Satisfiability Solvers

Part 1: Systematic Solvers
SAT solving has made some progress…
SAT solving has made some progress…

“Whoa! until 1990, SAT speedups didn’t even keep pace with hardware speedups.”
Why is this wrong & misleading?

graphs thanks to Daniel Kroening and thocp.net
Exhaustive search

// Suppose formula uses 5 variables: A, B, C, D, E

- for A ∈ {false, true}
  - for B ∈ {false, true}
    - for C ∈ {false, true}
      - for D ∈ {false, true}
        - for E ∈ {false, true}
          if formula is true
            immediately return (A,B,C,D,E)
        return UNSAT
  return UNSAT
Exhaustive search

// Suppose formula uses 5 variables: A, B, C, D, E

■ for $A \in \{0, 1\}$
  ■ for $B \in \{0, 1\}$
    ■ for $C \in \{0, 1\}$
      ■ for $D \in \{0, 1\}$
        ■ for $E \in \{0, 1\}$
          if formula is true
            immediately return $(A, B, C, D, E)$
  ■ return UNSAT
Short-circuit evaluation

- for $A \in \{0, 1\}$
  - If formula is now definitely true, immediately return SAT
  - If formula is now definitely false, loop back & try next value of $A$
- for $B \in \{0, 1\}$
  - If formula is now def. true, immediately return SAT
  - If formula is now def. false, loop back & try next value of $B$
- for $C \in \{0, 1\}$
  - ...
  - for $D \in \{0, 1\}$
    - ...
    - for $E \in \{0, 1\}$
      - if formula is true
        - immediately return SAT
- return UNSAT
Declarative Methods

Short-circuit evaluation

\[(X \lor Y \lor Z) \land (\neg X \lor Y) \land (\neg Y \lor Z) \land (\neg X \lor \neg Y \lor \neg Z)\]

\[(x, y, z), (\neg x, y), (\neg y, z), (\neg x, \neg y, \neg z)\]

\[(\neg (x, y, z), (\neg x, y), (\neg y, z), (\neg x, \neg y, \neg z))\]

\[(\neg (x, y, z), (\neg x, y), (\neg y, z), (\neg x, \neg y, \neg z))\]

slide thanks to Daniel Kroening (modified)
Short-circuit evaluation

\[(X \lor Y \lor Z) \land (\neg X \lor Y) \land (\neg Y \lor Z) \land (\neg X \lor \neg Y \lor \neg Z)\]

\[(x,y,z),(-x,y),(-y,z),(-x,-y,-z)\]

Hmm, looks like we're about to get a contradiction
Short-circuit evaluation

\[(X \lor Y \lor Z) \land (~X \lor Y) \land (~Y \lor Z) \land (~X \lor ~Y \lor ~Z)\]

\[(x,y,z),(-x,y),(-y,z),(-x,-y,-z)\]

\[(y),(-y,z),(-y,-z)\]

\[(z),(-z)\]

\[\square\]

\[(y),(-y)\]

\[\square\]

\[(x),\]
Variable ordering might matter

- How do we pick the order A,B,C,D,E? And the order 0,1?
  - Any order is correct, but affects how quickly we can short-circuit.
- Suppose we have \( A \lor D \) among our clauses:
  - **Trying** \( A=0 \) forces \( D=1 \)
  - So after setting \( A=0 \), it would be best to consider \( D \) next
    - (Rule out \( D=0 \) **once**, not separately for all vals of \( B,C \))
- What if we **also** have \( \neg A \lor B \lor C \)?
  - **Trying** \( A=1 \) forces \( B=1 \) or \( C=1 \)
  - So after setting \( A=1 \), it might be good to consider \( B \) or \( C \) next
- **So, can we order** \( \{B,C,D,E\} \) **differently for** \( A=0 \) **and** \( A=1 \)?
  - What did we actually do on previous slide?

---

**Variable and value ordering** is an important topic.  
Hope to pick a satisfying assignment on the first try!  
We’ll come back to this ... many heuristics.
The most important variable ordering trick

“Unit propagation” or “Boolean constraint propagation”

- Suppose we try $A = 0$ ...
- Then all clauses containing $\neg A$ are satisfied and can be deleted.
- We must also remove $A$ from all clauses.
- Suppose one of those clauses is $(A \lor D)$. It becomes $(D)$, a “unit clause.”
- Now we know $D = 1$. Might as well deal with that right away.
- Chain reaction:
  - All clauses containing $D$ are satisfied and can be deleted.
  - We must also remove $\neg D$ from all clauses
  - Suppose we also have $(\neg D \lor C) \ldots$? It becomes $(C)$, a “unit clause.”
  - Now we know $C = 1$. Might as well deal with that right away.
  - Suppose we also have $(A \lor \neg C \lor \neg B) \ldots$? $A$ is already gone…
The most important variable ordering trick

