An Extensible SAT-solver  
[ver. 0.4]

Niklas Eén, Niklas Sörensson

Chalmers University of Technology, Sweden
{een,nik}@cs.chalmers.se

Abstract. In this article, we present a small, complete, and efficient SAT-solver in the style of conflict-driven learning, as exemplified by CHAFF. We aim to give sufficient details about implementation to enable the reader to construct his or her own solver in a very short time. This will allow users of SAT-solvers to make domain specific extensions or adaptations of current state-of-the-art SAT-techniques, to meet the needs of a particular application area. The presented solver is designed with this in mind, and includes among other things a mechanism for adding arbitrary boolean constraints. It also supports solving a series of related SAT-problems efficiently by an incremental SAT-interface.

1 Introduction

The use of SAT-solvers in various applications is on the march. As insight on how to efficiently encode problems into SAT is increasing, a growing number of problem domains are successfully being tackled by SAT-solvers. This is particularly true for the electronic design automation (EDA) industry [BCCFZ99,Lar92]. The success is further magnified by current state-of-the-art solvers being extended and adapted to meet the specific characteristics of these problem domains [ARMS02,ES03].

However, modifying an existing solver, even with a thorough understanding of both the problem domain and of modern SAT-techniques, can become a time consuming and bewildering journey into the mysterious inner workings of a thousand-line software package. Likewise, writing a solver from scratch can also be a daunting task, as there are numerous pitfalls hidden in the intricate details of a correct and efficient solver. The problem is that although the techniques used in a modern SAT-solver are well documented, the details necessary for an implementation have not been adequately presented before.

In the fall of 2002, the authors implemented the solvers SATZOO and SATNIK. In order to sufficiently understand the implementation tricks needed for a modern SAT-solver, it was necessary to consult the source-code of previous implementations.\footnote{LIMMAT at \url{http://www.inf.ethz.ch/personal/biere/projects/limmat/}
ZCHAFF at \url{http://www.ee.princeton.edu/~chaff/zchaff}} We find that the material contained therein can be made more accessible, which is desirable for the SAT-community. Thus, the principal goal of this article is to bridge the gap between existing descriptions of SAT-techniques and their actual implementation.

We will do this by presenting the code of a minimal SAT-solver MINISAT, based on the ideas for conflict-driven backtracking [MS96], together with watched literals and dynamic variable ordering [MZ01]. The original C++ source code
(downloadable from http://www.cs.chalmers.se/~een) for MINISAT is under
600 lines (not counting comments), and is the result of rethinking and simplifying
the designs of SATZOO and SATNIK without sacrificing efficiency. We will present
all the relevant parts of the code in a manner that should be accessible to anyone
acquainted with either C++ or Java.

The presented code includes an incremental SAT-interface, which allows for
a series of related problems to be solved with potentially huge efficiency gains
[ES03]. We also generalize the expressiveness of the SAT-problem formulation
by providing a mechanism for arbitrary constraints over boolean variables to be
defined. Paragraphs discussing implementation alternatives are marked “[Dis-
ussion]” and can be skipped on a first reading.

From the documentation in this paper we hope it is possible for you to
implement a fresh SAT-solver in your favorite language, or to grab the C++
version of MINISAT from the net and start modifying it to include new and
interesting ideas.

2 Application Programming Interface

We start by presenting MINISAT’s external interface, with which a user applica-
tion can specify and solve SAT-problems. A basic knowledge about SAT is
assumed (see for instance [MS96]). The types `var`, `lit`, and `Vec` for variables,
literals, and vectors respectively are explained in detail in section 4.

```plaintext
class Solver - Public interface
  var    newVar ()
  bool   addClause (Vec<lit> literals)
  bool   add ... (...)  # Placeholder for additional constraints
  bool   simplifyDB ()
  bool   solve (Vec<lit> assumptions)
  Vec<bool> model - If found, this vector has the model.
```

The “add...” method should be understood as a place-holder for additional
constraints implemented in an extension of MINISAT.

For a standard SAT-problem, the interface is used in the following way: Vari-
ables are introduced by calling `newVar()`. From these variables, clauses are built
and added by `addClause()`. Trivial conflicts, such as two unit clauses \{x\} and \{\neg x\}
being added, can be detected by `addClause()`, in which case it returns `FALSE`.
From this point on, the solver state is undefined and must not be used further.
If no such trivial conflict is detected during the clause insertion phase, `solve()`
is called with an empty list of assumptions. It returns `FALSE` if the problem is
`unsatisfiable`, and `TRUE` if it is `satisfiable`, in which case the model can be read
from the public vector “model”.

The `simplifyDB()` method can be used before calling `solve()` to simplify the
set of problem constraints (often called the constraint database). In our imple-
mentation, `simplifyDB()` will first propagate all unit information, then remove
all satisfied constraints. As for `addClause()`, the simplifier can sometimes detect a
conflict, in which case \texttt{FALSE} is returned and the solver state is, again, undefined and must not be used further.

