Algorithms for Computational Logic

SAT Algorithms

Emmanuel Hebrard (adapted from) João Marques Silva

Outline

Outline

How to Solve SAT?

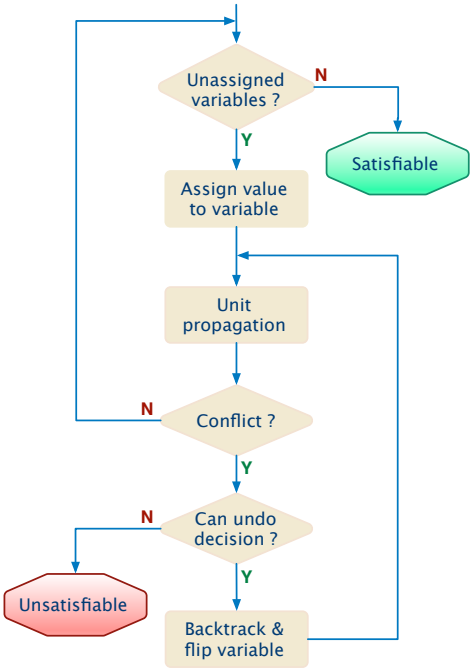
- Tableau: Deductive/syntactic system
- DP: Resolution
  - Davis & Putnam procedure in 1960 [DP60]
- DPLL: Semantic system, tree search for a model, with unit propagation
  - Davis, Logemann & Loveland procedure in 1962 [DLL62]
- CDCL: Conflict-Driven Clause Learning
  - Marques Silva & Sakallah in 1999
  - Moskewicz, Madigan, Zhao, Zhang & Malik in 2001
- Local search and heuristics

How to Solve SAT?

- Resolution is a powerfull proof system, but DP is exponential in memory
- DPLL is memory efficient, but tree search is a weak proof system
  - The length of the shortest refutation is at least as long as in resolution
  - There are cases where it is exponentially larger
- CDCL is memory efficient, very efficient in practice, and as powerfull as resolution as a proof system

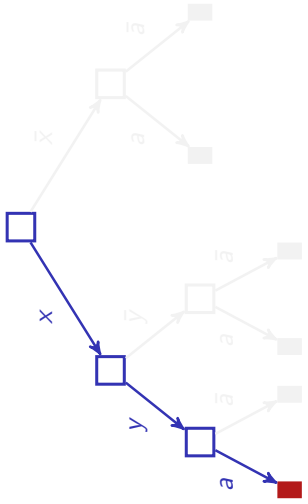
Outline

The DPLL Algorithm



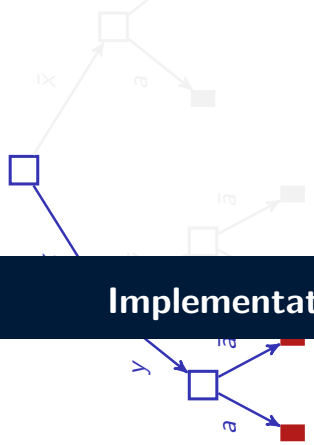
$$\mathcal{F} = (x \vee y) \wedge (a \vee b) \wedge (\bar{a} \vee b) \wedge (a \vee \bar{b}) \wedge (\bar{a} \vee \bar{b})$$

Level	Dec.	Unit Prop.
0	$\emptyset$	
1	x	
2	y	
3	a	$\rightarrow b \rightarrow \bot$



Level	Dec.	Unit Prop.
0	$\emptyset$	
1	x	
2	y	
3	$\bar{a}$	$\rightarrow \bar{b} \rightarrow \bot$

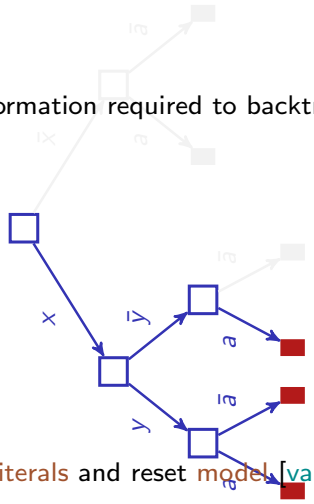
Level	Dec.	Unit Prop.
0	$\emptyset$	
1	x	
2	$\bar{y}$	
3	$\bar{a}$	$\rightarrow \bar{b} \rightarrow \bot$



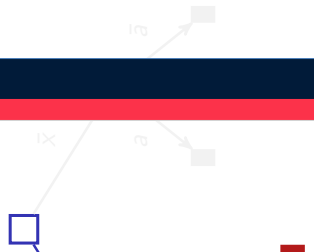
Implementation

- Data structures
  - trail: stores the information required to backtrack
    - ★ |trail| is the current level in the search tree
    - ★ trail(i) is the number of true literals at level i
    - ★ Stack: push(),back(),pop-back() in $O(1)$
- Functions
  - unassign-back() pop l from unit-literals and reset model[var(l)]

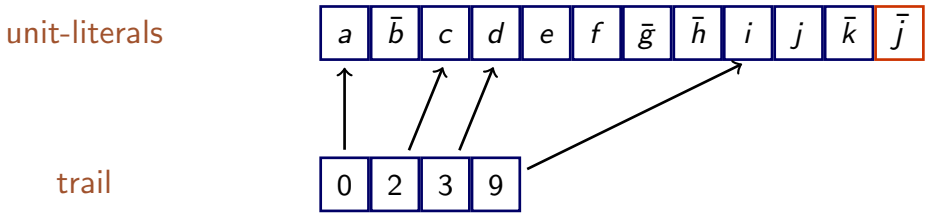
Level	Dec.	Unit Prop.
0	$\emptyset$	
1	$\bar{x}$	$\rightarrow y$
2	a	$\rightarrow b \rightarrow \bot$



Level	Dec.	Unit Prop.
1	x	$\rightarrow y$
2	$\bar{a}$	$\rightarrow \bar{b} \rightarrow \bot$



