The Search for Efficient Boolean Satisfiability Solvers: An Abbreviated History

Sharad Malik
Dept. of Electrical Engineering
Princeton University
Acknowledgements

- Chaff authors:
  - Matthew Moskewicz (now at UC Berkeley)
  - Conor Madigan (now at MIT)

- Princeton University SAT group:
  - Daijue Tang
  - Yinlei Yu
  - Yogesh Mahajan
  - Zhaohui Fu
  - Lintao Zhang (now at Microsoft Research)
Outline

- Introduction
- Davis Putnam (DP)
  - Resolution based existential quantification
- Davis Logemann Loveland (DLL)
  - Search based algorithms
- Conflict driven learning (GRASP)
- Efficient deduction and branching (Chaff)
- Summary
**SAT in a Nutshell**

- Given a Boolean formula (propositional logic formula), find a variable assignment such that the formula evaluates to 1, or prove that no such assignment exists.
  
  $$F = (a + b)(a' + b' + c)$$

- For $n$ variables, there are $2^n$ possible truth assignments to be checked.

- First established NP-Complete problem.

Problem Representation

- Conjunctive Normal Form
  - $F = (a + b)(a' + b' + c)$
  - Simple representation (more efficient data structures)

- Logic circuit representation
  - Circuits have structural and direction information

- Circuit – CNF conversion is straightforward

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

\[
\begin{align*}
e &\equiv (c \cdot d) \\
(c' + d' + e) \\
(d + e') \\
(c + e')
\end{align*}
\]
Why Bother?

- Core computational engine for major applications
  - EDA
    - Testing and Verification
    - Logic synthesis
    - FPGA routing
    - Path delay analysis
    - And more…
  - AI
    - Knowledge base deduction
    - Automatic theorem proving
Outline

- Introduction
- Davis Putnam (DP)
  - Resolution based existential quantification
- Davis Logemann Loveland (DLL)
  - Search based algorithms
- Conflict driven learning (GRASP)
- Efficient deduction and branching (Chaff)
- Summary
Resolution

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

\[ a + b + c' + f \]
\[ g + h' + c + f \]
\[ a + b + g + h' + f \]
Davis Putnam Algorithm


- Existential abstraction using resolution
- Iteratively select a variable for resolution till no more variables are left.

\[
F = (a + b + c)(b + c' + f')(b' + e)
\]

\[
\exists b F = (a + c + e)(c' + e + f)
\]

\[
\exists bc F = (a + e + f)
\]

\[
\exists bcaef F = 1
\]

SAT

Potential memory explosion problem!

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

\[
\exists b F = (a)(a' + c)(a' + c')
\]

\[
\exists ba F = (c)(c')
\]

\[
\exists bac F = ()
\]

UNSAT
Outline

- Introduction
- Davis Putnam (DP)
  - Resolution based existential quantification
- Davis Logemann Loveland (DLL)
  - Search based algorithms
- Conflict driven learning (GRASP)
- Efficient deduction and branching (Chaff)
- Summary
DLL Algorithm

- Davis, Logemann and Loveland


- Also known as DPLL for historical reasons

- Basic framework for many modern SAT solvers
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)

⇐ Decision
Basic DLL Procedure - DFS

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

\[\leftarrow \text{Decision} \]
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)

Diagram:

- Node a connected to b with label 0.
- Node b connected to c with label 0.
- Node c has a decision label (⇒ Decision).

Nodes and labels:
- a
- b
- c

Decision point:

⇒ 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)

d=1
a=0

Implication Graph

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)

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)

⇐ Backtrack
Basic DLL Procedure - DFS

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

\[c=1\]
\[d=0\]
\[d=1\]
\[c=1\]

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)

 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!

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

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

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

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

\[
\begin{array}{c}
\text{Forced Decision}
\end{array}
\]
Basic DLL Procedure - DFS

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

Diagram:

- Node a with children b and c.
- Node b with children c.
- Node c with children 0 and 1.

