Satisfiability Solvers

Part 1: Systematic Solvers

SAT solving has made some progress...

Exhaustive search

Exhaustive search

Short-circuit evaluation

How would we tell? When would these cases happen?
Short-circuit evaluation

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

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

slide thanks to Daniel Kroenin (modified)

Short-circuit evaluation

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

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

slide thanks to Daniel Kroenin (modified)

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 v 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 ~A v B v 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
    - What did we actually do on previous slide?
      - 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 ~A are satisfied and can be deleted.
  - We must also remove A from all clauses.
  - Suppose one of those clauses is (A v 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 ~D from all clauses.
  - Suppose we also have (~D v C) ...
  - It becomes (C), a "unit clause."
  - Now we know C=1. Might as well deal with that right away.
  - Suppose we also have (A v C) v B ...

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-Loveland**

- for each of $A, \neg A$
  - Add it to the formula and try doing unit propagation
    - If formula is now true, immediately return SAT
    - If formula is now 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 true, immediately return SAT
    - If formula is now false, abandon this iteration (loop back)
- for each of $C, \neg C$
  - What if we want to choose $C$?  What if propagating $A$ already forced $B=0$?  If so, skip this.
- return UNSAT

### Compare with the older DP algorithm

**Davis-Putnam**

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

- **DP($\phi$)** tail-recurses once, by incorporating the “or” into the formula:
  - If DP($\phi \land \neg X$) = 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 $\neg X$. Let's see how...

---

### Basic DLL Procedure

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

---

### 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 $X$ that still appears in $\phi$
    - if DLL($\phi \land X$) = SAT or DLL($\phi \land \neg X$) = 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 $X$ and $\neg X$ as soon as we recursively call DLL.

---

### 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

Green means "crossed out"

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

0 = Decision

Backtrack

Basic DLL Procedure

Unit clauses force both \(d = 1\) and \(d = 0\) contradictory:

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

Other Decision

Conflict!
Basic DLL Procedure

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

\[b\]
\[0\]
\[1\]
\[c\]
\[0\]
\[1\]
\[d\]
\[0\]
\[1\]

\[\text{Backtrack (2 levels)}\]

\[\text{Decision}\]

\[\text{Conflict!}\]

\[\text{Other Decision}\]

\[\text{Conflict!}\]

\[\text{Backtrack}\]

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

\[(a + b + c)\]
\[(a + c + d)\]
\[(a + c' + d)\]
\[(a + b + c')\]
\[= \text{Other Decision}\]

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

Unit clause that propagates without contradiction (Finally)! Often you get these much sooner

\[(c' + d')\]
\[= \text{Conflict!}\]

\[(a' + b + c)\]
\[= \text{Forced by unit clause}\]

slide thanks to Sharad Malik (modified)
Basic DLL Procedure

\[
\begin{align*}
(a' + b' + c) & = 1 \\
(a + c + d) & = 1 \\
(a' + b + c) & = 1 \\
(a + c' + d) & = 1 \\
(a' + b + c') & = 1 \\
(b' + c' + d) & = 1 \\
\end{align*}
\]

Forced by unit clause

SAT

Tricks used by zChaff and similar DLL solvers

- 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
- Preprocess
- 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>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>

DLL: Obvious data structures

Current variable assignments

\[
\begin{array}{cccccccccccc}
A & B & C & D & E & F & G & H & I & J & K & L & M \\
1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]

Stack of assignments used for backtracking

\[
\begin{array}{cccccccccccc}
C & = 1 & F & = 0 & A & = 1 & G & = 0 & J & = 0 \\
\end{array}
\]

Guess a new assignment J=0

- forced by propagation
- first guess
- second guess

Unit propagation implies assignments K=1, L=1

- forced by propagation
- first guess
- second guess
- 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>0</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

Now make those assignments, one at a time

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

Also implies A=1, but we already knew that

Oops!

Backtrack to last yellow, undoing all assignments
**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>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Stack of assignments used for backtracking

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

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

Nothing left to propagate. Now what?

Again, guess on unassigned variable and proceed ...

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

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 literals
  - 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
- **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).

How to speed up unit propagation?
(with a million large clauses to keep track of)

**Chaff/zChaff’s unit propagation algorithm**

- Find length-k clauses with k-1 false literals & 1 unassigned literal.
  - When variable A is assigned, either A or ~A becomes false
    - Decrement counters of all clauses containing the newly false literal
    - Clause only becomes unit when its counter reaches 1
  - Too many clauses to visit!

- **Sequence**: Always watch first 2 literals in each clause

- **Insight**: If any clause has only one literal left that can be watched
  - **Watch each clause only once**

- **Example**: thanks to Moskewicz, Madigan, Zhang, Zhao & Malik
Chaff/zChaff's unit propagation algorithm

Always watch first 2 literals in each clause.
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.

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

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 ? ? ?), 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.)
Chaff/zChaff’s unit propagation algorithm

\[
\begin{array}{cccccc}
A & B & C & D & E \\
0 & 0 & 1 &  &  \\
\end{array}
\]

\[
\begin{array}{ccccccc}
A & B & C & D & E \\
1 & 0 & 0 & 0 &  \\
\end{array}
\]

\[
\begin{array}{cccccc}
A & B & C & D & E \\
0 & 0 & 0 & 1 &  \\
\end{array}
\]

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

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

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

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

We decided to set \(D\) true. So \(\neg D\) 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!)

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.

Why the technique is fast:

- Assigning 0 to a watched literal takes work
- 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.

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 \(E\), 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 shimming locally in the search tree in an inner loop, trying many assignments for the last few decision variables that lead to a conflict.)
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?)

### 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.
  - 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>C=1</td>
<td>F=0</td>
<td>A=1</td>
<td>G=0</td>
<td>J=0</td>
<td>K=1</td>
<td>B=0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Decision levels</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</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!

### Which decisions triggered the conflict?

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

- Decision Variable
- Variable assigned at previous d-level
- Variable assigned at current d-level
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
  - Neither decision variable 4, nor any vars set by propagation from it

A possible clause to learn
(avoid 5-variable combination that will always trigger this V_18 conflict)

\[-V_5(5)\vee V_3(5)\vee V_1(5)\vee V_8(2)\vee V_6(1)\]
\[-V_11(5)\vee V_13(2)\vee V_16(5)\]
\[-V_12(5)\vee V_10(5)\vee V_18(5)\]
\[-V_17(1)\vee V_19(3)\vee V_18(5)\]

A different clause we could learn
(avoid 4-variable combination that would have caught the problem sooner)

\[-V_5(5)\vee V_3(5)\vee V_1(5)\vee V_8(2)\vee V_6(1)\]
\[-V_11(5)\vee V_13(2)\vee V_16(5)\]
\[-V_12(5)\vee V_10(5)\vee V_18(5)\]
\[-V_17(1)\vee V_19(3)\vee V_18(5)\]

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(\(\phi\vee X\)) = SAT or DLL(\(\phi\vee \neg X\)) = SAT
        - then return SAT
      - else return UNSAT

Heuristic: Most Constrained First

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

  \[\text{First find an assignment for the most constrained variables ABC (often hard, much backtracking)}\]
  \[\text{If that succeeds, may be easy to extend it into a full assignment: other vars WXY fall easily into place (e.g., by unit propagation from vars already chosen, or because either W=0 or W=1 could lead to success)}\]
Heuristic: Most Constrained First

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

Good: solve for most constrained first

Bad: start with least constrained. All assignments to XZY 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

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 (2^n 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