4th International Seminar on New Issues in Artificial Intelligence

Madrid, Jan. 31st - Feb. 4th 2011

EFFICIENT BOOLEAN REASONING: SAT, PREFERENCES & QBFs

PROPOSITIONAL SATISFIABILITY (SAT)

Enrico Giunchiglia

DIST, University of Genoa, Italy

giunchiglia@unige.it

http://www.star.dist.unige.it/~enrico

(Most of the) Slides by: Roberto Sebastiani

http://disi.unitn.it/~rseba

Last update: February 1, 2011.

Motivations

▷ Last ten years: impressive advance in Boolean reasoning techniques

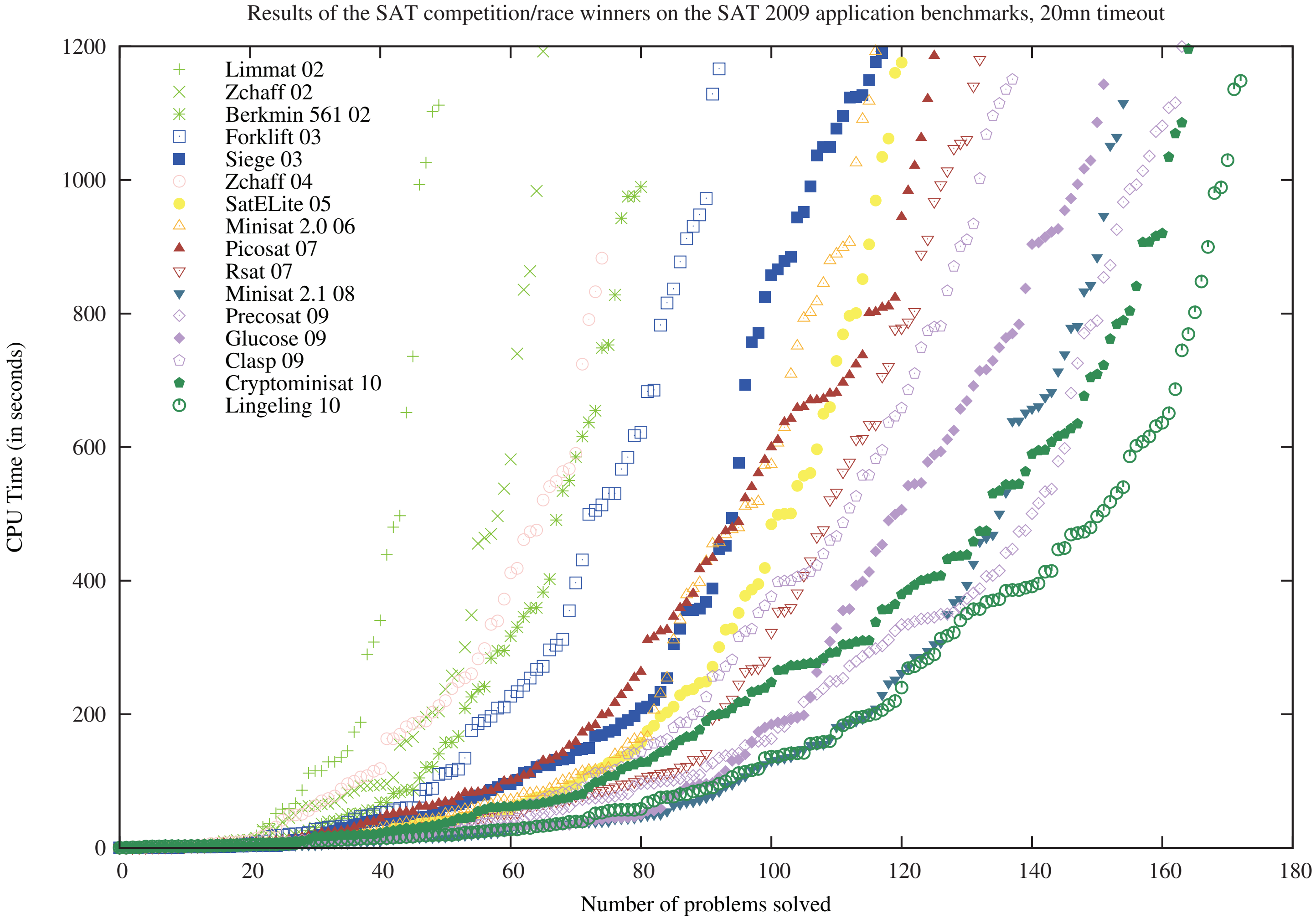
- extremely efficient solvers [52, 46, 4, 29, 36, 55, 23]
- hard "real-world" problems encoded into SAT (e.g.,
  - planning
  - model checking
  - circuit and software testing
  - security & criptanalysis
  - reasoning on conceptual models
  - bioinformatics
  - feature extraction from images
  - ...

Motivations

Application benchmarks submitted to the last SAT competition (2009):

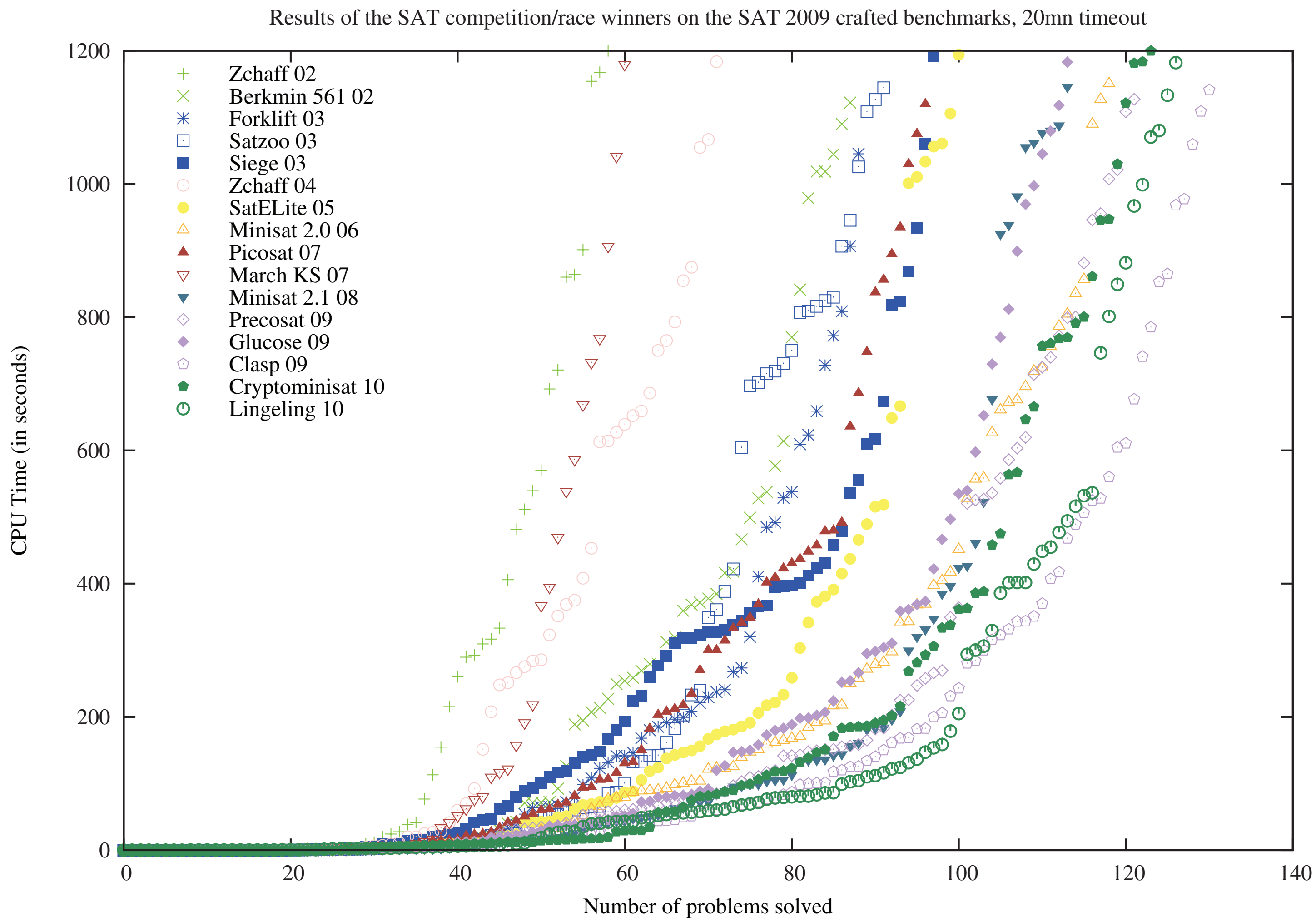
1. Aprove: Term Rewriting systems benchmarks.
2. BioInfo I: Queries to nd the maximal size of a biological behavior without cycles in discrete genetic networks.
3. BioInfo I I: Evolutionary trees.
4. Bit Verif: Bit precise software verication generated by the SMT solver Boolector.
5. C32SAT: Software verication generated by the C32SAT satisability checker for C programs.
6. Crypto: Encode attacks for both the DES and MD5 crypto systems.
7. Diagnosis: 4 dierent encodings of discrete event systems.

Motivations



Courtesy by Daniel Le Berre

Motivations



Courtesy by Daniel Le Berre

Content

Basic notation & definitions

▷ Boolean formula

- $\top, \bot$ are formulas
- A propositional atom $A_1, A_2, A_3, \ldots$ is a formula;
- if φ_1 and φ_2 are formulas, then $\neg\varphi_1$, $\varphi_1 \wedge \varphi_2$, $\varphi_1 \vee \varphi_2$, $\varphi_1 \rightarrow \varphi_2$, $\varphi_1 \leftrightarrow \varphi_2$ are formulas.

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

▷ N.B.: if $l := \neg A_i$, then $\neg l := A_i$

▷ $Atoms(\varphi)$: the set $\{A_1, \ldots, A_N\}$ of atoms occurring in φ.

▷ a Boolean formula can be represented as a tree or as a DAG

TREE and DAG representation of formulas: example

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

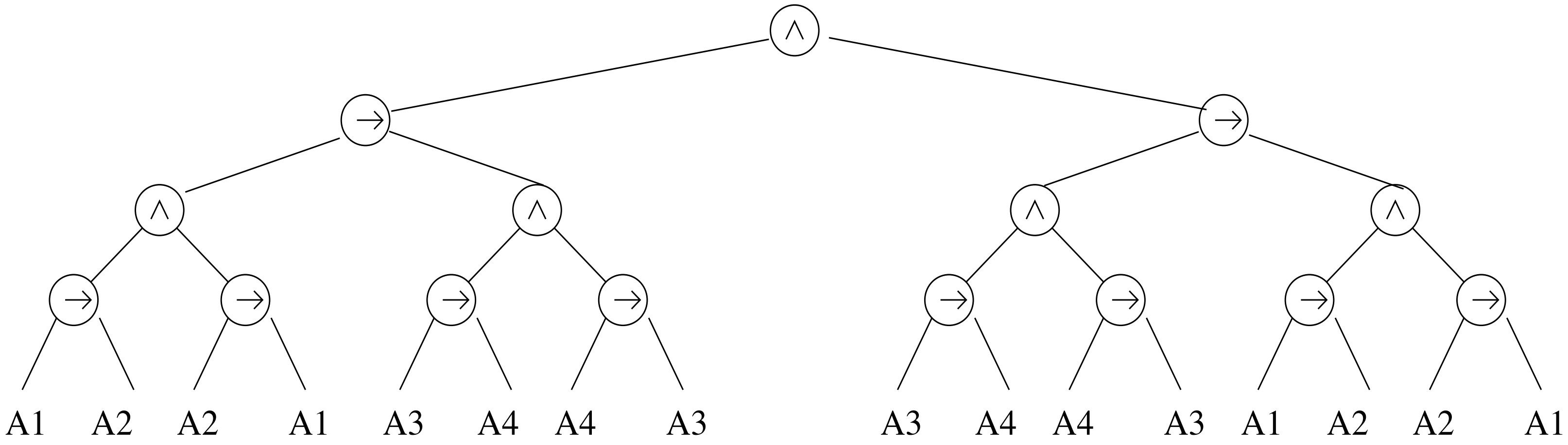
$$\Downarrow$$

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

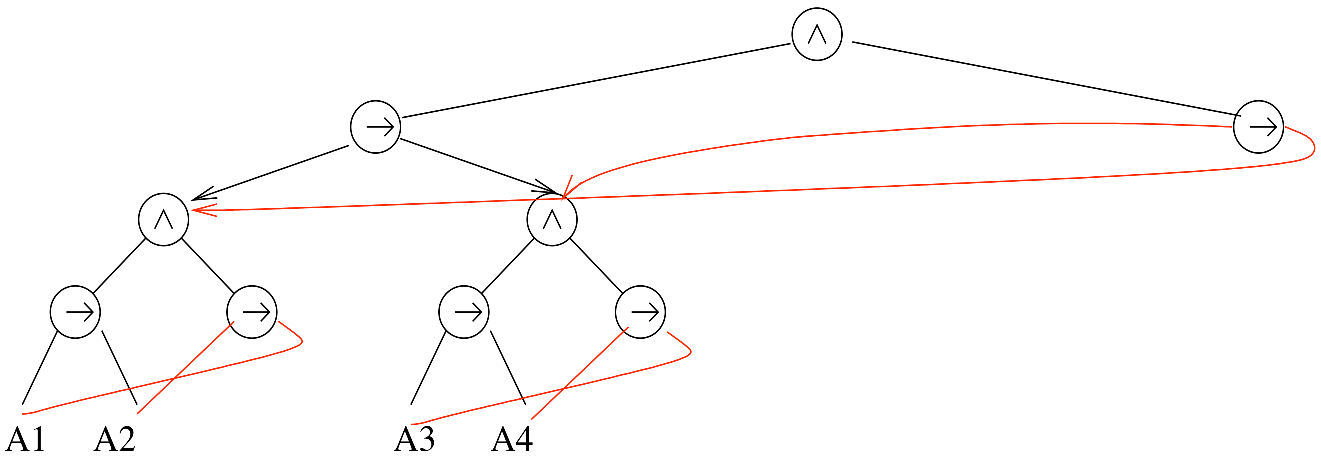
$$\Downarrow$$

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

TREE and DAG representation of formulas: example (cont)



Tree Representation



DAG Representation

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)

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

N.B.:

$\varphi_1 \vee \varphi_2 := \neg(\neg\varphi_1 \wedge \neg\varphi_2)$,

$\varphi_1 \rightarrow \varphi_2 := (\neg\varphi_1 \vee \varphi_2)$,

$\varphi_1 \leftrightarrow \varphi_2 := (\varphi_1 \rightarrow \varphi_2) \wedge (\varphi_2 \rightarrow \varphi_1)$.

Basic notation & definitions (cont)

▷ $\mu \models \varphi$ (μ satisfies φ):

- $\mu \models A_i \Longleftrightarrow \mu(A_i) = \top$
- $\mu \models \neg\varphi \Longleftrightarrow not\ \mu \models \varphi$
- $\mu \models \varphi_1 \wedge \varphi_2 \Longleftrightarrow \mu \models \varphi_1\ and\ \mu \models \varphi_2$
- $\mu \models \varphi_1 \vee \varphi_2 \Longleftrightarrow \mu \models \varphi_1\ or\ \mu \models \varphi_2$
- $\mu \models \varphi_1 \rightarrow \varphi_2 \Longleftrightarrow\ if\ \mu \models \varphi_1,\ then\ \mu \models \varphi_2$
- $\mu \models \varphi_1 \leftrightarrow \varphi_2 \Longleftrightarrow \mu \models \varphi_1\ iff\ \mu \models \varphi_2$

▷ 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 $\Longleftrightarrow$ $\neg\varphi$ 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 $(\varphi_1 \vee \neg A_3) \wedge (A_3 \vee \varphi_2)$ are in general equi-satisfiable but not equivalent.

Complexity

- For N variables, there are up to 2^N truth assignments to be checked.
- The problem of deciding the satisfiability of a propositional formula is NP-complete [10].
- 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.

Content

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:

$$\varphi_1 \rightarrow \varphi_2 \implies \neg\varphi_1 \vee \varphi_2$$

$$\varphi_1 \leftrightarrow \varphi_2 \implies (\neg\varphi_1 \vee \varphi_2) \wedge (\varphi_1 \vee \neg\varphi_2)$$

  2. pushing down negations recursively:

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

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

$$\neg\neg\varphi_1 \implies \varphi_1$$

- Preserves the equivalence of formulas.

NNF: example

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

$$\Downarrow$$

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

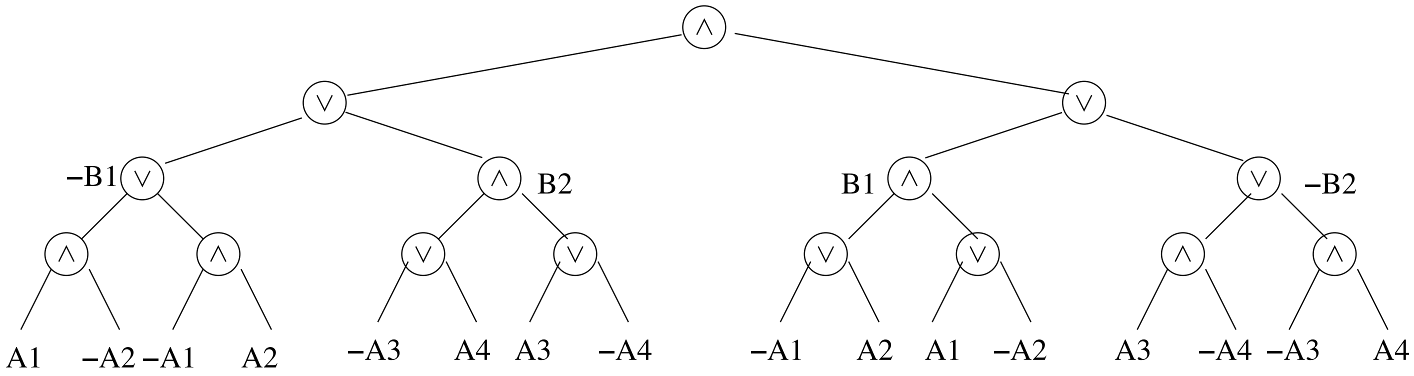
$$\Downarrow$$

$$((\neg((\neg A_1 \vee A_2) \wedge (A_1 \vee \neg A_2)) \vee ((\neg A_3 \vee A_4) \wedge (A_3 \vee \neg A_4)))\wedge$$
$$(((\neg A_1 \vee A_2) \wedge (A_1 \vee \neg A_2)) \vee \neg((\neg A_3 \vee A_4) \wedge (A_3 \vee \neg A_4))))$$

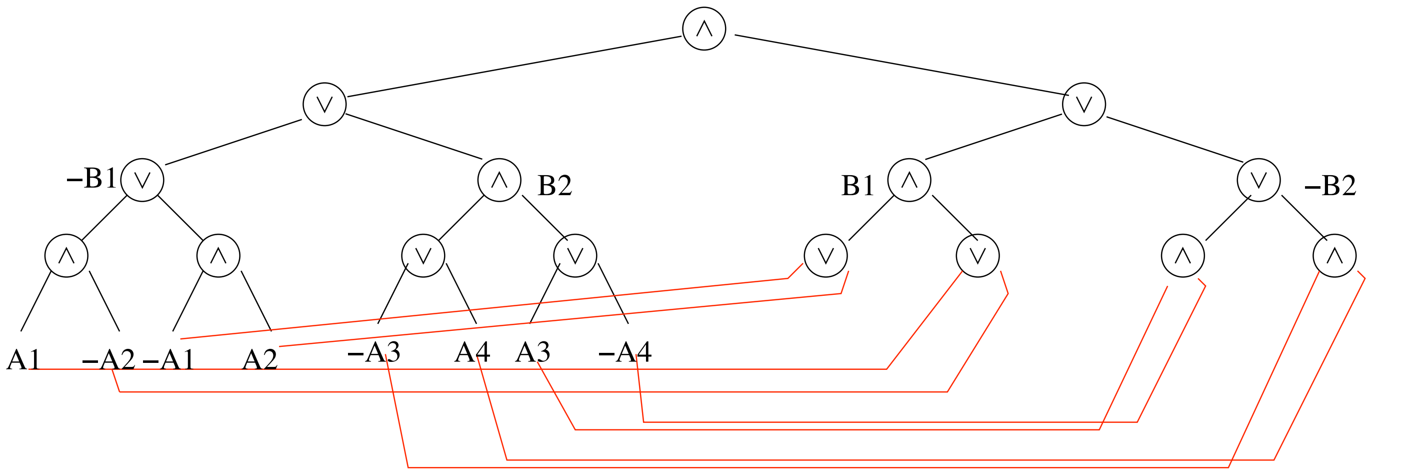
$$\Downarrow$$

$$((((A_1 \wedge \neg A_2) \vee (\neg A_1 \wedge A_2)) \vee ((\neg A_3 \vee A_4) \wedge (A_3 \vee \neg A_4)))\wedge$$
$$(((\neg A_1 \vee A_2) \wedge (A_1 \vee \neg A_2)) \vee ((A_3 \wedge \neg A_4) \vee (\neg A_3 \wedge A_4))))$$

NNF: example (cont)



Tree Representation



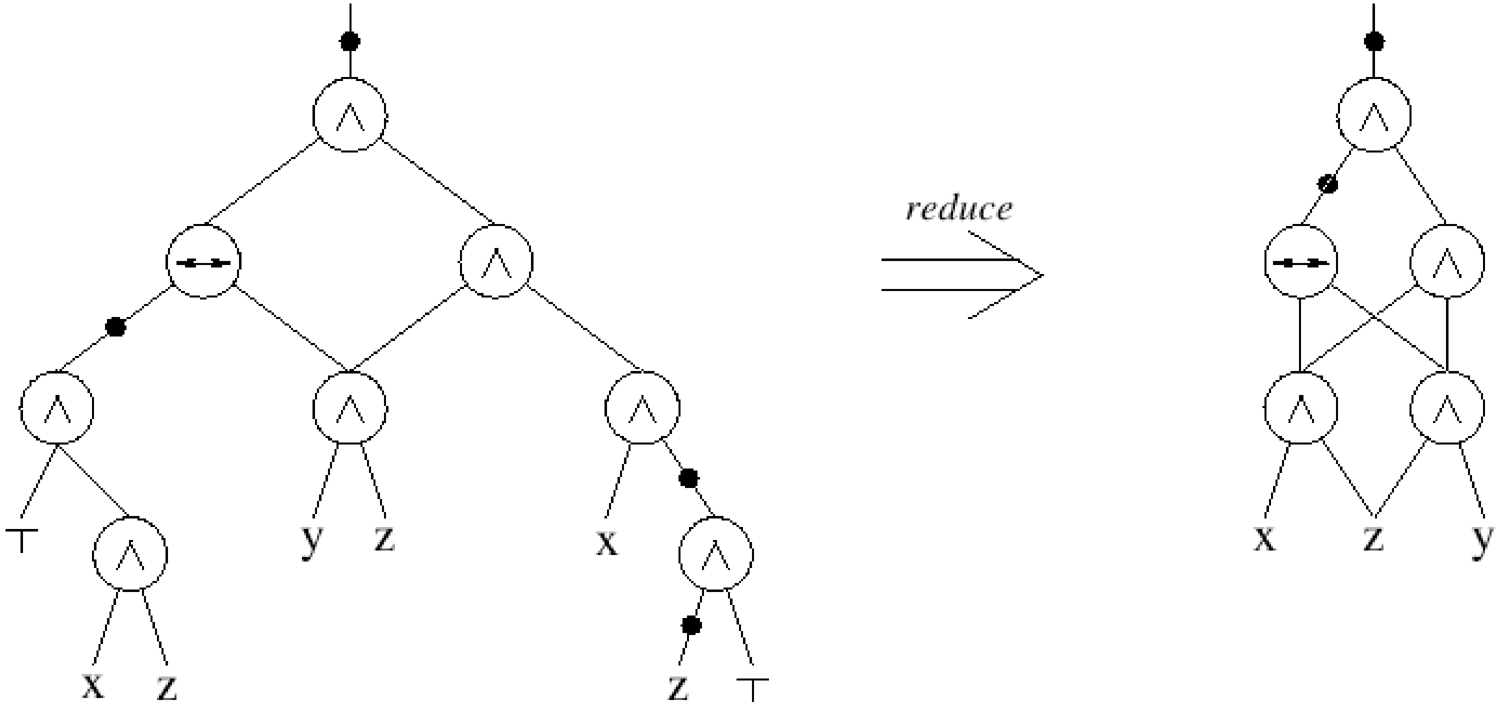
DAG Representation

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

Optimized polynomial representations

Reduced Boolean Circuits [1], Boolean Expression Diagrams [51].

