Solvers for the Problem of Boolean Satisfiability (SAT)

Will Klieber
15-414
Aug 31, 2011
Why study SAT solvers?

- Many problems reduce to SAT.
  - Formal verification
  - CAD, VLSI
  - Optimization
  - AI, planning, automated deduction
- Modern SAT solvers are often fast.
- Other solvers (QBF, SMT, etc.) borrow techniques from SAT solvers.
- SAT solvers and related solvers are still active areas of research.
Negation-Normal Form (NNF)

- A formula is in negation-normal form iff:
  - all negations are directly in front of variables, and
  - the only logical connectives are: “∧”, “∨”, “¬”.

- A **literal** is a variable or its negation.

- Convert to NNF by pushing negations inward:
  \[
  \neg(P \land Q) \iff (\neg P \lor \neg Q) \quad \text{and} \quad \neg(P \lor Q) \iff (\neg P \land \neg Q)
  \]
  (De Morgan’s Laws)
Disjunctive Normal Form (DNF)

- Recall: A *literal* is a variable or its negation.
- A formula is in DNF iff:
  - it is a disjunction of conjunctions of literals.

\[
\begin{align*}
  \left( \ell_{11} \land \ell_{12} \land \ell_{13} \right) & \lor \\
  \left( \ell_{21} \land \ell_{22} \land \ell_{23} \right) & \lor \\
  \left( \ell_{31} \land \ell_{32} \land \ell_{33} \right)
\end{align*}
\]

- conjunction 1  conjunction 2  conjunction 3

- Every formula in DNF is also in NNF.
- A simple (but inefficient) way convert to DNF:
  - Make a truth table for the formula \( \varphi \).
  - Each row where \( \varphi \) is true corresponds to a conjunct.
Conjunctive Normal Form (CNF)

- A formula is in CNF iff:
  - it is a conjunction of disjunctions of literals.
    
    \[(\ell_{11} \lor \ell_{12} \lor \ell_{13}) \land (\ell_{21} \lor \ell_{22} \lor \ell_{23}) \land (\ell_{31} \lor \ell_{32} \lor \ell_{33})\]
    
    clause 1  clause 2  clause 3

- Modern SAT solvers use CNF.
- Any formula can be converted to CNF.
  - Equivalent CNF can be exponentially larger.
- Equi-satisfiable CNF (Tseitin encoding):
  - Only linearly larger than original formula.
Tseitin transformation to CNF

- Introduce new variables to represent subformulas.
  
  Original: $\exists \vec{x}. \phi(\vec{x})$
  
  Transformed: $\exists \vec{x}. \exists \vec{g}. \psi(\vec{x}, \vec{g})$

- E.g., to convert $(A \lor (B \land C))$:
  
  Replace $(B \land C)$ with a new variable $g_1$.
  
  Add clauses to equate $g_1$ with $(B \land C)$.
  
  $\begin{align*}
  (A \lor g_1) & \land (B \lor \neg g_1) \land (C \lor \neg g_1) \land (\neg B \lor \neg C \lor g_1) \\
  (\neg B & \rightarrow \neg g_1) \land (\neg C \rightarrow \neg g_1) \land ((B \land C) \rightarrow g_1)
  \end{align*}$

  - Gives value of $g_1$ for all 4 possible assignments to $\{B, C\}$. 
Tseitin transformation to CNF

Convert \((A \lor (B \land C))\) to CNF by introducing new variable \(g_1\) for \((B \land C)\).

\[
\begin{align*}
(A \lor g_1) \land (\neg g_1 \lor B) \land (\neg g_1 \lor C) \land (\neg B \lor \neg C \lor g_1) \\
(g_1 \rightarrow B) \quad (g_1 \rightarrow C) \quad ((B \land C) \rightarrow g_1) \\
(g_1 \rightarrow (B \land C)) \land ((B \land C) \rightarrow g_1) \\
(g_1 \iff (B \land C))
\end{align*}
\]
SAT Solvers -- Representation

- A CNF formula is represented by a set of clauses.
  - Empty set represents a true formula.
- A clause is represented by a set of literals
  - Empty set represents a false clause.
- A variable is represented by a positive integer.
- The logical negation of a variable is represented by the arithmetic negation of its number.
- E.g., \(((x_1 \lor x_2) \land (\neg x_1 \lor \neg x_2))\) is represented by \(\{\{1, 2\}, \{-1, -2\}\}\)
Naïve Approach

- SAT problem: Given a boolean formula \( \varphi \), does there exist an assignment that satisfies \( \varphi \)?

- Naïve approach: Search all assignments!
  - \( n \) variables \( \rightarrow 2^n \) possible assignments
  - Explosion!