If the solver returns \texttt{satisfiable}, new constraints can be added repeatedly to the existing database and \texttt{solve()} run again. However, more interesting sequences of SAT-problems can be solved by the use of \texttt{unit assumptions}. When passing a non-empty list of assumptions to \texttt{solve()}, the solver temporarily assumes the literals to be true. After finding a model or a contradiction, these assumptions are undone, and the solver is returned to a usable state, even when \texttt{solve()} return \texttt{FALSE}, which now should be interpreted as \texttt{unsatisfiable under assumptions}.

For this to work, calling \texttt{initializeDB()} before \texttt{solve()} is no longer optional. It is the mechanism for detecting conflicts independent of the assumptions – referred to as a \textit{top-level} conflict from now on – which puts the solver in an undefined state. We wish to remark that the ability to pass unit assumptions to \texttt{solve()} is more powerful than it might appear at first. For an example of its use, see [ES03].

An alternative interface would be for \texttt{solve()} to return one of three values: \texttt{satisfiable}, \texttt{unsatisfiable}, or \texttt{unsatisfiable under assumptions}. The current interface, however, represents the smallest modification of a non-incremental SAT-solver. The early non-incremental version of SATZOO was made compliant to the above interface by adding just 3 lines of code. [Discussion]

3 Overview of the SAT-solver

This article will treat the popular style of SAT-solvers based on the DPLL algorithm [DLL62], backtracking by conflict analysis and clause recording (also referred to as \textit{learning}) [MS96], and boolean constraint propagation (BCP) using \textit{watched literals} [MZ01]. We will refer to this style of solver as a \textit{conflict-driven SAT-solver}.

The components of such a solver, and indeed a more general constraint solver, can be conceptually divided into three categories:

\begin{itemize}
\item \textbf{Representation}. Somehow the SAT-instance must be represented by internal data structures, as must any derived information.
\item \textbf{Inference}. Brute force search is seldom good enough on its own. A solver also needs some mechanism for computing and propagating the direct implications of the current state of information.
\item \textbf{Search}. Inference is almost always combined with search to make the solver complete. The search can be viewed as another way of deriving information.
\end{itemize}

A standard conflict-driven SAT-solver can represent \textit{clauses} (with two literals or more) and \textit{assignments}. Although the assignments can be viewed as unit-clauses, they are treated specially in many ways, and are best viewed as a separate type of information.

The only inference mechanism used by a standard solver is \textit{unit propagation}. As soon as a clause becomes \textit{unit} under the current assignment (all literals except one are \texttt{false}), the remaining unbound literal is set to \texttt{true}, possibly making

\footnote{2 This includes SAT-solvers such as: \texttt{ZChaff}, \texttt{LIMMAT}, \texttt{BERKMIN}.}
more clauses unit. The process is continued until no more information can be propagated.

The search procedure of a modern solver is the most complex part. Heuristically, variables are picked and assigned values (assumptions are made), until the propagation detects a conflict (all literals of a clause have become false). At that point, a so called conflict clause is constructed and added to the SAT problem. Assumptions are then canceled by backtracking until the conflict clause becomes unit, from which point this unit clause is propagated and the search process continues.

MINISat is extensible with arbitrary boolean constraints. This will affect the representation, which must be able to store these constraints; the inference, which must be able to derive unit information from these constraints; and the search, which must be able to analyze and generate conflict clauses from the constraints. The mechanism we suggest for managing general constraints is very lightweight, and by making the dependencies between the SAT-algorithm and the constraints implementation explicit, we feel it rather adds to the clarity of the solver than obscures it.

Propagation. The propagation procedure of MINISat is largely inspired by that of CHAFF [MZ01]. For each literal, a list of constraints is kept. These are the constraints that may propagate unit information (assignments) if the literal becomes TRUE. For clauses, no unit information can be propagated until all literals except one have become FALSE. Two unbound literals \( p \) and \( q \) of the clause are therefore selected, and references to the clause are added to the lists of \( \overline{p} \) and \( \overline{q} \) respectively. The literals are said to be watched and the lists of constraints are referred to as watcher lists. As soon as a watched literal becomes TRUE, the constraint is invoked to see if information may be propagated, or to select new unbound literals to be watched.

A feature of the watcher system for clauses is that on backtracking, no adjustment to the watcher lists need to be done. Backtracking is therefore very cheap. However, for other constraint types, this is not necessarily a good approach. MINISat therefore supports undo lists which stores constraints that need to be updated when a variable becomes unbound by backtracking.

Learning. The learning procedure of MINISat follows the ideas of Marques-Silva and Sakallah in [MS96]. The process starts when a constraint becomes conflicting (impossible to satisfy) under the current assignment. The conflicting constraint is then asked for a set of variable assignments that make it contradictory. For a clause, this would be all the literals of the clause (which are FALSE under a conflict). Each of the variable assignments returned must be either an assumption of the search procedure, or the result of some propagation of a constraint. The propagating constraints are in turn asked for the set of variable assignments that forced the propagation to occur, continuing the analysis backwards. The procedure is repeated until some termination condition is fulfilled, resulting in a set of variable assignments that implies the conflict. A clause prohibiting that particular assignment is added to the clause database. This learnt clause must always, by construction, be implied by the original problem constraints.
Learned clauses serve two purposes: they drive the backtracking (as we shall see) and they speed up future conflicts by “caching” the reason for the conflict. Each clause will prevent only a constant number of inferences, but as the recorded clauses start to build on each other and participate in the unit propagation, the accumulated effect of learning can be massive. However, as the set of learnt clauses increase, propagation is slowed down. Therefore, the number of learnt clauses is periodically reduced, keeping only the clauses that seem useful by some heuristic.

