Little Engines of Proof

N. Shankar, L. de Moura, H. Ruess, A. Tiwari

shankar@csl.sri.com

URL: http://www.csl.sri.com/~shankar/LEP.html

Computer Science Laboratory
SRI International
Menlo Park, CA

Basic Davis Putnam

Davis Putnam = Unit resolution + Split rule.

$$\frac{\Gamma}{\Gamma, p \mid \Gamma, \neg p}\,\mathit{split} \qquad p \text{ and } \neg p \text{ are not in } \Gamma.$$

$$\frac{C \vee \bar{l}, l}{C, l}\,\mathit{unit}$$

Used in the most efficient SAT solvers.

The final state of *Basic DP* is $\bot$ or a set of configurations (distinct *satisfying assignments*).

Basic DP can be used to enumerate *all satisfying assignments*.

Ex: Prove correctness.

Ex: Show that unit simulates the elim rule of the truth table procedure.

Basic Davis Putnam (example 1)

A DP refutation of $\{p \vee q, \neg p \vee q, p \vee \neg q, \neg p \vee \neg q\}$

$$\begin{array}{c}
p \vee q, \neg p \vee q, p \vee \neg q, \neg p \vee \neg q \\ \hline
p \vee q, \neg p \vee q, p \vee \neg q, \neg p \vee \neg q, p \mid p \vee q, \neg p \vee q, p \vee \neg q, \neg p \vee \neg q, \neg p \\ \hline
p \vee q, q, p \vee \neg q, \neg p \vee \neg q, p \mid p \vee q, \neg p \vee q, p \vee \neg q, \neg p \vee \neg q, \neg p \\ \hline
p \vee q, q, p \vee \neg q, \neg p, p \mid p \vee q, \neg p \vee q, p \vee \neg q, \neg p \vee \neg q, \neg p \\ \hline
\bot \mid p \vee q, \neg p \vee q, p \vee \neg q, \neg p \vee \neg q, \neg p \\ \hline
p \vee q, \neg p \vee q, p \vee \neg q, \neg p \vee \neg q, \neg p \\ \hline
q, \neg p \vee q, p \vee \neg q, \neg p \vee \neg q, \neg p \\ \hline
q, \neg p \vee q, p, \neg p \vee \neg q, \neg p \\ \hline
\bot
\end{array}$$

Ex: Use Basic DP to refute $\{\neg p \vee \neg q \vee r,\ p \vee r,\ q \vee r,\ \neg r\}$.

Basic Davis Putnam (example 2)

A satisfying assignment for $\{\neg p \vee q, p \vee \neg q, \neg p \vee \neg q\}$:

$$\begin{array}{c}
\neg p \vee q, p \vee \neg q, \neg p \vee \neg q \\ \hline
\neg p \vee q, p \vee \neg q, \neg p \vee \neg q, p \mid \neg p \vee q, p \vee \neg q, \neg p \vee \neg q, \neg p \\ \hline
q, p \vee \neg q, \neg p \vee \neg q, p \mid \neg p \vee q, p \vee \neg q, \neg p \vee \neg q, \neg p \\ \hline
q, p \vee \neg q, \neg p, p \mid \neg p \vee q, p \vee \neg q, \neg p \vee \neg q, \neg p \\ \hline
\bot \mid \neg p \vee q, p \vee \neg q, \neg p \vee \neg q, \neg p \\ \hline
\neg p \vee q, p \vee \neg q, \neg p \vee \neg q, \neg p \\ \hline
\neg p \vee q, \neg q, \neg p \vee \neg q, \neg p \\ \hline
\neg q, \neg p \vee \neg q, \neg p
\end{array}$$

Ex: Implement Basic DP.

Ex: Use Basic DP to find a satisfying assignments for: $\{p \vee \neg q, \neg p \vee q, q \vee \neg r, \neg q \vee \neg r\}$, and $\{p \vee q \vee \neg r, p \vee \neg q, \neg p, \neg r, \neg u\}$.