▷ Maximize the sharing in DAG representations:
{∧, ↔, ¬}-only, negations on arcs, sorting of subformulae, lifting of ¬'s over ↔'s,...



Conjunctive Normal Form (CNF)

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

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

- ▷ the disjunctions of literals $\bigvee_{j_i=1}^{K_i} l_{j_i}$ are called clauses
- ▷ Easier to handle: list of lists of literals.

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

Classic CNF Conversion $CNF(\varphi)$

- Every φ can be reduced into CNF by, e.g.,
  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.
- Rarely used in practice.

Labeling CNF conversion $CNF_{label}(\varphi)$ [39, 13]

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

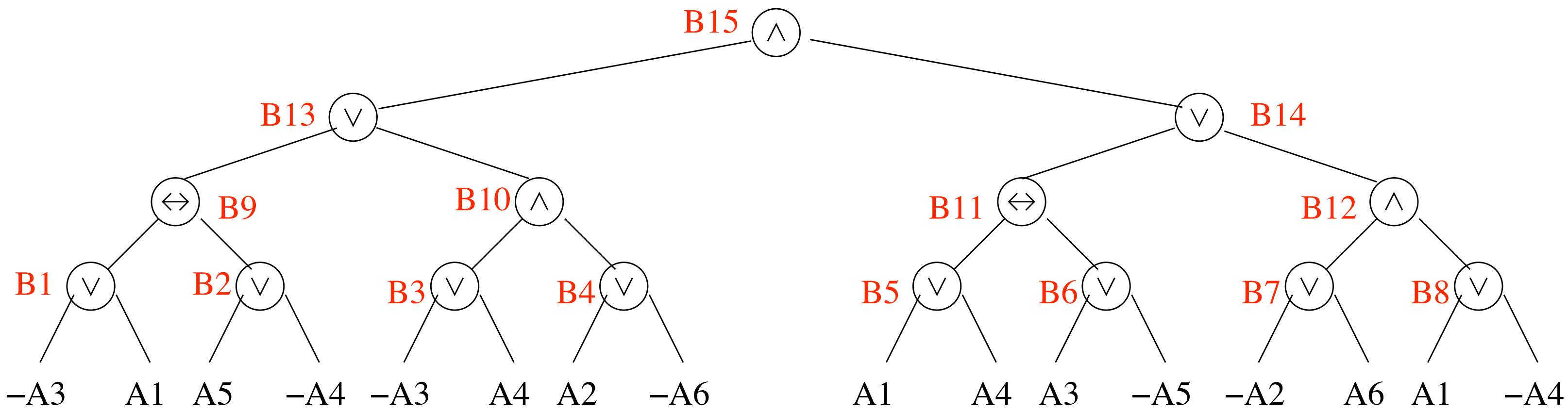
$$\varphi \implies \varphi[(l_i \vee l_j)|B] \wedge CNF(B \leftrightarrow (l_i \vee l_j))$$
$$\varphi \implies \varphi[(l_i \wedge l_j)|B] \wedge CNF(B \leftrightarrow (l_i \wedge l_j))$$
$$\varphi \implies \varphi[(l_i \leftrightarrow l_j)|B] \wedge CNF(B \leftrightarrow (l_i \leftrightarrow l_j))$$

  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 $CNF_{label}(\varphi)$ (cont.)

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

Labeling CNF conversion CNF_{label} – example



$CNF(B_1 \leftrightarrow (\neg A_3 \vee A_1)) \quad \wedge$

$\ldots \quad \wedge$

$CNF(B_8 \leftrightarrow (A_1 \vee \neg A_4)) \quad \wedge$

$CNF(B_9 \leftrightarrow (B_1 \leftrightarrow B_2)) \quad \wedge$

$\ldots \quad \wedge$

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

$CNF(B_{13} \leftrightarrow (B_9 \vee B_{10})) \quad \wedge$

$CNF(B_{14} \leftrightarrow (B_{11} \vee B_{12})) \quad \wedge$

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

B_{15}

Labeling CNF conversion CNF_{label} (improved)

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

$$\varphi \implies \varphi[(l_i \vee l_j)|B] \quad \wedge\, CNF(B \rightarrow (l_i \vee l_j)) \quad if\ (l_i \vee l_j)\ pos.$$
$$\varphi \implies \varphi[(l_i \vee l_j)|B] \quad \wedge\, CNF((l_i \vee l_j) \rightarrow B) \quad if\ (l_i \vee l_j)\ neg.$$
$$\varphi \implies \varphi[(l_i \wedge l_j)|B] \quad \wedge\, CNF(B \rightarrow (l_i \wedge l_j)) \quad if\ (l_i \wedge l_j)\ pos.$$
$$\varphi \implies \varphi[(l_i \wedge l_j)|B] \quad \wedge\, CNF((l_i \wedge l_j) \rightarrow B) \quad if\ (l_i \wedge l_j)\ neg.$$
$$\varphi \implies \varphi[(l_i \leftrightarrow l_j)|B] \quad \wedge\, CNF(B \rightarrow (l_i \leftrightarrow l_j)) \quad if\ (l_i \leftrightarrow l_j)\ pos.$$
$$\varphi \implies \varphi[(l_i \leftrightarrow l_j)|B] \quad \wedge\, CNF((l_i \leftrightarrow l_j) \rightarrow B) \quad if\ (l_i \leftrightarrow l_j)\ neg.$$

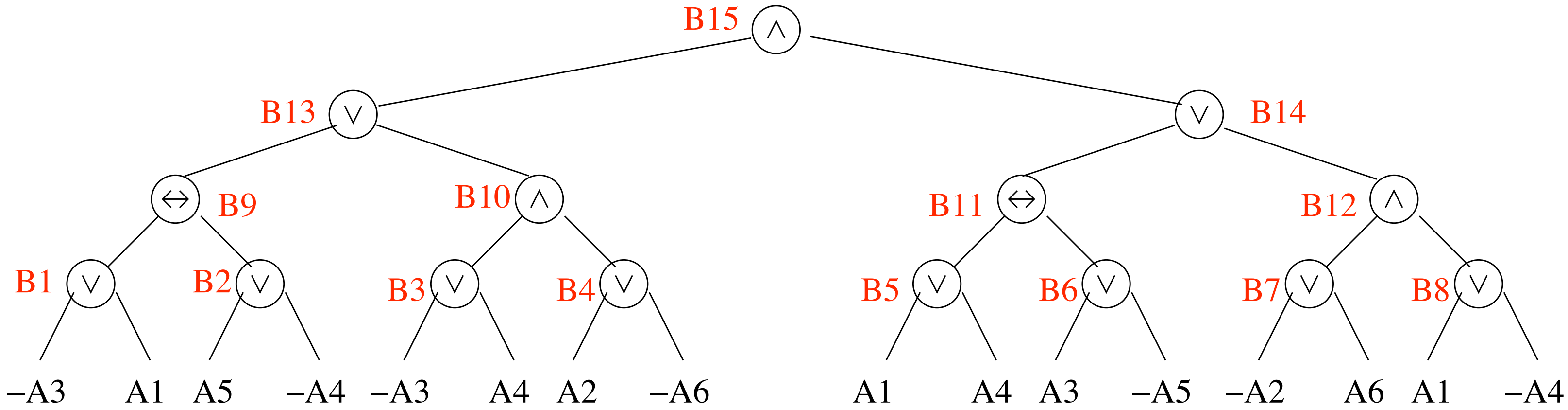
- Smaller in size:

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

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

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

Labeling CNF conversion CNF_{label} – example



Basic		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}	

Labeling CNF conversion CNF_{label} – further optimizations

- ▷ Do not apply CNF_{label} when not necessary: (e.g., $CNF_{label}(\varphi_1 \wedge \varphi_2) \Longrightarrow CNF_{label}(\varphi_1) \wedge \varphi_2$, if φ_2 already in CNF)
- ▷ Apply Demorgan's rules where it is more effective: [13] (e.g., $CNF_{label}(\varphi_1 \wedge (A \rightarrow (B \wedge C))) \Longrightarrow CNF_{label}(\varphi_1) \wedge (\neg A \vee B) \wedge (\neg A \vee C)$
- ▷ exploit the associativity of $\wedge$'s and $\vee$'s: $\ldots \underbrace{(A_1 \vee (A_2 \vee A_3))}_{B} \ldots \Longrightarrow \ldots CNF(B \leftrightarrow (A_1 \vee A_2 \vee A_3))\ldots$
- ▷ before applying CNF_{label}, rewrite the initial formula so that to maximize the sharing of subformulas (RBC, BED)
- ▷ ...

Content

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
(100 variables $\Rightarrow > 10^{30}$ assignment $\Rightarrow > 10^{12}$ years assuming the evaluation of one assignment takes 1ns.)

Resolution [41, 12]

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

Resolution Rule

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

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

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

NOTE: many standard inference rules subcases of resolution:

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

Resolution Rules [12]: unit propagation

▷ Unit resolution:

$$\frac{\Gamma' \wedge (l) \wedge (\neg l \vee \bigvee_i l_i)}{\Gamma' \wedge (l) \wedge (\bigvee_i l_i)}$$

▷ Unit subsumption:

$$\frac{\Gamma' \wedge (l) \wedge (l \vee \bigvee_i l_i)}{\Gamma' \wedge (l)}$$

Applied before general resolution rule!

Resolution: basic strategy [12]

```
function Resolution(Γ)
    if ⊥ ∈ Γ                                              /* unsat */
        then return False;
    if (Resolve() is no more applicable to Γ)             /* sat     */
        then return True;
    if {a unit clause (l) occurs in Γ}                    /* unit    */
        then  Γ := Unit_Propagate(l, Γ));
        return  Resolution(Γ)
    v := select-variable(Γ);                              /* resolve */
    Γ = Γ ∪ ⋃_{v∈C′,¬v∈C″} {Resolve(C′, C″)/{C′, C″}};
    return  Resolution(Γ)
```

Resolution: Examples

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

$$\Downarrow$$

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

$$\Downarrow$$

$$\bot$$

$\Longrightarrow$ UNSAT

Resolution: Examples (cont.)

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

$$\Downarrow$$

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

$$\Downarrow$$

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

$\Longrightarrow$SAT

Resolution: Examples

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

$$\Downarrow$$

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

$$\Downarrow$$

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

$$\Downarrow$$

$$\bot$$

$\Longrightarrow$UNSAT

Resolution – summary

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

Semantic tableaux [47]

- ▷ 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

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

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

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

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

Semantic Tableaux – example

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

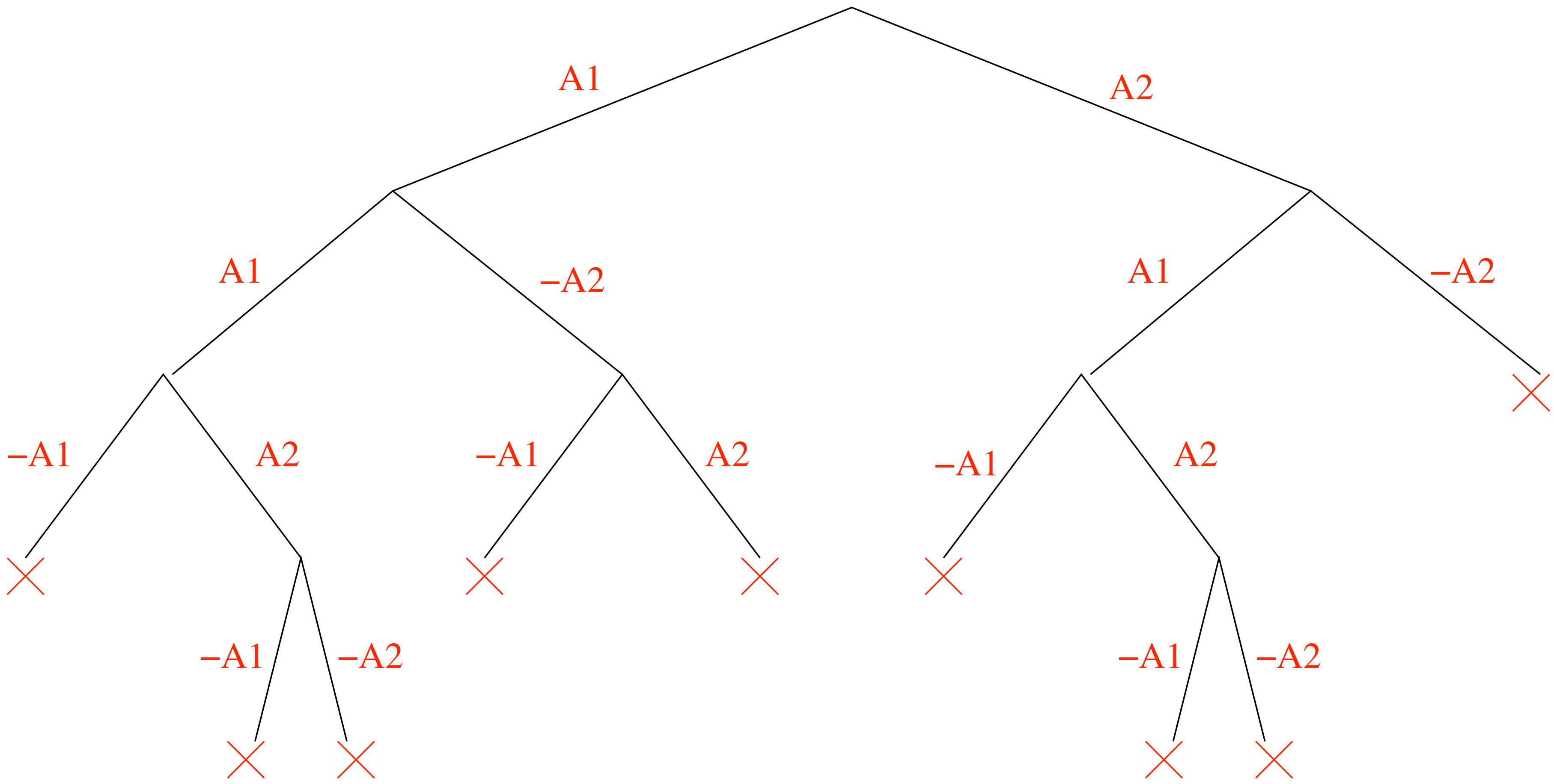


Tableau algorithm

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 [12, 11]

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

DPLL rules

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

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

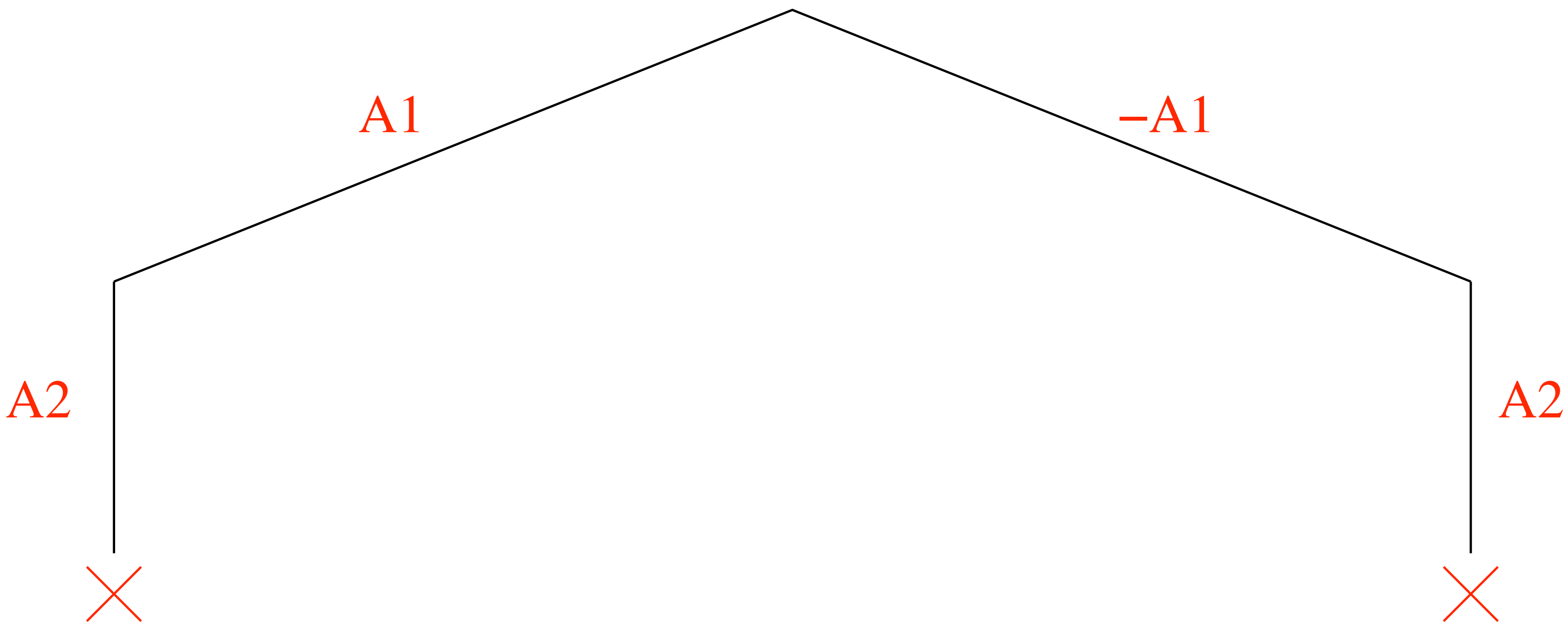
$$\frac{\varphi}{\varphi[l|\top] \quad \varphi[l|\bot]} \ (split)$$

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

- Split applied if and only if the others cannot be applied.
- Richer formalisms described in [49, 37]

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

DPLL – summary

- ▷ Handles CNF formulas (non-CNF variant known [2, 22]).
- ▷ 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 [52, 46, 4, 36].

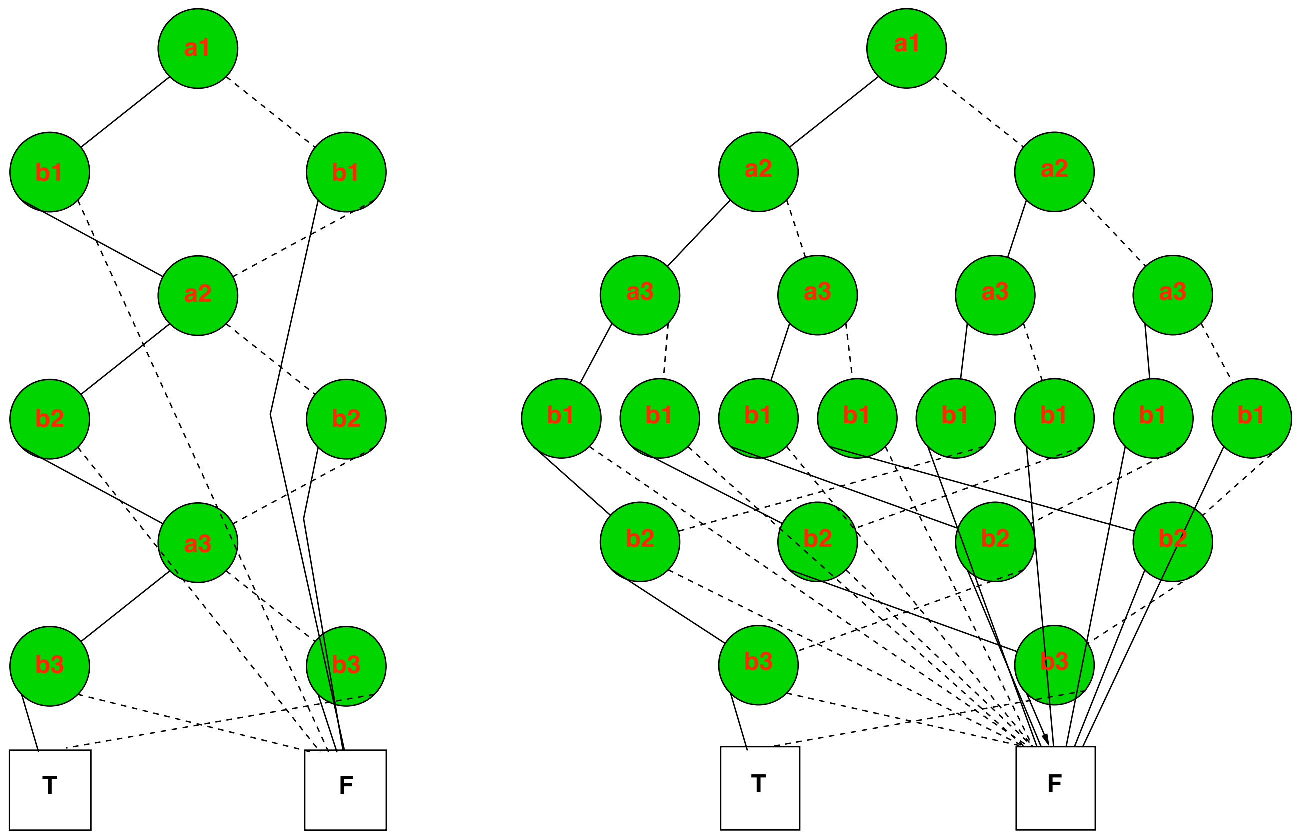
Ordered Binary Decision Diagrams (OBDDs) [8]

▷ "If-then-else" binary DAGs with two leaves: 1 and 0

▷ Paths leading to 1 represent models
  Paths leading to 0 represent counter-models

▷ Variable ordering $A_1, A_2, ..., A_n$ imposed a priori.