**Search.** The search procedure of a conflict-driven SAT-solver is somewhat implicit. Although a recursive definition of the procedure might be more elegant, it is typically described (and implemented) iteratively. The procedure will start by selecting an unassigned variable $x$ (called the decision variable) and assume a value for it, say TRUE. The consequences of $x = \text{TRUE}$ will then be propagated, possibly resulting in more variable assignments. All variables assigned as a consequence of $x$ is said to be from the same decision level, counting from 1 for the first assumption made and so forth. Assignments made before the first assumption (decision level 0) are called top-level.

All assignments will be stored on a stack in the order they were made; from now on referred to as the trail. The trail is divided into decision levels and is used to undo information during backtracking.

The decision phase will continue until either all variables have been assigned, in which case we have a model, or a conflict has occurred. On conflicts, the learning procedure will be invoked and a conflict clause produced. The trail will be used to undo decisions, one level at a time, until precisely one of the literals of the learnt clause becomes unbound (they are all FALSE at the point of conflict). By construction, the conflict clause cannot go directly from conflicting to a clause with two or more unbound literals. If the clause remains unit for several decision levels, it is advantageous to choose the lowest level (referred to as backjumping or non-chronological backtracking [MS96]).

```
loop
    propagate() - propagate unit clauses
    if not conflict then
        if all variables assigned then
            return SATISFIABLE
        else
            decide() - pick a new variable and assign it
    else
        analyze() - analyze conflict and add a conflict clause
        if top-level conflict found then
            return UNSATISFIABLE
        else
            backtrack() - undo assignments until conflict clause is unit
```

An important part of the procedure is the heuristic for `decide()`. Like CHAFF, MINISAT uses a dynamic variable order that gives priority to variables involved in recent conflicts.
Although this is a good default order, domain specific heuristics have successfully been used in various areas to improve the performance [Stri00]. Variable ordering is a traditional target for improving SAT-solvers.

**Activity heuristics.** One important technique introduced by CHAFF [MZ01] is a dynamic variable ordering based on activity (referred to as the VSIDS heuristic). The original heuristic imposes an order on literals, but borrowing from SATZOO, we make no distinction between \( p \) and \( \overline{p} \) in MINISAT.

Each variable has an activity attached to it. Every time a variable occurs in a recorded conflict clause, its activity is increased. We will refer to this as bumping. After recording the conflict, the activity of all the variables in the system are multiplied by a constant less than 1, thus dumping the activity of variables over time. Recent increments count more than old. The current sum determines the activity of a variable.

In MINISAT we use a similar idea for clauses. When a learnt clause is used in the analysis process of a conflict, its activity is bumped. Inactive clauses are periodically removed.

**Constraint removal.** The constraint database is divided into two parts: the problem constraints and the learnt clauses. As we have noted, the set of learnt clauses can be periodically reduced to increase the performance of propagation. Learnt clauses are used to crop future branches in the search tree, so we risk getting a bigger search space instead. The balance between the two forces is delicate, and there are SAT-instances for which a big learnt clause set is advantageous, and others where a small set is better. MINISAT’s default heuristic starts with a small set and gradually increases the size.

Problem constraints can also be removed if they are satisfied at the top-level. The API method `simplifyDB()` is responsible for this. The procedure is particularly important for incremental SAT-problems, where techniques for clause removal build on this feature.

**Top-level solver.** Although the pseudo-code for the search procedure presented above suffices for a simple conflict-driven SAT-solver, a solver strategy can improve the performance. A typical strategy applied by modern conflict-driven SAT-solvers is the use of restarts to escape from futile parts of the search tree. In MINISAT we also vary the number of learnt clauses kept at a given time. Furthermore, the `solve()` method of the API supports incremental assumptions, not handled by the above pseudo-code.

4 Implementation

The following conventions are used in the code. Atomic types start with a lower-case letter and are passed by value. Composite types start with a capital letter and are passed by reference. Blocks are marked only by indentation level. The bottom symbol \( \bot \) will always mean undefined; the symbol FALSE will be used to denote the boolean false.

We will use, but not specify an implementation of, the following abstract data types: `Vec(T)` an extensible vector of type `T`; `lit` the type of literals containing
### class Vec(T) - Public interface
- **Constructors:**
  - Vec()
  - Vec(int size)
  - Vec(int size, T pad)
- **Size operations:**
  - int size ()
  - void shrink (int nof elems)
  - void pop ()
  - void growTo (int size)
  - void growTo (int size, T pad)
  - void clear ()
- **Stack interface:**
  - void push ()
  - void push (T elem)
  - T last ()
- **Vector interface:**
  - T op [ ] (int index)
- **Dynamization:**
  - void copyTo (Vec(T) copy)
  - void moveTo (Vec(T) dest)

### class lit - Public interface
- **Global functions:**
  - bool op (lit p)
  - int var (lit p)
  - int index (lit p)

### class lbool - Public interface
- **Global functions:**
  - bool op (lbool x)
- **Global constants:**
  - lbool FALSE, TRUE

### class Queue(T) - Public interface
- **Queue ()
  - void insert (T x)
  - T dequeue ()
  - void clear ()
  - int size ()

---

**Fig. 1. Basic abstract data types used throughout the code.** The vector data type can push a default constructed element by the `push()` method with no argument. The `moveTo()` method will move the contents of a vector to another vector in constant time, clearing the source vector. The literal data type has an `index()` method which converts the literal to a "small" integer suitable for array indexing. The `var()` method returns the underlying variable of the literal, and the `sign()` method if the literal is signed (FALSE for x and TRUE for \( \overline{x} \)).

