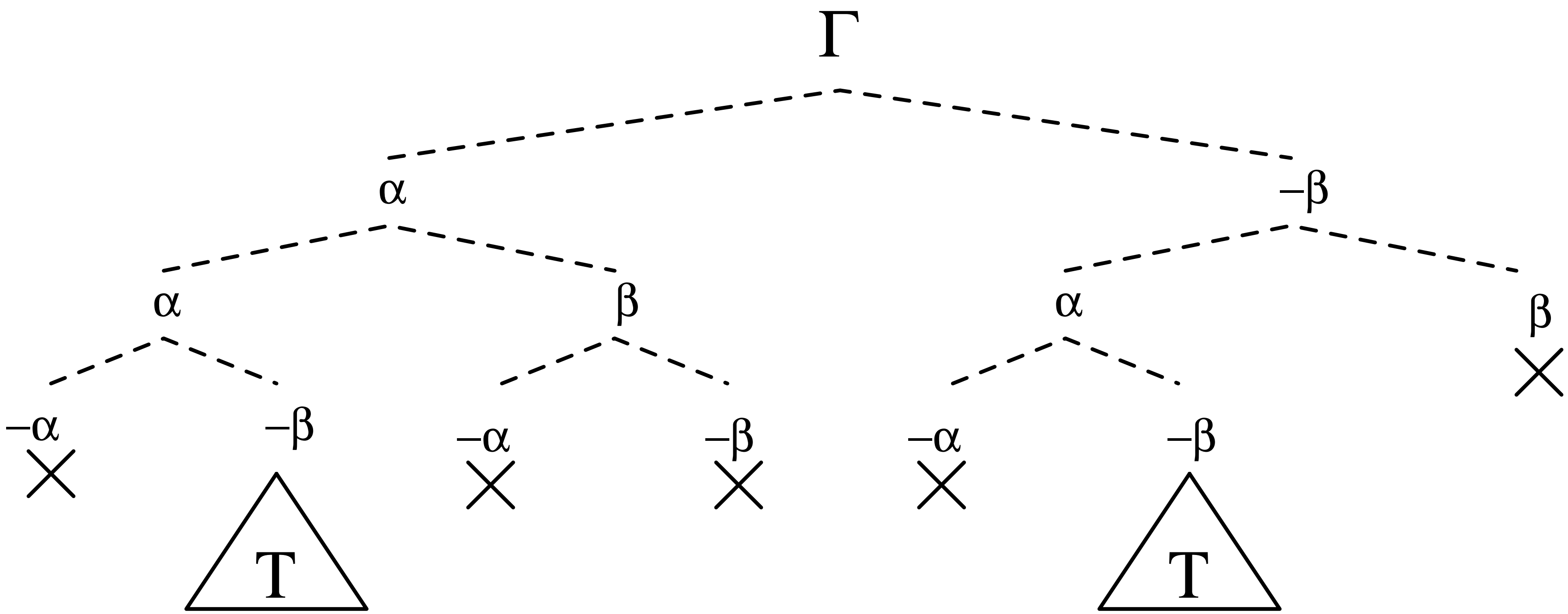


Tableau search graph for $(\alpha \vee \neg\beta) \wedge (\alpha \vee \beta) \wedge (\neg\alpha \vee \neg\beta)$.

Detecting constraints violations: example

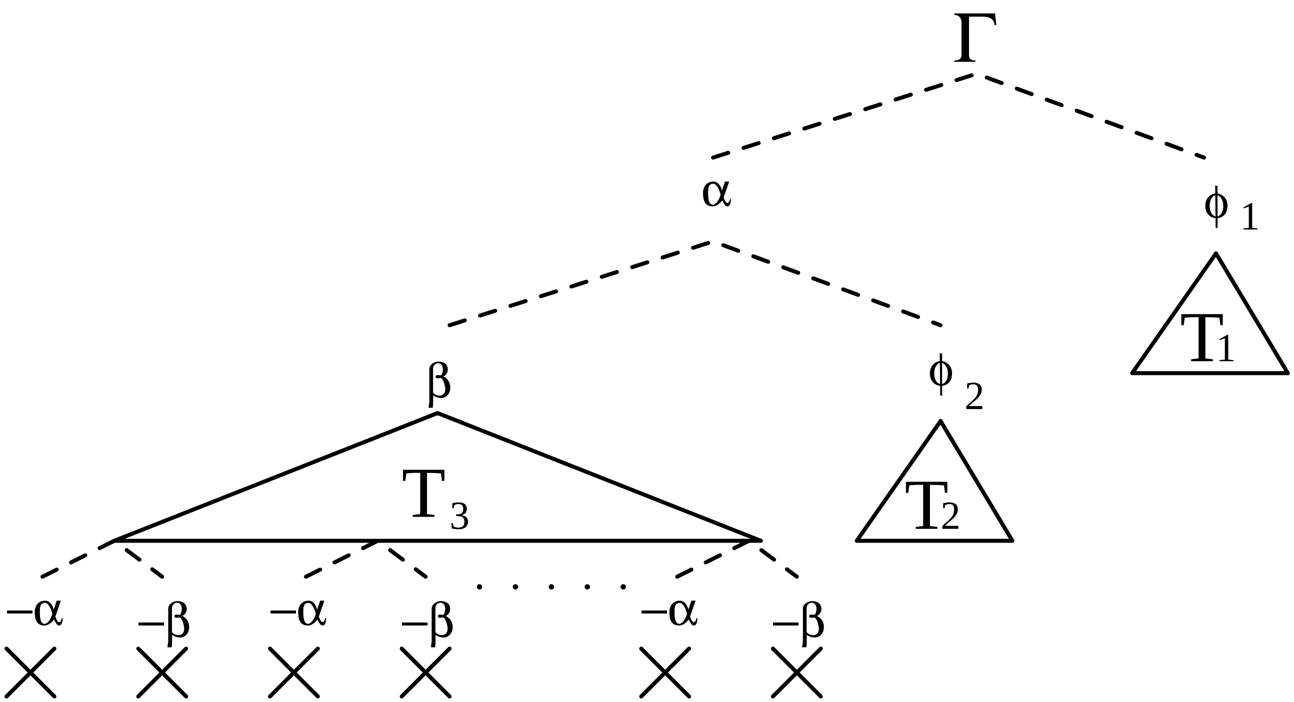


Tableau search graph for

$(\alpha \vee \phi_1) \wedge (\beta \vee \phi_2) \wedge \phi_3 \wedge (\neg\alpha \vee \neg\beta)$

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

$$\left\{ \begin{array}{ll} \dfrac{\varphi_1 \wedge (l) \wedge \varphi_2}{(\varphi_1 \wedge \varphi_2)[l|\top]} & (Unit) \\ \\ \dfrac{\varphi}{\varphi[l|\top] \quad \varphi[l|\bot]} & (split) \end{array} \right\} \cup \left\{ \begin{array}{c} \mathcal{L}\text{-specific} \\ \text{Rules} \end{array} \right\}$$

– No Pure Literal Rule: Pure literal causes incomplete assignment sets!

Pure literal and Generalized DPLL: Example

$$
\begin{aligned}
\varphi = \quad & ((x - y \leq 1) \vee A_1) \wedge \\
& ((y - z \leq 2) \vee A_2) \wedge \\
& (\neg(x - z \leq 4) \vee A_2) \wedge \\
& (\neg A_2 \vee A_3) \wedge \\
& (\neg A_2 \vee \neg A_3)
\end{aligned}
$$

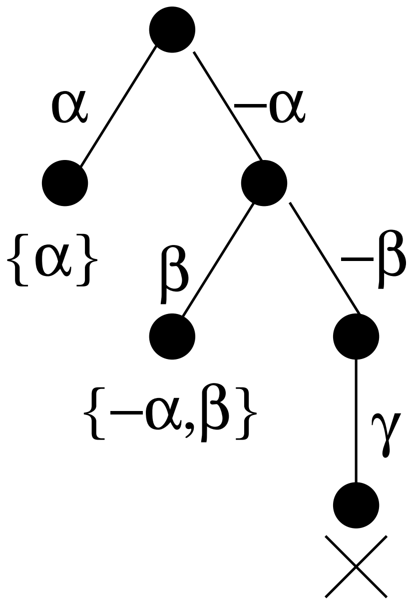
- A satisfiable assignment propositionally satisfying φ is: $\mu = \{A_1, \neg A_2, (y - z \leq 2), \neg(x - z \leq 4)\}$
- No satisfiable assignment propositionally satisfying φ contains $(x - y \leq 1)$
- Pure literal may assign $(x - y \leq 1) := \top$ as first step $\Longrightarrow$return unsatisfiable.

