COS 326: Functional Programming

Lecture

Boolean Satisfiability (SAT) Solvers

Aarti Gupta

Acknowledgements: Sharad Malik, Emina Torlak
What is SAT?

Given a propositional logic (Boolean) formula,
find a variable assignment such that the formula evaluates to true,
or prove that no such assignment exists.

\[ F = (a \lor b) \land (a' \lor b' \lor c) \]

For \( n \) variables, there are \( 2^n \) possible truth assignments to be checked.

First established NP-Complete problem.

SAT and logics

Propositional logic is a subset of
• First order logic
• Higher-order logic

Validity of formulas (i.e., checking if all variable assignments make the formula true)
• Propositional logic: decidable
• First order logic: semi-decidable
• Arithmetic and higher-order logic: undecidable

Complexity of SAT: NP-complete
• There is no known polynomial-time algorithm
• … but often tractable in practice on real-world problems!
Why is it of interest now?

SAT: is a Boolean formula $f$ satisfiable?

<table>
<thead>
<tr>
<th>Year</th>
<th>System</th>
<th>Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>1952</td>
<td>Quine</td>
<td>$\approx 10$ var</td>
</tr>
<tr>
<td>1960</td>
<td>DP</td>
<td>$\approx 10$ var</td>
</tr>
<tr>
<td>1962</td>
<td>DLL</td>
<td>$\approx 10$ var</td>
</tr>
<tr>
<td>1986</td>
<td>BDD</td>
<td>$\approx 100$ Var</td>
</tr>
<tr>
<td>1988</td>
<td>SOCRATES</td>
<td>$\approx 3k$ Var</td>
</tr>
<tr>
<td>1996</td>
<td>GRASP</td>
<td>$\approx 1k$ Var</td>
</tr>
<tr>
<td>1997</td>
<td>SATO</td>
<td>$\approx 1k$ Var</td>
</tr>
<tr>
<td>2001</td>
<td>Chaff</td>
<td>$\approx 10k$ Var</td>
</tr>
<tr>
<td>2003</td>
<td>MiniSAT</td>
<td>$\approx 10k$ Var</td>
</tr>
<tr>
<td>2004</td>
<td>DPLL(T)</td>
<td></td>
</tr>
<tr>
<td>2008</td>
<td>Z3</td>
<td></td>
</tr>
<tr>
<td>2009</td>
<td>SMT</td>
<td></td>
</tr>
</tbody>
</table>

(Source: Sharad Malik)

SMT (Satisfiability Modulo Theory): is a first-order logic formula theory-satisfiable?
Where are we today?

Intractability of the problem no longer daunting
• can regularly solve practical instances with millions of variables and constraints

SAT has matured from theoretical interest to practical impact
• Electronic Design Automation (EDA)
  • Widely used in many aspects of chip design: equivalence checking, assertion verification, synthesis, debugging, post-silicon validation
• Software verification
  • Commercial use at Microsoft, Amazon, Google, Facebook, …
• AI and Planning problems

• CAV 2009 Award due to industrial impact
  • Chaff solver team from Princeton shared the award!
Where are we today?

Significant SAT community
- SatLive Portal (http://www.satlive.org/)
- Annual SAT competitions (http://www.satcompetition.org/)
- SAT Conference (http://www.satisfiability.org/)

Emboldened researchers to take on even harder problems related to SAT
- Max-SAT: for optimization
- Satisfiability Modulo Theories (SMT): for more expressive theories
- Quantified Boolean Formulas (QBF): for more complex problems

- Many ideas from SAT solvers are applied here
Some basics first ...
Boolean formulas: Syntax

Formula $f =$  // inductive definition
  $true \mid false \mid$  // base cases: constants
  $v \mid$  // base case: variable
  $NOT\ g \mid g\ AND\ h \mid g\ OR\ h$  // inductive cases

where

$v$ is a propositional variable, i.e., it takes value $true$ or $false$

$NOT, AND, OR$ are the usual Boolean operators
**Boolean formulas: Semantics**

Given a Boolean formula $f$, and an *Interpretation* $M$, which maps variables to true/false.

We can *evaluate* $f$ under $M$ to produce a Boolean result (true or false).

