Satisfiability of Propositional Formulas

Mooly Sagiv

Slides by Sharad Malik, Ohad Shacham, Daniel Kroening and Ofer Strichman
The SAT Problem

- Given a propositional formula (Boolean function)
  \[ \varphi = (a \lor b) \land (\neg a \lor \neg b \lor c) \]

- Determine if \( \varphi \) is valid

- Determine if \( \varphi \) is satisfiable
  - Find a satisfying assignment or report that such does not exit

- For \( n \) variables, there are \( 2^n \) possible truth assignments to be checked
Why Bother?

• Core computational engine for major applications
  – Artificial Intelligence
    • Knowledge base deduction
    • Automatic theorem proving
  – Electronic Design Automaton
    • Testing and Verification
    • Logic synthesis
    • FPGA routing
    • Path delay analysis
    • And more…
  – Software Verification
Problem Representation

- Represent the formulas in Conjunctive Normal Form (CNF)
- Conversion to CNF is straightforward
  - \( a \lor (b \land \neg (c \lor \neg d)) \equiv (a \lor (b \land \neg c \land \neg d)) \equiv (a \lor (b \land \neg c \land d)) \equiv (a \lor b) \land (a \lor \neg c) \land (a \lor d) \)
  - May need to add variables
- Notations
  - Literals
    - Variable or its negation
  - Clauses
    - Disjunction of literals
  - \( \varphi = (a \lor b) \land (\neg a \lor \neg b \lor c) \equiv (a + b)(a' + b' + c) \)
- Advantages of CNF
  - Simple data structure
  - Compact
  - Compositional
  - All the clauses need to be satisfied
Complexity Results

• First established NP-Complete problem
  – Even when at most 3 literals per clause (3-SAT)
    – No polynomial algorithm for all instances unless P = NP

• Becomes polynomial when
  – At most two literals per clause (2-SAT)
  – At most one positive literal in every clause (Horn)
Goals

• Develop algorithms which solve all SAT instances
• Exponential worst case complexity
• But works well on many instances
  – Interesting Heuristics
  – Annual SAT conferences
  – SAT competitions
    • Randomly, Handmade, Industrial, AI
  – 10 Millions variables!
SAT made some progress…
Clause Resolution

• Resolution of a pair of clauses with exactly ONE incompatible variable

• What if more than one incompatible variables?
Davis Putnam Algorithm


- Iteratively select a variable for resolution till no more variables are left
- Report UNSAT when the empty clause occurs
- Can discard resolved clauses after each iteration

Potential memory explosion problem!
Can we avoid using exponential space?
DLL Algorithm

• Davis, Logemann and Loveland


• Basic framework for many modern SAT solvers

• Also known as DPLL for historical reasons
Basic DLL Procedure - DFS

(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 DLL Procedure - DFS

(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
Basic DLL Procedure - DFS

- $(a' + b + c)$
- $(a + c + d)$
- $(a + c + d')$
- $(a + c' + d)$
- $(a + c' + d')$
- $(b' + c' + d)$
- $(a' + b + c')$
- $(a' + b' + c)$
Basic DLL Procedure - DFS

\[(a' + b + c)\]
\[(a + c + d)\]
\[(a + c + d')\]
\[(a + c' + d)\]
\[(a + c' + d')\]
\[(b' + c' + d)\]
\[(a' + b + c')\]
\[(a' + b' + c)\]
Basic DLL Procedure - DFS

(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 DLL Procedure - DFS

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

Implication Graph

Conflict!
Basic DLL Procedure - DFS

Implication Graph

Conflict!
Basic DLL Procedure - DFS

(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
- Arrow from a to b
- Arrow from b to c
- Arrow from c to red

Backtrack
Basic DLL Procedure - DFS

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

\(a = 0\)
\(d = 1\)
\(c = 1\)
\(d = 0\)

Conflict!

