LEARNSAT
User's Guide and Software Documentation

Moti Ben-Ari

http://www.weizmann.ac.il/sci-tea/benari/

Version 2.0

1 Overview

LEARNSAT is a program for learning about SAT solving. It implements the *Davis-Putnam-Logemann-Loveland (DPLL)* algorithm with *conflict-driven clause learning (CDCL)*, *non-chronological backtracking (NCB)* and *lookahead*. For a gentle introduction to SAT solvers, see [1, Chapter 6]. The comprehensive reference is the *Handbook of Satisfiability* [2].

The design of LEARNSAT is based on the following principles:

- A detailed trace of the algorithm's execution is displayed. Its content can be set by the user.
- Graphical representations of implication graphs and assignment trees are generated automatically.
- The implementation is in PROLOG so that the program will be concise and easy to understand. Sections 10–15 document the software and the program has extensive comments.
- LEARNSAT is an open-source project.
- The software is easy to install and use.
- There is a *tutorial* on SAT solving with LEARNSAT and many example programs are provided.

2 Installation

LEARNSAT can be found on my website:

`http://www.weizmann.ac.il/sci-tea/benari/software-and-learning-materials/`.

Download and unzip the archive `learnsat-N.zip`. The PROLOG source code is in the directory `src`, programs are in the directory `examples` and the documentation is in the directory `docs`.

Download and install SWI-PROLOG:

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

The source files use the extension `pro` instead of the more usual `pl` to avoid conflicts. 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.

The directory `docs` contains the PDF and LaTeX files for this document, an overview and the tutorial. It also includes a file `version.tex` to facilitate keeping the version numbers of the two documents consistent. The file `tikz.tex` contains the source code of diagrams that were generated using the Ti*k*Z graphics package. The tutorial includes the generated PDF files (renamed for clarity) so that the it can be modified without installing the package.

3 Running LEARNSAT

The main module is in the file `dpll.pro`. It exports the following predicates which are the only predicates that you need to run LEARNSAT:

Predicates	
`dpll`	Run the DPLL algorithm
`set_display`	Set display options
`clear_display`	Clear display options
`set_mode`	Set the algorithmic mode
`set_look`	Set the lookahead mode
`set_decorate_mode`	Set decoration as color or black-and-white
`set_order`	Set the variable assignment order
`show_config`	Show the current mode and display options
`usage`	Show the modes and display options

To check the satisfiability of a CNF formula, create a PROLOG program that calls the predicate `dpll` with the clausal form of the formula represented as a list of lists of literals. The predicate `use_module` specifies the location of the LEARNSAT source code files. Alternatively, the LEARNSAT source files can be copied to the directory with the program.

The file `pigeon.pro` contains programs for the pigeonhole principle; here is the program for two holes and three pigeons, where `pij` means that pigeon `i` is in hole `j`:

```
:- use_module('../src/dpll').

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

The result (a satisfying assignment or `[]` if unsatisfiable) is returned as the second argument, but can be left anonymous if only the trace is of interest.

Once this file has been loaded (by double-clicking or by consulting `[pigeon]`), the query:

```
?- hole2.
```

can be run. It reports that the clauses are unsatisfiable. After it terminates with `true`, press return to get a new prompt.

The output will be a trace of the DPLL algorithm:

```
1 ?- hole2.
LearnSAT (version 2.0)
Decision assignment: p11=0
Propagate unit:  p12 (p12=1) derived from: [p11,p12]
Propagate unit: ~p22 (p22=0) derived from: [~p12,~p22]
Propagate unit:  p21 (p21=1) derived from: [p21,p22]
Propagate unit: ~p31 (p31=0) derived from: [~p21,~p31]
Propagate unit:  p32 (p32=1) derived from: [p31,p32]
Conflict clause: [~p12,~p32]
Decision assignment: p11=1
Propagate unit: ~p21 (p21=0) derived from: [~p11,~p21]
Propagate unit:  p22 (p22=1) derived from: [p21,p22]
Propagate unit: ~p31 (p31=0) derived from: [~p11,~p31]
Propagate unit:  p32 (p32=1) derived from: [p31,p32]
Conflict clause: [~p22,~p32]
Unsatisfiable:
Statistics: clauses=9, variables=6, units=9, decisions=2, conflicts=2
true.
```

The trace output can be directed to a file:

```
?- tell('hole2.txt'), hole2, told.
```

4 Controlling the algorithm

There are predicates that set the mode of the algorithm and the display options. If you need to set a specific mode and set of display options when writing and experimenting with a program, you can write a predicate that you can run whenever LEARNSAT is initiated.

