# Inprocessing in SAT solvers

## Presentation at SAT/SMT Summer School 2015

Mate Soos

Cisco

16th of July 2011

# Outline

Motivation

Clause massaging

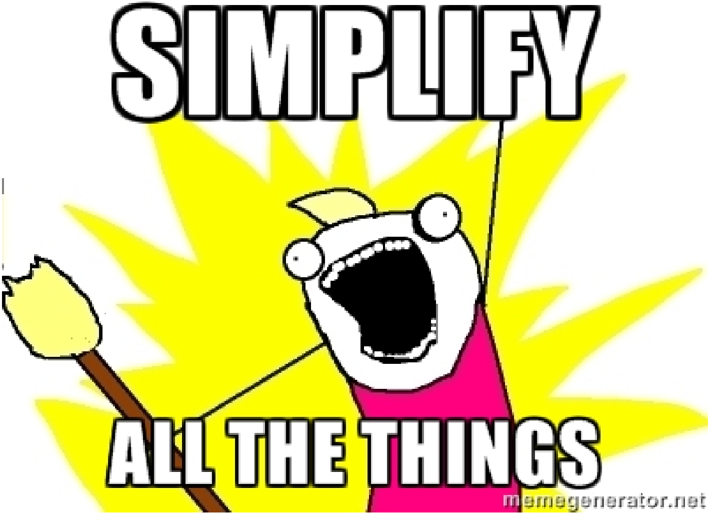
Variable massaging

Takeaways

# The problem

- CDCL is nice but doesn't do everything
- No satisfied clause removal
- No replacement when $a = (\neg)b$
- No removal of trivially satisfiable clauses
- etc.

# The solution



# Outline

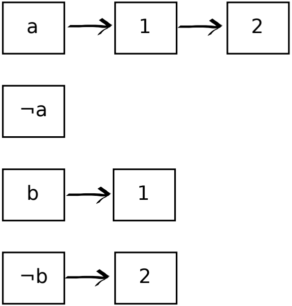
Motivation

Clause massaging

Variable massaging

Takeaways

# Concept: literal occurrence list



a → 1 → 2

¬a

b → 1

¬b → 2

1. $a \vee b$
2. $a \vee \neg b$

Note: misnomer, it's really an array of arrays.
Array provides better cache locality.

# Tautologies

- No point in having a clause $a \vee b \vee \neg b$
- We remove it during parsing of CNF
- Sort clause $\rightarrow O(n \log n)$
- Check linearly if previous literal is inverse of current one
- Mem need: 1 lit + clause for in-place sort

# Unit clauses

- Keep an array of assignments: Undefined, True, False
- Unit clauses are only stored here
- Remove clauses containing a literal set to True
- Remove from clauses literals set to False
- Time: $O(\sum \text{lits})$
- Note: we keep 1B to store the 3 values. 2b would be enough, but it's accessed often and x86/amd64 is slow at bit manipulation

# Pure literal rule

- If variable $x$ only appears in positive or only in negative form, all clauses it appears in can be trivially satisfied
- Time: $O(\sum \text{lits})$, Mem: $O(\text{num vars})$
- Alternatively, once occur list is built, we have the info already so time is $O(\text{nvars})$

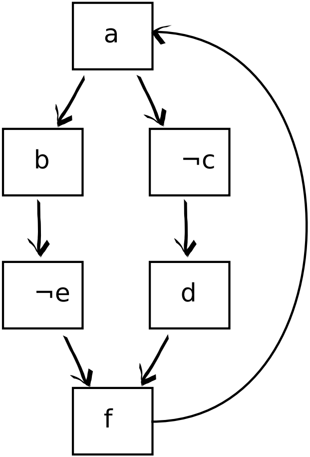
# Subsumption [EenBiere'07]

- $a \vee b$ is stronger than $a \vee b \vee c$. No need for former
- Use occurrence list of clauses
- Backward subsumtion: check if $a \vee b$ subsumes anything
- Cheaper, because otherwise we don't know which literal(s) from $a \vee b \vee c$ are missing and so we have to try them all
- Also, we can start with shorter clauses and ignore longer ones
- Tricks used: bloom filter, fingerprint stored inside occurance list (no dereferencing of ptr)
- Time need: go through occurrance lits of lit with smallest occur size for each clause
- Note: unit clause removing clauses is subsumption

# Self-subsuming resolution [EenBiere'07]

- $a \vee \neg b \vee c$ resolved with $a \vee b$ is $a \vee c$ which subsumes former
- Use occurrence list of clauses
- We do it backwards for same reson as before: take small $A$ and find all longer clauses $B$ that can be shortened to $B'$
- Sort the clauses' literals, go through linearly, abort if any variable is in $A$ but not in $B$, only allow one inverted variable.
- Tricks used: bloom filter, fingerprint stored inside occur list (no dereferencing of ptr), if none are inverted it's subsumed
- Time need: just like subsumption, but have to go through both the lit's and its inverse's occur list
- Note: unit clause removing literals from clauses is self-subsuming resolution

