LEARNSAT

User's Guide

Version 1.0.0

Mordechai (Moti) Ben-Ari
http://www.weizmann.ac.il/sci-tea/benari/

April 17, 2012

1 Introduction

LEARNSAT is a program for learning about SAT solving. It implements the classic *Davis-Putnam-Logemann-Loveland (DPLL)* algorithm for SAT solving, together will modern extensions of the algorithm: *conflict-driven clause learning (CDCL)* and *non-chronological backtracking (NCB)*.

For a gentle introduction to SAT solvers, see [1, Chapter 6]. The comprehensive reference is the *Handbook of Satisfiability* [2]. The algorithms and notation of LEARNSAT follow [4].

The design of LEARNSAT is based on the following principles:

- A very detailed trace of the algorithm's execution can be displayed and the specific content of the trace can be set by the user.
- The PROLOG language is used so that the programs will be very concise and easy to understand. The source code includes extensive comments. Very little knowledge of PROLOG is needed just to run the program.
- The distribution includes test programs, including those taken from published descriptions of the algorithms.
- A utility is provided for converting between sets of clauses represented as PROLOG terms and those in DIMACS format. While the former are easier to read, existing examples in DIMACS format can be downloaded and converted.
- LEARNSAT is an open-source project.

2 Downloading

LEARNSAT can be downloaded from Google Code at:[1]

`http://code.google.com/p/mlcs/`.

Download the zip archive `learnsat-N.zip` which has all the files in a single directory.

The programs were developed using SWI-PROLOG Version 6.0.0:

`http://www.swi-prolog.org/`.

The source files use the extension `pro` instead of the more usual `pl` to avoid conflict with programs in Perl. During the installation of SWI-PROLOG, if you associate the extension `pro` with SWI-PROLOG, a program can be launched by double-clicking on its name in a file list.

[1] This project contains an archive of PROLOG programs for the algorithms in my textbook *Mathematical Logic for Computer Science (Third Edition)* [1].

3 Running LEARNSAT

3.1 Preparing a file to check satisfiability

To use LEARNSAT, prepare a PROLOG program that calls the predicate `dpll` with a set of clauses structured as a list of lists of literals. Here is an extract from the file `pigeon.pro`; it expresses the pigeon-hole problem for two holes and three pigeons, where `pij` means that pigeon `i` is in hole `j`:

```
:- ensure_loaded([negation,display,dpll]).

hole2 :-
  dpll(
  [
  [p11, p12],   [p21, p22],   [p31, p32],   % Each pigeon in hole 1 or 2
  [~p11, ~p21], [~p11, ~p31], [~p21, ~p31], % No pair in hole 1
  [~p12, ~p22], [~p12, ~p32], [~p22, ~p32], % No pair in hole 2
  ], _).
```

The first line requests that the modules of LEARNSAT be loaded; then, a predicate `hole2` is defined. It calls `dpll` with the set of clauses. The result can be returned as the second argument, but we have left it anonymous since we are interested in the display of the execution trace.

3.2 Loading and running a file

Once this file has been loaded (by double-clicking or by entering `[pigeon]` at the PROLOG prompt), the predicate `hole2` can be run. The output will be a trace of the DPLL algorithm:

```
1 ?- hole2.
LearnSAT v1.0.0. Copyright 2012 by Moti Ben-Ari. GNU GPL.
Decision assignment: p11=0@1
Propagate unit p12 derived from: 1. [p11,p12]
Propagate unit ~p22 derived from: 7. [~p12,~p22]
Propagate unit p21 derived from: 2. [p21,p22]
Propagate unit ~p31 derived from: 6. [~p21,~p31]
Propagate unit p32 derived from: 3. [p31,p32]
Conflict clause: 8. [~p12,~p32]
Decision assignment: p11=1@1
                ...
Unsatisfiable
Clauses=9, variables=6, units propagated=56, choices=12, conflicts=12

true.
```

All PROLOG queries must be terminated by a period and after the execution terminates with `true` you must press return to get a new prompt.
The trace output can be directed to a file:

```
hole2_file :- tell('hole2.txt'), hole2, told.
```

3.3 Algorithmic mode

LEARNSAT can run in three modes:

- `dpll`: DPLL algorithm (this is the default);
- `cdcl`: DPLL with conflict-directed clause learning;
- `ncb`: DPLL with CDCL and non-chronological backtracking.

Set the mode as follows:

```
?- set_mode(ncb).
```

The pigeon-hole program runs much more efficiently in `cdcl` mode:

```
Clauses=9, variables=6, units propagated=29, choices=12, conflicts=12
```

3.4 Display options

You can control the content of the trace output of LEARNSAT with the predicates `set_display` and `clear_display`. The argument to these predicates can be `all`, `default` or a single option or a list of options taken from table in the Appendix.

3.5 Online help

The predicate `usage` shows the commands, and the mode and display arguments.
The version, the default and current mode, and the default and current display options can be shown by running `show_config`.

3.6 Changing the defaults

If you need to change the mode and options frequently, you can write a predicate to do so:

```
init :-
  set_display(all),
  clear_display([graph,incremental,dot]),
  set_mode(ncb).
```

Alternatively, the defaults can be changed by editing the following lines in the file `config.pro`:

```
default_mode(dpll).
default_display([backtrack, conflict, decision, learned, result, unit]).
```

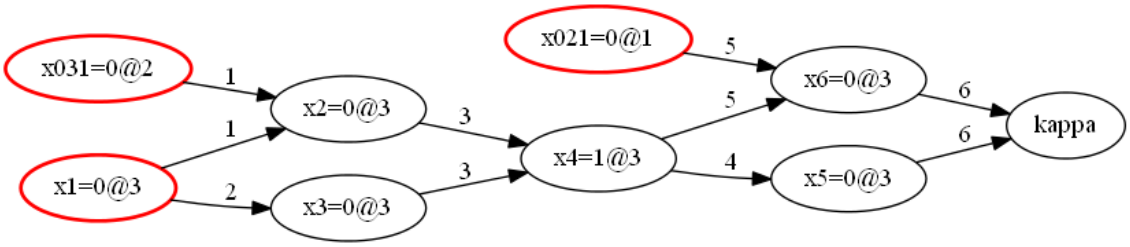

Figure 1: Implication graph

3.7 Implication graphs

LEARNSAT constructs the implication graph when a conflict clause is encountered. The trace output can display a list of the nodes and edges of the final graph, as well as of the graphs that are built during each incremental stage of the construction.

Files with the implication graphs in DOT format can be written by LEARNSAT. The graphs can then be rendered using the DOT tool of GRAPHVIZ (`http://www.graphviz.org/`):

```
dot -Tpng examples-graph-0.dot > examples-graph-0.png
```

Figure 1 shows the graph that was generated from running LEARNSAT on the example from [4]; the clauses for this example are:

```
1. [x1,x031,~x2]
2. [x1,~x3]
3. [x2,x3,x4]
4. [~x4,~x5]
5. [x021,~x4,~x6]
6. [x5,x6]
```

3.8 Test programs

Several test program are included in the LEARNSAT archive.

- `queens.pro` contains the four-queens problem as given in [1, Section 6.4].
- `tseitin.pro` contains the Tseitin clauses for the graphs $K_{2,2}$, $K_{3,3}$ and the example from [1, Section 4.5]. The sets of clauses are unsatisfiable, but, for each set of clauses, a satisfiable variant is given; these were obtained by complementing one literal in the set.
- `pigeon.pro` contains the pigeon-hole problem for two and three holes [1, Exercise 6.4].
- `examples.pro` contains examples from papers on SAT solvers [3, 4, 5].

3.9 DIMACS transformation

The predicates in the file `dimacs.pro` convert a set of clauses in PROLOG format (a list of lists of literals) to and from *simplified DIMACS cnf format* that is used by SAT solvers.

- `to_dimacs(File, Comment, Clauses)` converts the PROLOG `Clauses` into DIMACS format and writes them to the `File` with the `Comment`.
- `from_dimacs(Predicate, InFile, OutFile)` reads `InFile` in DIMACS format and converts the data into a program written on `OutFile`. `Predicate` is the name to be given to the predicate for running `dpll`:

  ```
  Predicate :- dpll(List, _).
  ```

The test program `dimacs-test.pro` transforms the set of clauses for the four-queens problem from PROLOG clauses to DIMACS and back.

4 Software documentation

4.1 Module structure

The LEARNSAT software consists of the following source files:[2]

- `negation.pro` is a file that declares the negation operator:

  ```
  :- op(610, fy,  ~).
  ```

- `dpll.pro` is the main module for the DPLL algorithms. It is described in more detail below.

- `io.pro` is a module that writes: (a) assignments in the format `p1=0@3` used in [4]; (b) clauses together with their position numbers from the list of clauses input to the program; (c) implication graphs.

- `counters.pro` is the module that maintains and displays counters for units, choices and conflicts, as well as a counter for adding a number to the file names for implication graphs.

- `display.pro` is the module that maintains the mode and display options, and writes the data. The main predicate is `display/2,3,4`, whose first argument is a display option and whose additional arguments supply the data to be displayed. The dummy display option `none` is used to distinguish between the initial state (no options, so set the default options) and a state where all options have been cleared.

- `config.pro` is a module that facilitates changing the default options. It consists of just three facts: `version`, `default_mode` and `default_display`.

4.2 The DPLL algorithm

The predicate `dpll` implements the DPLL algorithm on a set of clauses represented as a list of lists of literals. It always succeeds, returning either a list of satisfying assignments or the empty list if the clauses are unsatisfiable. As part of its initialization, the set of variables in the clauses is obtained from the list.

The predicate `dpll` invokes `dpll1` which is the main recursive predicate for performing the algorithm. If the set of variables is empty, the set of clauses is unsatisfiable. Otherwise, `dpll1` tries to perform unit propagation by searching for a unit and then evaluating the set of clauses. When no more units remain, it chooses a decision assignment and evaluates the set of clauses.

The predicate `ok_or_conflict` is called with the result of the evaluation of unit propagation or the choice of an assignment. If the result is not a conflict clause, the variable chosen is deleted and `dpll1` is called recursively. If there was a conflict clause, the implication graph is constructed and a learned clause is generated from the graph; then `ok_or_conflict` fails so that backtracking can try a new assignment.

[2]The list does not including the test programs and the program for DIMACS conversion discussed above.

4.3 Conflict-directed clause learning

See [4] for a detailed description of CDCL.

An *implication graph* represents the result of unit propagation that causes a conflict. Consider the graph in Figure 1: There are three source nodes, one for each of the decision assignments (to `x021`, `x031`, `x1`) that are sufficient to cause a conflict by unit propagation. The conflict is represented by the sink node `kappa` and the assignments that were forced by unit propagation are represented by nodes labeled with the assignments. The edges of the graph are labeled with (the numbers of) the clauses that are *antecedents* of each node; these are the unit clauses that forced the assignment labeling that node. There is an edge for each literal in the unit clause (except the one whose assignment is forced) and the sources of these edges are the assignments to those literals. For example, the decision assignments at levels 2 and 3 to `x031` and `x1` cause clause 1 (`[x1, x031, ~x2]`) to become a unit clause and force the assignment of 0 to `x2`.

In LEARNSAT, the implication graph is built incrementally. Whenever a unit clause is found, the predicate `get_graph` is called with the unit clause, its number (to label the new edges), the assignment it forces (to create the new target of the edges) and the graph constructed so far. For each literal (except the one implied), a new edge is created and when the list of literals has been traversed, the new node is created. When a conflict is encountered, the `kappa` node and its incoming edges are added and the graph is passed to `compute_learned_clause`.

