A Modern SAT Solver

Emina Torlak
emina@cs.washington.edu
Today

Last lecture
- Review of propositional logic and the DPLL algorithm

Today
- The CDCL algorithm at the core of modern SAT solvers:
  - 3 important extensions of DPLL
  - Engineering matters
A brief review of DPLL

// Returns true if the CNF formula F is // satisfiable; otherwise returns false.

DPLL(F)
  G ← BCP(F)
  if G = T then return true
  if G = ⊥ then return false
  p ← choose(vars(G))
  return DPLL(G{p ↦ T}) ||
       DPLL(G{p ↦ ⊥})

Boolean constraint propagation applies unit resolution until fixed point:

\[ \beta b_1 \lor \ldots \lor b_m \lor \neg \beta \]

\[ b_1 \lor \ldots \lor b_m \]
A brief review of DPLL

// Returns true if the CNF formula F is // satisfiable; otherwise returns false.

DPLL(F)
G ← BCP(F)
if G = T then return true
if G = ⊥ then return false
p ← choose(vars(G))
return DPLL(G{p → T}) ||
    DPLL(G{p → ⊥})

Boolean constraint propagation applies unit resolution until fixed point:
β b₁ ∨ … ∨ bₘ ∨ ¬β
b₁ ∨ … ∨ bₘ

Okay for randomly generated CNFs, but not for practical ones. Why?
A brief review of DPLL

// Returns true if the CNF formula F is satisfiable; otherwise returns false.

DPLL(F)
G ← BCP(F)
if G = ⊤ then return true
if G = ⊥ then return false
p ← choose(vars(G))
return DPLL(G{p ↦ ⊤}) ||
DPLL(G{p ↦ ⊥})

No learning: throws away all the work performed to conclude that the current partial assignment (PA) is bad. Revisits bad PAs that lead to conflict due to the same root cause.
A brief review of DPLL

// Returns true if the CNF formula F is satisfiable; otherwise returns false.

DPLL(F)
    G ← BCP(F)
    if G = T then return true
    if G = ⊥ then return false
    p ← choose(vars(G))
    return DPLL(G{p ↦ T}) ||
             DPLL(G{p ↦ ⊥})

No learning: throws away all the work performed to conclude that the current partial assignment (PA) is bad. Revisits bad PAs that lead to conflict due to the same root cause.

Chronological backtracking: backtracks one level, even if it can be deduced that the current PA became doomed at a lower level.
A brief review of DPLL

// Returns true if the CNF formula F is satisfiable; otherwise returns false.

DPLL(F)
    G ← BCP(F)
    if G = T then return true
    if G = ⊥ then return false
    p ← choose(vars(G))
    return DPLL(G{p ↦ T}) || DPLL(G{p ↦ ⊥})

---

**Chronological backtracking:** backtracks one level, even if it can be deduced that the current PA became doomed at a lower level.

**No learning:** throws away all the work performed to conclude that the current partial assignment (PA) is bad. Revisits bad PAs that lead to conflict due to the same root cause.

**Naive decisions:** picks an arbitrary variable to branch on. Fails to consider the state of the search to make heuristically better decisions.
Conflict-Driven Clause Learning (CDCL)

CDCL(F)
A ← {}
if BCP(F,A) = conflict then return false
level ← 0
while hasUnassignedVars(F)
    level ← level + 1
    A ← A ∪ \{ \text{Decide}(F,A) \}
while BCP(F,A) = conflict
    ⟨b, c⟩ ← \text{AnalyzeConflict}()
    F ← F ∪ \{c\}
    if b < 0 then return false
    else BACKTRACK(F,A,b)
        level ← b
return true
Conflict-Driven Clause Learning (CDCL)

CDCL(F)
A ← {}
if BCP(F, A) = conflict then return false
level ← 0
while hasUnassignedVars(F)
    level ← level + 1
    A ← A ∪ { DECIDE(F, A) }
while BCP(F, A) = conflict
    ⟨b, c⟩ ← ANALYZECONFLICT()
    F ← F ∪ {c}
    if b < 0 then return false
else BACKTRACK(F, A, b)
    level ← b
return true

Learning: F augmented with a conflict clause that summarizes the root cause of the conflict.
Conflict-Driven Clause Learning (CDCL)

CDCL(F)
A ← {} 
if BCP(F,A) = conflict then return false
level ← 0
while hasUnassignedVars(F)
  level ← level + 1
  A ← A ∪ { DECIDE(F,A) }
while BCP(F,A) = conflict
  ⟨b, c⟩ ← ANALYZECONFLICT()
  F ← F ∪ {c}
  if b < 0 then return false
  else BACKTRACK(F, A, b)
  level ← b
return true

Learning: F augmented with a conflict clause that summarizes the root cause of the conflict.