“Unit propagation” or “Boolean constraint propagation”

- This leads to a “propagation” technique:
  - If we have any unit clause (1 literal only), it is a fact that lets us immediately shorten or eliminate other clauses.
  - What if we have more than one unit clause?
    - Deal with all of them, in any order
  - What if we have no unit clauses?
    - Can’t propagate facts any further.
    - We have to guess: pick an unassigned variable X and try both X=0, X=1 to make more progress.
    - This never happens on the LSAT exam.

- For satisfiable instances of 2-CNF-SAT, this finds a solution in $O(n)$ time with the right variable ordering (which can also be found in $O(n)$ time)!

Constraint propagation tries to eliminate future options as soon as possible, to avoid eliminating them repeatedly later.

We’ll see this idea again!
DLL algorithm (often called DPLL)

*Davis-(Putnam)-Logemann-Loveland*

- for each of $A$, $\neg A$
  - Add it to the formula and try doing unit propagation
  - If formula is now definitively true, immediately return SAT
  - If formula is now definitively false, abandon this iteration (loop back)
- for each of $B$, $\neg B$
  - Add it to the formula and try doing unit propagation
  - If formula is now definitively true, immediately return SAT
  - If formula is now definitively false, abandon this iteration (loop back)
- for each of $C$, $\neg C$
  - ...  
  - for $e$
    - If propagating $A$ already forced $B=0$? Then we have already established $\neg B$ and propagated it. So skip this.
- ...  
- return UNSAT
DLL algorithm (often called DPLL)

Cleaned-up version

Function DLL(\(\phi\)):

- while \(\phi\) contains at least one unit clause:
  - pick any unit clause \(X\)
  - remove all clauses containing \(X\)
  - remove \(\neg X\) from all remaining clauses

- if \(\phi\) now has no clauses left, return SAT
- else if \(\phi\) now contains an empty clause, return UNSAT
- else
  - pick any variable \(Z\) that still appears in \(\phi\)
  - if DLL(\(\phi \land Z\)) = SAT or DLL(\(\phi \land \neg Z\)) = SAT
    - then return SAT
  - else return UNSAT

How would you fix this to actually return a satisfying assignment?

Can we avoid the need to copy \(\phi\) on a recursive call?

Hint: assume the recursive call does so
Compare with the older DP algorithm

*Davis-Putnam*

- **DLL(φ)** recurses twice (unless we’re lucky and the first recursion succeeds):
  - If $\text{DLL}(\phi^X) = \text{SAT}$ or $\text{DLL}(\phi^{\sim X}) = \text{SAT}$ // for some X we picked
  - Adds unit clause X or $\sim X$, to be simplified out by unit propagation (along with all copies of X and $\sim X$) as soon as we recursively call DLL.

- **DP(φ)** tail-recurses once, by incorporating the “or” into the formula:
  - If $\text{DP}((\phi^X) \lor (\phi^{\sim X})) = \text{SAT}$ // for some X we picked
  - No branching: we tail-recurse once … on a formula with n-1 variables!
    - Done in n steps. We’ll see this “variable elimination” idea again…
  - *So what goes wrong?*
    - We have to put the argument into CNF first.
    - This procedure (resolution) eliminates all copies of X and $\sim X$. Let’s see how …
Compare with the older DP algorithm

Davis-Putnam

Resolution fuses each pair \((V v W v \neg X)^\land (X v Y v Z)\) into \((V v W v Y v Z)\)

Justification #1: Valid way to eliminate \(X\) (reverses CNF \(\rightarrow\) 3-CNF idea).

Justification #2: Want to recurse on a CNF version of \(((\varphi^X) v (\varphi^\neg X))\)

Suppose \(\varphi = \alpha^\land \beta^\land \gamma\)

where \(\alpha\) is clauses with \(\neg X\), \(\beta\) with \(X\), \(\gamma\) with neither

Then \(((\varphi^X) v (\varphi^\neg X)) = (\alpha'^\land \gamma) v (\beta'^\land \gamma)\) by unit propagation

where \(\alpha'\) is \(\alpha\) with the \(\neg X\)'s removed, \(\beta'\) similarly.