- Base case true: return true
- Base case false: return false
- Base case variable $v$: return value of $v$ in $M$
- Inductive cases: return result by using the *truth tables* shown below

<table>
<thead>
<tr>
<th>$g$</th>
<th>Not $g$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$g$</th>
<th>$h$</th>
<th>$g$ \ AND \ $h$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$g$</th>
<th>$h$</th>
<th>$g$ \ OR \ $h$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Example: Evaluate $f$: $(a \ OR \ b) \ AND \ (NOT \ c)$ under $M$: $\{a \mapsto \text{true}, \ b \mapsto \text{false}, \ c \mapsto \text{false}\}$

\[ f = (1 \ OR \ 0) \ AND \ (NOT \ 0) \]

\[ \ldots \ f \text{ evaluates to true under } M \]
### Boolean formulas: Semantics

Given a **Boolean formula** $f$, and an **Interpretation** $M$, which maps variables to true/false.

If $f$ evaluates to true under $M$, we say that $M$ **satisfies** $f$.

**Example:** $f: (a \lor b) \land (a' \lor b' \lor c)$, $M_1: \{a \rightarrow \text{true}, b \rightarrow \text{true}, c \rightarrow \text{false}\}$

**(Q1) Does $M_1$ satisfy $f?$**

*No, because $f$ evaluates to false under $M_1.*

**(Q2) Is $f$ satisfiable, i.e., does there exist an $M$ such that $M$ satisfies $f?$**

*Yes, $f$ is satisfiable.*

For example, $M_2: \{a \rightarrow \text{true}, b \rightarrow \text{true}, c \rightarrow \text{true}\}$ satisfies $f$.

**SAT solvers can automatically find a satisfying interpretation! (if it exists)**
SAT solvers: a condensed history

Deductive
- Davis-Putnam 1960 [DP]
- Based on “resolution”

Backtracking Search
- Davis, Putnam, Logemann and Loveland 1962 [DLL, DPLL]
- Exhaustive search for satisfying assignment

Conflict Driven Clause Learning [CDCL]
- GRASP: Integrate a constraint learning procedure, 1996

Locality Based Search
- Emphasis on exhausting local sub-spaces, e.g., Chaff, 2001
- Added focus on efficient implementations

“Pre-processing”
- Peephole optimization, e.g., MiniSat, 2005

…
SAT problem representation

Boolean formulas represented in: Conjunctive Normal Form (CNF)

\[(a \lor b \lor c) \land (a' \lor b' \lor c) \land (a' \land b \land c') \land (a \land b' \land c')\]

- **formula**: is a conjunction of clauses
- **clause**: is a disjunction of literals
- **literal**: is a variable or its negation

• for formula to be true: each clause must be satisfied
• in each clause: some literal must be true
CNF Representation

Q: Can any Boolean formula be converted to CNF?
   Yes!

Q: How can I convert a Boolean formula to CNF?
   Many different translations exist …

You will explore two translations in Assignment 3

1. A naïve translation based on de Morgan’s Laws
2. Tseitin transformation (useful for checking SAT)
Translation to CNF

We can view CNF as a special kind of expression tree for a Boolean formula.

CNF has maximum 3 levels, where

- top-level node is AND
- second-level nodes are OR
- NOT (i.e., negation) can be optionally applied *only at the leaves*
- leaf level nodes are variables (with/without negation)

Example: \((x_1 \lor x_3' \lor x_4) \land (x_2 \lor x_3' \lor x_4)\) is in CNF
(1) Naïve Translation to CNF

Given an *arbitrary* expression tree for a Boolean formula

- #1: Push NOT nodes inside an AND or OR
  - *de Morgan’s Laws*
    - $\text{NOT} (p \text{ || } q) = (\text{NOT } p) \&\& (\text{NOT } q)$
    - $\text{NOT} (p \&\& q) = (\text{NOT } p) \text{ || } (\text{NOT } q)$

- #2: Distribute outer OR nodes over an inner AND
  - $(p \&\& q) \text{ || } (x \&\& y) = (p \text{ || } x) \&\& (p \text{ || } y) \&\& (q \text{ || } x) \&\& (q \text{ || } y)$

- Simplifications
  - $\text{NOT}(\text{NOT } p) = p$
  - Nested AND nodes to a single AND (when possible)
  - Nested OR nodes to a single OR node (when possible)

Correctness of Naïve translation

- The generated CNF formula is *equivalent* to the given formula
- However, its size can be *exponentially bigger* 😞
(1) Naïve translation to CNF: Example

Example: \((x_1 \&\& x_2) \lor (\neg (x_3 \&\& \neg x_4))\)

\[
= (x_1 \&\& x_2) \lor (\neg x_3 \lor \neg (\neg x_4)) \quad \ldots \quad \#1
\]

\[
= (x_1 \&\& x_2) \lor (\neg x_3 \lor x_4) \quad \ldots \quad \text{NOT simplification}
\]

\[
= (x_1 \lor \neg x_3 \lor x_4) \&\& (x_2 \lor \neg x_3 \lor x_4) \quad \ldots \quad \#2
\]

\[
= (x_1 \lor x_3' \lor x_4) \&\& (x_2 \lor x_3' \lor x_4)
\]