Decision arrow pointing to 0.
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)

(b = 0)
(a' + b + c)

(c = 0)
(a' + b + c')

Conflict!
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)

Backtrack
Basic DLL Procedure - DFS

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

\(a=1\)
\(b=1\)
\(c=1\)

\(\iff \text{Forced Decision}\)
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=1
b=1
c=1
d=1

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

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

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

- 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.
  - a.k.a. Unit Propagation

- Workhorse of DLL based algorithms.
Features of DLL

- Eliminates the exponential memory requirements of DP
- Exponential time is still a problem
- Limited practical applicability – largest use seen in automatic theorem proving
- Very limited size of problems are allowed
  - 32K word memory
  - Problem size limited by total size of clauses (1300 clauses)
Outline

- Introduction
- Davis Putnam (DP)
  - Resolution based existential quantification
- Davis Logemann Loveland (DLL)
  - Search based algorithms
- Conflict driven learning (GRASP)
- Efficient deduction and branching (Chaff)
- Summary
GRASP

- Marques-Silva and Sakallah [SS96, SS99]
- Incorporates conflict driven learning and non-chronological backtracking
- Practical SAT instances can be solved in reasonable time
- Bayardo and Schrag’s ReISAT also proposed conflict driven learning [BS97]
Conflict Driven Learning and Non-chronological Backtracking

\[ x1 + x4 \]
\[ x1 + x3' + x8' \]
\[ x1 + x8 + x12 \]
\[ x2 + x11 \]
\[ x7' + x3' + x9 \]
\[ x7' + x8 + x9' \]
\[ x7 + x8 + x10' \]
\[ x7 + x10 + x12' \]
Conflict Driven Learning and Non-chronological Backtracking

\[ x_1 + x_4 \]
\[ x_1 + x_3' + x_8' \]
\[ x_1 + x_8 + x_{12} \]
\[ x_2 + x_{11} \]
\[ x_{7'} + x_3' + x_9 \]
\[ x_{7'} + x_8 + x_{9'} \]
\[ x_{7} + x_8 + x_{10'} \]
\[ x_{7} + x_{10} + x_{12'} \]

\[ x_1 = 0 \]
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

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

- \( x_1 = 0, x_4 = 1 \)
- \( x_3 = 1 \)
Conflict Driven Learning and Non-chronological Backtracking

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

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

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

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

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

- \( x_1 = 0, x_4 = 1 \)
- \( x_3 = 1, x_8 = 0, x_{12} = 1 \)
- \( x_2 = 0, x_{11} = 1 \)
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

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

$x_1 + x_4$
$x_1 + x_3' + x_8'$
$x_1 + x_8 + x_{12}$
$x_2 + x_{11}$
$x_7' + x_3' + x_9$
$x_7' + x_8 + x_9'$
$x_7 + x_8 + x_{10}'$
$x_7 + x_{10} + x_{12}'$

$x_1 = 0, x_4 = 1$
$x_3 = 1, x_8 = 0, x_{12} = 1$
$x_2 = 0, x_{11} = 1$
$x_7 = 1, x_9 = 1$

$x_3 = 1 \land x_7 = 1 \land x_8 = 0 \rightarrow \text{conflict}$
Conflict Driven Learning and Non-chronological Backtracking

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

Add conflict clause: \( x_3' + x_7' + x_8 \)

\( x_1=0, x_4=1 \)
\( x_3=1, x_8=0, x_{12}=1 \)
\( x_2=0, x_{11}=1 \)
\( x_7=1, x_9=1 \)

\( x_4=1 \)
\( x_3=1 \)
\( x_7=1 \)
\( x_9=1 \)
\( x_8=0 \)

\( x_{12}=1 \)
\( x_2=0 \)
\( x_{11}=1 \)

Add conflict clause: \( x_3' + x_7' + x_8 \)
Conflicts Driven Learning and Non-chronological Backtracking