\(= (\alpha' v \beta')^\land \gamma = (\alpha'_1 v \beta'_1)^\land (\alpha'_1 v \beta'_2)^\land \ldots^\land (\alpha'_{99} v \beta'_{99})^\land \gamma\)

- if \(\text{DP}(((\varphi^X) v (\varphi^\neg X)) = \text{SAT}\)  // for some \(X\) we picked

- No branching: we tail-recurse once … on a formula with \(n-1\) variables!

- But we have to put the argument into CNF first.

- This procedure (resolution) eliminates all copies of \(X\) and \(\neg X\).
  - Done in \(n\) steps. So what goes wrong?
  - Size of formula can square at each step.
  - (Also, not so obvious how to recover the satisfying assignment.)
Basic DLL Procedure

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

slide thanks to Sharad Malik (modified)
Basic DLL Procedure

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

slide thanks to Sharad Malik (modified)
Basic DLL Procedure

Green means “crossed out”

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

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

\[\text{Decision} \iff 0\]
Basic DLL Procedure

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

Decision slide thanks to Sharad Malik (modified)
Basic DLL Procedure

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

Unit clauses force both d=1 and d=0: contradiction

Implication Graph
(shows that the problem was caused by a=0 ^ c=0; nothing to do with b)

Conflict!

slide thanks to Sharad Malik (modified)
Basic DLL Procedure

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

Slide thanks to Sharad Malik (modified)
Basic DLL Procedure

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

\[
\begin{align*}
a &= 0 \\
c &= 1 \\
d &= 0
\end{align*}
\]

Conflict!

Other Decision

slide thanks to Sharad Malik (modified)
Basic DLL Procedure

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

\(\Leftarrow\) Backtrack (2 levels)
Basic DLL Procedure

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

Other Decision

slide thanks to Sharad Malik (modified)
Basic DLL Procedure

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

\[c = 0\]
\[d = 1\]
\[\text{Conflict!}\]

\[\leftarrow \text{Decision}\]
Basic DLL Procedure

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

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

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

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

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

Slide thanks to Sharad Malik (modified)
Basic DLL Procedure

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

Slide thanks to Sharad Malik (modified)
Basic DLL Procedure

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

\(\iff\) Other Decision
Basic DLL Procedure

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

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

\(\text{Again choose } b \text{ next (not required)}\)
\(\text{Decision}\)
Basic DLL Procedure

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

Conflict!

slide thanks to Sharad Malik (modified)
Basic DLL Procedure

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

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

unit clause that propagates without contradiction (finally!)
Often you get these much sooner

slide thanks to Sharad Malik (modified)
Basic DLL Procedure

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

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

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

\[\text{Forced by unit clause}\]
\[\iff \text{SAT}\]
Tricks used by zChaff and similar DLL solvers
(Overview only; details on later slides)

- **Make unit propagation / backtracking speedy** (80% of the cycles!)
- **Variable ordering heuristics:** Which variable/value to assign next?
- **Conflict analysis:** When a contradiction is found, analyze what subset of the assigned variables was responsible. Why?
  - **Better heuristics:** Like to branch on vars that caused recent conflicts
  - **Backjumping:** When backtracking, avoid trying options that would just lead to the same contradictions again.
  - **Clause learning:** Add new clauses to block bad sub-assignments.
  - **Random restarts** (maybe): Occasionally restart from scratch, but keep using the learned clauses. (Example: crosswords ...)
    - Even without clause learning, random restarts can help by abandoning an unlucky, slow variable ordering. Just break ties differently next time.
- **Preprocess** the input formula (maybe)
- **Tuned implementation:** Carefully tune data structures
  - improve memory locality and avoid cache misses
### Motivating Metrics: Decisions, Instructions, Cache Performance and Run Time

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

<table>
<thead>
<tr>
<th></th>
<th>Z-Chaff</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>

*slide thanks to Moskewicz, Madigan, Zhang, Zhao, & Malik*
## DLL: Obvious data structures

Current variable assignments

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>J</th>
<th>K</th>
<th>L</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Stack of assignments used for backtracking

- C=1
- F=0
- A=1
- G=0

- = forced by propagation
- = first guess
- = second guess
### DLL: Obvious data structures

**Current variable assignments**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>J</th>
<th>K</th>
<th>L</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Stack of assignments used for backtracking**

<table>
<thead>
<tr>
<th>C=1</th>
<th>F=0</th>
<th>A=1</th>
<th>G=0</th>
<th>J=0</th>
</tr>
</thead>
</table>

*Guess a new assignment* J=0

- **= forced by propagation**
- **= first guess**
- **= second guess**
DLL: Obvious data structures

