Technische
Universität
Braunschweig

Institut für Betriebssysteme
und Rechnerverbund
Algorithmik

Algorithm Engineering
Lecture 9: SAT

SAT: Satisfiability (in Conjunctive Normal Form, CNF)

SAT: Satisfiability (in Conjunctive Normal Form, CNF)

SAT: Given

$$\varphi(x_1, \ldots, x_n) = \bigwedge_{j=1}^{k} \bigvee_{i=1}^{s_j} \ell_{ij} \text{ ; each } \ell_{ij} \in \{x_q, \overline{x_q}\} \text{ (conjunction of disjunction of literals),}$$

is there a satisfying assignment of the x_i?

SAT: Satisfiability (in Conjunctive Normal Form, CNF)

SAT: Given

$$\varphi(x_1, \ldots, x_n) = \bigwedge_{j=1}^{k} \bigvee_{i=1}^{s_j} \ell_{ij}$$; each $\ell_{ij} \in \{x_q, \overline{x_q}\}$ (conjunction of disjunction of literals), clause

is there a satisfying assignment of the x_i?

SAT: Satisfiability (in Conjunctive Normal Form, CNF)

each clause is essentially a ≥ 1-constraint

SAT: Given

$$\varphi(x_1, \ldots, x_n) = \bigwedge_{j=1}^{k} \bigvee_{i=1}^{s_j} \ell_{ij}$$; each $\ell_{ij} \in \{x_q, \overline{x_q}\}$ (conjunction of clause disjunction of literals),

is there a satisfying assignment of the x_i?

SAT: Satisfiability (in Conjunctive Normal Form, CNF)

each clause is essentially a ≥ 1-constraint

SAT: Given

$$\varphi(x_1, \ldots, x_n) = \bigwedge_{j=1}^{k} \bigvee_{i=1}^{s_j} \ell_{ij}$$; each $\ell_{ij} \in \{x_q, \overline{x_q}\}$ (conjunction of disjunction of literals),

clause

is there a satisfying assignment of the x_i?

In other words: pick at least one satisfied literal from each clause.

SAT: Satisfiability (in Conjunctive Normal Form, CNF)

each clause is essentially a ≥ 1-constraint

SAT: Given

$$\varphi(x_1, \ldots, x_n) = \bigwedge_{j=1}^{k} \bigvee_{i=1}^{s_j} \ell_{ij} \text{ ; each } \ell_{ij} \in \{x_q, \overline{x_q}\}$$ (conjunction of clause: disjunction of literals),

is there a satisfying assignment of the x_i?

In other words: pick at least one satisfied literal from each clause.

The original **NP-complete** problem!

Classical Backtracking & Propagation: DPLL

$\mathrm{DPLL}(X, C, \theta)$

Classical Backtracking & Propagation: DPLL

DPLL(X, C, θ)

while C contains pure literal ℓ:

$\theta = \theta \cup \{\ell \leftarrow \texttt{true}\}$

$C = \{c_i \mid c_i \in C, \ell \notin c_i\}$

Pure Literal Elimination

We can eliminate variables that occur only positively/only negatively (pure literals) by setting them to that value; that also removes all clauses containing them.

Classical Backtracking & Propagation: DPLL

DPLL(X, C, θ)

while C contains pure literal ℓ:

$\theta = \theta \cup \{\ell \leftarrow \texttt{true}\}$

$C = \{c_i \mid c_i \in C, \ell \notin c_i\}$

while C contains unary clause $c_i = \{\ell\}$:

$\theta = \theta \cup \{\ell \leftarrow \texttt{true}\}$

$C = \{c_i \setminus \{-\ell\} \mid c_i \in C, \ell \notin c_i\}$

Pure Literal Elimination

We can eliminate variables that occur only positively/only negatively (pure literals) by setting them to that value; that also removes all clauses containing them.

Unit Propagation (UP) / Binary Constraint Propagation (BCP)

Unary clauses (clauses with only one literal, *units*) force assigning that literal to true. Clauses satisfied by this assignment can be removed. The negated literal is removed from all other clauses. This, in turn, can produce more unit clauses; iterate until fix point.

Classical Backtracking & Propagation: DPLL

DPLL(X, C, θ)

while C contains pure literal ℓ:
  $\theta = \theta \cup \{\ell \leftarrow \texttt{true}\}$
  $C = \{c_i \mid c_i \in C, \ell \notin c_i\}$

while C contains unary clause $c_i = \{\ell\}$:
  $\theta = \theta \cup \{\ell \leftarrow \texttt{true}\}$
  $C = \{c_i \setminus \{-\ell\} \mid c_i \in C, \ell \notin c_i\}$

if θ assigns multiple values to a variable:
  return $\bot$

if empty clause $\emptyset \in C$:
  return $\bot$

if θ assigns all variables:
  return θ

Pure Literal Elimination
We can eliminate variables that occur only positively/only negatively (pure literals) by setting them to that value; that also removes all clauses containing them.

Unit Propagation (UP) / Binary Constraint Propagation (BCP)
Unary clauses (clauses with only one literal, *units*) force assigning that literal to true. Clauses satisfied by this assignment can be removed. The negated literal is removed from all other clauses. This, in turn, can produce more unit clauses; iterate until fix point.

Classical Backtracking & Propagation: DPLL

DPLL(X, C, θ)

while C contains pure literal ℓ:
  $\theta = \theta \cup \{\ell \leftarrow \texttt{true}\}$
  $C = \{c_i \mid c_i \in C, \ell \notin c_i\}$

while C contains unary clause $c_i = \{\ell\}$:
  $\theta = \theta \cup \{\ell \leftarrow \texttt{true}\}$
  $C = \{c_i \setminus \{-\ell\} \mid c_i \in C, \ell \notin c_i\}$

if θ assigns multiple values to a variable:
  return $\bot$

if empty clause $\emptyset \in C$:
  return $\bot$

if θ assigns all variables:
  return θ

select unassigned variable x_i

Pure Literal Elimination
We can eliminate variables that occur only positively/only negatively (pure literals) by setting them to that value; that also removes all clauses containing them.

Unit Propagation (UP) / Binary Constraint Propagation (BCP)
Unary clauses (clauses with only one literal, *units*) force assigning that literal to true. Clauses satisfied by this assignment can be removed. The negated literal is removed from all other clauses. This, in turn, can produce more unit clauses; iterate until fix point.

Unassigned Variable Selection
There are multiple heuristics to select the next variable.

Classical Backtracking & Propagation: DPLL

DPLL(X, C, θ)

while C contains pure literal ℓ:
  $\theta = \theta \cup \{\ell \leftarrow \texttt{true}\}$
  $C = \{c_i \mid c_i \in C, \ell \notin c_i\}$

while C contains unary clause $c_i = \{\ell\}$:
  $\theta = \theta \cup \{\ell \leftarrow \texttt{true}\}$
  $C = \{c_i \setminus \{-\ell\} \mid c_i \in C, \ell \notin c_i\}$

if θ assigns multiple values to a variable:
  return $\bot$

if empty clause $\emptyset \in C$:
  return $\bot$

if θ assigns all variables:
  return θ

select unassigned variable x_i

$\theta' = \theta \cup \{x_i \leftarrow \texttt{true}\}$
$C' = \{c_i \setminus \{-x_i\} \mid c_i \in C, x_i \notin c_i\}$

if $r = \text{DPLL}(X, C', \theta') \neq \bot$:
  return r

Pure Literal Elimination
We can eliminate variables that occur only positively/only negatively (pure literals) by setting them to that value; that also removes all clauses containing them.

Unit Propagation (UP) / Binary Constraint Propagation (BCP)
Unary clauses (clauses with only one literal, *units*) force assigning that literal to true. Clauses satisfied by this assignment can be removed. The negated literal is removed from all other clauses. This, in turn, can produce more unit clauses; iterate until fix point.

Unassigned Variable Selection
There are multiple heuristics to select the next variable.

Classical Backtracking & Propagation: DPLL

DPLL(X, C, θ)

while C contains pure literal ℓ:
  $\theta = \theta \cup \{\ell \leftarrow \texttt{true}\}$
  $C = \{c_i \mid c_i \in C, \ell \notin c_i\}$

while C contains unary clause $c_i = \{\ell\}$:
  $\theta = \theta \cup \{\ell \leftarrow \texttt{true}\}$
  $C = \{c_i \setminus \{-\ell\} \mid c_i \in C, \ell \notin c_i\}$

if θ assigns multiple values to a variable:
  return $\bot$

if empty clause $\emptyset \in C$:
  return $\bot$

if θ assigns all variables:
  return θ

select unassigned variable x_i

$\theta' = \theta \cup \{x_i \leftarrow \texttt{true}\}$
$C' = \{c_i \setminus \{-x_i\} \mid c_i \in C, x_i \notin c_i\}$

if $r = \text{DPLL}(X, C', \theta') \neq \bot$:
  return r

$\theta' = \theta \cup \{x_i \leftarrow \texttt{false}\}$
$C' = \{c_i \setminus \{x_i\} \mid c_i \in C, -x_i \notin c_i\}$

return DPLL(X, C', θ')

Pure Literal Elimination
We can eliminate variables that occur only positively/only negatively (pure literals) by setting them to that value; that also removes all clauses containing them.

Unit Propagation (UP) / Binary Constraint Propagation (BCP)
Unary clauses (clauses with only one literal, *units*) force assigning that literal to true. Clauses satisfied by this assignment can be removed. The negated literal is removed from all other clauses. This, in turn, can produce more unit clauses; iterate until fix point.

Unassigned Variable Selection
There are multiple heuristics to select the next variable.

Classical Backtracking & Propagation: DPLL

DPLL(X, C, θ)

while C contains pure literal ℓ:
  $\theta = \theta \cup \{\ell \leftarrow \texttt{true}\}$
  $C = \{c_i \mid c_i \in C, \ell \notin c_i\}$

while C contains unary clause $c_i = \{\ell\}$:
  $\theta = \theta \cup \{\ell \leftarrow \texttt{true}\}$
  $C = \{c_i \setminus \{-\ell\} \mid c_i \in C, \ell \notin c_i\}$

if θ assigns multiple values to a variable:
  return $\bot$

if empty clause $\emptyset \in C$:
  return $\bot$

if θ assigns all variables:
  return θ

select unassigned variable x_i

$\theta' = \theta \cup \{x_i \leftarrow \texttt{true}\}$
$C' = \{c_i \setminus \{-x_i\} \mid c_i \in C, x_i \notin c_i\}$
if $r = \text{DPLL}(X, C', \theta') \neq \bot$:
  return r
$\theta' = \theta \cup \{x_i \leftarrow \texttt{false}\}$
$C' = \{c_i \setminus \{x_i\} \mid c_i \in C, -x_i \notin c_i\}$
return DPLL(X, C', θ')

Pure Literal Elimination
We can eliminate variables that occur only positively/only negatively (pure literals) by setting them to that value; that also removes all clauses containing them.

Unit Propagation (UP) / Binary Constraint Propagation (BCP)
Unary clauses (clauses with only one literal, *units*) force assigning that literal to true. Clauses satisfied by this assignment can be removed. The negated literal is removed from all other clauses. This, in turn, can produce more unit clauses; iterate until fix point.

Unassigned Variable Selection
There are multiple heuristics to select the next variable.

As always with pseudo-code:

- Naïve implementation inefficient
- Efficient implementation non-trivial
- Need: good implementation of UP, avoid copying clauses
- Ideally, no actual recursion

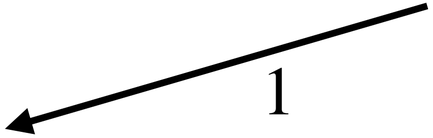
DPLL: Example

DPLL: Example

$$C = \{12\overline{3}, 23\overline{4}, 134, \overline{1}24, \overline{1}\overline{2}3, \overline{2}\overline{3}4, \overline{1}\overline{3}\overline{4}, 1\overline{2}\overline{4}\}$$

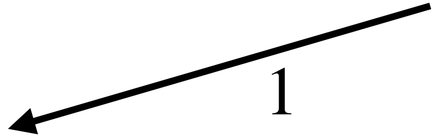
DPLL: Example

$$C = \{12\overline{3}, 23\overline{4}, 134, \overline{1}24, \overline{1}\overline{2}3, \overline{2}\overline{3}4, \overline{1}\overline{3}\overline{4}, 1\overline{2}\overline{4}\}$$



DPLL: Example

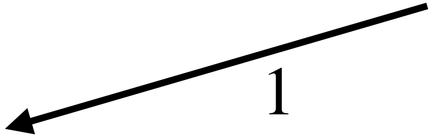
$$C = \{12\overline{3}, 23\overline{4}, 134, \overline{1}24, \overline{1}\,\overline{2}3, \overline{2}\,\overline{3}4, \overline{1}\,\overline{3}\,\overline{4}, 1\overline{2}\,\overline{4}\}$$



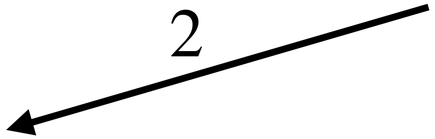
$$\{23\overline{4}, 24, \overline{2}3, \overline{2}\,\overline{3}4, \overline{3}\,\overline{4}\}$$

DPLL: Example

$$C = \{12\overline{3}, 23\overline{4}, 134, \overline{1}24, \overline{1}\overline{2}3, \overline{2}\overline{3}4, \overline{1}\overline{3}\overline{4}, 1\overline{2}\overline{4}\}$$



$$\{23\overline{4}, 24, \overline{2}3, \overline{2}\overline{3}4, \overline{3}\overline{4}\}$$



DPLL: Example

$$C = \{12\overline{3}, 23\overline{4}, 134, \overline{1}24, \overline{1}\overline{2}3, \overline{2}\overline{3}4, \overline{1}\overline{3}\overline{4}, 1\overline{2}\overline{4}\}$$

1

$$\{23\overline{4}, 24, \overline{2}3, \overline{2}\overline{3}4, \overline{3}\overline{4}\}$$

2

$$\{3, \overline{3}4, \overline{3}\overline{4}\}$$

DPLL: Example

$$C = \{12\overline{3}, 23\overline{4}, 134, \overline{1}24, \overline{1}\overline{2}3, \overline{2}\overline{3}4, \overline{1}\overline{3}\overline{4}, 1\overline{2}\overline{4}\}$$

1

$$\{23\overline{4}, 24, \overline{2}3, \overline{2}\overline{3}4, \overline{3}\overline{4}\}$$

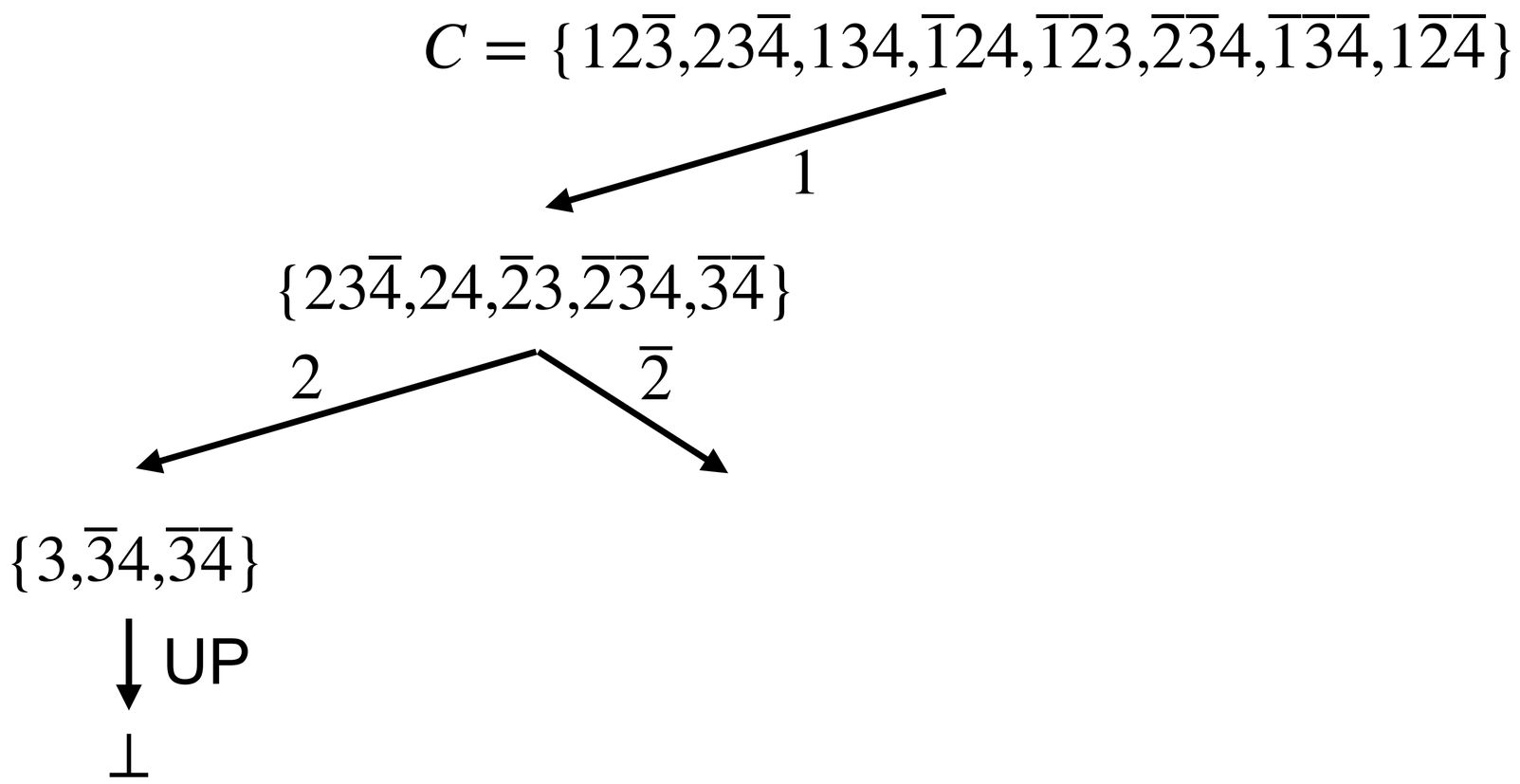
2

$$\{3, \overline{3}4, \overline{3}\overline{4}\}$$

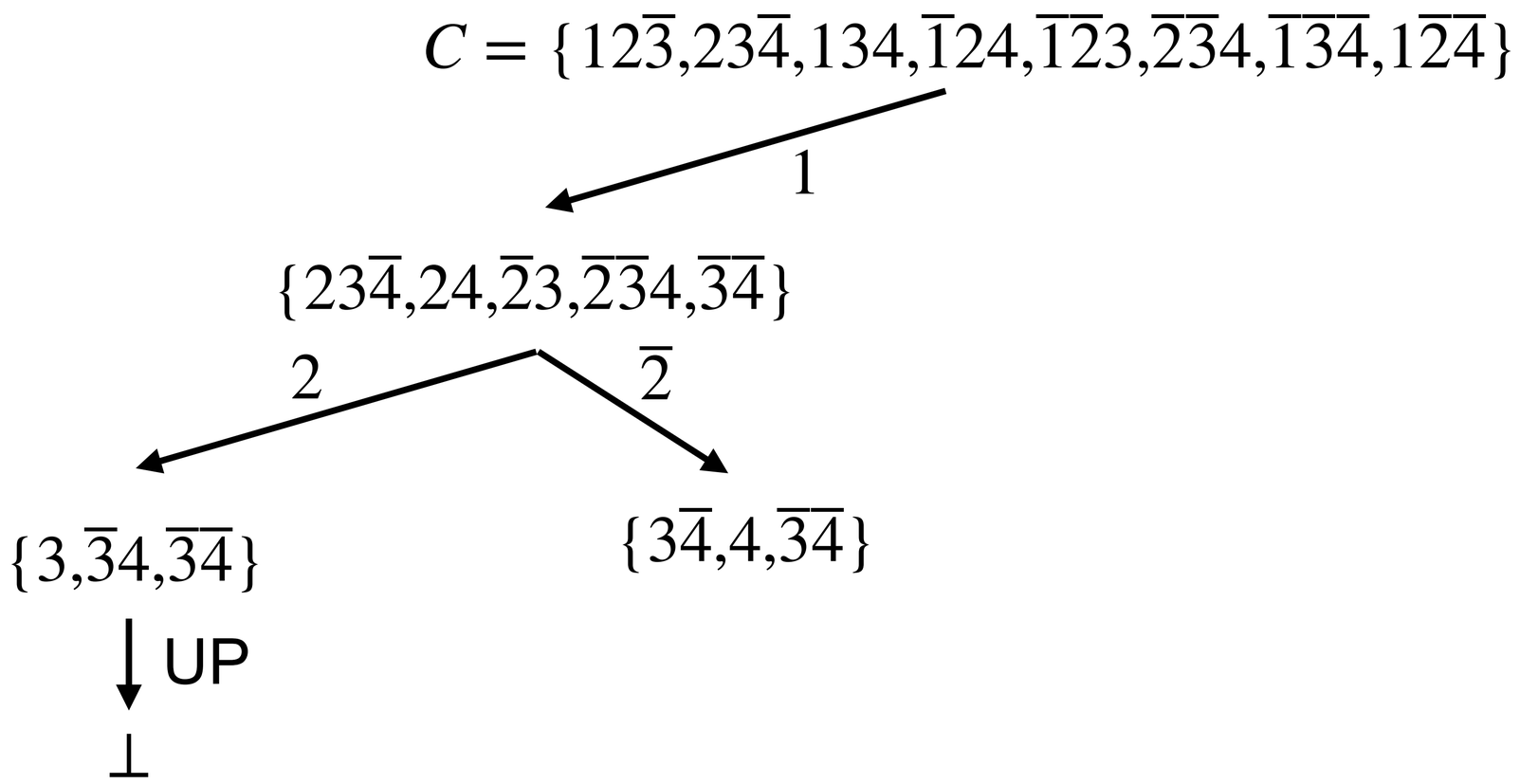
UP

$$\bot$$

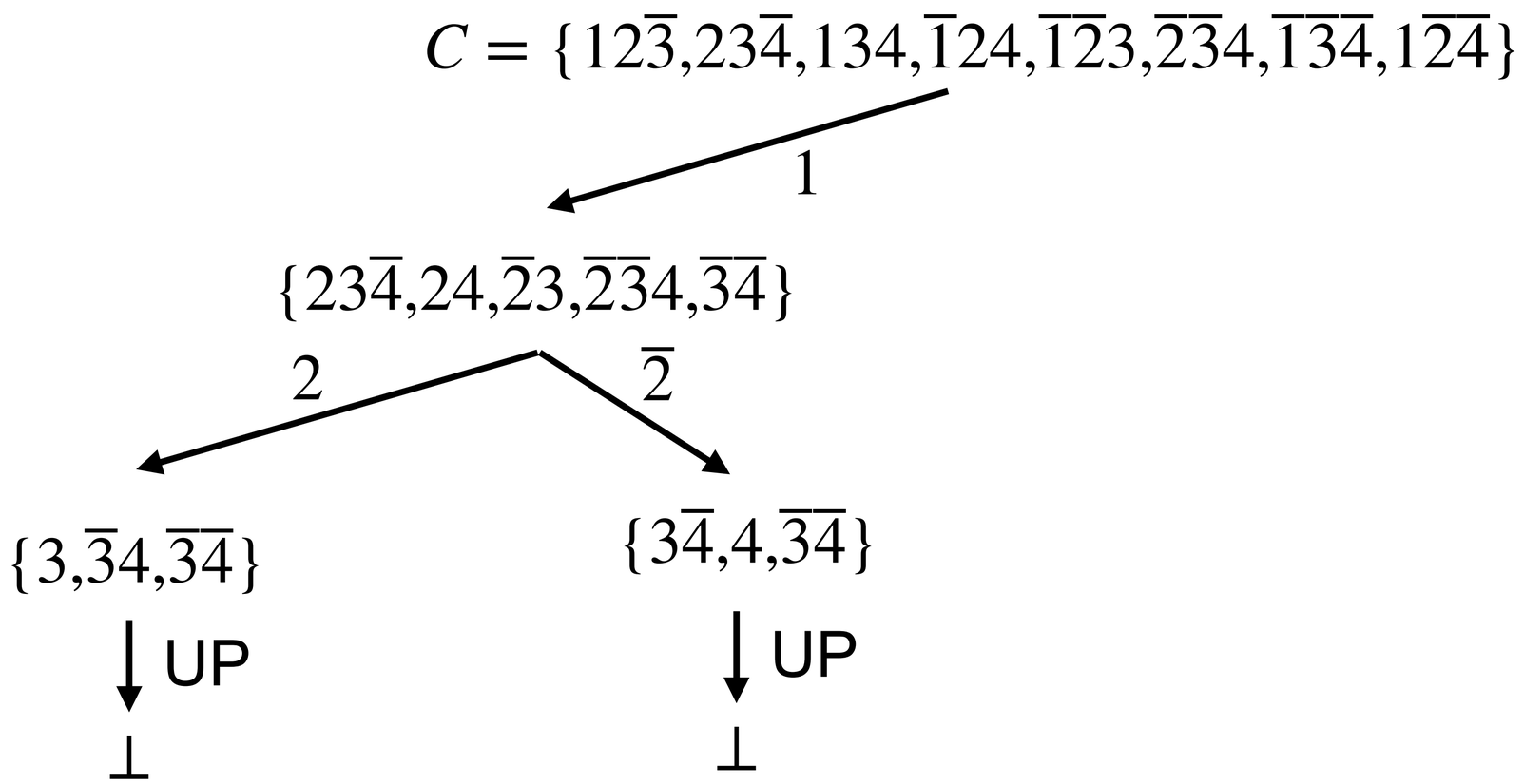
DPLL: Example



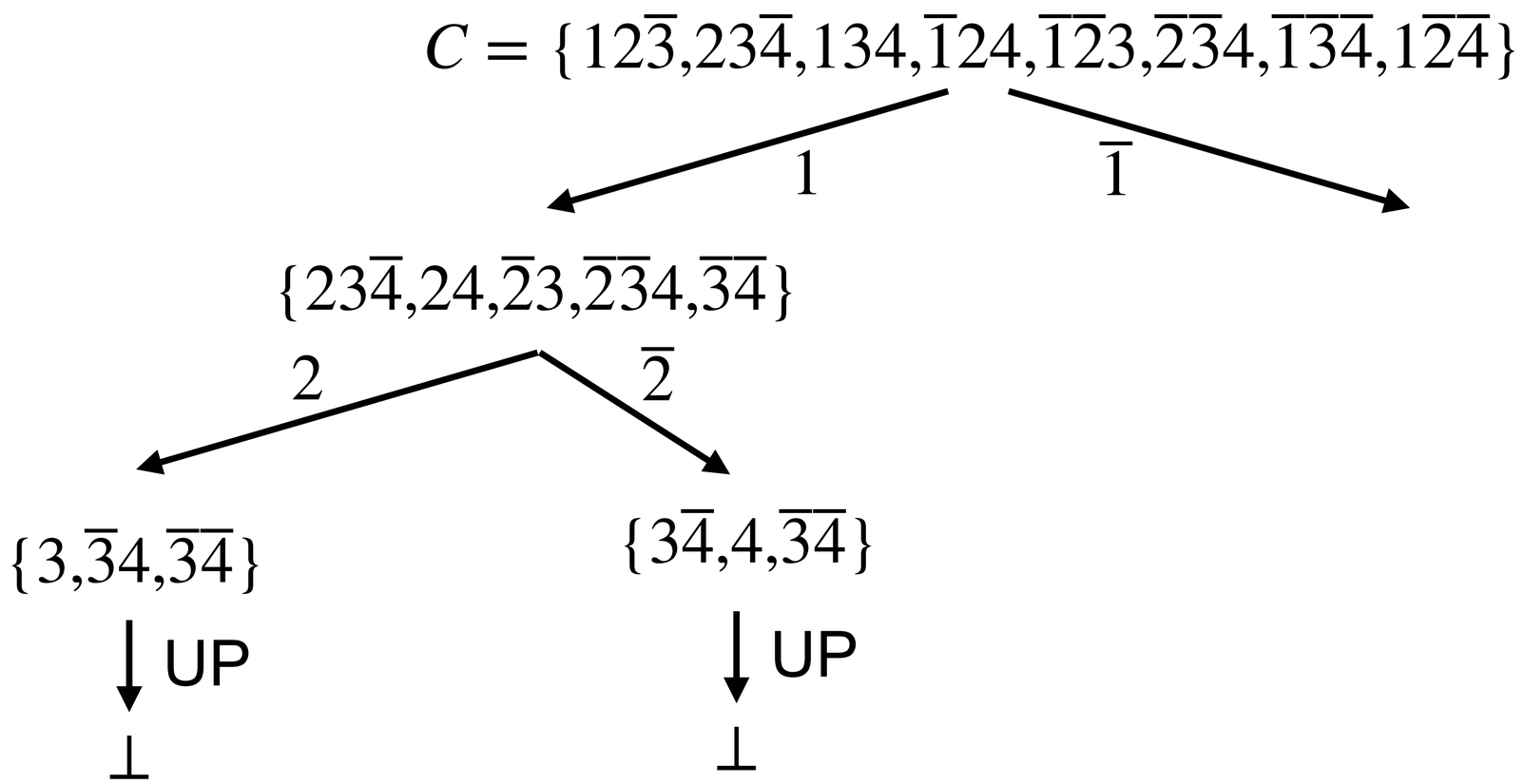
DPLL: Example



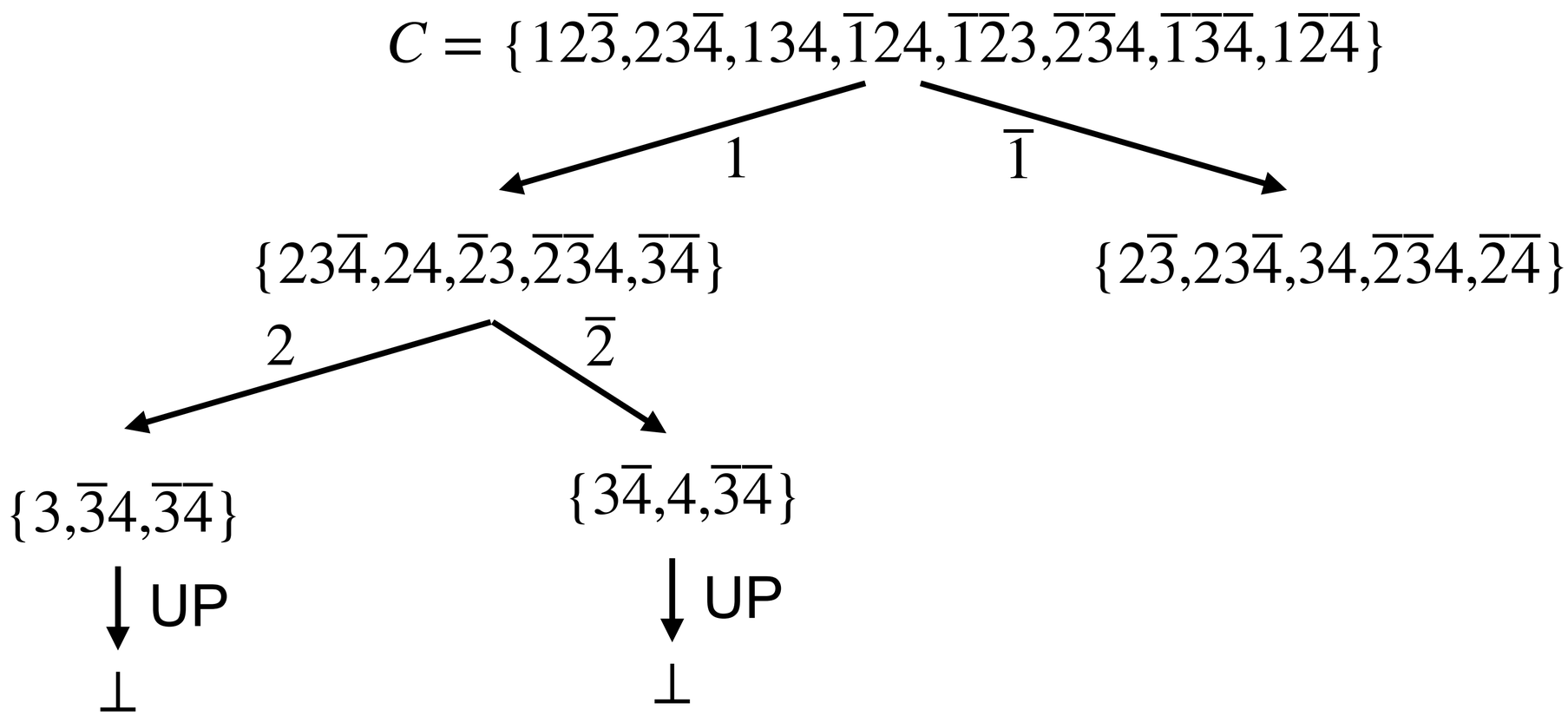
DPLL: Example



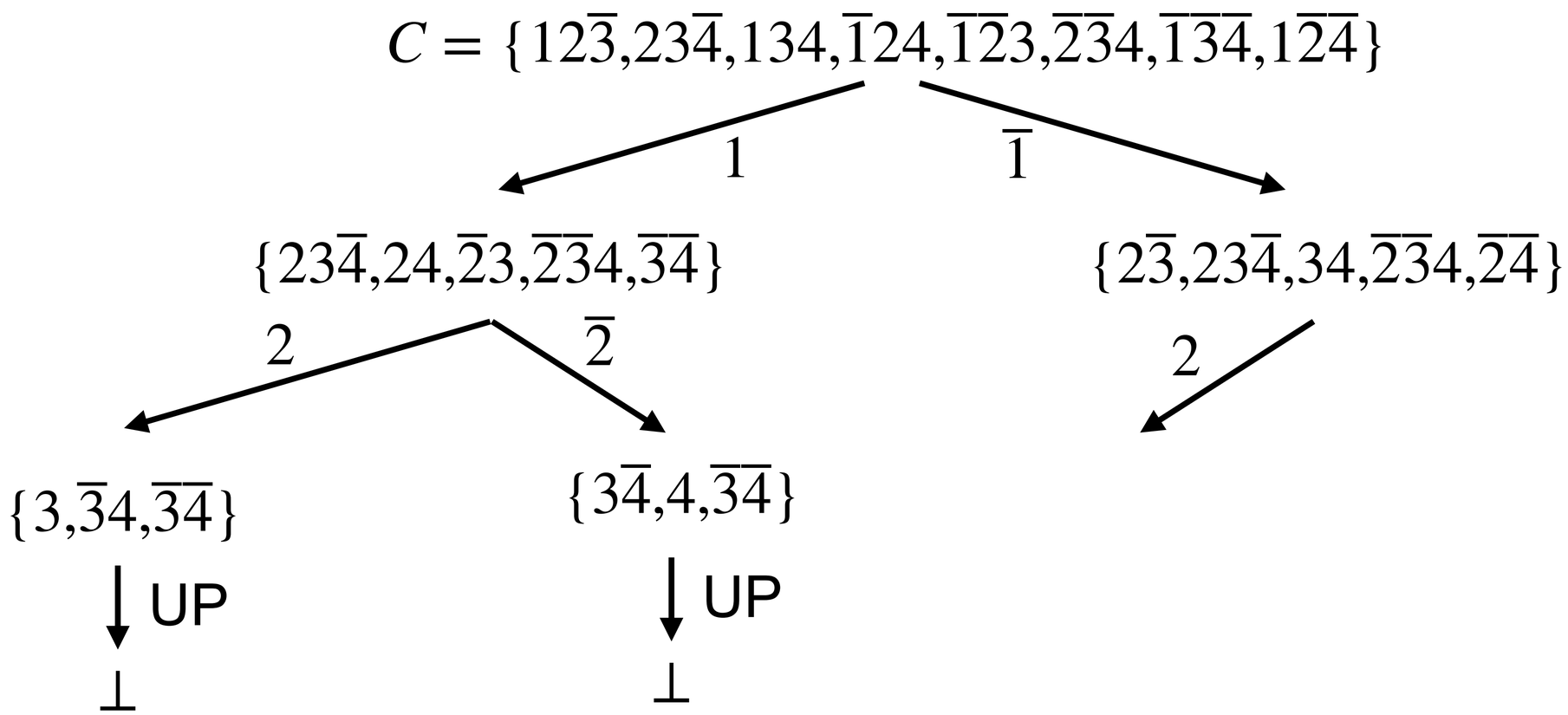
DPLL: Example



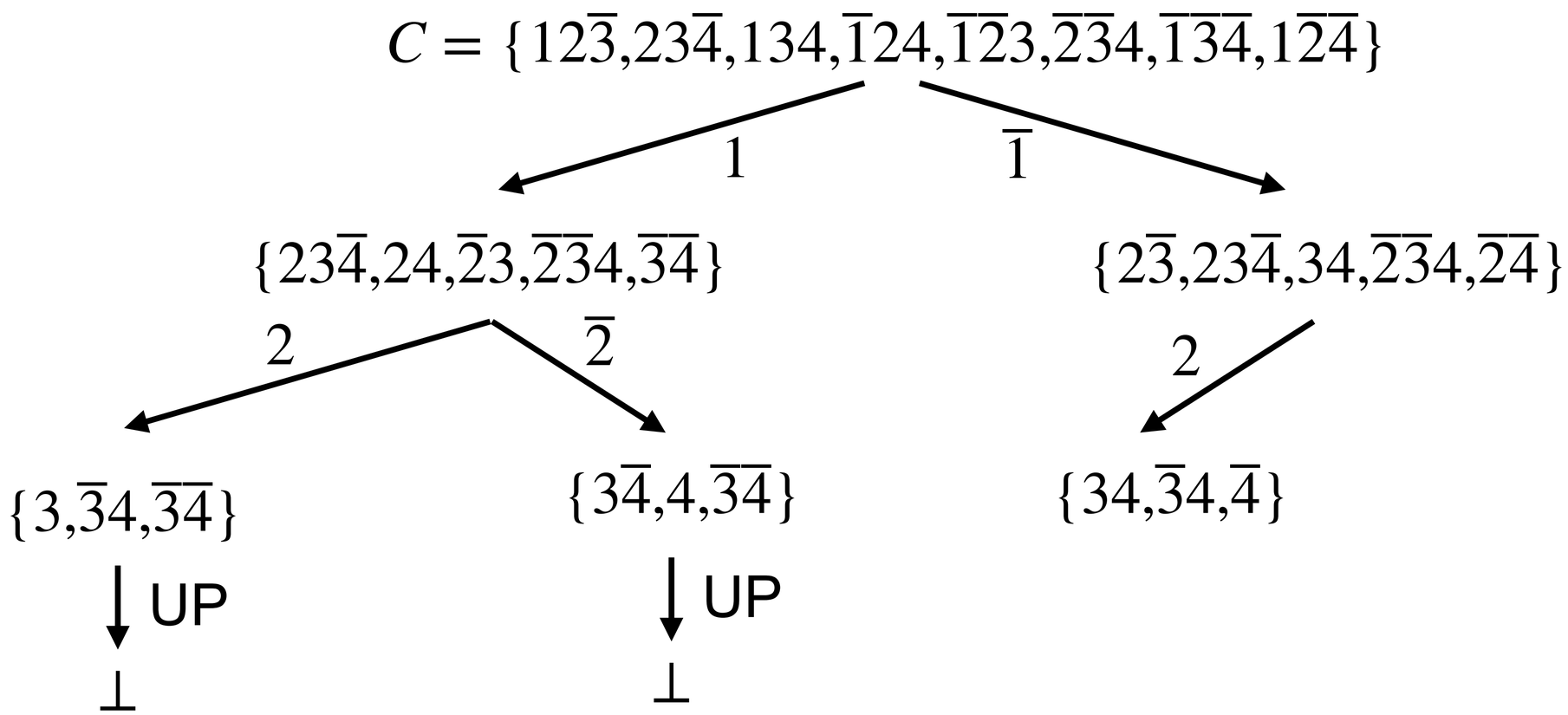
DPLL: Example



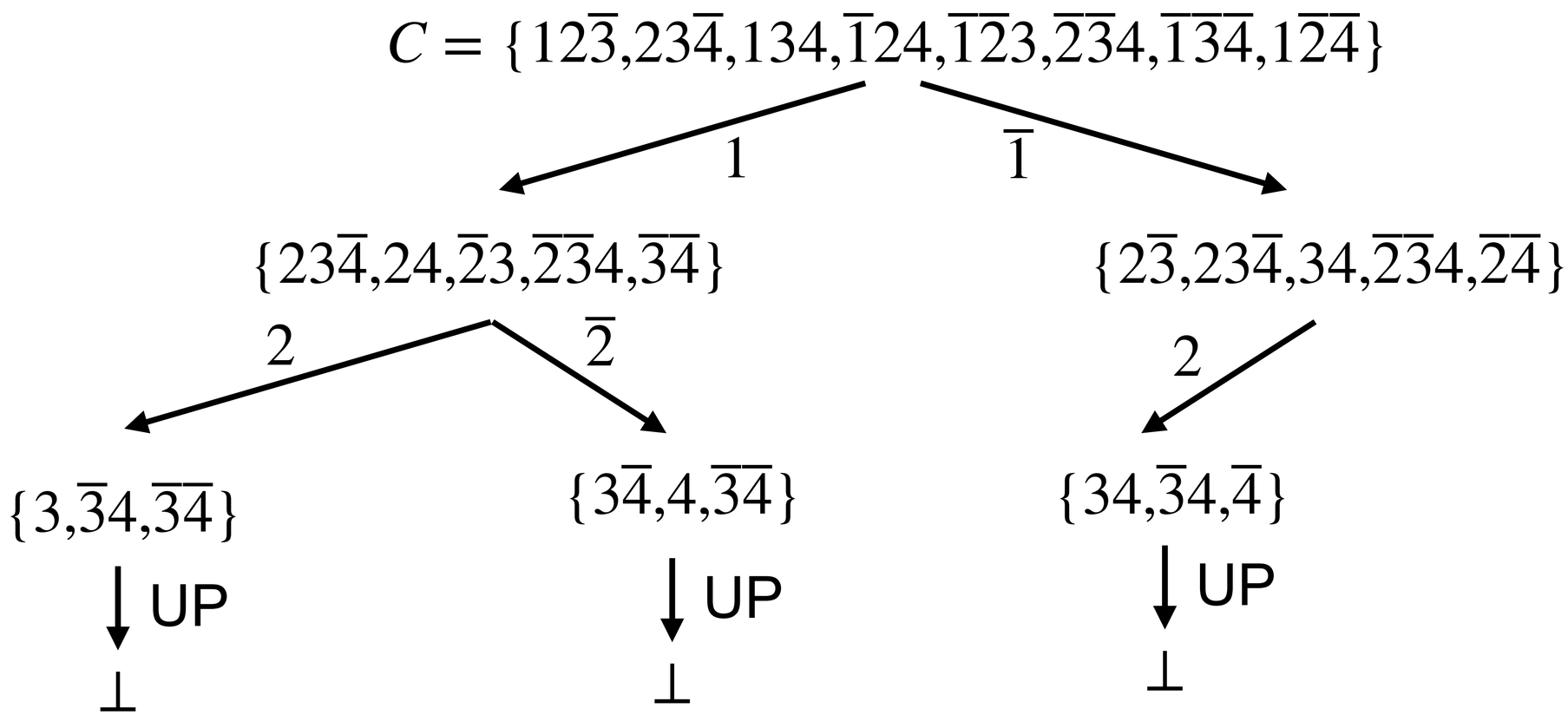
DPLL: Example



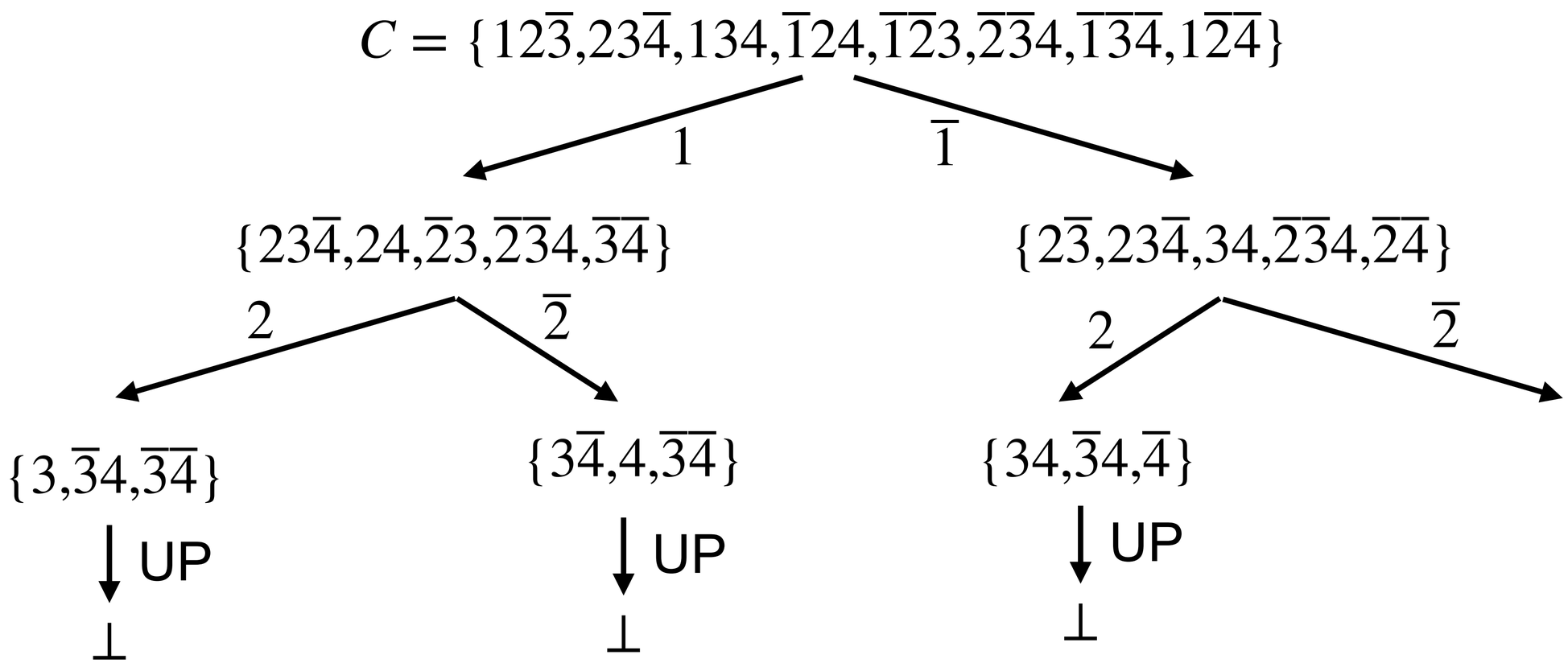
DPLL: Example



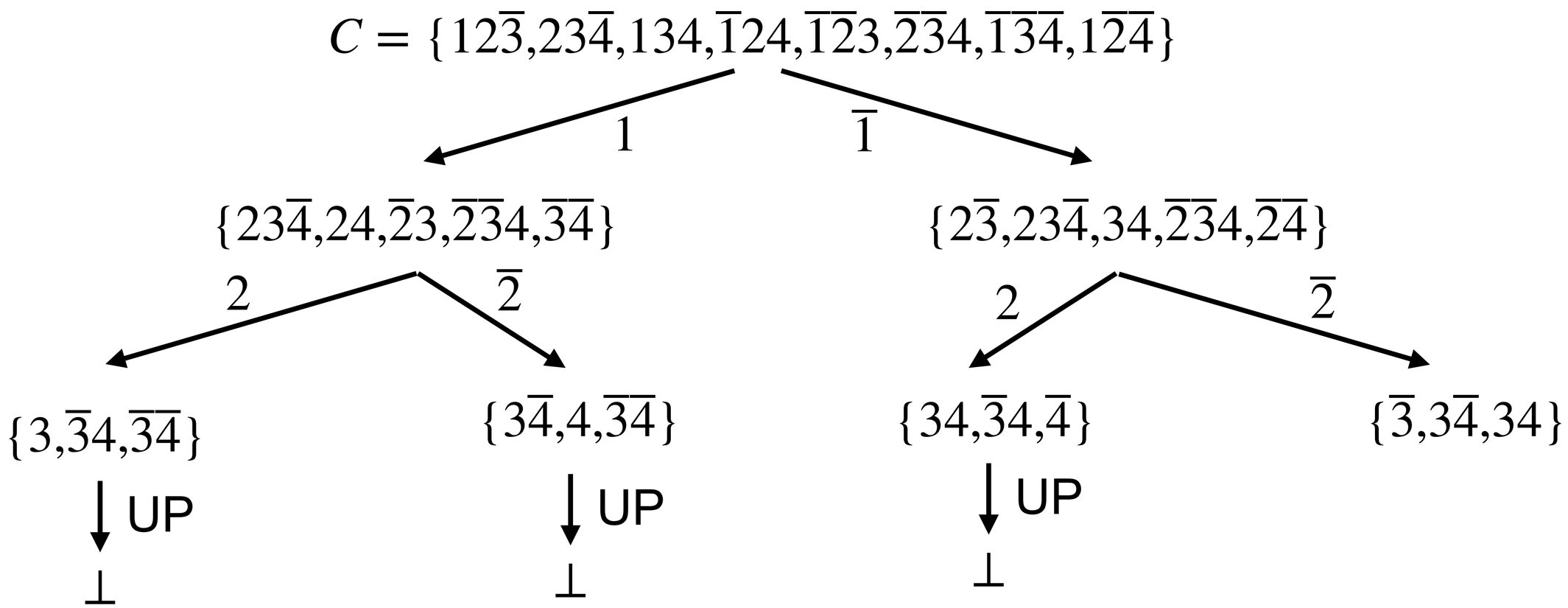
DPLL: Example



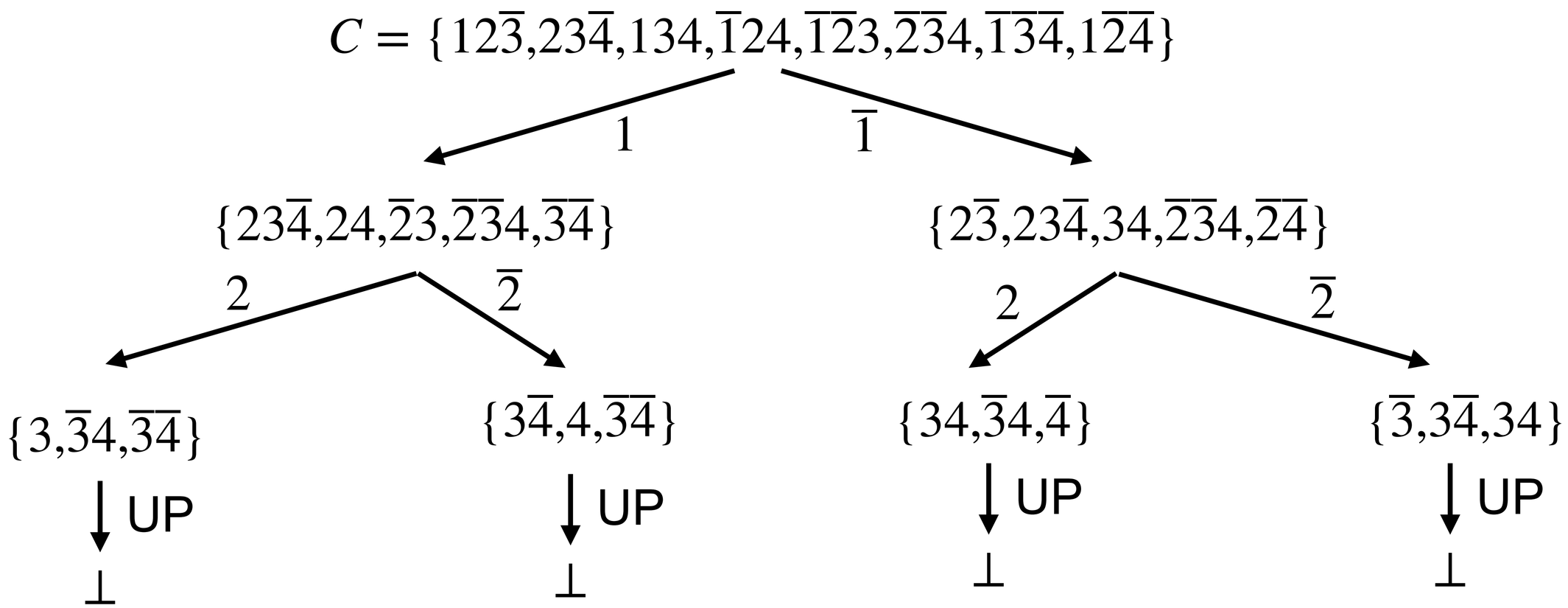
DPLL: Example



DPLL: Example



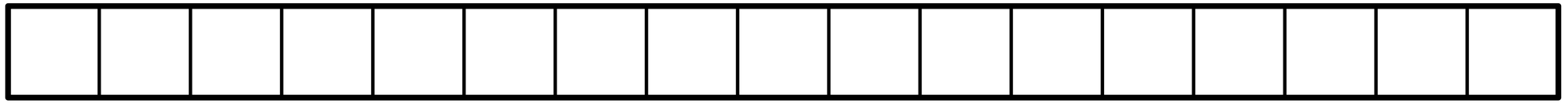
DPLL: Example



Implementation: Trail Data Structure

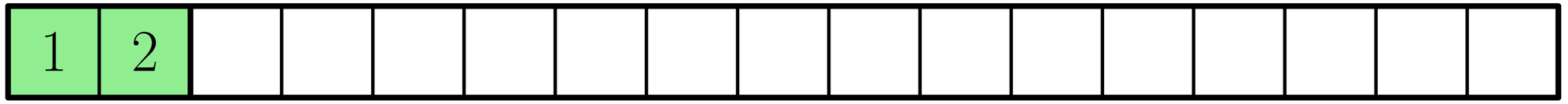
Implementation: Trail Data Structure

$$C = \{\overline{1}2, 23\overline{4}, 134, \overline{1}24, \overline{12}3, \overline{23}4, \overline{134}, 1\overline{24}\}$$



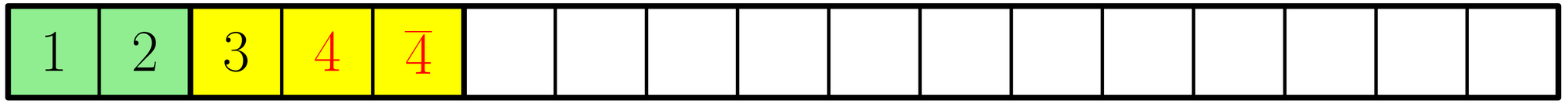
Implementation: Trail Data Structure

$$C = \{\overline{1}2, 23\overline{4}, 134, \overline{1}24, \overline{12}3, \overline{23}4, \overline{134}, 1\overline{24}\}$$



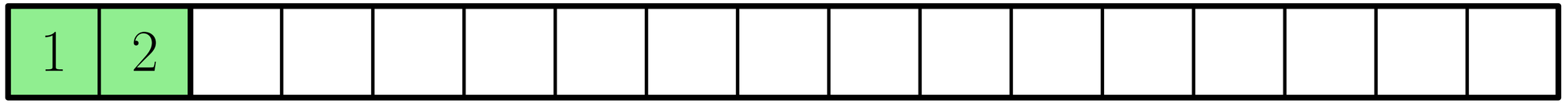
Implementation: Trail Data Structure

$$C = \{\overline{1}2, 23\overline{4}, 134, \overline{1}24, \overline{12}34, \overline{23}4, \overline{134}, 1\overline{24}\}$$



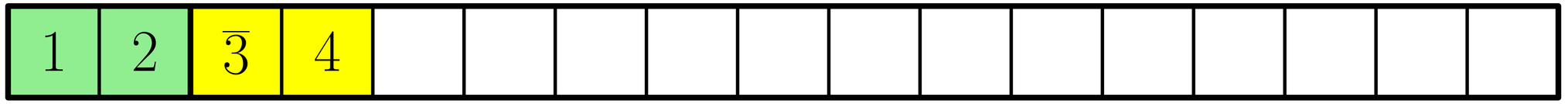
Implementation: Trail Data Structure

$$C = \{\overline{1}2, 23\overline{4}, 134, \overline{1}24, \overline{12}34, \overline{23}4, \overline{134}, 1\overline{24}\}$$



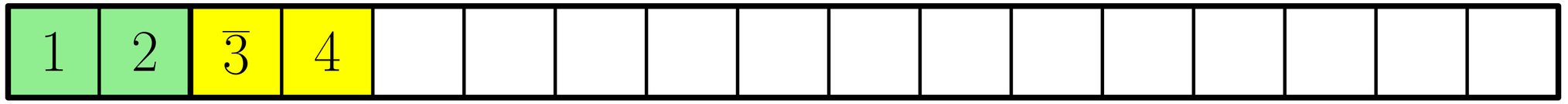
Implementation: Trail Data Structure

$$C = \{\overline{1}2, 23\overline{4}, 134, \overline{1}24, \overline{12}34, \overline{23}4, \overline{134}, 1\overline{24}\}$$



Implementation: Trail Data Structure

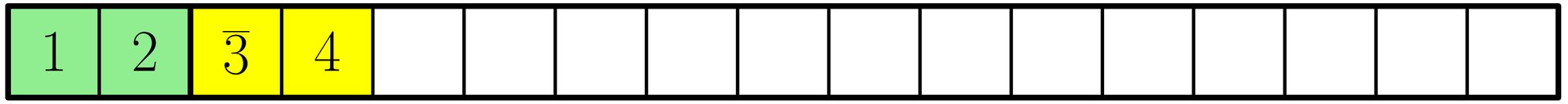
$$C = \{\overline{1}2, 23\overline{4}, 134, \overline{1}24, \overline{12}34, \overline{23}4, \overline{134}, 1\overline{24}\}$$



Trail: Stack of true literals

Implementation: Trail Data Structure

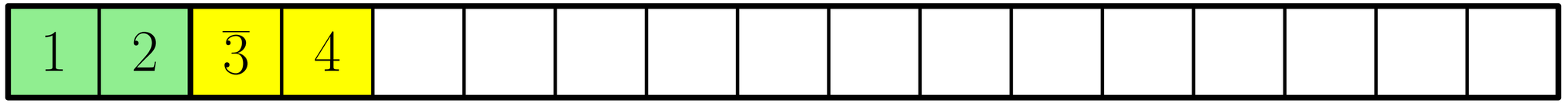
$$C = \{\overline{1}2, 23\overline{4}, 134, \overline{1}24, \overline{1}234, \overline{23}4, \overline{134}, 1\overline{24}\}$$



Trail: Stack of true literals
Subdivided into *decision levels* by *decisions* (assignments not based on propagation)

Implementation: Trail Data Structure

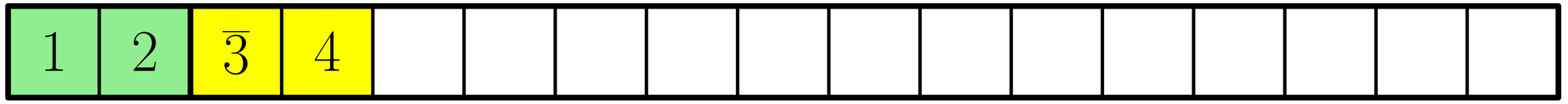
$$C = \{\overline{1}2, 23\overline{4}, 134, \overline{1}24, \overline{1}23\overline{4}, \overline{23}4, \overline{13}\overline{4}, 1\overline{24}\}$$



Trail: Stack of true literals
Subdivided into *decision levels* by *decisions* (assignments not based on propagation)
Level 0: always true literals (implied by unary clauses)

Implementation: Trail Data Structure

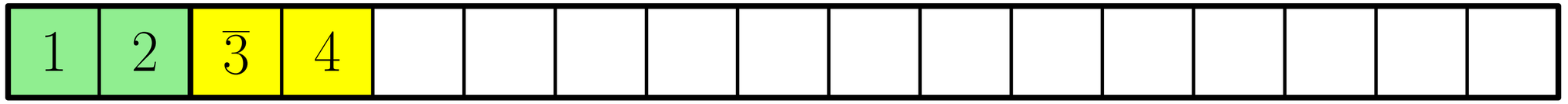
$$C = \{\overline{1}2, 23\overline{4}, 134, \overline{1}24, \overline{1}234, \overline{23}4, \overline{134}, 1\overline{24}\}$$



Trail: Stack of true literals
Subdivided into *decision levels* by *decisions* (assignments not based on propagation)
Level 0: always true literals (implied by unary clauses)
Every other level starts on a decision

Implementation: Trail Data Structure

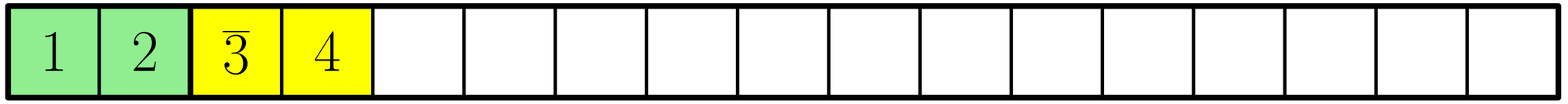
$$C = \{\overline{1}2, 23\overline{4}, 134, \overline{1}24, \overline{1}2\overline{3}4, \overline{23}4, \overline{13}\overline{4}, 1\overline{24}\}$$



Trail: Stack of true literals
Subdivided into *decision levels* by *decisions* (assignments not based on propagation)
Level 0: always true literals (implied by unary clauses)
Every other level starts on a decision
Variable states: false, true, open

Implementation: Trail Data Structure

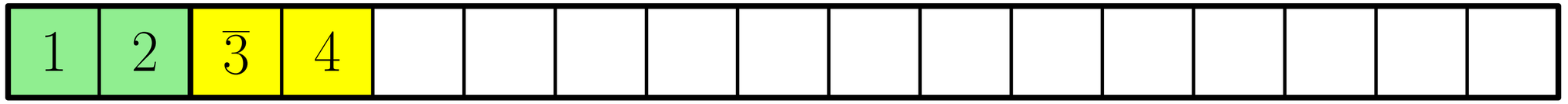
$$C = \{\overline{1}2, 23\overline{4}, 134, \overline{1}24, \overline{1}23\overline{4}, \overline{23}4, \overline{13}\overline{4}, 1\overline{24}\}$$



Trail: Stack of true literals
Subdivided into *decision levels* by *decisions* (assignments not based on propagation)
Level 0: always true literals (implied by unary clauses)
Every other level starts on a decision
Variable states: false, true, open
Backtracking: by pushing decision + UP (down) or popping a decision level (up)

Implementation: Trail Data Structure

$$C = \{\overline{1}2, 23\overline{4}, 134, \overline{1}24, \overline{1}23\overline{4}, \overline{23}4, \overline{13}\overline{4}, 1\overline{24}\}$$

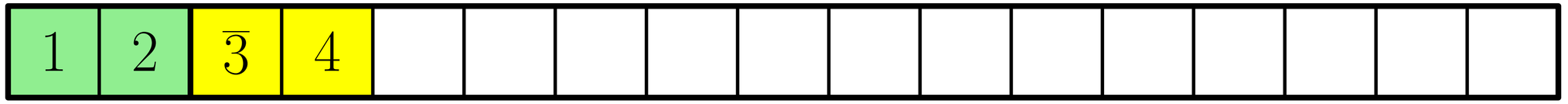


Trail: Stack of true literals
Subdivided into *decision levels* by *decisions* (assignments not based on propagation)
Level 0: always true literals (implied by unary clauses)
Every other level starts on a decision
Variable states: false, true, open
Backtracking: by pushing decision + UP (down) or popping a decision level (up)

No pure-literal elimination: only in preprocessing

Implementation: Trail Data Structure

$$C = \{\overline{1}2, 23\overline{4}, 134, \overline{1}24, \overline{1}23\overline{4}, \overline{23}4, \overline{134}, 1\overline{24}\}$$



Trail: Stack of true literals
Subdivided into *decision levels* by *decisions* (assignments not based on propagation)
Level 0: always true literals (implied by unary clauses)
Every other level starts on a decision
Variable states: false, true, open
Backtracking: by pushing decision + UP (down) or popping a decision level (up)

No pure-literal elimination: only in preprocessing
Key data idea for unit propagation: 2 watched literals

Implementation: Two Watched Literals

Implementation: Two Watched Literals

Idea: We must notice when a clause has only one open and no true literal.

Implementation: Two Watched Literals

Idea: We must notice when a clause has only one open and no true literal.

- Such a clause is effectively unit and forces an assignment of the open literal

Implementation: Two Watched Literals

Idea: We must notice when a clause has only one open and no true literal.

- Such a clause is effectively unit and forces an assignment of the open literal
- We “watch” two literals in each clause (watched literals)

Implementation: Two Watched Literals

Idea: We must notice when a clause has only one open and no true literal.

- Such a clause is effectively unit and forces an assignment of the open literal
- We “watch” two literals in each clause (watched literals)
- Invariant: always watch two non-false literals (true or open)

Implementation: Two Watched Literals

Idea: We must notice when a clause has only one open and no true literal.

- Such a clause is effectively unit and forces an assignment of the open literal
- We “watch” two literals in each clause (watched literals)
- Invariant: always watch two non-false literals (true or open)
  - For satisfied clauses: true literal + any other literal (can be false)

Implementation: Two Watched Literals

Idea: We must notice when a clause has only one open and no true literal.

- Such a clause is effectively unit and forces an assignment of the open literal
- We “watch” two literals in each clause (watched literals)
- Invariant: always watch two non-false literals (true or open)
  - For satisfied clauses: true literal + any other literal (can be false)
- On $\ell \leftarrow$ true: search *non-satisfied* clauses c_i, in which $\overline{\ell}$ is watched, for replacement

Implementation: Two Watched Literals

Idea: We must notice when a clause has only one open and no true literal.

- Such a clause is effectively unit and forces an assignment of the open literal
- We “watch” two literals in each clause (watched literals)
- Invariant: always watch two non-false literals (true or open)
  - For satisfied clauses: true literal + any other literal (can be false)
- On $\ell \leftarrow$ true: search *non-satisfied* clauses c_i, in which $\overline{\ell}$ is watched, for replacement
  - If there is no replacement: unary clause (new assignment) or conflict

Implementation: Two Watched Literals

Idea: We must notice when a clause has only one open and no true literal.

- Such a clause is effectively unit and forces an assignment of the open literal
- We “watch” two literals in each clause (watched literals)
- Invariant: always watch two non-false literals (true or open)
  - For satisfied clauses: true literal + any other literal (can be false)
- On $\ell \leftarrow$ true: search *non-satisfied* clauses c_i, in which $\overline{\ell}$ is watched, for replacement
  - If there is no replacement: unary clause (new assignment) or conflict
- **Question:** What has to happen on backtracking (popping decision level)?

Implementation: Two Watched Literals

Idea: We must notice when a clause has only one open and no true literal.

- Such a clause is effectively unit and forces an assignment of the open literal
- We “watch” two literals in each clause (watched literals)
- Invariant: always watch two non-false literals (true or open)
  - For satisfied clauses: true literal + any other literal (can be false)
- On $\ell \leftarrow$ true: search *non-satisfied* clauses c_i, in which $\overline{\ell}$ is watched, for replacement
  - If there is no replacement: unary clause (new assignment) or conflict
- **Question:** What has to happen on backtracking (popping decision level)?
- **Have to:** remove decision level from trail; what needs to happen to watched literals?

Implementation: Two Watched Literals

Implementation: Two Watched Literals

Answer: nothing!

Implementation: Two Watched Literals

Answer: <u>nothing</u>!

- Amazing, doing nothing is really cheap! But why?

Implementation: Two Watched Literals

Answer: <u>nothing</u>!

- Amazing, doing nothing is really cheap! But why?
  - Backtracking cannot make non-false literals false!

Implementation: Two Watched Literals

Answer: nothing!

- Amazing, doing nothing is really cheap! But why?
  - Backtracking cannot make non-false literals false!
  - Invariant remains satisfied for clauses with two non-false literals

Implementation: Two Watched Literals

Answer: nothing!

- Amazing, doing nothing is really cheap! But why?
  - Backtracking cannot make non-false literals false!
  - Invariant remains satisfied for clauses with two non-false literals
  - Clauses with no two non-false literals are satisfied or caused a conflict

Implementation: Two Watched Literals

Answer: nothing!

- Amazing, doing nothing is really cheap! But why?
  - Backtracking cannot make non-false literals false!
  - Invariant remains satisfied for clauses with two non-false literals
  - Clauses with no two non-false literals are satisfied or caused a conflict
  - **Satisfied clauses:** can become unsatisfied (true → open-transition)

Implementation: Two Watched Literals

Answer: nothing!

- Amazing, doing nothing is really cheap! But why?
  - Backtracking cannot make non-false literals false!
  - Invariant remains satisfied for clauses with two non-false literals
  - Clauses with no two non-false literals are satisfied or caused a conflict
  - **Satisfied clauses:** can become unsatisfied (true → open-transition)
    - Can a false literal be watched after backtracking?

Implementation: Two Watched Literals

Answer: nothing!

- Amazing, doing nothing is really cheap! But why?
  - Backtracking cannot make non-false literals false!
  - Invariant remains satisfied for clauses with two non-false literals
  - Clauses with no two non-false literals are satisfied or caused a conflict
  - **Satisfied clauses:** can become unsatisfied (true → open-transition)
    - Can a false literal be watched after backtracking?
    - No! Literal only became false after the clause became satisfied.

Implementation: Two Watched Literals

Answer: nothing!

- Amazing, doing nothing is really cheap! But why?
  - Backtracking cannot make non-false literals false!
  - Invariant remains satisfied for clauses with two non-false literals
  - Clauses with no two non-false literals are satisfied or caused a conflict
  - **Satisfied clauses:** can become unsatisfied (true → open-transition)
    - Can a false literal be watched after backtracking?
    - No! Literal only became false after the clause became satisfied.
    - If the clause becomes unsatisfied, the literal also becomes open.