Backtracking



- Backtrack to *decision level* 3
- Backtrack to *decision level* 2

DPLL: Pseudocode

Algorithm: DPLL

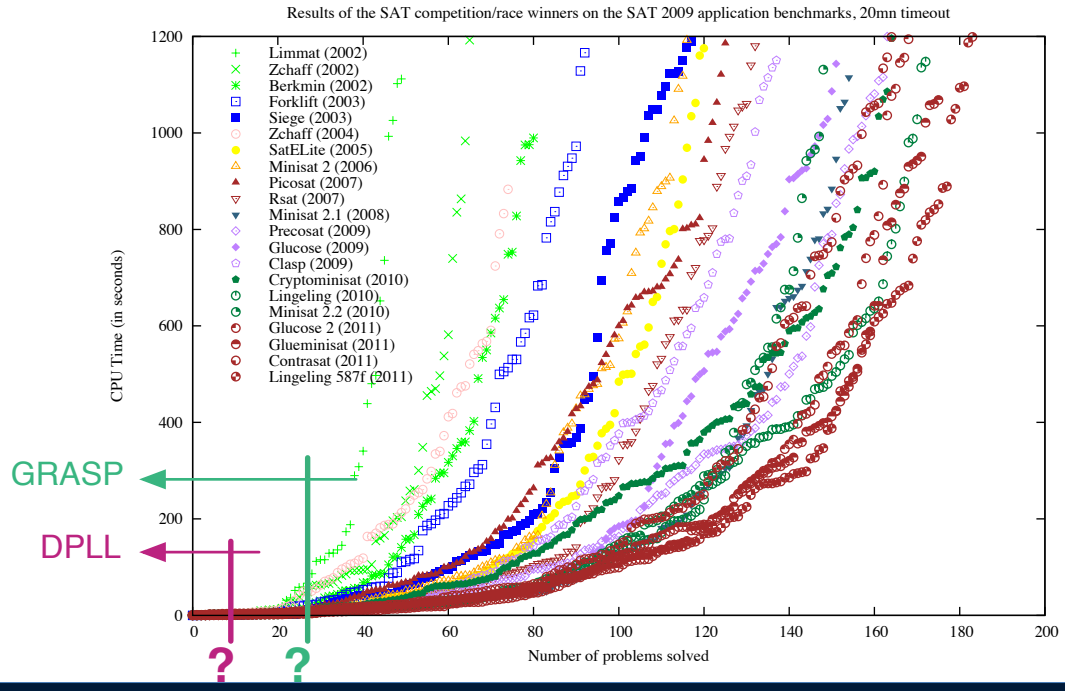
```
while satisfiability = UNKNOWN do
    if unit-propagate() then
        if |unit-literals| = n then satisfiability ← SAT // a model is found
        else
            trail.push(|unit-literals|) // save current level
            assign(select-lit()) // add a new true literal
    else
        if |trail| = 0 then satisfiability ← UNSAT // search tree exhausted
        else
            d ← unit-literals[trail.back()] // retrieve previous decision
            while |unit-literals| > trail.back() do unassign-back() // backtrack
            to-propagate ← trail.back()
            trail.pop-back()
            assign(d̄) // branch out of previous decision
```

Outline

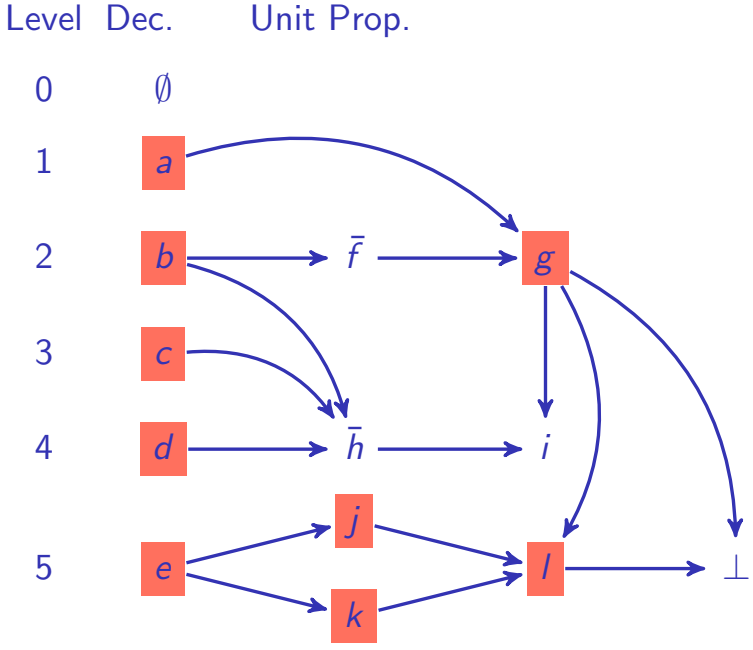
What is a CDCL SAT Solver?

- Extend DPLL SAT solver with: [DP60,DLL62]
  - Clause learning & non-chronological backtracking [MSS96,BS97,Z97]
    - ★ Exploit UIPs [MSS96,SSS12]
    - ★ Minimize learned clauses [SB09,VG09]
    - ★ Opportunistically delete clauses [MSS96,MSS99,GN02]
  - Search restarts [GSK98,BMS00,H07,B08]
  - Lazy data structures
    - ★ Watched literals [MMZZM01]
  - Conflict-guided branching
    - ★ Activity-based branching heuristics [MMZZM01]
    - ★ Phase saving [PD07]
  - ...

How Significant are CDCL SAT Solvers?



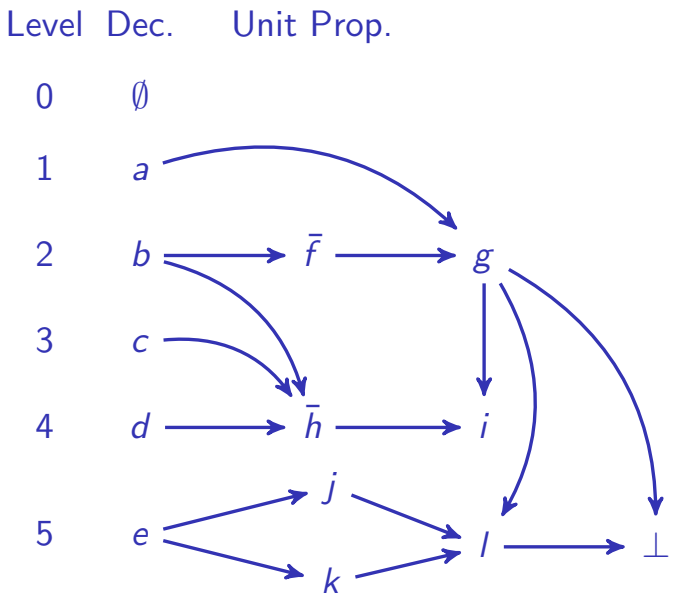
Implication Graph



$$
\begin{aligned}
\varphi = &(\bar{b} \vee \bar{f}) \quad \wedge \\
&(\bar{a} \vee f \vee g) \quad \wedge \\
&(\bar{d} \vee \bar{c} \vee \bar{b} \vee \bar{h}) \quad \wedge \\
&(h \vee \bar{g} \vee i) \quad \wedge \\
&(\bar{e} \vee j) \quad \wedge \\
&(\bar{e} \vee k) \quad \wedge \\
&(\bar{j} \vee \bar{k} \vee \bar{g} \vee l) \quad \wedge \\
&(\bar{l} \vee \bar{g}) \\
(a \wedge b \wedge c \wedge d \wedge e) &\implies \bot \\
(a \wedge b \wedge e) &\implies \bot \\
(g \wedge j \wedge k) &\implies \bot \\
(l \wedge g) &\implies \bot
\end{aligned}
$$