Current variable assignments

<p>| | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>M</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Stack of assignments used for backtracking

C=1  F=0  A=1  G=0  J=0  K=1  L=1

Unit propagation implies assignments K=1, L=1

- = forced by propagation
- = first guess
- = second guess
- = currently being propagated
- = assignment still pending
DLL: Obvious data structures

Current variable assignments

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>J</th>
<th>K</th>
<th>L</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Stack of assignments used for backtracking

| C=1 | F=0 | A=1 | G=0 | J=0 | K=1 | L=1 |

Now make those assignments, one at a time

= forced by propagation
= first guess
= second guess
= currently being propagated
= assignment still pending
DLL: Obvious data structures

Current variable assignments

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>J</th>
<th>K</th>
<th>L</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Stack of assignments used for backtracking

- **C=1**
- **F=0**
- **A=1**
- **G=0**
- **J=0**
- **K=1**
- **L=1**
- **B=0**

Chain reaction: K=1 propagates to imply B=0

- **= forced by propagation**
- **= first guess**
- **= second guess**
- **= currently being propagated**
- **= assignment still pending**
DLL: Obvious data structures

Current variable assignments

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>J</th>
<th>K</th>
<th>L</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Stack of assignments used for backtracking

| C=1 | F=0 | A=1 | G=0 | J=0 | K=1 | L=1 | B=0 |

Also implies A=1, but we already knew that

- = forced by propagation
- = first guess
- = second guess
- = currently being propagated
- = assignment still pending
### DLL: Obvious data structures

#### Current variable assignments

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>J</th>
<th>K</th>
<th>L</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

#### Stack of assignments used for backtracking

- **C=1**
- **F=0**
- **A=1**
- **G=0**
- **J=0**
- **K=1**
- **L=1**
- **B=0**

- **= forced by propagation**
- **= first guess**
- **= second guess**
- **= currently being propagated**
- **= assignment still pending**
# DLL: Obvious data structures

## Current variable assignments

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>J</th>
<th>K</th>
<th>L</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## Stack of assignments used for backtracking

- **C=1**
- **F=0**
- **A=1**
- **G=0**
- **J=0**
- **K=1**
- **L=1**
- **B=0**
- **F=1**

*Oops!*

L=1 propagates to imply F=1, but we already had F=0

- **= forced by propagation**
- **= first guess**
- **= second guess**
- **= currently being propagated**
- **= assignment still pending**
DLL: Obvious data structures

Current variable assignments

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>J</th>
<th>K</th>
<th>L</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Stack of assignments used for backtracking

| C=1 | F=0 | A=1 | G=0 | J=0 | K=1 | L=1 | B=0 | F=1 |

Backtrack to last yellow, undoing all assignments

- = forced by propagation
- = first guess
- = second guess
- = currently being propagated
- = assignment still pending

Oops!
DLL: Obvious data structures

Current variable assignments

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>J</th>
<th>K</th>
<th>L</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Stack of assignments used for backtracking

| C=1 | F=0 | A=1 | G=0 | J=0 |   |   |   |   |   |

- = forced by propagation
- = first guess
- = second guess
- = currently being propagated
- = assignment still pending
## DLL: Obvious data structures

### Current variable assignments

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>J</th>
<th>K</th>
<th>L</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Stack of assignments used for backtracking

- **C=1**
- **F=0**
- **A=1**
- **G=0**
- **J=1**

- J=0 didn’t work out, so try J=1

- = forced by propagation
- = first guess
- = second guess
- = currently being propagated
- = assignment still pending
### DLL: Obvious data structures

Current variable assignments

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>J</th>
<th>K</th>
<th>L</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Stack of assignments used for backtracking

- **C=1**
- **F=0**
- **A=1**
- **G=0**
- **J=1**
- **B=0**

- = forced by propagation
- = first guess
- = second guess
- = currently being propagated
- = assignment still pending
DLL: Obvious data structures

Current variable assignments

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>J</th>
<th>K</th>
<th>L</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Stack of assignments used for backtracking

C=1  F=0  A=1  G=0  J=1  B=0

Nothing left to propagate. Now what?

- = forced by propagation
- = first guess
- = second guess
- = currently being propagated
- = assignment still pending
DLL: Obvious data structures

Current variable assignments

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>J</th>
<th>K</th>
<th>L</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td></td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Stack of assignments used for backtracking

<table>
<thead>
<tr>
<th></th>
<th>C=1</th>
<th>F=0</th>
<th>A=1</th>
<th>G=0</th>
<th>J=1</th>
<th>B=0</th>
<th>L=1</th>
<th></th>
</tr>
</thead>
</table>

Again, guess an unassigned variable and proceed ...