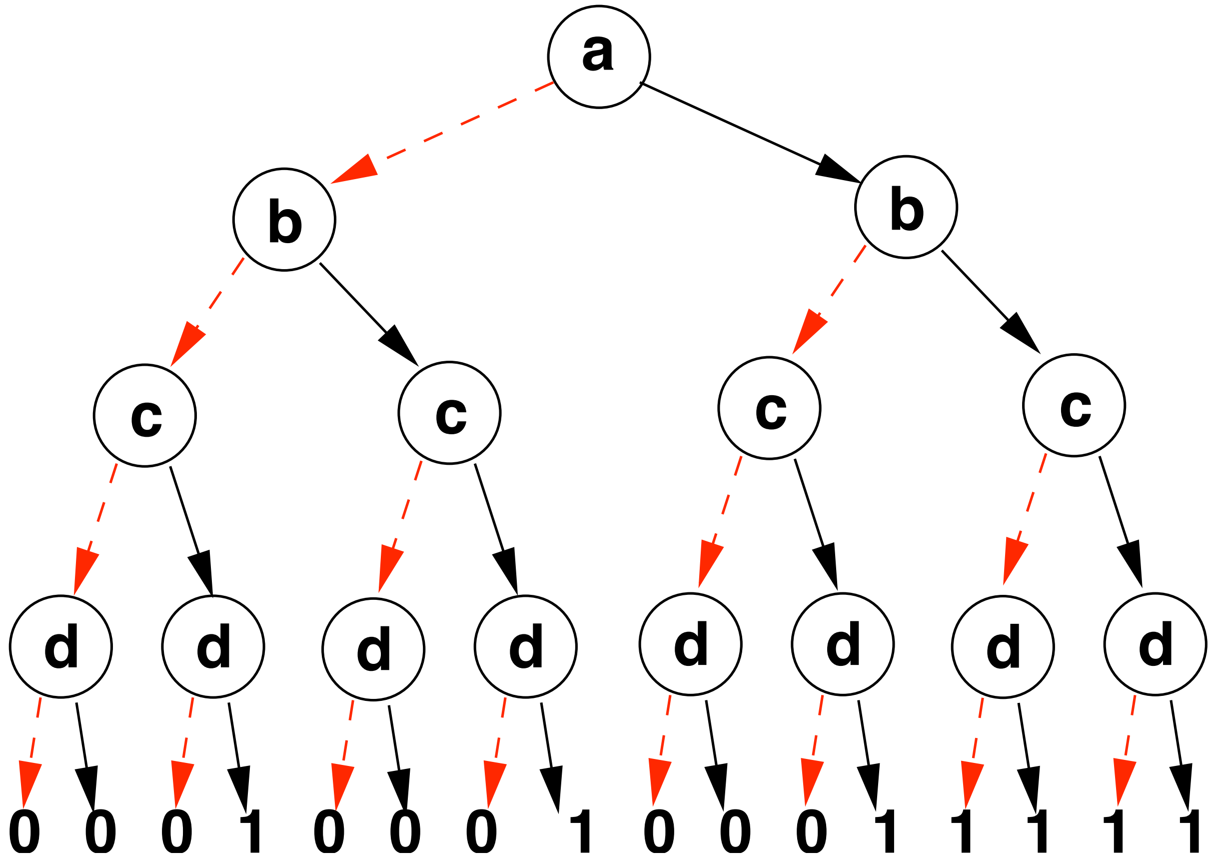
OBDD - Examples



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

Ordered Decision Trees

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

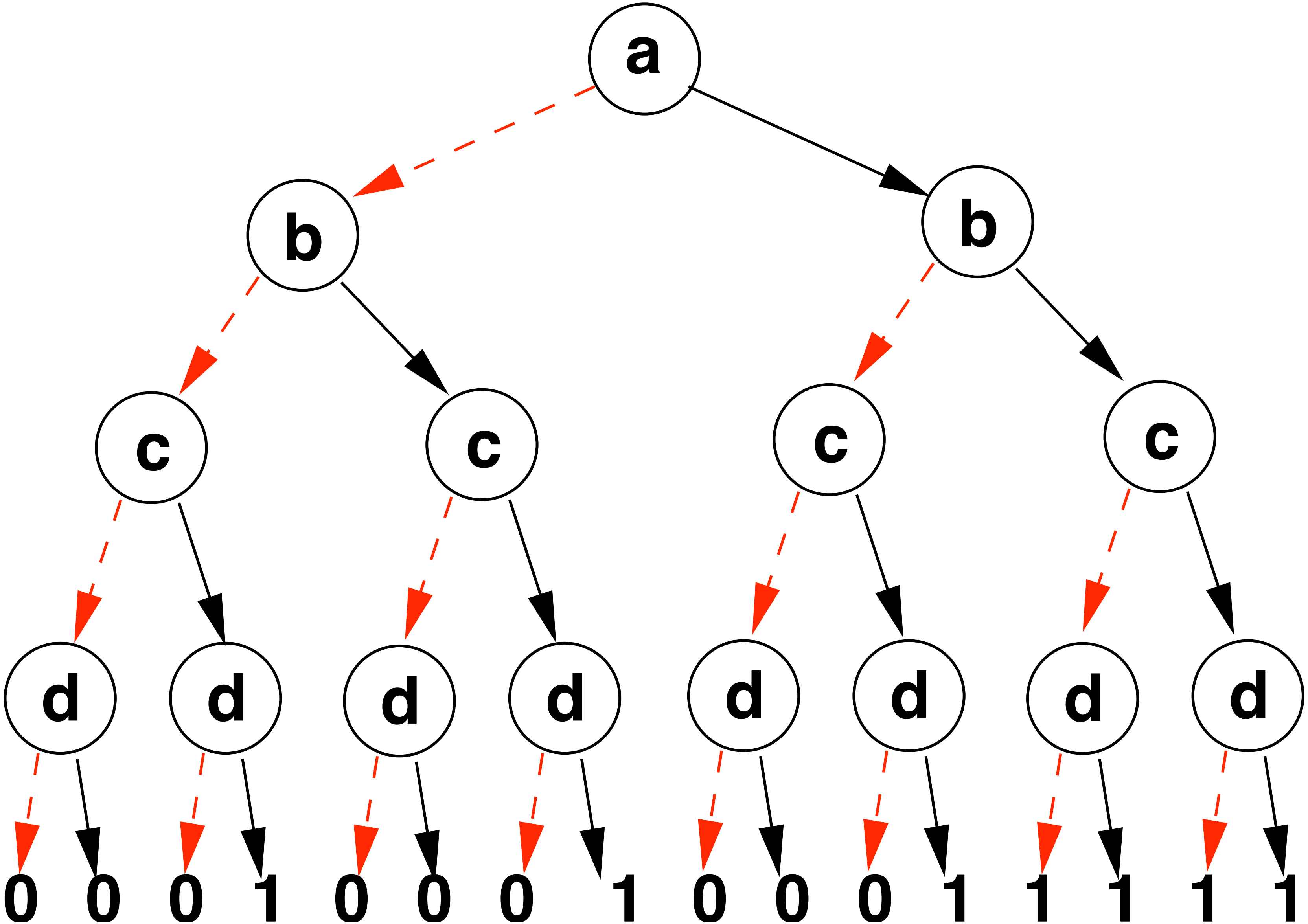


From Ordered Decision Trees to OBDD's: reductions

▷ Recursive applications of the following reductions:

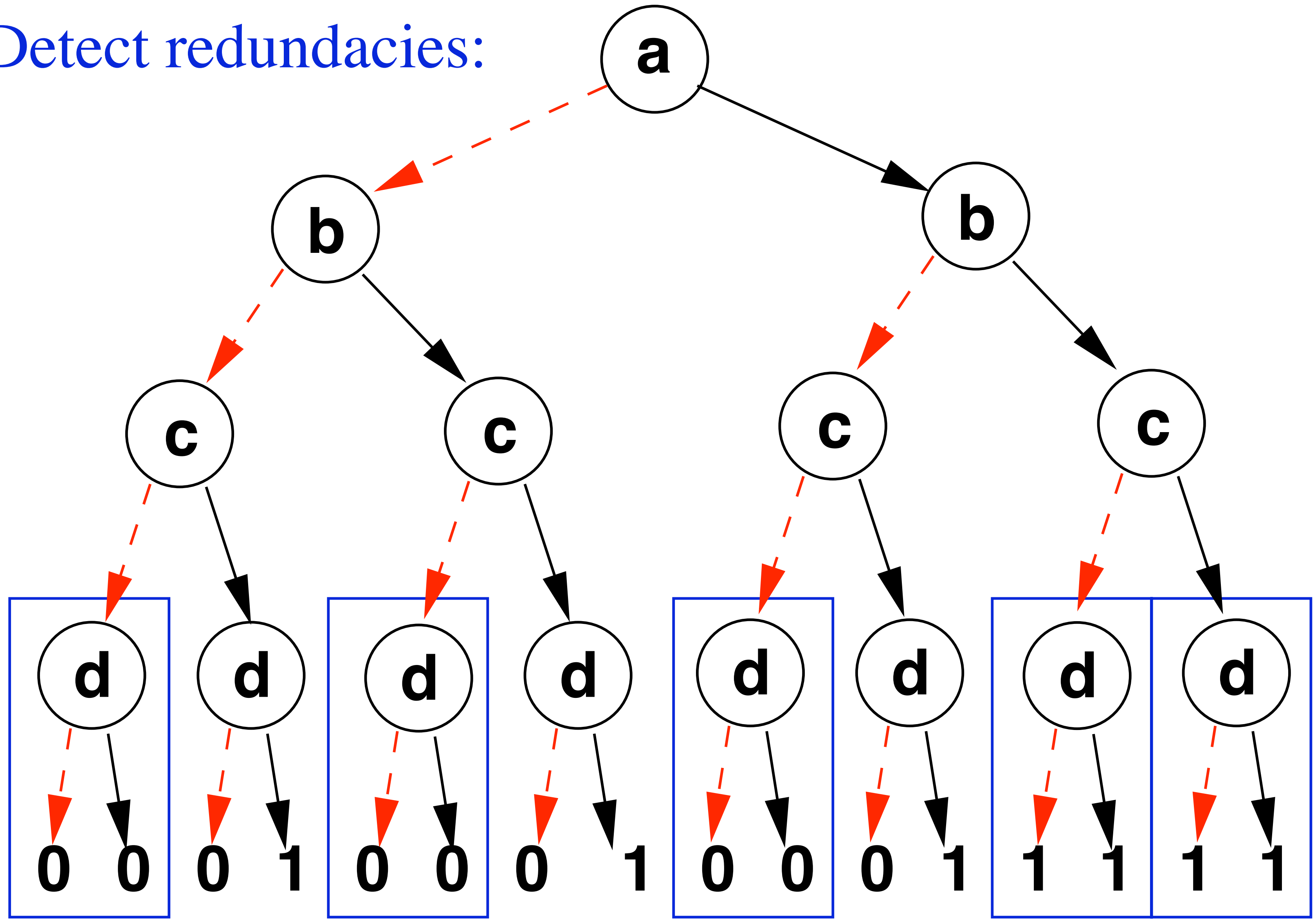
- share subnodes: point to the same occurrence of a subtree
- remove redundancies: nodes with same left and right children can be eliminated

Reduction: example



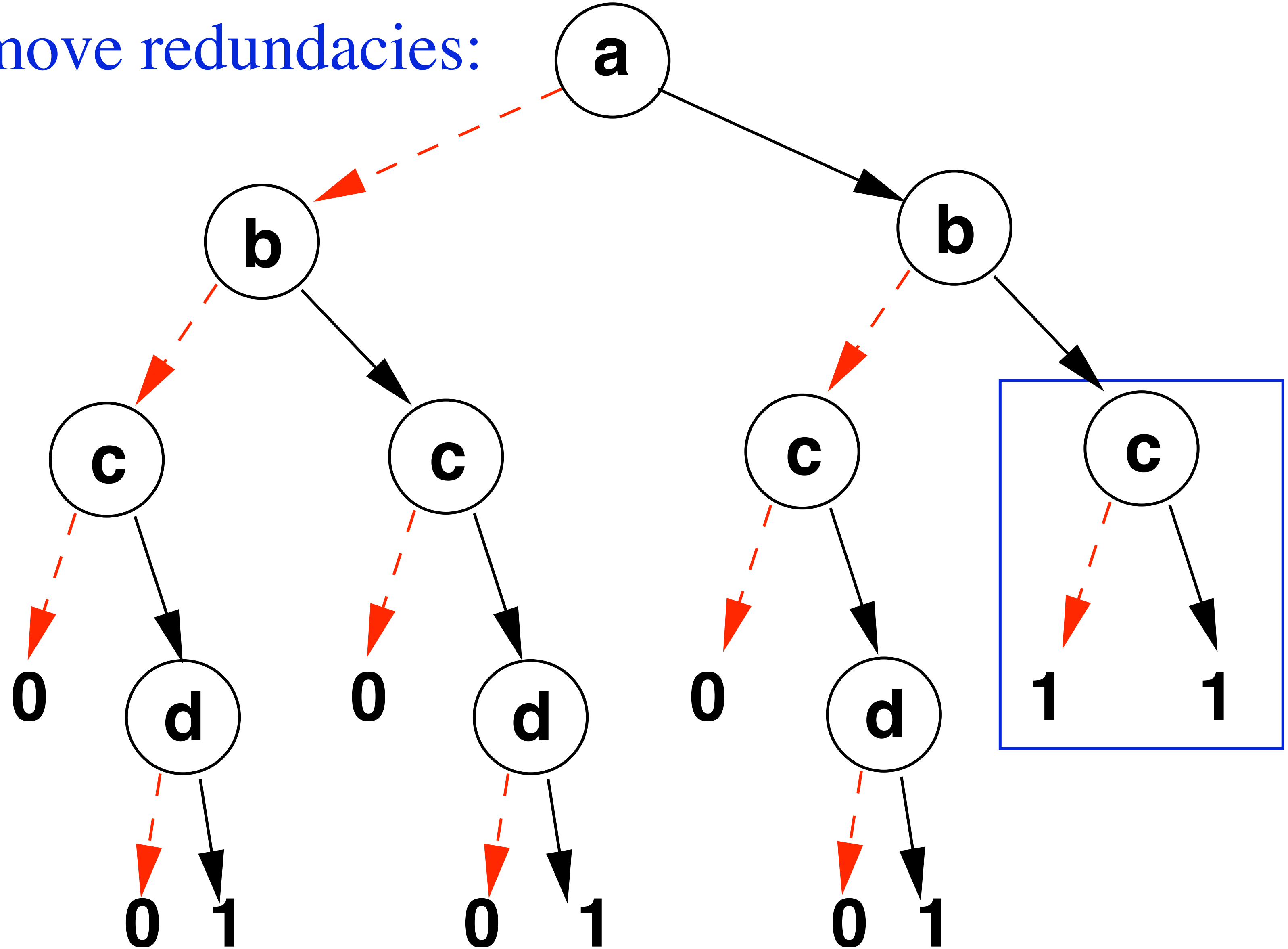
Reduction: example [cont.]

Detect redundacies:



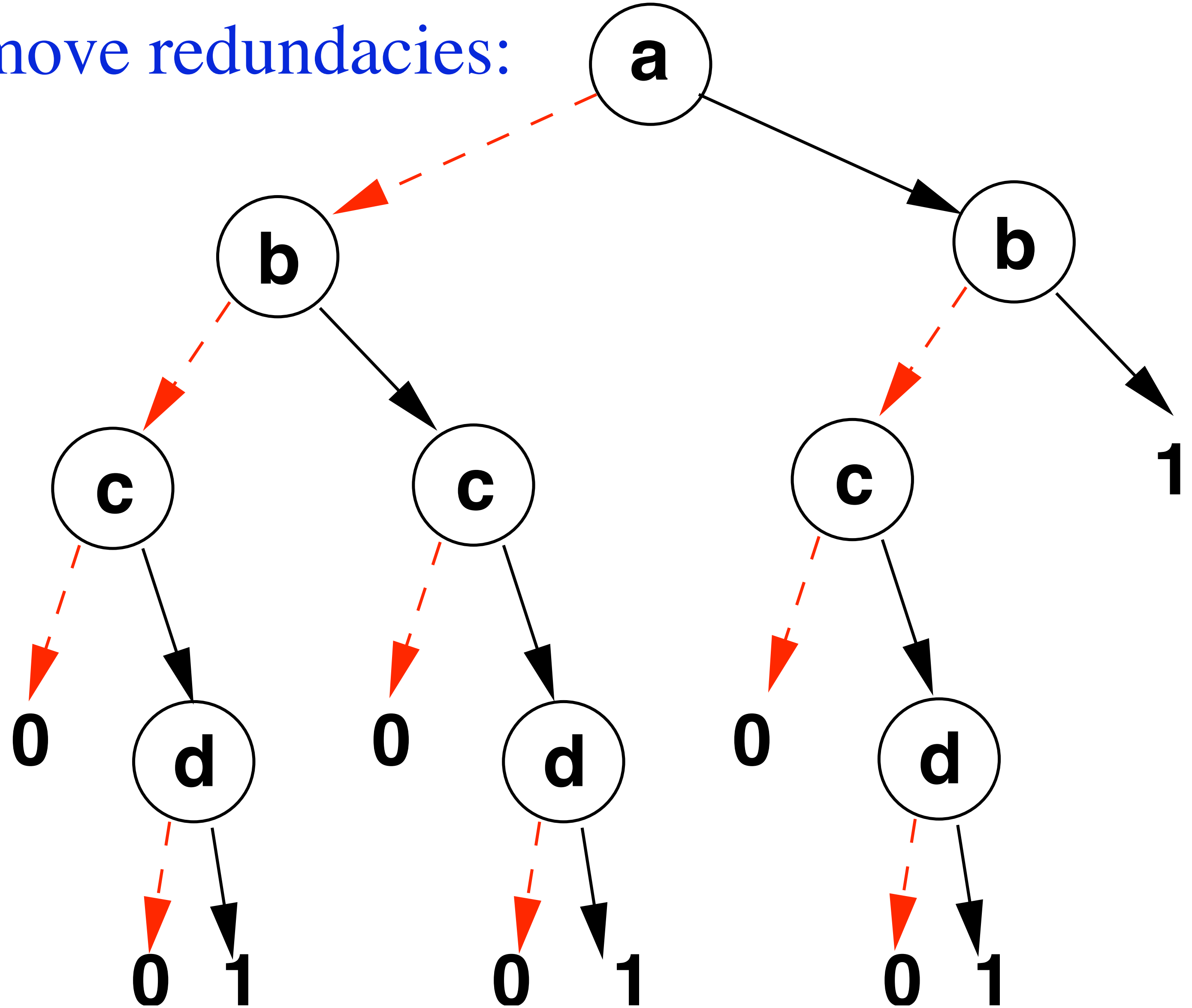
Reduction: example [cont.]

Remove redundacies:



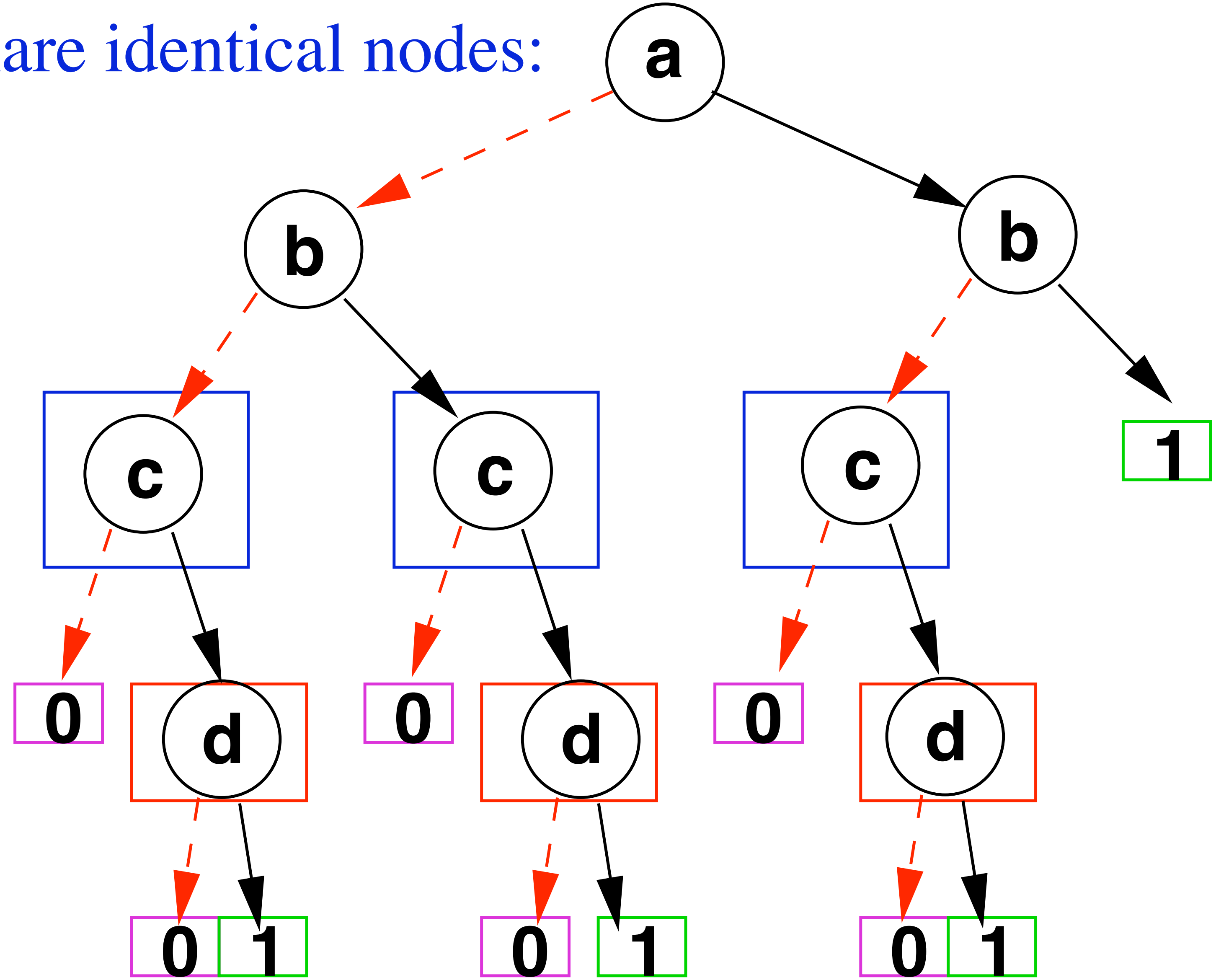
Reduction: example [cont.]

Remove redundacies:



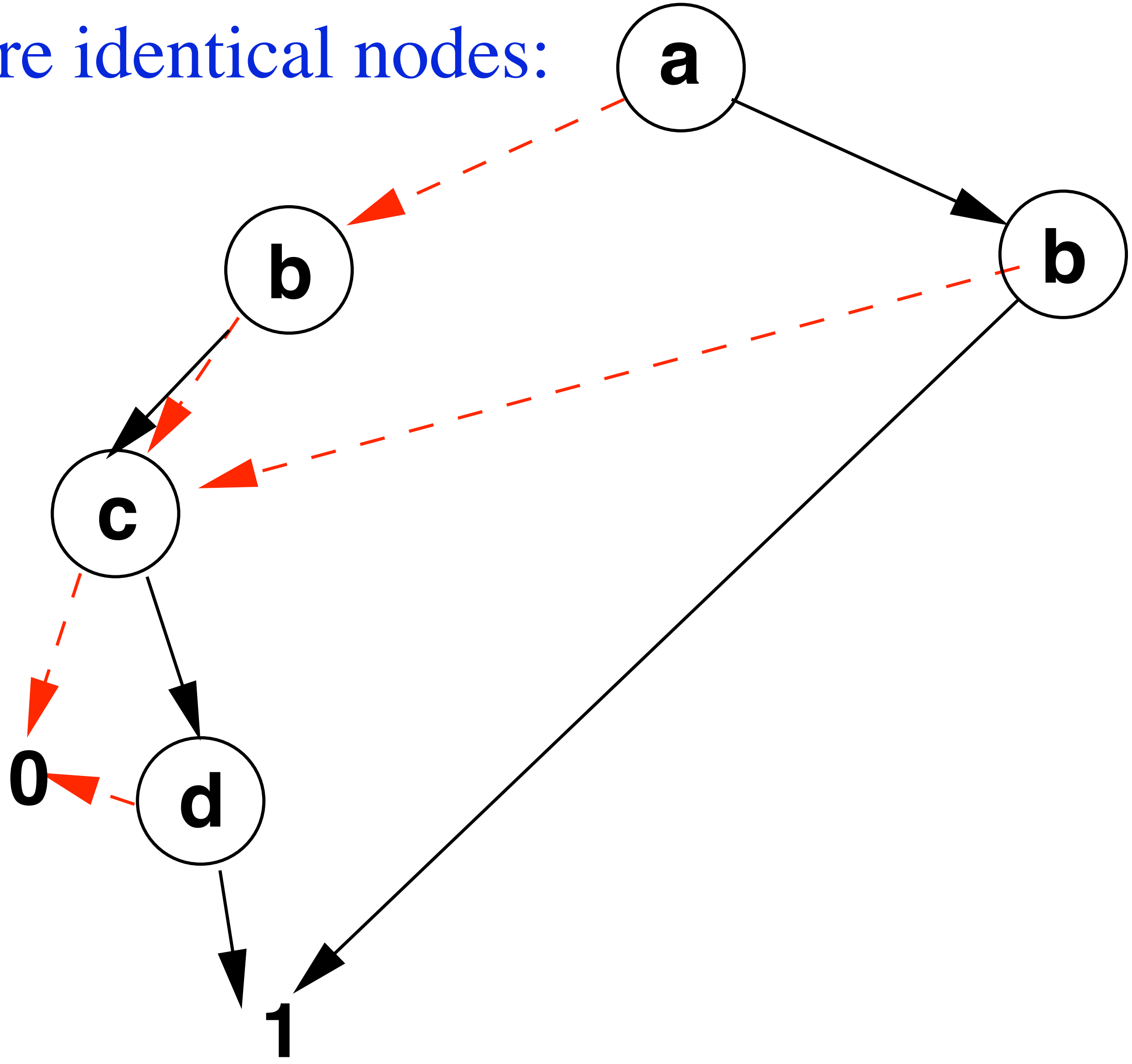
Reduction: example [cont.]

