SAT Preprocessing

Mate Soos

SAT Winter School’2023

IIIT Hyderabad, India

December 17, 2023

Based on slides by **Armin Biere**

Rewrite system for CDLC

- You can think of CNF as a language
- CDCL solver is a machine that either accepts/rejects a CNF
- But CDCL is poor:
  - Cannot remove irredundant clauses. Cleans redundant ones, but not irredundant ones
  - Cannot remove variables, it can only set them
  - Doesn’t by default do a number of very simple rewrite rules — only does them by accident.
- We can try to re-write CNF to make it easier for CDCL to accept/reject:
  - Fewer variables (variable elimination)
  - Fewer constraints (subsumption)
  - Stronger constraints (strengthening)

Probing

```
for lit in literals:
  new_decision_level()
  enqueue(lit)
  ret = propagate()
  backtrack()
  if ret == False:
    enqueue(-lit)
    continue
```

- Enqueues each literal and propagates
- If propagation fails, clearly that literal must be false
- Cheap, except if you have 100M variables

Stalmarck's method – [Stal1989]

```
for lit in literals:
  new_decision_level()
  enqueue(lit)
  ret = propagate()
  lits_set = get_lits_set()
  backtrack()

  new_decision_level()
  enqueue(-lit)
  ret = propagate()
  lits_set2 = get_lits_set()
  backtrack()

  enqueue(intreserction(lits_set1, lits_set2))
```

- Enqueues lit, propagates. Get set of literals forced
- Enqueues $\neg lit$, propagates. Get set of literals forced
- If there is a solution, either lit or $\neg lit$ is set. So whatever the intersection of their forced literals can be set.

Backbone

```
def backbone(F):
  for lit in literals:
    s = sat_solver(F && lit)
    ret = s.solve()
    if ret == UNSAT:
      F = F && (-lit)
  return F
```

- Runs a full SAT solver checking if there is a solution with lit
- If there is no solution, $\neg lit$ can be added to F
- Obviously not useful if you are only trying to check for satisfiability
- Great for e.g. counting

Equivalent Literal Substitution – [Bac02, BW03]

Let's see these binary (2-long) clauses:

$$\begin{array}{ccc} a & \vee & \neg b \\ b & \vee & \neg c \\ c & \vee & \neg a \end{array}$$

- If $a = F$ is set, $b = F$ is propagated, then $c = F$ is propagated, which propagates $a = F$.
- If $a = T$ is set, $c = T$ is propagated, then $b = T$ is propagated, which propagates $a = T$.
- There is a loop here! It's a **strongly connected component** (SCC)
- Therefore, $a = b = c$
- Replace b and c with a everywhere. Two fewer variables!
- When SCC, always think: Tarjan's algorithm. Super-quick.

Subsumption

$$\text{Replace} \quad \begin{array}{l} (\neg a \vee \neg b) \\ (\neg a \vee \neg b \vee c) \\ (\neg a \vee \neg b \vee \neg d) \end{array} \quad \text{by} \quad \begin{array}{l} (\neg a \vee \neg b) \\ \cancel{(\neg a \vee \neg b \vee c)} \\ \cancel{(\neg a \vee \neg b \vee \neg d)} \end{array}$$

- Removes all clauses that clause is subset of
- One of the few techniques that is confluent
- Impementation:
  - What is $(a \vee \neg b)$ a subset of?
  - Uses Bloom filter, hash is literals in clause

Strengthening/Weakening

Strengthening = Self-subsuming Resolution

Replace $\begin{array}{l}(a \vee \neg b \vee c \vee d) \\ (a \vee b)\end{array}$ by $\begin{array}{l}(a \cancel{\vee \neg b}\ c \vee d) \\ (a \vee b)\end{array}$

- Notice: $(a \vee \neg b \vee c \vee d) \odot (a \vee b) = a \vee c \vee d$, which happens to subsume $(a \vee \neg b \vee c \vee d)$
- Everything that $(a \vee \neg b)$ can subsume can be strengthened to remove $\neg b$
- Everything that $(\neg a \vee b)$ can subsume can be strengthened to remove $\neg a$
- Implementation: what can $(a \vee b)$ strengthen?