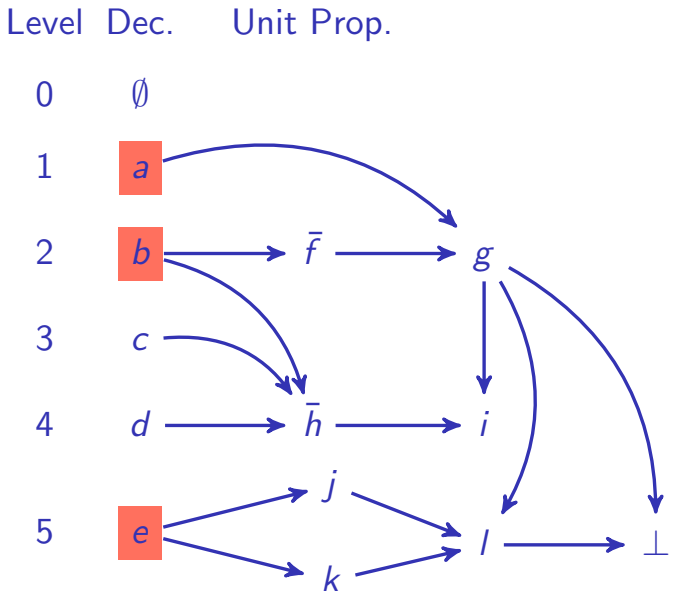
Cut of the Implication Graph

- Any *cut* that separate the *decisions* from the *fail* in the decision graph
- Cuts correspond to clauses
  - $\varphi \vDash (a \wedge b \wedge c \wedge d \wedge e) \implies \bot$: $\varphi \vDash (\bar{a} \vee \bar{b} \vee \bar{c} \vee \bar{d} \vee \bar{e})$
  - $\varphi \vDash (a \wedge b \wedge e) \implies \bot$: $\varphi \vDash (\bar{a} \vee \bar{b} \vee \bar{e})$
  - $\varphi \vDash (g \wedge j \wedge k) \implies \bot$: $\varphi \vDash (\bar{g} \vee \bar{j} \vee \bar{k})$
  - $\varphi \vDash (g \wedge l) \implies \bot$: $\varphi \vDash (\bar{g} \vee \bar{l} \vee \bar{k})$
- DPLL (bactracks) equivalent to learning that *one decision must be changed*
- CDCL learn non-trivial cuts

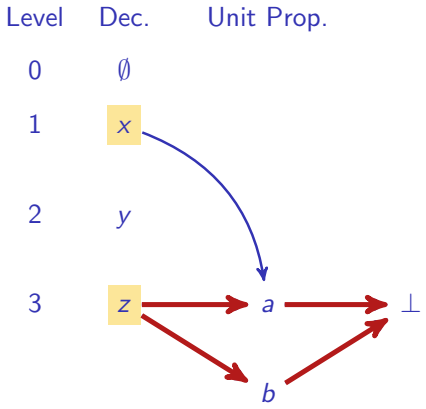


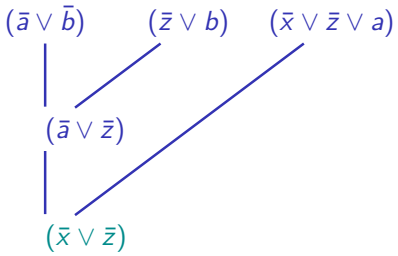
Clause Learning

- Learnt clause prevent the algorithm from repeating the same mistake later on
- Consider what DPLL would do next:
  - Explore branch $a \wedge b \wedge c \wedge \bar{e}$
  - Explore branch $a \wedge b \wedge \bar{c} \wedge e$
  - Explore branch $a \wedge \bar{b} \wedge c \wedge e$
  - Explore branch $a \wedge \bar{b} \wedge \bar{c} \wedge e$
- Adding the clause $(\bar{a} \vee \bar{b} \vee \bar{e})$ makes sure that the solver does not explore the last three branches



Clause Learning and Resolution





- Analyze conflict
  - Reasons: x and z
    - ★ Decision variable & literals assigned at lower decision levels
  - Create **new** clause: $(\bar{x} \vee \bar{z})$
- Can relate clause learning with resolution
  - Learned clauses result from (**selected**) resolution operations

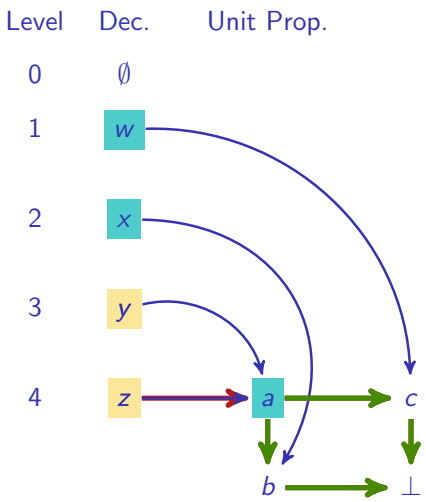
Computing a Cut

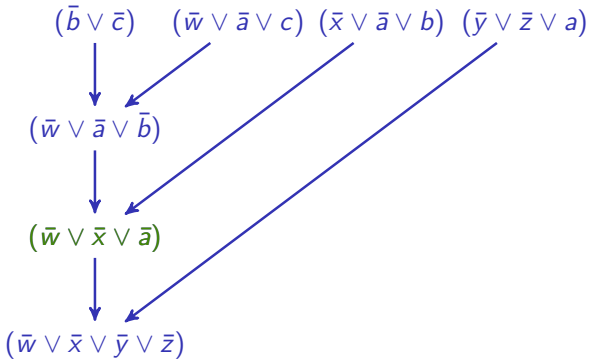
- Computing a minimum cut is polynomial (e.g., with Edmonds–Karp algorithm)
  - But costly and more importantly, might often return the failed clause (not asserting!)
- Computing a cut by exploring the implication graph up from the fail
  - At any time the list of open nodes is a valid cut
  - removing a literal from the current cut and replacing it by its parents is a resolution step

Unique Implication Point (UIP)

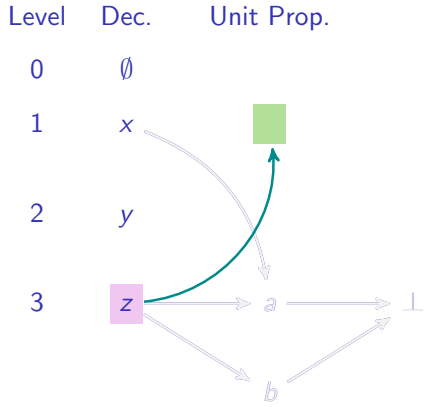
A *Unique Implication Point* is a node of the current decision level such that any path from the decision variable to the conflict node must pass through it