# Concept: binary implication graph (BIG)



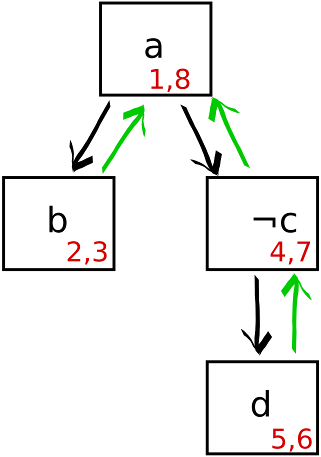
CNF clauses

- $\neg a \vee b,\ \neg a \vee \neg c$
- $\neg b \vee \neg e,\ c \vee d$
- $e \vee f,\ \neg d \vee f$
- $\neg f \vee a$

Transitive clauses:

- $\neg a \vee \neg e$
- $\neg a \vee d$
- $\neg a \vee f$
- $c \vee f$
- $\ldots$

# Stamping [HelueJarvisaloBiere'11]



- $\neg a \vee b$, $\neg a \vee \neg c$, $c \vee d$
- DFS through BIG, note "START, END" times
- Dec sort clauses' lits according to START, if previous lit's END is smaller than current, remove current lit
- Inc sort clauses' lits according to START of inverted lit, if previous inverted lit's END is larger than the current's, remove current lit
- $g \vee b \vee a$. 1st case, sorted: $b \vee a \vee g$

# Outline

Motivation

Clause massaging

Variable massaging

Takeaways

# Equivalent literal substitution [WarnersMaaren'1999]

- Find strongly connected component (SCC) in BIG
- SCC: every node can be reached from every node
- It's an equivalence class
- Use Tarjan's algorithm to find it, linear time
- Replace equivalent literals
- Some clauses become tautologies, some become shorter
- Ex.: $a = \neg b$, so $a \vee b \vee c$ is a tautology
- Ex.: $a = \neg b$, so $a \vee \neg b \vee d \rightarrow a \vee d$
- Note: above two can be simulated by respectively subsumption and self-subsuming resolution if we take trasitive binary clauses into account

# Failed Literal Probing [Freeman'95], [Stalmarck'92]

- Enqueue $a$, see if it fails. If fails → set $\neg a$
- Enqueue $\neg a$. If both $a$ and $\neg a$ propagate $b$, set $b$
- Keep an array of variables set for $a$, check against $\neg a$
- Can be *very* expensive — we need to roll back every time
- Trick: build stack from binary clauses, enqueue iteratively, reuse old propagations
- Example $a \rightarrow b \rightarrow c$ So enqueue $b$, propagate. Then enqueue $a$ and propagate
- No need to roll back so often!

# Bounded variable elimination [EenBiere'05]

- Through clause distribution
- $(a \vee b)$ x $(\neg a \vee c, \neg a \vee d)$
- Becomes $b \vee c$, $b \vee d$
- Note: lots of resolvents are Tautologies
- Bounded: don't increase num cl, abort if resolvent too large
- Call backward subsumption with resolved clause
- Order matters, e.g. order according to least occurred variable
- Trick: try to reverse strengthen to remove resolvent clause
- Note: simulates pure literal rule

# Bounded variable addition [HeuleJarvisaloBiere'13]

Replace

$$\begin{array}{cc} (a \vee d) & (a \vee e) \\ (b \vee d) & (b \vee e) \\ (c \vee d) & (c \vee e) \end{array}$$

by

$$\begin{array}{ccc} (\neg x \vee a) & (\neg x \vee b) & (\neg x \vee c) \\ (x \vee d) & (x \vee e) & \end{array}$$

- Started off with 6 clauses and 12 literals
- Finished with 5 clauses and 10 literals
- We can have any number of "columns" (here: 2)
- We can have any set of literals (here: $d$, $e$)

# Variable renumbering

- Variables are set, eliminated, replaced
- Renumber them to pack them tightly, use a simple map
- Lowers memory footprint
- More tightly packed structs → better cache usage
- But maintaining the map with BVE-(un)eliminated, BVA-added (and later BVE-removed…), replaced, and new variables can be complicated

# Outline

Motivation

Clause massaging

Variable massaging

Takeaways

# Takeaways

- We can make the work of SAT solvers easier
- Improving the clause set
- Improving the variable set
- Thereby improving efficiency of heuristics, memory footprint, cache locality, etc.
- Some methods can be (partially) simulated by other, more general methods — but runtimes will be *vastly* different