Weakening = Reverse Self-subsuming Resolution

Replace $\begin{array}{l}(a \cancel{\vee \neg b}\ c \vee d) \\ (a \vee b)\end{array}$ by $\begin{array}{l}(a \vee \neg b \vee c \vee d) \\ (a \vee b)\end{array}$

- Do the reverse
- Yes, this will come handy, you just wait

Binary Implication Graphs – BIG

Transitive Reduction – [HJB13]

$$\text{Replace} \quad \begin{array}{l} a \vee \neg b \\ b \vee c \\ a \vee c \end{array} \quad \text{by} \quad \begin{array}{l} a \vee \neg b \\ b \vee c \\ \cancel{(a \vee c)} \end{array}$$

- $(a \vee \neg b) \odot (b \vee c) = a \vee c$
- In terms of edges: $a \to b \to c$, so we can reach c from a. No need for edge $a \to c$

Hyper-Binary Resolution – [Bie09, HJS11]

$$\text{Add to} \quad \begin{array}{l} a \vee \neg b \\ a \vee \neg c \\ b \vee c \vee d \end{array} \quad \text{redundant binary clause} \quad a \vee d$$

- Substitute $a = False$, propagates d
- In terms of edges: $a \to b$, $a \to c$, and $b \vee c \vee d$ is $(b, c) \to d$. So we can reach d from a
- Why is this useful? Because $a \vee d$ means that $d = False$ propagates $a = True$. Stronger propagation!
- Notice: $a \vee d$ could contribute to SCC

Time Stamping (unhiding) – [HJB11]

Clauses: $(h \vee \neg g), (h \vee \neg f), (f \vee \neg c), (f \vee \neg d), (d \vee \neg a)$

Question: is $\neg h \vee a$ implied? **Answer**: Yes, it can be reached via the BIG.

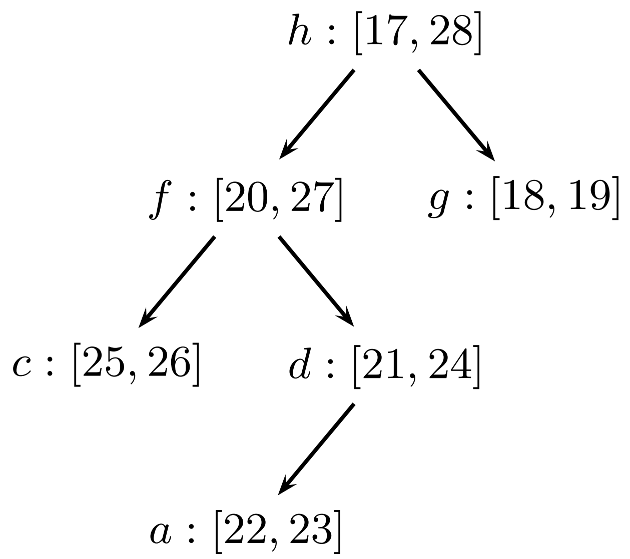
Question: Is there a fast (constant-time) way to decide it?

$$\begin{tikzcd}
 & & h : [17, 28] \arrow[dl] \arrow[dr] & \\
 & f : [20, 27] \arrow[dl] \arrow[dr] & & g : [18, 19] \\
c : [25, 26] & & d : [21, 24] \arrow[dl] & \\
 & a : [22, 23] & &
\end{tikzcd}$$

- Do a DFS. First number: time first visited. Second number: time last visited. Parenthesis Theorem.
- Since $h[0] = 17 > a[0] = 22$ and $h[0] = 28 > a[1] = 23$, it is implied.