Davis Putnam

The *pure literal rule*, *subsumption*, and *model elimination* are commonly used rules:

$$\frac{\Gamma, C \vee l}{\Gamma}\mathit{pure\_l} \qquad \bar{l} \text{ is not in } \Gamma.$$

$$\frac{C \vee l, l}{l}\mathit{sub}$$

$$\frac{\Gamma, l_1 \vee \ldots \vee l_n}{\Gamma, l_1 \mid \ \ldots \ \mid \Gamma, l_n}\mathit{m\_elim} \qquad l_1, \ldots, l_n \text{ are not in } \Gamma.$$

We say that *non-case-splitting rule* is a *constraint propagation rule*.

Ex: Show the new rules preserve correctness.

Ex: Implement the new rules.

"Lookahead rules"

$\Gamma_1 \vdash^* \Gamma_2$, when Γ_2 is obtained after the application of *zero or more reduction rules* starting at the configuration Γ_1.

LA(1) rule:

$$\frac{\Gamma \quad \Gamma, p \vdash^* \Gamma_1 \quad \Gamma, \neg p \vdash^* \Gamma_2}{\Gamma, \Gamma_1 \cap \Gamma_2}$$

In practice, only *constraint propagation* rules are used in $\vdash^*$.

Example:

$$\dfrac{p \vee q, \neg p \vee q \qquad \dfrac{\dfrac{p \vee q, \neg p \vee q, p}{p \vee q, q, p}}{q, p} \qquad \dfrac{\dfrac{p \vee q, \neg p \vee q, \neg p}{q, \neg p \vee q, \neg p}}{q, \neg p}}{p \vee q, \neg p \vee q, q}$$

"Lookahead" rules (cont.)

LA(2) rule:

$$\frac{\Gamma \quad \Gamma, p, q \vdash^* \Gamma_1 \quad \Gamma, \neg p, q \vdash^* \Gamma_2 \quad \Gamma, p, \neg q \vdash^* \Gamma_3 \quad \Gamma, \neg p, \neg q \vdash^* \Gamma_4}{\Gamma, \Gamma_1 \cap \Gamma_2 \cap \Gamma_3 \cap \Gamma_4}$$

Recursive Learning:

$$\frac{\Gamma, l_1 \vee \ldots \vee l_n \quad \Gamma, l_1 \vdash^* \Gamma_1 \quad \ldots \quad \Gamma, l_n \vdash^* \Gamma_n}{\Gamma, \Gamma_1 \cap \ldots \cap \Gamma_n}$$

Ex: Show that the "lookahead" rules are sound.

Davis Putnam in practice

Depth-first search (*stack* of configurations).

A commonly used strategy is: $\mathit{pure\_l}^*$; $(\mathit{split}$; $\mathit{unit}^*)^*$.

Another strategy is: $\mathit{pure\_l}^*$; $(\mathit{la(1)}^*$; split ; $\mathit{unit}^*)^*$.

A *more efficient* version of the *unit rule* is used: *literals* are not *removed* from clauses.

A clause C is *satisfied* if it contains a *literal* assigned to *true*.

A clause C is a *conflicting clause* if *all literals* are assigned to *false*.

A clause C is a *unit clause* if it is not *satisfied*, and all but one literal are assigned to false.

Davis Putnam in practice (cont.)

The unit rule is "broken" in two rules:

$$\frac{C \vee l}{C \vee l, l}\,unit_{\top}$$ $C \vee l$ is a *unit clause*, and l is *unassigned*.

$$\frac{C}{\perp}\,unit_{\perp}$$ C is a *conflicting clause*.

The term BCP (*boolean constraint propagation*) is usually used to reference the rules $unit_{\top}$ and $unit_{\perp}$.

$unit_{\perp}$ is the $elim$ of the truth table procedure.

The configuration is implemented as a partial assignment.