- SAT is NP-complete:
  - Worst case is likely \( O(2^n) \), unless P=NP.
  - But for many cases that arise in practice, we can do much better.
Unit Propagation

- Davis-Putnam-Logemann-Loveland (DPLL)
- Unit Clause: Clause with exactly one literal.
- Algorithm:
  - If a clause has exactly one literal, then assign it true.
  - Repeat until there are no more unit clauses.
- Example:
  - \(((x_1 \lor x_2) \land (\neg x_1 \lor \neg x_2) \land (x_1))\)
  - \(((T \lor x_2) \land (F \lor \neg x_2) \land (T))\)
  - \(((T) \land (\neg x_2))\)
  - T
Helper function

def AssignLit(ClauseList, lit):
    ClauseList = deepcopy(ClauseList)
    for clause in copy(ClauseList):
        if lit in clause: ClauseList.remove(clause)
        if -lit in clause: clause.remove(-lit)
    return ClauseList

>>> AssignLit([[1, 2, -3], [-1, -2, 4], [3, 4]], 1)
[[2, -3], [3, 4]]

>>> AssignLit([[1, 2, -3], [-1, -2, 4], [3, 4]], -1)
[[2, -3], [3, 4]]

Assumption: No clause contains both a variable and its negation.
Naïve Solver

```python
def AssignLit(ClauseList, lit):
    ClauseList = deepcopy(ClauseList)
    for clause in copy(ClauseList):
        if lit in clause: ClauseList.remove(clause)
        if -lit in clause: clause.remove(-lit)
    return ClauseList

def IsSatisfiable(ClauseList):
    # Test if no unsatisfied clauses remain
    if len(ClauseList) == 0: return True

    # Test for presence of empty clause
    if [] in ClauseList: return False

    # Split on an arbitrarily decided literal
    DecLit = ClauseList[0][0]
    return (IsSatisfiable(AssignLit(ClauseList, DecLit)) or
            IsSatisfiable(AssignLit(ClauseList, -DecLit)))
```
def IsSatisfiable(ClauseList):
    # Unit propagation
    repeat until fixed point:
        for each unit clause UC in ClauseList:
            ForcedLit = UC[0]
            ClauseList = AssignLit(ClauseList, ForcedLit)

    # Test if no unsatisfied clauses remain
    if len(ClauseList) == 0: return True

    # Test for presence of empty clause
    if [] in ClauseList: return False

    # Split on an arbitrarily decided literal
    DecLit = (choose a variable occurring in ClauseList)
    return (IsSatisfiable(AssignLit(ClauseList, DecLit)) or
            IsSatisfiable(AssignLit(ClauseList, -DecLit)))
GRASP: an efficient SAT solver

Original Slides by Pankaj Chauhan
Modified by Will Klieber

Please interrupt me if anything is not clear!
Terminology

- **CNF formula** \( \varphi \)
  - \( x_1, \ldots, x_n : n \) variables
  - \( \omega_1, \ldots, \omega_m : m \) clauses

- **Assignment** \( A \)
  - Set of (variable, value) pairs.
  - Notation: \( \{(x_1,1), (x_2,0)\}, \{x_1:1, x_2:0\}, \{x_1=1, x_2=0\}, \{x_1, \neg x_2\} \)
  - \( |A| < n \rightarrow \text{partial assignment} \) \( \{x_1=0, x_2=1, x_4=1\} \)
  - \( |A| = n \rightarrow \text{complete assignment} \) \( \{x_1=0, x_2=1, x_3=0, x_4=1\} \)
  - \( \varphi|_A = 0 \rightarrow \text{falsifying assignment} \) \( \{x_1=1, x_4=1\} \)
  - \( \varphi|_A = 1 \rightarrow \text{satisfying assignment} \) \( \{x_1=0, x_2=1, x_4=1\} \)
  - \( \varphi|_A = X \rightarrow \text{unresolved assignment} \) \( \{x_1=0, x_2=0, x_4=1\} \)

\[ \varphi = \omega_1 \land \omega_2 \land \omega_3 \]

\[ \omega_1 = (x_2 \lor x_3) \]

\[ \omega_2 = (\neg x_1 \lor \neg x_4) \]

\[ \omega_3 = (\neg x_2 \lor x_4) \]

\[ A = \{x_1=0, x_2=1, x_3=0, x_4=1\} \]
Terminology

- An assignment partitions the clause database into three classes:
  - Satisfied, falsified, unresolved
- **Free literal**: an unassigned literal
- **Unit clause**: has exactly one free literal
Basic Backtracking Search