- = forced by propagation
- = first guess
- = second guess
- = currently being propagated
- = assignment still pending
DLL: Obvious data structures

Current variable assignments

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>J</th>
<th>K</th>
<th>L</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Stack of assignments used for backtracking

| C=1 | F=0 | A=1 | G=0 | J=1 | B=0 | L=0 | ... |

If L=1 doesn’t work out, we know L=0 in this context

- = forced by propagation
- = first guess
- = second guess
- = currently being propagated
- = assignment still pending
## DLL: Obvious data structures

### Current variable assignments

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>J</th>
<th>K</th>
<th>L</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

### Stack of assignments used for backtracking

<table>
<thead>
<tr>
<th>C=1</th>
<th>F=0</th>
<th>A=1</th>
<th>G=0</th>
<th>J=1</th>
<th>B=0</th>
<th>L=0</th>
<th>...</th>
</tr>
</thead>
</table>

If L=0 doesn’t work out either, backtrack to ...?

- = forced by propagation
- = first guess
- = second guess
- = currently being propagated
- = assignment still pending
Question: When should we return SAT or UNSAT?

- = forced by propagation
- = first guess
- = second guess
- = currently being propagated
- = assignment still pending
How to speed up unit propagation?
(with a million large clauses to keep track of)

Every step in DLL is fast, except propagation:

Objective: When a variable is assigned to 0 or 1, detect which clauses become unit clauses.

- Obvious strategy: “crossing out” as in previous slides. Too slow, especially since you have to un-cross-out when backtracking.
- Better: Don’t modify or delete clauses. Just search for k-clauses with \( k-1 \) currently false literals & 1 currently unassigned literal.
- But linear search of all the clauses is too slow.

Sounds like grid method!
How to speed up unit propagation?
(with a million large clauses to keep track of)

- Find length-k clauses with k-1 false literals & 1 unassigned literal.

- **Index** the clauses for fast lookup:
  - Every literal (A or ~A) maintains a list of clauses it’s in
  - If literal becomes false, only check if those clauses became unit
  - Could use **counters** so that checking each clause is fast:
    - Every clause remembers how many non-false literals it has
    - If this counter ever gets to 1, we might have a new unit clause
      - Scan clause to find the remaining non-false literal
      - It’s either true, or unassigned, in which case we assign it true!
  - When variable A is assigned, either A or ~A becomes false
    - Increment counters of all clauses containing the newly false literal
    - When undoing the assignment on backtracking, decrement counters

Too many clauses to visit!
(worse, a lot of memory access)
How to speed up unit propagation?
(with a million large clauses to keep track of)

- Find length-k clauses with \( k-1 \) false literals & 1 unassigned literal.
  - When variable A is assigned, either A or \( \sim A \) becomes false
    - Decrement counters of all clauses containing the newly false literal
    - Clause only becomes unit when its counter reaches 1

- **Hope:** Don’t keep visiting clause just to adjust its counter.
  - So, can’t afford to keep a counter.
  - Visit clause only when it’s really in danger of becoming unit.

- **Insight:** A clause with at least 2 non-false literals is safe.
  - So pick any 2 non-false literals (true/unassigned), and watch them.
  - As long as both stay non-false, don’t have to visit the clause at all!

- **Plan:** Every literal maintains a list of clauses in which it’s watched.
  - If it becomes false, we go check those clauses (only).
Chaff/zChaff’s unit propagation algorithm

How about clauses with only one literal?

Always watch first 2 literals in each clause

example thanks to Moskewicz, Madigan, Zhang, Zhao, & Malik
Chaff/zChaff’s unit propagation algorithm

A = 0

Invariant: Keep false vars out of these positions as long as possible.
Why? Watched positions must be the last ones to become false - so that we’ll notice when that happens!
A became false. It is watched only here. Look to see if either of these 2 clauses became unit.

example thanks to Moskewicz, Madigan, Zhang, Zhao, & Malik
Chaff/zChaff’s unit propagation algorithm

A became false. It is watched only here. Look to see if either of these 2 clauses became unit.