Original formula

CNF representation
Tseitin Transformation Example

Main idea: Introduce fresh variable for each subformula and write "equations"

New variables: y1, y2, y3, y4, y5

Equations
- y1 = x1 && x2
- y2 = y1 || y3
- y3 = NOT y4
- y4 = x3 && y5
- y5 = NOT x4

CNF
- (x1 || y1’) && (x2 || y1’) && (x1’ || x2’ || y1) && (y1’ || y2) && (y3’ || y2) && (y1 || y3 || y2’) && (y3 || y4) && (y3’ || y4’) && (x3 || y4’) && (y5 || y4’) && (x3’ || y5 || y4) && (x4 || y5) && (x4’ || y5’) && (y2)

<table>
<thead>
<tr>
<th>Equation</th>
<th>CNF to implement the Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>z = NOT x</td>
<td>(x</td>
</tr>
<tr>
<td>z = x &amp;&amp; y</td>
<td>(x</td>
</tr>
<tr>
<td>z = x</td>
<td></td>
</tr>
</tbody>
</table>
(2) Tseitin Transformation

Main idea: Introduce new (fresh) variables for each subformula
- Write "equations": new variable = subformula
- Generate CNF for each equation, depending on the operator, as follows:

<table>
<thead>
<tr>
<th>Equation</th>
<th>CNF to implement the Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$z = \text{NOT } x$</td>
<td>$(x \lor z) \land (x' \lor z')$</td>
</tr>
<tr>
<td>$z = x \land y$</td>
<td>$(x \lor z') \land (y \lor z') \land (x' \lor y' \lor z)$</td>
</tr>
<tr>
<td>$z = x \lor y$</td>
<td>$(x' \lor z) \land (y' \lor z) \land (x \lor y \lor z')$</td>
</tr>
</tbody>
</table>

- AND together the CNFs for all equations
- AND a clause with a single literal for the top-level formula (if you want to check satisfiability of the top-level formula)

Correctness of Tseitin transformation
- For a given formula $f$, let $\text{Tseitin}(f)$ denote the generated CNF formula
- Size of $\text{Tseitin}(f)$ is $\textit{linear}$ in the size of $f$
- $\text{Tseitin}(f)$ is $\textit{equi-satisfiable}$ with $f$
  - i.e., $\text{Tseitin}(f)$ is satisfiable $\textit{if and only if}$ $f$ is satisfiable
Boolean circuit to CNF

Tseitin transformation for Boolean (combinational) circuits

\[ d \equiv (a \lor b) \]
\[ (a \lor b \lor d') \]
\[ (a' \lor d) \]
\[ (b' \lor d) \]

\[ e \equiv (c \land d) \]
\[ (c' \land d' \land e) \]
\[ (d' \land e') \]
\[ (c \land e') \]

linear time conversion of any Boolean circuit into CNF using auxiliary variables
Applications of SAT

- Checking circuit equivalence

Formula f:
\[ \text{CNF}(C1) \land \text{CNF}(C2) \land (o1 \lor o2) \land (o1' \lor o2') \]

Use a SAT solver to check if formula f is satisfiable
- If f is satisfiable, then C1 and C2 are not equivalent
- If f is unsatisfiable, then C1 and C2 are equivalent

SAT solver checks over all inputs (without enumeration)!
Another application

Automatic test generation

```plaintext
if (c1) then {
    if (c2) then {
        ...
    } else if (c3) then {
        ...
        assert(c4)
        ...
    }
}
```

- Suppose you are testing a program to check if the `assert` can fail
- Note that the `assert` can fail only when `c1` is true, `c2` is false, `c3` is true, and `c4` is false.
- It may be difficult to manually come up with such an input

**SAT solvers are used for automatically generating test inputs!**

- Represent `int` variables in programs as 64-bit bitvectors
- Construct formula for checking satisfiability of `path-condition`
- Use SMT solvers, e.g., Z3 (which uses SAT solver for bitvectors)
SAT solvers: a condensed history

Deductive
- Davis-Putnam 1960 [DP]
- Based on “resolution”

➢ Backtracking Search
- Davis, Putnam, Logemann and Loveland 1962 [DLL, DPLL]
- Exhaustive search for satisfying assignment

Conflict Driven Clause Learning [CDCL]
- GRASP: Integrate a constraint learning procedure, 1996