Generalized DPLL algorithm

```
function L-DPLL(φ, μ)
    if φ = ⊤                                              /* base      */
        then return (L-SOLVE(μ)=satisfiable);
    if φ = ⊥                                              /* backtrack */
        then return False;
    if {a unit clause (l) occurs in φ}                    /* unit      */
        then return L-DPLL(assign(l, φ), μ ∧ l);
    l := choose-literal(φ);                               /* split     */
    return  L-DPLL(assign(l, φ), μ ∧ l)  or
            L-DPLL(assign(¬l, φ), μ ∧ ¬l);
```

General DPLL: example

DPLL search graph



DPLL search graph for $(\alpha \vee \beta \vee \gamma) \wedge (\alpha \vee \beta \vee \neg\gamma)$.

Generalized DPLL vs. generalized tableaux

Two big advantages: [16, 30, 31]

– semantic vs. syntactic branching
  - branch on truth values
  - no duplicate or subsumed branches
    $\Longrightarrow$strongly non redundant
  - no search duplicates

– constraint violation detection
  - backtracks as soon as the current branch violates a constraint
    $\Longrightarrow$no redundant propositional search.

Semantic branching: example

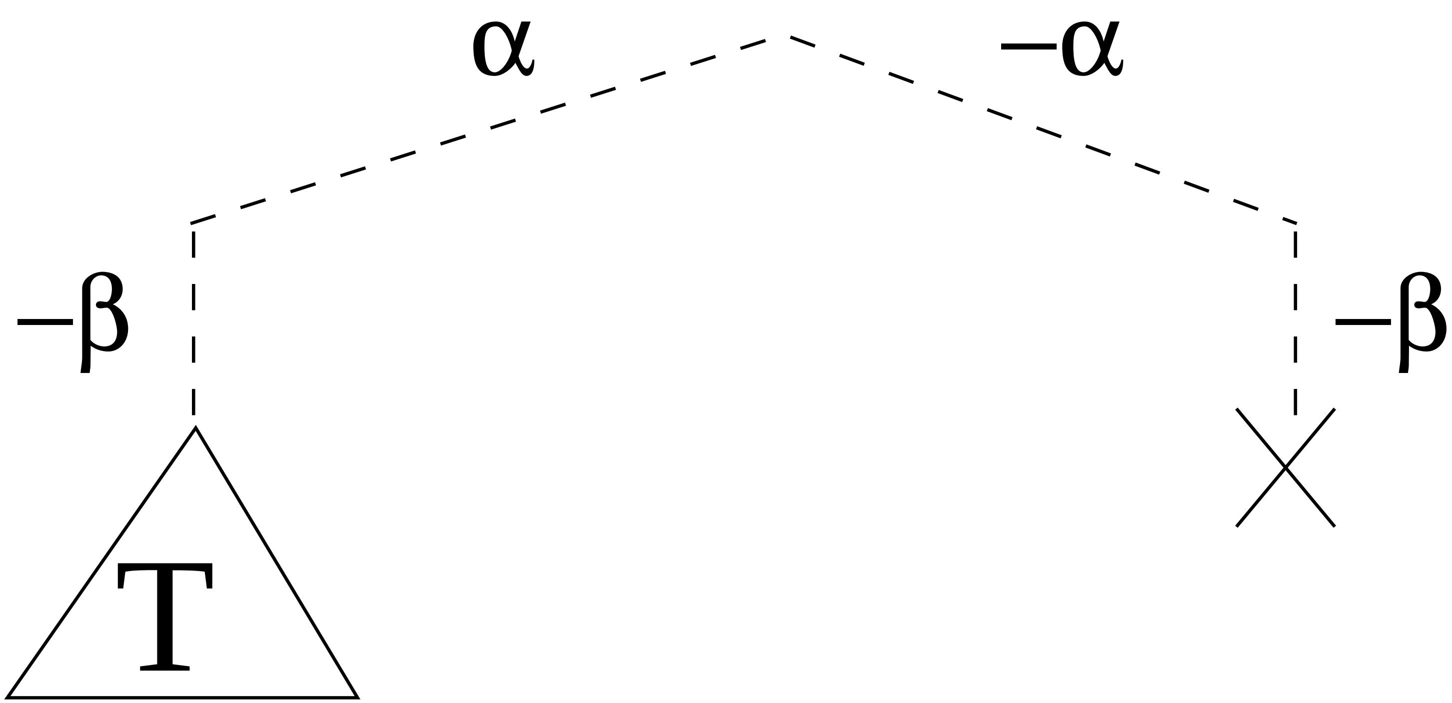
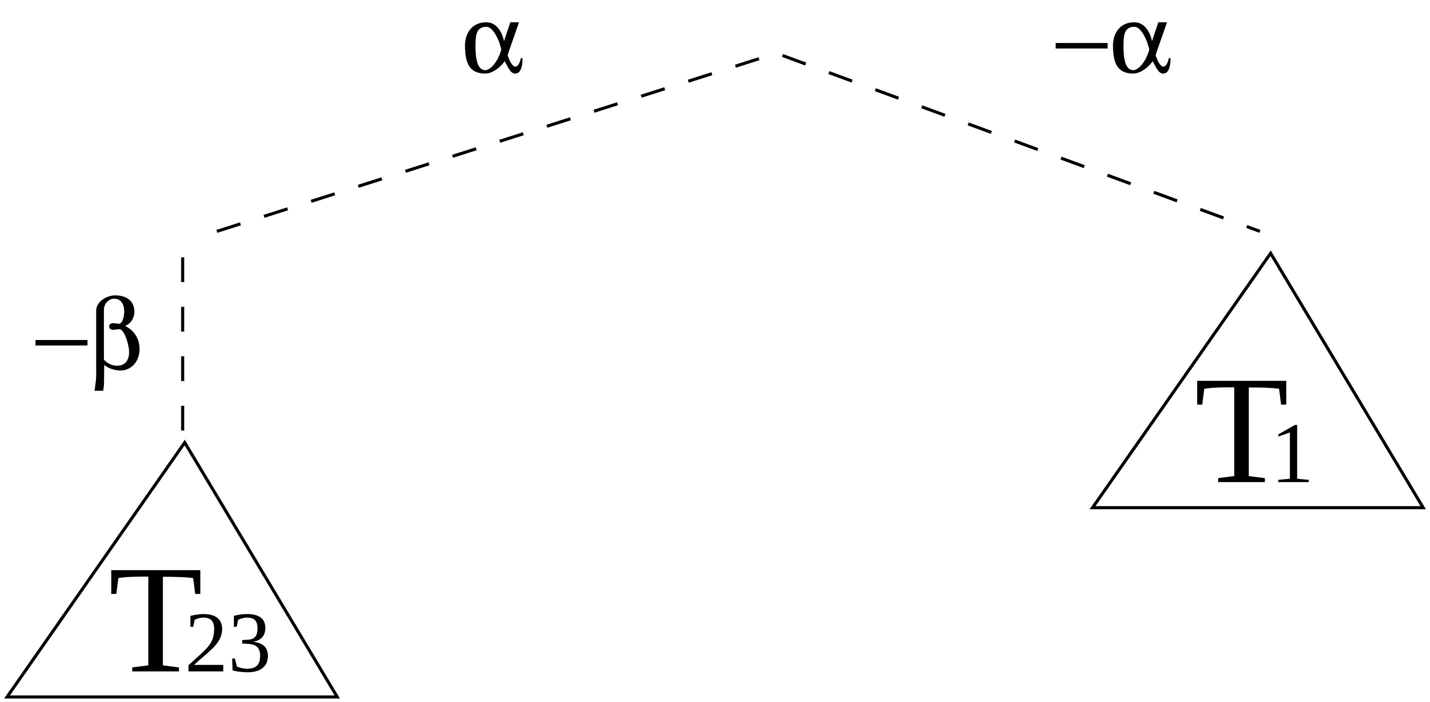


Tableau search graph for $(\alpha \vee \neg\beta) \wedge (\alpha \vee \beta) \wedge (\neg\alpha \vee \neg\beta)$.