```
set_my_modes :-
  set_mode(ncb),
  set_mode(look),
  set_display([default, dominator, dot, label, look]).
```

4.1 Algorithmic mode

LEARNSAT can run in one of three modes set by the predicate `set_mode`:

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

For the three-layer grid-pebbling problem, the statistics for the three modes are:

```
dpll: units=74, decisions=50, conflicts=26
cdcl: units=25, decisions=14, conflicts=8, learned clauses=5
ncb:  units=17, decisions=9,  conflicts=3, learned clauses=3
```

The predicate `set_look` is used to set whether lookahead is used or not:

`none`	Disable lookahead (default)
`current`	Lookahead based on current clauses
`original`	Lookahead based on original clauses

With `current` lookahead enabled the statistics for the above problem are:

```
dpll: units=21, decisions=10, conflicts=6
cdcl: units=18, decisions=8,  conflicts=5, learned clauses=5
ncb:  units=16, decisions=6,  conflicts=3, learned clauses=3
```

The lookahead algorithm is to choose the variable with the largest number of occurrences in the set of clauses resulting from evaluating them under the `current` partial assignment or in the `original` set of clauses. In both cases, learned clauses are taken into account.

4.2 Order of the variables

By default (and when lookahead is not enabled), decision assignments are made in lexicographical order of the atomic propositions. The order can be specified by using the predicate `set_order`. The list argument to `set_order` must be a permutation of the variables in the clauses. To restore the default order run `set_order(default)`.

The `mlm` example finds a satisfying assignment *without* conflicts if the default order is changed:

```
set_order([x1,x2,x021,x031,x3,x4,x5,x6]).
```

`antecedent`	antecedents of the implied literals
`assignment`	assignments that caused a conflict
`backtrack` *	level of non-chronological backtracking
`clause`	current set of clauses
`conflict` *	conflict clauses
`decision` *	decision assignments
`dominator`	computation of the dominator
`dot`	implication graphs (final) in dot format
`dot_inc`	implication graphs (incremental) in dot format
`graph`	implication graphs (final) in textual format
`incremental`	implication graphs (incremental) in textual format
`label`	dot graphs and trees labeled with clauses
`learned` *	learned clause by resolution
`look`	occurrences of variables for lookahead
`partial`	partial assignments so far
`resolvent` *	resolvents created during CDCL
`result` *	result of the algorithm with statistics
`skipped` *	assignments skipped when backtracking
`sorted` *	assignments displayed in sorted order
`tree`	trees of assignments (final) in dot format
`tree_inc`	trees of assignments (incremental) in dot format
`uip` *	unique implication points
`unit` *	unit clauses
`variables`	variables that are not assigned so far

Table 1: Display options

5 Display options

LEARNSAT writes extensive trace output as it executes the algorithms. Each line starts with a string followed by a colon to make it easy to postprocess the output. The content of the trace is controlled using `set_display` and `clear_display`. The argument to these predicates can be `all` or `default`, or a single option or a list of options from Table 1 (where * denotes the default options).

If `default` appears first in a list of display options, the others options will be added to the default ones.

The display options depend on the algorithmic mode. If `dpll` is chosen, there are no learned clauses and the option `learn` has no effect. When lookahead is enabled, the display option `look` shows the number of occurrences of each variable when the clauses are evaluated under the current partial assignment. These clauses will be shown if the display option `partial` is enabled.

In `cdcl` and `ncb` modes, the assignments are written together with their levels in the format `p1=0@3`.

`antecedent` displays an implied assignment together with its antecedent clause : `p1@3/[~p1,p3]`.

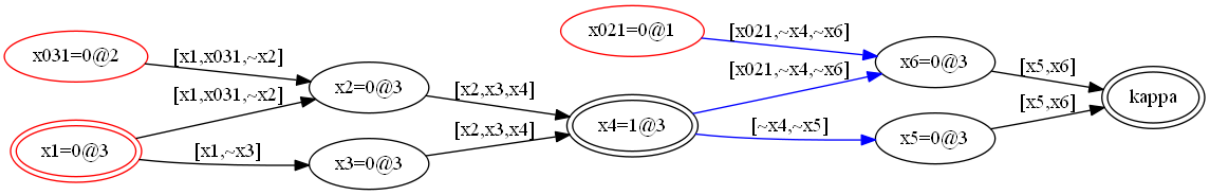
6 Configuration

`config.pro` contains: the default modes, the default display options, and the DOT prologue and decorations. The predicate `decorate_mode` determines which family of decorations will be used.

`show_config` displays: the version and copyright notice, the defaults from `config.pro`, changes from the default values, and the variable order.

7 Implication graphs

When a conflict clause is encountered, LEARNSAT generates the implication graph. Display option `graph` generates a textual representation, while `dot` generates a graphical representation. Display option `label` displays the antecedent clauses on each edge, not just their numbers. Display option `dominator` emphasizes the following nodes: the decision assignment at the current level, the dominator and `kappa`.



The graph can be displayed with color (default) or black-and-white decoration:

```
set_decorate_mode(bw).
```

The decorations are: decision assignments in red or bold; the decision assignment at the highest level, the `kappa` node and the dominator node with double borders; the cut with blue or dashed lines.

Display options `incremental` and `dot_inc` generate graphs after each step of the algorithm.

The graphics files in DOT format are rendered using GRAPHVIZ (`http://www.graphviz.org/`):

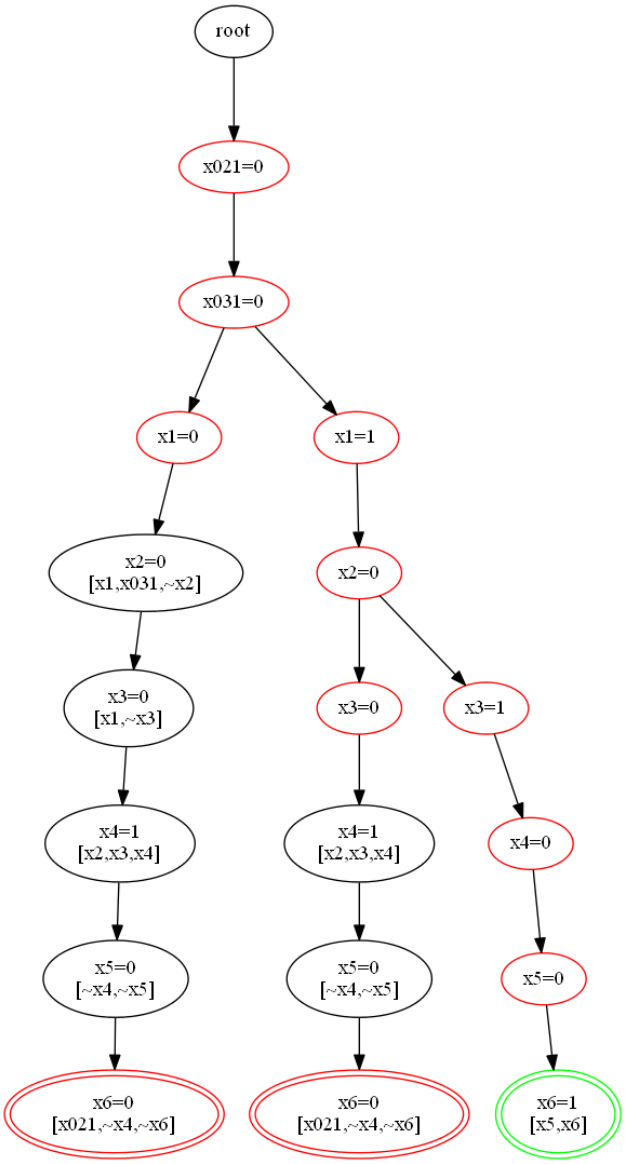
```
dot -Tpng examples-ig-00.dot > examples-ig-00.png
```

A script can be used to run DOT on all the generated files; in Windows, the batch command is:

```
for %%F in (*.dot) do c:\"program files"\graphviz\bin\dot -Tpng %%F > %%~nF.png
```

8 Trees of assignments

A tree showing the assignments to variables is generated by selecting display option `tree` (next page). Trees are shown as directed acyclic graphs whose nodes are assignments rather than variables. Decision nodes are decorated with a red (or bold) border and conflict nodes with a double red (or bold) border. A double green (or triple) border indicates the node (if any) that causes the formula to become satisfiable. The rendering of the DOT files is as described above for the implication graphs.



9 DIMACS transformation

The predicates in `dimacs.pro` convert a set of clauses in PROLOG format (a list of lists of literals) to and from *DIMACS cnf format*:

- `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 writes a PROLOG program to `OutFile` as `Predicate :- dpll(List, _).`

10 Module structure

The LEARNSAT software consists of the following source files (not including the test programs and the program for DIMACS conversion):

- `dpll.pro`: Main module.
- `cdcl.pro`: Algorithms for CDCL and the implication graph.
- `auxpred.pro`: Auxiliary predicates for the algorithms.
- `io.pro`: Predicates for writing assignments, clauses and implication graphs.
- `display.pro`: Display the trace using predicates `display/n`, where the first argument is a display option and the additional arguments supply the data to be displayed.
- `config.pro`: Default configuration data.
- `counters.pro`: Maintains counters for the number of clauses, variables, units, decisions and conflicts. These are used for printing the statistics. Defines counters for adding a number to the file names for implication graphs and trees of assignments.
- `modes.pro`: Sets, clears and checks the modes and the display options. Implements `usage` and `show_config`. 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.
- `dot.pro`: Generate the DOT files of the implication graphs and the assignment trees.

11 Data structures

The following dynamic predicates are used:

- In `dpll.pro`: The counts of variable `occurrences` when `original` lookahead is enabled.
- In `cdcl.pro`: The non-chronological backtracking level in the `backtrack`.
- In `cdcl.pro`: The learned clauses are stored as a list in `learned`.
- In `auxpred.pro`: If the user specifies the order of assignment to variables, the list is stored in `variables_list`.
- In `dot.pro`: `node` and `edge` are used when building the assignment trees. They store the paths leading to previous conflict nodes and are updated with each new path.
- In `mode.pro`: `alg_mode`, `look_mode`,`decorate_mode`, `display_option`.
- In `counters.pro`: Counters for reporting statistics and for generating `dot` file names.

Two functors are used to create terms:

- `assign` represents an assignment and takes four arguments: a variable, its value, its level and either `yes` if this is a decision assignment or the antecedent clause of an implied assignment.
- `graph` is an implication graph. Its arguments are a list of nodes which are assignments and a list of edges which are terms with functor `edge` and three arguments: the source and target nodes and the clause that labels the edge.

The initial list of clauses is retained and completely re-evaluated as assignments are added. This is not efficient but it facilitates the display of the trace of the algorithms.

12 The DPLL algorithm

`dpll/2` implements the DPLL algorithm on a set of clauses represented as a list of lists of literals. It returns 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.

`dpll/2` invokes `dpll/6` which is the main recursive predicate for performing the algorithm. If the set of variables to be assigned to is empty, the set of clauses is satisfiable. Otherwise, `dpll/6` 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.

`ok_or_conflict` is called with the result of the evaluation of unit propagation or the decision assignment. If the result is not a conflict, the variable chosen is deleted and `dpll/6` is called recursively. If there was a conflict, 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.

`evaluate` receives a set of clauses and evaluates them, returning `ok` or `conflict`. For each clause it calls `evaluate_clause`, which returns `satisfied`, `unsatisfied`, `unit` or `not_resolved`.

`find_unit` calls `evaluate_clause` on each clause and succeeds if it returns `unit`.

`choose_assignment` returns a decision assignment. The decision variable is the first unassigned variables unless lookahead is enabled (Section 13). `choose_value` assigns first 0 and then 1. A conditional construct with an internal cut-fail implements non-chronological backtracing. (See Section 4.7 of the SWI-PROLOG Reference Manual.)

13 Lookahead

The predicates for lookahead appear in `dpll.pro`. `choose_variable_current` takes the set of clauses and the current assignment and returns the chosen variable. First the clauses are evaluated under the current assignment; then the clauses are flattened into a single list of literals, which is transformed to a list of variables. The variable with the largest number of occurrences is returned. The number of occurrences of a variable is maintained as pair `Count-Variable`, which is sorted in descending order of `Count`. The list of pairs is computed by the predicate `count_occurrences`.

For lookahead using the original set of clauses, `set_original_lookahead` is called during initialization to create the database `occurrences`. `choose_variable_original` finds the variable with the largest number of occurrences that has not yet been assigned. `update_variable_occurrences` is called when a clause has been learned to modify `occurrences`.

14 CDCL and NCB

The implication graph is built incrementally. Whenever a unit clause is found, `extend_graph` is called with the unit clause, its number (to label the new edges), the assignment it implies (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. Two predicates for computing a learned clause are now called.

`compute_learned_clause_by_resolution` starts with the conflict clause (the antecedent clause of the `kappa` node) as the current clause. `learn_clause_from_antecedents` uses the list of assignments so far to locate the antecedent of the assignment associated with the current clause. `resolve` resolves the two clauses and `learn_clause_from_antecedents` is called with the resolvent as the new current clause. The algorithm terminates when `check_uip` identifies the current clause as a UIP and this clause becomes the learned clause. The clause learned by resolution is added to the list of learned clauses in the dynamic predicate `learned`.

`compute_learned_clause_by_dominator` locates a dominator of the highest-level decision assignment node relative to the `kappa` node. `get_paths_to_kappa` returns a list of all paths from the decision node to the `kappa` node. This list is an argument to `get_dominator` which finds a node that appears on all the paths. `get_paths_to_kappa` is called again to find paths from decision assignments at *lower* levels to `kappa` and then paths that go through the dominator are removed. From the remaining paths, edges are located that go from nodes of lower level to nodes of the highest level. The complements of the assignments at the source nodes of these edges, together with the complement of the assignment at the dominator, define the learned clause.

`compute_backtrack_level` returns the highest level of an assignment in the learned clause except for the current level.

Open issue After learning a clause, the computation backtracks to choose a new decision assignment. However, the learned clause can cause a variable to be assigned and this should be handled before the new decision assignment. In the `mlm` example presented in the tutorial:

```
Learned clause from resolution: [x021,~x4]
Decision assignment: x1=1@3
Propagate unit: ~x4 (x4=0@3) derived from: [x021,~x4]
```

since `x021` was assigned 0 at level 1, unit propagation of the learned clause `[x021,~x4]` will cause `x4` to be assigned *at the same level*: `x4=0@1`, not `x4=0@3` resulting from the new decision assignment to `x1`. This issue can affect the computation of subsequent learned clauses.

15 Auxiliary predicates

- `is_assigned` checks if a literal has been assigned a value and if so returns that value.
- `get_variables_of_clauses` gets the list of occurrences of variables from a set of clauses (for lookahead) and the list of variables with duplicates removed (for making decision assignments).
- `literals_to_variables` takes a list of literals and returns a sorted set of the variables corresponding to the literals.
- Transformations of variables, literals, assignments and clauses: `to_variable`, `to_complement`, `to_assignment`, `to_literal`, `to_complemented_clause`.
- `set_order` changes the order of the variables for decision assignments. `get_order` used to display the order. `order_variables` returns the ordered list of variables if one has been set.

References

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

[2] A. Biere, M. Heule, H. Van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*. 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.