Locality Based Search
- Emphasis on exhausting local sub-spaces, e.g., Chaff, 2001
- Added focus on efficient implementations

“Pre-processing”
- Peephole optimization, e.g., MiniSat, 2005

…
Basic DPLL Search


Performs backtracking search over variable assignments

- We will represent the formula in CNF
  - Implicitly a set of clauses (i.e., the AND is implicit)
- We will make “decisions” by assigning values to variables
- We will keep track of a “decision tree” that records the current partial assignment to variables
- We will “backtrack” when the latest decision cannot lead to a satisfying assignment ("solution")

During the search:

- If all clauses are satisfied, we have found a satisfying assignment (and can terminate)
- If we have exhausted all possible assignments without finding a solution, then the formula is unsatisfiable
Basic DPLL Search

Definitions: under a given partial assignment (PA)

- A variable may be
  - assigned (true or false literal)
  - unassigned

- A clause may be
  - satisfied (≥1 true literal)
  - unsatisfied (all false literals)
  - unit (one unassigned literal, rest false)
  - unresolved (otherwise)
Basic DPLL Search

(a’ + b + c)
(a + c + d)
(a + c + d’)
(a + c’ + d)
(a + c’ + d’)
(b’ + c’ + d)
(a’ + b + c’)
(a’ + b’ + c)

+ denotes OR
’ denotes NOT
Basic DPLL Search

\[ (a' + b + c) \]
\[ (a + c + d) \]
\[ (a + c + d') \]
\[ (a + c' + d) \]
\[ (a + c' + d') \]
\[ (b' + c' + d) \]

\[ (a' + b + c') \]
\[ (a' + b' + c) \]

\[ \leftrightarrow \text{Decision} \]
Basic DPLL Search

- \((a' + b + c)\)
- \((a + c + d)\)
- \((a + c + d')\)
- \((a + c' + d)\)
- \((a + c' + d')\)
- \((b' + c' + d)\)
- \((a' + b + c')\)
- \((a' + b' + c)\)

Decision

\(\rightarrow\)

\(\leftarrow\)
Basic DPLL Search

(a' + b + c)
(a + c + d)
(a + c + d')

(a + c' + d)
(a + c' + d')
(b' + c + d)
(a' + b + c')
(a' + b' + c)

Decision

Diagram of nodes a, b, c with arrows indicating the search process.
Basic DPLL Search

BCP: Boolean Constraint Propagation repeatedly applies *Unit Clause Rule*

If all but one literals in a clause are false, then the remaining literal is *implied* to true.
Basic DPLL Search

BCP: Boolean Constraint Propagation repeatedly applies *Unit Clause Rule*

If all but one literals in a clause are false, then the remaining literal is *implied* to true.
Basic DPLL Search

(a' + b + c) → (a + c + d) → (a + c + d') → (a + c' + d) → (a + c' + d') → (a' + b + c) → (a' + b' + c) → Unit

d=1, d=0
Basic DPLL Search

\[(a' + b + c)\]  
\[(a + c + d)\]  
\[(a + c + d')\]  
\[(a + c' + d)\]  
\[(a + c' + d')\]  
\[(b' + c' + d)\]  
\[(a' + b + c')\]  
\[(a' + b' + c)\]  

Conflict!
Basic DPLL Search

(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)

→
→
→
→

→

Backtrack

0

0

0
Basic DPLL Search

Forced Decision

\[
\begin{align*}
(a' + b + c) \\
(a + c + d) \\
(a + c + d') \\
\end{align*}
\]

\[
\begin{align*}
(a + c' + d) \\
(a + c' + d') \\
\end{align*}
\]

\[
\begin{align*}
(b' + c' + d) \\
(a' + b + c') \\
(a' + b' + c) \\
\end{align*}
\]
Basic DPLL Search

Forced Decision

\[(a' + b + c)
(a + c + d)
(a + c + d')\]

\[\rightarrow\]

\[(a + c' + d)
(a + c' + d')\]

\[\rightarrow\]

\[(b' + c' + d)
(a' + b + c')
(a' + b' + c)\]

Conflict!

\[d=1, d=0\]
Basic DPLL Search

\[(a' + b + c)\]

\[(a + c + d)\]

\[(a + c + d')\]

\[(a + c' + d)\]

\[(a + c' + d')\]

\[(b' + c' + d)\]

\[(a' + b + c')\]

\[(a' + b' + c)\]
Basic DPLL Search

- \((a' + b + c)\)
- \((a + c + d)\)
- \((a + c + d')\)
- \((a + c' + d)\)
- \((a + c' + d')\)
- \((b' + c' + d)\)
- \((a' + b + c')\)
- \((a' + b' + c)\)