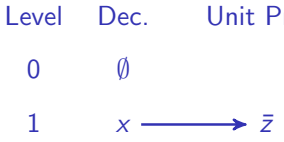
- The decision variable is a UIP
- There might be other UIPs





- Learn clause $(\bar{w} \vee \bar{x} \vee \bar{y} \vee \bar{z})$
- But a is an UIP
- Learn clause $(\bar{w} \vee \bar{x} \vee \bar{y} \vee \bar{z})$
- But a is an UIP: learn clause $(\bar{w} \vee \bar{x} \vee \bar{a})$





- Clause $(\bar{x} \vee \bar{z})$ is asserting at decision level 1 (it unit propagates at previous level)
- We want to learn UIP-clauses (clauses containing a UIP) because they are *asserting*
  - A learned clause is asserting if and only if it contains exactly one literal of the current level because literals from older levels are all falsified
  - A learned clause must contain *at least* one literal of the current level (since unit propagation did not detect an inconsistency at the previous level)
- Backjump to the highest level of any literal but the UIP

- Functions
  - unit-propagate() — return the failed clause if there is an inconsistency (null otherwise)
  - backjump(**Clause**:*c*) — conflict analysis and backjump

CDCL: Pseudocode

Algorithm: CDCL

```
while satisfiability = UNKNOWN do
    c = unit-propagate()
    if c = Null then
        if |unit-literals| = n then
            satisfiability ← SAT
        else
            trail.push(|unit-literals|)
            assign(select-lit())
    else
        if |trail| = 0 then
            satisfiability ← UNSAT
        else
            backjump(c)
```

Algorithm: Backjump

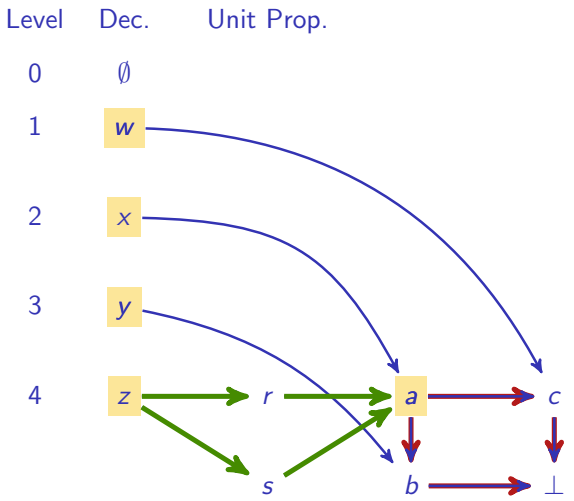
Input: Conflict clause c

```
learnt ← analyze-conflict(c)
l ← arg max_l({level(l) | l ∈ learnt})
lvl ← max({level[p] | p ≠ l ∈ learnt})
while |unit-literals| > trail[lvl] do unassign-back()
while |trail| > lvl do trail.pop-back()
add(learnt)          // l should be watched by learnt!
assign(l)
```


Implementing analyze-conflict

- We first need to encode the conflict graph
- The parents of a literal l node are the $k-1$ falsified literals of the clause that unit-propagated l
- For every variable x, store reason[x] the clause responsible for x's unit propagation
  - Encoding of the conflict graph
- Which cut(s) should we keep?
  - *First UIP clauses*

Multiple UIPs



- First UIP:
  - Learn clause $(\bar{w} \vee \bar{y} \vee \bar{a})$
- But there can be more than 1 UIP
- Second UIP:
  - Learn clause $(\bar{x} \vee \bar{z} \vee a)$
- In practice smaller clauses more effective
  - Compare with $(\bar{w} \vee \bar{x} \vee \bar{y} \vee \bar{z})$

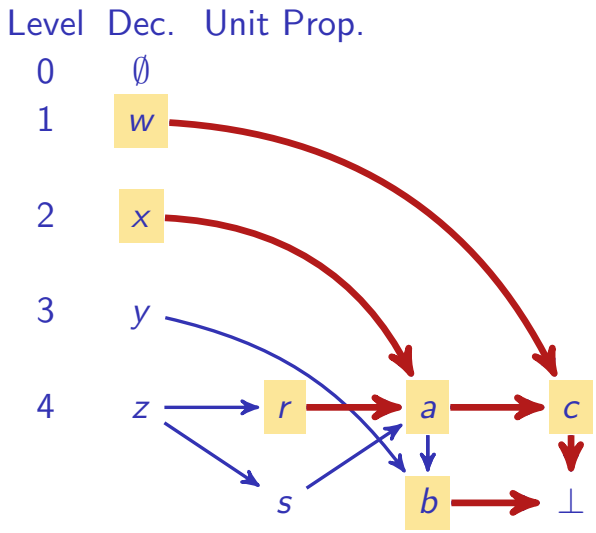
- Multiple UIPs used in GRASP [MSS96]
- First UIP learning used in Chaff [MMZZM01] and in most modern solvers

Why are First UIP Good?

- Mainly empirical evidences
- Can be seen as a way to detect "hubs"
- How to effectively vaccinate a population against a contagious desease if you have only a limited number of doses?
  - Pick a person randomly, ask her to name a friend, give a vaccine shot to the friend
  - Repeat until there is no dose
- People nominated as friends are more likely to know many people, and hence be super-spreaders
- The decision at failure level is always a UIP (random)
- Other UIPs are "friends" (linked via unit propagation)

Computing First UIPs

- Not all traversal orders reach the first UIP clause
  - E.g., resolve c then resolve a
- Solution: resolve literals in reverse chronological order (of unit propagation)
- The first UIP literal is not resolved until all its descendants are
  - By definition, once all its descendants are resolved, it is the only literal of the current level and the exploration can stop



Implementation

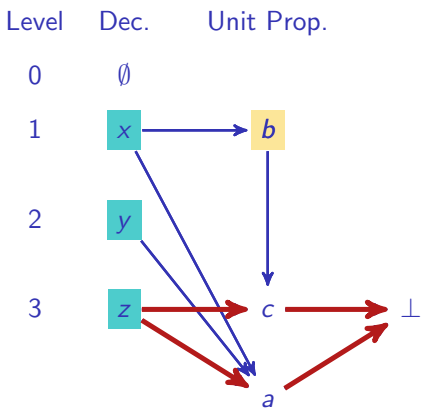
- Data structures
  - level [**Variable** : x] $\mapsto$ **int** — the decision level at which x was unit propagated
  - reason [**Variable** : x] $\mapsto$ **Clause** — the clause responsible for x's unit propagation
    - ★ Change assign(**Literal**:*l*) and unassign-back(**Literal**:*l*)
- Functions
  - analyze-conflict(**Clause**:c) $\mapsto$ **Clause** — analyze conflict on clause c and returns a firt UIP clause
  - backjump(**Clause**:c) $\mapsto$ Boolean — returns **false** if the search tree is exhausted and **true** otherwise