Forced Decision
Basic DLL Procedure - DFS

(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
Basic DLL Procedure - DFS

Forced Decision
Basic DLL Procedure - DFS

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

Conflict!
Basic DLL Procedure - DFS

(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 DLL Procedure - DFS

(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)  
(a + c + d)  
(a + c + d’)  

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

Forced Decision

Conflict!
Basic DLL Procedure - DFS

(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 DLL Procedure - DFS

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

Forced Decision
Basic DLL Procedure - DFS

(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)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)
Basic DLL Procedure - DFS

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

Conflict!
Basic DLL Procedure - DFS

(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:

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

← Backtrack
Basic DLL Procedure - DFS

(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

(a' + b' + c)
Basic DLL Procedure - DFS

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

a=1
b=1
c=1
d=1

0 1
0 1
0 1
0 1
Basic DLL Procedure - DFS

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

(a' + b' + c)

(a' + b' + c)

(b' + c' + d)

(a' + b' + c)

(b' + c' + d)

(a' + b' + c)

(b' + c' + d)

(a' + b' + c)
Implications and Boolean Constraint Propagation

- **Implication**
  - A variable is forced to be assigned to be True or False based on previous assignments

- **Unit clause rule** (rule for elimination of one literal clauses)
  - An **unsatisfied** clause is a **unit** clause if it has exactly one unassigned literal

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

\[a = T, \ b = T, \ c \text{ is unassigned}\]

  - The unassigned literal is implied because of the unit clause

- **Boolean Constraint Propagation** (BCP)
  - Iteratively apply the unit clause rule until there is no unit clause available

- Workhorse of DLL based algorithms
A Basic SAT algorithm

While (true) {
    if (!Decide()) return (SAT)
    while (!BCP())
        if (!Resolve_Conflict()) return (UNSAT)

Apply repeatedly the \textit{unit clause rule}. Return False if reached a conflict

Choose the next variable and value. Return False if all variables are assigned

Backtrack until no conflict. Return False if impossible
x1 + x4
x1 + x3’ + x8’
x1 + x8 + x12
x2 + x11
x7’ + x3’ + x9
x7’ + x8 + x9’
x7 + x8 + x10’
x7 + x10 + x12’

x1 = 0
Conflict Driven Learning and Non-chronological Backtracking

\[ x_1 + x_4 \]
\[ x_1 + x_3' + x_8' \]
\[ x_1 + x_8 + x_{12} \]
\[ x_2 + x_{11} \]
\[ x_7' + x_3' + x_9 \]
\[ x_7' + x_8 + x_9' \]
\[ x_7' + x_8 + x_{10'} \]
\[ x_7 + x_{10} + x_{12'} \]

\[ x_4 = 1 \]
\[ x_1 = 0 \]

\[ x_1 = 0, \ x_4 = 1 \]
Conflict Driven Learning and Non-chronological Backtracking

\[ x_1 + x_4 \]
\[ x_1 + x_3' + x_8' \]
\[ x_1 + x_8 + x_{12} \]
\[ x_2 + x_{11} \]
\[ x_7' + x_3' + x_9 \]
\[ x_7' + x_8 + x_9' \]
\[ x_7 + x_8 + x_{10'} \]
\[ x_7 + x_{10} + x_{12'} \]

\[ x_4 = 1 \]
\[ x_1 = 0 \]
\[ x_3 = 1 \]
Conflict Driven Learning and Non-chronological Backtracking

\[ x_1 + x_4 \]
\[ x_1 + x_3' + x_8' \]
\[ x_1 + x_8 + x_{12} \]
\[ x_2 + x_{11} \]
\[ x_7' + x_3' + x_9 \]
\[ x_7' + x_8 + x_9' \]
\[ x_7 + x_8 + x_{10'} \]
\[ x_7 + x_{10} + x_{12'} \]

\[ x_1 = 0, \; x_4 = 1 \]
\[ x_3 = 1, \; x_8 = 0 \]

\[ x_4 = 0 \]
\[ x_1 = 0 \]
\[ x_3 = 1 \]
\[ x_8 = 0 \]
Conflict Driven Learning and Non-chronological Backtracking

\[
x_1 + x_4 \\
x_1 + x_3' + x_8' \\
x_1 + x_8 + x_{12} \\
x_2 + x_{11} \\
x_7' + x_3' + x_9 \\
x_7' + x_8 + x_9' \\
x_7 + x_8 + x_{10'} \\
x_7 + x_{10} + x_{12'}
\]