Share identical nodes:



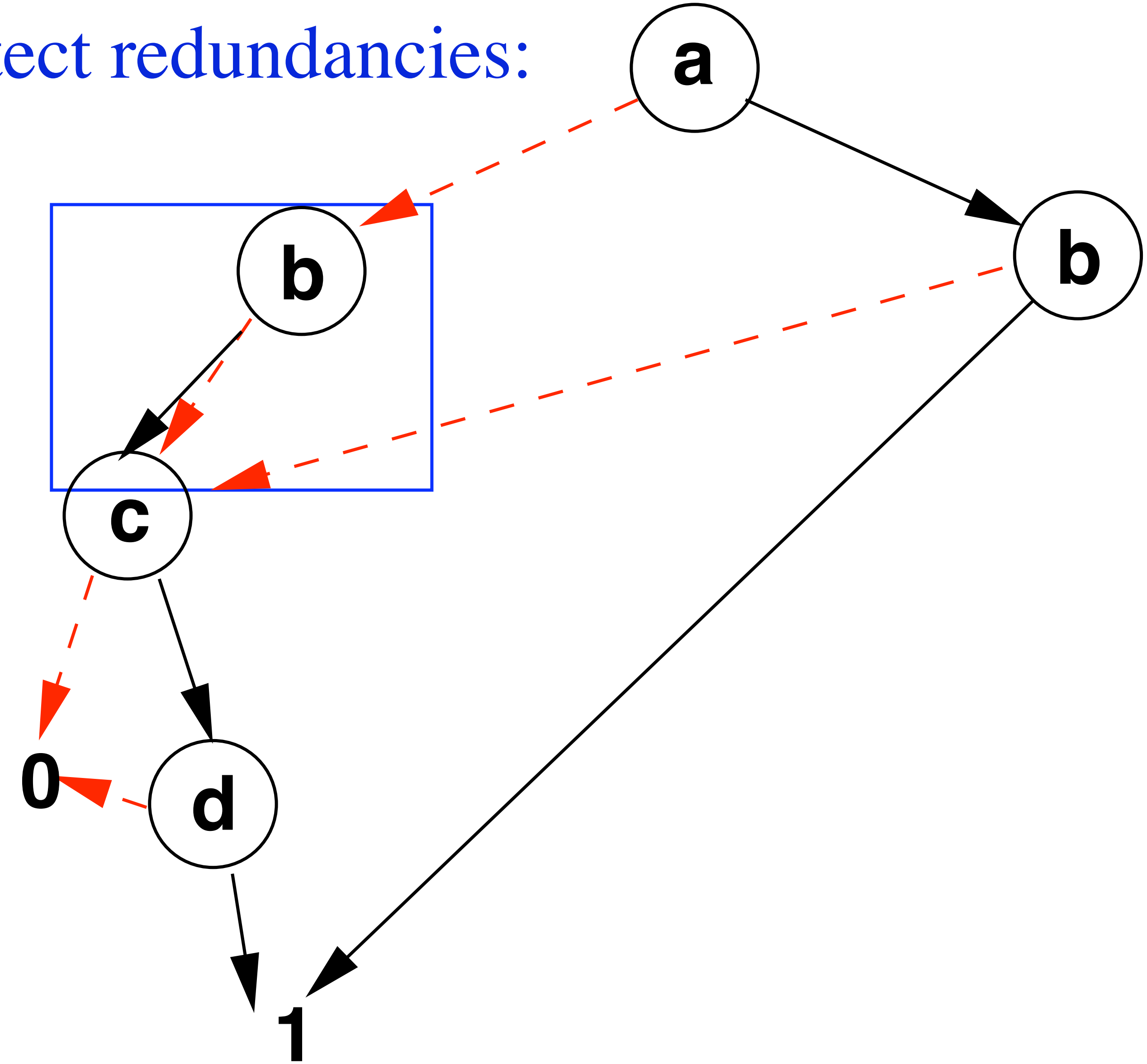
Reduction: example [cont.]

Share identical nodes:



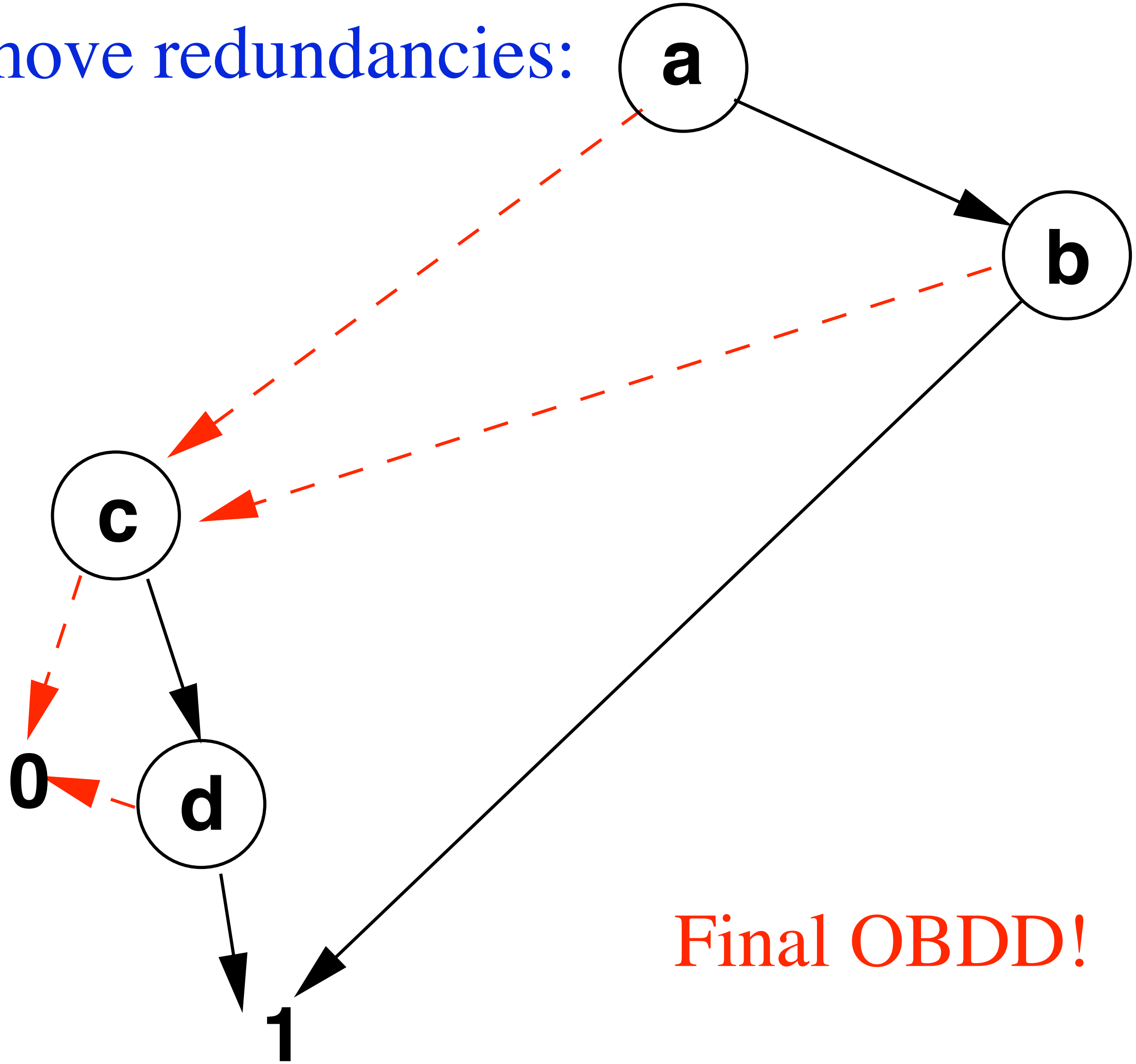
Reduction: example [cont.]

Detect redundancies:



Reduction: example [cont.]

Remove redundancies:



Final OBDD!

Recursive structure of an OBDD

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

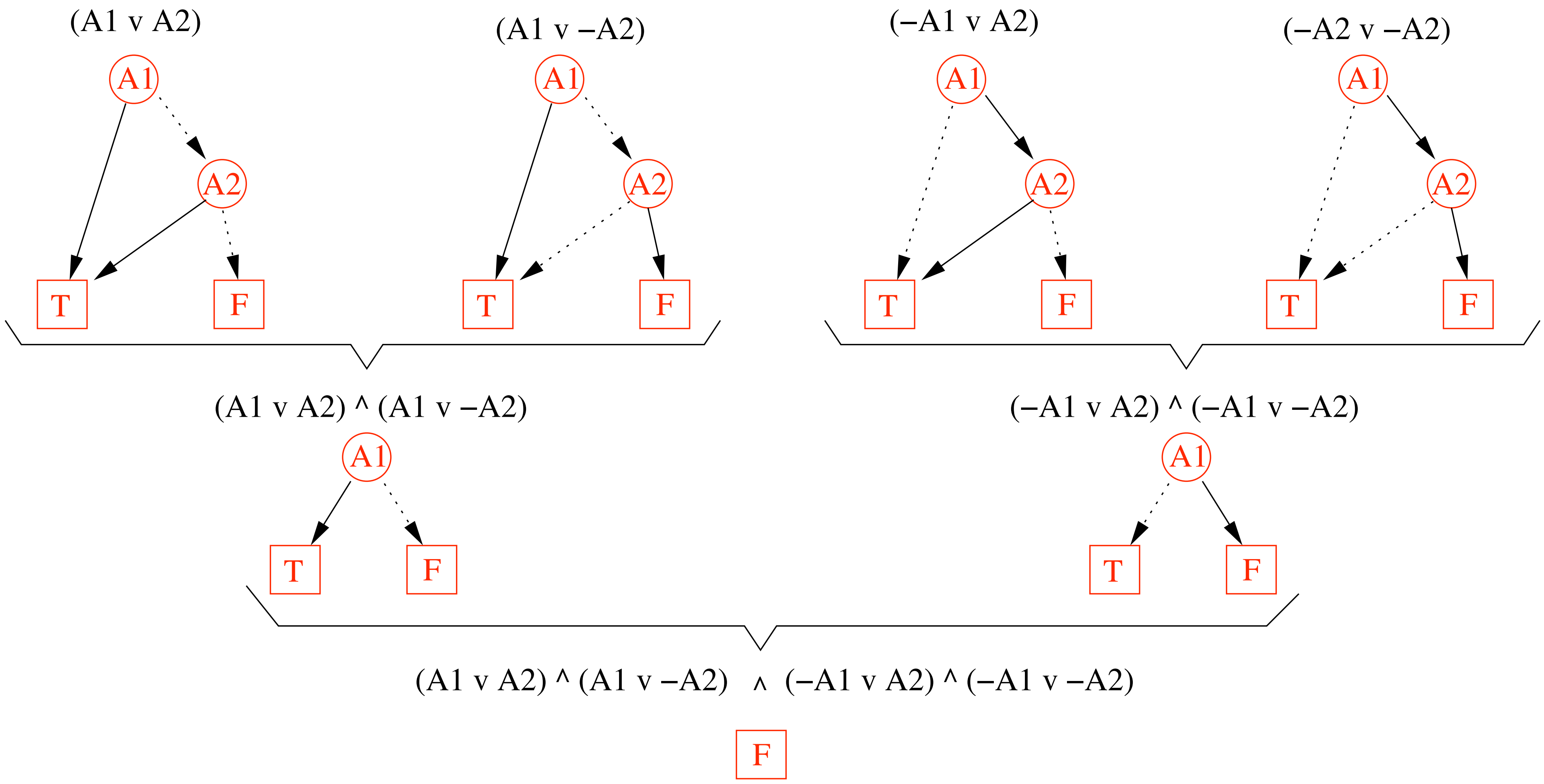
Incrementally building an OBDD

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

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

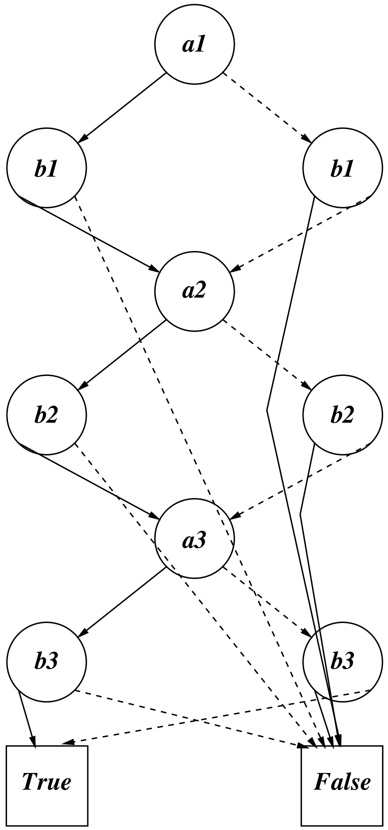
OBBD incremental building – example

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

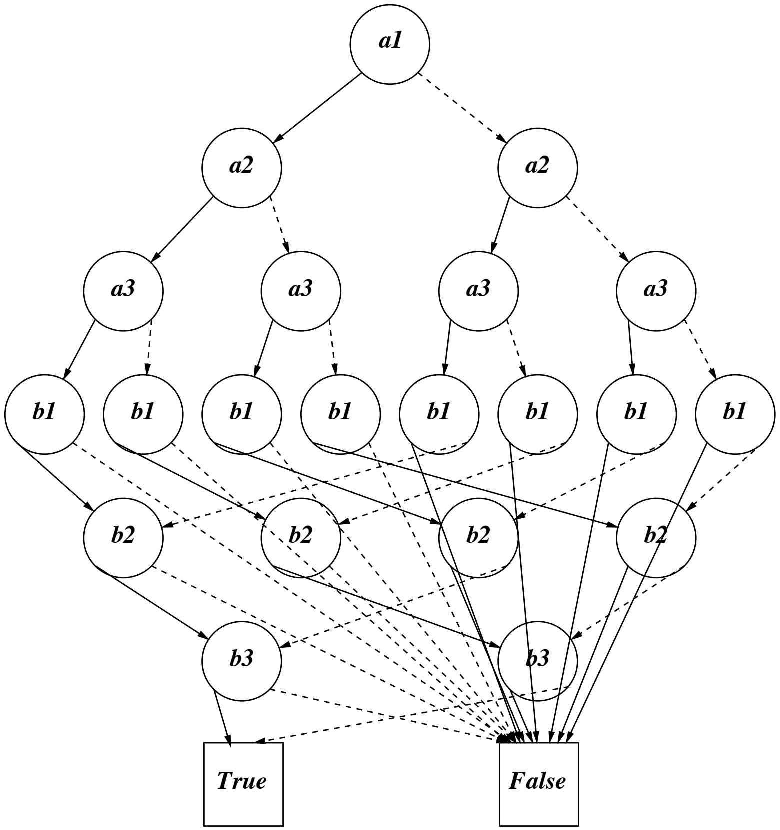


Critical choice of variable Orderings in OBDD's

$$\varphi = (a1 \leftarrow b1) \wedge (a2 \leftarrow b2) \wedge (a3 \leftarrow b3)$$



Linear size



Exponential size

OBDD's as canonical representation of Boolean formulas

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

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

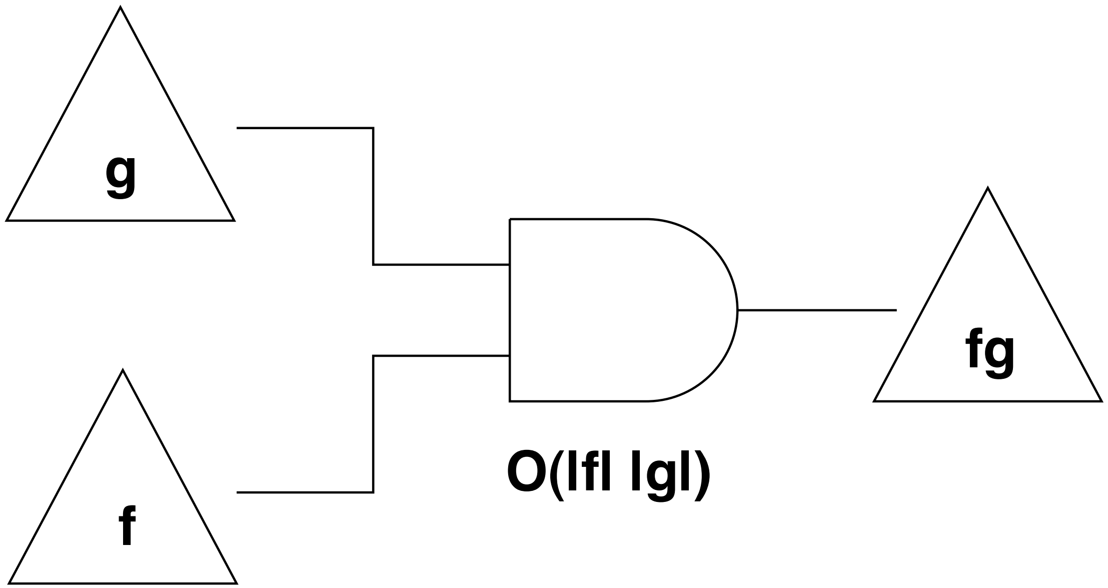
- equivalence check requires constant time!
  $\Longrightarrow$validity check requires constant time! $(\varphi \leftrightarrow \top)$
  $\Longrightarrow$(un)satisfiability check requires constant time! $(\varphi \leftrightarrow \bot)$
- the set of the paths from the root to 1 represent all the models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

Exponentiality of OBDD's

- The size of OBDD's may grow exponentially wrt. the number of variables in worst-case
- Consequence of the canonicity of OBDD's (unless P = co-NP)
- Example: there exist no polynomial-size OBDD representing the electronic circuit of a bitwise multiplier
- N.B.: the size of intermediate OBDD's may be bigger than that of the final one (e.g., inconsistent formula)

Useful Operations over OBDDs

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



(but typically much smaller on average).

Boolean quantification

- If v is a Boolean variable, then

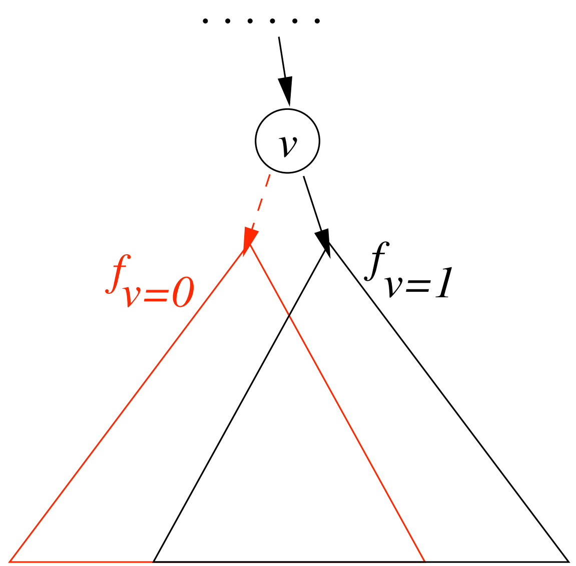
$$\exists v.f \quad := \quad f|_{v=0} \vee f|_{v=1}$$
$$\forall v.f \quad := \quad f|_{v=0} \wedge f|_{v=1}$$

- Multi-variable quantification: $\exists(w_1, \ldots, w_n).f \; := \; \exists w_1 \ldots \exists w_n.f$
- Example: $\exists(b,c).((a \wedge b) \vee (c \wedge d)) \; = \; a \vee d$
- naive expansion of quantifiers to propositional logic may cause a blow-up in size of the formulae
- OBDD's handle very efficiently quantification operations

OBDD's and Boolean quantification

▷ OBDD's handle quantification operations rather efficiently

- if f is a sub-OBDD labeled by variable v, then $f|_{v=1}$ and $f|_{v=0}$ are the "then" and "else" branches of f



$\Longrightarrow$ lots of sharing of subformulae!

OBDD – summary

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

Incomplete SAT techniques: GSAT, WSAT [45, 44]

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

The GSAT algorithm

```
function GSAT(φ)
    for i := 1 to Max-tries do
        μ := rand-assign(φ);
        for j := 1 to Max-flips do
            if (score(φ, μ) = 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".
```

The WalkSAT algorithm

Slide contributed by the student Silvia Tomasi

```
WALKSAT(φ,MAX-STEPS,MAX-TRIES, select())
for i ← 1 to MAX-TRIES
do μ ← a randomly generated truth assignment;
    for j ← 1 to MAX-STEPS
    do if μ satisfies φ
           then return μ;
           else C ← randomly selected clause unsatisfied under μ;
                x ← variable selected from C according to heuristic select();
                μ ← μ with x flipped;
return error "no solution found"
```

GSAT & WSAT– summary

- Handle only CNF formulas.
- Incomplete
- Extremely efficient for some (satisfiable) problems.
- Require polynomial space
- Non-CNF Variants: NC-GSAT [42], DAG-SAT [43]

Content

Variants of DPLL

DPLL is a family of algorithms.

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

Modern DPLL implementations [46, 4, 55, 23]

- ▷ Non-recursive: stack-based representation of data structures
- ▷ Efficient data structures for doing and undoing assignments
- ▷ Perform non-chronological backtracking and learning
- ▷ May perform search restarts
- ▷ Reason on total assignments

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

Iterative description of DPLL [46, 55]

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

Iterative description of DPLL [46, 55]

- ▷ `preprocess`(φ, μ) simplifies φ into an easier equisatisfiable formula (and updates μ if it is the case)
- ▷ `decide_next_branch`(φ, μ) chooses a new decision literal from φ according to some heuristic, and adds it to μ
- ▷ `deduce`(φ, μ, η) performs all deterministic assignments (unit), and updates φ, μ accordingly. If this causes a conflict, η is the subset of μ causing the conflict (conflict set).
- ▷ `analyze_conflict`(φ, μ, η) returns the "wrong-decision" level suggested by η ("0" means that a conflict exists even without branching)
- ▷ `backtrack(blevel,`φ, μ`)` undoes the branches up to blevel, and updates φ, μ accordingly

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
  - forward checking (branching heuristics)
- Look-back: exploit information about search which has already taken place
  - Backjumping & learning
- Others
  - restarts
  - ...

Preprocessing: (sorting plus) subsumption

▷ Detect and remove subsumed clauses:

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

$$\Downarrow$$

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

Preprocessing: detect & collapse equivalent literals [7]

Repeat:

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

Until (no more simplification is possible).

▷ Ex:

$$\varphi_1 \wedge (\neg l_2 \vee l_1) \wedge \varphi_2 \wedge (\neg l_3 \vee l_2) \wedge \varphi_3 \wedge (\neg l_1 \vee l_3) \wedge \varphi_4$$

$$\Downarrow_{l_1 \leftrightarrow l_2 \leftrightarrow l_3}$$

$$(\varphi_1 \wedge \varphi_2 \wedge \varphi_3 \wedge \varphi_4)[l_2 \leftarrow l_1; l_3 \leftarrow l_1;]$$

▷ Very effective in many application domains.

Preprocessing: resolution (and subsumption) [3]

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

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

$$\Downarrow_{resolve}$$

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

$$\Downarrow_{unit-propagate}$$

$$(\varphi_1 \wedge \varphi_2 \wedge \varphi_3)[l_2 \leftarrow \top]$$

Branching heuristics

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

Some example heuristics

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

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

  $\Longrightarrow$estimates l's contribution to the satisfiability of φ
- Satz [29]: selects a candidate set of literals, perform unit propagation, chooses the one leading to smaller clause set
  $\Longrightarrow$maximizes the effects of unit propagation
- VSIDS [36]: variable state independent decaying sum
  - "static": scores updated only at the end of a branch
  - "local": privileges variable in recently learned clauses

"Classic" chronological backtracking

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

Classic chronological backtracking – example (1)

$c_1 : \neg A_1 \vee A_2$

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

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

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

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

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

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

$c_8 : A_1 \vee A_8$

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

...

Classic chronological backtracking – example (2)

$c_1 : \neg A_1 \vee A_2$

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

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

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

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

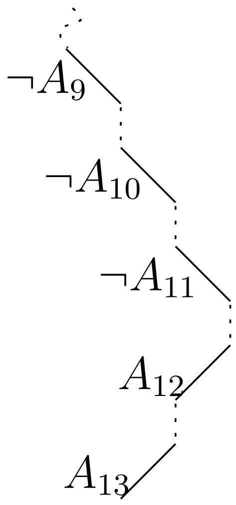
$c_6 : \neg A_5 \vee \neg A_6$

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

$c_8 : A_1 \vee A_8$

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

...



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

Classic chronological backtracking – example (3)

$c_1 : \neg A_1 \vee A_2$

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

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

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

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

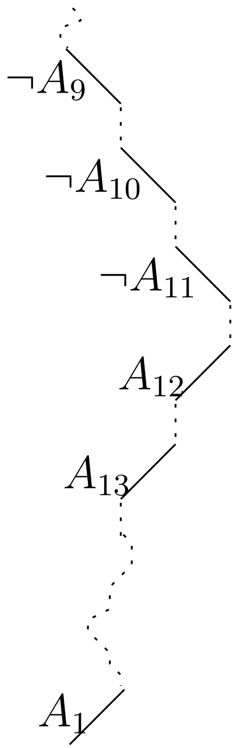
$c_6 : \neg A_5 \vee \neg A_6$

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

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

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

...



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

... (branch on A_1)

Classic chronological backtracking – example (4)

$c_1 : \neg A_1 \vee A_2$ ✓

$c_2 : \neg A_1 \vee A_3 \vee A_9$ ✓

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

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

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

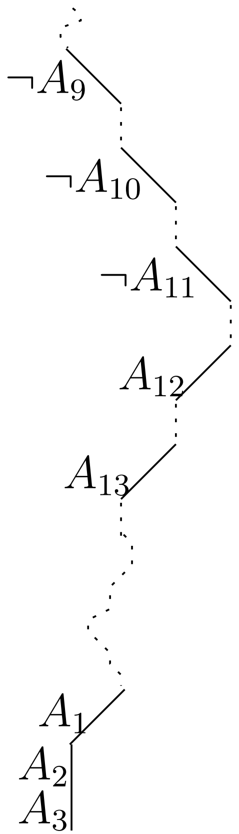
$c_6 : \neg A_5 \vee \neg A_6$

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

$c_8 : A_1 \vee A_8$ ✓

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

...



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

(unit A_2, A_3)

Classic chronological backtracking – example (5)

$c_1 : \neg A_1 \vee A_2$ √

$c_2 : \neg A_1 \vee A_3 \vee A_9$ √

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

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

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

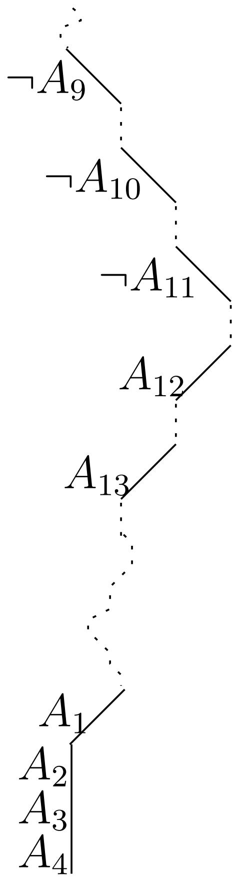
$c_6 : \neg A_5 \vee \neg A_6$

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

$c_8 : A_1 \vee A_8$ √

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

...



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

(unit A_4)

Classic chronological backtracking – example (6)

$c_1 : \neg A_1 \vee A_2$ ✓

$c_2 : \neg A_1 \vee A_3 \vee A_9$ ✓

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

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

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

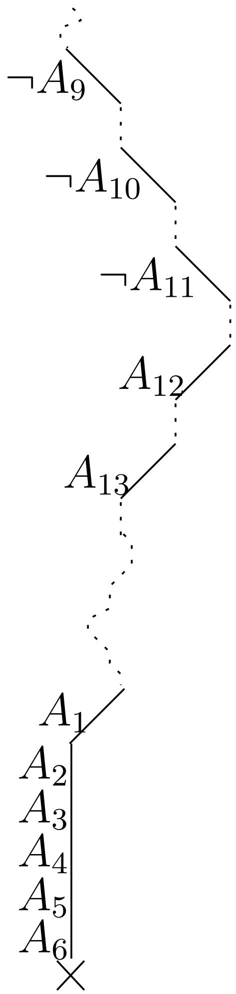
$c_6 : \neg A_5 \vee \neg A_6$ ×

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

$c_8 : A_1 \vee A_8$ ✓

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

...



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

(unit A_5, A_6) $\Longrightarrow$ conflict

Classic chronological backtracking – example (7)

$c_1 : \neg A_1 \vee A_2$

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

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

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

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

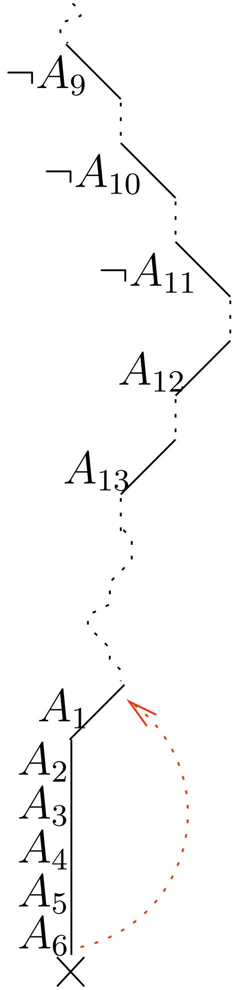
$c_6 : \neg A_5 \vee \neg A_6$

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

$c_8 : A_1 \vee A_8$

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

...



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

$\Longrightarrow$ backtrack up to A_1

Classic chronological backtracking – example (8)

$c_1 : \neg A_1 \vee A_2$ √

$c_2 : \neg A_1 \vee A_3 \vee A_9$ √

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

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

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

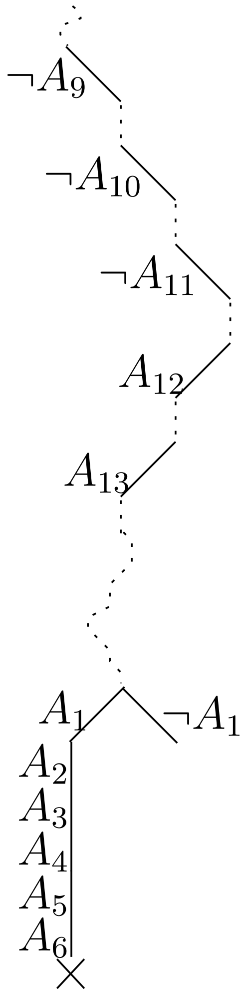
$c_6 : \neg A_5 \vee \neg A_6$

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

$c_8 : A_1 \vee A_8$

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

...



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

(unit $\neg A_1$)

Classic chronological backtracking – example (9)

$c_1 : \neg A_1 \vee A_2$ √

$c_2 : \neg A_1 \vee A_3 \vee A_9$ √

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

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

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

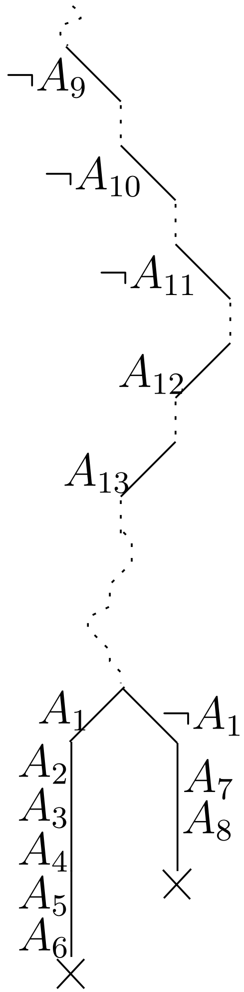
$c_6 : \neg A_5 \vee \neg A_6$

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

$c_8 : A_1 \vee A_8$ √

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

...



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

(unit A_7, A_8) $\Longrightarrow$ conflict

Classic chronological backtracking – example (10)

$c_1 : \neg A_1 \vee A_2$

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

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

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

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

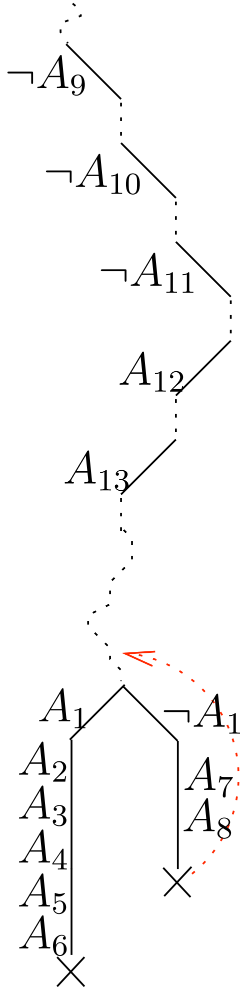
$c_6 : \neg A_5 \vee \neg A_6$

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

$c_8 : A_1 \vee A_8$

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

...



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

$\Longrightarrow$ backtrack to the most recent open branching point

Classic chronological backtracking – example (10)

$c_1 : \neg A_1 \vee A_2$

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

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

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

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

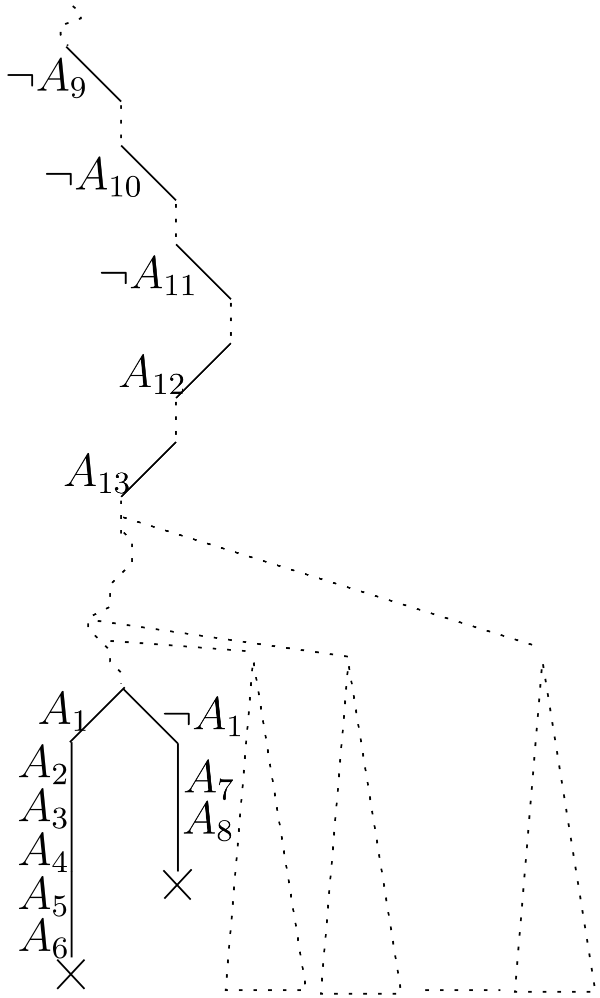
$c_6 : \neg A_5 \vee \neg A_6$

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

$c_8 : A_1 \vee A_8$

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

...



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

$\Longrightarrow$ lots of useless search before backtracking up to A_{13}!

Classic chronological backtracking: drawbacks

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

Conflict-directed backtracking (backjumping) and learning [4, 46]

- General idea: when a branch μ fails,
  1. conflict analysis: reveal the sub-assignment $\eta \subseteq \mu$ causing the failure (conflict set η)
  2. learning: add the conflict clause $C \stackrel{\text{def}}{=} \neg\eta$ to the clause set
  3. backjumping: use η to decide the point where to backtrack
- may jump back up much more than one decision level in the stack $\Longrightarrow$may avoid lots of redundant search!!.
- we illustrate two main backjumping & learning strategies:
  - the original strategy presented in [46]
  - the state-of-the-art 1^{st}UIP strategy [54]

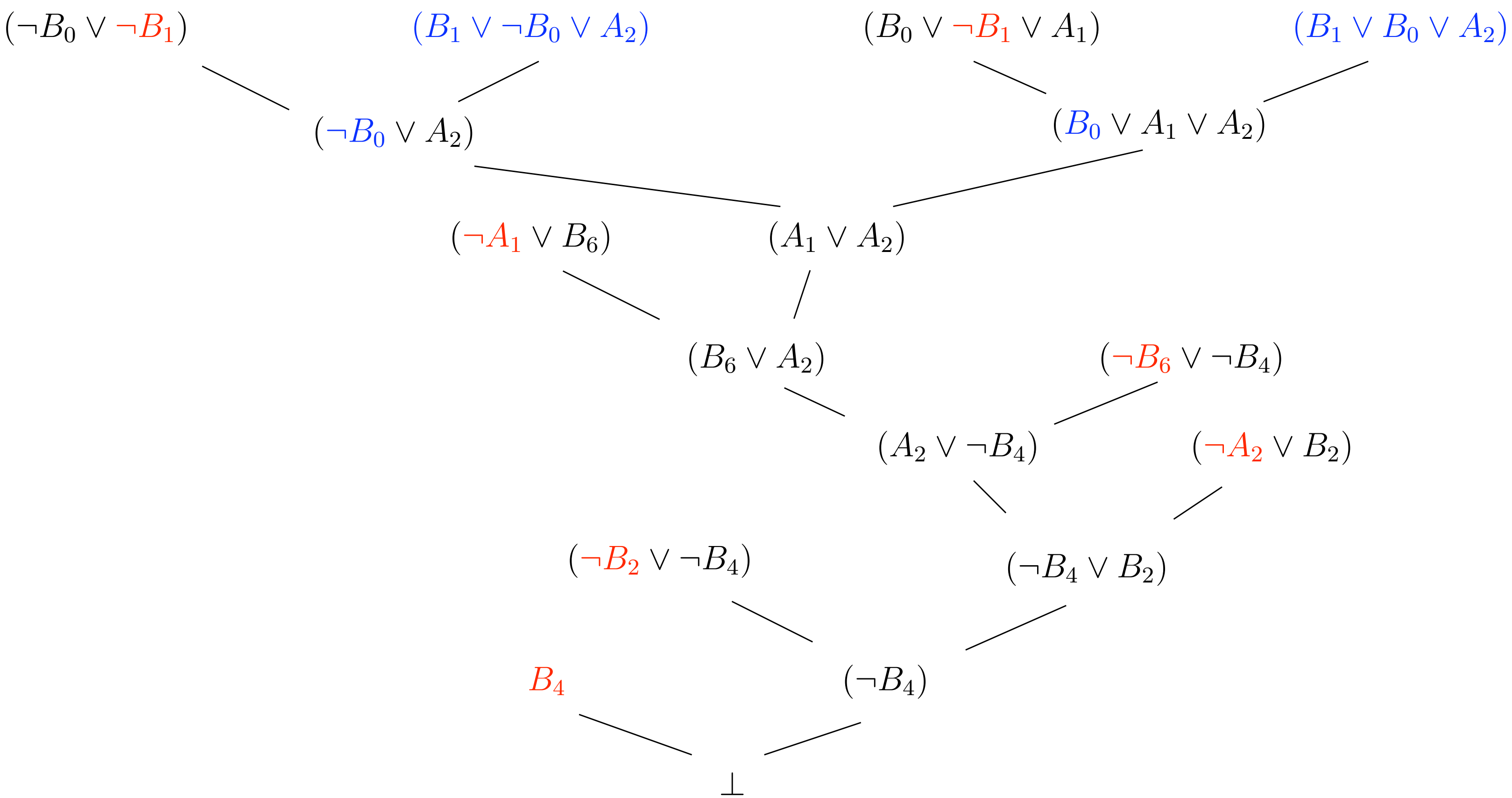
Preliminary: Correspondence between Search trees and Resolution Proofs

In the case of an unsatisfiable formula, the search tree explored by DPLL corresponds to a (tree) resolution proof of its unsatisfiability.

Given the above, “learning” corresponds to storing intermediate resolution steps computed during the search.

Example

$$(B_0 \lor \neg B_1 \lor A_1) \land (B_0 \lor B_1 \lor A_2) \land (\neg B_0 \lor B_1 \lor A_2) \land (\neg B_0 \lor \neg B_1) \land (\neg B_2 \lor \neg B_4) \land$$
$$(\neg A_2 \lor B_2) \land (\neg A_1 \lor B_3) \land B_4 \land (A_2 \lor B_5) \land (\neg B_6 \lor \neg B_4) \land (B_6 \lor \neg A_1) \land B_7$$



The original backjumping and learning strategy of [46]

▷ Idea: when a branch μ fails,

1. conflict analysis: find the conflict set $\eta \subseteq \mu$ by generating the conflict clause $C \stackrel{\text{def}}{=} \neg\eta$ via resolution from the falsified clause (conflicting clause)
2. learning: add the conflict clause C to the clause set
3. backjumping: backtrack to the most recent branching point s.t. the stack does not fully contain η, and then unit-propagate the unassigned literal on C

Construction of a conflict set: implication graph

▷ An implication graph is a DAG s.t.:

- each node represents a variable assignment (literal)
- each edge $l_i \overset{c}{\longmapsto} l$ is labeled with a clause
- the node of a decision literal has no incoming edges
- all edges incoming into a node l are labeled with the same clause c, s.t. $l_1 \overset{c}{\longmapsto} l, \ldots, l_n \overset{c}{\longmapsto} l$ iff $c = \neg l_1 \vee \ldots \vee \neg l_n \vee l$
  (c is said to be the antecedent clause of l)
- when both l and $\neg l$ occur in the graph, we have a conflict.

▷ Intuition:

- the graph contains $l_1 \overset{c}{\longmapsto} l, \ldots, l_n \overset{c}{\longmapsto} l$ iff l has been obtained from $l_1, \ldots, l_n$ by unit propagation on c
- a partition of the graph with all decision literals on one side and the conflict on the other represents a conflict set

The original backjumping strategy – example [46] (1)

$c_1 : \neg A_1 \vee A_2$

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

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

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

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

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

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

$c_8 : A_1 \vee A_8$

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

...

The original backjumping strategy – example (2)

$c_1 : \neg A_1 \vee A_2$

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

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

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

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

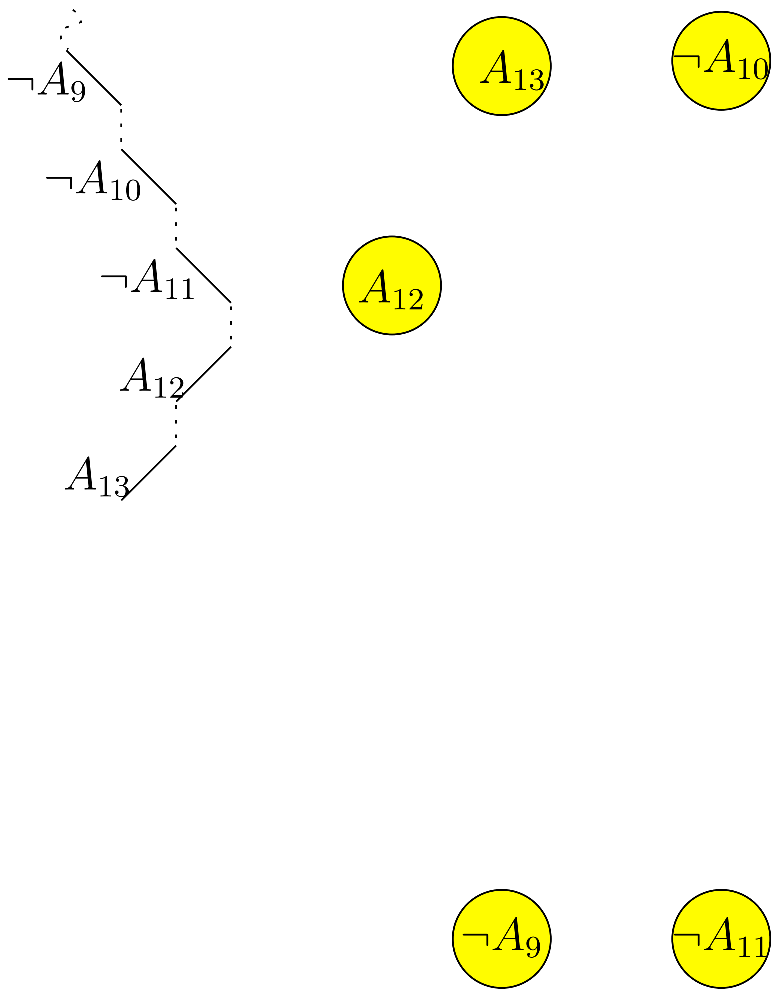
$c_6 : \neg A_5 \vee \neg A_6$

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

$c_8 : A_1 \vee A_8$

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

...



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

The original backjumping strategy – example (3)

$c_1 : \neg A_1 \vee A_2$

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

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

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

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

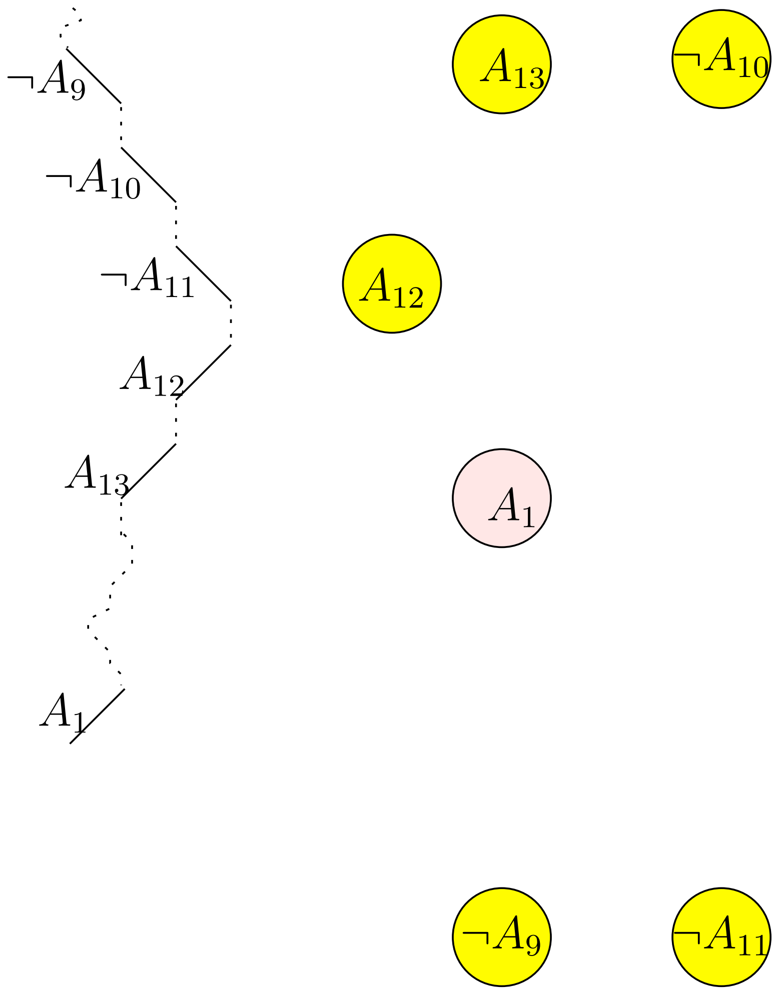
$c_6 : \neg A_5 \vee \neg A_6$

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

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

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

$\ldots$



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

... (branch on A_1)

The original backjumping strategy – example (4)

$c_1 : \neg A_1 \vee A_2$ √

$c_2 : \neg A_1 \vee A_3 \vee A_9$ √

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

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

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

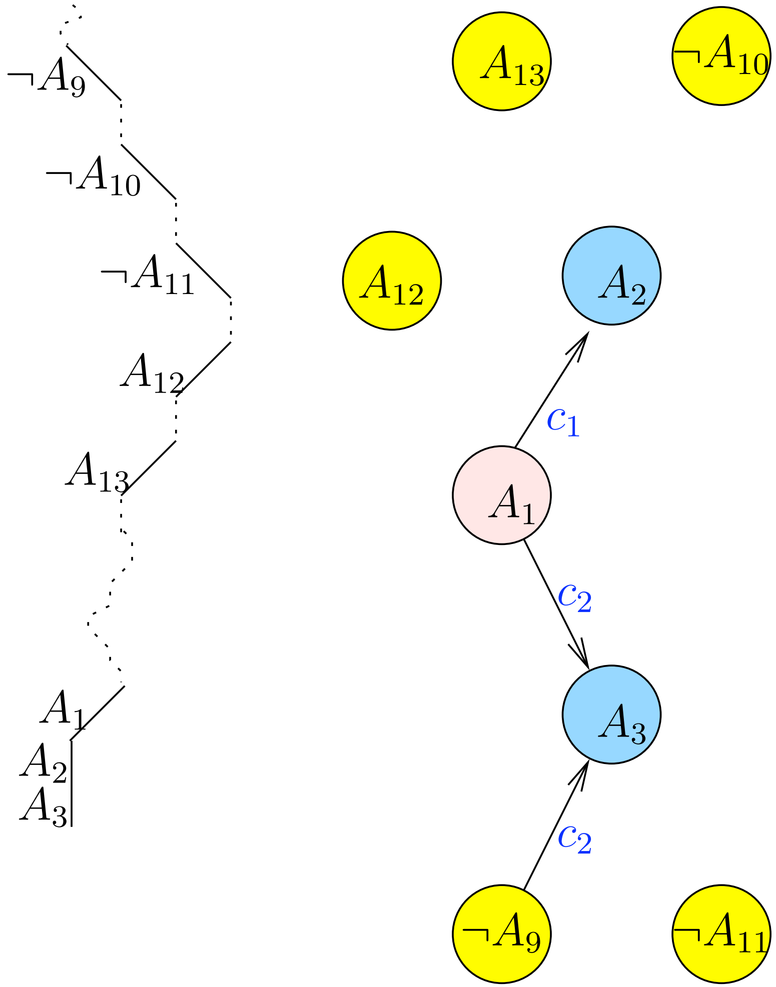
$c_6 : \neg A_5 \vee \neg A_6$

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

$c_8 : A_1 \vee A_8$ √

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

...



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

The original backjumping strategy – example (5)

$c_1 : \neg A_1 \vee A_2$ √

$c_2 : \neg A_1 \vee A_3 \vee A_9$ √

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

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

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

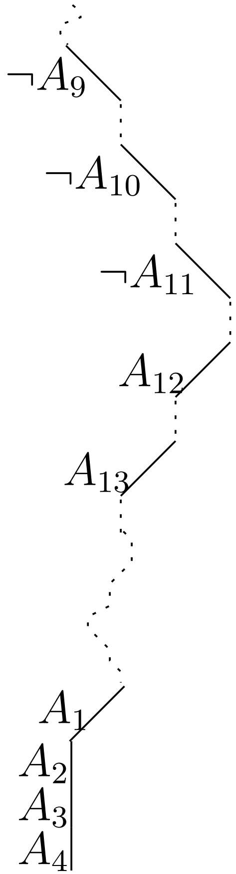
$c_6 : \neg A_5 \vee \neg A_6$

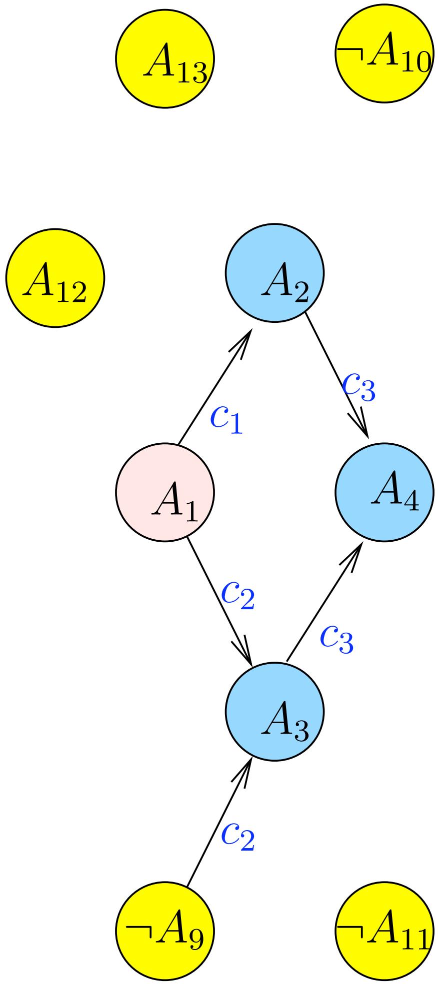
$c_7 : A_1 \vee A_7 \vee \neg A_{12}$ √

$c_8 : A_1 \vee A_8$ √

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

...





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

The original backjumping strategy – example (6)

$c_1 : \neg A_1 \vee A_2$ √

$c_2 : \neg A_1 \vee A_3 \vee A_9$ √

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

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

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

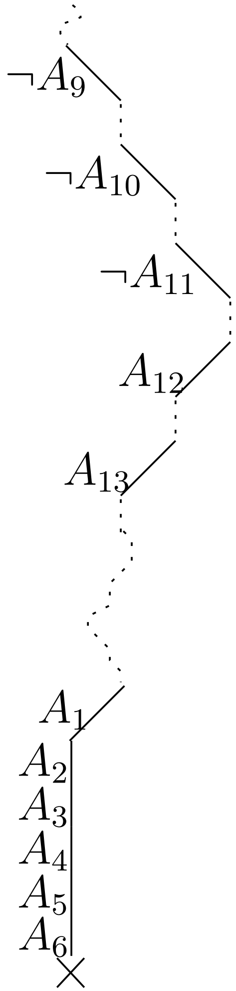
$c_6 : \neg A_5 \vee \neg A_6$ ×

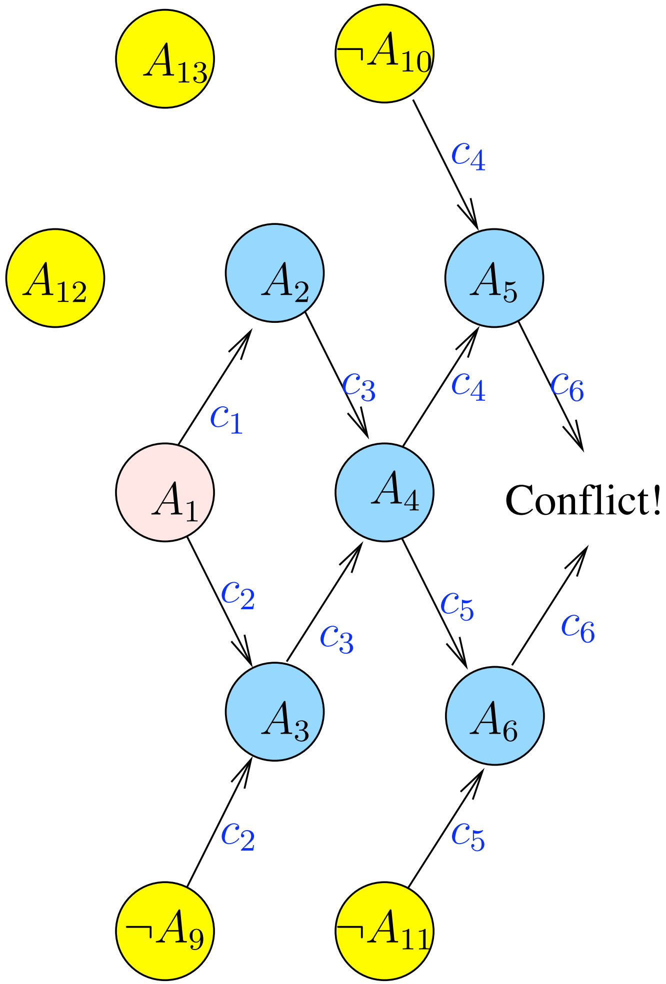
$c_7 : A_1 \vee A_7 \vee \neg A_{12}$ √

$c_8 : A_1 \vee A_8$ √

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

...





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

(unit A_5, A_6) $\Longrightarrow$ conflict

The original backjumping strategy – example (7)

$c_1 : \neg A_1 \vee A_2$ √

$c_2 : \neg A_1 \vee A_3 \vee A_9$ √

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

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

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

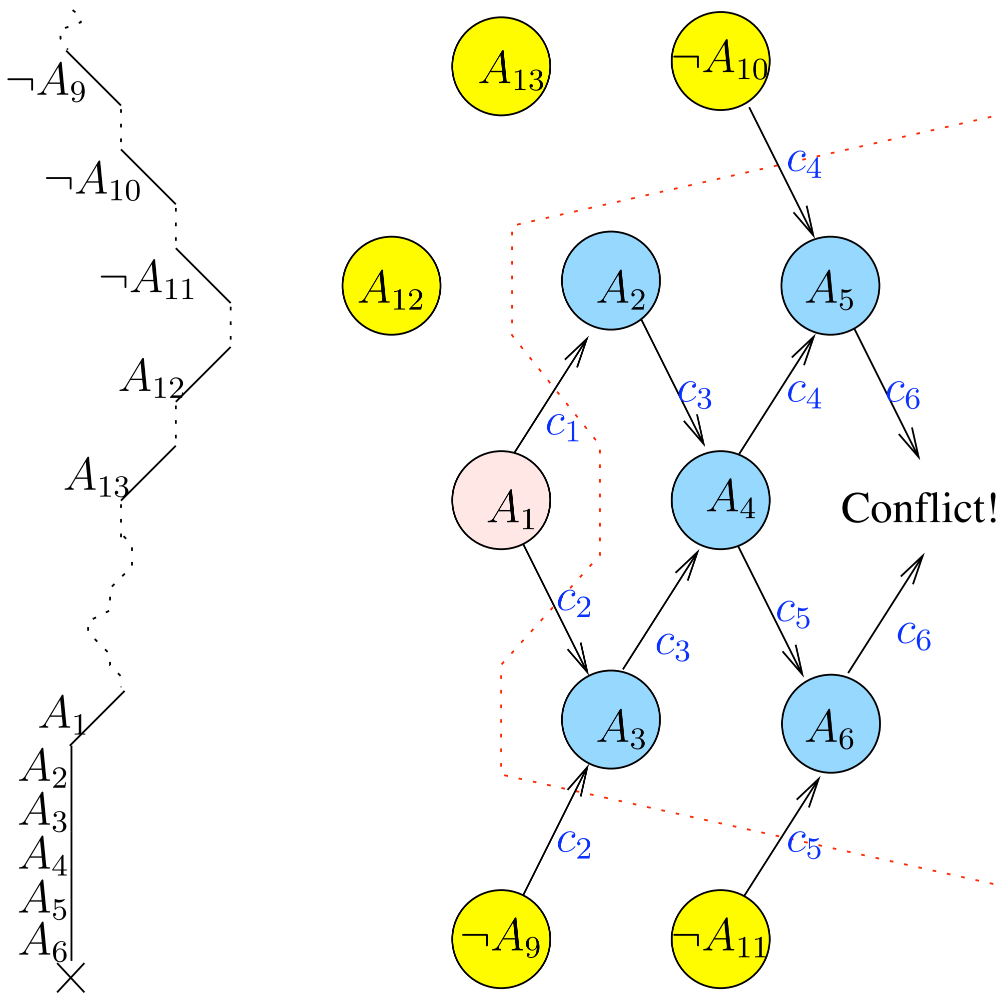
$c_6 : \neg A_5 \vee \neg A_6$ ×

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

$c_8 : A_1 \vee A_8$ √

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

...



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

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

Implementation of the implication graph

- ▷ an implication graph is implemented by tagging each non-decision literal in the stack with its antecedent clause
- ▷ (the partition representing) a conflict set is constructed from the conflict by traversing backwards the implication graph
- ▷ a conflict set can be constructed starting from the conflicting clause, each time resolving the current clause with the antecedent clause of one of its literals l
  (undo the unit propagation of l)

Building a conflict set/clause by resolution

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

   until C verifies some given termination criteria
   (e.g., until C contains only decision literals)

$$\frac{\neg A_4 \vee A_6 \vee A_{11} \qquad \overbrace{\neg A_5 \vee \neg A_6}^{Conflicting\ cl.}}{\neg A_4 \vee \neg A_5 \vee A_{11}} \ (A_6$$

$$\frac{\neg A_4 \vee A_5 \vee A_{10} \qquad \neg A_4 \vee \neg A_5 \vee A_{11}}{\neg A_4 \vee A_{10} \vee A_{11}} \ (A_5)$$

$$\frac{\neg A_2 \vee \neg A_3 \vee A_4 \qquad \neg A_4 \vee A_{10} \vee A_{11}}{\neg A_2 \vee \neg A_3 \vee A_{10} \vee A_{11}} \ (A_4)$$

$$\frac{\neg A_1 \vee A_3 \vee A_9 \qquad \neg A_2 \vee \neg A_3 \vee A_{10} \vee A_{11}}{\neg A_2 \vee \neg A_1 \vee A_9 \vee A_{10} \vee A_{11}} \ (A_3)$$

$$\frac{\neg A_1 \vee A_2 \qquad \neg A_2 \vee \neg A_1 \vee A_9 \vee A_{10} \vee A_{11}}{\neg A_1 \vee A_9 \vee A_{10} \vee A_{11}} \ (A_2)$$

The original backjumping strategy – example (7)

$c_1 : \neg A_1 \vee A_2$ ✓

$c_2 : \neg A_1 \vee A_3 \vee A_9$ ✓

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

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

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

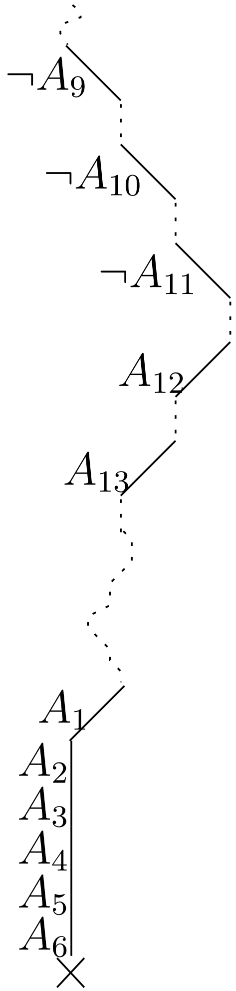
$c_6 : \neg A_5 \vee \neg A_6$ ×

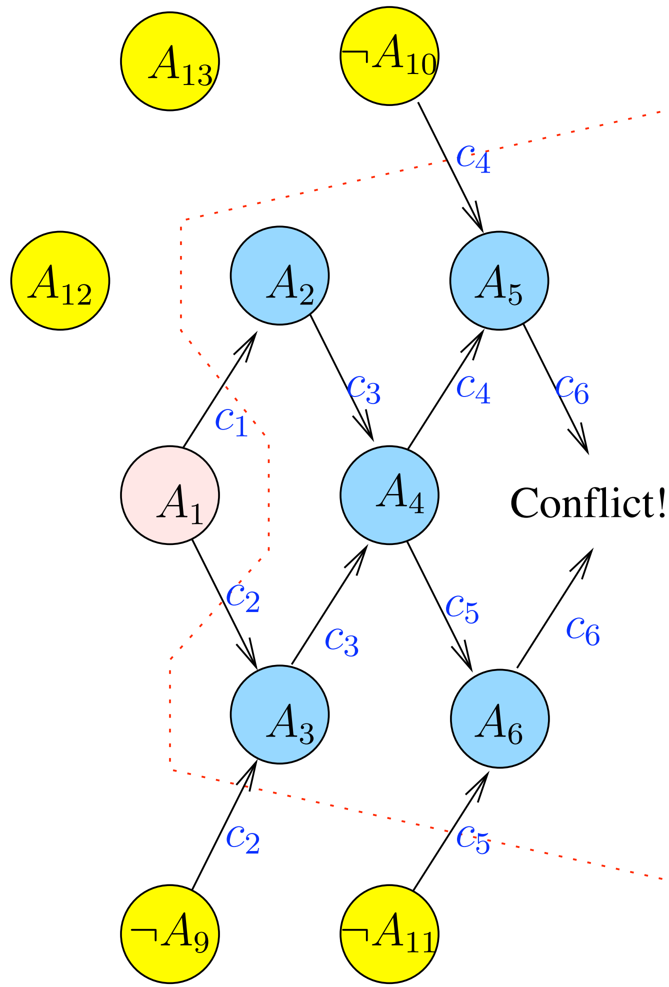
$c_7 : A_1 \vee A_7 \vee \neg A_{12}$ ✓

$c_8 : A_1 \vee A_8$ ✓

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

...





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

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

The original backjumping strategy – example (8)

$c_1 : \neg A_1 \vee A_2$

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

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

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

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

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

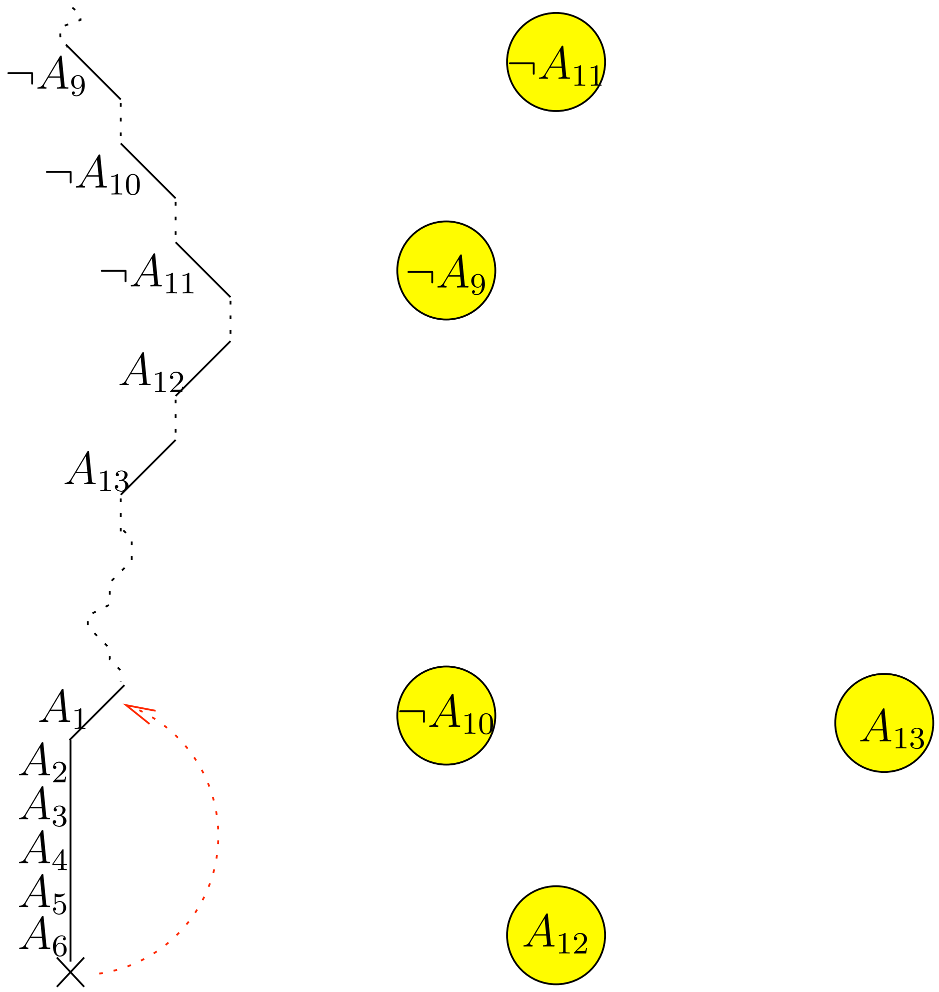
$c_7 : A_1 \vee A_7 \vee \neg A_{12}$

$c_8 : A_1 \vee A_8$

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

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

...



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

$\Longrightarrow$ backtrack up to A_1

The original backjumping strategy – example (9)

$c_1 : \neg A_1 \vee A_2$ √

$c_2 : \neg A_1 \vee A_3 \vee A_9$ √

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

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

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

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

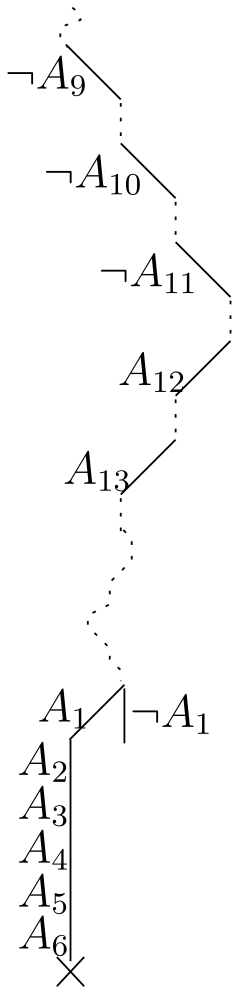
$c_7 : A_1 \vee A_7 \vee \neg A_{12}$

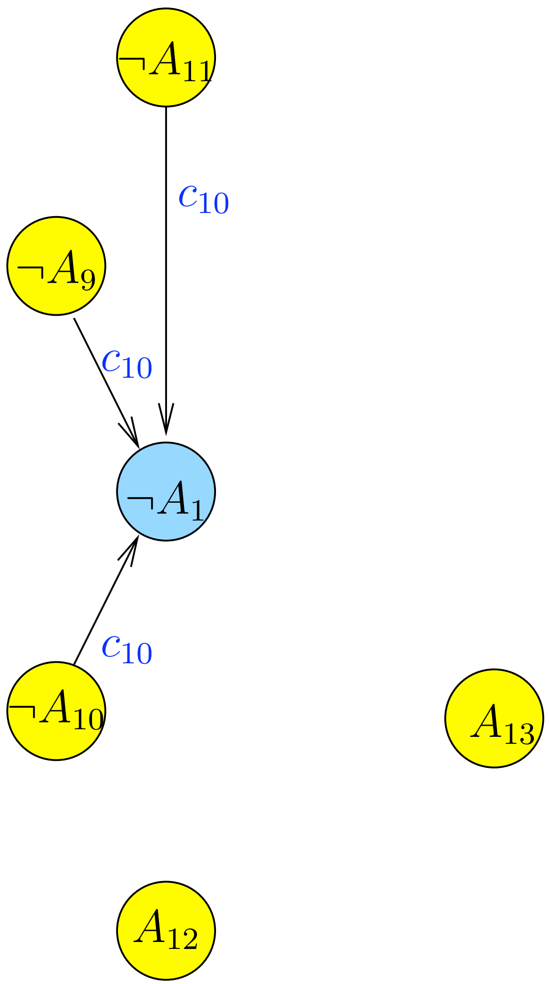
$c_8 : A_1 \vee A_8$

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

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

...





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

(unit $\neg A_1$)

The original backjumping strategy – example (10)

$c_1 : \neg A_1 \vee A_2$ √

$c_2 : \neg A_1 \vee A_3 \vee A_9$ √

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

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

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

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

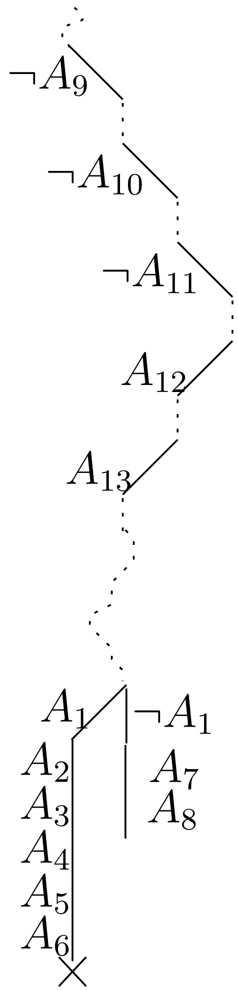
$c_7 : A_1 \vee A_7 \vee \neg A_{12}$ √

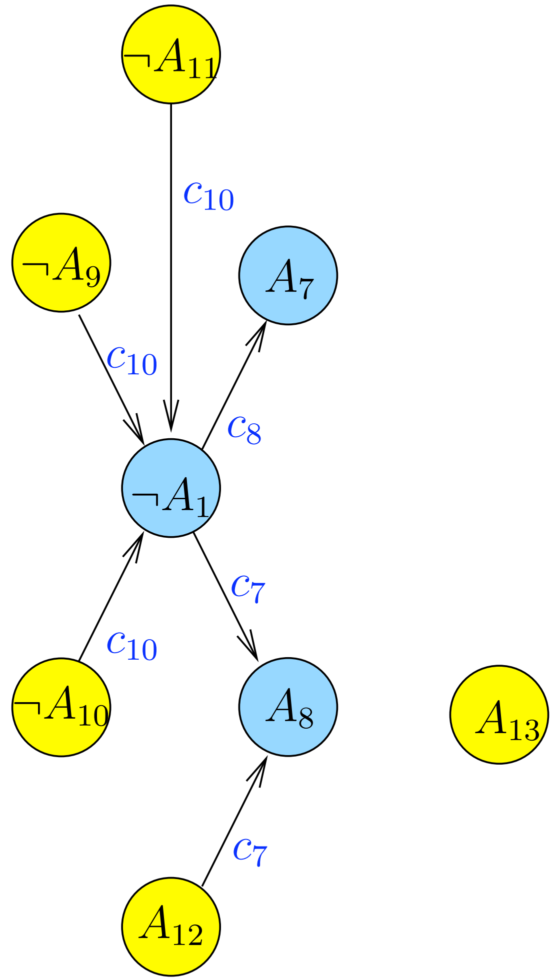
$c_8 : A_1 \vee A_8$ √

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

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

...





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

(unit A_7, A_8)

The original backjumping strategy – example (11)

$c_1 : \neg A_1 \vee A_2$ √

$c_2 : \neg A_1 \vee A_3 \vee A_9$ √

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

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

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

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

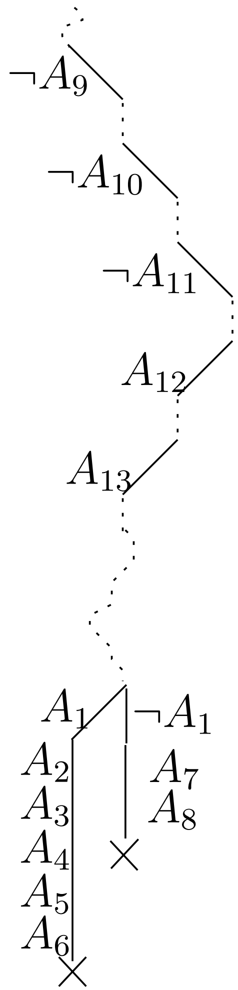
$c_7 : A_1 \vee A_7 \vee \neg A_{12}$ √

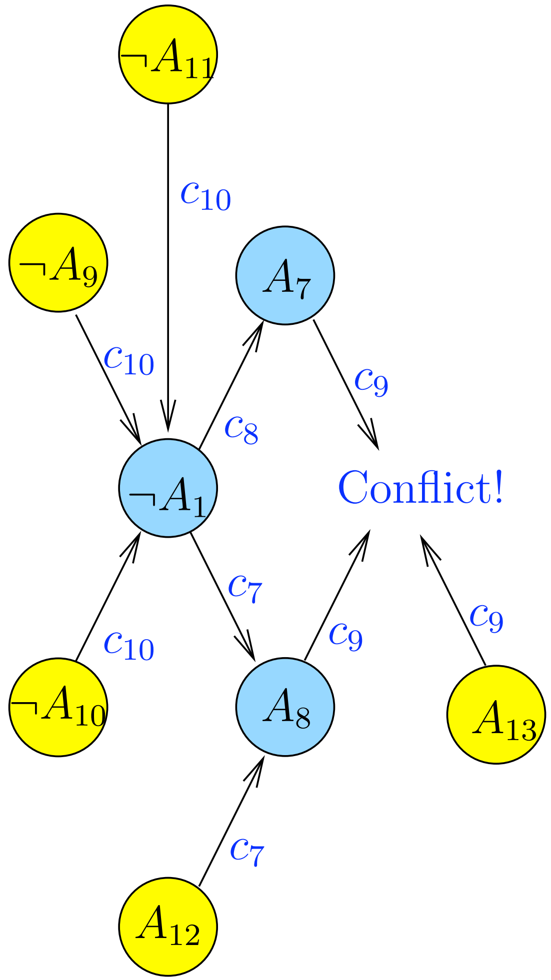
$c_8 : A_1 \vee A_8$ √

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

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

...





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

Conflict!

The original backjumping strategy – example (12)

$c_1 : \neg A_1 \vee A_2$ ✓

$c_2 : \neg A_1 \vee A_3 \vee A_9$ ✓

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

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

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

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

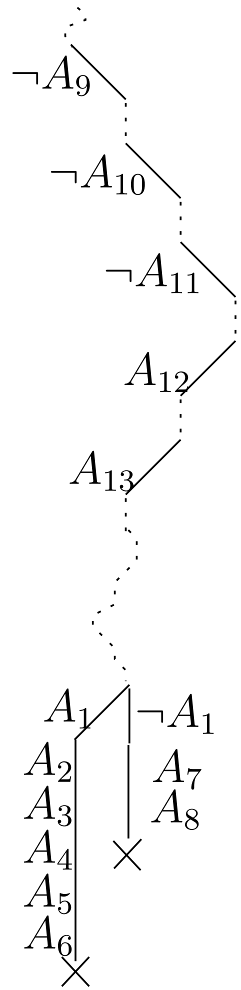
$c_7 : A_1 \vee A_7 \vee \neg A_{12}$ ✓

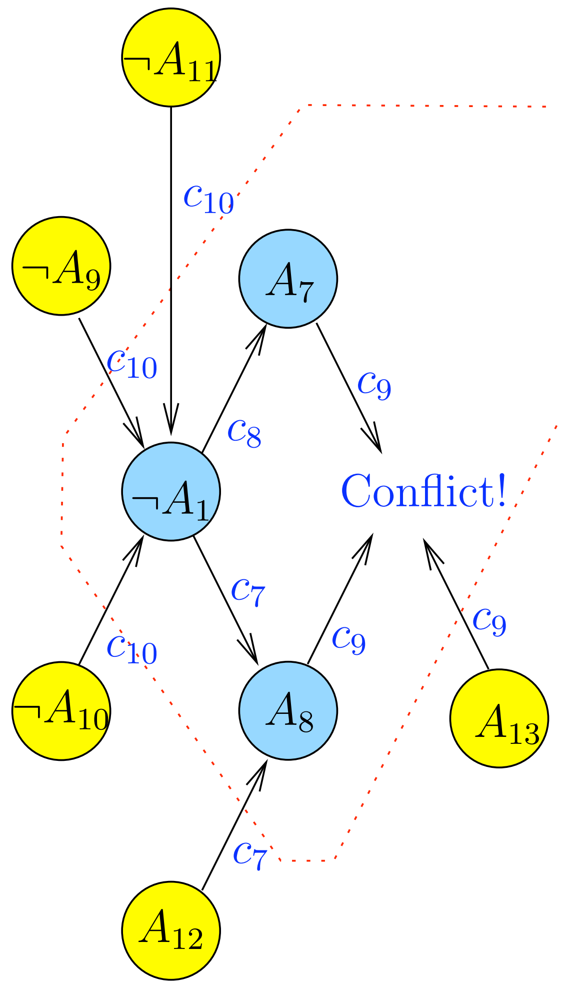
$c_8 : A_1 \vee A_8$ ✓

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

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

...





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

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

The original backjumping strategy – example (13)

$c_1 : \neg A_1 \vee A_2$

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

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

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

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

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

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

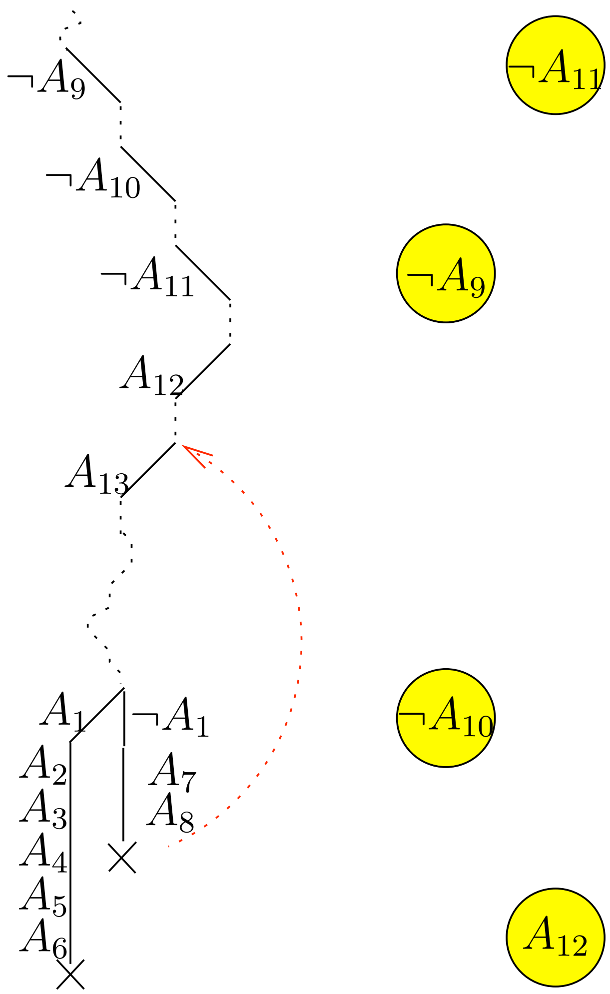
$c_8 : A_1 \vee A_8$

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

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

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

...



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

The original backjumping strategy – example (14)

$c_1 : \neg A_1 \vee A_2$ √

$c_2 : \neg A_1 \vee A_3 \vee A_9$ √

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

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

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

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

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

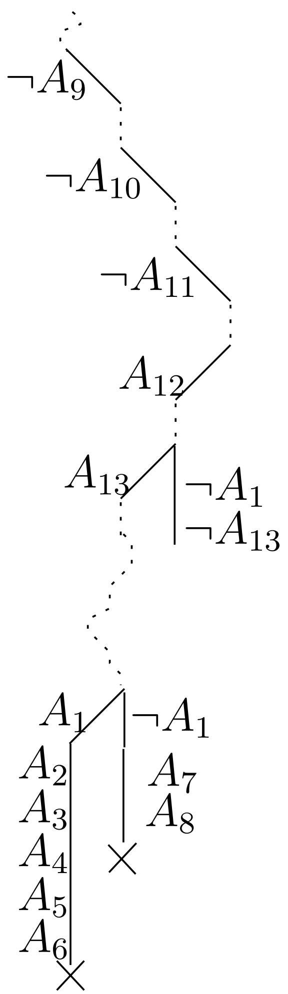
$c_8 : A_1 \vee A_8$

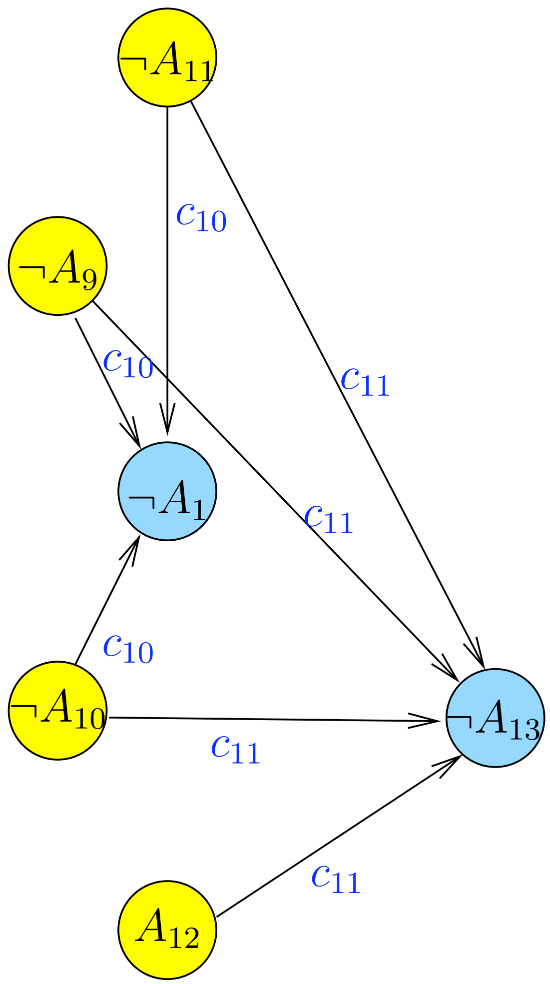
$c_9 : \neg A_7 \vee \neg A_8 \vee \neg A_{13}$ √

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

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

...





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

Learning [4, 46]

Idea: When a conflict set C is revealed, then $\neg C$ added to φ
$\Longrightarrow$DPLL will no more generate an assignment containing C: when $|C| - 1$ literals in C are assigned, the other is set $\bot$

Drastic pruning of the search!!!

Learning – example (cont.)

$c_1 : \neg A_1 \vee A_2$

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

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

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

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

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

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

$c_8 : A_1 \vee A_8$

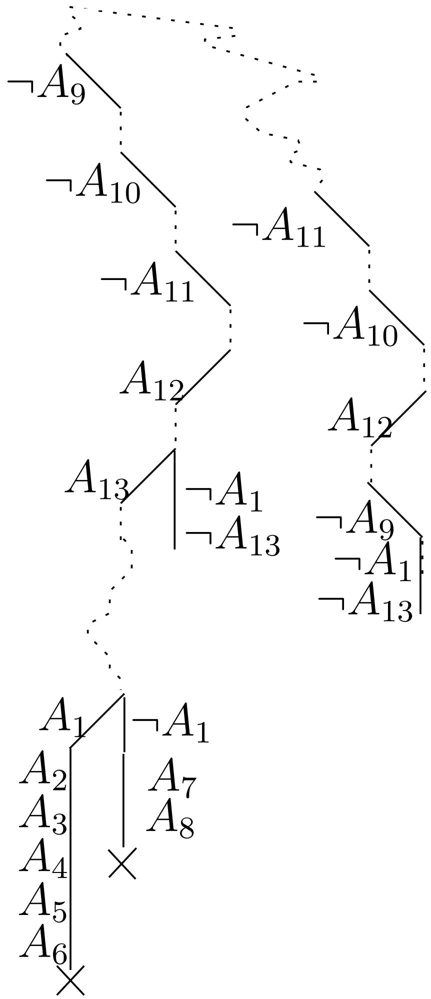
$c_9 : \neg A_7 \vee \neg A_8 \vee \neg A_{13}$ $\surd$

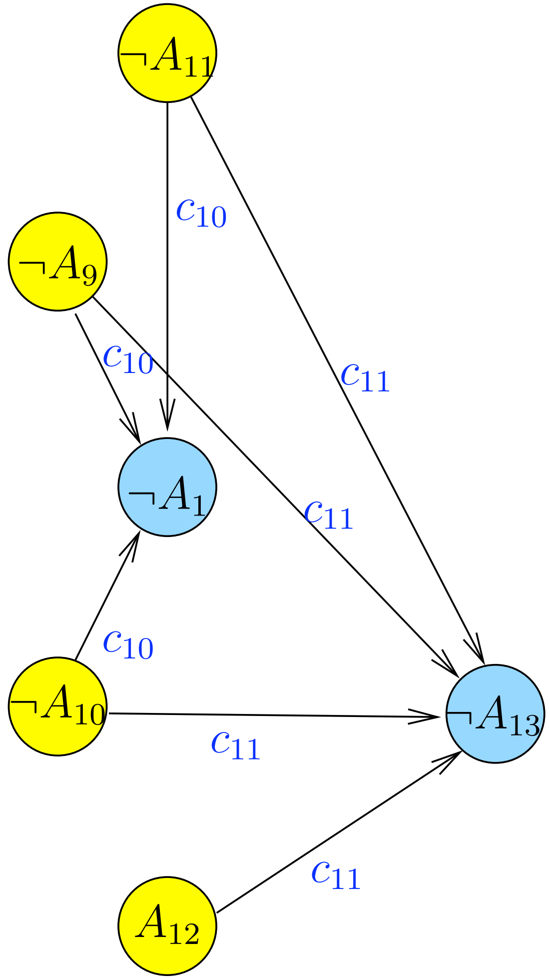
$c_{10} : A_9 \vee A_{10} \vee A_{11} \vee \neg A_1$ $\surd$

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

...

$\Longrightarrow$ Unit: $\{\neg A_1, \neg A_{13}\}$





State-of-the-art backjumping and learning [54]

▷ Idea: when a branch μ fails,

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

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

▷ Backjumping: allows for climbing up to many decision levels in the stack
⟹may avoid lots of redundant search

- intuition: "go back to the oldest decision where you'd have done something different if only you had known C"

▷ Learning: in future branches, when all-but-one literals in η are assigned, the remaining literal is assigned to false by unit-propagation:
⟹avoid finding the same conflict again

- intuition: "when you're about to repeat the mistake, do the opposite of the last step"

State-of-the-art in backjumping & learning [54]

▷ A node l in an implication graph is an unique implication point (UIP) for the last decision level iff any path from the last decision node to both the conflict nodes passes through l.

- the most recent decision node is an UIP (last UIP)
- all other UIP's have been assigned after the most recent decision

State-of-the-art in backjumping & learning [54]

First Unique Implication Point (1st UIP) strategy:

- 1st UIP strategy: adopt the partition involving the 1st UIP for the last decision level.
- corresponds to consider the first clause encountered containing one literal of the current level (1st UIP).

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

1st UIP strategy – example (7)

$c_1 : \neg A_1 \vee A_2$ ✓

$c_2 : \neg A_1 \vee A_3 \vee A_9$ ✓

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

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

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

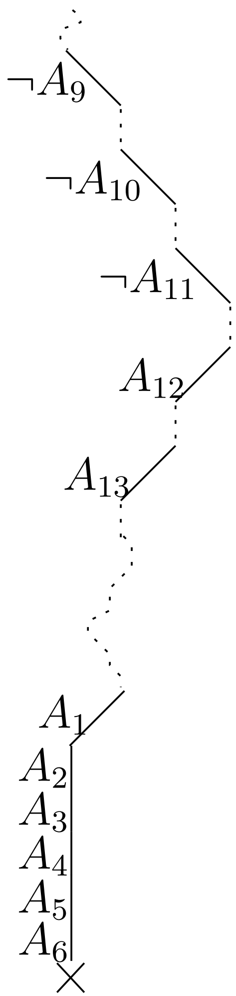
$c_6 : \neg A_5 \vee \neg A_6$ ×

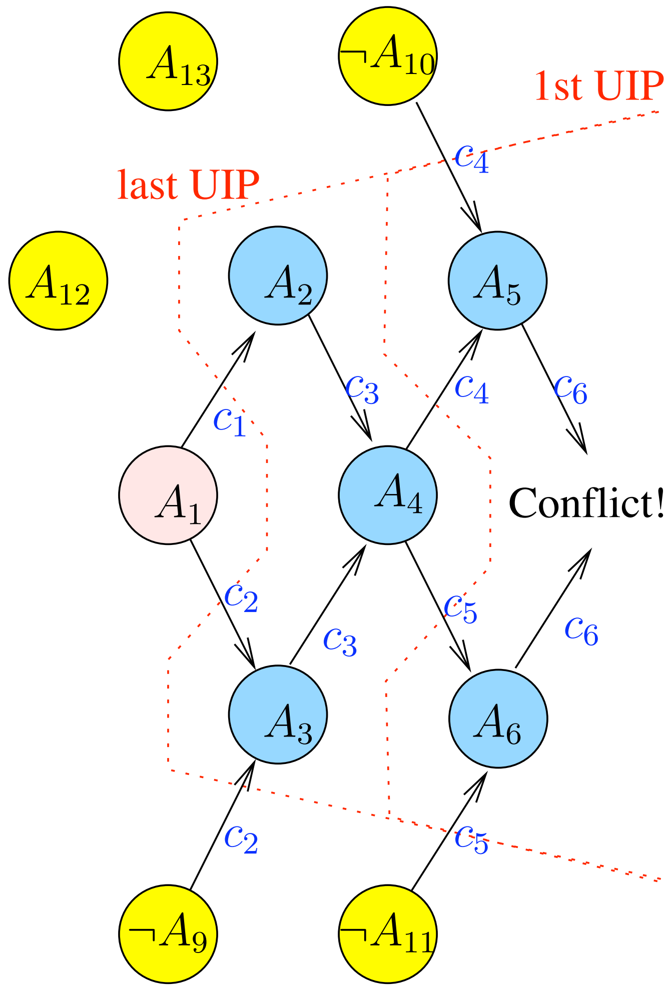
$c_7 : A_1 \vee A_7 \vee \neg A_{12}$ ✓

$c_8 : A_1 \vee A_8$ ✓

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

...





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

1st UIP strategy and backjumping [54]

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

E.g.: $c_{10} := A_{10} \vee A_{11} \vee \neg A_4$

$\Longrightarrow$backtrack to A_{11}, then assign $\neg A_4$

1st UIP strategy – example (7)

$c_1 : \neg A_1 \vee A_2$ ✓

$c_2 : \neg A_1 \vee A_3 \vee A_9$ ✓

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

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

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

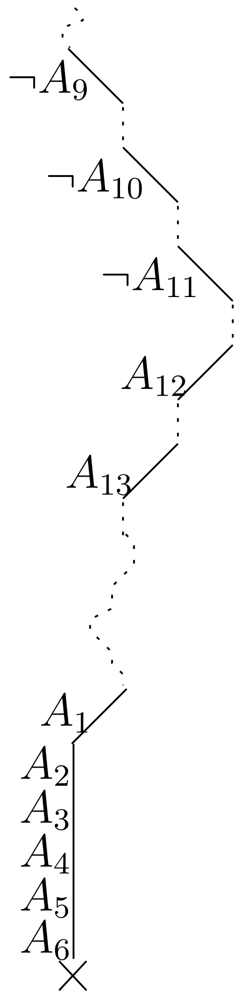
$c_6 : \neg A_5 \vee \neg A_6$ ×

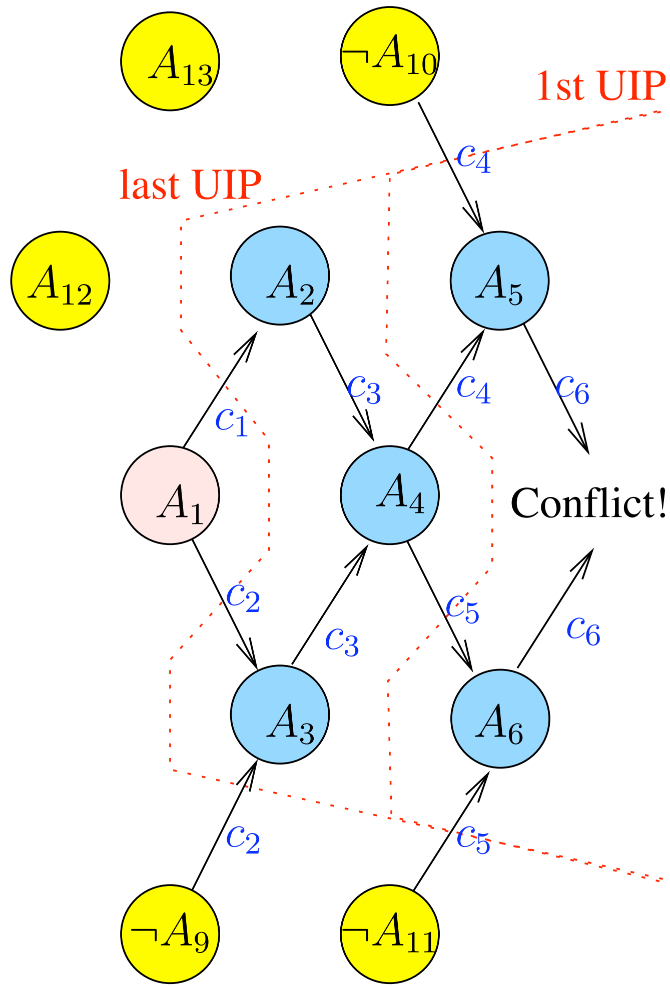
$c_7 : A_1 \vee A_7 \vee \neg A_{12}$ ✓

$c_8 : A_1 \vee A_8$ ✓

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

...





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

1st UIP strategy – example (8)

$c_1 : \neg A_1 \vee A_2$

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

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

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

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

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

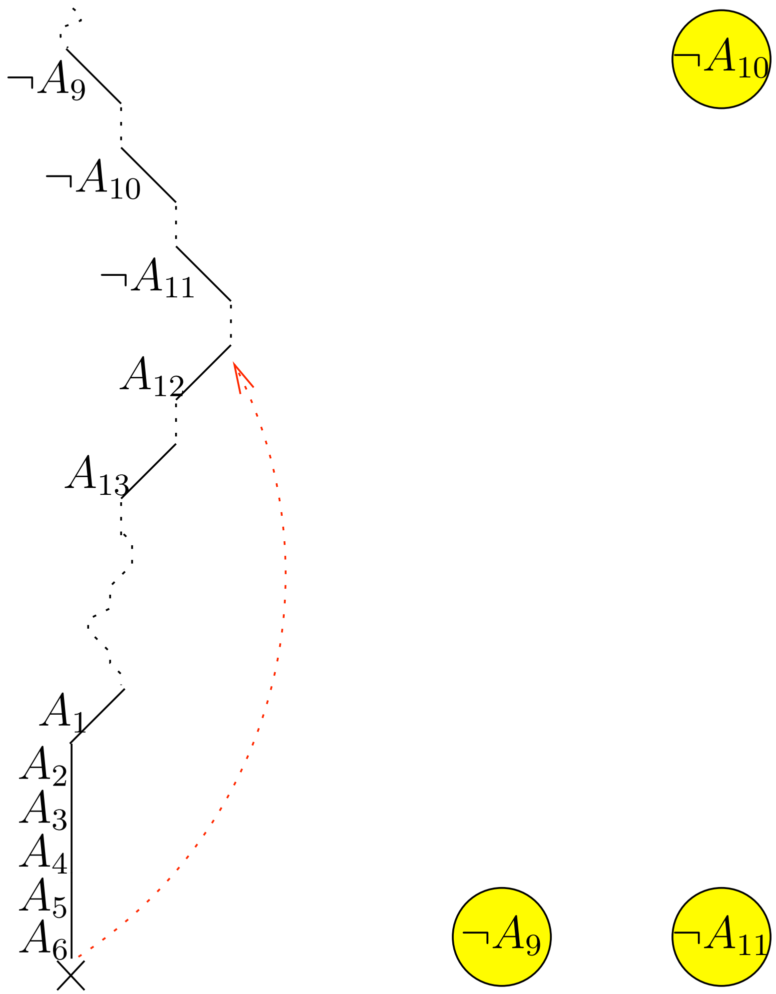
$c_7 : A_1 \vee A_7 \vee \neg A_{12}$

$c_8 : A_1 \vee A_8$

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

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

...



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

1st UIP strategy – example (9)

$c_1 : \neg A_1 \vee A_2$

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

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

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

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

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

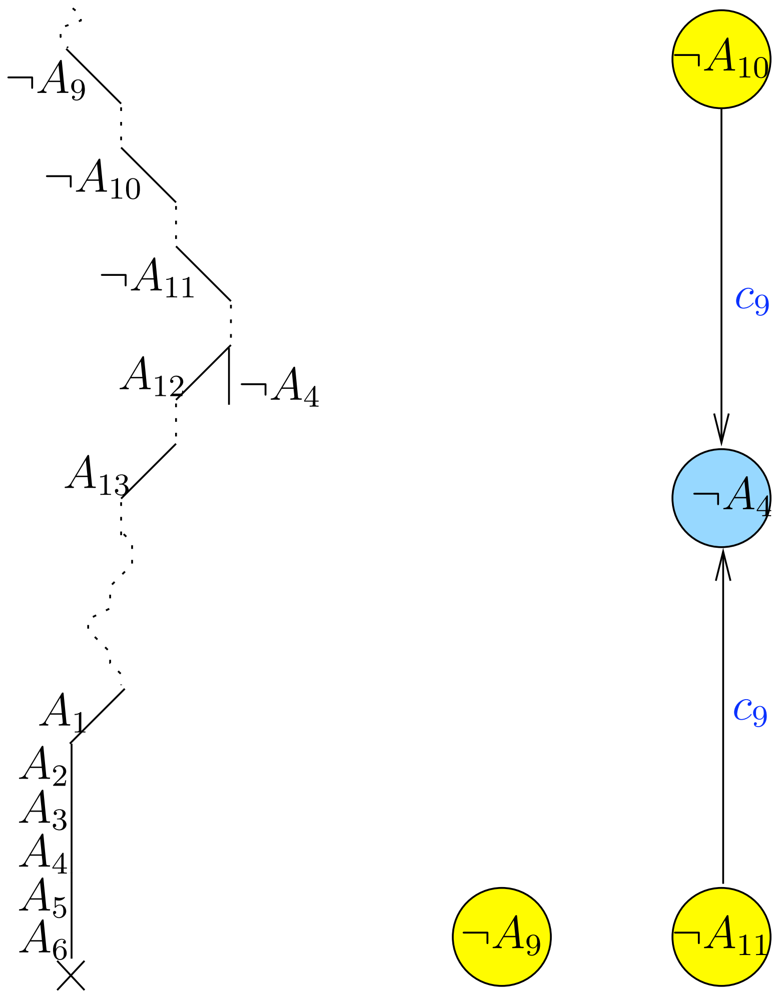
$c_7 : A_1 \vee A_7 \vee \neg A_{12}$

$c_8 : A_1 \vee A_8$

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

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

...



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

1st UIP strategy and backjumping – intuition

▷ An UIP is a single reason implying the conflict at the current level

▷ substituting the 1st UIP for the last UIP

- does not enlarge the conflict
- may require involving less decision literals from other levels

▷ by backtracking to the most recent decision level of the variables in the conflict clause which are not the UIP:

- jump higher
- allows for assigning (the negation of) the UIP as high as possible in the search tree.

Remark: the "quality" of conflict sets

▷ Different ideas of "good" conflict set

- Backjumping: if causes the highest backjump ("local" role)
- Learning: if causes the maximum pruning ("global" role)

▷ Many different strategies implemented (see, e.g., [4, 46, 54])

Drawbacks of Learning

▷ Prunes drastically the search.

▷ Problem: may cause a blowup in space

$\Longrightarrow$techniques to drop learned clauses when necessary

- according to their size
- according to their activity.

Restarts [24]

(according to some strategy) restart DPLL

- ▷ abandon the current search tree and reconstruct a new one
- ▷ The clauses learned prior to the restart are still there after the restart and can help pruning the search space
- ▷ may significantly reduce the overall search space

What is missing?

...Many things:

1. Data structures for effective BCP (binary clauses, two literal watching, BCP ordering)
2. Forgetting policies
3. Effective use of L1 and L2 cache
4. Parallel and manycore SAT solvers
5. Phase saving
6. ...

Content

Advanced functionalities

Advanced SAT functionalities (very important in formal verification):

- ▷ Building proofs of unsatisfiability
- ▷ Extracting unsatisfiable Cores

Building Proofs of Unsatisfiability

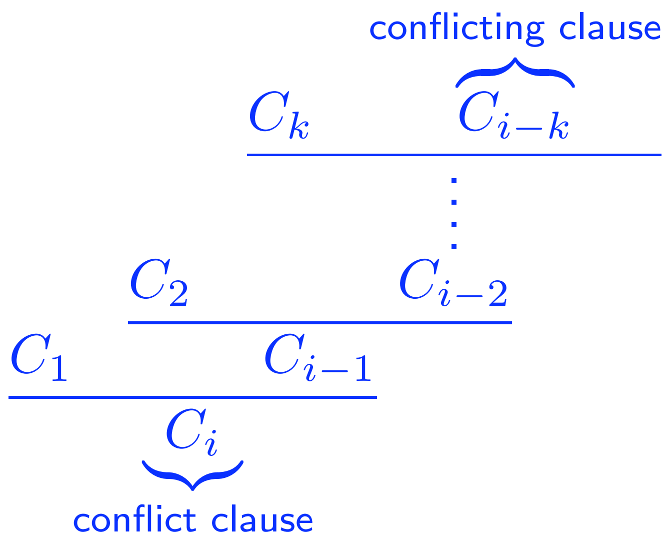
▷ When φ is unsat, it is very important to build a (resolution) proof of unsatisfiability:

- to verify the result of the solver
- to understand a "reason" for unsatisfiability
- to build unsatisfiable cores and interpolants

▷ can be built by keeping track of the resolution steps performed when constructing the conflict clauses.

Building Proofs of Unsatisfiability

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

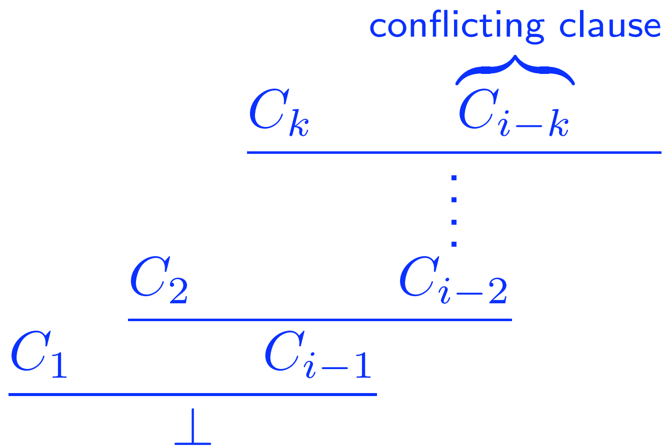


- $C_1, ..., C_k$, and C_{i-k} can be original or learned clauses
- each resolution (sub)proof can be easily tracked:

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

Building Proofs of Unsatisfiability

- ... in particular, if φ is unsatisfiable, the last step produces "false" as conflict clause:

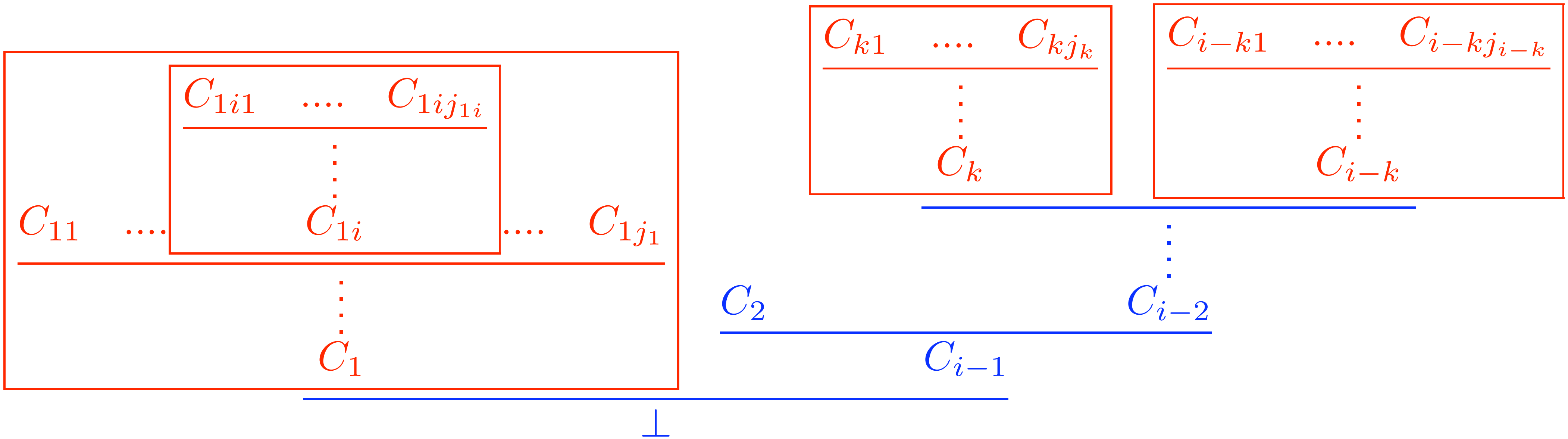


- note: $C_1 = l$, $C_{i-1} = \neg l$ for some literal l
- $C_1, ..., C_k$, and C_{i-k} can be original or learned clauses...

Building Proofs of Unsatisfiability

Starting from the previous proof of unsatisfiability, repeat recursively:

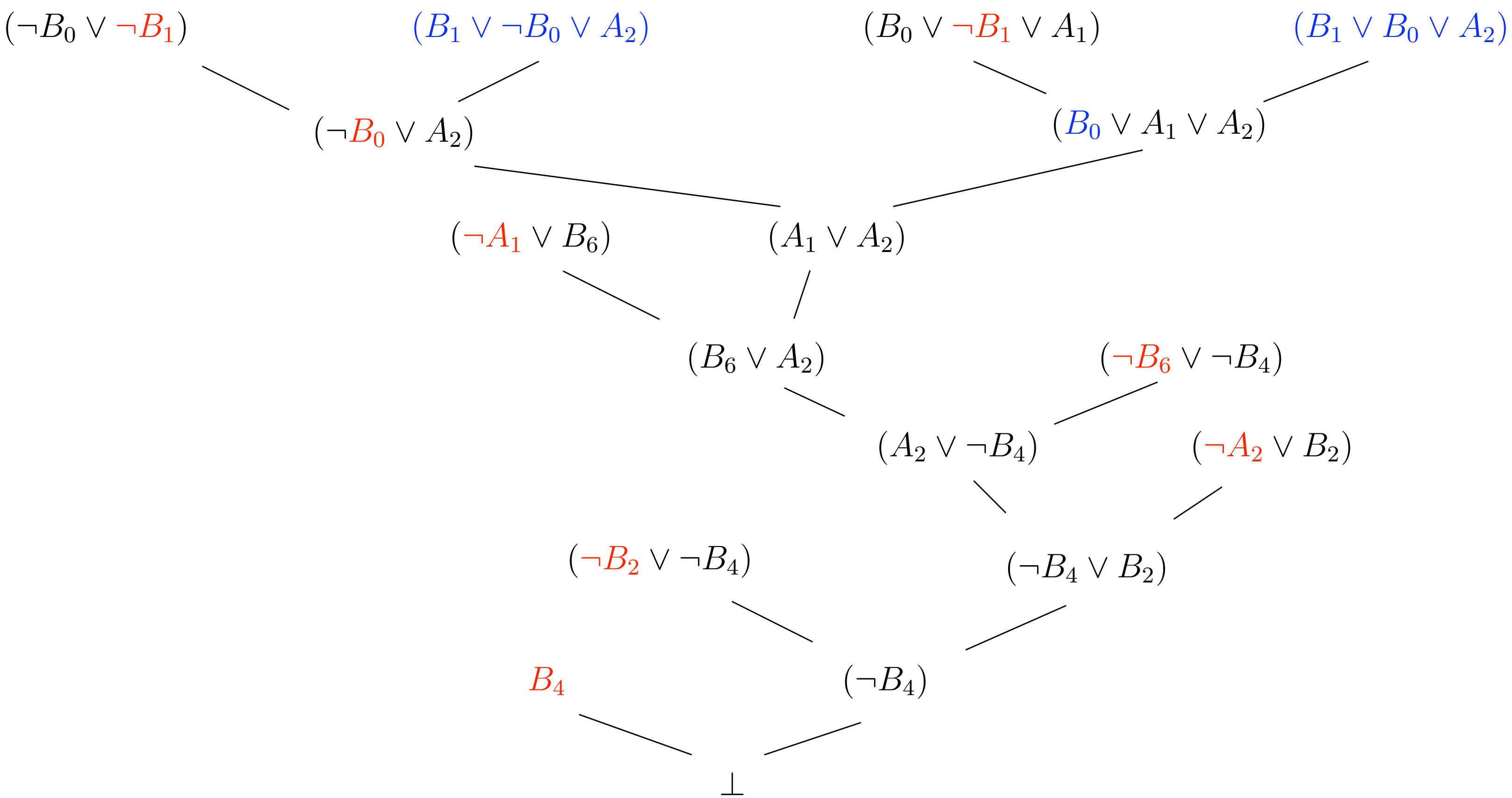
▷ for every learned leaf clause C_i, substitute C_i with the resolution proof generating it until all leaf clauses are original clauses



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

Building Proofs of Unsatisfiability: example

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



Extraction of unsatisfiable cores

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

▷ Lots of literature on the topic [56, 30, 32, 38, 53, 25, 19, 6]

▷ We recognize two main approaches:

- Proof-based approach [56]: byproduct of finding a resolution proof
- Assumption-based approach [30]: use extra variables labeling clauses

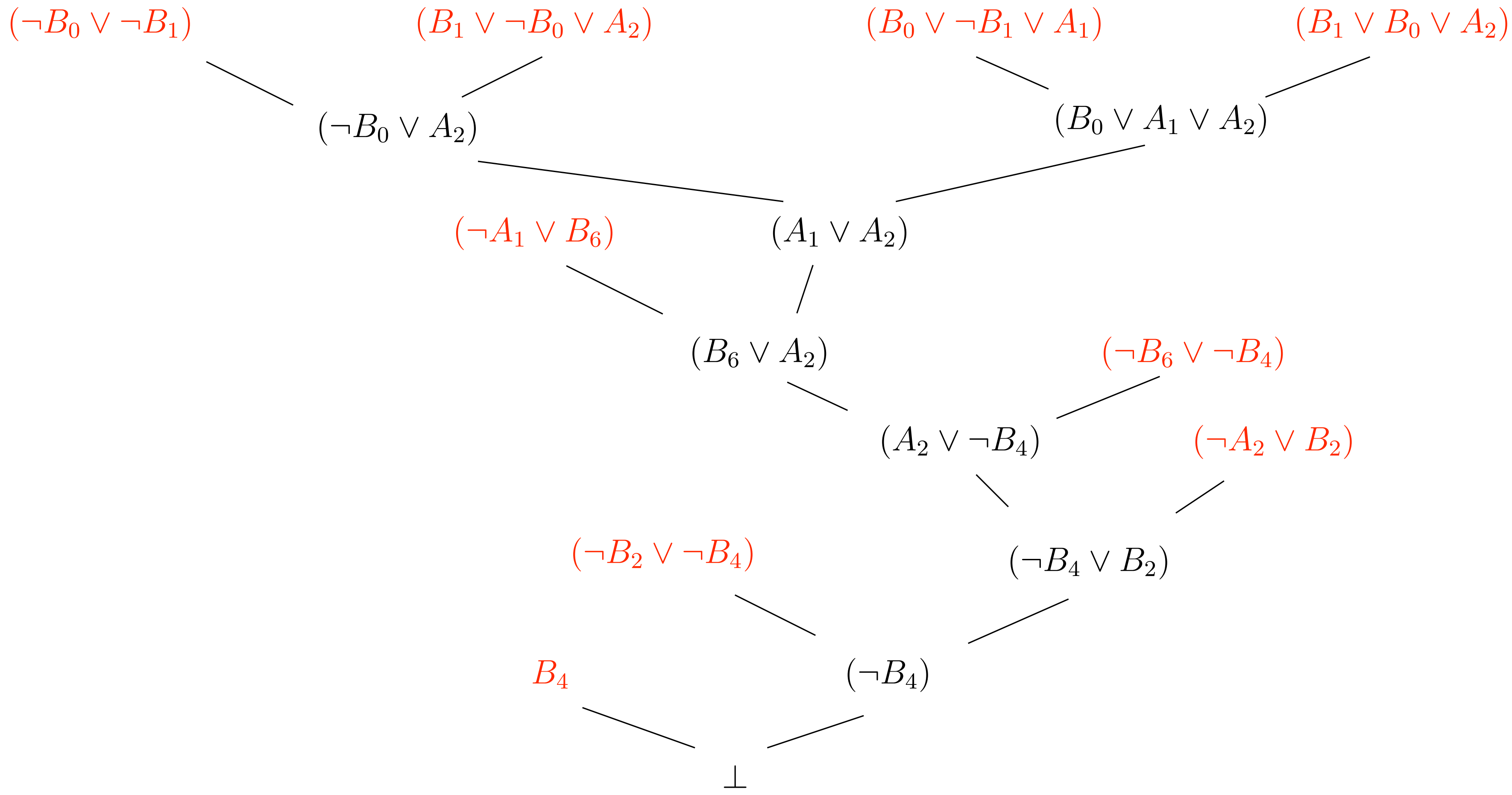
▷ many optimizations for further reducing the size of the core:

- repeat the process up to fixpoit
- remove clauses one-by one, until satisfiability is obtained
- combinations of the two processed above
- ...

The proof-based approach to unsat-core extraction

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

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



The assumption-based approach to unsat-core extraction

Based on the following process:

1. each clause C_i is substituted by $S_i \rightarrow C_i$, s.t. S_i fresh "selector" variable
2. before starting the search each S_i is forced to true.
3. final conflict clause at dec. level 0: $\bigvee_j \neg S_j$
   $\Longrightarrow \{C_j\}_j$ is the unsat core

The assumption-based approach to unsat-core extraction

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

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

$B_4 \wedge (A_2 \vee B_5) \wedge (\neg B_6 \vee \neg B_4) \wedge (B_6 \vee \neg A_1) \wedge B_7$

add selector variables:

$S_1 \rightarrow (B_0 \vee \neg B_1 \vee A_1) \wedge S_2 \rightarrow (B_0 \vee B_1 \vee A_2) \wedge S_3 \rightarrow (\neg B_0 \vee B_1 \vee A_2) \wedge$

$S_4 \rightarrow (\neg B_0 \vee \neg B_1) \wedge S_5 \rightarrow (\neg B_2 \vee \neg B_4) \wedge S_6 \rightarrow (\neg A_2 \vee B_2) \wedge (S_7 \rightarrow \neg A_1 \vee B_3) \wedge$

$S_8 \rightarrow B_4 \wedge S_9 \rightarrow (A_2 \vee B_5) \wedge S_{10} \rightarrow (\neg B_6 \vee \neg B_4) \wedge S_{11} \rightarrow (B_6 \vee \neg A_1) \wedge S_{12} \rightarrow B_7$

The conflict analysis returns:

$\neg S_1 \vee \neg S_2 \vee \neg S_3 \vee \neg S_4 \vee \neg S_5 \vee \neg S_6 \vee \neg S_8 \vee \neg S_{10} \vee \neg S_{11}$,

corresponding to the unsat core:

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

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

$B_4 \wedge (\neg B_6 \vee \neg B_4) \wedge (B_6 \vee \neg A_1)$

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

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

[4] R. J. Bayardo, Jr. and R. C. Schrag. Using CSP Look-Back Techniques to Solve Real-World SAT instances. In *Proc. AAAI'97*, pages 203–208. AAAI Press, 1997.

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

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

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

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

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

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

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

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

[13] 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.

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

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

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

[17] A. Van Gelder. A satisfiability tester for non-clausal propositional calculus. *Information and Computation*, 79:1–21, October 1988.

[18] 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.

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

[20] 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.

[21] 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.

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

[23] E. Giunchiglia and A. Tacchella, editors. *Sixth International Conference on Theory and Applications of Satisfiability Testing* , volume 2919 of *LNCS*. Springer, May 2003.

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

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

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

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

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

[29] Chu Min Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 366–371, 1997.

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

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

[32] Ken McMillan and Nina Amla. Automatic abstraction without counterexamples. In *Proc. of TACAS*, 2003.

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

[34] 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.

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

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

[37] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Abstract DPLL and abstract DPLL modulo theories. In F. Baader and A. Voronkov, editors, *Proceedings of the 11th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR'04), Montevideo, Uruguay*, volume 3452 of *Lecture Notes in Computer Science*, pages 36–50. Springer, 2005.

[38] Y. Oh, M. N. Mneimneh, Z. S. Andraus, K. A. Sakallah, and I. L. Markov. Amuse: A Minimally-Unsatisfiable Subformula Extractor. In *Proc. DAC'04*. ACM/IEEE, 2004.

[39] D.A. Plaisted and S. Greenbaum. A Structure-preserving Clause Form Translation. *Journal of*

Symbolic Computation, 2:293–304, 1986.

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

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

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

[43] 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.

[44] B. Selman, H. Kautz, and B. Cohen. Local Search Strategies for Satisfiability Testing. In *Cliques, Coloring, and Satisfiability*, volume 26 of *DIMACS*, pages 521–532, 1996.

[45] 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.

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

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

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

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

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

[51] 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.

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

[53] J. Zhang, S. Li, and S. Shen. Extracting Minimum Unsatisfiable Cores with a Greedy Genetic Algorithm. In *Proc. ACAI*, volume 4304 of *LNCS*. Springer, 2006.

[54] L. Zhang, C. F. Madigan, M. H. Moskewicz, and S. Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *ICCAD '01: Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, pages 279–285, Piscataway, NJ, USA, 2001. IEEE Press.

[55] L. Zhang and S. Malik. The quest for efficient boolean satisfiability solvers. In *Proc. CAV'02*, number 2404 in LNCS, pages 17–36. Springer, 2002.

[56] Lintao Zhang and Sharad Malik. Extracting small unsatisfiable cores from unsatisfiable boolean formula. In *Proc. of SAT*, 2003.

DISCLAIMER

The list of references above is by no means intended to be all-inclusive. The author of these slides apologizes both with the authors and with the readers for all the relevant works which are not cited here.

The papers (co)authored by the author of these slides are availlable at:
`http://www.dit.unitn.it/~rseba/publist.html`.

Related web sites:

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