Davis Putnam in practice (example 1)

A refutation of: $\Gamma \equiv \{\neg p \vee \neg q \vee r,\ p \vee r,\ q \vee r,\ p \vee \neg r,\ \neg p \vee \neg r\}$.

Γ	split
$\Gamma, \neg p \mid \Gamma, p$	$\neg p \vee \neg r$ is a unit clause
$\Gamma, \neg p \mid \Gamma, p, \neg r$	$\neg p \vee \neg q \vee r$ is a unit clause
$\Gamma, \neg p \mid \Gamma, p, \neg r, \neg q$	$q \vee r$ is a conflicting clause
$\Gamma, \neg p \mid \perp$	backtrack
$\Gamma, \neg p$	$p \vee r$ is a unit clause
$\Gamma, \neg p, r$	$p \vee \neg r$ is a conflicting clause
$\perp$	

Davis Putnam in practice (example 2)

The satisfying assignments of: $\Gamma \equiv \{\neg p \vee q, p \vee \neg q\}$:

Γ	split
$\Gamma, \neg p \mid \Gamma, p$	$\neg p \vee q$ is a unit clause
$\Gamma, \neg p \mid \Gamma, p, q$	p, q is a *satisfying assignment*
$\Gamma, \neg p \mid \Gamma, p, q$	backtrack to search next assignment
$\Gamma, \neg p$	$p \vee \neg q$ is a unit clause
$\Gamma, \neg p, \neg q$	$\neg p$, $\neg q$ is a *satisfying assignment*

Davis Putnam in practice (example 3)

Consider the following set of clauses:
$\Gamma \equiv \neg p_1 \vee \neg p_3 \vee \neg p_4, \neg p_1 \vee p_3 \vee \neg p_4, p_2 \vee p_4 \vee q, p_3 \vee p_4, \neg p_3 \vee p_4$

Γ	split
$\Gamma, \neg p_1 \mid \Gamma, p_1$	split
$\Gamma, \neg p_1 \mid \Gamma, p_1, \neg p_2 \mid \Gamma, p_1, p_2$	split
$\Gamma, \neg p_1 \mid \Gamma, p_1, \neg p_2 \mid \Gamma, p_1, p_2, \neg p_3 \mid \Gamma, p_1, p_2, p_3$	$\neg p_1 \vee \neg p_3 \vee \neg p_4$ is unit
$\Gamma, \neg p_1 \mid \Gamma, p_1, \neg p_2 \mid \Gamma, p_1, p_2, \neg p_3 \mid \Gamma, p_1, p_2, p_3, \neg p_4$	$\neg p_3 \vee p_4$ is conflicting
$\Gamma, \neg p_1 \mid \Gamma, p_1, \neg p_2 \mid \Gamma, p_1, p_2, \neg p_3 \mid \perp$	backtrack
$\Gamma, \neg p_1 \mid \Gamma, p_1, \neg p_2 \mid \Gamma, p_1, p_2, \neg p_3$	$\neg p_1 \vee p_3 \vee \neg p_4$ is unit
$\Gamma, \neg p_1 \mid \Gamma, p_1, \neg p_2 \mid \Gamma, p_1, p_2, \neg p_3, \neg p_4$	$p_3 \vee p_4$ is conflicting
$\Gamma, \neg p_1 \mid \Gamma, p_1, \neg p_2 \mid \perp$	backtrack
$\Gamma, \neg p_1 \mid \Gamma, p_1, \neg p_2$	split
$\Gamma, \neg p_1 \mid \Gamma, p_1, \neg p_2, \neg p_3 \mid \Gamma, p_1, \neg p_2, p_3$	$\neg p_1 \vee \neg p_3 \vee \neg p_4$ is unit
...	

Lemma Generation

Lemma generation is a commonly used technique in SAT solvers to avoid *redundant work*.

The configuration is composed of a set of clauses and a *partial function* f from *literals* to *clauses*.