Detecting constraints violations: example



DPLL search graph for $(\alpha \vee \phi_1) \wedge (\beta \vee \phi_2) \wedge \phi_3 \wedge (\neg\alpha \vee \neg\beta)$

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] $\Longrightarrow$ they may be assigned different [resp. identical] truth values.

Solution: rewrite trivially equivalent atoms into one.

Preprocessing atoms (cont.)

- Sorting: $(v_1 + v_2 \leq v_3 + 1)$, $(v_2 + v_1 \leq v_3 + 1)$, $(v_1 + v_2 - 1 \leq v_3) \Longrightarrow (v_1 + v_2 - v_3 \leq 1)$);
- Rewriting dual operators: $(v_1 < v_2)$, $(v_1 \geq v_2) \Longrightarrow (v_1 < v_2)$, $\neg(v_1 < v_2)$
- Exploiting associativity: $(v_1 + (v_2 + v_3) = 1)$, $((v_1 + v_2) + v_3) = 1) \Longrightarrow$ $(v_1 + v_2 + v_3 = 1)$;
- Factoring $(v_1 + 2.0v_2 \leq 4.0)$, $(-2.0v_1 - 4.0v_2 \geq -8.0)$, $\Longrightarrow (0.25v_1 + 0.5v_2 \leq 1.0)$;
- Exploiting properties of $\mathcal{L}$: $(v_1 \leq 3)$, $(v_1 < 4) \Longrightarrow (v_1 \leq 3)$ 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., $\neg(x - y < 3) \vee (x - y < 5)$)

– (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 μ' is unsatisfiable, then all its extensions are unsatisfiable.
- the unsatisfiability of μ' detected during its construction,
  $\Longrightarrow$avoids checking the satisfiability of all the up to $2^{|Atoms(\varphi)|-|\mu'|}$ assignments extending μ'.
- Introduce a satisfiability test on incomplete assignments just before every branching step:

```
if Likely-Unsatisfiable(μ)                /* early pruning */
    if (L-SOLVE(μ) = False)
        then return False;
```

DPLL+Early pruning

function $\mathcal{L}$-*DPLL*(φ, μ)

  if $\varphi = \top$ /* base */

    then return ($\mathcal{L}$-SOLVE(μ)=satisfiable);

  if $\varphi = \bot$ /* backtrack */

    then return *False*;

  if {a unit clause (l) occurs in φ} /* unit */

    then return $\mathcal{L}$-*DPLL*($assign(l, \varphi), \mu \wedge l$);

  if Likely-Unsatisfiable(μ) /* early pruning */

    if ($\mathcal{L}$-SOLVE(μ) $=$ $False$)

      then return *False;*

  l := choose-literal(φ); /* split */

  return $\mathcal{L}$-*DPLL*($assign(l, \varphi), \mu \wedge l$) **or**

    $\mathcal{L}$-*DPLL*($assign(\neg l, \varphi), \mu \wedge \neg l$);

Early pruning: example

$$
\begin{aligned}
\varphi = \quad & \{\neg(2v_2 - v_3 > 2) \vee A_1\} \wedge \\
& \{\neg A_2 \vee (2v_1 - 4v_5 > 3)\} \wedge \\
& \{(3v_1 - 2v_2 \leq 3) \vee A_2\} \wedge \\
& \{\neg(2v_3 + v_4 \geq 5) \vee \neg(3v_1 - v_3 \leq 6) \vee \neg A_1\} \wedge \\
& \{A_1 \vee (3v_1 - 2v_2 \leq 3)\} \wedge \\
& \{(v_1 - v_5 \leq 1) \vee (v_5 = 5 - 3v_4) \vee \neg A_1\} \wedge \\
& \{A_1 \vee (v_3 = 3v_5 + 4) \vee A_2\}.
\end{aligned}
$$

– Suppose it is built the intermediate assignment:

$$\mu' = \neg(2v_2 - v_3 > 2) \wedge \neg A_2 \wedge (3v_1 - 2v_2 \leq 3) \wedge \neg(3v_1 - v_3 \leq 6).$$

– If $\mathcal{L}$-SOLVE is invoked on μ', it returns $False$, and $\mathcal{L}$-DPLL backtracks without exploring any extension of μ'.

Early pruning: drawback

- Reduces drastically the search
- Drawback: possibly lots of useless calls to $\mathcal{L}\text{-SOLVE}$
  $\Longrightarrow$to be used with care when $\mathcal{L}\text{-SOLVE}$ calls recursively $\mathcal{L}\text{-SAT}$ (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 $\mathcal{L}\text{-SOLVE}$

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 $\mathcal{L}\text{-SOLVE}$

– With early pruning, lots of incremental calls to $\mathcal{L}\text{-SOLVE}$:

$$
\begin{array}{lcl}
\mathcal{L}\text{-SOLVE}(\mu) & \Longrightarrow & \text{satisfiable} \\
\mathcal{L}\text{-SOLVE}(\mu \cup \mu') & \Longrightarrow & \text{satisfiable} \\
\mathcal{L}\text{-SOLVE}(\mu \cup \mu' \cup \mu'') & \Longrightarrow & \text{satisfiable}
\end{array}
$$

...

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

– requires saving the status of $\mathcal{L}\text{-SOLVE}$

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 $\mathcal{L}\text{-SOLVE}$ 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 η s.t. $\mu \models \eta$, and return it.

– The literals in η are then unit-propagated away.

Enhanced Early Pruning: Examples

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

– 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 μ.

– 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 μ.

Enhanced Early Pruning: summary

- Further improves efficiency with DPLL
- Presumably scarcely effective with semantic tableaux
- Effective with OBDDs?
- Requires a sophisticated $\mathcal{L}\text{-SOLVE}$

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

$$
\begin{aligned}
\varphi = \quad & \{\neg(2v_2 - v_3 > 2) \vee A_1\} \wedge \\
& \{\neg A_2 \vee (2v_1 - 4v_5 > 3)\} \wedge \\
& \{(3v_1 - 2v_2 \leq 3) \vee A_2\} \wedge \\
& \{\neg(2v_3 + v_4 \geq 5) \vee \neg(3v_1 - v_3 \leq 6) \vee \neg A_1\} \wedge \\
& \{A_1 \vee (3v_1 - 2v_2 \leq 3)\} \wedge \\
& \{(v_1 - v_5 \leq 1) \vee (v_5 = 5 - 3v_4) \vee \neg A_1\} \wedge \\
& \{A_1 \vee (v_3 = 3v_5 + 4) \vee A_2\}.
\end{aligned}
$$

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

– $\mathcal{L}$-SOLVE(μ) returns *false* with the conflict set:

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

– $\mathcal{L}$-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 $\mathcal{L}$-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
- $\mathcal{L}\text{-SOLVE}(\mu)$ checks first if (any subset of) μ is cached. If yes, returns unsatisfiable.

– Idea 2:

- When a satisfying (sub)-assignment μ' is found, then μ' can be cached into an ad hoc data structure
- $\mathcal{L}\text{-SOLVE}(\mu)$ checks first if (any superset of) μ 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
  $\Longrightarrow$DPLL will never again generate an assignment containing C.
- May avoid a lot of redundant search.
- Problem: may cause a blowup in space
  $\Longrightarrow$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

$$\neg(3v_1 - 2v_2 \leq 3) \vee (2v_2 - v_3 > 2) \vee (3v_1 - v_3 \leq 6)$$

– Prunes up to 2^{N-3} assignments

$\Longrightarrow$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 $\mu \wedge l \wedge l'$ inconsistent, then $\mu \wedge l \models \neg l'$
- $assign(\varphi, l)$ substituted with $fc\_assign(\varphi, \mu \wedge l)$:
  $fc\_assign(\varphi, \mu \wedge l)$ replaces $cl \vee l'$ with cl if
  $\mathcal{L}$-$\mathrm{SOLVE}(\mu \wedge l \wedge l')$ returns false, for every l'
- can significantly prune search
- significant overhead: many possibly redundant calls to $\mathcal{L}$-SOLVE

Triggering [55, 4]

Proposition Let C be a non-boolean atom occurring only positively [resp. negatively] in φ. Let $\mathcal{M}$ be a complete set of assignments for φ, and let

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

Then φ is satisfiable if and only if there exist a satisfiable $\eta' \in \mathcal{M}'$ s.t. $\eta' \models_p \varphi$.

Triggering (cont.)