Time Stamping cont. (unhiding/hidden literal elimination)

Clauses: $(h \vee \neg g), (h \vee \neg f), (f \vee \neg c), (f \vee \neg d), (d \vee \neg a)$



- Remember: $h \vee \neg a$ is implied. So we can strengthen: $x_1 \vee x_2 \vee x_3 \vee h \vee a$
- Easy! Order by 1st value, check if 2nd value is larger.
- Homework: the same timestamps can be used to remove clauses
- Funny part: sorting clauses can be ”expensive”

Vivification – [HS07, PHS08]

```
def vivify(cl):
  remove_clause(cl), sort(cl)
  implied = false, new_cl = {}, new_decision_level()
  for lit in clause:
    if false(lit): continue
    if true(lit): implied = true, break
    new_cl.insert(lit)
    enqueue(-lit), propagate()
  if not implied: attach_clause(new_cl)
  backtrack()
```

- Enqueues literals' negations one-by-one and propagates them
- So at literal n we have enqueued $\neg l_1 \wedge \neg l_2 \ldots \wedge \neg l_{n-1}$
- If the n-th literal has been enqueued $True$, the clause $l_1 \vee l_2 \vee l_{n-1} \vee l_n$ must be implied by the formula
  - But that subsumes the original clause! $\rightarrow$ we can remove the original clause
- If a literal is already falsified through propagation, the clause $l_1 \vee l_2 \ldots \vee l_{n-1} \vee \neg l_n$ is implied by the formula
  - But that strengthens the original clause! $\rightarrow$ we can remove the literal l_n

SSTD-Oracle Vivification – [Korhonen, Jarvisalo]

```
def sstd_oracle_vivify(cl, qlit):
  new_cl = cl - lit
  for lit in new_cl:
    if qlit != lit: enqueue(-lit)
  ret = sat_solve()
  if ret == UNSAT: return new_cl
  else: return cl
```

- Enqueues all literals' negation in the clause except the literal we query
- Constructs query: $F \land \neg l_1 \land \neg l_2 \ldots \land \neg l_n$, where $C = l_1 \lor l_2 \ldots \lor l_n \lor l_{qlit}$
- If this formula is UNSAT, then $C' = l_1 \lor l_2 \ldots \lor l_n$ must be implied by F!
- But C' subsumes C by exactly one literal, l_{qlit}. So we can remove l_{qlit} from C

SSTD-Oracle Sparsification – [Korhonen, Jarvisalo]

```
def sstd_oracle_vivify(F, cl):
  F'=F-cl
  s = solver(F')
  for lit in cl: enqueue(-lit)
  ret = s.sat_solve()
  if ret == UNSAT: F = F-cl
  return F
```

- Constructs F' that doesn't have C in it
- Enqueues all literals' negation in the clause
- Constructs query: $F' \wedge \neg l_1 \wedge \neg l_2 \ldots \wedge \neg l_n$, where $C = l_1 \vee l_2 \ldots \vee l_n$
- If this query is UNSAT, then C must be implied by F
- So we can remove C from F

Resolve and Subsume – [EénBiere-SAT'05]

Replace	$(\neg x \vee a \vee b)_1$ $(x \vee c \vee d)_2$ $(a \vee b \vee c \vee d)$	by	$(\neg x \vee a \vee b)_1$ $(x \vee c \vee d)_2$ $\cancel{(a \vee b \vee c \vee d)_{12}}$

- Generate resolvents for all variables, remove all clauses they subsume (except themselves)
- If the resolvent in ternary, put it into the learned clause database (Ternary Resolution – [BillionnetS92])

Bounded Variable Elimination (BVE) – [EénBiere-SAT'05]

$$\text{Replace}\quad \begin{array}{ll} (\neg x \vee a)_1 & (x \vee \neg a \vee \neg b)_4 \\ (\neg x \vee b)_2 & (x \vee d)_5 \\ (\neg x \vee c)_3 & \end{array} \quad \text{by} \quad \begin{array}{lll} \cancel{(a \vee \neg a \vee \neg b)}_{14} & (a \vee d)_{15} & (c \vee d)_{35} \\ \cancel{(b \vee \neg a \vee \neg b)}_{24} & (b \vee d)_{25} & \\ (c \vee \neg a \vee \neg b)_{34} & & \end{array}$$

- Most important preprocessor
- A lot of the previous ones are just to make this one work better:
  - Weakening: resolvent more likely to be a tautology
  - Subsumption: fewer resolvents, resolvents can subsume other cls
  - SSTD-Oracle Sparisifcation: see above, but stronger
  - Strengthening: together with subsumption can remove clauses, which can reduce the number of resolvents
  - Vivification: see above, but stronger
  - SSTD-Vivification: see above, but stronger