A special literal \( \bot_{\text{lit}} \), `lbool` for the lifted boolean domain containing elements \( \text{TRUE}, \text{FALSE}, \text{\bot} \); `Queue(T)` a queue of type `T`. We also use `var` as a type synonym for `int` (for implicit documentation) with the special constant `\bot_{\text{var}}`. The interfaces of the abstract data types are presented in **Figure 1**.

#### 4.1 The solver state

A number of things need to be stored in the solver state. **Figure 2** shows the complete set of member variables of the solver type of MINISAT. Together with the state variables we define some short helper methods in **Figure 3**, as well as the interface of `VarOrderer` (**Figure 4**), explained in section 4.6.

The state does not contain a boolean "conflict" to remember if a top-level conflict has been reached. Instead we impose an invariant that the solver must never be in a conflicting state. As a consequence, any method that puts the solver in a conflicting state must communicate this. Using the solver object after this point is illegal. The invariant makes the interface slightly more cumbersome to use, but simplifies the implementation, which is important when extending and experimenting with new techniques.
class Solver
- Configures database
  Vec (Constr) constrs - List of problem constraints.
  Vec (Clause) learnts - List of learnt clauses.
double clajnc - Clause activity increment - amount to bump with.
double clajdecay - Decay factor for clause activity.
- Variable order
Vec (double) activity - Heuristic measurement of the activity of a variable.
double varinc - Variable activity increment - amount to bump with.
double vardecay - Decay factor for variable activity.
VarOrder order - Keeps track of the dynamic variable order.
- Propagation
Vec (Var{Constr}) watches - For each literal 'p', a list of constraints watching 'p'.
Vec (Var{Constr}) undos - A constraint will be inspected when 'p' becomes true.
Queue (lit) propQ - Propagation queue.
- Assignments
Vec (bool) assigns - The current assignments indexed on variables.
Vec (lit) trail - List of assignments in chronological order.
Vec (int) traillim - Separates indices for different decision levels in 'trail'.
Vec (Constr) reason - For each variable, the constraint that implied its value.
Vec (int) level - For each variable, the decision level it was assigned.
int rootlevel - Separates incremental and search assumptions.

Fig. 2. Internal state of the solver.

int Solver.nVars() return assigns.size()
int Solver.nAssigns() return trail.size()
int Solver.nConstraints() return constrs.size()
int Solver.nLearnts() return learnts.size()
bool Solver.value (var x) return assigns[x]
bool Solver.value (lit p) return sign(p) ? ~assigns[var(p)] : assigns[var(p)]
int Solver.decisionLevel() return traillim.size()

Fig. 3. Small helper methods. For instance, nLearnts() returns the number of learnt clauses.

class VarOrder - Public interface
VarOrder (Vec (bool) ref_to_assigns, Vec (double) ref_to_activity)
  void newVar() - Called when a new variable is created.
  void update (var x) - Called when variable has increased in activity.
  void updateAll() - Called when all variables have been assigned new activity.
  void undo (var x) - Called when variable is unbound (and may be selected again).
  var select() - Called to select a new, unassigned variable.

Fig. 4. Assisting ADT for the dynamic variable ordering of the solver. The constructor takes references to the assignment vector and the activity vector of the solver. The method select() will return the unassigned variable with the highest activity.
<table>
<thead>
<tr>
<th>Class Constr</th>
<th>Function</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>virtual void remove</td>
<td>(Solver $S$)</td>
<td>must be defined</td>
</tr>
<tr>
<td></td>
<td>virtual bool propagate</td>
<td>(Solver $S$, lit $p$)</td>
<td>must be defined</td>
</tr>
<tr>
<td></td>
<td>virtual bool simplify</td>
<td>(Solver $S$)</td>
<td>defaults to return false</td>
</tr>
<tr>
<td></td>
<td>virtual void undo</td>
<td>(Solver $S$, lit $p$)</td>
<td>defaults to do nothing</td>
</tr>
<tr>
<td></td>
<td>virtual void calcReason</td>
<td>(Solver $S$, lit $p$, Vec(lit) out_reason)</td>
<td>must be defined</td>
</tr>
</tbody>
</table>

Fig. 5. Abstract base class for constraints.

### 4.2 Constraints

MINISat can handle arbitrary constraints over boolean variables through the abstraction presented in Figure 5. Each constraint type needs to implement methods for constructing, removing, propagating and calculating reasons. In addition, methods for simplifying the constraint and updating the constraint on backtrack can be specified. We explain the meaning and responsibilities of these methods in detail:

**Constructor.** The constructor may only be called at the top-level. It must create and add the constraint to appropriate watcher lists after enqueuing any unit information derivable under the current top-level assignment. Should a conflict arise, this must be communicated to the caller.