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

– 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: $(v_1 - v_2 > 3.2) \vee (v_1 - v_2 < 3.2)$.

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
- ...

Case study: Modal Logic(s)

Satisfiability in Modal logics

- Propositional logics enhanced with modal operators $\Box_i$, 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: $K(m) \Longleftrightarrow \mathcal{ALC}$)
- 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 $\{\neg, \wedge, \vee, \rightarrow, \leftrightarrow\}$ and the set of modal operators in $\mathcal{B}$.

– depth(φ) is the maximum number of nested modal operators in φ.
– "$\Box_i\varphi$" can be interpreted as "Agent i knows φ"

Semantics

– A Kripke structure for $\mathcal{L}$ is a tuple $M = <\mathcal{U}, \pi, \mathcal{R}_1, \ldots, \mathcal{R}_m>$, where

- $\mathcal{U}$ is a set of states $u_1, u_2, ...$
- π is a function $\pi : \mathcal{A} \times \mathcal{U} \longmapsto \{\top, \bot\}$,
- each $\mathcal{R}_r$ is a binary relation on the states of $\mathcal{U}$.

Semantics (cont)

Given M, u s.t. $u \in \mathcal{U}$, $M, u \models \varphi$ is defined as follows:

$$M, u \models A_i,\ A_i \in \mathcal{A} \iff \pi(A_i, u) = \top;$$

$$M, u \models \neg\varphi_1 \iff M, u \not\models \varphi_1;$$

$$M, u \models \varphi_1 \wedge \varphi_2 \iff M, u \models \varphi_1 \textit{ and } M, u \models \varphi_2;$$

$$M, u \models \varphi_1 \vee \varphi_2 \iff M, u \models \varphi_1 \textit{ or } M, u \models \varphi_2.$$

...

$M, u \models \Box_r\varphi_1,\ \Box_r \in \mathcal{B} \iff M, v \models \varphi_1$ for every $v \in \mathcal{U}$ s.t. $\mathcal{R}_r(u, v)$ holds in M.

$M, u \models \neg\Box_r\varphi_1,\ \Box_r \in \mathcal{B} \iff M, v \models \neg\varphi_1$ for some $v \in \mathcal{U}$ s.t. $\mathcal{R}_r(u, v)$ holds in M.

Semantics (cont)

The (normal) modal logics vary with the properties of $\mathcal{R}_r$:

Axiom	Property of $\mathcal{R}$	Description
B	symmetric	$\forall\, u\, v\, \mathcal{R}(u,v) \Longrightarrow \mathcal{R}(v,u)$
D	serial	$\forall\, u\, \exists\, v\, \mathcal{R}(u,v)$
T	reflexive	$\forall\, u\, \mathcal{R}(u,u)$
4	transitive	$\forall\, u\, v\, w\, \mathcal{R}(u,v)\ e\ \mathcal{R}(v,w) \Longrightarrow \mathcal{R}(u,w)$
5	euclidean	$\forall\, u\, v\, w\, \mathcal{R}(u,v)\ e\ \mathcal{R}(u,w) \Longrightarrow \mathcal{R}(v,w)$

Normal Modal Logic	Properties of $\mathcal{R}_r$
K	—
KB	symmetric
KD	serial
KT = KDT (T)	reflexive
K4	transitive
K5	euclidean
KBD	symmetric and serial
KBT = KBDT (B)	symmetric and reflexive
KB4 = KB5 = KB45	symmetric and transitive
KD4	serial and transitive
KD5	serial and euclidean
KT4 = KDT4 (S4)	reflexive and transitive
KT5 = KBD4 = KBD5 = KBT4 = KBT5 = KDT5 = KT45 = KBD45 = KBT45 = KDT45 = KBDT4 = KBDT5 = KBDT45 (S5)	reflexive, transitive and symmetric (equivalence)
K45	transitive and euclidean
KD45	serial, transitive and euclidean

Axiomatic framework

- Basic Axioms:

$$I. \quad \alpha \rightarrow (\beta \rightarrow \alpha),$$

$$II. \quad (\alpha \rightarrow (\beta \rightarrow \gamma)) \rightarrow ((\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \gamma)),$$

$$III. \quad (\neg\alpha \rightarrow \beta) \rightarrow ((\neg\alpha \rightarrow \neg\beta) \rightarrow \alpha),$$

$$K: \quad \Box_r\alpha \rightarrow (\Box_r(\alpha \rightarrow \beta) \rightarrow \Box_r\beta)$$

- Specific Axioms:

$$B. \quad \alpha \rightarrow \Box_r\neg\Box_r\neg\alpha,$$

$$D. \quad \Box_r\alpha \rightarrow \neg\Box_r\neg\alpha,$$

$$T. \quad \Box_r\alpha \rightarrow \alpha,$$

$$4. \quad \Box_r\alpha \rightarrow \Box_r\Box_r\alpha,$$

$$5. \quad \neg\Box_r\alpha \rightarrow \Box_r\neg\Box_r\alpha.$$

Axiomatic framework (cont.)

– Inference rules:

$$\frac{\alpha \quad \alpha \rightarrow \beta}{\beta} \text{ (modus ponens)},$$

$$\frac{\alpha}{\Box_r \alpha} \text{ (necessitation)}.$$

– Correctness & completeness:
φ is valid $\Longleftrightarrow$ φ can be deduced

Tableaux for modal K(m)/$\mathcal{ACL}$ [21]

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

$$\left\{ \begin{array}{ccc} \dfrac{\varphi_1 \wedge \varphi_2}{\begin{array}{c}\varphi_1\\ \varphi_2\end{array}} & \dfrac{\neg(\varphi_1 \vee \varphi_2)}{\begin{array}{c}\neg\varphi_1\\ \neg\varphi_2\end{array}} & \dfrac{\neg(\varphi_1 \rightarrow \varphi_2)}{\begin{array}{c}\varphi_1\\ \neg\varphi_2\end{array}} \\ & \dfrac{\neg\neg\varphi}{\varphi} & \\ \dfrac{\varphi_1 \vee \varphi_2}{\varphi_1 \qquad \varphi_2} & \dfrac{\neg(\varphi_1 \wedge \varphi_2)}{\neg\varphi_1 \qquad \neg\varphi_2} & \dfrac{\varphi_1 \rightarrow \varphi_2}{\neg\varphi_1 \qquad \varphi_2} \\ \dfrac{\varphi_1 \leftrightarrow \varphi_2}{\begin{array}{cc}\varphi_1 & \neg\varphi_1\\ \varphi_2 & \neg\varphi_2\end{array}} & \dfrac{\neg(\varphi_1 \leftrightarrow \varphi_2)}{\begin{array}{cc}\varphi_1 & \neg\varphi_1\\ \neg\varphi_2 & \varphi_2\end{array}} & \end{array} \right\} \cup \left\{ \dfrac{\Box_r\alpha_1, ..., \Box_r\alpha_N, \neg\Box_r\beta_j}{\alpha_1, ..., \alpha_N, \neg\beta_j} \right\}$$

DPLL for K(m)/$\mathcal{ALC}$: K-SAT [29, 30]

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

$$\left\{ \begin{array}{c} \dfrac{\varphi_1 \wedge (l) \wedge \varphi_2}{(\varphi_1 \wedge \varphi_2)[l|\top]} \; (Unit) \\ \\ \dfrac{\varphi}{\varphi[l|\top] \quad \varphi[l|\bot]} \; (split) \end{array} \right\} \cup \left\{ \dfrac{\Box_r\alpha_1, ..., \Box_r\alpha_N, \neg\Box_r\beta_j}{\alpha_1, ..., \alpha_N, \neg\beta_j} \right\}$$

The K-SAT algorithm [29, 30]

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

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

The K-SAT algorithm (cont.)

function K-SOLVE($\bigwedge_i \Box_1 \alpha_{1i} \wedge \bigwedge_j \neg\Box_1 \beta_{1j} \wedge \ldots \wedge \bigwedge_i \Box_m \alpha_{mi} \wedge \bigwedge_j \neg\Box_m \beta_{mj} \wedge \gamma$)
  for each box index r **do**
    if not K-SOLVE$_{restr}$($\bigwedge_i \Box_r \alpha_{ri} \wedge \bigwedge_j \neg\Box_r \beta_{rj}$)
      then return *False*;
  return *True*;

function K-SOLVE$_{restr}$($\bigwedge_i \Box_r \alpha_{ri} \wedge \bigwedge_j \neg\Box_r \beta_{rj}$)
  for each conjunct "$\neg\Box_r \beta_{rj}$" **do**
    if not K-SAT($\bigwedge_i \alpha_{ri} \wedge \neg\beta_{rj}$)
      then return *False*;
  return *True*;