Non-chronological backtracking: backtracks b levels, based on the cause of the conflict.
Conflict-Driven Clause Learning (CDCL)

\[
\text{CDCL}(F)
\]
\[
A \leftarrow \{\}
\]
\[
\text{if } BCP(F, A) = \text{conflict then return false}
\]
\[
\text{level } \leftarrow 0
\]
\[
\text{while hasUnassignedVars}(F)
\]
\[
\text{level } \leftarrow \text{level } + 1
\]
\[
A \leftarrow A \cup \{ \text{DECIDE}(F, A) \}
\]
\[
\text{while } BCP(F, A) = \text{conflict}
\]
\[
\langle b, c \rangle \leftarrow \text{ANALYZECONFLICT}()
\]
\[
F \leftarrow F \cup \{c\}
\]
\[
\text{if } b < 0 \text{ then return false}
\]
\[
\text{else BACKTRACK}(F, A, b)
\]
\[
\text{level } \leftarrow b
\]
\[
\text{return true}
\]

Decision heuristics choose the next literal to add to the current partial assignment based on the state of the search.

Learning: F augmented with a conflict clause that summarizes the root cause of the conflict.

Non-chronological backtracking: backtracks b levels, based on the cause of the conflict.
CDCL by example

CDCL(F)
A ← \{\}
if BCP(F,A) = conflict then return false
level ← 0
while hasUnassignedVars(F)
    level ← level + 1
    A ← A ∪ \{ DECIDE(F,A) \}
while BCP(F,A) = conflict
    ⟨b, c⟩ ← ANALYZECONFLICT()
    F ← F ∪ \{c\}
    if b < 0 then return false
    else BACKTRACK(F,A,b)
        level ← b
return true

F = \{ c_1, c_2, c_3, c_4, c_5, c_6, \ldots, c_9 \}
c_1 : \neg x_1 \lor x_2 \lor \neg x_4
c_2 : \neg x_1 \lor \neg x_2 \lor x_3
c_3 : \neg x_3 \lor x_4
c_4 : x_4 \lor x_5 \lor x_6
c_5 : \neg x_5 \lor x_7
c_6 : \neg x_6 \lor x_7 \lor \neg x_8
\ldots
\ldots
CDCL by example

```
CDCL(F)
A ← {} if BCP(F,A) = conflict then return false
level ← 0 while hasUnassignedVars(F)
    level ← level + 1
    A ← A ∪ { Decide(F,A) }
while BCP(F,A) = conflict
    ⟨b, c⟩ ← AnalyzeConflict()
    F ← F ∪ {c}
    if b < 0 then return false
    else Backtrack(F,A, b)
    level ← b
return true
```

F = { c1, c2, c3, c4, c5, c6, ..., c9 }
c1 : ¬x1 ∨ x2 ∨ ¬x4
c2 : ¬x1 ∨ ¬x2 ∨ x3
c3 : ¬x3 ∨ ¬x4
c4 : x4 ∨ x5 ∨ x6
c5 : ¬x5 ∨ x7
c6 : ¬x6 ∨ x7 ∨ ¬x8
... ...

x1@1
CDCL by example

CDCL(F)
A ← {}
if BCP(F, A) = conflict then return false
level ← 0
while hasUnassignedVars(F)
    level ← level + 1
    A ← A ∪ { DECIDE(F, A) }
while BCP(F, A) = conflict
    ⟨b, c⟩ ← ANALYZECONFLICT()
    F ← F ∪ {c}
    if b < 0 then return false
    else BACKTRACK(F, A, b)
    level ← b
return true

F = { c₁, c₂, c₃, c₄, c₅, c₆, ..., c₉ }
c₁ : ¬x₁ ∨ x₂ ∨ ¬x₄
¬x₁ ∨ x₂ ∨ x₃
¬x₃ ∨ x₄
c₄ : x₄ ∨ x₅ ∨ x₆
c₅ : ¬x₅ ∨ x₇
c₆ : ¬x₆ ∨ x₇ ∨ ¬x₈

...
CDCL by example

CDCL(F)
A ← {}  
if BCP(F,A) = conflict then return false  
level ← 0  
while hasUnassignedVars(F)  
    level ← level + 1  
    A ← A ∪ { DECIDE(F,A) }  
while BCP(F,A) = conflict  
    ⟨b, c⟩ ← ANALYZECONFLICT()  
    F ← F ∪ {c}  
    if b < 0 then return false  
    else BACKTRACK(F,A, b)  
        level ← b  
return true
CDCL by example