**Remove.** The remove method supplants the destructor by receiving the solver state as a parameter. It should dispose the constraint and remove it from the watcher lists.

**Propagate.** The propagate method is called if the constraint is found in a watcher list during propagation of unit information $p$. The constraint is removed from the list and is required to insert itself into a new or the same watcher list. Any unit information derivable as a consequence of $p$ should be enqueued. If successful, TRUE is returned; if a conflict is detected, FALSE is returned. The constraint may add itself to the undo list of var($p$) if it needs to be updated when $p$ becomes unbound.

**Simplify.** At the top-level, a constraint may be given the opportunity to simplify its representation (returns TRUE) or state that the constraint is satisfied under the current assignment (returns FALSE). A constraint must not be simplifiable to produce unit information or to be conflicting; in that case the propagation has not been correctly defined.

**Undo.** During backtracking, this method is called if the constraint added itself to the undo list of var($p$) in propagate(). The current variable assignments are guaranteed to be identical to that of the moment before propagate() was called.

**Calculate Reason.** This method is given a literal $p$ and an empty vector. The constraint is the reason for $p$ being true, that is, during propagation, the current constraint enqueued $p$. The received vector is extended to include a set of assignments (represented as literals) implying $p$. The current variable assignments are guaranteed to be identical to that of the moment before
the constraint propagated $p$. The literal $p$ is also allowed to be the special
constant $\bot_{\text{lit}}$ in which case the reason for the clause being conflicting should
be returned through the vector.

The code for the Clause constraint is presented in Figure 7. It is also used for learnt clauses,
which are unique in that they can be added to the clause database while the solver is not at
top-level. This makes the constructor code a bit more complicated than it would be for a normal
constraint.

Implementing the addClause() method of the solver API is just a matter of calling Clause-
new() and pushing the new constraint on the “constrs” vector, storing the list of problem con-
straints. For completeness, we also display the code for creating variables in the solver (Fig-
ure 6).

[Discussion] There are a number of tricks for smart-coding that can be used in a C++ implementation of
Clause. In particularly the “lits” vector can be implemented as an zero-sized array placed last in the class, and then extra memory allocated for the clause to contain the data. We observed a 20% speedup for this trick. Furthermore, memory can be saved by not storing activity for problem clauses.

[Discussion] Of the methods defining a constraint, propagate() should be the primary target for efficient implementation. The SAT-solver spends about 80% of the time propagating, so the method will be called frequently. In SATZOO a performance gain was achieved by remembering the position of the last watched literal and start looking for a new literal to watch from that position. Further speedups may be achieved by specializing the code for small clause sizes.

4.3 Propagation

Given the mechanism for adding constraints, we now move on to describe the
propagation of unit information on these constraints.

The propagation routine keeps a set of literals (unit information) that is to
be propagated. We call this the propagation queue. When a literal is inserted into
the queue, the corresponding variable is immediately assigned. For each literal
in the queue, the watcher list of that literal determines the constraints that may
be affected by the assignment. Through the interface described in the previous
section, each constraint is asked by a call to its propagate() method if more unit
information can be inferred, which will then be enqueued. The process continues
until either the queue is empty or a conflict is found.

An implementation of this procedure is displayed in Figure 9. It starts by
dequeuing a literal and clearing the watcher list for that literal by moving it to
“tmp”. The propagate method is then called for each constraint of “tmp”. This
will re-insert watches into new lists. Should a conflict be detected during the
traversal of “tmp”, the remaining watches will be copied back to the original
watcher list.

```plaintext
var Solver.newVar()
    int index
    index = nVars()
    watches .push()
    watches .push()
    undos .push()
    reason .push(NULL)
    assigns .push(\bot)
    level .push(-1)
    activity .push(0)
    order .newVar()
return index
```

Fig. 6. Creates a new SAT variable in the solver.
class Clause : public Constr

  bool learnt
  float activity
  Vec(lit) lits

- Constructor – creates a new clause and adds it to watcher lists:
static bool Clause_new(Solver S, Vec(lit) ps, bool learnt, Clause out_clause)

  "Implementation in Figure 8"

- Learnt clauses only:
bool locked(Solver S)
  return S.reason[\text{var}(\text{lits}[0])] == this

- Constraint interface:
void remove(Solver S)
  removeElem(this, S.watches[index(\text{~lits}[0])])
  removeElem(this, S.watches[index(\text{~\text{lits}[1]})])
  delete this

bool simplify(Solver S) // only called at top-level with empty prop. queue
int j = 0
for (int i = 0; i < lits.size(); i++)
  if (S.value(lits[i]) == \text{TRUE}_\perp)
    return True
  else if (S.value(lits[i]) == \text{⊥})
    lits[i+1] = lits[i] - false literals are not copied (only occur for \text{i} \geq 2)
    lits.shrink(lits.size() - j)
  return False

bool propagate(Solver S, lit p)
- Make sure the false literal is \text{lits}[1]:
if (\text{lits}[0] == \text{~p})
  \text{lits}[0] = \text{lits}[1], \text{lits}[1] = \text{~p}