K-SAT: Example

$$\begin{aligned}\varphi = \; & \{\neg\Box_1(\neg A_3 \vee \neg A_1 \vee A_2) \vee A_1 \vee A_5\} \wedge \\ & \{\neg A_2 \vee \neg A_5 \vee \Box_2(\neg A_2 \vee \neg A_4 \vee \neg A_3)\} \wedge \\ & \{A_1 \vee \Box_2(\neg A_4 \vee A_5 \vee A_2) \vee A_2\} \wedge \\ & \{\neg\Box_2(A_4 \vee \neg A_3 \vee A_1) \vee \neg\Box_1(A_4 \vee \neg A_2 \vee A_3) \vee \neg A_5\} \wedge \\ & \{\neg A_3 \vee A_1 \vee \Box_2(\neg A_4 \vee A_5 \vee A_2)\} \wedge \\ & \{\Box_1(\neg A_5 \vee A_4 \vee A_3) \vee \Box_1(\neg A_1 \vee A_4 \vee A_3) \vee \neg A_1\} \wedge \\ & \{A_1 \vee \Box_1(\neg A_2 \vee A_1 \vee A_4) \vee A_2\}\end{aligned}$$

$$\Downarrow \quad \text{K-SOLVE()}$$

$$\begin{aligned}\mu = \; & \Box_1(\neg A_5 \vee A_4 \vee A_3) \wedge \quad \Box_1(\neg A_2 \vee A_1 \vee A_4) \wedge && [\textstyle\bigwedge_i \Box_1 \alpha_{1i}] \\ & \neg\Box_1(\neg A_3 \vee \neg A_1 \vee A_2) \wedge \quad \neg\Box_1(A_4 \vee \neg A_2 \vee A_3) \wedge && [\textstyle\bigwedge_j \neg\Box_1 \beta_{1j}] \\ & \Box_2(\neg A_4 \vee A_5 \vee A_2) \wedge && [\textstyle\bigwedge_i \Box_2 \alpha_{2i}] \\ & \neg A_2. && [\gamma]\end{aligned}$$

K-SAT: Example (cont.)

$$
\begin{array}{rclll}
\mu & = & \Box_1(\neg A_5 \vee A_4 \vee A_3) \wedge & \Box_1(\neg A_2 \vee A_1 \vee A_4) \wedge & [\bigwedge_i \Box_1 \alpha_{1i}] \\
 & & \neg\Box_1(\neg A_3 \vee \neg A_1 \vee A_2) \wedge & \neg\Box_1(A_4 \vee \neg A_2 \vee A_3) \wedge & [\bigwedge_j \neg\Box_1 \beta_{1j}] \\
 & & \Box_2(\neg A_4 \vee A_5 \vee A_2) \wedge & & [\bigwedge_i \Box_2 \alpha_{2i}] \\
 & & \neg A_2. & & [\gamma]
\end{array}
$$

$$\Downarrow \ \text{K-SOLVE}_{restr}()$$

$$
\begin{array}{rclll}
\mu^1 & = & \Box_1(\neg A_5 \vee A_4 \vee A_3) \wedge & \Box_1(\neg A_2 \vee A_1 \vee A_4) \wedge & [\bigwedge_i \Box_1 \alpha_{1i}] \\
 & & \neg\Box_1(\neg A_3 \vee \neg A_1 \vee A_2) \wedge & \neg\Box_1(A_4 \vee \neg A_2 \vee A_3) & [\bigwedge_j \neg\Box_1 \beta_{1j}] \\
\mu^2 & = & \Box_2(\neg A_4 \vee A_5 \vee A_2) & & [\bigwedge_i \Box_2 \alpha_{2i}].
\end{array}
$$

$$\Downarrow \ \text{K-SAT}()$$

$$
\begin{array}{rcl}
\varphi^{11} & = & (\neg A_5 \vee A_4 \vee A_3) \wedge (\neg A_2 \vee A_1 \vee A_4) \wedge A_3 \wedge A_1 \wedge \neg A_2, \\
\varphi^{12} & = & (\neg A_5 \vee A_4 \vee A_3) \wedge (\neg A_2 \vee A_1 \vee A_4) \wedge \neg A_4 \wedge A_2 \wedge \neg A_3
\end{array}
$$

K-SAT: Example (cont.)

$$\varphi^{11} = (\neg A_5 \vee A_4 \vee A_3) \wedge (\neg A_2 \vee A_1 \vee A_4) \wedge A_3 \wedge A_1 \wedge \neg A_2,$$

$$\varphi^{12} = (\neg A_5 \vee A_4 \vee A_3) \wedge (\neg A_2 \vee A_1 \vee A_4) \wedge \neg A_4 \wedge A_2 \wedge \neg A_3$$

$$\Downarrow \quad \text{K-SOLVE()}$$

$$\mu^{11} = A_3 \wedge A_1 \wedge \neg A_2$$

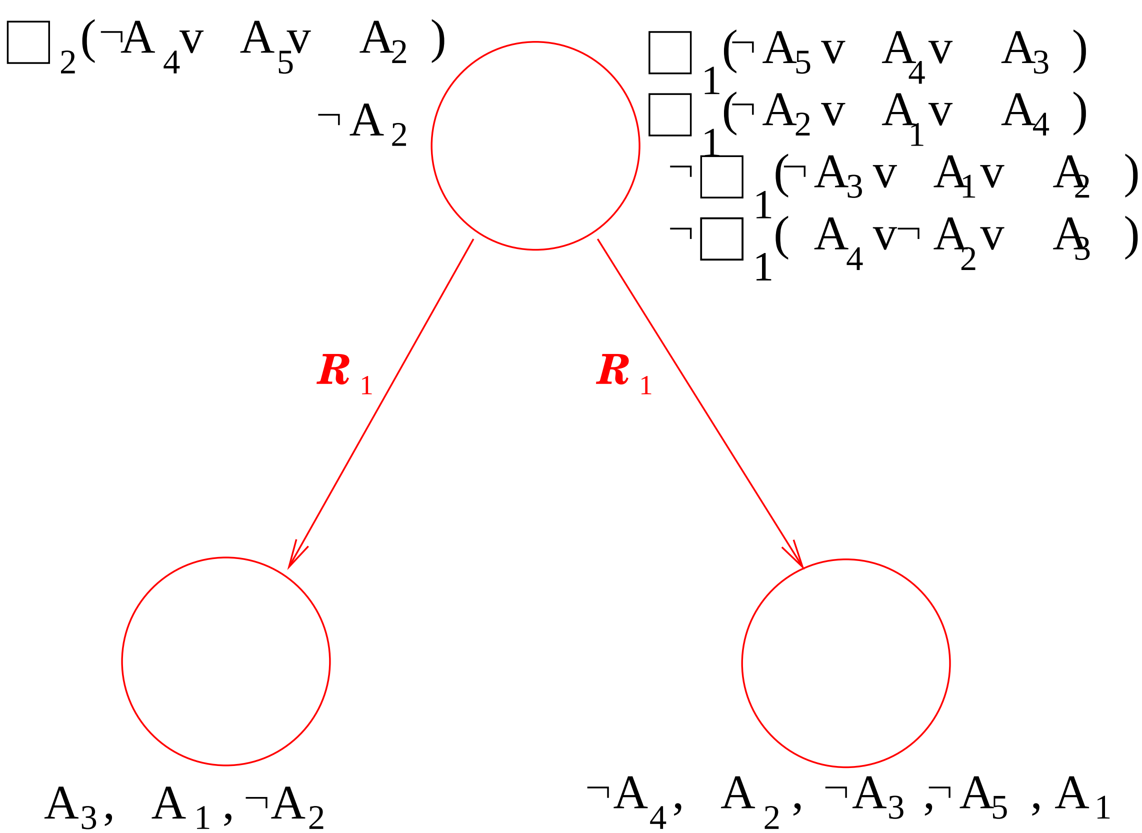
$$\mu^{12} = \neg A_4 \wedge A_2 \wedge \neg A_3 \wedge \neg A_5 \wedge A_1$$

$$\Downarrow$$

$$Satisfiable$$

Example

Resulting Kripke Model:

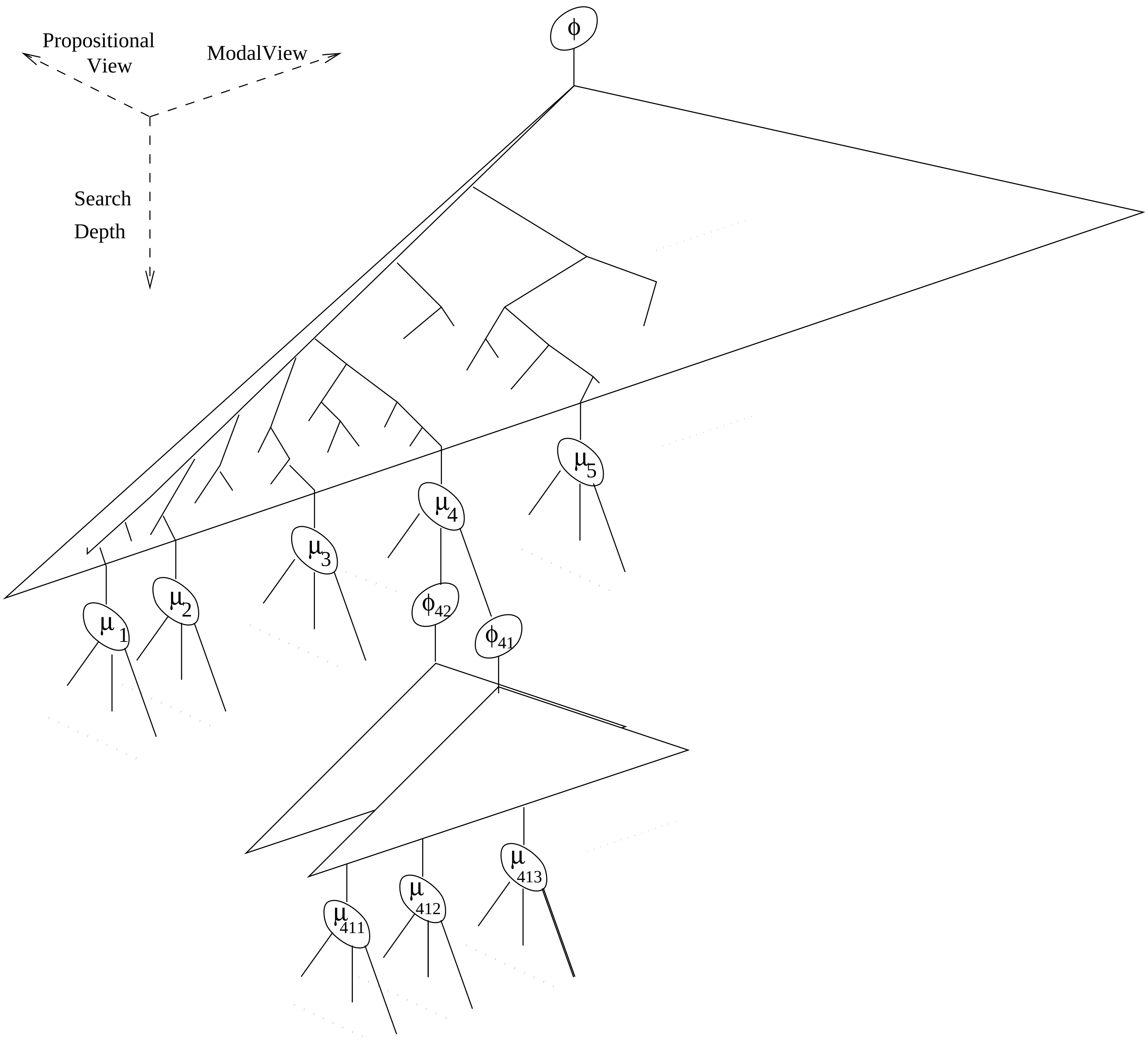


Search in modal logic:

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


Propositional View
ModalView
Search Depth
ϕ
μ_5
μ_4
μ_3
μ_2
μ_1
ϕ_{42}
ϕ_{41}
μ_{413}
μ_{412}
μ_{411}

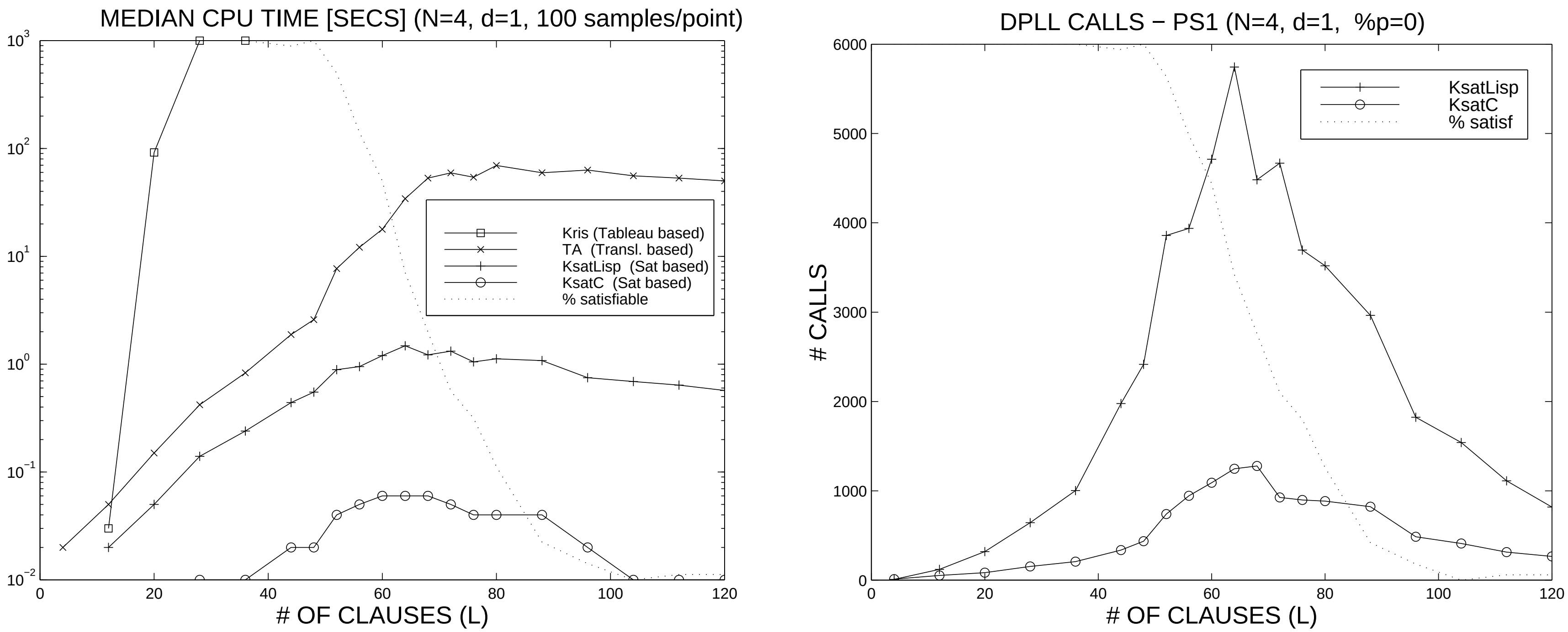
Some Systems

- 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]