CDCL(F)
A ← {} if BCP(F,A) = conflict then return false level ← 0 while hasUnassignedVars(F)
level ← level + 1 A ← A ∪ { DECIDE(F,A) }
while BCP(F,A) = conflict
⟨b, c⟩ ← ANALYZECONFLICT() F ← F ∪ {c}
if b < 0 then return false else BACKTRACK(F,A, b)
level ← b return true

F = { c₁, c₂, c₃, c₄, c₅, c₆, ..., c₉ }
c₁: ┐x₁ ∨ x₂ ∨ ┐x₄
c₂: ┐x₁ ∨ ┐x₂ ∨ x₃
c₃: ┐x₃ ∨ ┐x₄
c₄: x₄ ∨ x₅ ∨ x₆
c₅: ┐x₅ ∨ x₇
c₆: ┐x₆ ∨ x₇ ∨ ┐x₈
...
...

x₈@2
x₁@1
x₇@3
¬x₅@3
CDCL by example

CDCL(F)
A ← {}  
if BCP(F,A) = conflict then return false
level ← 0  
while hasUnassignedVars(F)
    level ← level + 1  
    A ← A ∪ { DECIDE(F,A) }
while BCP(F,A) = conflict
    ⟨b, c⟩ ← ANALYZECONFLICT()
    F ← F ∪ {c}
if b < 0 then return false
else BACKTRACK(F,A, b)
    level ← b
return true


F = { c_1, c_2, c_3, c_4, c_5, c_6, \ldots, c_9 }
c_1: \neg x_1 \lor x_2 \lor \neg x_4
c_2: \neg x_1 \lor \neg x_2 \lor x_3
c_3: \neg x_3 \lor x_4
c_4: x_4 \lor x_5 \lor x_6
c_5: \neg x_5 \lor x_7
c_6: \neg x_6 \lor x_7 \lor \neg x_8

\ldots
\ldots
CDCL by example

CDCL(F)
A ← {}
if BCP(F,A) = conflict then return false
level ← 0
while hasUnassignedVars(F)
    level ← level + 1
    A ← A ∪ { DECIDE(F,A) }
while BCP(F,A) = conflict
    ⟨b, c⟩ ← ANALYZECONFLICT()
    F ← F ∪ {c}
    if b < 0 then return false
    else BACKTRACK(F, A, b)
    level ← b
return true

F = { c₁, c₂, c₃, c₄, c₅, c₆, …, c₉ }
c₁: ¬x₁ ∨ x₂ ∨ ¬x₄

...
CDCL by example

CDCL(F)
A ← {} if BCP(F,A) = conflict then return false
level ← 0
while hasUnassignedVars(F)
  level ← level + 1
  A ← A ∪ { DECIDE(F,A) }
while BCP(F,A) = conflict
  ⟨b, c⟩ ← ANALYZECONFlict()
  F ← F ∪ {c}
  if b < 0 then return false
  else BACKTRACK(F,A, b)
    level ← b
return true

F = { c1, c2, c3, c4, c5, c6, ..., c9 }
c1 : ¬x1 ∨ x2 ∨ ¬x4
[c1 : ¬x1 ∨ ¬x2 ∨ x3  (color)]
c2 : ¬x1 ∨ ¬x2 ∨ x3
[c3 : ¬x3 ∨ ¬x4  (color)]
c4 : x4 ∨ x5 ∨ x6
[c4 : x4 ∨ x5 ∨ x6  (color)]
c5 : ¬x5 ∨ x7
[c5 : ¬x5 ∨ x7  (color)]
c6 : ¬x6 ∨ x7 ∨ ¬x8
[c6 : ¬x6 ∨ x7 ∨ ¬x8  (color)]
...
CDCL by example

CDCL(F)
A ← {} if $\text{BCP}(F,A) = \text{conflict}$ then return false
level ← 0
while hasUnassignedVars(F)
level ← level + 1
A ← A ∪ \{ \text{DECIDE}(F,A) \}
while $\text{BCP}(F,A) = \text{conflict}$
⟨b, c⟩ ← $\text{ANALYZECONFLICT}()$
F ← F ∪ \{c\}
if b < 0 then return false
else $\text{BACKTRACK}(F,A,b)$
level ← b
return true

F = \{ c_1, c_2, c_3, c_4, c_5, c_6, \ldots, c_9 \}
c_1 : \neg x_1 \lor x_2 \lor \neg x_4
c_2 : \neg x_1 \lor \neg x_2 \lor x_3
c_3 : \neg x_3 \lor \neg x_4
c_4 : x_4 \lor x_5 \lor x_6
c_5 : \neg x_5 \lor x_7
c_6 : \neg x_6 \lor x_7 \lor \neg x_8