- If 0th watch is true, then clause is already satisfied.
if (S.value(\text{lits}[0]) == \text{TRUE}_\perp)
  S.watches[index(p)].push(this)  // re-insert clause into watcher list
  return True

- Look for a new literal to watch:
for (int i = 2; i < size(); i++)
  if (S.value(lits[i]) != \text{FALSE}_\perp)
    lits[1] = lits[i], lits[i] = \text{~p}
    S.watches[index(\text{~lits}[1])].push(this)  // insert clause into watcher list
    return True

- Clause is unit under assignment:
  S.watches[index(p)].push(this)
  return S.enqueue(lits[0], this)  // enqueue for propagation

void calcReason(Solver S, lit p, vec(lit) out_reason)
- invariant: (p == \text{⊥}) or (p == \text{~lits}[0])
for (int i = ((p == \text{⊥}) ? 0 : 1); i < size(); i++)
  out_reason.push(\text{~lits}[i])  // invariant: S.value(lits[i]) == \text{FALSE}_\perp
if (learnt) S换句话Activity(this)

Fig. 7. Implementation of the Clause constraint.
```c
bool Clause_new(Solver S, Vec<lit> ps, bool learnt, Clause out_clause)
    out_clause = NULL
    - Normalize clause:
    if (! learnt)
        if ("any literal in ps is true") return TRUE
        if ("both p and ~p occurs in ps") return TRUE
        "remove all false literals from ps"
        "remove all duplicates from ps"
    if (ps.size() == 0)
        return FALSE
    else if (ps.size() == 1)
        return S.enqueue(ps[0])  // unit facts are enqueued
    else
        - Allocate clause:
        Clause c = new Clause
        ps.moveTo(c.lits)
        c.learnt = learnt
        c.activity = 0  // only relevant for learnt clauses
        if (learnt)
            - Pick a second literal to watch:
            "Let max_j be the index of the literal with highest decision level"
            c.lits[1] = ps[max_j], c.lits[max_j] = ps[1]
            - Bumping:
            S.clkBumpActivity(c)  // newly learnt clauses should be considered active
            for (int i = 0; i < ps.size(); i++)
                S.varBumpActivity(ps[i])  // variables in conflict clauses are bumped
            - Add clause to watcher lists:
            S.watches[index (~c.lits[0])], push(c)
            S.watches[index (~c.lits[1])], push(c)
        out_clause = c
    return TRUE
```

**Fig. 8.** Constructor function for clauses. Returns FALSE if top-level conflict is detected. 'out_clause' may be set to NULL if the new clause is already satisfied under the current top-level assignment. **Post-condition:** 'ps' is cleared. For learnt clauses, all literals will be false except 'lits[0]' (this by design of the analyze() method). For the propagation to work, the second watch must be put on the literal which will first be unbound by backtracking. (Note that none of the learnt-clause specific things needs to be done for a user defined constraint type.)
```plaintext
Constr Solver::propagate()

while (propQ.size() > 0)
    lit p = propQ.dequeue()  // 'p' is now the enqueued fact to propagate
    Vec<Constr> tmp
    watches[index(p)].moveTo(tmp)
    for (int i = 0; i < tmp.size(); i++)
        if (!tmp[i].propagate(this, p))
            // Constraint is conflicting; copy remaining watches to 'watches[p]'
            // and return constraint:
        for (int j = i + 1; j < tmp.size(); j++)
            watches[index(p)].push(tmp[j])
    return NULL

bool Solver::enqueue(lit p, Constr from = NULL)

if (!value(p))
    if (value(p) == FALSE)
        // Conflicting enqueued assignment
        propQ.clear()
        return FALSE
    else
        // Existing consistent assignment - don’t enqueue
        // “Possibly change reason for p here (sound if p is still in prop. queue)”
        return TRUE
else
    // New fact, store it
    assigns[var(p)] = bool(!sign(p))
    level[var(p)] = decisionLevel()
    reason[var(p)] = from
    trail.push(p)
    propQ.insert(p)
return TRUE
```

![Fig. 9. propagate()](image)

Fig. 9. *propagate()*: Propagates all enqueued facts. If a conflict arises, the conflicting
clause is returned, otherwise NULL. *enqueue()*: Puts a new fact on the propagation
queue, as well as immediately updating the variable’s value in the assignment vector. If
a conflict arises, FALSE is returned and the propagation queue is cleared. The parameter
‘from’ contains a reference to the constraint from which ‘p’ was propagated (defaults
to NULL if omitted).

The method for enqueuing unit information is relatively straightforward. Note that the same
fact can be enqueued several times, as it may be propagated from different constraints, but it will only be put on the propagation
queue once.

It may be that later enqueuings have a “better” reason (determined heuristically) and a small performance gain was achieved in SATZOO by changing reason
if the new reason was smaller than the previously stored. The changing affects
the conflict clause generation described in the next section.

[Discussion]
4.4 Learning

This section describes the conflict-driven clause learning. It was first described in [MS96] and is one of the major advances of SAT-technology in the last decade.