\[
x_4 = 1 \\
x_1 = 0 \\
x_3 = 1 \\
x_8 = 0 \\
x_{12} = 1
\]
Conflict Driven Learning and Non-chronological Backtracking

\[
\begin{align*}
x_1 + x_4 & \\
x_1 + x_3' + x_8' & \\
x_1 + x_8 + x_{12} & \\
x_2 + x_{11} & \\
x_7' + x_3' + x_9 & \\
x_7' + x_8 + x_9' & \\
x_7 + x_8 + x_{10}' & \\
x_7 + x_{10} + x_{12}' & \\
\end{align*}
\]
Conflict Driven Learning and Non-chronological Backtracking

\[ x_1 + x_4 \]
\[ x_1 + x_3' + x_8' \]
\[ x_1 + x_8 + x_{12} \]
\[ x_2 + x_{11} \]
\[ x_7' + x_3' + x_9 \]
\[ x_7' + x_8 + x_9' \]
\[ x_7 + x_8 + x_{10}' \]
\[ x_7 + x_{10} + x_{12}' \]
Conflict Driven Learning and Non-chronological Backtracking

\[ x_1 + x_4 \]
\[ x_1 + x_3' + x_8' \]
\[ x_1 + x_8 + x_{12} \]
\[ x_2 + x_{11} \]
\[ x_7' + x_3' + x_9 \]
\[ x_7' + x_8 + x_9' \]
\[ x_7 + x_8 + x_{10'} \]
\[ x_7 + x_{10} + x_{12'} \]
Conflict Driven Learning and Non-chronological Backtracking

\[ x_1 + x_4 \]
\[ x_1 + x_3' + x_8' \]
\[ x_1 + x_8 + x_12 \]
\[ x_2 + x_11 \]
\[ x_7' + x_3' + x_9 \]
\[ x_7' + x_8 + x_9' \]
\[ x_7 + x_8 + x_10' \]
\[ x_7 + x_10 + x_12' \]
Conflict Driven Learning and Non-chronological Backtracking

\[ x_1 + x_4 \]
\[ x_1 + x_3' + x_8' \]
\[ x_1 + x_8 + x_{12} \]
\[ x_2 + x_11 \]
\[ x_7' + x_3' + x_9 \]
\[ x_7' + x_8 + x_9' \]
\[ x_7 + x_8 + x_{10}' \]
\[ x_7 + x_{10} + x_{12}' \]

\[ x_3 = 1, x_8 = 0, x_{12} = 1 \]
\[ x_2 = 0, x_{11} = 1 \]
\[ x_7 = 1, x_9 = 1 \]
\[ x_3 = 1 \land x_7 = 1 \land x_8 = 0 \rightarrow \text{conflict} \]
Conflict Driven Learning and Non-chronological Backtracking

x1 + x4
x1 + x3' + x8'
x1 + x8 + x12
x2 + x11
x7' + x3' + x9
x7' + x8 + x9'
x7 + x8 + x10'
x7 + x10 + x12'

Add conflict clause: x3' + x7' + x8

x1 = 0, x4 = 1
x3 = 1, x8 = 0, x12 = 1
x2 = 0, x11 = 1
x7 = 1, x9 = 1

x3 = 1 \land x7 = 1 \land x8 = 0 \rightarrow \text{conflict}

Add conflict clause: x3' + x7' + x8
Conflict Driven Learning and Non-chronological Backtracking

\[ x_1 + x_4 \]
\[ x_1 + x_3' + x_8' \]
\[ x_1 + x_8 + x_{12} \]
\[ x_2 + x_{11} \]
\[ x_7' + x_3' + x_9 \]
\[ x_7' + x_8 + x_9' \]
\[ x_7 + x_8 + x_{10}' \]
\[ x_7 + x_{10} + x_{12}' \]

Add conflict clause: \[ x_3' + x_7' + x_8 \]
Conflict Driven Learning and Non-chronological Backtracking

\[ x_1 + x_4 \\
 x_1 + x_3' + x_8' \\
 x_1 + x_8 + x_{12} \\
 x_2 + x_{11} \\
 x_7' + x_3' + x_9 \\
 x_7' + x_8 + x_9' \\
 x_7 + x_8 + x_{10'} \\
 x_7 + x_{10} + x_{12'} \\
 x_3' + x_7' + x_8 \]