Diagram:
- Node a
- Node b
- Node c
- Backtrack arrow
- Red boxes
Basic DPLL Search

(a’ + b + c)
(a + c + d)
(a + c + d’)
(a + c’ + d)
(a + c’ + d’)
(b’ + c’ + d)
(a’ + b + c’)
(a’ + b’ + c)

Forced Decision

Diagram:

- Node a connected to 0
- Node b connected to 0 and 1
- Node c connected to 0 and 1
- Red rectangles below c
Basic DPLL Search

Note: same two clauses are unit (as before) cause the same conflict!
Basic DPLL Search

\[(a' + b + c)\]
\[\rightarrow (a + c + d)\]
\[\rightarrow (a + c + d')\]
\[\rightarrow (a + c' + d)\]
\[\rightarrow (a + c' + d')\]
\[\rightarrow (b' + c' + d)\]
\[\rightarrow (a' + b + c')\]
\[\rightarrow (a' + b' + c)\]

\[\text{Backtrack}\]

Diagram:
- Node a with 0 and 0 branches
- Node b with 0, 1, and 0 branches
- Node c with 0 and 1 branches
- Node c with 0 and 0 branches
Basic DPLL Search

Forced Decision

\[ (a' + b + c) \]
\[ (a + c + d) \]
\[ (a + c + d') \]

\[ (a + c' + d) \]
\[ (a + c' + d') \]
\[ (a' + b + c) \]

\[ (a' + b + c') \]
\[ (a' + b' + c) \]

Conflict!
Basic DPLL Search

(a’ + b + c)
→ (a + c + d)
→ (a + c + d’)
→ (a + c’ + d)
→ (b’ + c’ + d)
→ (a’ + b + c’)
→ (a’ + b’ + c)

Diagram:

```
   a
  /   \
 /     \
 b       c
   |     |
   0     0
   |     |
 c               c
   |     |
   1     1
```

Backtrack
Basic DPLL Search

\((a' + b + c)\)
\((a + c + d)\)
\((a + c + d')\)
\((a + c' + d)\)
\((a + c' + d')\)
\((b' + c + d)\)
\((a' + b + c')\)
\((a' + b' + c)\)

Diagram:

- Backtrack
Basic DPLL Search

Forced Decision
Basic DPLL Search

(a’ + b + c)
(a + c + d)
(a + c + d’)
(a + c’ + d)
(a + c’ + d’)
(b’ + c’ + d)
(a’ + b + c’)
(a’ + b’ + c)

\[ a \]
\[ b \]
\[ c \]

Decision: \( c = 1 \)
Implication: \( a, b, c \)
Basic DPLL Search

(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)

→

b

Implication

c=1, d=1

0 1
0 1
0 1

46
Basic DPLL Search

Backtracking search with BCP (unit clause rule)
DPLL SAT Solver

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

unit clause rule

decision heuristics

backtracking search

Much research, many heuristics over >40 years …
Poor scalability: why?

DPLL(F)

G ← BCP(F)

if G = ⊤ then return true
if G = ⊥ then return false
p ← choose(vars(G))

return DPLL(G{p ↦ ⊤}) = “SAT” or DPLL(G{p ↦ ⊥})

No learning:
Throws away all the work that concluded that current PA is bad

Naïve decision heuristics:
Usually choice is independent of “state” of search

Chronological backtracking:
backtracks one level, even if current PA was doomed at an earlier level

JRW project ideas!
SAT (Satisfiability): is a Boolean formula \( f \) satisfiable?

SMT (Satisfiability Modulo Theory): is a first-order logic formula \( f \) theory-satisfiable?

**SAT/SMT Timeline**

- **1952**: Quine » 10 var
- **1960**: DP » 10 var
- **1962**: DLL » 10 var
- **1986**: BDD » 100 Var
- **1988**: SOCRATES » 3k Var
- **1996**: GRASP » 1k Var
- **1996**: Stålmarck » 1k Var
- **1997**: SATO » 1k Var
- **2001**: Chaff » 10k var
- **2003**: MiniSAT » 10k var
- **2004**: DPLL(T)
- **2008**: Z3
- **2009**: SMT

(Source: Sharad Malik)

**Key Techniques**

- **Conflict Driven Clause Learning**: Non-chronological backtracking
- **VSIDS decision heuristic, 2-literal watching in BCP**
SAT solvers in verification

COS 516 covers many of these topics
Optional Readings

Sharad Malik, Lintao Zhang:

Leonardo Mendonça de Moura, Nikolaj Bjørner: