Getting Better Backjump Clauses

By repeating this process, we will eventually obtain a clause that consists only of complements of decision literals and can be used in the "Backjump" rule.

Moreover, such a clause is a good candidate for learning.

Learning Clauses

The DPLL system can be extended by two rules to learn and to forget clauses:

Learn:

$$(M; N) \ \Rightarrow_{\mathrm{DPLL}} \ (M; N \cup \{C\})$$

if $N \models C$.

Forget:

$$(M; N \uplus \{C\}) \ \Rightarrow_{\mathrm{DPLL}} \ (M; N)$$

if $N \models C$.

Learning Clauses

If we ensure that no clause is learned infinitely often, then termination is guaranteed.

The other properties of the basic DPLL system hold also for the extended system.

Restart

Part of the CDCL system the restart rule:

Restart:

$$(M; N) \Rightarrow_{\mathrm{DPLL}} (\mathrm{nil}; N)$$

The restart rule is typically applied after a certain number of clauses have been learned or a unit is derived. It is closely coupled with the variable order heuristic.

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

Variable Order Heuristic

For every propositional variable P_i there is a positive score k_i. At start k_i may for example be 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.

Variable Order Heuristic

- Every time a learned clause is computed after a conflict, the involved propositional variables obtain a bonus b, i.e., $k_i = k_i + b$.
- After each restart, the variable order is recomputed, using the new scores.
- After each j^{th} restart, the scores a leveled: $k_i = k_i/l$ for some l.

The purpose of these mechanisms is to keep the search focused. Parameter b directs the search around the conflict, parameter j decides how many learned clauses are "sufficient" to move in "speed " of parameter l away from this conflict.

Preprocessing

Before DPLL search, and computation of the variable order heuristics, a number of preprocessing steps are performed:

(i) Subsumption

Non-strict version.

(ii) Purity Deletion

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

(iii) Subsumption Resolution

Preprocessing

(iv) Tautology Deletion

(v) Literal Elimination

do all possible resolution steps on a literal L and then throw away all clauses containing L or $\overline{L}$; repeat this as long as $|N|$ does not grow.

Further Information

The ideas described so far have been implemented in all modern SAT solvers: zChaff, miniSAT,picoSAT. Because of clause learning the algorithm is now called CDCL: Conflict Driven Clause Learning.

It has been shown in 2009 that CDCL can polynomially simulate resolution, a long standing open question:

Knot Pipatsrisawat, Adnan Darwiche: On the Power of Clause-Learning SAT Solvers with Restarts. CP 2009, 654-668

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 precedure 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/`.

2.8 Example: Sudoku

	1	2	3	4	5	6	7	8	9
1								1	
2	4								
3		2							
4					5		4		7
5			8				3		
6			1		9				
7	3			4			2		
8		5		1					
9				8		6			

Idea: $p^d_{i,j}$=true iff the value of square i, j is d

For example:

$p^8_{3,5} = true$

Coding Sudoku by Propositional Clauses

- Concrete values result in units: $p^d_{i,j}$
- For every square (i, j) we generate $p^1_{i,j} \vee \ldots \vee p^9_{i,j}$
- For every square (i, j) and pair of values $d < d'$ we generate $\neg p^d_{i,j} \vee \neg p^{d'}_{i,j}$
- For every value d and column i we generate $p^d_{i,1} \vee \ldots \vee p^d_{i,9}$
  (Analogously for rows and 3×3 boxes)
- For every value d, column i, and pair of rows $j < j'$ we generate $\neg p^d_{i,j} \vee \neg p^d_{i,j'}$
  (Analogously for rows and 3×3 boxes)

Constraint Propagation is Unit Propagation

	1	2	3	4	5	6	7	8	9
1								1	
2	4								
3		2							
4					5		4		7
5			8				3		
6			1		9				
7	3			4	**7**		2		
8		5		1					
9				8		6			

From $\neg p^3_{1,7} \vee \neg p^3_{5,7}$ and $p^3_{1,7}$ we obtain by unit propagating $\neg p^3_{5,7}$ and further from $p^1_{5,7} \vee p^2_{5,7} \vee p^3_{5,7} \vee p^4_{5,7} \vee \ldots \vee p^9_{5,7}$ we get $p^1_{5,7} \vee p^2_{5,7} \vee p^4_{5,7} \vee \ldots \vee p^9_{5,7}$ (and finally $p^7_{5,7}$).

2.9 Other Calculi

OBDDs (Ordered Binary Decision Diagrams):

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

$\Rightarrow$ canonical representation of formulas.

see script of the Computational Logic course,

see Chapter 6.1/6.2 of Michael Huth and Mark Ryan: *Logic in Computer Science: Modelling and Reasoning about Systems*, Cambridge Univ. Press, 2000.

Other Calculi

FRAIGs (Fully Reduced And-Inverter Graphs)

Minimized graph representation of boolean circuits.

$\Rightarrow$ semi-canonical representation of formulas.

Implementation needs DPLL (and OBDDs) as subroutines.

Other Calculi

Tableau calculus

Hilbert calculus

Sequent calculus

Natural deduction

Part 3: First-Order Logic

First-order logic

- formalizes fundamental mathematical concepts
- is expressive (Turing-complete)
- is not too expressive (e. g. not axiomatizable: natural numbers, uncountable sets)
- has a rich structure of decidable fragments
- has a rich model and proof theory

First-order logic is also called (first-order) predicate logic.

3.1 Syntax

Syntax:

- non-logical symbols (domain-specific)
  $\Rightarrow$ terms, atomic formulas
- logical connectives (domain-independent)
  $\Rightarrow$ Boolean combinations, quantifiers

Signature

A signature $\Sigma = (\Omega, \Pi)$ fixes an alphabet of non-logical symbols, where

- Ω is a set of function symbols f with arity $n \geq 0$, written $\mathrm{arity}(f) = n$,
- Π is a set of predicate symbols P with arity $m \geq 0$, written $\mathrm{arity}(P) = m$.

Function symbols are also called operator symbols.
If $n = 0$ then f is also called a constant (symbol).
If $m = 0$ then P is also called a propositional variable.

Signature

We will usually use

b, c, d for constant symbols,

f, g, h for non-constant function symbols,

P, Q, R, S for predicate symbols.

Convention: We will usually write $f/n \in \Omega$ instead of $f \in \Omega$, $\text{arity}(f) = n$ (analogously for predicate symbols).

Signature

Refined concept for practical applications:
many-sorted signatures (corresponds to simple type systems in programming languages); not so interesting from a logical point of view.

Variables

Predicate logic admits the formulation of abstract, schematic assertions. (Object) variables are the technical tool for schematization.

We assume that X is a given countably infinite set of symbols which we use to denote variables.

Context-Free Grammars

We define many of our notions on the bases of context-free grammars. Recall that a context-free grammar $G = (N, T, P, S)$ consists of:

- a set of non-terminal symbols N
- a set of terminal symbols T
- a set P of rules $A ::= w$ where $A \in N$ and $w \in (N \cup T)^*$
- a start symbol S where $S \in N$

For rules $A ::= w_1$, $A ::= w_2$ we write $A ::= w_1 \mid w_2$

Terms

Terms over Σ and X (Σ-terms) are formed according to these syntactic rules:

$$
\begin{array}{lcll}
s, t, u, v & ::= & x & , x \in X \qquad\qquad \text{(variable)} \\
& | & f(s_1, \ldots, s_n) & , f/n \in \Omega \quad \text{(functional term)}
\end{array}
$$

By $\mathrm{T}_\Sigma(X)$ we denote the set of Σ-terms (over X). A term not containing any variable is called a ground term. By T_Σ we denote the set of Σ-ground terms.

Terms

In other words, terms are formal expressions with well-balanced brackets which we may also view as marked, ordered trees. The markings are function symbols or variables. The nodes correspond to the subterms of the term. A node v that is marked with a function symbol f of arity n has exactly n subtrees representing the n immediate subterms of v.