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

Add conflict clause: \(x_3' + x_7' + x_8\)

\[
\begin{align*}
x_1 &= 0, x_4 = 1 \\
x_3 &= 1, x_8 = 0, x_{12} = 1 \\
x_2 &= 0, x_{11} = 1 \\
x_7 &= 1, x_9 = 1 \\
x_4 &= 1 \\
x_9 &= 0 \\
x_{12} &= 1 \\
x_2 &= 0 \\
x_{11} &= 1
\end{align*}
\]

\(x_3 = 1 \land x_7 = 1 \land x_8 = 0 \rightarrow \text{conflict}\)

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

\[ x_1 + x_4 \]
\[ x_1 + x_3' + x_8' \]
\[ x_1 + x_8 + x_{12} \]
\[ x_2 + x_{11} \]
\[ x_{7'} + x_3' + x_9 \]
\[ x_{7'} + x_8 + x_{9'} \]
\[ x_7 + x_8 + x_{10'} \]
\[ x_7 + x_{10} + x_{12'} \]
\[ x_3' + x_8 + x_{7'} \]

\textbf{Backtrack to the decision level of } x_3 = 1 \textbf{ With implication } x_7 = 0
What’s the big deal?

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

Useful in generating future conflict clauses.
Restart

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

Conflict clause: $x_1' + x_3 + x_5'$
SAT becomes practical!

- Conflict driven learning greatly increases the capacity of SAT solvers (several thousand variables) for structured problems
- Realistic applications became plausible
  - Usually thousands and even millions of variables
  - Typical EDA applications that can make use of SAT
    - circuit verification
    - FPGA routing
    - many other applications…
- Research direction changes towards more efficient implementations
Outline

● Introduction
● Davis Putnam (DP)
  ● Resolution based existential quantification
● Davis Logemann Loveland (DLL)
  ● Search based algorithms
● Conflict driven learning (GRASP)
● Efficient deduction and branching (Chaff)
● Summary
One to two orders of magnitude faster than other solvers…


- Widely Used:
  - Formal verification
    - Hardware and software
  - BlackBox – AI Planning
    - Henry Kautz (UW)
  - NuSMV – Symbolic Verification toolset
  - GrAnDe – Automatic theorem prover
  - Alloy – Software Model Analyzer at M.I.T.
  - haRVey – Refutation-based first-order logic theorem prover
  - Several industrial users – Intel, IBM, Microsoft, …
Large Example: Tough

- Industrial Processor Verification
  - Bounded Model Checking, 14 cycle behavior

- Statistics
  - 1 million variables
  - 10 million literals initially
    - 200 million literals including added clauses
    - 30 million literals finally
  - 4 million clauses (initially)
    - 200K clauses added
  - 1.5 million decisions
  - 3 hours run time
Chaff Philosophy

- Make the core operations fast
  - profiling driven, most time-consuming parts:
    - Boolean Constraint Propagation (BCP) and Decision
- Emphasis on coding efficiency and elegance
- Emphasis on optimizing data cache behavior
- As always, good search space pruning (i.e. conflict resolution and learning) is important

Recognition that this is as much a large (in-memory) database problem as it is a search problem.
Motivating Metrics: Decisions, Instructions, Cache Performance and Run Time

<table>
<thead>
<tr>
<th></th>
<th>1dlx_c_mc_ex_bp_f</th>
</tr>
</thead>
<tbody>
<tr>
<td>Num Variables</td>
<td>776</td>
</tr>
<tr>
<td>Num Clauses</td>
<td>3725</td>
</tr>
<tr>
<td>Num Literals</td>
<td>10045</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>zChaff</th>
<th>SATO</th>
<th>GRASP</th>
</tr>
</thead>
<tbody>
<tr>
<td># Decisions</td>
<td>3166</td>
<td>3771</td>
<td>1795</td>
</tr>
<tr>
<td># Instructions</td>
<td>86.6M</td>
<td>630.4M</td>
<td>1415.9M</td>
</tr>
<tr>
<td># L1/L2 accesses</td>
<td>24M / 1.7M</td>
<td>188M / 79M</td>
<td>416M / 153M</td>
</tr>
<tr>
<td>% L1/L2 misses</td>
<td>4.8% / 4.6%</td>
<td>36.8% / 9.7%</td>
<td>32.9% / 50.3%</td>
</tr>
<tr>
<td># Seconds</td>
<td>0.22</td>
<td>4.41</td>
<td>11.78</td>
</tr>
</tbody>
</table>
What “causes” an implication? When can it occur?