(A B \neg C) is now (0 ? ?), so not unit yet. To keep 0 vars out of watched positions, swap A with \neg C. (So if \neg C is the last to become false, giving a unit clause, we’ll notice.)
Chaff/zChaff’s unit propagation algorithm

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A became false. It is watched only here. Look to see if either of these 2 clauses became unit.

\[(A \neg B)\] is now \((0 ?)\), so unit. We find that \(\neg B\) is the unique \(?\) variable, so we must make it true.
Chaff/zChaff’s unit propagation algorithm

\[
\begin{array}{ccccc}
A & B & C & D & E \\
0 & 0 & & & \\
\end{array}
\]

\[
(\sim C \ B \ A)
\]

\[
(\sim A \ D)
\]

\[
(\ B \ C \ A \ D \ E )
\]

\[
(A \ B)
\]

B became false. It is watched only here. Look to see if either of these 2 clauses became unit.

example thanks to Moskewicz, Madigan, Zhang, Zhao, & Malik
Chaff/zChaff’s unit propagation algorithm

\[
\begin{array}{ccccc}
A & B & C & D & E \\
0 & 0 & & & \\
\end{array}
\]

B became false. It is watched only here. Look to see if either of these 2 clauses became unit.

(B C A D E) is now (0 ? 0 ? ?), so not unit yet. To keep 0 vars out of watched positions, swap B with D. (So if D is the last to become false, giving a unit clause, we’ll notice.)

element thanks to Moskewicz, Madigan, Zhang, Zhao, & Malik
Chaff/zChaff’s unit propagation algorithm

\[
\begin{array}{cccc}
A & B & C & D & E \\
0 & 0 & & & \\
\end{array}
\]

\[
\begin{array}{c}
(A ~B) \\
(\sim A ~D) \\
(\sim C B A) \\
(D C A B E) \\
\end{array}
\]

B became false. It is watched only here. Look to see if either of these 2 clauses became unit.

\((\sim C B A)\) is now \((? 0 0)\), so unit. We find that \(\sim C\) is the unique ? variable, so we must make it true.

text

example thanks to Moskewicz, Madigan, Zhang, Zhao, & Malik
### Chaff/zChaff's unit propagation algorithm

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

C became false. It is watched only here. Look to see if this clause became unit.

\[(D \lor C \lor A \lor B \lor E)\] is now \(\land 0 \land 0 \land 0 \land ?\), so not unit yet.
To keep 0 vars out of watched positions, swap C with E.
(\text{So if E is the last to become false, giving a unit clause, we'll notice.})

---

*example thanks to Moskewicz, Madigan, Zhang, Zhao, & Malik*
Chaff/zChaff’s unit propagation algorithm

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

\[ (D \quad E \quad A \quad B \quad C) \]
\[ (\sim C \quad B \quad A) \]
\[ (A \quad \sim B) \]
\[ (\sim A \quad D) \]

We decided to set D true.
So \( \sim D \) became false … but it’s not watched anywhere, so nothing to do.

(First clause became satisfied as \( (1 \ ? \ 0 \ 0 \ 0) \), but we haven’t noticed yet. In fact, we haven’t noticed that all clauses are now satisfied!)

example thanks to Moskewicz, Madigan, Zhang, Zhao, & Malik
Chaff/zChaff’s unit propagation algorithm

We decided to set E false. It is watched only here. Look to see if this clause became unit.

(D E A B C) is now (1 0 0 0 0). This is not a unit clause. (It is satisfied because of the 1.)

All variables have now been assigned without creating any conflicts, so we are SAT.
Chaff/zChaff’s unit propagation algorithm

Why the technique is fast:

- Assigning 0 to a watched literal takes work
  - Have to check for unit clause, just in case
  - Must try to move the 0 out of watched position

- But everything else costs nothing …
  - Assigning 1 to a watched literal
  - Unassigning a watched literal (whether it’s 0 or 1)
  - Doing anything to an unwatched literal.

\[
\begin{array}{cccccc}
A & B & C & D & E \\
0 & 0 & 0 & & \\
\end{array}
\]

\[(D, C, A, B, E)\]

\[(\sim C, B, A)\]

\[(A, \sim B)\]

\[(\sim A, D)\]

\begin{tabular}{|c|c|c|}
\hline
A=0 & B=0 & C=0 \\
\hline
\end{tabular}

example thanks to Moskewicz, Madigan, Zhang, Zhao, & Malik
### Chaff/zChaff’s unit propagation algorithm

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Why it’s even faster than you realized:**
  - Assigning 0 to a watched literal takes work
    - But then we’ll move it out of watched position if we can!
    - So **next** time we try to assign 0 to the same literal, it costs nothing!
  - This is very common: backtrack to change B, then retry C=0

- **Deep analysis:** Suppose a clause is far from becoming a unit clause (it has a few true or unassigned vars). Then we shouldn’t waste much time repeatedly checking whether it’s become unit. And this algorithm doesn’t: Currently “active” variables get swapped out of the watch positions, in favor of “stable” vars that are true/unassigned. “Active” vars tend to exist because we spend most of our time shimmying locally in the search tree in an inner loop, trying many assignments for the last few decision variables that lead to a conflict.