...
CDCL by example

CDCL(F)
A ← \{\}
if BCP(F, A) = conflict then return false
level ← 0
while hasUnassignedVars(F)
  level ← level + 1
  A ← A ∪ \{DECIDE(F, A)\}
while BCP(F, A) = conflict
  \langle b, c \rangle ← ANALYZECONFLICT()
  F ← F ∪ \{c\}
  if b < 0 then return false
  else BACKTRACK(F, A, b)
  level ← b
return true

F = \{c_1, c_2, c_3, c_4, c_5, c_6, \ldots, c_9\}
c_1 : \overline{x_1} \lor x_2 \lor \overline{x_4}
c_2 : \overline{x_1} \lor \overline{x_2} \lor x_3
c_3 : \overline{x_3} \lor \overline{x_4}
c_4 : x_4 \lor x_5 \lor x_6
...
CDCL by example

CDCL(F)
A ← {}
if BCP(F,A) = conflict then return false
level ← 0
while hasUnassignedVars(F)
    level ← level + 1
    A ← A ∪ { Decide(F,A) }
while BCP(F,A) = conflict
    ⟨b, c⟩ ← AnalyzeConflict()
    F ← F ∪ {c}
    if b < 0 then return false
    else Backtrack(F,A,b)
return true

F = { c1, c2, c3, c4, c5, c6, …, c9 }
c1 : ¬x1 ∨ x2 ∨ x4
c2 : ¬x1 ∨ ¬x2 ∨ x3
c3 : ¬x3 ∨ ¬x4
c4 : x4 ∨ x5 ∨ x6
c5 : ¬x5 ∨ x7
c6 : ¬x6 ∨ x7 ∨ ¬x8
CDCL by example

CDCL(F)
A ← {}  
if BCP(F, A) = conflict then return false
level ← 0
while hasUnassignedVars(F)
    level ← level + 1
    A ← A ∪ { DECIDE(F, A) }
while BCP(F, A) = conflict
    ⟨b, c⟩ ← ANALYZECONFLICT()
    F ← F ∪ {c}
    if b < 0 then return false
    else BACKTRACK(F, A, b)
    level ← b
return true

F = { c₁, c₂, c₃, c₄, c₅, c₆, ..., c₉, c }  
c₁ : ¬x₁ ∨ x₂ ∨ ¬x₄  
c₂ : ¬x₁ ∨ ¬x₂ ∨ x₃  
c₃ : ¬x₃ ∨ x₄  
c₄ : x₄ ∨ x₅ ∨ x₆  
c₅ : ¬x₅ ∨ x₇  
c₆ : ¬x₆ ∨ x₇ ∨ ¬x₈  
...  
c : ¬x₁ ∨ ¬x₄

 ⟨1, ¬x₁ ∨ x₄⟩
CDCL by example

CDCL(F)
A ← {} if BCP(F,A) = conflict then return false
level ← 0
while hasUnassignedVars(F)
   level ← level + 1
   A ← A ∪ { DECIDE(F,A) }
while BCP(F,A) = conflict
   ⟨b, c⟩ ← ANALYZE_CONFLICT()
   F ← F ∪ {c}
   if b < 0 then return false
else BACKTRACK(F,A, b)
   level ← b
return true

F = { c₁, c₂, c₃, c₄, c₅, c₆, ..., c₉, c }
c₁: ¬x₁ ∨ x₂ ∨ ¬x₄

c₂: ¬x₁ ∨ ¬x₂ ∨ x₃

c₃: ¬x₃ ∨ ¬x₄

c₄: x₄ ∨ x₅ ∨ x₆

c₅: ¬x₅ ∨ x₇

c₆: ¬x₆ ∨ x₇ ∨ ¬x₈
...
c₉: ¬x₁ ∨ ¬x₄
CDCL by example

CDCL(F)
A ← {} if BCP(F,A) = conflict then return false
level ← 0
while hasUnassignedVars(F)
  level ← level + 1
  A ← A ∪ { DECIDE(F,A) }
while BCP(F,A) = conflict
  ⟨b, c⟩ ← ANALYZECONFLICT()
  F ← F ∪ {c} if b < 0 then return false else BACKTRACK(F,A, b)
  level ← b
return true

F = { c₁, c₂, c₃, c₄, c₅, c₆, …, c₉, c }
c₁ : ¬x₁ ∨ x₂ ∨ x₄
c₂ : ¬x₁ ∨ ¬x₂ ∨ x₃
c₃ : ¬x₃ ∨ x₄
c₄ : x₄ ∨ x₅ ∨ x₆
c₅ : x₅ ∨ x₇
c₆ : x₆ ∨ x₇ ∨ ¬x₈
…
c₀ : ¬x₁ ∨ ¬x₄