- All literals in a clause but one are assigned to False
  - \((v_1 + v_2 + v_3)\): 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 False
- 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: \((v_1 + v_2 + v_3 + v_4 + v_5)\)
  - \((v_1=X + v_2=X + v_3=? \text{ i.e. } X \text{ or } 0 \text{ or } 1) + v_4=? + v_5=?\)
BCP Algorithm (1.1/8)

- Big Invariants
  - Each clause has two watched literals.
  - If a clause can become unit via any sequence of assignments, then this sequence will include an assignment of one of the watched literals to F.
    - Example again: \((v1 + v2 + v3 + v4 + v5)\)
    - \((v1=\text{X} + v2=\text{X} + v3=? + v4=? + v5=? )\)

- BCP consists of identifying unit (and conflict) clauses (and the associated implications) while maintaining the “Big Invariants”
Let’s illustrate this with an example:

\[ v_2 + v_3 + v_1 + v_4 + v_5 \]
\[ v_1 + v_2 + v_3' \]
\[ v_1 + v_2' \]
\[ v_1' + v_4 \]
\[ v_1' \]
Let’s illustrate this with an example:

Initially, we identify any two literals in each clause as the watched ones.

- Clauses of size one are a special case.
- One literal clause breaks invariants: handled as a special case (ignored hereafter).
BCP Algorithm (3/8)

- We begin by processing the assignment $v_1 = F$ (which is implied by the size one clause)

<table>
<thead>
<tr>
<th>State: $(v_1=F)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pending:</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>$v_2 + v_3 + v_1 + v_4 + v_5$</td>
</tr>
<tr>
<td>$v_1 + v_2 + v_3'$</td>
</tr>
<tr>
<td>$v_1 + v_2'$</td>
</tr>
<tr>
<td>$v_1' + v_4$</td>
</tr>
</tbody>
</table>
We begin by processing the assignment $v_1 = F$ (which is implied by the size one clause).

To maintain our invariants, we must examine each clause where the assignment being processed has set a watched literal to $F$. 
We begin by processing the assignment $v_1 = F$ (which is implied by
the size one clause)

- To maintain our invariants, we must examine each clause where the
  assignment being processed has set a watched literal to $F$.
- We need not process clauses where a watched literal has been set to $T$,
  because the clause is now satisfied and so can not become unit.
BCP Algorithm (3.3/8)

- We begin by processing the assignment \( v_1 = F \) (which is implied by the size one clause)

\[
\begin{align*}
\text{State: } & (v_1=F) \\
\text{Pending: } & \quad v_2 + v_3 + v_1 + v_4 + v_5 \\
& \quad v_1 + v_2 + v_3' \\
& \quad v_1 + v_2' \\
& \quad v_1' + v_4
\end{align*}
\]

- To maintain our invariants, we must examine each clause where the assignment being processed has set a watched literal to F.
- We need not process clauses where a watched literal has been set to T, because the clause is now satisfied and so can not become unit.
- We *certainly* need not process any clauses where neither watched literal changes state (in this example, where \( v_1 \) is not watched).
BCP Algorithm (4/8)

- Now let’s actually process the second and third clauses:

\[
\begin{align*}
& v_2 + v_3 + v_1 + v_4 + v_5 \\
& v_1 + v_2 + v_3' \\
& v_1 + v_2' \\
& v_1' + v_4
\end{align*}
\]

State: \((v_1=F)\)
Pending:
Now let’s actually process the second and third clauses:

For the second clause, we replace \( v_1 \) with \( v_3' \) as a new watched literal. Since \( v_3' \) is not assigned to F, this maintains our invariants.
Now let’s actually process the second and third clauses:

For the second clause, we replace $v_1$ with $v_3'$ as a new watched literal. Since $v_3'$ is not assigned to $F$, this maintains our invariants.