---

**example thanks to Moskewicz, Madigan, Zhang, Zhao, & Malik**
There is a bit of overhead.

Each literal maintains an array of clauses watching it.

When C becomes 0, we iterate through its array of clauses:

- We scan clause 1: discover it’s not unit, but we must swap C, E.
- So take clause 1 off C’s watch list and add it to E’s watch list.
- Not hard to make this fast (see how?)

example thanks to Moskewicz, Madigan, Zhang, Zhao, & Malik
Big trick #2: Conflict analysis

- When a contradiction is found, analyze what subset of the assigned variables was responsible. Why?
  - **Backjumping**: When backtracking, avoid trying options that would just lead to the same contradictions again.
  - **Clause learning**: Add new clauses to block bad sub-assignments.
  - **Random restarts**: Occasionally restart from scratch, but keep using the learned clauses (or try a new variable ordering).
  - **Better heuristics**: Like to branch on vars that caused recent conflicts

Each var records its decision level

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>J</th>
<th>K</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td></td>
<td>0</td>
<td>0</td>
<td></td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>

Decision levels

<table>
<thead>
<tr>
<th></th>
<th>C=1</th>
<th>F=0</th>
<th>A=1</th>
<th>G=0</th>
<th>J=0</th>
<th>K=1</th>
<th>L=1</th>
<th>B=0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Big trick #2: Conflict analysis

- Each var also remembers **why** it became 0 or 1
  - A yellow (or red) var remembers it was a decision variable
  - A white var was set by unit propagation
    - Remembers which clause was responsible
    - That is, points to the clause that became a unit clause & set it

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>J</th>
<th>K</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Each var records its decision level

<table>
<thead>
<tr>
<th>Decision levels</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C=1</th>
<th>F=0</th>
<th>A=1</th>
<th>G=0</th>
<th>J=0</th>
<th>K=1</th>
<th>L=1</th>
<th>B=0</th>
</tr>
</thead>
</table>
Can regard those data structures as an “implication graph”

E.g., remember, variable $V_{10}$ stores a pointer saying why $V_{10} = 0$: because $(V_{4} + V_{2} + V_{10})$ became a unit clause at level 5.

In other words, $V_{10} = 0$ because $V_{4} = 1$ and $V_{2} = 0$: we draw this!
Can regard those data structures as an “implication graph”

When we get a conflict, build graph by tracing back from V18, \(~V18\)

Decision Variable

Variable assigned at previous d-level

Variable assigned at current d-level

slide thanks to Zhang, Madigan, Moskewicz, & Malik (modified)
Which decisions triggered the conflict?
(in this case, only decisions 1,2,3,5 – not 4)
Which decisions triggered the conflict?

*(in this case, only decisions 1,2,3,5 – not 4)*

- Our choices at decision levels 1,2,3,5 were jointly responsible
- So decision level 4 was **not** responsible
  - Neither decision variable 4, nor any vars set by propagation from it
- Suppose we now backtrack from decision 5
  - We just found =1 led to conflict; suppose we found =0 did too
  - And suppose level 4 wasn’t responsible **in either case**
  - Then we can “backjump” over level 4 back to level 3
    - Trying other value of decision variable 4 can’t avoid conflict!
    - Also called “non-chronological backtracking”
    - Should now copy conflict information up to level 3: all the choices at levels 1,2,3 were responsible for this conflict below level 3
A possible clause to learn

*(avoid 5-variable combination that will always trigger this V₁₈ conflict)*

\[
\sim (V₁ \wedge V₃ \wedge V₅ \wedge V₁₇ \wedge V₁₉) = (V₁' \vee V₃' \vee V₅' \vee V₁₇' \vee V₁₉')
\]

Gives us one step of lookahead that would have avoided this conflict
A different clause we could learn

(would have caught the problem sooner: but why not as great an idea?)

