Restart

Runtimes of CDCL-style procedures depend extremely on the choice of branching variables.

If no solution is found within a certain time limit, it can be useful to *restart* from scratch with an adapted variable selection heuristics. Learned clauses, however, are kept.

In addition, it is useful to restart after a unit clause has been learned.

The restart rule is typically applied after a certain number of clauses have been learned or a unit is derived:

Restart:

$M \parallel N \;\Rightarrow_{\text{CDCL}}\; \varepsilon \parallel N$

If Restart is only applied finitely often, termination is guaranteed.

2.8 Implementing CDCL

The formalization of CDCL that we have seen so far leaves many aspects unspecified.

To get a fast solver, we must use good heuristics, for instance to choose the next undefined variable, and we must implement basic operations efficiently.

Variable Order Heuristic

Choosing the right undefined variable to branch is important for efficiency, but the branching heuristics may be expensive itself.

State of the art: Use branching heuristics that need not be recomputed too frequently.

In general: Choose variables that occur frequently; after a restart prefer variables from recent conflicts.

The VSIDS (Variable State Independent Decaying Sum) heuristic:

- We associate a positive *score* to every propositional variable P_i. At the start, k_i is the number of occurrences of P_i in N.
- The variable order is then the descending ordering of the P_i according to the k_i.

The scores k_i are adjusted during a CDCL run.

- Every time a learned clause is computed after a conflict, the propositional variables in the learned clause obtain a bonus b, i.e., $k_i := k_i + b$.
- Periodically, the scores are leveled: $k_i := k_i/l$ for some l.

- After each restart, the variable order is recomputed, using the new scores.

The purpose of these mechanisms is to keep the search focused. The parameter b directs the search around the conflict,

Further refinements:

- Add the bonus to all literals in the clauses that occur in the resolution steps to generate a backjump clause.
- If the score of a variable reaches a certain limit, all scores are rescaled by a constant.
- Occasionally (with low probability) choose a variable at random, otherwise choose the undefined variable with the highest score.

Implementing Unit Propagation Efficiently

For applying the unit rule, we need to know the number of literals in a clause that are not false.

Maintaining this number is expensive, however.

Better approach: *"Two watched literals":*

In each clause, select two (currently undefined) "watched" literals.

For each variable P, keep a list of all clauses in which P is watched and a list of all clauses in which $\neg P$ is watched.

If an undefined variable is set to 0 (or to 1), check all clauses in which P (or $\neg P$) is watched and watch another literal (that is true or undefined) in this clause if possible.

Watched literal information need not be restored upon backtracking.

Preprocessing

Some operations are only needed once at the beginning of the CDCL run.

(i) Deletion of tautologies

(ii) Deletion of duplicated literals

Inprocessing

Some operations are useful, but expensive. They are performed only initially and after restarts (before computation of the variable order heuristics), possibly with time limits.

Note: Some of these operations are only satisfiability-preserving; they do not yield equivalent clause sets.

Examples:

(i) Subsumption

$$N \cup \{C\} \cup \{D\} \;\Rightarrow\; N \cup \{C\}$$

if $C \subseteq D$ considering C, D as multisets of literals.

(ii) Purity deletion

Delete all clauses containing a literal L where $\overline{L}$ does not occur in the clause set.

(iii) Merging replacement resolution

$$N \cup \{C \vee L\} \cup \{D \vee \overline{L}\} \;\Rightarrow\; N \cup \{C \vee L\} \cup \{D\}$$

if $C \subseteq D$ considering C, D as multisets of literals.

(vi) Bounded variable elimination

Compute all possible resolution steps

$$\frac{C \vee L \qquad D \vee \overline{L}}{C \vee D}$$

on a literal L with premises in N; add all non-tautological conclusions to N; then throw away all clauses containing L or $\overline{L}$; repeat this as long as $|N|$ does not grow.

(v) RAT (“Resolution asymmetric tautologies”)

C is called an *asymmetric tautology* w. r. t. N, if its negation can be refuted by unit propagation using clauses in N.

C has the *RAT property* w. r. t. N, if it is an asymmetric tautology w. r. t. N, or if there is a literal L in C such that $C = C' \vee L$ and all clauses $D' \vee C'$ for $D' \vee \overline{L} \in N$ are asymmetric tautologies w. r. t. N.

RAT elimination:

$$N \cup \{C\} \;\Rightarrow\; N$$

if C has the RAT property w. r. t. N.

Literature

Lintao Zhang and Sharad Malik: The Quest for Efficient Boolean Satisfiability Solvers; Proc. CADE-18, LNAI 2392, pp. 295–312, Springer, 2002.

Robert Nieuwenhuis, Albert Oliveras, Cesare Tinelli: Solving SAT and SAT Modulo Theories; From an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T), pp. 937–977, Journal of the ACM, 53(6), 2006.

Armin Biere, Marijn Heule, Hans van Maaren, Toby Walsh (eds.): Handbook of Satisfiability; IOS Press, 2009

Daniel Le Berre's slides at VTSA'09: `http://www.mpi-inf.mpg.de/vtsa09/`.

2.9 OBDDs

Goal:

Efficient manipulation of (equivalence classes of) propositional formulas.

Method: Minimized graph representation of decision trees, based on a fixed ordering on propositional variables.

⇒ Canonical representation of formulas.

⇒ Satisfiability checking as a side effect.

BDD (Binary decision diagram):

Labelled DAG (directed acyclic graph).

Leaf nodes:

labelled with a truth value (0 or 1).

Non-leaf nodes (interior nodes):

labelled with a propositional variable,
exactly two outgoing edges, labelled with 0 (- - →) and 1 (⟶)