We describe the basic conflict-analysis algorithm by an example. Assume the database contains the clause \{x, y, z\} which just became unsatisfied during propagation. This is our conflict. We call \( \pi \wedge \rho \wedge \sigma \) the reason set of the conflict. Now \( x \) is false because \( \pi \) was propagated from some constraint. We ask that constraint to give us the reason for propagating \( \pi \) (the \texttt{calcReason()} method). It will respond with another conjunction of literals, say \( u \wedge v \). These were the variable assignment that implied \( \pi \). The constraint may in fact have been the clause \( \{u, \pi, \sigma\} \). From this little analysis we know that \( u \wedge v \wedge \rho \wedge \sigma \) must also lead to a conflict. We may prohibit this conflict by adding the clause \( \{\pi, \rho, y, z\} \) to the clause database. This would be an example of a learnt conflict clause.

In the example, we picked only one literal and analyzed it one step. The process of expanding literals with their reason sets can be continued, in the extreme case until all the literals of the conflict set are decision variables (which were not propagated by any constraints). Different learning schemes based on this process have been proposed. Experimentally the “First Unique Implication Point” (First UIP) heuristic has been shown effective [ZM01]. We will not give the definition of UIPs here, but just state the algorithm: In a breadth-first manner, continue to expand literals of the current decision level, until there is just one left.

In the code for \texttt{analyze()}, displayed in \textit{Figure 10}, we make use of the fact that a breadth-first traversal can be achieved by inspecting the trail backwards. Especially, the variables of the reason set of \( p \) is always before \( p \) in the trail. Furthermore, in the algorithm we initialize \( p \) to \( \perp \), which will make \texttt{calcReason()} return the reason for the conflict.

[Discussion] Assuming \( x \) to be the unit information that causes the conflict, an alternative implementation would be to calculate the reason for \( \pi \) and just add \( x \) to that set. The code would be slightly more cumbersome but the contract for \texttt{calcReason()} would be simpler, as we no longer need the special case for \( \perp \).

Finally, the analysis not only returns a conflict clause, but also the backtracking level. This is the lowest decision level for which the conflict clause is unit. It is advantageous to backtrack as far as possible [MS96], and is referred to as back-jumping or non-chronological backtracking in the literature.

4.5 Search

The search method in \textit{Figure 13} works basically as described in section 3 but with the following additions:

**Replots.** The first argument of the search method is “no conflicts”. The search for a model or a contradiction will only be conducted for this many conflicts. If failing to solve the SAT-problem within the bound, all assumptions will be canceled and \( \perp \) returned. The surrounding solver strategy will then restart the search, possibly with a new set of parameters.
void Solver.analyze(Constr confl, Vec<lit> out_learnt, Int out_Jlevel)

Vec<bool> seen[nVars(), FALSE]
int counter = 0
lit p = ⊥ia
Vec<lit> p_reason

out_learnt.push() — leave room for the asserting literal
out_Jlevel = 0
do
    p_reason.clear()
    confl.calcReason(this, p, p_reason) — invariant here: confl != NULL

    — Trace reason for p:
    for (int j = 0; j < p_reason.size(); j++)
        lit q = p_reason[j]
        if (seen[var(q)])
            seen[var(q)] = TRUE
        if (level[var(q)] == decisionLevel())
            counter++
        else if (level[var(q)] > 0) — exclude variables from decision level 0
            out_learnt.push(!q)
            out_Jlevel = max(out_Jlevel, level[var(q)])

    — Select next literal to look at:
    do
        p = trail.last()
        confl = reason[var(p)]
        undoOne()
        while (seen[var(p)])
            counter--
        while (counter > 0)
            out_learnt[0] = ¬p

Fig. 10. Analyze a conflict and produce a reason clause. Pre-conditions: (1) 'out_learnt' is assumed to be cleared. (2) Current decision level must be greater than root level. Post-conditions: (1) 'out_learnt[0]' is the asserting literal at level 'out_Jlevel'. Effect: Will undo part of the trail, but not beyond last decision level.

void Solver.record(Vec<lit> clause)

Clause c — will be set to created clause, or NULL if 'clause[]' is unit
Clause_new(this, clause, TRUE, c) — cannot fail at this point
enqueue(clause[0], c) — cannot fail at this point

if (c != NULL) learnts.push(c)

Fig. 11. Record a clause and drive backtracking. Pre-condition: 'clause[0]' must contain the asserting literal. In particular, 'clause[]' must not be empty.
Reduce. The second argument, “noflearnts”, sets an upper limit on the number of learnt clauses that are kept. Once this number is reached, reduceDB() is called. Clauses that are currently the reason for a variable assignment are said to be locked and cannot be removed by reduceDB(). For this reason, the limit is extended by the number of assigned variables, which approximates the number of locked clauses.

Parameters. The third argument to the search method groups some tuning constants. In the current version of MiniSAT, it only contains the decay factors for variables and clauses.

Root-level. To support incremental SAT, the concept of a root-level is introduced. The root-level acts as a new top-level. Above the root-level are the incremental assumptions passed to solve (if any). The search procedure is not allowed to backtrack above the root-level, as this would change the incremental assumptions. If we reach a conflict at root-level, the search will return FALSE.