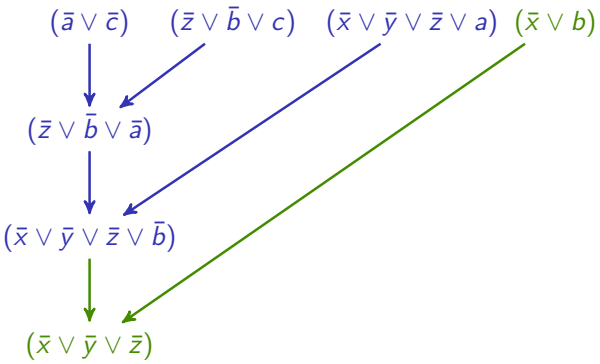
Algorithm: First UIP

Input: c

$seen \leftarrow \emptyset$ $learnt \leftarrow ()$
$reason \leftarrow c$
$n_{cur} \leftarrow 0$
$l \leftarrow$ None
$i \leftarrow |\text{unit-literals}| - 1$

repeat
  foreach $p \neq l \in reason \setminus seen$ **do**
    add p to $seen$
    if level$[p] = |\text{trail}|$ **then**
      $n_{cur} \leftarrow n_{cur} + 1$
    else
      add p to $learnt$
  while unit-literals$[i]$ *is not in seen* **do** $i \leftarrow i - 1$
  $l \leftarrow$ unit-literals$[i]$
  $reason \leftarrow$ reason$[l]$
  $n_{cur} \leftarrow n_{cur} - 1$
until $n_{cur} > 0$
add the last explore literal l to $learnt$



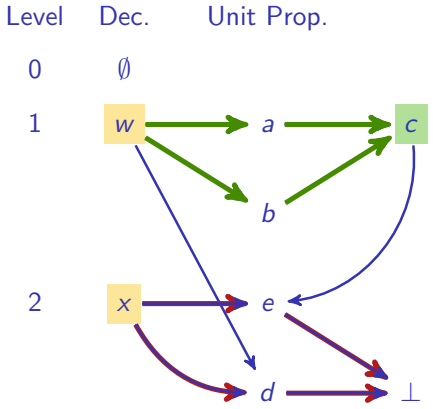


- Learn clause $(\bar{x} \vee \bar{y} \vee \bar{z} \vee \bar{b})$
- Apply self-subsuming resolution (i.e. local minimization)

[SB09]

- ~~Learn clause $(\bar{x} \vee \bar{y} \vee \bar{z} \vee \bar{b})$~~
- Apply self-subsuming resolution (i.e. local minimization)
- Learn clause $(\bar{x} \vee \bar{y} \vee \bar{z})$

Clause Minimization II



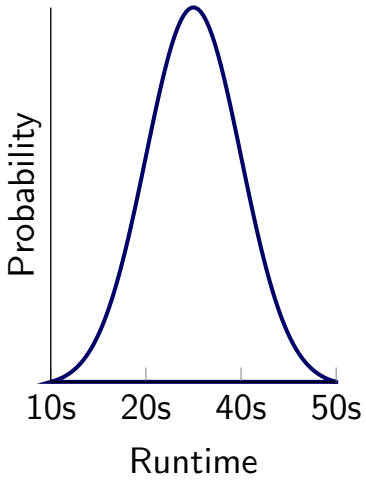
- Learn clause $(\bar{w} \vee \bar{x} \vee \bar{c})$ ~~Learn clause $(\bar{w} \vee \bar{x} \vee \bar{c})$~~
- Cannot apply self-subsuming resolution
  - Resolving with reason of c yields $(\bar{w} \vee \bar{x} \vee \bar{a} \vee \bar{b})$
- Can apply recursive minimization
- Learn clause $(\bar{w} \vee \bar{x})$

- Marked nodes: literals in learned clause [SB09]
- Trace back from c until marked nodes or new decision nodes
  - Learn clause if only marked nodes visited

Outline

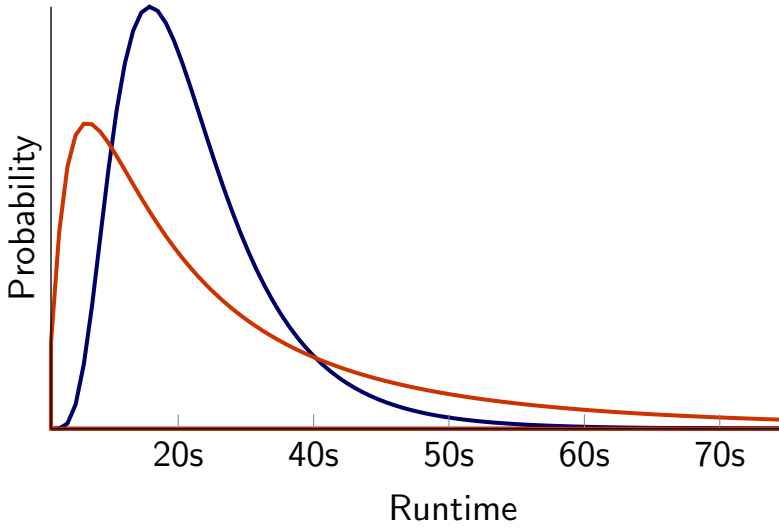
Runtime distribution

- Let sat-sol be a *randomized* SAT solver, and x be a SAT instance
- The *duration* of a run of sat-sol(x) depends on the random seed
- SAT solvers are Las-Vegas algorithms: guaranteed correctness, unknown runtime
  - Their runtime distribution can be leveraged to improve their efficiency !
- This is true of all exact solvers (MIP, CSP, etc.)