We say the *partial function* f is the *implication graph*.

In a configuration Γ, $f(l) = c$ if the value of l was implied by c using $unit_\top$. We say c is a *justification* for l.

In the initial configuration, the *implication graph* is empty.

Let $f(l := c)$ be the *function update*.

$$\frac{f; C \vee l}{f(l := C \vee l); C \vee l, l}\, unit_\top \qquad C \vee l \text{ is a } \textit{unit clause}\text{, and } l \text{ is } \textit{unassigned}.$$

Lemma Generation (cont.)

Lemma generation rule:

$$\frac{f; C_1 \vee \bar{l}}{f; C_1 \vee \bar{l}, C_1 \vee C_2}\, l\_gen \qquad C_1 \vee \bar{l} \text{ is a } \textit{conflicting clause}\text{, and } f(l) = C_2 \vee l.$$

The l_gen is the *resolution rule*.

The new clause $C_1 \vee C_2$ is also a *conflicting clause*, and it is *implied* by the *initial set of clauses*.

We say $C_1 \vee C_2$ is a *lemma*.

$$\frac{\Gamma_1 \mid \ldots \mid \Gamma_n \mid \Gamma, C}{\Gamma_1, C \mid \ldots \mid \Gamma_n, C \mid \Gamma, C}\, lemma \qquad C \text{ is a } \textit{lemma}.$$

Lemma Generation (cont.)

The *implication graph* can be refined when *lemmas* are inserted in a *configuration*.

Scenario: $C \vee l$ is a lemma in a configuration, where l is true, $f(l)$ is undefined (i.e., l does not have a *justification*), and C is a conflicting clause. So, f can be *refined* because $C \vee l$ is a *justification* for l.

$$\frac{f; C \vee l, l}{f(l := C \vee l); C \vee l}\, refine \qquad f(l) \text{ is undefined, } C \text{ is a } \textit{conflicting clause}.$$

Lemma Generation (example)

Consider (again) the following set of clauses:
$S \equiv \{\neg p_1 \vee \neg p_3 \vee \neg p_4, \neg p_1 \vee p_3 \vee \neg p_4, p_2 \vee p_4 \vee q, p_3 \vee p_4, \neg p_3 \vee p_4\}$
Let f_0 be the empty function (implication graph).

$f_0; S$	*split*
$f_0; S, \neg p_1 \mid f_0; S, p_1$	*split*
$f_0; S, \neg p_1 \mid f_0; S, p_1, \neg p_2 \mid f_0; S, p_1, p_2$	*split*
$\ldots \mid f_0; S, p_1, p_2, \neg p_3 \mid f_0; S, p_1, p_2, p_3$	$unit_\top$
$\ldots \mid f_1 \equiv f_0(\neg p_4 := \neg p_1 \vee \neg p_3 \vee \neg p_4); S, p_1, p_2, p_3, \neg p_4$	l_gen *at* $\neg p_3 \vee p_4$
$\ldots \mid f_1; S_1 \equiv (S, \neg p_1 \vee \neg p_3), p_1, p_2, p_3, \neg p_4$	*lemma*
$f_0; S_1, \neg p_1 \mid \ldots \mid f_1; S_1, p_1, p_2, p_3, \neg p_4$	$unit_\bot$
$f_0; S_1, \neg p_1 \mid \ldots \mid f_0; S_1, p_1, p_2, \neg p_3 \mid \bot$	*backtrack*