- Organize the search in the form of a decision tree.
  - Each node is a decision variable.
  - Outgoing edges: assignment to the decision variable.
  - Depth of node in decision tree is decision level $\delta(x)$.
  - “$x=v @ d$” means variable $x$ is assigned value $v$ at decision level $d$.

```latex
x_1 = 1 @ 1 \\
x_1 = 0 @ 1 \\
x_2 = 1 @ 2 \\
x_2 = 0 @ 2
```
Basic Backtracking Search

1. Make new decision assignments.
2. Infer **implied assignments** by a deduction process (unit propagation).
   - May lead to falsifying clauses, **conflict!**
   - The assignment is called “conflicting assignment”.
3. Conflicting assignments leads to **backtrack**.
Backtracking Search in Action

Example 1

\[ \omega_1 = (x_2 \lor x_3) \]
\[ \omega_2 = (\neg x_1 \lor \neg x_4) \]
\[ \omega_3 = (\neg x_2 \lor x_4) \]

\( x_1 = 0 \)
\( x_2 = 0 \)
\( \Rightarrow x_3 = 1 \)

\( \{(x_1, 0), (x_2, 0), (x_3, 1)\} \)

No backtrack in this example!
Backtracking Search in Action

Example 2

\[ \omega_1 = (x_2 \lor x_3) \]
\[ \omega_2 = (\neg x_1 \lor \neg x_4) \]
\[ \omega_3 = (\neg x_2 \lor x_4) \]

\[
x_1 = 1 \Rightarrow x_4 = 0 \Rightarrow x_2 = 0 \Rightarrow x_3 = 1
\]

No backtrack in this example!
Backtracking Search in Action

Example 3

\[ \omega_1 = (x_2 \lor x_3) \]
\[ \omega_2 = (\neg x_1 \lor \neg x_4) \]
\[ \omega_3 = (\neg x_2 \lor x_4) \]
\[ \omega_4 = (\neg x_1 \lor x_2 \lor \neg x_3) \]

\[
\begin{align*}
x_1 &= 1 @ 1 \\
  & \Rightarrow x_4 = 0 @ 1 \\
  & \Rightarrow x_2 = 0 @ 1 \\
  & \Rightarrow x_3 = 1 @ 1 \\
\end{align*}
\]

\[
\begin{align*}
x_1 &= 0 @ 1 \\
  x_2 &= 0 @ 2 \Rightarrow x_3 = 1 @ 2 \\
\end{align*}
\]

\{(x_1, 0), (x_2, 0), (x_3, 1)\}
GRASP

- GRASP is Generalized SeaRch Algorithm for the Satisfiability Problem (Silva, Sakallah, ’96).
- Features:
  - Implication graphs for Unit Propagation and conflict analysis.
  - Learning of new clauses.
  - Non-chronological backtracking!
Learning

- GRASP can learn new clauses that are logically implied by the original formula.
- Goal is to allow Unit Prop to deduce more forced literals, pruning the search space.
- Example:
  - $\phi$ contains clauses $(x \lor y \lor z)$ and $(x \lor y \lor \neg z)$.
  - **Resolving** on $z$ yields a new clause $(x \lor y)$.
  - If $y$ is false, then $x$ must be true for $\phi$ to be true.
    - But not discoverable by simple Unit Prop w/o resolvent clause.
  - Clause $(x \lor y)$ allows Unit Prop to force $x=1$ when $y=0$.
- New clauses learned from conflicting assignments.
Resolution

From

\((x_1 \lor \cdots \lor x_n \lor r) \land (\neg r \lor y_1 \lor \cdots \lor y_m)\)

deduce

\((x_1 \lor \cdots \lor x_n \lor y_1 \lor \cdots \lor y_m)\)
Top-level of GRASP-like solver

1. CurAsgn = {};  
2. **while** (true) {  
3.     **while** (value of $\varphi$ under CurAsgn is unknown) {  
4.         DecideLit();  // Add decision literal to CurAsgn.  
5.         Propagate();  // Add forced literals to CurAsgn.  
6.     }  
7.     if (CurAsgn satisifies $\varphi$) {**return** true;}  
8.     Analyze conflict and learn a new clause;  
9.     if (the learned clause is empty) {**return** false;}  
10.    Backtrack();  
11.    Propagate();  // Learned clause will force a literal  
12. }
GRASP Decision Heuristics

- **Procedure** DecideLit()
- Choose the variable that satisfies the most clauses
- Other possibilities exist
GRASP Deduction