Heavy tails

- Runtime distributions are rarely Gaussian
- Often *Heavy tailed*
- The average may be greatly skewed to the right



Example: pigeon hole

- Pigeon hole formula $PHP^{n \to n-1}$:

$$
\begin{aligned}
&(x_{1,1} \vee x_{1,2} \vee \ldots \vee x_{1,n-1}) \wedge && \text{Pigeon 1 needs a hole} \\
&\ldots \\
&(x_{n,1} \vee x_{n,2} \vee \ldots \vee x_{n,n-1}) \wedge && \text{Pigeon n needs a hole} \\
&\bigwedge_{1 \le i < j \le n} (\bar{x_{i,1}} \vee \bar{x_{j,1}}) \wedge && \text{Hole 1 can contain at most 1 pigeon} \\
&\bigwedge_{1 \le i < j \le n} (\bar{x_{i,2}} \vee \bar{x_{j,2}}) \wedge && \text{Hole 2 can contain at most 1 pigeon} \\
&\ldots \\
&\bigwedge_{1 \le i < j \le n} (x_{i,\bar{n-1}} \vee x_{j,\bar{n-1}}) && \text{Hole } n-1 \text{ can contain at most 1 pigeon}
\end{aligned}
$$

Example: pigeon hole

- DPLL on the Pigeon hole formula takes exponential time

$$
\begin{aligned}
&(x_{1,1} \vee x_{1,2} \vee \ldots \vee x_{1,n-1} \vee x_1) \wedge && \text{Pigeon 1 needs a hole} \\
&\ldots
\end{aligned}
$$

- Variable x_1, if $\mathbf{true}$, allows Pigeon 1 to have its own hole, making the problem easy
- If Variable x_1 is set to $\mathbf{false}$, the problem is not satisfiable, and it takes a time exponential in n to prove it
- If we suppose that the solver branch on x_1 first and uniformly randomly pick the value $\mathbf{true}$ or $\mathbf{false}$:
  - It will solve the problem in under a second with probability $\frac{1}{2}$
  - It will solve the problem in $\Theta(2^n)$ time with probability $\frac{1}{2}$
  - In expectation: $\Theta(2^{n-1})$ time!

Search Restarts

- What if we *restart* the solver if no solution is found after 1s?
- Chances of taking more than 10 second is $\frac{1}{2^{10}}$
- Search restarts can reduce the runtime expectation when the runtime distribution is heavy tailed

Search Restarts I

- When a time limit τ is reached, we stop and resume search from the start
- Let t be a random variable equal to the runtime of the solver

$$T = p(t \leq \tau) \cdot \mathbb{E}_p[t \mid t \leq \tau] + (1 - p(t \leq \tau)) \cdot (\tau + T)$$

$$T = \mathbb{E}_p[t \mid t \leq \tau] + \frac{(1 - p(t \leq \tau))\tau}{p(t \leq \tau)}$$

- Simple Markov Decision Process with two states ("solved" and "not solved")
  - There is a stationary (constant) policy τ^* that minimizes the runtime $T(\tau^*)$

Search Restarts II

- When the expectation of the runtime is unknown, the *Luby's universal strategy* guarantees a runtime of $T(\tau^*) \log T(\tau^*)$

$$\tau_i = \begin{cases} 2^{k-1}, \text{ if } i = 2^{k-1} - 1 \\ \tau_{i-2^{k-1}+1}, \text{ if } 2^{k-1} \leq i < 2^k - 1 \end{cases}$$

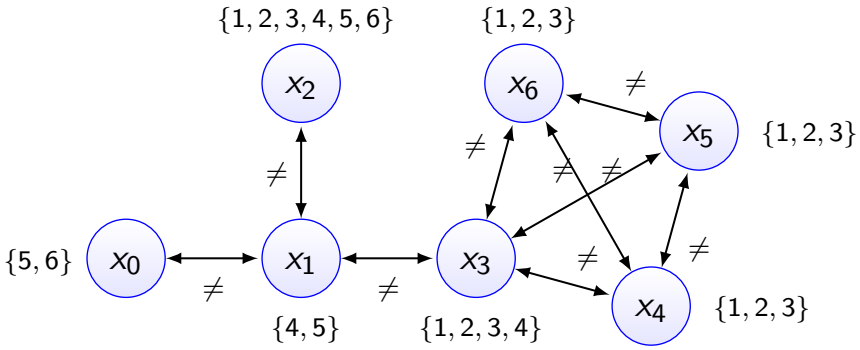
i:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
k:	1	2	2	3	3	3	3	4	4	4	4	4	4	4	4	5
2^{k-1}:	1	2	2	4	4	4	4	8	8	8	8	8	8	8	8	16
τ_i:	1	2	2	4	2	2	4	8	2	2	4	2	2	4	8	16

- In practice, the geometric sequence $\tau_i = f^i$ works well

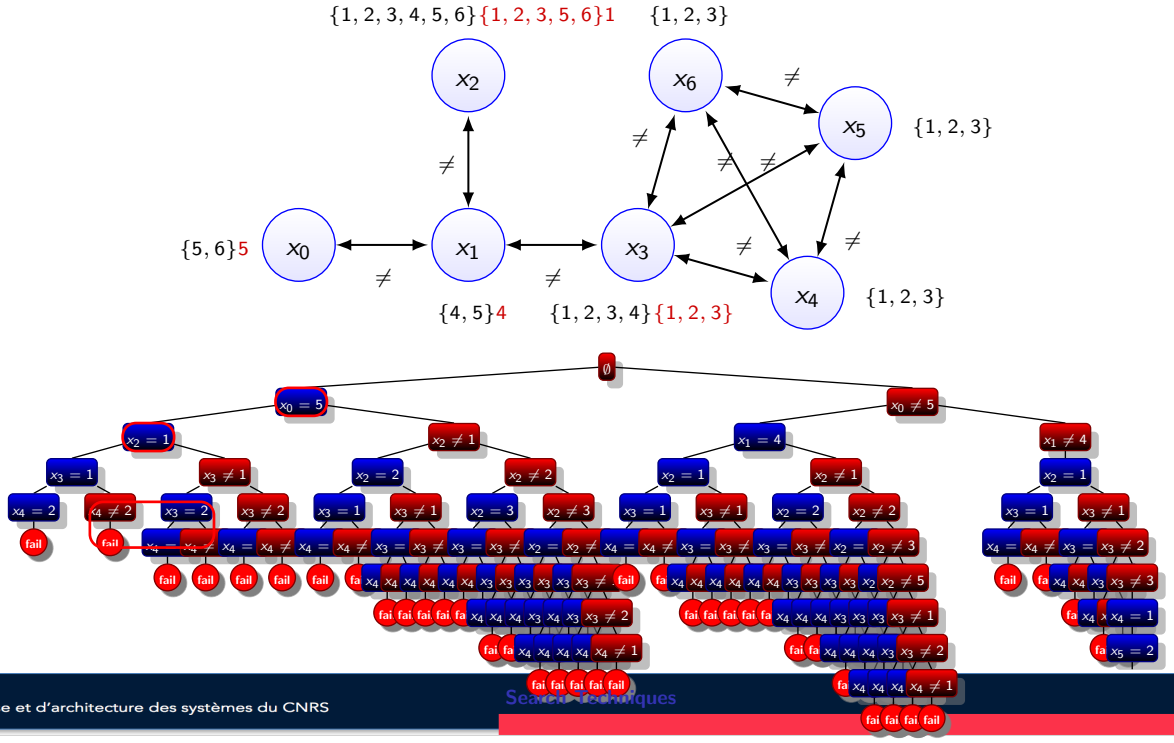
Search Heuristics

- Unit propagation reduce the size of search tree by cutting branches
- The branching choice also has an impact on the size of the tree
  - It can have a huge impact, but it is hard to know which choice is best
- Some principles:
  - If we are in an *unsatisfiable* subproblem, try to detect it as soon as possible
    - ★ By branching first on part of the problem that is most *constrained*
  - If we are in a *satisfiable* subproblem, try to stay on a branch leading to a solution
    - ★ Choice of the most *promising* branch