BVE with gates – [JarvisaloBH11]

Let's try $x = a \wedge b$

$$\text{Replace} \quad \begin{array}{ll} (\neg x \vee a)_1 & (x \vee \neg a \vee \neg b)_3 \\ (\neg x \vee b)_2 & \end{array} \quad \text{by} \quad \begin{array}{l} \cancel{(a \vee \neg a \vee \neg b)_{13}} \\ \cancel{(b \vee \neg a \vee \neg b)_{23}} \end{array}$$

Interesting! Waaaaait a moment. And what if there are other clauses?

$$\text{Replace} \quad \begin{array}{ll} (\neg x \vee a)_1 & (x \vee \neg a \vee \neg b)_4 \\ (\neg x \vee b)_2 & (x \vee c)_5 \\ (\neg x \vee d)_3 & \end{array} \quad \text{by} \quad \begin{array}{lll} \cancel{(a \vee \neg a \vee \neg b)_{14}} & (a \vee c)_{15} & (d \vee \neg a \vee \neg b)_{34} \\ \cancel{(b \vee \neg a \vee \neg b)_{24}} & (b \vee c)_{25} & \cancel{(d \vee c)_{35}} \end{array}$$

Waaaait! Why did $(d \vee c)_{35}$ get removed? Notice: $(d \vee \neg a \vee \neg b) \odot_a (a \vee c)_{15}) \odot_b (b \vee c) = (d \vee c)$

Let's call:

$$\begin{aligned} G_p &= (\neg x \vee a), (\neg x \vee b) \\ G_n &= (x \vee \neg a \vee \neg b) \\ O_p &= (x \vee c) \\ O_n &= (\neg x \vee d) \end{aligned}$$

For most gates, we can add ONLY: $G_p \odot O_n \cup G_n \odot O_p$ — for ITE gates, we also need $G_n \odot G_p$

Gate Constraints — [Tseitin'68]

EQ gate:
$$\begin{aligned} x \leftrightarrow y &\Leftrightarrow (x \to y) \wedge (y \to x) \\ &\Leftrightarrow (x \vee \neg y) \wedge (\neg x \vee y) \end{aligned}$$

OR gate:
$$\begin{aligned} x \leftrightarrow (y \vee z) &\Leftrightarrow (y \to x) \wedge (z \to x) \wedge (x \to (y \vee z)) \\ &\Leftrightarrow (\neg y \vee x) \wedge (\neg z \vee x) \wedge (\neg x \vee y \vee z) \end{aligned}$$

AND gate:
$$\begin{aligned} x \leftrightarrow (y \wedge z) &\Leftrightarrow (x \to y) \wedge (x \to z) \wedge ((y \wedge z) \to x) \\ &\Leftrightarrow (\neg x \vee y) \wedge (\neg x \vee z) \wedge (\neg(y \wedge z) \vee x) \\ &\Leftrightarrow (\neg x \vee y) \wedge (\neg x \vee z) \wedge (\neg y \vee \neg z \vee x) \end{aligned}$$

ITE gate:
$$\begin{aligned} x \leftrightarrow (c\ ?\ t : e) &\Leftrightarrow (x \to (c \to t)) \wedge (x \to (\neg c \to e)) \wedge (\neg x \to (c \to \neg t)) \wedge (\neg x \to (\neg c \to \neg e)) \\ &\Leftrightarrow (\neg x \vee \neg c \vee t) \wedge (\neg x \vee c \vee e) \wedge (x \vee \neg c \vee \neg t) \wedge (x \vee c \vee \neg e) \end{aligned}$$

XOR gate:
$$\begin{aligned} l_1 \oplus l_2 \oplus l_3 = 1 &\Leftrightarrow l_1 \vee l_2 \vee l_3 \wedge \\ &\quad\ \neg l_1 \vee \neg l_2 \vee l_3 \wedge \\ &\quad\ \neg l_1 \vee l_2 \vee \neg l_3 \wedge \\ &\quad\ l_1 \vee \neg l_2 \vee \neg l_3 \wedge \end{aligned}$$