The predicate `compute_learned_clause` starts with the conflict clause (the antecedent clause of the `kappa` node) as the current clause and starts resolving. The second clause for each resolution step is one that clashes with the current clause on a literal that is assigned at the current level *by unit propagation*. For example, given the conflict clause `[x5,x6]` from Figure 1, both `x5` and `x6` were assigned by unit propagation at this level, so we can successively resolve with the antecedent clauses of the assignments `[~x4,~x5]` and `[x021,~x4,~x6]`. The resolution terminates when a *unique implication point (UIP)* is encountered; this is a clause where only one literal is assigned at the current level. This clause is the *learned clause* and is added to the list that is the argument of the single fact in the database `learned`. In the example, the learned clause is `[x021,~x4]`.

When a learned clause has been obtained, the backtrack level for non-chronological backtracking is computed. This is the highest level of an assignment in the learned clause except for the current level. The level is stored in a dynamic database `backtrack` and is used when choosing an assignment to avoid choosing variables at a lower level than the backtrack level. In the example, the highest level is 1 where `x021` was assigned.

4.4 Auxiliary predicates

The following simple auxiliary predicates are used:

- `evaluate` a set of clauses; returns `ok` or `conflict`.
- `evaluate_clause` returns one of `satisfied`, `unsatisfied`, `unit`, `unresolved`.
- `choose_assignment` and `choose_value` return an assignment. Backtracking is used to return subsequent choices.

The only delicate part is the use of the predicate *Condition* $\rightarrow$ *Action* with an internal `!, fail` to implement non-chronological backtracing; see the comments and Section 4.7 of the SWI-PROLOG Reference Manual.

- `is_assigned` checks if a literal is assigned a value; if so, the value of the literal (0 or 1) is returned.
- `resolve` performs resolution on two clauses.
- `literals_to_variables` takes a list of literals and returns a sorted set of the variables corresponding to the literals.
- `to_variable` returns the variable of a literal and `to_complement` returns the complement of a literal.
- `to_assignment` and `to_literal` convert to and from a literal and an assignment expressed as a term `assign(Variable, Value, Level, Decision)`, where `Level` is the level at which the assignment was made and `Decision` is `yes` if the assignment was made as a decision and `no` if the assignment was the result of unit propagation.

References

[1] M. Ben-Ari. *Mathematical Logic for Computer Science (Third Edition)*. Springer, 2012. (forthcoming).

[2] A. Biere, M. Heule, H. Van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*. Frontiers in Artificial Intelligence and Applications. IOS Press, 2009.

[3] S. Malik and L. Zhang. Boolean satisfiability from theoretical hardness to practical success. *Commun. ACM*, 52(8):76–82, 2009.

[4] J. P. Marques-Silva, I. Lynce, and S. Malik. *Conflict-Driven Clause Learning SAT Solvers*, chapter 4, pages 131–153. In Biere et al. [2], 2009.

[5] J. P. Marques-Silva and K. A. Sakallah. Grasp—a new search algorithm for satisfiability. In *Proceedings of the 1996 IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '96, pages 220–227, 1996.

A Reference

Predicates	
`dpll`	Run the DPLL algorithm
`usage`	Show the predicates, modes and display options
`show_config`	Show the current mode and display options
`set_display`	Set display options
`clear_display`	Clear display options
`set_mode`	Set the algorithmic mode
`to_dimacs`	Convert from PROLOG term to DIMACS
`from_dimacs`	Convert from DIMACS to PROLOG term

Modes	
`dpll`	DPLL algorithm (default)
`cdcl`	DPLL with conflict-directed clause learning
`ncb`	DPLL with CDCL and non-chronological backtracking

Display options	
`all`	all of the options
`default`	default options (indicated by *)
`assignments`	assignments that caused a conflict
`backtrack*`	level of non-chronological backtracking
`clauses`	clauses to be checked for satisfiability
`conflict*`	conflict clauses
`decision*`	decision assignments
`dot`	implication graphs in dot format
`graph`	implication graphs
`incremental`	incremental build of the implication graphs
`learned*`	learned clauses
`literal`	literals found assigned during CDCL
`resolvent`	resolvents created during CDCL
`result*`	result of the algorithm
`skipping`	assignments skipped when backtracking
`uip`	unique implication points
`unit*`	unit clauses
`variables`	variables that are not assigned so far