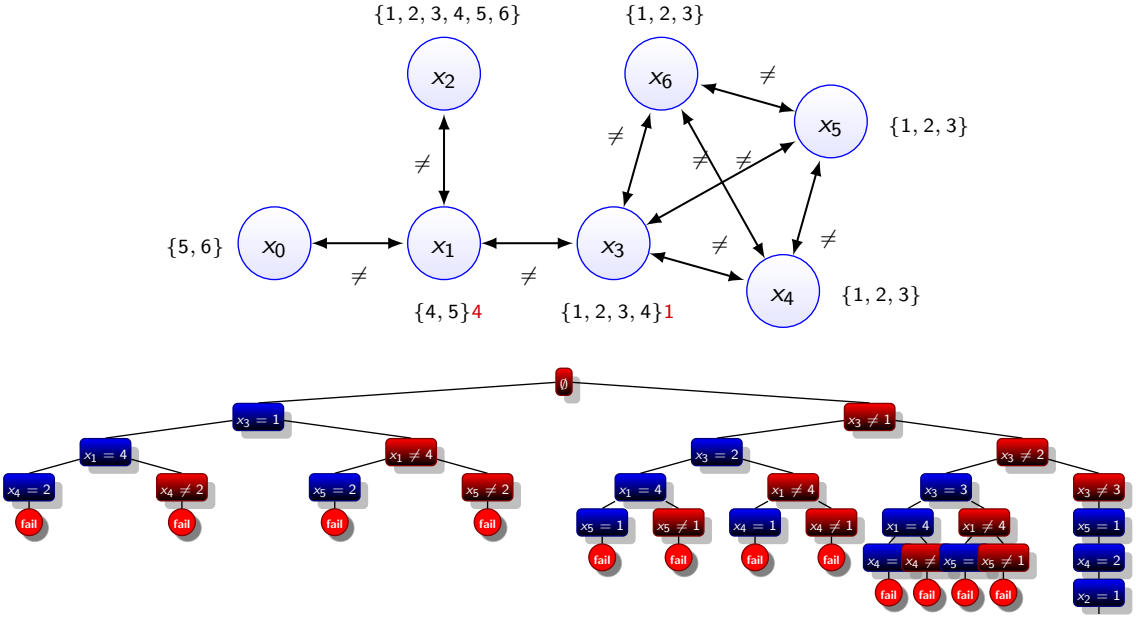
Coloring



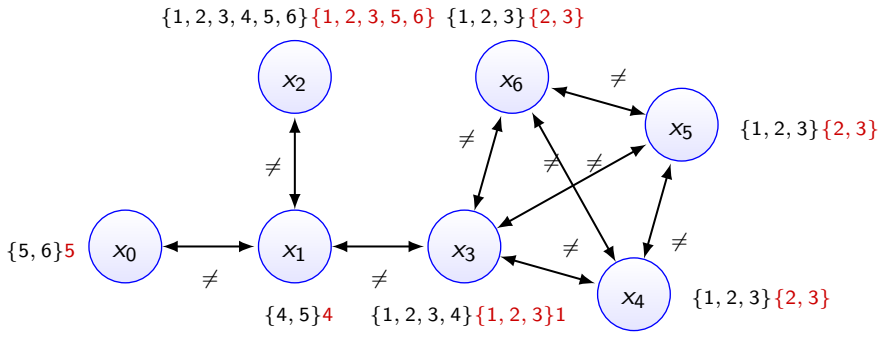
Lexicographic

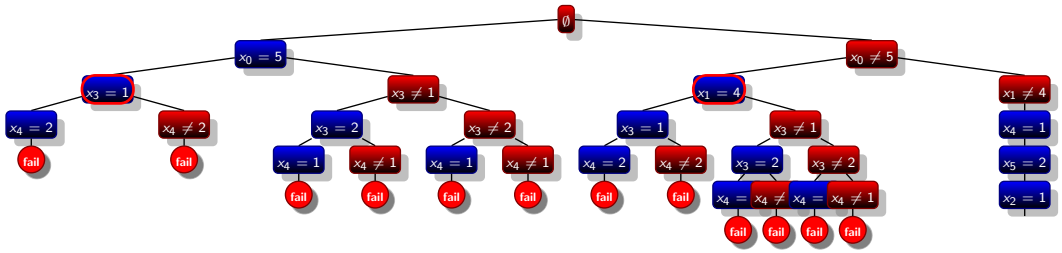


Maximum Degree

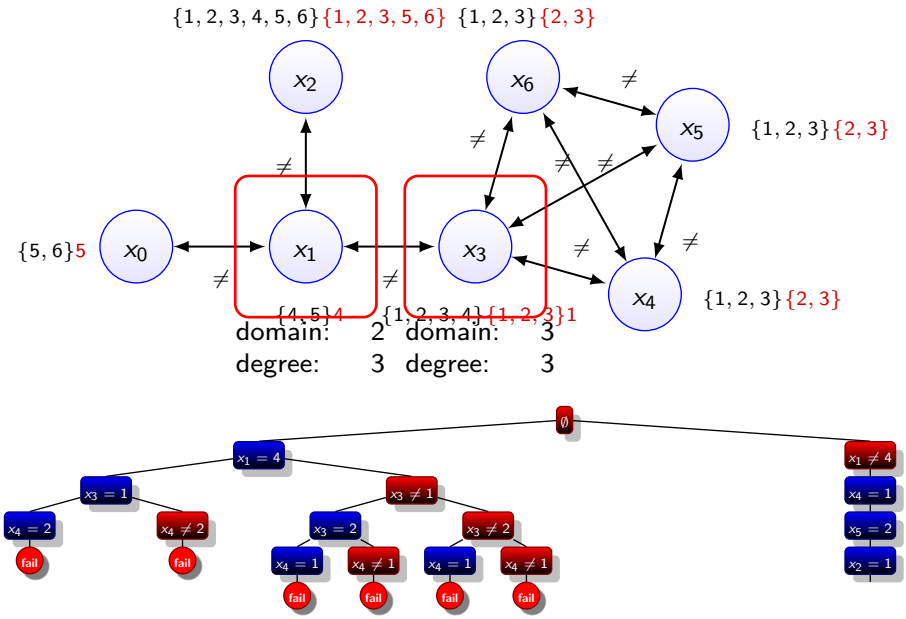


Minimum Domain

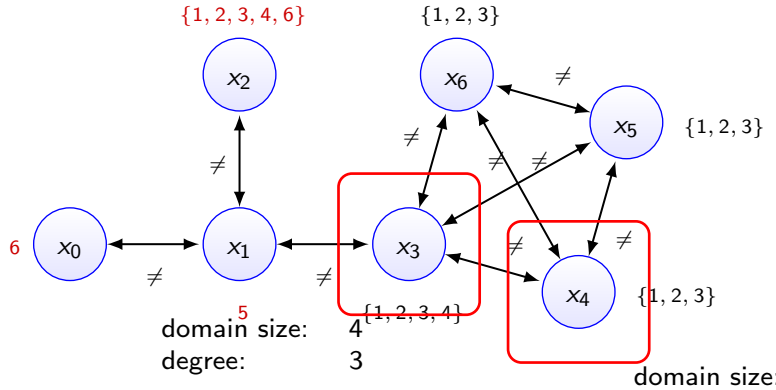


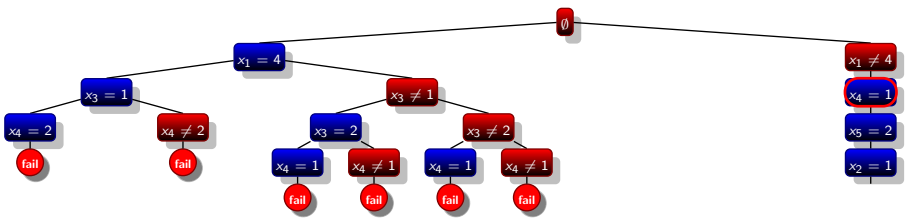


Minimum Domain / Degree



Minimum Domain / Degree





Branching Strategies in SAT

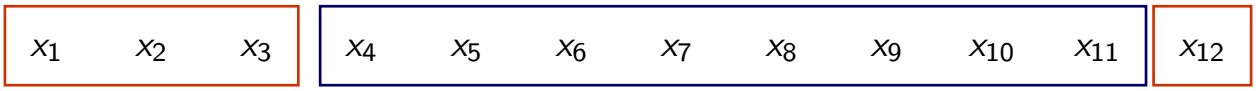
- Same principles in SAT and all other tree-search methods:
  - *Variable ordering*: on which variable should we branch first?
    - ★ The one on which we will fail on both subtrees, to get out of the *unsatisfiable* branch
    - ★ Otherwise, on the one that will *minimize* the size of the subtrees
  - *Value ordering*: on which variable should we branch first?
    - ★ The one most likely to lead to a solution
    - ★ If the current subtree is not satisfiable, it does not matter (much), both branches must be explored
  - Most of the time is spent getting out of *unsatisfiable subtrees*: the variable ordering is more important than the value ordering
    - ★ When solving an optimization problem top-down, finding good quality solutions quickly is important
    - ★ Interaction with clause-learning

Variable Ordering

- *Variable State Independent Decaying Sum* (VSIDS)
- Assigns a weight to variables *involved in conflicts*: activity score
- Variants exist:
  - Increment weight of the literals in the learned clause
  - Increment weight of the literals in the learned clause *and all variables resolved during conflict analysis*
- The activity score $A(i)$ of a variable x_i is the *decayed* sum of the weight increments:
  - Let $b_j(i)$ be equal to 1 if variable x_i's activity was incremented in the j-th fail, and let $0 < \gamma \leq 1$ be a constant, and k the number of fails

$$A(i) = \sum_{j=1}^{k} \gamma^{k-j} b_j(i)$$

Value Ordering



subproblem size up to 2^8

- When backjumping with an asserting clause, we undo potentially useful search
- Suppose that the variables between the conflict and assertion levels encode a (relatively) independent problem: its solution is lost
- *Phase saving*: branch using the previous value
  - If the previous solution still stands, it will be found efficiently
- Synergy with clause learning
  - Intuitively, we want to learn clauses that constrain variables in an *unsatisfiable core*: recently learned clauses are still asserting if we use phase saving