The learned clause says “don’t have this particular combination of decision variables.” But we’re about to backtrack and change that combination anyway - so the learned clause won’t be useful for a while.
A different clause we could learn
(would have caught the problem sooner)

Learned clause has only one blue variable, namely $V_2$
(i.e., only one variable that was set by propagation at level 5)
$V_2$ is a “unique implication point”
(so is $V_{10}$, but we prefer the earliest unique implication point)

Our next step is to backtrack, undoing level 5. This undoes the propagation that set $V_2=0$.
Now back at level 4, propagation kicks in on our learned clause, forcing $V_2=1$ (and maybe more!).
Variable ordering heuristics

- Function DLL(\(\phi\)):
  - do unit propagation
  - if we got a conflict, return UNSAT
  - else if all variables are assigned, return SAT
  - else
    - pick an unassigned variable X
    - if DLL(\(\phi \land X\)) = SAT or DLL(\(\phi \land \neg X\)) = SAT
    - then return SAT else return UNSAT

- How do we choose the next variable \(X\)? (variable ordering)
  - And how do we choose whether to try \(X\) or \(\neg X\) first? (value ordering)
- Would like choice to lead quickly to a satisfying assignment …
- … or else lead quickly to a conflict so we can backtrack.
- Lots of heuristics have been tried!
Heuristic: Most Constrained First

- Pack suits/dresses before toothbrush
  - First try possibilities for highly constrained variables
  - Hope the rest fall easily into place

First find an assignment for the most constrained variables ABC (often hard, much backtracking)

If that succeeds, may be easy to extend it into a full assignment: other vars WXY fall easily into place (e.g., by unit propagation from vars already chosen, or because either W=0 or W=1 could lead to success)
**Heuristic: Most Constrained First**

- Pack suits/dresses before toothbrush
  - First try possibilities for highly constrained variables
  - Hope the rest fall easily into place

**Diagram:**

- Good: solve for most constrained first
- Bad: start with least constrained. All assignments to WXY start out looking good, but then we usually can’t extend them.
Heuristic: Most Constrained First

- **So, how do we guess which vars are “most constrained”?**
- Which variable appears in the most clauses?
- **Wait:** shorter clauses are stronger constraints
  - (A B) rules out 1/4 of all solutions
  - (A B C D E) only rules out 1/32 of all solutions; doesn’t mean A is especially likely to be true
- **Which variable appears in the most “short clauses”?**
  - E.g., consider clauses only of minimum length
- **As we assign variables, “crossing out” will shorten or eliminate clauses. Should we consider that when picking next variable?**
  - “Dynamic” variable ordering - depends on assignment so far
  - Versus “fixed” ordering based on original clauses of problem
  - Dynamic can be helpful, but you pay a big price in bookkeeping

slide thanks to Ian Gent (modified)
Heuristic: Most satisfying first
(Jeroslow-Wang heuristic for variable and value ordering)

- Greedily satisfy (eliminate) as many clauses as we can
  - Which literal appears in the most clauses?
  - If X appears in many clauses, try X=1 to eliminate them
  - If ~X appears in many clauses, try X=0 to eliminate them

- Try especially to satisfy hard (short) clauses
  - When counting clauses that contain X,
    - length-2 clause counts twice as much as a length-3 clause
    - length-3 clause counts twice as much as a length-4 clause
  - In general, let a length-i clause have weight $2^{-i}$
    - Because it rules out $2^{-i}$ of the $2^{-n}$ possible assignments
Heuristic: Most simplifying first
(again, does variable and value ordering)

- We want to simplify problem as much as possible
  - I.e. get biggest possible cascade of unit propagation
  - Motivation: search is exponential in the size of the problem
    so making the problem small quickly minimizes search

- One approach is to try it and see
  - make an assignment, see how much unit propagation occurs
  - after testing all assignments, choose the one which caused the biggest cascade
  - exhaustive version is expensive (2n probes necessary)
  - Successful variants probe a small number of promising variables (e.g. from the “most-constrained” heuristic)
Heuristic: What does zChaff use?

- Use variables that appeared in many “recent” learned clauses or conflict clauses
  - Keep a count of appearances for each variable
  - Periodically divide all counts by a constant, to favor recent appearances

- Really a subtler kind of “most constrained” heuristic
  - Look for the “active variables” that are currently hard to get right

- Definitely a dynamic ordering … but fast to maintain
  - Only update it once per conflict, not once per assignment
  - Only examine one clause to do the update, not lots of them
Random restarts

- Sometimes it makes sense to keep restarting the search, with a different variable ordering each time
  - Avoids the very long run-times of unlucky variable ordering
  - On many problems, yields faster algorithms

- Why would we get a different variable ordering?
  - We break ties randomly in the ordering heuristic
    - Or could add some other kind of randomization to the heuristic
  - Clauses learned can be carried over across restarts
    - So after restarting, we’re actually solving a different formula