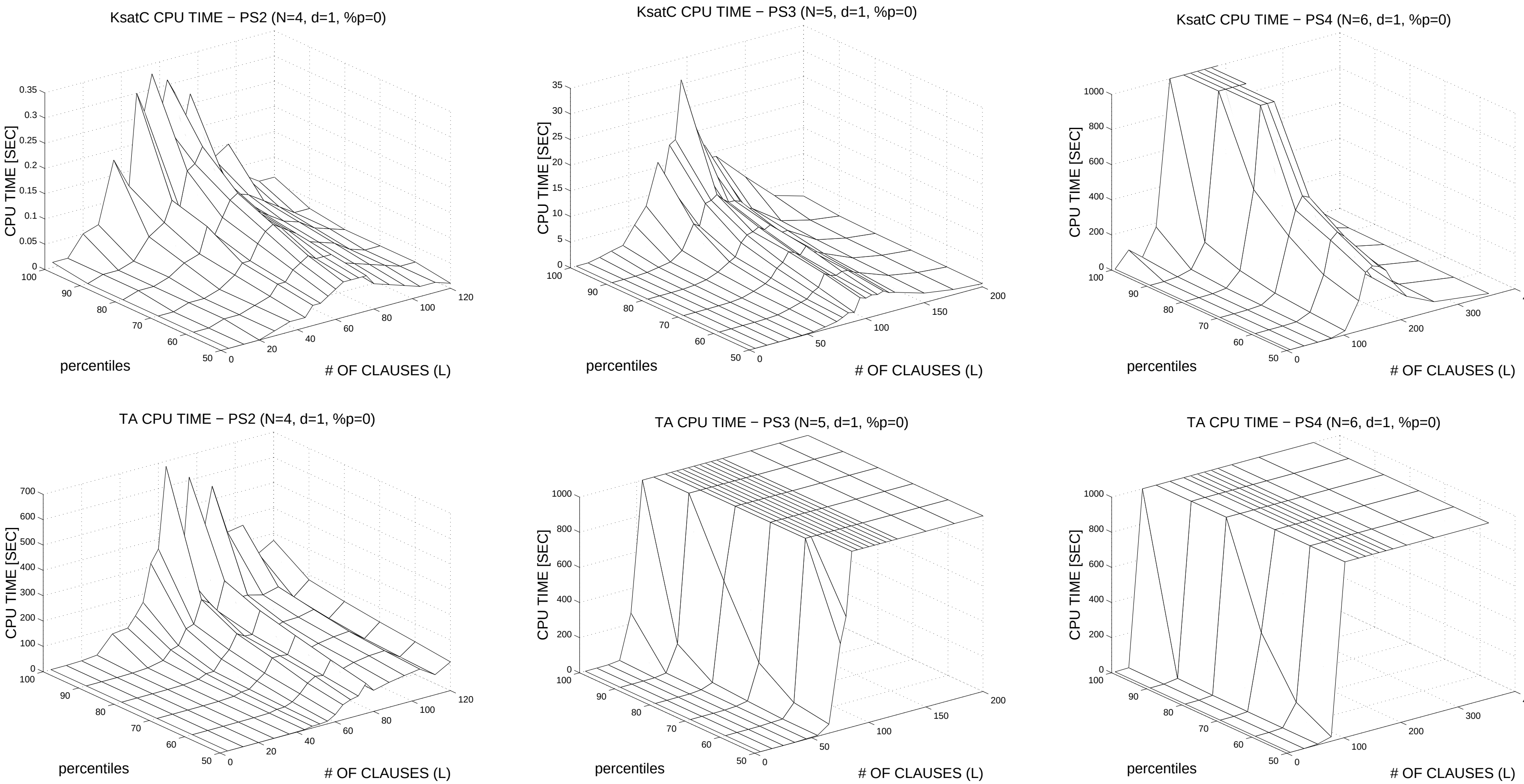


Left: KRIS, TA, K-SAT (LISP), K-SAT (C) median CPU time, 100 samples/point.

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]

	branch		*d4*		*dum*		*grz*		*lin*		*path*		*ph*		*poly*		*t4p*	
K	*p*	*n*	*p*	*n*	*p*	*n*	*p*	*n*	*p*	*n*	*p*	*n*	*p*	*n*	*p*	*n*	*p*	*n*
leanK 2.0	1	0	1	1	0	0	0	$\geq$21	$\geq$21	4	2	0	3	1	2	0	0	0
□KE	13	3	13	3	4	4	3	1	$\geq$21	2	17	5	4	3	17	0	0	3
LWB 1.0	6	7	8	6	13	19	7	13	11	8	12	10	4	8	8	11	8	7
TA	9	9	$\geq$21	18	$\geq$21	$\geq$21	$\geq$21	$\geq$21	$\geq$21	$\geq$21	20	20	6	9	16	17	$\geq$21	19
***SAT 1.2**	$\geq$21	12	$\geq$21	$\geq$21	$\geq$21	$\geq$21	$\geq$21	$\geq$21	$\geq$21	$\geq$21	$\geq$21	$\geq$21	8	12	$\geq$21	$\geq$21	$\geq$21	$\geq$21
Crack 1.0	2	1	2	3	3	$\geq$21	1	$\geq$21	5	2	2	6	2	3	$\geq$21	$\geq$21	1	1
Kris	3	3	8	6	15	$\geq$21	13	$\geq$21	6	9	3	11	4	5	11	$\geq$21	7	5
Fact 1.2	6	4	$\geq$21	8	$\geq$21	$\geq$21	$\geq$21	$\geq$21	$\geq$21	$\geq$21	7	6	6	7	$\geq$21	$\geq$21	$\geq$21	$\geq$21
DLP 3.1	19	13	$\geq$21	$\geq$21	$\geq$21	$\geq$21	$\geq$21	$\geq$21	$\geq$21	$\geq$21	$\geq$21	$\geq$21	7	9	$\geq$21	$\geq$21	$\geq$21	$\geq$21

	45		*branch*		*dum*		*grz*		*md*		*path*		*ph*		*poly*		*t4p*	
KT	*p*	*n*	*p*	*n*	*p*	*n*	*p*	*n*	*p*	*n*	*p*	*n*	*p*	*n*	*p*	*n*	*p*	*n*
TA	17	6	13	9	17	9	≥ 21	≥ 21	16	20	≥ 21	16	5	12	≥ 21	1	11	0
Kris	4	3	3	3	3	14	0	5	3	4	1	13	3	3	2	2	1	7
FaCT 1.2	≥ 21	≥ 21	6	4	11	≥ 21	≥ 21	≥ 21	4	5	5	3	6	7	≥ 21	7	4	2
DLP 3.1	≥ 21	≥ 21	19	12	≥ 21	≥ 21	≥ 21	≥ 21	3	≥ 21	16	14	7	≥ 21	≥ 21	12	≥ 21	≥ 21

	45		*branch*		*dum*		*grz*		*md*		*path*		*ph*		*poly*		*t4p*	
S4	*p*	*n*	*p*	*n*	*p*	*n*	*p*	*n*	*p*	*n*	*p*	*n*	*p*	*n*	*p*	*n*	*p*	*n*
KT4	1	6	2	3	0	17	5	8	≥ 21	18	1	2	2	2	2	2	0	3
leanS4 2.0	0	0	0	0	0	0	1	1	2	2	1	0	1	0	1	1	0	0
$\Box$KE	8	0	≥ 21	≥ 21	0	≥ 21	6	4	3	3	9	6	4	3	1	≥ 21	3	1
LWB 1.0	3	5	11	7	9	≥ 21	8	7	8	6	8	6	4	8	4	9	9	12
TA	9	0	≥ 21	4	14	0	6	≥ 21	9	10	15	≥ 21	5	5	≥ 21	1	11	0
FaCT 1.2	≥ 21	≥ 21	4	4	2	≥ 21	5	4	8	4	2	1	5	4	≥ 21	2	5	3
DLP 3.1	≥ 21	≥ 21	18	12	≥ 21	≥ 21	10	≥ 21	3	≥ 21	15	15	7	≥ 21	≥ 21	≥ 21	≥ 21	≥ 21

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 $\mathcal{OP}_{\mathcal{D}}$ of arithmetical operators.

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

Syntax: math-terms and math-formulas

– a constant $c_i \in \mathbb{R}[\mathbb{Z}]$ is a math-term;

– a variable v_i over $\mathbb{R}[\mathbb{Z}]$ 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 $\mathbb{R}[\mathbb{Z}]$;

– 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 $(t_1 \bowtie t_2)$ is a math-formula, $\bowtie \in \{=, \neq, >, <, \geq, \leq\}$;

– if φ_1, φ_2 are math-formulas, then $\neg\varphi_1$, $(\varphi_1 \wedge \varphi_2)$, $(\varphi_1 \vee \varphi_2)$, $(\varphi_1 \to \varphi_2)$ 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 $\otimes \in \{+, -, \cdot\}$;
- $\mathcal{I}(t_1 \bowtie t_2) = \mathcal{I}(t_1) \bowtie \mathcal{I}(t_2)$, for all math-terms t_1, t_2 and $\bowtie \in \{=, \neq, >, <, \geq, \leq\}$;
- $\mathcal{I}(\neg\varphi_1) = \neg\mathcal{I}(\varphi_1)$, for every math-formula φ_1;
- $\mathcal{I}(\varphi_1 \wedge \varphi_2) = \mathcal{I}(\varphi_1) \wedge \mathcal{I}(\varphi_2)$, for all math-formulas φ_1,φ_2.