Backtrack to the decision level of \( x_3 = 1 \)

\( x_7 = 0 \)
Conflict Driven Learning and Non-chronological Backtracking

\[ x_1 + x_4 \]
\[ x_1 + x_3' + x_8' \]
\[ x_1 + x_8 + x_{12} \]
\[ x_2 + x_{11} \]
\[ x_7' + x_3' + x_9 \]
\[ x_7' + x_8 + x_9' \]
\[ x_7 + x_8 + x_{10'} \]
\[ x_7 + x_{10} + x_{12'} \]
\[ x_3' + x_7 + x_8' \]
What’s the big deal?

Conflict clause: $x_1' + x_3 + x_5'$

Significantly prune the search space – learned clause is useful forever!

Useful in generating future conflict Clauses

No longer polynomial space
Restart

- Abandon the current search tree and reconstruct a new one.
- The clauses learned prior to the restart are *still there* after the restart and can help pruning the search space.
- Adds to robustness in the solver.

Conflict clause: \( x_1 + x_3 + x_5 \)
What “causes” an implication? When can it occur?

- All literals in a clause but one are assigned to F
  - $(v_1 + v_2 + v_3)$: implied cases: $(0 + 0 + v_3)$ or $(0 + v_2 + 0)$ or $(v_1 + 0 + 0)$
- For an N-literal clause, this can only occur after N-1 of the literals have been assigned to F
- So, (theoretically) we could completely ignore the first N-2 assignments to this clause
- In reality, we pick two literals in each clause to “watch” and thus can ignore any assignments to the other literals in the clause
  - Example: $(v_1 + v_2 + v_3 + v_4 + v_5)$
  - $(v_1=X + v_2=X + v_3=? \{\text{i.e. X or 0 or 1}\} + v_4=? + v_5=?)$
Chaff Decision Heuristic - VSIDS

• Variable State Independent Decaying Sum
  – Rank variables by literal count in the initial clause database
  – Periodically, divide all counts by a constant
  – Only increment counts as new clauses are added

• Quasi-static:
  – Static because it doesn’t depend on var state
  – Not static because it gradually changes as new clauses are added
    • Decay causes bias toward *recent* conflicts
Finding a Solution to a SAT problem is can be viewed as 2 player game

- **Player 1**: tries to find satisfying assignment
- **Player 2**: tries to show that such assignment does not exist

Let A be an arbitrary assignment
while true:
  if $A \models C$ then return SAT
  if $() \in C$ then return UNSAT
  let $c \in C$ such that not $A \models c$ and let $A'$ such that $A' \models c$
  $A := A'$
  $||$
  let $c' \not\in C$ such that $C \models c'$ and not $A \models c'$
  $C := C \cup \{c'\}$
Example Game 1

\[(a + b) (a + b') (a' + c)(a' + c')\]
Example Game 2

\[(a + b + c)(b + c' + f')(b' + e)\]
Some Bibliography

- **Chaff: Engineering an Efficient SAT Solver**
  Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, Sharad Malik (DAC'01)

- **Efficient Conflict Driven Learning in a Boolean Satisfiability Solver**
  Lintao Zhang, Conor F. Madigan, Matthew H. Moskewicz (IJCAD’01)

- **A New Method for Solving Hard Satisfiability Problems**
  Bart Selman, Hector Levesque, David Mitchell (AAI’92)
Post Chaff SAT solvers
  - BerkMin
  - Seige
  - miniSat
  - HaifaSAT
  - JeruSAT (Alex Nadel)

The Stålmarck’s algorithm

Hyperresolution

Local Search
Open Question

• Is there a subset of a useful propositional logic beyond Horn clauses which:
  – Allows polynomial SAT
  – Includes many of the practical instances
  – Some recent ideas in
Summary

• Rich history of emphasis on practical efficiency
• Need to account for computation cost in search space pruning
• Need to match algorithms with underlying processing system architectures
• Specific problem classes can benefit from specialized algorithms
  – Identification of problem classes?
  – Dynamically adapting heuristics?
• We barely understand the tip of the iceberg here
  – much room to learn and improve