Conflict clause is unit after backtracking!
CDCL in depth

CDCL(F)
A ← {}
if BCP(F, A) = conflict then return false
level ← 0
while hasUnassignedVars(F)
    level ← level + 1
    A ← A ∪ { DECIDE(F, A) }
while BCP(F, A) = conflict
    ⟨b, c⟩ ← ANALYZECONFLICT()
    F ← F ∪ {c}
    if b < 0 then return false
    else BACKTRACK(F, A, b)
        level ← b
return true

• Definitions
• ANALYZECONFLICT
• DECIDE heuristics
• Implementation
Basic definitions

Under a given partial assignment (PA), a variable may be

- **assigned** (true/false literal)
- **unassigned**.

\[ F = \{ c_1, c_2, c_3, c_4, c_5, c_6, \ldots, c_9 \} \]

- \( c_1 \): \( \neg x_1 \lor x_2 \lor \neg x_4 \)
- \( c_2 \): \( \neg x_1 \lor \neg x_2 \lor x_3 \)
- \( \ldots \)
- \( c_8 \): \( x_9 \lor \neg x_2 \)
- \( c_9 \): \( x_9 \lor x_{10} \lor x_3 \)

True literals highlighted in green; false literals highlighted in red.
Basic definitions

Under a given partial assignment (PA), a variable may be
• **assigned** (true/false literal)
• **unassigned**.

A clause may be
• **satisfied** \((\geq 1\ \text{true literal})\)
• **unsatisfied** (all false literals)
• **unit** (one unassigned literal, rest false)
• **unresolved** (otherwise)

\(F = \{ c_1, c_2, c_3, c_4, c_5, c_6, \ldots, c_9 \} \)

\(c_1 : \neg x_1 \lor x_2 \lor x_4 \)
\(c_2 : \neg x_1 \lor \neg x_2 \lor x_3 \)
\(\ldots\)
\(c_8 : x_9 \lor \neg x_2 \)
\(c_9 : x_9 \lor x_10 \lor x_3 \)

True literals highlighted in green; false literals highlighted in red.
An implication graph $G = (V, E)$ is a DAG that records the history of decisions and the resulting deductions derived with BCP.
Implication graph

An implication graph $G = (V, E)$ is a DAG that records the history of decisions and the resulting deductions derived with BCP.

- $v \in V$ is a literal (or $\kappa$) and the decision level at which it entered the current PA.
Implication graph

An implication graph $G = (V, E)$ is a DAG that records the history of decisions and the resulting deductions derived with BCP.

- $v \in V$ is a literal (or $\kappa$) and the decision level at which it entered the current PA.
- $\langle v, w \rangle \in E$ iff $v \neq w$, $\neg v \in \text{antecedent}(w)$, and $\langle v, w \rangle$ is labeled with $\text{antecedent}(w)$.
An implication graph $G = (V, E)$ is a DAG that records the history of decisions and the resulting deductions derived with BCP.

- $v \in V$ is a literal (or $\kappa$) and the decision level at which it entered the current PA.
- $\langle v, w \rangle \in E$ iff $v \neq w$, $\neg v \in \text{antecedent}(w)$, and $\langle v, w \rangle$ is labeled with $\text{antecedent}(w)$.

A unit clause $c$ is the antecedent of its sole unassigned literal.
Implication graph: a quick exercise

What clauses gave rise to this implication graph?
Implication graph: a quick exercise

What clauses gave rise to this implication graph?

\[ C_1 : \neg x_1 \lor x_2 \]
Implication graph: a quick exercise

What clauses gave rise to this implication graph?

\[ C_1 : \lnot x_1 \lor x_2 \]
\[ C_2 : \lnot x_1 \lor x_3 \lor x_5 \]
Implication graph: a quick exercise

What clauses gave rise to this implication graph?

\[ c_1 : \neg x_1 \lor x_2 \]
\[ c_2 : \neg x_1 \lor x_3 \lor x_5 \]
\[ c_3 : \neg x_2 \lor x_4 \]
Implication graph: a quick exercise

What clauses gave rise to this implication graph?

\[ C_1 : \neg x_1 \lor x_2 \]
\[ C_2 : \neg x_1 \lor x_3 \lor x_5 \]
\[ C_3 : \neg x_2 \lor x_4 \]
\[ C_4 : \neg x_3 \lor \neg x_4 \]
Implication graph: an even quicker exercise

What clauses gave rise to this implication graph?
Implication graph: an even quicker exercise

What clauses gave rise to this implication graph?

\( c_1 : \neg x_1 \lor x_2 \)
Implication graph: an even quicker exercise

What clauses gave rise to this implication graph?

\[ \neg \chi_1 \lor \chi_2 \]
\[ \neg \chi_1 \lor \chi_3 \lor \chi_5 \]
Implication graph: an even quicker exercise

What clauses gave rise to this implication graph?

\[
\begin{align*}
\text{c}_1 : & \quad \neg x_1 \lor x_2 \\
\text{c}_2 : & \quad \neg x_1 \lor x_3 \lor x_5 \\
\text{c}_3 : & \quad \neg x_2 \lor x_4
\end{align*}
\]
Implication graph: an even quicker exercise

What clauses gave rise to this implication graph?

\[ \begin{align*}
C_1 & : \neg x_1 \lor x_2 \\
C_2 & : \neg x_1 \lor x_3 \lor x_5 \\
C_3 & : \neg x_2 \lor x_4 \\
C_4 & : \neg x_3 \lor \neg x_4
\end{align*} \]
Implication graph: an even quicker exercise

What clauses gave rise to this implication graph?

\[ \neg x_1 \lor x_2 \]
\[ \neg x_1 \lor x_3 \lor x_5 \]
\[ \neg x_2 \lor x_4 \]
\[ \neg x_3 \lor \neg x_4 \]
\[ \neg x_5 \]

Assignments at ground (0) level are implied by unary clauses.
Using an implication graph to analyze a conflict

CDCL(F)
A ← {} if BCP(F, A) = conflict then return false
level ← 0
while hasUnassignedVars(F)
   level ← level + 1
   A ← A ∪ { DECIDE(F, A) }
while BCP(F, A) = conflict
   ⟨b, c⟩ ← AnalyzeConflict()
   F ← F ∪ {c}
   if b < 0 then return false
   else BACKTRACK(F, A, b)
      level ← b
return true

A conflict clause is implied by F and it blocks partial assignments (PAs) that lead to the current conflict.
Using an implication graph to analyze a conflict

A conflict clause is implied by F and it blocks partial assignments (PAs) that lead to the current conflict.

Every cut that separates sources from the sink defines a valid conflict clause.

CDCL(F)
A ← {}  
if $\text{BCP}(F,A) = \text{conflict}$ then return false
level ← 0
while hasUnassignedVars(F)
  level ← level + 1
  A ← A ∪ { $\text{DECIDE}(F,A)$ }
while $\text{BCP}(F,A) = \text{conflict}$
  $\langle b, c \rangle$ ← $\text{ANALYZECONFLICT}(\cdot)$
  F ← F ∪ {c}
  if b < 0 then return false
  else $\text{BACKTRACK}(F,A,b)$
return true
Using an implication graph to analyze a conflict

A conflict clause is implied by F and it blocks partial assignments (PAs) that lead to the current conflict.

Every cut that separates sources from the sink defines a valid conflict clause.
Using an implication graph to analyze a conflict

\[
\text{CDCL}(F) \\
\text{A} \leftarrow \{\} \\
\text{if } \text{BCP}(F,A) = \text{conflict then return } \text{false} \\
\text{level} \leftarrow 0 \\
\text{while } \text{hasUnassignedVars}(F) \\
\quad \text{level} \leftarrow \text{level} + 1 \\
\quad \text{A} \leftarrow \text{A} \cup \{ \text{DECIDE}(F,A) \} \\
\text{while } \text{BCP}(F,A) = \text{conflict} \\
\quad \langle b, c \rangle \leftarrow \text{ANALYZECONFLICT()} \\
\quad F \leftarrow F \cup \{c\} \\
\quad \text{if } b < 0 \text{ then return } \text{false} \\
\quad \text{else } \text{BACKTRACK}(F,A,b) \\
\quad \text{level} \leftarrow b \\
\text{return } \text{true}
\]
A unique implication point (UIP) is any node in the implication graph other than the conflict that is on all paths from the current decision literal (lit@d) to the conflict (κ@d).

A first UIP is the UIP that is closest to the conflict.
A unique implication point (UIP) is any node in the implication graph other than the conflict that is on all paths from the current decision literal (lit@d) to the conflict (κ@d).

A first UIP is the UIP that is closest to the conflict.
`ANALYZECONFLICT`: computing the conflict clause

```
ANALYZECONFLICT()
    d ← level(conflict)
    if d = 0 then return -1
    c ← antecedent(conflict)
    repeat
        t ← lastAssignedLitAtLevel(c, d)
        v ← varOfLit(t)
        a ← antecedent(t)
        c ← resolve(ante, c, v)
    until oneLitAtLevel(c, d)
    b ← ...
    return ⟨b, c⟩
```
\textbf{ANALYZECONFLICT:} computing the conflict clause

\begin{verbatim}
ANALYZECONFLICT()
  d ← level(conflict)
  if d = 0 then return -1
  c ← antecedent(conflict)
  repeat
    t ← lastAssignedLitAtLevel(c, d)
    v ← varOfLit(t)
    a ← antecedent(t)
    c ← resolve(ante, c, v)
  until oneLitAtLevel(c, d)
  b ← ...
  return ⟨b, c⟩
\end{verbatim}

\textbf{Binary resolution rule}

\[(a_1 \lor \ldots \lor a_n \lor \beta) \land (b_1 \lor \ldots \lor b_m \lor \neg \beta) \implies (a_1 \lor \ldots \lor a_n \lor b_1 \lor \ldots \lor b_m)\]
**ANALYZECONFLICT:** computing the conflict clause

```
ANALYZECONFLICT()
  d ← level(conflict)
  if d = 0 then return -1
  c ← antecedent(conflict)
  repeat
    t ← lastAssignedLitAtLevel(c, d)
    v ← varOfLit(t)
    a ← antecedent(t)
    c ← resolve(ante, c, v)
  until oneLitAtLevel(c, d)
  b ← ...
  return ⟨b, c⟩
```

**Binary resolution rule**

\[(a_1 \lor \ldots \lor a_n \lor \beta) \quad (b_1 \lor \ldots \lor b_m \lor \neg \beta) \]

\[
(a_1 \lor \ldots \lor a_n \lor b_1 \lor \ldots \lor b_m)
\]
**ANALYZECONFLICT:** computing the conflict clause

```
ANALYZECONFLICT()
  d ← level(conflict)
  if d = 0 then return -1
  c ← antecedent(conflict)
  repeat
    t ← lastAssignedLitAtLevel(c, d)
    v ← varOfLit(t)
    a ← antecedent(t)
    c ← resolve(ante, c, v)
  until oneLitAtLevel(c, d)
  b ← ...
  return ⟨b, c⟩
```

**Binary resolution rule**

\[(a_1 \lor \ldots \lor a_n \lor \beta) \land (b_1 \lor \ldots \lor b_m \lor \neg \beta)\]  

\[(a_1 \lor \ldots \lor a_n \lor b_1 \lor \ldots \lor b_m)\]

**Example:**
- \(c = c_2\), \(t = x_2\), \(v = x_2\), \(a = c_1\)
**ANALYZECONFLICT**: computing the conflict clause

**Binary resolution rule**

\[(a_1 \lor \ldots \lor a_n \lor \beta) \land (b_1 \lor \ldots \lor b_m \lor \neg \beta) \Rightarrow (a_1 \lor \ldots \lor a_n \lor b_1 \lor \ldots \lor b_m)\]

**Example:**

- \(c = c_2, t = x_2, v = x_2, a = c_1\)
- \(c = \neg x_1 \lor x_3 \lor \neg x_4, t = x_3, v = x_3, a = c_3\)
**ANALYZECONFLICT**: computing the conflict clause

**ANALYZECONFLICT()**

- `d ← level(conflict)`
- **if** `d = 0` **then** return `-1`
- `c ← antecedent(conflict)`
- **repeat**
  - `t ← lastAssignedLitAtLevel(c, d)`
  - `v ← varOfLit(t)`
  - `a ← antecedent(t)`
  - `c ← resolve(ante, c, v)`
- **until** oneLitAtLevel(c, d)
- `b ← ...`
- **return** ⟨`b, c`⟩

**Example:**
- `c = c_2, t = x_2, v = x_2, a = c_1`
- `c = \neg x_1 \lor x_3 \lor \neg x_4, t = x_3, v = x_3, a = c_3`
- `c = \neg x_1 \lor \neg x_4`, done!

**Binary resolution rule**

\[(a_1 \lor \ldots \lor a_n \lor \beta) (b_1 \lor \ldots \lor b_m \lor \neg \beta) \Rightarrow (a_1 \lor \ldots \lor a_n \lor b_1 \lor \ldots \lor b_m)\]
**ANALYZECONFLICT:** computing backtracking level

```
ANALYZECONFLICT()
  d ← level(conflict)
  if d = 0 then return -1
  c ← antecedent(conflict)
  repeat
    t ← lastAssignedLit(c)
    v ← varOfLit(t)
    a ← antecedent(t)
    c ← resolve(ante, c, v)
  until oneLitAtLevel(c, d)
  b ← assertingLevel(c)
  return ⟨b, c⟩
```
**ANALYZECONFLICT**: computing backtracking level

```
ANALYZECONFLICT()
  d ← level(conflict)
  if d = 0 then return -1
  c ← antecedent(conflict)
  repeat
    t ← lastAssignedLit(c)
    v ← varOfLit(t)
    a ← antecedent(t)
    c ← resolve(ante, c, v)
  until oneLitAtLevel(c, d)
  b ← assertingLevel(c)
  return ⟨b, c⟩
```