The third clause is unit. We record the new implication of $v_2'$, and add it to the queue of assignments to process. Since the clause cannot again become unit, our invariants are maintained.
Next, we process v2’. We only examine the first 2 clauses.

For the first clause, we replace v2 with v4 as a new watched literal. Since v4 is not assigned to F, this maintains our invariants.

The second clause is unit. We record the new implication of v3’, and add it to the queue of assignments to process. Since the clause cannot again become unit, our invariants are maintained.
Next, we process \( v_3' \). We only examine the first clause.

For the first clause, we replace \( v_3 \) with \( v_5 \) as a new watched literal. Since \( v_5 \) is not assigned to \( F \), this maintains our invariants.

Since there are no pending assignments, and no conflict, BCP terminates and we make a decision. Both \( v_4 \) and \( v_5 \) are unassigned. Let’s say we decide to assign \( v_4 = T \) and proceed.
Next, we process v4. We do nothing at all.

Since there are no pending assignments, and no conflict, BCP terminates and we make a decision. Only v5 is unassigned. Let’s say we decide to assign v5=F and proceed.
Next, we process $v_5=F$. We examine the first clause.

- The first clause is already satisfied by $v_4$ so we ignore it.
- Since there are no pending assignments, and no conflict, BCP terminates and we make a decision. No variables are unassigned, so the instance is SAT, and we are done.

```
State: (v1=F, v2=F, v3=F, v4=T, v5=F)
```

```
State: (v1=F, v2=F, v3=F, v4=T, v5=F)
```


- The Invariants
  - Each clause has a head pointer and a tail pointer.
  - All literals in a clause before the head pointer and after the tail pointer have been assigned false.
  - If a clause can become unit via any sequence of assignments, then this sequence will include an assignment to one of the literals pointed to by the head/tail pointer.
Chaff vs. SATO: A Comparison of BCP

**Chaff:** \( v_1 + v_2' + v_4 + v_5 + v_8' + v_{10} + v_{12} + v_{15} \)

**SATO:** \( v_1 + v_2' + v_4 + v_5 + v_8' + v_{10} + v_{12} + v_{15} \)
Chaff vs. SATO: A Comparison of BCP

**Chaff:** \( v_1 + v_2' + v_4 + v_5 + v_8' + v_{10} + v_{12} + v_{15} \)

**SATO:** \( v_1 + v_2' + v_4 + v_5 + v_8' + v_{10} + v_{12} + v_{15} \)
Chaff vs. SATO: A Comparison of BCP

**Chaff:** \( v_1 + v_2' + v_4 + v_5 + v_8' + v_{10} + v_{12} + v_{15} \)

**SATO:** \( v_1 + v_2' + v_4 + v_5 + v_8' + v_{10} + v_{12} + v_{15} \)
Chaff vs. SATO: A Comparison of BCP

**Chaff:** $v_1 \ + \ v_2' \ + \ v_4 \ + \ v_5 \ + \ v_8' \ + \ v_{10} \ + \ v_{12} \ + \ v_{15}$

**SATO:** $v_1 \ + \ v_2' \ + \ v_4 \ + \ v_5 \ + \ v_8' \ + \ v_{10} \ + \ v_{12} \ + \ v_{15}$
Chaff vs. SATO: A Comparison of BCP

Chaff: \( v_1 + v_2' + v_4 + v_5 + v_8' + v_{10} + v_{12} + v_{15} \)

SATO: \( v_1 + v_2' + v_4 + v_5 + v_8' + v_{10} + v_{12} + v_{15} \)