- Unit Propagation is a type of Boolean Constraint Propagation (BCP).
- Grasp does Unit Prop using **implication graphs**: E.g., for the clause \( \omega = (x \lor \neg y) \), if \( y = 1 \), then \( x = 1 \) is forced; the antecedent of \( x \) is \{y=1\}.
- If a variable \( x \) is forced by a clause during BCP, then assignment of 0 to all other literals in the clause is called the **antecedent assignment** \( A(x) \).
  - E.g., for \( \omega = (x \lor y \lor \neg z) \), 
    \[
    A(x) = \{y:0, z:1\}, \ A(y) = \{x:0, z:1\}, \ A(z) = \{x:0, y:0\}
    \]
  - Variables directly responsible for forcing the value of \( x \).
  - Antecedent assignment of a decision variable is empty.
Implication Graphs

- Depicts the antecedents of assigned variables.
- A node is an assignment to a variable.
  - (decision or implied)
- Predecessors of $x$ correspond to antecedent $A(x)$.
  - No predecessors for decision assignments!
- For special conflict vertex $\kappa$, antecedent $A(\kappa)$ is assignment to vars in the falsified clause.

```
x_2=1@6  x_{10}=0@3  x_5=1@6  
x_1=1@6  x_3=1@6  x_4=1@6  x_6=1@6
x_9=0@1  x_{11}=0@3
```

conflict
Example Implication Graph

Current truth assignment: \{x_9=0@1, x_{12}=1@2, x_{13}=1@2, x_{10}=0@3, x_{11}=0@3\}

Current decision assignment: \{x_1=1@6\}

\[\omega_1 = (\neg x_1 \lor x_2)\]
\[\omega_2 = (\neg x_1 \lor x_3 \lor x_9)\]
\[\omega_3 = (\neg x_2 \lor \neg x_3 \lor x_4)\]
\[\omega_4 = (\neg x_4 \lor x_5 \lor x_{10})\]
\[\omega_5 = (\neg x_4 \lor x_6 \lor x_{11})\]
\[\omega_6 = (\neg x_5 \lor \neg x_6)\]
\[\omega_7 = (x_1 \lor x_7 \lor \neg x_{12})\]
\[\omega_8 = (x_1 \lor x_8)\]
\[\omega_9 = (\neg x_7 \lor \neg x_8 \lor \neg x_{13})\]
GRASP Conflict Analysis

- After a conflict arises, analyze the implication graph.
- Add new clause that would prevent the occurrence of the same conflict in the future.
  ⇒ Learning
- Determine decision level to backtrack to; this might not be the immediate one.
  ⇒ Non-chronological backtrack
Learning Algorithm

1. Let CA be the assignment of False to all literals in the falsified clause. ("CA" is short for "conflict assignment".)
   - Example: $CA = \{x_5=1@6, \ x_6 = 1@6\}$

2. A literal $l \in CA$ is a unique implication point (UIP) iff every other literal in CA has an earlier decision level than $l$.

3. Loop:
   - Remove the most recently assigned literal from CA and replace it by its antecedent.
   - if (CA is empty or has a UIP): break;

4. Let $\{L_1, \ldots, L_n\} = CA$; learn clause ($\neg L_1 \lor \ldots \lor \neg L_n$).

5. Backtrack to the earliest decision level at which the learned clause will force the UIP to be false.
   - Why is this guaranteed to be possible?
Example Implication Graph

Current truth assignment: \{x_9=0@1, x_{12}=1@2, x_{13}=1@2, x_{10}=0@3, x_{11}=0@3\}

Current decision assignment: \{x_1=1@6\}

\(\omega_1 = (\neg x_1 \lor x_2)\)
\(\omega_2 = (\neg x_1 \lor x_3 \lor x_9)\)
\(\omega_3 = (\neg x_2 \lor \neg x_3 \lor x_4)\)
\(\omega_4 = (\neg x_4 \lor x_5 \lor x_{10})\)
\(\omega_5 = (\neg x_4 \lor x_6 \lor x_{11})\)
\(\omega_6 = (\neg x_5 \lor \neg x_6)\)
\(\omega_7 = (x_1 \lor x_7 \lor \neg x_{12})\)
\(\omega_8 = (x_1 \lor x_8)\)
\(\omega_9 = (\neg x_7 \lor \neg x_8 \lor \neg x_{13})\)
Example

\[
\omega_1 = (\neg x_1 \lor x_8 \lor x_9)
\]

\[
\omega_2 = (\neg x_1 \lor x_8 \lor \neg x_9)
\]

\[
\omega_3 = (\neg x_1 \lor \neg x_8 \lor x_9)
\]

\[
\omega_4 = (\neg x_1 \lor \neg x_8 \lor \neg x_9)
\]

\[
\omega_5 = (x_1 \lor x_3)
\]

\[
\omega_6 = (x_1 \lor \neg x_3)
\]
Is that all?

- Huge overhead for boolean constraint propagation (BCP)
- Better decision heuristics
- Better learning, problem specific

**Better engineering!**

Chaff