Clauses Deletion

- A SAT solver typically fail (tenth of) thousands times per second
  - Learn a new clause on every fail
  - Learned clauses tend to be long
- Unit propagation via watched literal is efficient, but still accounts for most of the run time
- Moreover, not all clauses are equally useful, some never unit propagate
- Can we reliably predict which clauses are more promising and forget the rest?

Clause Activity

- Some intuitive criteria:
  - **Length**: long clauses unit propagate (probably) less often
  - **Activity**: clauses with less active literals have (historically) unit propagated more often
- Deleting long and inactive learned clauses is useful
- Clause deletion is very important, but difficult to parameterized (how often?, how many?)
- Length and activity are not perfect predictors

Variable (in)dependence

- Some clauses are long but useful
- In general, a clause of length L can be satisfied in $2^L - 1$ ways
- The clause $x_1 \vee \ldots \vee x_{100}$ from the direct encoding of the CSP variable $x \in \{1, \ldots, 100\}$ can be satisfied in only 100 ways (the variable takes exactly one of the 100 values)
  - The unit literal x_i unit propagates $\bar{x}_j$ for all $j \neq i$ via pairwise or sequential clauses
- Clauses involving *inter-dependent* literals are more likely to unit propagate: the implicit relation on dependent variables is tighter

Literal Block Distance

- We want something *efficient*
- Idea: variables that unit propagated at the same level tend to be more linked together

Literal Block Distance $lbd(0)$

Let level[l] be the decision level at which literal l was inferred.

$$lbd(c) = |\{\text{level}[l] \mid l \in c\}|$$

- Solver "Glucose" was the first to use this idea of "Glue clauses" and was very successful [Audemard & Simon]

Outline

What is a CDCL SAT Solver?

- Extend DPLL SAT solver with: [DP60,DLL62]
  - Clause learning & non-chronological backtracking [MSS96,BS97,Z97]
    - ★ Learn First-UIP clauses [MSS96,SSS12]
    - ★ Minimize learned clauses [SB09,VG09]
    - ★ Opportunistically delete clauses (LBD) [MSS96,MSS99,GN02]
  - Search restarts [GSK98,BMS00,H07,B08]
  - Lazy data structures
    - ★ Watched literals [MMZZM01]
  - Conflict-guided branching
    - ★ Activity-based branching heuristics [MMZZM01]
    - ★ Phase saving [PD07]
  - ...