$f_0; S_1, \neg p_1 \mid \ldots \mid f_0; S_1, p_1, p_2, \neg p_3$	*refine at* $\neg p_1 \vee \neg p_3$
$\ldots \mid f_2 \equiv f_0(\neg p_3 := \neg p_1 \vee \neg p_3); S_1, p_1, p_2, \neg p_3$	$unit_\top$
$\ldots \mid f_3 \equiv f_2(\neg p_4 := \neg p_1 \vee p_3 \vee \neg p_4); S_1, p_1, p_2, \neg p_3, \neg p_4$	*l_gen at* $p_3 \vee p_4$
$\ldots \mid f_3; S_2 \equiv (S_1, \neg p_1 \vee p_3), p_1, p_2, \neg p_3, \neg p_4$	*l_gen at* $\neg p_1 \vee \neg p_3$
$\ldots \mid f_3; S_3 \equiv (S_2, \neg p_1), p_1, p_2, \neg p_3, \neg p_4$	*lemma at* $\neg p_1$
$f_0; S_1, \neg p_1 \mid f_0; S_1, \neg p_1, p_1, \neg p_2 \mid f_3; S_3, p_1, p_2, \neg p_3, \neg p_4$	$unit_\bot$
$f_0; S_1, \neg p_1 \mid f_0; S_1, \neg p_1, p_1, \neg p_2 \mid \bot$	*backtrack*
$f_0; S_1, \neg p_1 \mid f_0; S_1, \neg p_1, p_1, \neg p_2$	$unit_\bot$
$f_0; S_1, \neg p_1 \mid \bot$	*backtrack*
$f_0; S_1, \neg p_1$	

The following strategy can be used when a conflicting clause is detected:

$l\_gen^* \; ; \; lemma \; ; \; (unit_\bot \; ; \; backtrack)^* \; ; \; refine$

Optimizations

Optimizing the application of $unit_\top$ and $unit_\bot$.

Simple idea: Create a list of *positive* and *negative occurrences* for each propositional variable.

When a *propositional* is assigned to *true*(*false*), only the list of *negative*(*positive*) occurrences need to be visited.

Watch literals: A clause is *irrelevant* if it contains *two or more unassigned literals*. So, each clause is referenced only by two proposition variables.

When a *propositional* is assigned to *true*(*false*), only the list of *negative*(*positive*) *watched* occurrences need to be visited. The watch literals are reassigned.

The number of visited clauses is minimized.

Splitting Heuristics

Perform *case-splits* based on a variable order.

Put *"related" variables* close to each other.

Most important variables first (number of positive/negative occurrences).

Variables that *participate* of several *conflicts* are *important*.

Reorder the variables from time to time.

Randomization (motivation: try to avoid to get stuck in *bad variable order*).

Try to *satisfy* the most recent generated *lemmas*. Remark: This heuristic is not based on the variable order.

Ex: Implement a DP based procedure using the reduction rules described in this lecture.

Solving Real Problems.

Under and *over-constrained* problems are surprisingly *easy*.

Modern *SAT solvers* can handle (*easy*) instances with *hundreds of thousands* variables.

Is SAT *polynomial* in *practice*?

The *hardest problems* are *critically constrained instances*. For hard instances, DP based solvers can only handle something between *400-700 variables*.

Do we find hard instances in practice?

Yes. Example: There is no *polynomial* size *resolution proof* for the *pigeon hole* problem: there is no 1-1 function from m objects (*pigeons*) to n objects (*holes*) if $m > n$.

Ex: Model the *pigeon hole* problem using propositional logic.

Stålmarck Method

Stålmarck Method = *Lookahead* + *equivalence classes*.

Input: *triplets* ($p = l_i \wedge l_j$, and $p = l_i \Leftrightarrow l_j$).

Ex: Write a program to convert a formula into a set of triplets.

Configuration: *triplets*, *equalities between literals*.

Equivalence classes are usually used to represent the set of equalities between literals.