Second highest decision level for any literal in c, unless c is unary. In that case, its asserting level is zero.
**ANALYZECONFLICT**: computing backtracking level

```
ANALYZECONFLICT()
  d ← level(conflict)
  if d = 0 then return -1
  c ← antecedent(conflict)
  repeat
    t ← lastAssignedLit(c)
    v ← varOfLit(t)
    a ← antecedent(t)
    c ← resolve(ante, c, v)
  until oneLitAtLevel(c, d)
  b ← assertingLevel(c)
  return ⟨b, c⟩
```

By construction, c is unit at b (since it has only one literal at the current level d).

Second highest decision level for any literal in c, unless c is unary. In that case, its asserting level is zero.
CDCL(F)
A ← {}
if BCP(F,A) = conflict then return false
level ← 0
while hasUnassignedVars(F)
  level ← level + 1
  A ← A ∪ {DECIDE(F,A)}
while BCP(F,A) = conflict
  ⟨b, c⟩ ← ANALYZECONFLICT()
  F ← F ∪ {c}
  if b < 0 then return false
  else BACKTRACK(F,A,b)
     level ← b
return true

Example heuristics:
• Dynamic Largest Individual Sum (DLIS)
• Variable State Independent Decaying Sum (VSIDS)
Decision heuristics: DLIS

CDCL(F)
A ← {}
if \( BCP(F, A) = \text{conflict} \) then return \( false \)
level ← 0
while hasUnassignedVars(F)
    level ← level + 1
    A ← A ∪ \{ DECIDE(F,A) \}
while \( BCP(F, A) = \text{conflict} \)
    \( \langle b, c \rangle \) ← ANALYZECONFLICT()
    F ← F ∪ \{c\}
    if \( b < 0 \) then return \( false \)
    else BACKTRACK(F, A, b)
        level ← b
return true

• Choose the literal that satisfies the most unresolved clauses.
• Simple and intuitive.
• But expensive: complexity of making a decision proportional to the number of clauses.
Decision heuristics: VSIDS (zChaff)

CDCL(F)
A ← {} if BCP(F,A) = conflict then return false
level ← 0
while hasUnassignedVars(F)
  level ← level + 1
  A ← A ∪ { DECIDE(F,A) }
while BCP(F,A) = conflict
  ⟨b, c⟩ ← ANALYZECONFLICT()
  F ← F ∪ {c}
  if b < 0 then return false
  else BACKTRACK(F,A, b)
  level ← b
return true

• Count the number of *all* clauses in which a literal appears, and periodically divide all scores by a constant (e.g., 2).
• Variables involved in more recent conflicts get higher scores.
• Constant decision time when literals kept in a sorted list.
CDCL(F)
A ← {}
if \textsc{BCP}(F, A) = \text{conflict} then return false
level ← 0
\textbf{while} hasUnassignedVars(F)
    level ← level + 1
    A ← A \cup \{ \textsc{Decide}(F, A) \}
\textbf{while} \textsc{BCP}(F, A) = \text{conflict}
    \langle b, c \rangle ← \textsc{AnalyzeConflict}()
    F ← F \cup \{c\}
    if b < 0 then return false
    else \textsc{Backtrack}(F, A, b)
        level ← b
\textbf{return} true

Engineering matters (a lot)

Solvers spend most of their time in BCP, so this must be efficient. Naive implementation won’t work on large problems.
Engineering matters (a lot)

Solvers spend most of their time in BCP, so this must be efficient. Naive implementation won’t work on large problems.

Most solvers heuristically discard conflict clauses that are old, long, irrelevant, etc. (Why won’t this cause the solver to run forever?)
BCP with watched literals (zChaff)

CDCL(F)
A ← {}  
if BCP(F,A) = conflict then return false  
level ← 0  
while hasUnassignedVars(F)  
  level ← level + 1  
  A ← A ∪ { DECIDE(F,A) }  
while BCP(F,A) = conflict  
  ⟨b, c⟩ ← ANALYZECONFLICT()  
  F ← F ∪ {c}  
  if b < 0 then return false  
  else BACKTRACK(F,A, b)  
  level ← b  
return true

Based on the observation that a clause can’t imply a new assignment if it has more than 2 unassigned literals left.

So, pick two unassigned literals per clause to watch.

If a watched literal is assigned, pick another unassigned literal to watch in its place.

If there is only one unassigned literal, it is implied by BCP.
Summary

Today

• The CDCL algorithm extends DPLL with
  • Non-chronological backtracking
  • Learning
  • Decision heuristics
  • Engineering matters

Next lecture

• Practical applications of SAT solving