DPLL for math-formulas [55, 2, 4, 5]

```
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

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 $((x < y) \vee (y < x))$ 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

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

[2] A. Armando, C. Castellini, and E. Giunchiglia. SAT-based procedures for temporal reasoning. In *Proc. European Conference on Planning, CP-99*, 1999.

[3] A. Armando and E. Giunchiglia. Embedding Complex Decision Procedures inside an Interactive Theorem Prover. *Annals of Mathematics and Artificial Intelligence*, 8(3–4):475–502, 1993.

[4] G. Audemard, P. Bertoli, A. Cimatti, A. Korniłowicz, and R. Sebastiani. A SAT Based Approach for Solving Formulas over Boolean and Linear Mathematical Propositions. In *Proc. CADE'2002.*, LNAI. Springer Verlag, July 2002.

[5] G. Audemard, P. Bertoli, A. Cimatti, A. Korniłowicz, and R. Sebastiani. Integrating Boolean and Mathematical Solving: Foundations, Basic Algorithms and Requirements. In *Proc. CALCULEMUS'2002.*, LNAI. Springer Verlag, 2002. To appear.

[6] G. Audemard, A. Cimatti, A. Korniłowicz, and R. Sebastiani. SAT-Based Bounded Model Checking for Timed Systems. In *Proc. FORTE'02.*, LNCS. Springer Verlag, January 2002. To appear.

[7] F. Baader, E. Franconi, B. Hollunder, B. Nebel, and H.J. Profitlich. An Empirical Analysis of Optimization Techniques for Terminological Representation Systems or: Making KRIS get a move on. *Applied Artificial Intelligence. Special Issue on Knowledge Base Management*, 4:109–132, 1994.

[8] R. J. Bayardo, Jr. and R. C. Schrag. Using CSP Look-Back Techniques to Solve Real-World SAT instances. In *American Association for Artificial Intelligence*, pages 203–208. AAAI Press, 1997.

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

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

[11] P. Bresciani, E. Franconi, and S. Tessaris. Implementing and testing expressive Description Logics: a preliminary report. In *Proc. International Workshop on Description Logics*, Rome, Italy, 1995.

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

[13] W. Chan, R. J. Anderson, P. Beame, and D. Notkin. Combining constraint solving and symbolic model checking for a class of systems with non-linear constraints. In *Proc. CAV'97*, volume 1254 of *LNCS*, pages 316–327, Haifa, Israel, June 1997. Springer-Verlag.

[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.

[15] S. A. Cook. The complexity of theorem proving procedures. In *3rd Annual ACM Symposium on the Theory of Computation*, pages 151–158, 1971.

[16] M. D'Agostino and M. Mondadori. The Taming of the Cut. *Journal of Logic and Computation*, 4(3):285–319, 1994.

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

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

[19] T. Boy de la Tour. Minimizing the Number of Clauses by Renaming. In *Proc. of the 10th Conference on Automated Deduction*, pages 558–572. Springer-Verlag, 1990.

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

[21] M. Fitting. First-Order Modal Tableaux. *Journal of Automated Reasoning*, 4:191–213, 1988.

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

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

search. In *Proceedings of AAAI-96*, pages 246–252, Menlo Park, 1996. AAAI Press / MIT Press.

[24] E. Giunchiglia, F. Giunchiglia, R. Sebastiani, and A. Tacchella. SAT vs. Translation based decision procedures for modal logics: a comparative evaluation. *Journal of Applied Non-Classical Logics*, 10(2):145–172, 2000.

[25] E. Giunchiglia, F. Giunchiglia, and A. Tacchella. SAT Based Decision Procedures for Classical Modal Logics. Journal of Automated Reasoning. Special Issue: Satisfiability at the start of the year 2000, 2001.

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

[27] E. Giunchiglia, M. Narizzano, A. Tacchella, and M. Vardi. Towards an Efficient Library for SAT: a Manifesto. In *Proc. SAT 2001*, Electronics Notes in Discrete Mathematics. Elsevier Science., 2001.

[28] E. Giunchiglia and R. Sebastiani. Applying the Davis-Putnam procedure to non-clausal formulas. In *Proc. AI*IA'99*, number 1792 in LNAI. Springer Verlag, 1999.

[29] F. Giunchiglia and R. Sebastiani. Building decision procedures for modal logics from propositional decision procedures - the case study of modal K. In *Proc. CADE'13*, LNAI, New Brunswick, NJ, USA, August 1996. Springer Verlag.

[30] F. Giunchiglia and R. Sebastiani. A SAT-based decision procedure for ALC. In

Proc. of the 5th International Conference on Principles of Knowledge Representation and Reasoning - KR'96, Cambridge, MA, USA, November 1996.

[31] F. Giunchiglia and R. Sebastiani. Building decision procedures for modal logics from propositional decision procedures - the case study of modal K(m). *Information and Computation*, 162(1/2), October/November 2000.

[32] J. Y. Halpern. The effect of bounding the number of primitive propositions and the depth of nesting on the complexity of modal logic. *Artificial Intelligence*, 75(3):361–372, 1995.

[33] J.Y. Halpern and Y. Moses. A guide to the completeness and complexity for modal logics of knowledge and belief. *Artificial Intelligence*, 54(3):319–379, 1992.

[34] A. Heuerding and S. Schwendimann. A benchmark method for the propositional modal logics K, KT, S4. Technical Report IAM-96-015, University of Bern, Switzerland, 1996.

[35] I. Horrocks and P. F. Patel-Schneider. FaCT and DLP. In *Proc. Tableaux'98*, pages 27–30, 1998.

[36] I. Horrocks, P. F. Patel-Schneider, and R. Sebastiani. An Analysis of Empirical Testing for Modal Decision Procedures. *Logic Journal of the IGPL*, 8(3):293–323, May 2000.

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

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

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

[40] S. A. Kripke. Semantical considerations on modal logic. In *Proc. A colloquium on Modal and Many-Valued Logics*, Helsinki, 1962.

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

[42] J. Moeller, J. Lichtenberg, H. R. Andersen, and H. Hulgaard. Fully symbolic model checking of timed systems using difference decision diagrams. In *Proc. Workshop on Symbolic Model Checking (SMC), Federated Logic Conference (FLoC)*, Trento, Italy, July 1999.

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

[44] D.A. Plaisted and S. Greenbaum. A Structure-preserving Clause Form Translation. *Journal of Symbolic Computation*, 2:293–304, 1986.

[45] K. D. Schild. A correspondence theory for terminological logics: preliminary report. In *Proc. of the 12th International Joint Conference on Artificial Intelligence*, pages 466–471, Sydney, Australia, 1991.

[46] R. Sebastiani. Applying GSAT to Non-Clausal Formulas. *Journal of Artificial Intelligence Research*, 1:309–314, 1994. Also DIST-Technical Report 94-0018, DIST, University of Genova, Italy.

[47] R. Sebastiani and A. Villafiorita. SAT-based decision procedures for normal modal logics: a theoretical framework. In *Proc. 6th International Conference on Artificial Intelligence: Methodology, Systems, Applications - AIMSA'98*, number 1480 in LNAI, Sozopol, Bulgaria, September 1998. Springer Verlag.

[48] B. Selman and H. Kautz. Domain-Independent Extension to GSAT: Solving Large Structured Satisfiability Problems. In *Proc. of the 13th International Joint Conference on Artificial Intelligence*, pages 290–295, 1993.

[49] B. Selman, H. Levesque., and D. Mitchell. A New Method for Solving Hard Satisfiability Problems. In *Proc. of the 10th National Conference on Artificial Intelligence*, pages 440–446, 1992.

[50] Ofer Shtrichmann. Tuning SAT checkers for bounded model checking. In *Proc. CAV00*, volume 1855 of *LNCS*, pages 480–494, Berlin, 2000. Springer.

[51] J. P. M. Silva and K. A. Sakallah. GRASP - a new search algorithm for satisfiability. Technical report, University of Michigan, 1996.

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

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

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

[55] S. Wolfman and D. Weld. The LPSAT Engine & its Application to Resource Planning. In *Proc. IJCAI*, 1999.

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

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

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