$\dfrac{l_1 = l_2, l_2 = l_3}{l_1 = l_2, l_2 = l_3, l_1 = l_3}\, trans$	$\dfrac{l_1 = l_2}{l_1 = l_2, l_2 = l_1}\, symm$
$\dfrac{l_1 = l_2, l_1 = \bar{l}_2}{\bot}$	

Stålmarck Method: $\wedge$-triplets rules

$\dfrac{p = l_1 \wedge l_2, p = \top}{p = \top, l_1 = \top, l_2 = \top}$	
$\dfrac{p = l_1 \wedge l_2, p = \bar{l}_1}{l_1 = \top, l_2 = \bot, p = \bot}$	$\dfrac{p = l_1 \wedge l_2, p = \bar{l}_2}{l_1 = \bot, l_2 = \top, p = \bot}$
$\dfrac{p = l_1 \wedge l_2, l_1 = \top}{p = l_2, l_1 = \top}$	$\dfrac{p = l_1 \wedge l_2, l_2 = \top}{p = l_1, l_2 = \top}$
$\dfrac{p = l_1 \wedge l_2, l_1 = \bot}{p = \bot, l_1 = \bot}$	$\dfrac{p = l_1 \wedge l_2, l_2 = \bot}{p = \bot, l_2 = \bot}$
$\dfrac{p = l_1 \wedge l_2, l_1 = l_2}{p = l_1, p = l_2, l_1 = l_2}$	$\dfrac{p = l_1 \wedge l_2, l_1 = \bar{l}_2}{p = \bot, l_1 = \bar{l}_2}$

Stålmarck Method: $\Leftrightarrow$-triplets rules

$\dfrac{p = l_1 \Leftrightarrow l_2, l_1 = \top}{p = l_2, l_1 = \top}$	$\dfrac{p = l_1 \Leftrightarrow l_2, l_2 = \top}{p = l_1, l_2 = \top}$
$\dfrac{p = l_1 \Leftrightarrow l_2, l_1 = \bot}{p = \bar{l}_2, l_1 = \bot}$	$\dfrac{p = l_1 \Leftrightarrow l_2, l_2 = \bot}{p = \bar{l}_1, l_2 = \bot}$
$\dfrac{p = l_1 \Leftrightarrow l_2, p = \top}{p = \top, l_1 = l_2}$	$\dfrac{p = l_1 \Leftrightarrow l_2, p = \bot}{p = \bot, l_1 = \bar{l}_2}$
$\dfrac{p = l_1 \Leftrightarrow l_2, l_1 = l_2}{p = \top, l_1 = l_2}$	$\dfrac{p = l_1 \Leftrightarrow l_2, l_1 = \bar{l}_2}{p = \bot, l_1 = \bar{l}_2}$
$\dfrac{p = l_1 \Leftrightarrow l_2, p = l_1}{p = l_1, l_2 = \top}$	$\dfrac{p = l_1 \Leftrightarrow l_2, p = l_2}{p = l_2, l_1 = \top}$
$\dfrac{p = l_1 \Leftrightarrow l_2, p = \bar{l}_1}{p = \bar{l}_1, l_2 = \bot}$	$\dfrac{p = l_1 \Leftrightarrow l_2, p = \bar{l}_2}{p = \bar{l}_2, l_1 = \bot}$

Ex: Show the *triplet* rules are sound.

Stålmarck Method (cont.)

The *triplet* rules are *constraint propagation rules*.

A formula is n-easy if it can be refuted using *LA(n)* and the *triplet* rules.

Strategy: $t\_rules^*; la(1)^*; la(2)^*; la(3)^*; \ldots$.

Stålmarck Method is usually used as an optimization, since it is infeasible to perform $la(n)^*$ ($n \leq 2$) for big formulas.

The Stålmarck Method is a *breadth-first search* procedure.

Remark: the *triplet* rules can be used in a *depth-first search* procedure based on *case-splits*.

Available SAT solvers

ZChaff (http://www.ee.princeton.edu/ chaff/zchaff.php)

Berkmin (http://eigold.tripod.com/BerkMin.html)

Grasp (http://sat.inesc-id.pt/ jpms/grasp/)

Repository of SAT solvers (http://www.satlive.org).

Repository of SAT problems (http://www.satlib.org).

Ex: Try to solve the *challenge problems* located at *satlib* using your SAT solver.