Implication
Chaff vs. SATO: A Comparison of BCP

**Chaff:** \[ v_1 + v_2' + v_4 + v_5 + v_8' + v_{10} + v_{12} + v_{15} \]

**SATO:** \[ v_1 + v_2' + v_4 + v_5 + v_8' + v_{10} + v_{12} + v_{15} \]
Chaff vs. SATO: A Comparison of BCP

**Chaff:** \( v_1 + v_2' + v_4 + v_5 + v_8' + v_{10} + v_{12} + v_{15} \)

**SATO:** \( v_1 + v_2' + v_4 + v_5 + v_8' + v_{10} + v_{12} + v_{15} \)

Backtrack in Chaff
Chaff vs. SATO: A Comparison of BCP

Chaff: \( v_1 + v_2' + v_4 + v_5 + v_8' + v_{10} + v_{12} + v_{15} \)

Backtrack in SATO

SATO: \( v_1 + v_2' + v_4 + v_5 + v_8' + v_{10} + v_{12} + v_{15} \)
**BCP Algorithm Summary**

- **During forward progress: Decisions and Implications**
  - Only need to examine clauses where watched literal is set to F
    - Can ignore any assignments of literals to T
    - Can ignore any assignments to non-watched literals

- **During backtrack: Unwind Assignment Stack**
  - Any sequence of chronological unassignments will maintain our invariants
    - *So no action is required at all to unassign variables.*

- **Overall**
  - Minimize clause access
Decision Heuristics – Conventional Wisdom

- DLIS (Dynamic Largest Individual Sum) is a relatively simple dynamic decision heuristic
  - Simple and intuitive: At each decision simply choose the assignment that satisfies the most unsatisfied clauses.
  - However, considerable work is required to maintain the statistics necessary for this heuristic – for one implementation:
    - Must touch *every* clause that contains a literal that has been set to true. Often restricted to initial (not learned) clauses.
    - Maintain “sat” counters for each clause
    - When counters transition 0→1, update rankings.
    - Need to reverse the process for unassignment.
  - The total effort required for this and similar decision heuristics is *much more* than for our BCP algorithm.

- Look ahead algorithms even more compute intensive
Chaff Decision Heuristic - VSIDS

- Variable State Independent Decaying Sum
  - Rank variables by literal count in the initial clause database
  - Only increment counts as new clauses are added.
  - Periodically, divide all counts by a constant.
- Quasi-static:
  - Static because it doesn’t depend on variable state
  - Not static because it gradually changes as new clauses are added
    - Decay causes bias toward *recent* conflicts.
- Use heap to find unassigned variable with the highest ranking
  - Even single linear pass though variables on each decision would dominate run-time!
- Seems to work fairly well in terms of # decisions
  - hard to compare with other heuristics because they have too much overhead
Interplay of BCP and the Decision Heuristic

- This is only an intuitive description …
  - Reality depends heavily on specific instance
- Take some variable ranking (from the decision engine)
  - Assume several decisions are made
    - Say \( v_2 = \text{T}, v_7 = \text{F}, v_9 = \text{T}, v_1 = \text{T} \) (and any implications thereof)
  - Then a conflict is encountered that forces \( v_2 = \text{F} \)
    - The next decisions may still be \( v_7 = \text{F}, v_9 = \text{T}, v_1 = \text{T} \) !
      - VSIDS variable ranks change slowly…
    - But the BCP engine has recently processed these assignments …
      - so these variables are unlikely to still be watched.
- In a more general sense, the more “active” a variable is, the more likely it is to *not* be watched.
SAT Solver Competition!

SAT03 Competition
http://www.lri.fr/~simon/contest03/results/mainlive.php
34 solvers, 330 CPU days, 1000s of benchmarks

SAT04 Competition is going on right now …
Outline

- Introduction
- Davis Putnam (DP)
  - Resolution based existential quantification
- Davis Logemann Loveland (DLL)
  - Search based algorithms
- Conflict driven learning (GRASP)
- Efficient deduction and branching (Chaff)
- Summary
Summary

Lessons learnt:

- **Space vs. time tradeoffs**
  - DP vs. DLL
- **Efficient pruning is critical**
  - BCP and conflict driven learning
- **Efficient implementations are key**
  - Large database problem
    - Need to optimize memory operations
  - Focus on main operations
    - BCP and decision making