Implementation: Two Watched Literals

Answer: nothing!

- Amazing, doing nothing is really cheap! But why?
  - Backtracking cannot make non-false literals false!
  - Invariant remains satisfied for clauses with two non-false literals
  - Clauses with no two non-false literals are satisfied or caused a conflict
  - **Satisfied clauses:** can become unsatisfied (true → open-transition)
    - Can a false literal be watched after backtracking?
    - No! Literal only became false after the clause became satisfied.
    - If the clause becomes unsatisfied, the literal also becomes open.
  - **Conflict clause:** watching two false literals

Implementation: Two Watched Literals

Answer: nothing!

- Amazing, doing nothing is really cheap! But why?
  - Backtracking cannot make non-false literals false!
  - Invariant remains satisfied for clauses with two non-false literals
  - Clauses with no two non-false literals are satisfied or caused a conflict
  - **Satisfied clauses:** can become unsatisfied (true → open-transition)
    - Can a false literal be watched after backtracking?
    - No! Literal only became false after the clause became satisfied.
    - If the clause becomes unsatisfied, the literal also becomes open.
  - **Conflict clause:** watching two false literals
    - These both became false in the last round of propagation/decision level

Implementation: Two Watched Literals

Answer: nothing!

- Amazing, doing nothing is really cheap! But why?
  - Backtracking cannot make non-false literals false!
  - Invariant remains satisfied for clauses with two non-false literals
  - Clauses with no two non-false literals are satisfied or caused a conflict
  - **Satisfied clauses:** can become unsatisfied (true → open-transition)
    - Can a false literal be watched after backtracking?
    - No! Literal only became false after the clause became satisfied.
    - If the clause becomes unsatisfied, the literal also becomes open.
  - **Conflict clause:** watching two false literals
    - These both became false in the last round of propagation/decision level
    - We remove at least the last decision, re-opening them both.

Implementation: Two Watched Literals

Answer: nothing!

- Amazing, doing nothing is really cheap! But why?
  - Backtracking cannot make non-false literals false!
  - Invariant remains satisfied for clauses with two non-false literals
  - Clauses with no two non-false literals are satisfied or caused a conflict
  - **Satisfied clauses:** can become unsatisfied (true → open-transition)
    - Can a false literal be watched after backtracking?
    - No! Literal only became false after the clause became satisfied.
    - If the clause becomes unsatisfied, the literal also becomes open.
  - **Conflict clause:** watching two false literals
    - These both became false in the last round of propagation/decision level
    - We remove at least the last decision, re-opening them both.
  - Invariant remains satisfied after backtracking!

DPLL: Problems

DPLL: Problems

We do not learn from our mistakes!

DPLL: Problems

We do not learn from our mistakes!

- Small parts of our formula φ can already be unsatisfiable

DPLL: Problems

We do not learn from our mistakes!

- Small parts of our formula φ can already be unsatisfiable
- Or they may enforce certain assignments

DPLL: Problems

We do not learn from our mistakes!

- Small parts of our formula φ can already be unsatisfiable
- Or they may enforce certain assignments
- Too complex to see for unit propagation

DPLL: Problems

We do not learn from our mistakes!

- Small parts of our formula φ can already be unsatisfiable
- Or they may enforce certain assignments
- Too complex to see for unit propagation
- If these parts are not at the top of the ‘tree’:

DPLL: Problems

We do not learn from our mistakes!

- Small parts of our formula φ can already be unsatisfiable
- Or they may enforce certain assignments
- Too complex to see for unit propagation
- If these parts are not at the top of the ‘tree’:
  - We run into the same conflicts again and again

DPLL: Problems

We do not learn from our mistakes!

- Small parts of our formula φ can already be unsatisfiable
- Or they may enforce certain assignments
- Too complex to see for unit propagation
- If these parts are not at the top of the ‘tree’:
  - We run into the same conflicts again and again
- One can prove that the runtime can become very large on instances

DPLL: Problems

We do not learn from our mistakes!

- Small parts of our formula φ can already be unsatisfiable
- Or they may enforce certain assignments
- Too complex to see for unit propagation
- If these parts are not at the top of the ‘tree’:
  - We run into the same conflicts again and again
- One can prove that the runtime can become very large on instances
- These proofs do not rely on any assumptions on the branching decisions!

How can we learn?

How can we learn?

Idea: learning = generating new clauses. **Tool for that: *resolution*!**

How can we learn?

Idea: learning = generating new clauses. **Tool for that: *resolution*!**

- Clauses $C_i = A \vee x_k, C_j = B \vee \overline{x_k}$: $C_i \diamond C_j := A \vee B$

How can we learn?

Idea: learning = generating new clauses. **Tool for that: *resolution*!**

- Clauses $C_i = A \vee x_k, C_j = B \vee \overline{x_k}$: $C_i \diamond C_j := A \vee B$
- φ satisfied $\Leftrightarrow \varphi \cup \{C_i \diamond C_j\}$ satisfied

How can we learn?

Idea: learning = generating new clauses. **Tool for that: *resolution*!**

- Clauses $C_i = A \vee x_k, C_j = B \vee \overline{x_k}$: $C_i \diamond C_j := A \vee B$
- φ satisfied $\Leftrightarrow \varphi \cup \{C_i \diamond C_j\}$ satisfied

Unsatisfiability proof (refutation) by resolution:

How can we learn?

Idea: learning = generating new clauses. **Tool for that: *resolution*!**

- Clauses $C_i = A \vee x_k, C_j = B \vee \overline{x_k}$: $C_i \diamond C_j := A \vee B$
- φ satisfied $\Leftrightarrow \varphi \cup \{C_i \diamond C_j\}$ satisfied

Unsatisfiability proof (refutation) by resolution:

- DAG (directed acyclic graph): original clauses as sources (“axioms”)

How can we learn?

Idea: learning = generating new clauses. **Tool for that: *resolution*!**

- Clauses $C_i = A \vee x_k$, $C_j = B \vee \overline{x_k}$: $C_i \diamond C_j := A \vee B$
- φ satisfied $\Leftrightarrow \varphi \cup \{C_i \diamond C_j\}$ satisfied

Unsatisfiability proof (refutation) by resolution:

- DAG (directed acyclic graph): original clauses as sources (“axioms”)
- Other nodes C have exactly two predecessors C_i, C_j with $C = C_i \diamond C_j$

How can we learn?

Idea: learning = generating new clauses. **Tool for that: *resolution*!**

- Clauses $C_i = A \vee x_k$, $C_j = B \vee \overline{x_k}$: $C_i \diamond C_j := A \vee B$
- φ satisfied $\Leftrightarrow$ $\varphi \cup \{C_i \diamond C_j\}$ satisfied

Unsatisfiability proof (refutation) by resolution:

- DAG (directed acyclic graph): original clauses as sources (“axioms”)
- Other nodes C have exactly two predecessors C_i, C_j with $C = C_i \diamond C_j$
- Graph has the empty clause ε (aka $\square$) as sink

How can we learn?

Idea: learning = generating new clauses. **Tool for that: *resolution*!**

- Clauses $C_i = A \vee x_k$, $C_j = B \vee \overline{x_k}$: $C_i \diamond C_j := A \vee B$
- φ satisfied $\Leftrightarrow$ $\varphi \cup \{C_i \diamond C_j\}$ satisfied

Unsatisfiability proof (refutation) by resolution:

- DAG (directed acyclic graph): original clauses as sources (“axioms”)
- Other nodes C have exactly two predecessors C_i, C_j with $C = C_i \diamond C_j$
- Graph has the empty clause ε (aka $\square$) as sink
- For unsatisfiable formulas, there is always such a proof (but it can be exponential)

How can we learn?

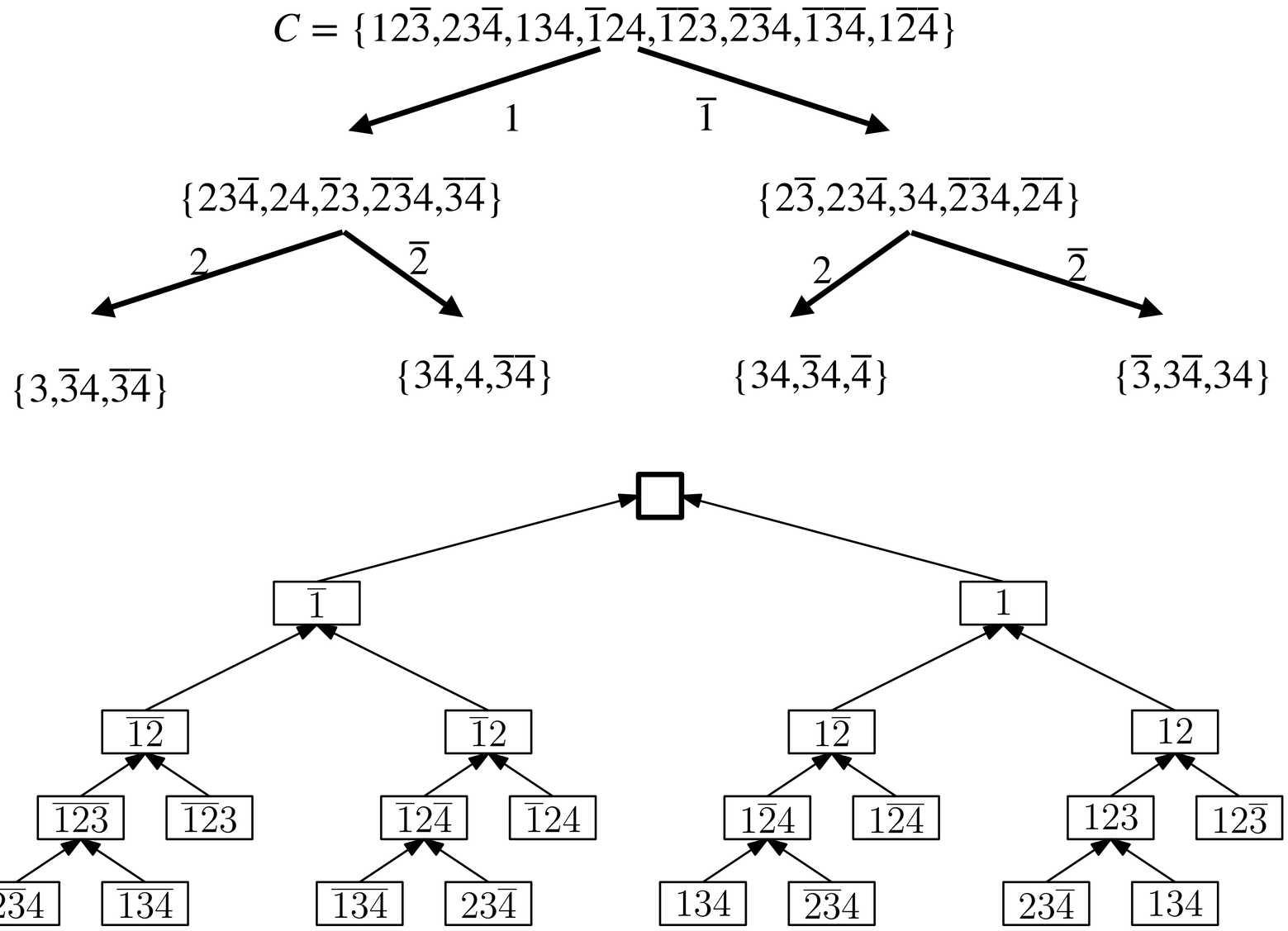
Idea: learning = generating new clauses. **Tool for that: *resolution*!**

- Clauses $C_i = A \vee x_k$, $C_j = B \vee \overline{x_k}$: $C_i \diamond C_j := A \vee B$
- φ satisfied $\Leftrightarrow$ $\varphi \cup \{C_i \diamond C_j\}$ satisfied

Unsatisfiability proof (refutation) by resolution:

- DAG (directed acyclic graph): original clauses as sources (“axioms”)
- Other nodes C have exactly two predecessors C_i, C_j with $C = C_i \diamond C_j$
- Graph has the empty clause ε (aka $\square$) as sink
- For unsatisfiable formulas, there is always such a proof (but it can be exponential)
- More general: resolution proof of A (if we can get to a sink clause A)

Unsatisfiability Proof Analogue to DPLL Search Tree



Conflict Driven Clause Learning (CDCL)

Conflict Driven Clause Learning (CDCL)

CDCL is the basis of modern SAT-Solvers. Basic idea:

Conflict Driven Clause Learning (CDCL)

CDCL is the basis of modern SAT-Solvers. Basic idea:

- At any point, we have a trail of literals

Conflict Driven Clause Learning (CDCL)

CDCL is the basis of modern SAT-Solvers. Basic idea:

- At any point, we have a trail of literals
- As in DPLL: split into decision levels by decisions

Conflict Driven Clause Learning (CDCL)

CDCL is the basis of modern SAT-Solvers. Basic idea:

- At any point, we have a trail of literals
- As in DPLL: split into decision levels by decisions
- Between decisions: literals produced by unit propagation

Conflict Driven Clause Learning (CDCL)

CDCL is the basis of modern SAT-Solvers. Basic idea:

- At any point, we have a trail of literals
- As in DPLL: split into decision levels by decisions
- Between decisions: literals produced by unit propagation
- When there is no propagation available, make *decision* on an unassigned variable

Conflict Driven Clause Learning (CDCL)

CDCL is the basis of modern SAT-Solvers. Basic idea:

- At any point, we have a trail of literals
- As in DPLL: split into decision levels by decisions
- Between decisions: literals produced by unit propagation
- When there is no propagation available, make *decision* on an unassigned variable
- Literals produced by UP store the clause that produced them as *reason*

Conflict Driven Clause Learning (CDCL)

CDCL is the basis of modern SAT-Solvers. Basic idea:

- At any point, we have a trail of literals
- As in DPLL: split into decision levels by decisions
- Between decisions: literals produced by unit propagation
- When there is no propagation available, make *decision* on an unassigned variable
- Literals produced by UP store the clause that produced them as *reason*
- Reasons are used to construct a *learnt clause* on each *conflict* by resolution

Conflict Driven Clause Learning (CDCL)

CDCL is the basis of modern SAT-Solvers. Basic idea:

- At any point, we have a trail of literals
- As in DPLL: split into decision levels by decisions
- Between decisions: literals produced by unit propagation
- When there is no propagation available, make *decision* on an unassigned variable
- Literals produced by UP store the clause that produced them as *reason*
- Reasons are used to construct a *learnt clause* on each *conflict* by resolution
- Backjumping: go back to the highest decision level where the conflict clause propagates

Conflict Driven Clause Learning (CDCL)

CDCL is the basis of modern SAT-Solvers. Basic idea:

- At any point, we have a trail of literals
- As in DPLL: split into decision levels by decisions
- Between decisions: literals produced by unit propagation
- When there is no propagation available, make *decision* on an unassigned variable
- Literals produced by UP store the clause that produced them as *reason*
- Reasons are used to construct a *learnt clause* on each *conflict* by resolution
- Backjumping: go back to the highest decision level where the conflict clause propagates
- Overall goal:

Conflict Driven Clause Learning (CDCL)

CDCL is the basis of modern SAT-Solvers. Basic idea:

- At any point, we have a trail of literals
- As in DPLL: split into decision levels by decisions
- Between decisions: literals produced by unit propagation
- When there is no propagation available, make *decision* on an unassigned variable
- Literals produced by UP store the clause that produced them as *reason*
- Reasons are used to construct a *learnt clause* on each *conflict* by resolution
- Backjumping: go back to the highest decision level where the conflict clause propagates
- Overall goal:
  - Extend trail to n literals with no conflict (satisfying assignment), or

Conflict Driven Clause Learning (CDCL)

CDCL is the basis of modern SAT-Solvers. Basic idea:

- At any point, we have a trail of literals
- As in DPLL: split into decision levels by decisions
- Between decisions: literals produced by unit propagation
- When there is no propagation available, make *decision* on an unassigned variable
- Literals produced by UP store the clause that produced them as *reason*
- Reasons are used to construct a *learnt clause* on each *conflict* by resolution
- Backjumping: go back to the highest decision level where the conflict clause propagates
- Overall goal:
  - Extend trail to n literals with no conflict (satisfying assignment), or
  - learn the empty clause (proof of unsatisfiability).

Example: waerden(3, 3; 9)

Example: waerden(3, 3; 9)

Find a bit string $x_1, \ldots, x_9 \in \mathbb{B}^9$, in which:

Example: waerden(3, 3; 9)

Find a bit string $x_1, \ldots, x_9 \in \mathbb{B}^9$, in which:

- no 3 0-entries have the same distance, and

Example: waerden(3, 3; 9)

Find a bit string $x_1, \ldots, x_9 \in \mathbb{B}^9$, in which:

- no 3 0-entries have the same distance, and
- no 3 1-entries have the same distance.

Example: waerden(3, 3; 9)

Find a bit string $x_1, \ldots, x_9 \in \mathbb{B}^9$, in which:

- no 3 0-entries have the same distance, and
- no 3 1-entries have the same distance.

For 8 instead of 9 such a bit string exists, but not for 9.

Example: waerden(3, 3; 9)

Find a bit string $x_1, \ldots, x_9 \in \mathbb{B}^9$, in which:

- no 3 0-entries have the same distance, and
- no 3 1-entries have the same distance.

For 8 instead of 9 such a bit string exists, but not for 9.

As SAT model:

Example: waerden(3, 3; 9)

Find a bit string $x_1, \ldots, x_9 \in \mathbb{B}^9$, in which:

- no 3 0-entries have the same distance, and
- no 3 1-entries have the same distance.

For 8 instead of 9 such a bit string exists, but not for 9.

As SAT model:

- variables $1,\ldots,9$

Example: waerden(3, 3; 9)

Find a bit string $x_1, \ldots, x_9 \in \mathbb{B}^9$, in which:

- no 3 0-entries have the same distance, and
- no 3 1-entries have the same distance.

For 8 instead of 9 such a bit string exists, but not for 9.

As SAT model:

- variables $1,\ldots,9$
- $123,\overline{123},234,\overline{234}, \ldots, 135,\overline{135},246,\overline{246}, \ldots, 147,\overline{147},258,\overline{258}, \ldots, 159,\overline{159}, \overline{1}$

Example: waerden(3, 3; 9)

Find a bit string $x_1, \ldots, x_9 \in \mathbb{B}^9$, in which:

- no 3 0-entries have the same distance, and
- no 3 1-entries have the same distance.

For 8 instead of 9 such a bit string exists, but not for 9.

As SAT model:

- variables $1,\ldots,9$
- $123,\overline{123},234,\overline{234}, \ldots, 135,\overline{135},246,\overline{246}, \ldots, 147,\overline{147},258,\overline{258}, \ldots, 159,\overline{159}, \overline{1}$
- These correspond to distances 1, 2, 3, 4; $\overline{1}$ for symmetry breaking.

Start of CDCL-Trail for Example

level	literal	reason
0	$\overline{1}$	$\overline{1}$

Start of CDCL-Trail for Example

level	literal	reason
0	$\overline{1}$	$\overline{1}$
1	$\overline{2}$	Δ
1	3	123

Start of CDCL-Trail for Example

level	literal	reason
0	$\overline{1}$	$\overline{1}$
1	$\overline{2}$	Δ
1	3	123
2	$\overline{4}$	Δ
2	6	246
2	7	147
2	$\overline{5}$	$\overline{357}$
2	8	258
2^*	$\overline{8}$	$\overline{678}$

Start of CDCL-Trail for Example

level	literal	reason
0	$\overline{1}$	$\overline{1}$
1	$\overline{2}$	Δ
1	3	123
2	$\overline{4}$	Δ
2	6	246
2	7	147
2	$\overline{5}$	$\overline{357}$
2	8	258
2^*	$\overline{8}$	$\overline{678}$

Goal: Learn clause enforcing a new literal at a level $d' < d$.

Necessary: at most one literal at level d.

Start of CDCL-Trail for Example

level	literal	reason
0	$\overline{1}$	$\overline{1}$
1	$\overline{2}$	Δ
1	3	123
2	$\overline{4}$	Δ
2	6	246
2	7	147
2	$\overline{5}$	$\overline{357}$
2	8	258
2^*	$\overline{8}$	$\overline{678}$

Goal: Learn clause enforcing a new literal at a level $d' < d$.

Necessary: at most one literal at level d.

Idea:

- Resolve reasons for conflict variable $\rightarrow L$
- Resolve with reason for latest literal in L
- Until only one from level d remains in L

Start of CDCL-Trail for Example

level	literal	reason
0	$\overline{1}$	$\overline{1}$
1	$\overline{2}$	Δ
1	3	123
2	$\overline{4}$	Δ
2	6	246
2	7	147
2	$\overline{5}$	$\overline{357}$
2	8	258
2^*	$\overline{8}$	$\overline{678}$

Goal: Learn clause enforcing a new literal at a level $d' < d$.

Necessary: at most one literal at level d.

Idea:

- Resolve reasons for conflict variable $\rightarrow L$
- Resolve with reason for latest literal in L
- Until only one from level d remains in L

Example:

$$258 \diamond_8 \overline{678} = 25\overline{67},\ 25\overline{67} \diamond_5 \overline{357} = 2\overline{367},$$
$$2\overline{367} \diamond_7 147 = 12\overline{3}4\overline{6},\ 12\overline{3}4\overline{6} \diamond_6 246 = 12\overline{3}4.$$

$12\overline{3}4$ has only one literal from level 2!

Here, 24 is an (obviously) better choice.

Continuation of Example

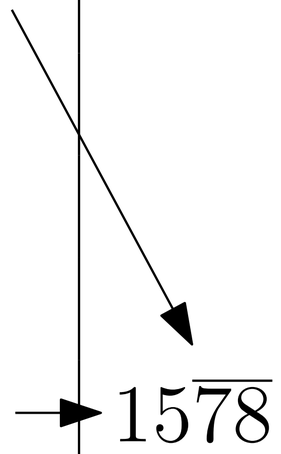
level	literal	reason
0	$\overline{1}$	$\overline{1}$
1	$\overline{2}$	Δ
1	3	123
1	4	24

Continuation of Example

level	literal	reason
0	$\overline{1}$	$\overline{1}$
1	$\overline{2}$	Δ
1	3	123
1	4	24
1	$\overline{5}$	$\overline{345}$
1	9	159
1	8	258
1	$\overline{6}$	$\overline{468}$
1	7	567
1	$\overline{9}$	$\overline{789}$

Continuation of Example

level	literal	reason
0	$\overline{1}$	$\overline{1}$
1	$\overline{2}$	Δ
1	3	123
1	4	24
1	$\overline{5}$	$\overline{345}$
1	9	159
1	8	258
1	$\overline{6}$	$\overline{468}$
1	7	567
1	$\overline{9}$	$\overline{789}$



Continuation of Example

level	literal	reason	
0	$\overline{1}$	$\overline{1}$	
1	$\overline{2}$	Δ	
1	3	123	$\rightarrow 12$
1	4	24	$\rightarrow 12\overline{3}$
1	$\overline{5}$	$\overline{345}$	$\rightarrow 12\overline{34}$
1	9	159	
1	8	258	$\rightarrow 125\overline{4}$
1	$\overline{6}$	$\overline{468}$	$\rightarrow 15\overline{48}$
1	7	567	$\rightarrow 156\overline{8}$
1	$\overline{9}$	$\overline{789}$	$\rightarrow 15\overline{78}$

Continuation of Example

level	literal	reason
0	$\overline{1}$	$\overline{1}$
0	2	12
1	4	Δ
1	$\overline{3}$	$\overline{234}$
1	5	135
1	$\overline{8}$	$\overline{258}$
1	$\overline{6}$	$\overline{246}$
1	7	678
1	9	369
1	$\overline{9}$	$\overline{579}$

Continuation of Example

level	literal	reason	
0	$\overline{1}$	$\overline{1}$	
0	2	12	
1	4	Δ	
1	$\overline{3}$	$\overline{234}$	$\rightarrow 1\overline{24}$
1	5	135	$\rightarrow 13\overline{24}$
1	$\overline{8}$	$\overline{258}$	$\rightarrow 3\overline{245}$
1	$\overline{6}$	$\overline{246}$	$\rightarrow 38\overline{245}$
1	7	678	$\rightarrow 368\overline{5}$
1	9	369	
1	$\overline{9}$	$\overline{579}$	$\rightarrow 36\overline{57}$

Continuation of Example

level	literal	reason
0	$\overline{1}$	$\overline{1}$
0	2	12
0	$\overline{4}$	$1\overline{24}$
0	7	147
1	$\overline{5}$	Δ
1	3	345
1	9	159
1	6	456
1	$\overline{8}$	$\overline{789}$
1	$\overline{3}$	$\overline{369}$

Continuation of Example

level	literal	reason	
0	$\overline{1}$	$\overline{1}$	
0	2	12	
0	$\overline{4}$	$1\overline{24}$	
0	7	147	
1	$\overline{5}$	Δ	
1	3	345	
1	9	159	145
1	6	456	$45\overline{9}$
1	$\overline{8}$	$\overline{789}$	
1	$\overline{3}$	$\overline{369}$	$45\overline{69}$

Continuation of Example

level	literal	reason
0	$\overline{1}$	$\overline{1}$
0	2	12
0	$\overline{4}$	$1\overline{24}$
0	7	147
0	5	145
0	$\overline{3}$	$\overline{357}$
0	$\overline{6}$	$\overline{567}$
0	8	468
0	9	369
0	$\overline{9}$	$\overline{579}$

Continuation of Example

level	literal	reason	
0	$\overline{1}$	$\overline{1}$	$\square$
0	2	12	1
0	$\overline{4}$	$1\overline{24}$	$1\overline{2}$
0	7	147	14
0	5	145	$14\overline{7}$
0	$\overline{3}$	$\overline{357}$	$\overline{57}$
0	$\overline{6}$	$\overline{567}$	$3\overline{57}$
0	8	468	
0	9	369	
0	$\overline{9}$	$\overline{579}$	$36\overline{57}$

Conflict Resolution

Conflict Resolution

General summary of what we saw in the example:

Conflict Resolution

General summary of what we saw in the example:

- On a conflict $x, \overline{x}$ in the trail (strictly speaking, $\overline{x}$ does not enter the trail):

Conflict Resolution

General summary of what we saw in the example:

- On a conflict $x, \overline{x}$ in the trail (strictly speaking, $\overline{x}$ does not enter the trail):
  - $x, \overline{x}$ cannot be decisions:

Conflict Resolution

General summary of what we saw in the example:

- On a conflict $x, \overline{x}$ in the trail (strictly speaking, $\overline{x}$ does not enter the trail):
  - $x, \overline{x}$ cannot be decisions:
    - Decisions are made and propagated at the beginning of the level

Conflict Resolution

General summary of what we saw in the example:

- On a conflict $x, \overline{x}$ in the trail (strictly speaking, $\overline{x}$ does not enter the trail):
  - $x, \overline{x}$ cannot be decisions:
    - Decisions are made and propagated at the beginning of the level
    - The conflict will be found on another variable; $x, \overline{x}$ have reason clauses

Conflict Resolution

General summary of what we saw in the example:

- On a conflict $x, \overline{x}$ in the trail (strictly speaking, $\overline{x}$ does not enter the trail):
  - $x, \overline{x}$ cannot be decisions:
    - Decisions are made and propagated at the beginning of the level
    - The conflict will be found on another variable; $x, \overline{x}$ have reason clauses
  - Initial conflict clause: resolution O on x of these reason clauses

Conflict Resolution

General summary of what we saw in the example:

- On a conflict $x, \overline{x}$ in the trail (strictly speaking, $\overline{x}$ does not enter the trail):
  - $x, \overline{x}$ cannot be decisions:
    - Decisions are made and propagated at the beginning of the level
    - The conflict will be found on another variable; $x, \overline{x}$ have reason clauses
  - Initial conflict clause: resolution O on x of these reason clauses
  - While O contains more than one literal from level d:

Conflict Resolution

General summary of what we saw in the example:

- On a conflict $x, \overline{x}$ in the trail (strictly speaking, $\overline{x}$ does not enter the trail):
  - $x, \overline{x}$ cannot be decisions:
    - Decisions are made and propagated at the beginning of the level
    - The conflict will be found on another variable; $x, \overline{x}$ have reason clauses
  - Initial conflict clause: resolution O on x of these reason clauses
  - While O contains more than one literal from level d:
    - Choose the latest such literal ℓ (it has a reason C); set $O \leftarrow O \diamond_\ell C$

Conflict Resolution

General summary of what we saw in the example:

- On a conflict $x, \overline{x}$ in the trail (strictly speaking, $\overline{x}$ does not enter the trail):
  - $x, \overline{x}$ cannot be decisions:
    - Decisions are made and propagated at the beginning of the level
    - The conflict will be found on another variable; $x, \overline{x}$ have reason clauses
  - Initial conflict clause: resolution O on x of these reason clauses
  - While O contains more than one literal from level d:
    - Choose the latest such literal ℓ (it has a reason C); set $O \leftarrow O \diamond_\ell C$
    - At some point (at the latest when the latest trail literal is the decision of level d):

Conflict Resolution

General summary of what we saw in the example:

- On a conflict $x, \overline{x}$ in the trail (strictly speaking, $\overline{x}$ does not enter the trail):
  - $x, \overline{x}$ cannot be decisions:
    - Decisions are made and propagated at the beginning of the level
    - The conflict will be found on another variable; $x, \overline{x}$ have reason clauses
  - Initial conflict clause: resolution O on x of these reason clauses
  - While O contains more than one literal from level d:
    - Choose the latest such literal ℓ (it has a reason C); set $O \leftarrow O \diamond_\ell C$
    - At some point (at the latest when the latest trail literal is the decision of level d):
      - Only one literal from level d, all other literals from $\leq d' < d$

Conflict Resolution

General summary of what we saw in the example:

- On a conflict $x, \overline{x}$ in the trail (strictly speaking, $\overline{x}$ does not enter the trail):
  - $x, \overline{x}$ cannot be decisions:
    - Decisions are made and propagated at the beginning of the level
    - The conflict will be found on another variable; $x, \overline{x}$ have reason clauses
  - Initial conflict clause: resolution O on x of these reason clauses
  - While O contains more than one literal from level d:
    - Choose the latest such literal ℓ (it has a reason C); set $O \leftarrow O \diamond_\ell C$
    - At some point (at the latest when the latest trail literal is the decision of level d):
      - Only one literal from level d, all other literals from $\leq d' < d$
      - Learn O, go back to level d' and append $\overline{\ell}$ to the trail with reason O

Conflict Resolution

General summary of what we saw in the example:

- On a conflict $x, \overline{x}$ in the trail (strictly speaking, $\overline{x}$ does not enter the trail):
  - $x, \overline{x}$ cannot be decisions:
    - Decisions are made and propagated at the beginning of the level
    - The conflict will be found on another variable; $x, \overline{x}$ have reason clauses
  - Initial conflict clause: resolution O on x of these reason clauses
  - While O contains more than one literal from level d:
    - Choose the latest such literal ℓ (it has a reason C); set $O \leftarrow O \diamond_\ell C$
    - At some point (at the latest when the latest trail literal is the decision of level d):
      - Only one literal from level d, all other literals from $\leq d' < d$
      - Learn O, go back to level d' and append $\overline{\ell}$ to the trail with reason O
  - Trail positions with only one remaining literal from d are called *Unique Implication Point* (UIP)

Conflict Resolution

General summary of what we saw in the example:

- On a conflict $x, \overline{x}$ in the trail (strictly speaking, $\overline{x}$ does not enter the trail):
  - $x, \overline{x}$ cannot be decisions:
    - Decisions are made and propagated at the beginning of the level
    - The conflict will be found on another variable; $x, \overline{x}$ have reason clauses
  - Initial conflict clause: resolution O on x of these reason clauses
  - While O contains more than one literal from level d:
    - Choose the latest such literal ℓ (it has a reason C); set $O \leftarrow O \diamond_\ell C$
    - At some point (at the latest when the latest trail literal is the decision of level d):
      - Only one literal from level d, all other literals from $\leq d' < d$
      - Learn O, go back to level d' and append $\overline{\ell}$ to the trail with reason O
  - Trail positions with only one remaining literal from d are called *Unique Implication Point* (UIP)
  - We usually take the first UIP we encounter (walking back up the trail)

Backjumping & Clause Learning: Advantages

Backjumping & Clause Learning: Advantages

Multiple Advantages:

Backjumping & Clause Learning: Advantages

Multiple Advantages:

- We learn from unsuccessful assignments

Backjumping & Clause Learning: Advantages

Multiple Advantages:

- We learn from unsuccessful assignments
- Similar to cutting planes: globally applicable

Backjumping & Clause Learning: Advantages

Multiple Advantages:

- We learn from unsuccessful assignments
- Similar to cutting planes: globally applicable
- Can skip over ‘independent’/‘unnecessary’ decision levels (example without $\overline{1}$):

Backjumping & Clause Learning: Advantages

Multiple Advantages:

- We learn from unsuccessful assignments
- Similar to cutting planes: globally applicable
- Can skip over ‘independent’/‘unnecessary’ decision levels (example without $\overline{1}$):

level	literal	reason
1	$\overline{6}$	Δ
2	$\overline{9}$	Δ
2	3	369
3	$\overline{4}$	Δ
3	5	456
3	8	468
3	2	246
3	$\overline{7}$	$\overline{357}$
3	$\overline{2}$	$\overline{258}$

Backjumping & Clause Learning: Advantages

Multiple Advantages:

- We learn from unsuccessful assignments
- Similar to cutting planes: globally applicable
- Can skip over ‘independent’/‘unnecessary’ decision levels (example without $\overline{1}$):

$\overline{258} \diamond_2 246 = 4\overline{5}6\overline{8},\ 4\overline{5}6\overline{8} \diamond_8 468 = 4\overline{5}6,$
$4\overline{5}6 \diamond_5 456 = 46$; we learn 46.

level	literal	reason
1	$\overline{6}$	Δ
2	$\overline{9}$	Δ
2	3	369
3	$\overline{4}$	Δ
3	5	456
3	8	468
3	2	246
3	$\overline{7}$	$\overline{357}$
3	$\overline{2}$	$\overline{258}$

Backjumping & Clause Learning: Advantages

Multiple Advantages:

- We learn from unsuccessful assignments
- Similar to cutting planes: globally applicable
- Can skip over ‘independent’/‘unnecessary’ decision levels (example without $\overline{1}$):

$\overline{258} \diamond_2 246 = 4\overline{5}6\overline{8},\ 4\overline{5}6\overline{8} \diamond_8 468 = 4\overline{5}6,$
$4\overline{5}6 \diamond_5 456 = 46$; we learn 46.

The decision $\overline{9}$ at the start of level 2 was unnecessary;
we go back to level 1 and propagate 4.

level	literal	reason
1	$\overline{6}$	Δ
2	$\overline{9}$	Δ
2	3	369
3	$\overline{4}$	Δ
3	5	456
3	8	468
3	2	246
3	$\overline{7}$	$\overline{357}$
3	$\overline{2}$	$\overline{258}$

Backjumping & Clause Learning: Advantages

Multiple Advantages:

- We learn from unsuccessful assignments
- Similar to cutting planes: globally applicable
- Can skip over ‘independent’/‘unnecessary’ decision levels (example without $\overline{1}$):

$\overline{258} \diamond_2 246 = 4\overline{5}6\overline{8},\ 4\overline{5}6\overline{8} \diamond_8 468 = 4\overline{5}6,$
$4\overline{5}6 \diamond_5 456 = 46$; we learn 46.

The decision $\overline{9}$ at the start of level 2 was unnecessary; we go back to level 1 and propagate 4.

It is also possible that the learnt clause does not contain any decision variable!

level	literal	reason
1	$\overline{6}$	Δ
2	$\overline{9}$	Δ
2	3	369
3	$\overline{4}$	Δ
3	5	456
3	8	468
3	2	246
3	$\overline{7}$	$\overline{357}$
3	$\overline{2}$	$\overline{258}$

Termination

Termination:

Termination

Termination:

- On each conflict we learn a new clause.

Termination

Termination:

- On each conflict we learn a new clause.
- The learnt clause enforces at least one literal ℓ in the target level

Termination

Termination:

- On each conflict we learn a new clause.
- The learnt clause enforces at least one literal ℓ in the target level
- If the clause was redundant or already present, we would have had ℓ earlier

Termination

Termination:

- On each conflict we learn a new clause.
- The learnt clause enforces at least one literal ℓ in the target level
- If the clause was redundant or already present, we would have had ℓ earlier
- So learning does not produce redundant or repeated clauses!

Termination

Termination:

- On each conflict we learn a new clause.
- The learnt clause enforces at least one literal ℓ in the target level
- If the clause was redundant or already present, we would have had ℓ earlier
- So learning does not produce redundant or repeated clauses!
- There are only finitely many possible clauses implied by φ.

Termination

Termination:

- On each conflict we learn a new clause.
- The learnt clause enforces at least one literal ℓ in the target level
- If the clause was redundant or already present, we would have had ℓ earlier
- So learning does not produce redundant or repeated clauses!
- There are only finitely many possible clauses implied by φ.

That implies termination!

Key Improvements

Key Improvements

Noteworthy improvements:

Key Improvements

Noteworthy improvements:

- (E)VSIDS

Key Improvements

Noteworthy improvements:

- (E)VSIDS
- Restarts

Key Improvements

Noteworthy improvements:

- (E)VSIDS
- Restarts
- Phase Saving

Key Improvements

Noteworthy improvements:

- (E)VSIDS
- Restarts
- Phase Saving
- Clause Deletion

Key Improvements

Noteworthy improvements:

- (E)VSIDS
- Restarts
- Phase Saving
- Clause Deletion
- Conflict Clause Minimization

Key Improvements

Noteworthy improvements:

- (E)VSIDS
- Restarts
- Phase Saving
- Clause Deletion
- Conflict Clause Minimization
- Pre- and Inprocessing

Key Improvements

Noteworthy improvements:

- (E)VSIDS
- Restarts
- Phase Saving
- Clause Deletion
- Conflict Clause Minimization
- Pre- and Inprocessing
- SAT/UNSAT Stages

Key Improvements

Noteworthy improvements:

- (E)VSIDS
- Restarts
- Phase Saving
- Clause Deletion
- Conflict Clause Minimization
- Pre- and Inprocessing
- SAT/UNSAT Stages
- Local Search

Key Improvements

Noteworthy improvements:

- (E)VSIDS
- Restarts
- Phase Saving
- Clause Deletion
- Conflict Clause Minimization
- Pre- and Inprocessing
- SAT/UNSAT Stages
- Local Search
- Extensions (Incrementality, Cardinality SAT, XOR clauses, …)

Exponential Variable State Independent Decaying Sum

Exponential Variable State Independent Decaying Sum

(E)VSIDS:

Exponential Variable State Independent Decaying Sum

(E)VSIDS:

- Scoring heuristic for variable selection (highest score is chosen)

Exponential Variable State Independent Decaying Sum

(E)VSIDS:

- Scoring heuristic for variable selection (highest score is chosen)
- Core idea: select variables that have recently been involved in many conflicts

Exponential Variable State Independent Decaying Sum

(E)VSIDS:

- Scoring heuristic for variable selection (highest score is chosen)
- Core idea: select variables that have recently been involved in many conflicts
- Implementation:

Exponential Variable State Independent Decaying Sum

(E)VSIDS:

- Scoring heuristic for variable selection (highest score is chosen)
- Core idea: select variables that have recently been involved in many conflicts
- Implementation:
  - Each variable has a score $S(x)$

Exponential Variable State Independent Decaying Sum

(E)VSIDS:

- Scoring heuristic for variable selection (highest score is chosen)
- Core idea: select variables that have recently been involved in many conflicts
- Implementation:
  - Each variable has a score $S(x)$
  - Variables that appear in conflict clause/during resolution: $S(x) \leftarrow S(x) + I$

Exponential Variable State Independent Decaying Sum

(E)VSIDS:

- Scoring heuristic for variable selection (highest score is chosen)
- Core idea: select variables that have recently been involved in many conflicts
- Implementation:
  - Each variable has a score $S(x)$
  - Variables that appear in conflict clause/during resolution: $S(x) \leftarrow S(x) + I$
  - Increment I keeps growing exponentially

Exponential Variable State Independent Decaying Sum

(E)VSIDS:

- Scoring heuristic for variable selection (highest score is chosen)
- Core idea: select variables that have recently been involved in many conflicts
- Implementation:
  - Each variable has a score $S(x)$
  - Variables that appear in conflict clause/during resolution: $S(x) \leftarrow S(x) + I$
  - Increment I keeps growing exponentially
  - In practice: periodically scale down all scores to avoid overflows

Exponential Variable State Independent Decaying Sum

(E)VSIDS:

- Scoring heuristic for variable selection (highest score is chosen)
- Core idea: select variables that have recently been involved in many conflicts
- Implementation:
  - Each variable has a score $S(x)$
  - Variables that appear in conflict clause/during resolution: $S(x) \leftarrow S(x) + I$
  - Increment I keeps growing exponentially
  - In practice: periodically scale down all scores to avoid overflows
- Focuses on a subset of variables (locality)

Exponential Variable State Independent Decaying Sum

(E)VSIDS:

- Scoring heuristic for variable selection (highest score is chosen)
- Core idea: select variables that have recently been involved in many conflicts
- Implementation:
  - Each variable has a score $S(x)$
  - Variables that appear in conflict clause/during resolution: $S(x) \leftarrow S(x) + I$
  - Increment I keeps growing exponentially
  - In practice: periodically scale down all scores to avoid overflows
- Focuses on a subset of variables (locality)
- Alternatives: VMTF, other conflict-based scores

Restarts

Restarts

It may seem stupid at first, but:

Restarts

It may seem stupid at first, but:

- SAT solvers restart fairly often

Restarts

It may seem stupid at first, but:

- SAT solvers restart fairly often
- Discard the decision levels from the trail

Restarts

It may seem stupid at first, but:

- SAT solvers restart fairly often
- Discard the decision levels from the trail
- Keep learnt clauses

Restarts

It may seem stupid at first, but:

- SAT solvers restart fairly often
- Discard the decision levels from the trail
- Keep learnt clauses
- Keep variable scores

Restarts

It may seem stupid at first, but:

- SAT solvers restart fairly often
- Discard the decision levels from the trail
- Keep learnt clauses
- Keep variable scores
- On the theoretical side: no longer a single resolution DAG → avoids theoretical LBs

Restarts

It may seem stupid at first, but:

- SAT solvers restart fairly often
- Discard the decision levels from the trail
- Keep learnt clauses
- Keep variable scores
- On the theoretical side: no longer a single resolution DAG → avoids theoretical LBs
- In practice: avoid getting ‘stuck’ in bad parts of the search space

Restarts

It may seem stupid at first, but:

- SAT solvers restart fairly often
- Discard the decision levels from the trail
- Keep learnt clauses
- Keep variable scores
- On the theoretical side: no longer a single resolution DAG → avoids theoretical LBs
- In practice: avoid getting ‘stuck’ in bad parts of the search space
- Often, restarts can be ‘partial’ (e.g., drop only some constant number of levels)

Restarts

It may seem stupid at first, but:

- SAT solvers restart fairly often
- Discard the decision levels from the trail
- Keep learnt clauses
- Keep variable scores
- On the theoretical side: no longer a single resolution DAG → avoids theoretical LBs
- In practice: avoid getting ‘stuck’ in bad parts of the search space
- Often, restarts can be ‘partial’ (e.g., drop only some constant number of levels)
- Typically based on number of conflicts

Phase Saving

Phase Saving

EVSIDS and the like select the variable; what value to try first?

Phase Saving

EVSIDS and the like select the variable; what value to try first?

- This is where phase saving comes into play

Phase Saving

EVSIDS and the like select the variable; what value to try first?

- This is where phase saving comes into play
- Essentially, one array/bitset P storing the previous value of each variable

Phase Saving

EVSIDS and the like select the variable; what value to try first?

- This is where phase saving comes into play
- Essentially, one array/bitset P storing the previous value of each variable
- On popping x from trail (backtracking/backjumping): save the value of x in P

Phase Saving

EVSIDS and the like select the variable; what value to try first?

- This is where phase saving comes into play
- Essentially, one array/bitset P storing the previous value of each variable
- On popping x from trail (backtracking/backjumping): save the value of x in P
- On making a decision: try $P[x]$ first rather than a constant or random value

Phase Saving

EVSIDS and the like select the variable; what value to try first?

- This is where phase saving comes into play
- Essentially, one array/bitset P storing the previous value of each variable
- On popping x from trail (backtracking/backjumping): save the value of x in P
- On making a decision: try $P[x]$ first rather than a constant or random value
- Very easy to implement; can have significant benefits

Phase Saving

EVSIDS and the like select the variable; what value to try first?

- This is where phase saving comes into play
- Essentially, one array/bitset P storing the previous value of each variable
- On popping x from trail (backtracking/backjumping): save the value of x in P
- On making a decision: try $P[x]$ first rather than a constant or random value
- Very easy to implement; can have significant benefits
- If we backjump past a component unrelated to our conflict,
  we would later recover the same assignment again

Phase Saving

EVSIDS and the like select the variable; what value to try first?

- This is where phase saving comes into play
- Essentially, one array/bitset P storing the previous value of each variable
- On popping x from trail (backtracking/backjumping): save the value of x in P
- On making a decision: try $P[x]$ first rather than a constant or random value
- Very easy to implement; can have significant benefits
- If we backjump past a component unrelated to our conflict, we would later recover the same assignment again
- It thus approximates some sort of ‘component caching’

Phase Saving

EVSIDS and the like select the variable; what value to try first?

- This is where phase saving comes into play
- Essentially, one array/bitset P storing the previous value of each variable
- On popping x from trail (backtracking/backjumping): save the value of x in P
- On making a decision: try $P[x]$ first rather than a constant or random value
- Very easy to implement; can have significant benefits
- If we backjump past a component unrelated to our conflict, we would later recover the same assignment again
- It thus approximates some sort of ‘component caching’
- Modern solvers: occasional rephasing (heuristics, longest conflict-free trail, random, …)

Clause Deletion

Clause Deletion

Original: keep, learnt: When to delete, how much to delete, which clauses to delete?

Clause Deletion

Original: keep, learnt: When to delete, how much to delete, which clauses to delete?

- Essentially, we need a scoring system for ‘usefulness’ of clauses

Clause Deletion

Original: keep, learnt: When to delete, how much to delete, which clauses to delete?

- Essentially, we need a scoring system for ‘usefulness’ of clauses
- Earliest approaches: based on clause length (length not a great quality measure)

Clause Deletion

Original: keep, learnt: When to delete, how much to delete, which clauses to delete?

- Essentially, we need a scoring system for ‘usefulness’ of clauses
- Earliest approaches: based on clause length (length not a great quality measure)
- Another idea: *relevance* (how many literals are unassigned, also not a great measure)

Clause Deletion

Original: keep, learnt: When to delete, how much to delete, which clauses to delete?

- Essentially, we need a scoring system for ‘usefulness’ of clauses
- Earliest approaches: based on clause length (length not a great quality measure)
- Another idea: *relevance* (how many literals are unassigned, also not a great measure)
- Activity-based (EVSIDS-like, clause score bumped for reasons in conflict analysis)

Clause Deletion

Original: keep, learnt: When to delete, how much to delete, which clauses to delete?

- Essentially, we need a scoring system for ‘usefulness’ of clauses
- Earliest approaches: based on clause length (length not a great quality measure)
- Another idea: *relevance* (how many literals are unassigned, also not a great measure)
- Activity-based (EVSIDS-like, clause score bumped for reasons in conflict analysis)
- **Glue score** (also Literal Block Distance, **LBD**): key improvement!

Clause Deletion

Original: keep, learnt: When to delete, how much to delete, which clauses to delete?

- Essentially, we need a scoring system for ‘usefulness’ of clauses
- Earliest approaches: based on clause length (length not a great quality measure)
- Another idea: *relevance* (how many literals are unassigned, also not a great measure)
- Activity-based (EVSIDS-like, clause score bumped for reasons in conflict analysis)
- **Glue score** (also Literal Block Distance, **LBD**): key improvement!
  - How many different decision levels among the literals of the clause?

Clause Deletion

Original: keep, learnt: When to delete, how much to delete, which clauses to delete?

- Essentially, we need a scoring system for ‘usefulness’ of clauses
- Earliest approaches: based on clause length (length not a great quality measure)
- Another idea: *relevance* (how many literals are unassigned, also not a great measure)
- Activity-based (EVSIDS-like, clause score bumped for reasons in conflict analysis)
- **Glue score** (also Literal Block Distance, **LBD**): key improvement!
  - How many different decision levels among the literals of the clause?
  - Computed at learning time; typically updated when clauses occur in conflict analysis

Clause Deletion

Original: keep, learnt: When to delete, how much to delete, which clauses to delete?

- Essentially, we need a scoring system for ‘usefulness’ of clauses
- Earliest approaches: based on clause length (length not a great quality measure)
- Another idea: *relevance* (how many literals are unassigned, also not a great measure)
- Activity-based (EVSIDS-like, clause score bumped for reasons in conflict analysis)
- **Glue score** (also Literal Block Distance, **LBD**): key improvement!
  - How many different decision levels among the literals of the clause?
  - Computed at learning time; typically updated when clauses occur in conflict analysis
  - Typically, at each deletion round, delete the highest LBD half of clauses

Clause Deletion

Original: keep, learnt: When to delete, how much to delete, which clauses to delete?

- Essentially, we need a scoring system for ‘usefulness’ of clauses
- Earliest approaches: based on clause length (length not a great quality measure)
- Another idea: *relevance* (how many literals are unassigned, also not a great measure)
- Activity-based (EVSIDS-like, clause score bumped for reasons in conflict analysis)
- **Glue score** (also Literal Block Distance, **LBD**): key improvement!
  - How many different decision levels among the literals of the clause?
  - Computed at learning time; typically updated when clauses occur in conflict analysis
  - Typically, at each deletion round, delete the highest LBD half of clauses
  - Keep clauses that have been used since the last deletion round

Clause Deletion

Original: keep, learnt: When to delete, how much to delete, which clauses to delete?

- Essentially, we need a scoring system for ‘usefulness’ of clauses
- Earliest approaches: based on clause length (length not a great quality measure)
- Another idea: *relevance* (how many literals are unassigned, also not a great measure)
- Activity-based (EVSIDS-like, clause score bumped for reasons in conflict analysis)
- **Glue score** (also Literal Block Distance, **LBD**): key improvement!
  - How many different decision levels among the literals of the clause?
  - Computed at learning time; typically updated when clauses occur in conflict analysis
  - Typically, at each deletion round, delete the highest LBD half of clauses
  - Keep clauses that have been used since the last deletion round
- **Modern (2015+)**: tiered approach, e.g.:

Clause Deletion

Original: keep, learnt: When to delete, how much to delete, which clauses to delete?

- Essentially, we need a scoring system for ‘usefulness’ of clauses
- Earliest approaches: based on clause length (length not a great quality measure)
- Another idea: *relevance* (how many literals are unassigned, also not a great measure)
- Activity-based (EVSIDS-like, clause score bumped for reasons in conflict analysis)
- **Glue score** (also Literal Block Distance, **LBD**): key improvement!
  - How many different decision levels among the literals of the clause?
  - Computed at learning time; typically updated when clauses occur in conflict analysis
  - Typically, at each deletion round, delete the highest LBD half of clauses
  - Keep clauses that have been used since the last deletion round
- **Modern (2015+)**: tiered approach, e.g.:
  - core tier: low LBD (1-3); keep essentially forever

Clause Deletion

Original: keep, learnt: When to delete, how much to delete, which clauses to delete?

- Essentially, we need a scoring system for ‘usefulness’ of clauses
- Earliest approaches: based on clause length (length not a great quality measure)
- Another idea: *relevance* (how many literals are unassigned, also not a great measure)
- Activity-based (EVSIDS-like, clause score bumped for reasons in conflict analysis)
- **Glue score** (also Literal Block Distance, **LBD**): key improvement!
  - How many different decision levels among the literals of the clause?
  - Computed at learning time; typically updated when clauses occur in conflict analysis
  - Typically, at each deletion round, delete the highest LBD half of clauses
  - Keep clauses that have been used since the last deletion round
- **Modern (2015+)**: tiered approach, e.g.:
  - core tier: low LBD (1-3); keep essentially forever
  - mid tier: medium LBD; demoted if they are not used for sufficiently long

Clause Deletion

Original: keep, learnt: When to delete, how much to delete, which clauses to delete?

- Essentially, we need a scoring system for ‘usefulness’ of clauses
- Earliest approaches: based on clause length (length not a great quality measure)
- Another idea: *relevance* (how many literals are unassigned, also not a great measure)
- Activity-based (EVSIDS-like, clause score bumped for reasons in conflict analysis)
- **Glue score** (also Literal Block Distance, **LBD**): key improvement!
  - How many different decision levels among the literals of the clause?
  - Computed at learning time; typically updated when clauses occur in conflict analysis
  - Typically, at each deletion round, delete the highest LBD half of clauses
  - Keep clauses that have been used since the last deletion round
- **Modern (2015+)**: tiered approach, e.g.:
  - core tier: low LBD (1-3); keep essentially forever
  - mid tier: medium LBD; demoted if they are not used for sufficiently long
  - local tier: high LBD/demoted mid tier; aggressively deleted based on activity

Clause Deletion

Original: keep, learnt: When to delete, how much to delete, which clauses to delete?

- Essentially, we need a scoring system for ‘usefulness’ of clauses
- Earliest approaches: based on clause length (length not a great quality measure)
- Another idea: *relevance* (how many literals are unassigned, also not a great measure)
- Activity-based (EVSIDS-like, clause score bumped for reasons in conflict analysis)
- **Glue score** (also Literal Block Distance, **LBD**): key improvement!
  - How many different decision levels among the literals of the clause?
  - Computed at learning time; typically updated when clauses occur in conflict analysis
  - Typically, at each deletion round, delete the highest LBD half of clauses
  - Keep clauses that have been used since the last deletion round
- **Modern (2015+)**: tiered approach, e.g.:
  - core tier: low LBD (1-3); keep essentially forever
  - mid tier: medium LBD; demoted if they are not used for sufficiently long
  - local tier: high LBD/demoted mid tier; aggressively deleted based on activity
  - Clauses can migrate between tiers (LBD recomputed in conflict analysis)

Conflict Clause Minimization

Conflict Clause Minimization

Attempt to minimize the conflict clause:

Conflict Clause Minimization

Attempt to minimize the conflict clause:

- A literal ℓ is *removable* from the learnt conflict clause O if every ℓ' in its reason is

Conflict Clause Minimization

Attempt to minimize the conflict clause:

- A literal ℓ is *removable* from the learnt conflict clause O if every ℓ' in its reason is
  - fixed (i.e., in the trail at level 0), or

Conflict Clause Minimization

Attempt to minimize the conflict clause:

- A literal ℓ is *removable* from the learnt conflict clause O if every ℓ' in its reason is
  - fixed (i.e., in the trail at level 0), or
  - itself present in the conflict clause, or

Conflict Clause Minimization

Attempt to minimize the conflict clause:

- A literal ℓ is *removable* from the learnt conflict clause O if every ℓ' in its reason is
  - fixed (i.e., in the trail at level 0), or
  - itself present in the conflict clause, or
  - itself removable (recursive definition, moving backwards in the trail).

Conflict Clause Minimization

Attempt to minimize the conflict clause:

- A literal ℓ is *removable* from the learnt conflict clause O if every ℓ' in its reason is
  - fixed (i.e., in the trail at level 0), or
  - itself present in the conflict clause, or
  - itself removable (recursive definition, moving backwards in the trail).
- Removal works, just like conflict clause construction, by resolution (up the trail).

Conflict Clause Minimization

Attempt to minimize the conflict clause:

- A literal ℓ is *removable* from the learnt conflict clause O if every ℓ' in its reason is
  - fixed (i.e., in the trail at level 0), or
  - itself present in the conflict clause, or
  - itself removable (recursive definition, moving backwards in the trail).
- Removal works, just like conflict clause construction, by resolution (up the trail).
- A decision (which has no reason) is never removable (but can count as ‘present’).

Conflict Clause Minimization

Attempt to minimize the conflict clause:

- A literal ℓ is *removable* from the learnt conflict clause O if every ℓ' in its reason is
  - fixed (i.e., in the trail at level 0), or
  - itself present in the conflict clause, or
  - itself removable (recursive definition, moving backwards in the trail).
- Removal works, just like conflict clause construction, by resolution (up the trail).
- A decision (which has no reason) is never removable (but can count as ‘present’).
- Memoization and additional heuristic filters allow an efficient implementation.

Conflict Clause Minimization

Attempt to minimize the conflict clause:

- A literal ℓ is *removable* from the learnt conflict clause O if every ℓ' in its reason is
  - fixed (i.e., in the trail at level 0), or
  - itself present in the conflict clause, or
  - itself removable (recursive definition, moving backwards in the trail).
- Removal works, just like conflict clause construction, by resolution (up the trail).
- A decision (which has no reason) is never removable (but can count as ‘present’).
- Memoization and additional heuristic filters allow an efficient implementation.
- Other technique: scan for $x, y \in O$ with $x \to y$; then remove x

Pre- and Inprocessing

Pre- and Inprocessing

Some preprocessing/inprocessing techniques (before and during solving):

Pre- and Inprocessing

Some preprocessing/inprocessing techniques (before and during solving):

- Subsumed Clause Detection

Pre- and Inprocessing

Some preprocessing/inprocessing techniques (before and during solving):

- Subsumed Clause Detection
- Bounded Variable Elimination, BVE

Pre- and Inprocessing

Some preprocessing/inprocessing techniques (before and during solving):

- Subsumed Clause Detection
- Bounded Variable Elimination, BVE
- Fixed, Failed and Equivalent Literals, Hyper Binary Resolution, Implication Graph

Pre- and Inprocessing

Some preprocessing/inprocessing techniques (before and during solving):

- Subsumed Clause Detection
- Bounded Variable Elimination, BVE
- Fixed, Failed and Equivalent Literals, Hyper Binary Resolution, Implication Graph
- Vivification (aka Distillation)

Pre- and Inprocessing

Some preprocessing/inprocessing techniques (before and during solving):

- Subsumed Clause Detection
- Bounded Variable Elimination, BVE
- Fixed, Failed and Equivalent Literals, Hyper Binary Resolution, Implication Graph
- Vivification (aka Distillation)
- Bounded Variable Addition

Pre- and Inprocessing

Some preprocessing/inprocessing techniques (before and during solving):

- Subsumed Clause Detection
- Bounded Variable Elimination, BVE
- Fixed, Failed and Equivalent Literals, Hyper Binary Resolution, Implication Graph
- Vivification (aka Distillation)
- Bounded Variable Addition
- On-the-fly Subsumption

Pre- and Inprocessing

Some preprocessing/inprocessing techniques (before and during solving):

- Subsumed Clause Detection
- Bounded Variable Elimination, BVE
- Fixed, Failed and Equivalent Literals, Hyper Binary Resolution, Implication Graph
- Vivification (aka Distillation)
- Bounded Variable Addition
- On-the-fly Subsumption
- Blocked Clause Elimination

Pre- and Inprocessing

Some preprocessing/inprocessing techniques (before and during solving):

- Subsumed Clause Detection
- Bounded Variable Elimination, BVE
- Fixed, Failed and Equivalent Literals, Hyper Binary Resolution, Implication Graph
- Vivification (aka Distillation)
- Bounded Variable Addition
- On-the-fly Subsumption
- Blocked Clause Elimination
- …

Pre- and Inprocessing

Some preprocessing/inprocessing techniques (before and during solving):

- Subsumed Clause Detection
- Bounded Variable Elimination, BVE
- Fixed, Failed and Equivalent Literals, Hyper Binary Resolution, Implication Graph
- Vivification (aka Distillation)
- Bounded Variable Addition
- On-the-fly Subsumption
- Blocked Clause Elimination
- …

Details can easily be found online; usually not deep theory, but hard engineering:

Pre- and Inprocessing

Some preprocessing/inprocessing techniques (before and during solving):

- Subsumed Clause Detection
- Bounded Variable Elimination, BVE
- Fixed, Failed and Equivalent Literals, Hyper Binary Resolution, Implication Graph
- Vivification (aka Distillation)
- Bounded Variable Addition
- On-the-fly Subsumption
- Blocked Clause Elimination
- …

Details can easily be found online; usually not deep theory, but hard engineering:

when to run, how long to run, what candidates to run on, implementation details, …

Simple Pre- and Inprocessing

Subsumed Clause Detection:

A clause A is **subsumed** if it contains another clause $B \subsetneq A$.

A is a strictly weaker constraint than B and can be dropped.

A clause $A \supseteq \{x, \bar{x}\}$ is a *tautology* and can also be dropped.

Simple Pre- and Inprocessing

Subsumed Clause Detection:
A clause A is **subsumed** if it contains another clause $B \subsetneq A$.
A is a strictly weaker constraint than B and can be dropped.
A clause $A \supseteq \{x, \bar{x}\}$ is a *tautology* and can also be dropped.

Bounded Variable Elimination (BVE): the single most effective preprocessing step.
Let φ consist of clauses C; let $S_+(x) = \{A \in C \mid x \in A\}$, $S\_(x) = \{B \in C \mid \bar{x} \in B\}$ and let $R_\pm(x) = \{A \diamond_x B \mid A \in S_+(x), B \in S\_(x)\}$ (drop tautologies).

Subsumed Clause Detection:
A clause A is **subsumed** if it contains another clause $B \subsetneq A$.
A is a strictly weaker constraint than B and can be dropped.
A clause $A \supseteq \{x, \bar{x}\}$ is a *tautology* and can also be dropped.

Bounded Variable Elimination (BVE): the single most effective preprocessing step.
Let φ consist of clauses C; let $S_+(x) = \{A \in C \mid x \in A\}$, $S\_(x) = \{B \in C \mid \bar{x} \in B\}$ and let $R_\pm(x) = \{A \diamond_x B \mid A \in S_+(x), B \in S\_(x)\}$ (drop tautologies).

Then, $\mathrm{eliminate}(\varphi, x) := \varphi \setminus (S_+(x) \cup S\_(x)) \cup R_\pm(x)$ is satisfiable iff φ is.
BVE replaces φ by $\mathrm{eliminate}(\varphi, x)$ if it has at most some constant g more clauses.

SAT/UNSAT Stages

SAT/UNSAT Stages

Different heuristics handle SAT and UNSAT formulas differently well.

SAT/UNSAT Stages

Different heuristics handle SAT and UNSAT formulas differently well.

Idea:

SAT/UNSAT Stages

Different heuristics handle SAT and UNSAT formulas differently well.

Idea:

- Switch between stages to cover both SAT and UNSAT formulas reasonably well.

SAT/UNSAT Stages

Different heuristics handle SAT and UNSAT formulas differently well.

Idea:

- Switch between stages to cover both SAT and UNSAT formulas reasonably well.
- Can help in both directions (finding good clauses vs. finding good assignments)

SAT/UNSAT Stages

Different heuristics handle SAT and UNSAT formulas differently well.

Idea:

- Switch between stages to cover both SAT and UNSAT formulas reasonably well.
- Can help in both directions (finding good clauses vs. finding good assignments)
- Change variable heuristics (e.g., from EVSIDS for SAT to VMTF for UNSAT)

SAT/UNSAT Stages

Different heuristics handle SAT and UNSAT formulas differently well.

Idea:

- Switch between stages to cover both SAT and UNSAT formulas reasonably well.
- Can help in both directions (finding good clauses vs. finding good assignments)
- Change variable heuristics (e.g., from EVSIDS for SAT to VMTF for UNSAT)
- Change clause management parameters

SAT/UNSAT Stages

Different heuristics handle SAT and UNSAT formulas differently well.

Idea:

- Switch between stages to cover both SAT and UNSAT formulas reasonably well.
- Can help in both directions (finding good clauses vs. finding good assignments)
- Change variable heuristics (e.g., from EVSIDS for SAT to VMTF for UNSAT)
- Change clause management parameters
- Enable extra heuristics (e.g., local search methods for SAT stage)

SAT/UNSAT Stages

Different heuristics handle SAT and UNSAT formulas differently well.

Idea:

- Switch between stages to cover both SAT and UNSAT formulas reasonably well.
- Can help in both directions (finding good clauses vs. finding good assignments)
- Change variable heuristics (e.g., from EVSIDS for SAT to VMTF for UNSAT)
- Change clause management parameters
- Enable extra heuristics (e.g., local search methods for SAT stage)
- Stage changes, e.g., on some restart, based on heuristics

Local Search

Local Search

Idea: Search a ‘good’ assignment via local search and put it into the saved phases.

Local Search

Idea: Search a ‘good’ assignment via local search and put it into the saved phases.

- **probSAT:** starting from an assignment (e.g., current phases), take an unsatisfied clause and flip a variable chosen with probability proportional to a function of its ‘break count’ (the number of clauses that become unsatisfied by the flip).

Local Search

Idea: Search a ‘good’ assignment via local search and put it into the saved phases.

- **probSAT:** starting from an assignment (e.g., current phases), take an unsatisfied clause and flip a variable chosen with probability proportional to a function of its ‘break count’ (the number of clauses that become unsatisfied by the flip).

- **YalSAT:** probSAT with restarts; used in kissat, CaDiCaL and other modern solvers.

Local Search

Idea: Search a ‘good’ assignment via local search and put it into the saved phases.

- **probSAT:** starting from an assignment (e.g., current phases), take an unsatisfied clause and flip a variable chosen with probability proportional to a function of its ‘break count’ (the number of clauses that become unsatisfied by the flip).

- **YalSAT:** probSAT with restarts; used in kissat, CaDiCaL and other modern solvers.

- Very efficient to implement (maintain `nsat[c]` for each clause).

Local Search

Idea: Search a ‘good’ assignment via local search and put it into the saved phases.

- **probSAT:** starting from an assignment (e.g., current phases), take an unsatisfied clause and flip a variable chosen with probability proportional to a function of its ‘break count’ (the number of clauses that become unsatisfied by the flip).
- **YalSAT:** probSAT with restarts; used in kissat, CaDiCaL and other modern solvers.
- Very efficient to implement (maintain `nsat[c]` for each clause).
- Alternative approaches exist (e.g., make and break count, more expensive scoring); so far, they seem to perform worse.

Local Search

Idea: Search a ‘good’ assignment via local search and put it into the saved phases.

- **probSAT:** starting from an assignment (e.g., current phases), take an unsatisfied clause and flip a variable chosen with probability proportional to a function of its ‘break count’ (the number of clauses that become unsatisfied by the flip).
- **YalSAT:** probSAT with restarts; used in kissat, CaDiCaL and other modern solvers.
- Very efficient to implement (maintain `nsat[c]` for each clause).
- Alternative approaches exist (e.g., make and break count, more expensive scoring); so far, they seem to perform worse.
- Some information transfer to CDCL (beyond phases), e.g., score updates.

Summary: SAT

Summary: SAT

- Modern SAT solvers are based on CDCL

Summary: SAT

- Modern SAT solvers are based on CDCL
- Most of the solver time is usually spent on unit propagation (highly optimized via 2WL)

Summary: SAT

- Modern SAT solvers are based on CDCL
- Most of the solver time is usually spent on unit propagation (highly optimized via 2WL)
- UP is extremely fast, allowing solvers to explore many assignments

Summary: SAT

- Modern SAT solvers are based on CDCL
- Most of the solver time is usually spent on unit propagation (highly optimized via 2WL)
- UP is extremely fast, allowing solvers to explore many assignments
- If logic is at the core of the problem:

Summary: SAT

- Modern SAT solvers are based on CDCL
- Most of the solver time is usually spent on unit propagation (highly optimized via 2WL)
- UP is extremely fast, allowing solvers to explore many assignments
- If logic is at the core of the problem:
  - if it can be stated as a (sequence of) decision problems, SAT may be the best tool

Summary: SAT

- Modern SAT solvers are based on CDCL
- Most of the solver time is usually spent on unit propagation (highly optimized via 2WL)
- UP is extremely fast, allowing solvers to explore many assignments
- If logic is at the core of the problem:
  - if it can be stated as a (sequence of) decision problems, SAT may be the best tool
  - incremental use can be worthwhile; in particular, adding clauses is cheap

Summary: SAT

- Modern SAT solvers are based on CDCL
- Most of the solver time is usually spent on unit propagation (highly optimized via 2WL)
- UP is extremely fast, allowing solvers to explore many assignments
- If logic is at the core of the problem:
  - if it can be stated as a (sequence of) decision problems, SAT may be the best tool
  - incremental use can be worthwhile; in particular, adding clauses is cheap
- Some modern solvers (CaDiCaL) allow customization (custom propagators, …)

Summary: SAT

- Modern SAT solvers are based on CDCL
- Most of the solver time is usually spent on unit propagation (highly optimized via 2WL)
- UP is extremely fast, allowing solvers to explore many assignments
- If logic is at the core of the problem:
  - if it can be stated as a (sequence of) decision problems, SAT may be the best tool
  - incremental use can be worthwhile; in particular, adding clauses is cheap
- Some modern solvers (CaDiCaL) allow customization (custom propagators, …)
- Many hyperparameters, optimizations and tweaks

Summary: SAT

- Modern SAT solvers are based on CDCL
- Most of the solver time is usually spent on unit propagation (highly optimized via 2WL)
- UP is extremely fast, allowing solvers to explore many assignments
- If logic is at the core of the problem:
  - if it can be stated as a (sequence of) decision problems, SAT may be the best tool
  - incremental use can be worthwhile; in particular, adding clauses is cheap
- Some modern solvers (CaDiCaL) allow customization (custom propagators, …)
- Many hyperparameters, optimizations and tweaks
- **Relevant for the exam:** CDCL, trail/decision levels, conflict resolution, high-level ideas (general idea of variable selection, existence of pre-/inprocessing, …)