A problem with the approach presented here is conflict clauses that are unit. For these, analyse() will always return a backtrack level of 0 (top-level). As unit clauses are treated specially, they are never added to the clause database. Instead they are enqueued as facts to be propagated (see the code of Clause_new()). There would be no problem if this was done at top-level. However, the search procedure will only undo until root-level, which means that the unit fact will be enqueued there. Once search() has solved the current SAT-problem, the surrounding solver strategy will undo any incremental assumption and put the solver back at the top-level. By this the unit clause will be forgotten, and the next incremental SAT problem will have to infer it again.

A solution to this is to store the learnt unit clauses in a vector and re-insert them at top-level before the next call to solve(). The reason for omitting this in MiniSAT is that we have not seen any performance gain by this extra handling in our applications [ES03,CS03]. Simplicity thus dictates that we leave it out of the presentation.

Simplify. Provided the root-level is 0 (no assumptions were passed to solve()) the search will return to the top-level every time a unit clause is learnt. At that point it is legal to call simplifyDB() to simplify the problem constraints according to the top-level assignment. If a stronger simplifier than presented here is implemented, a contradiction may be found, in which case the search should be aborted. As our simplifier is not stronger than normal propagation, it can never reach a contradiction, so we ignore the return value of simplify().

4.6 Activity heuristics

The implementation of activity is shown in Figure 14. Instead of actually multiplying all variables by a decay factor after each conflict, we bump variables with larger and larger numbers. Only relative values matter. Eventually we will reach
\begin{figure}[h]
\centering
\begin{tabular}{|l|l|}
\hline
\textbf{void Solver.undoOne()} & \textbf{bool Solver.assume(lit p)} \\
\hline
\textbf{lit} & \textbf{trail.last()} \\
\textbf{var} & \textbf{p = var(p)} \\
\textbf{assigns} & \textbf{[x] = ⊥} \\
\textbf{reason} & \textbf{[x] = NULL} \\
\textbf{level} & \textbf{[x] = -1} \\
\textbf{order.undo(x)} & \textbf{trail.pop()} \\
\hline
\textbf{while} (undos[x].size() > 0) & \textbf{int c = trail.size() - trailLim.last()} \\
& \textbf{undos[x].last().undo(this, p)} \\
& \textbf{undos[x].pop()} \\
& \textbf{for (; c != 0; c--)} \\
& \textbf{undoOne()} \\
& \textbf{trailLim.pop()} \\
\end{tabular}
\end{figure}

\textbf{Fig. 12. assume()}: returns FALSE if immediate conflict. \textbf{Pre-condition}: propagation queue is empty. \textbf{undoOne()}: unbinds the last variable on the trail. \textbf{cancel()}: reverts to the state before last push(). \textbf{Pre-condition}: propagation queue is empty. \textbf{cancelUntil()}: cancels several levels of assumptions.

the limit of what is representable by a floating point number. At that point, all activities are scaled down.

In the \textit{VarOrder} data type of MINISAT, the list of variables is kept sorted on activity at all time. The backtracking will always accurately choose the most active variable. The original suggestion for the VSIDS dynamic variable ordering was to sort periodically.

The polarity of a literal is ignored in MINISAT. However, storing the latest polarity of a variable might improve the search when restarts are used, but it remains to be empirically supported. Furthermore, the interface of \textit{VarOrder} can be used for other variable heuristics. In SATZOO, an initial static variable order computed from the clause structure was particularly successful on many problems.

\subsection{Constraint removal}

The methods for reducing the set of learnt clauses as well as the top-level simplification procedure can be found in \textit{Figure 15}.

When removing learnt clauses, it is important not to remove so-called \textit{locked} clauses. Locked clauses are those participating in the current backtracking branch by being the reason (through propagation) for a variable assignment. The reduce procedure keeps half of the learnt clauses, except for those which have decayed below a threshold limit. Such clauses can occur if the set of active constraints is very small.

Top-level simplification can be seen as a special case of propagation. Since \textit{Discussion} it is performed under no assumption, anything learnt can be kept forever. The freedom of not having to store derived information separately, with the ability to undo it later, makes it easier to implement stronger propagation.
```c
bool Solver.search(int nconflicts, int nlearnts, SearchParams params)
{
    int conflictC = 0
    var_decay = 1 / params.var_decay
    cla_decay = 1 / params.cla_decay
    model.clear()

    loop
        Constr confl = propagate()
        if (confl != NULL)
            - CONFLICT
                conflictC++
                Vec(lit) learnt_clause
                int backtrack_level
                if (decisionLevel() == root_level)
                    return FALSE
                analyze(confl, learnt_clause, backtrack_level)
                cancelUntil(max(backtrack_level, root_level))
                record(learnt_clause)
                decayActivities()
        else
            - NO CONFLICT
                if (decisionLevel() == 0)
                    - Simplify the set of problem clauses:
                        simplifyDB()
                        - our simplifier cannot return false here

                if (learnts.size() - nAssigns() >= nlearnts)
                    - Reduce the set of learnt clauses:
                        reduceDB()

                if (nAssigns() == nVars())
                    - Model found:
                        model.growTo(nVars())
                        for (int i = 0; i < nVars(); i++)
                            model[i] = (value(i) == True)
                        cancelUntil(root_level)
                        return TRUE
                else if (conflictC >= nconflicts)
                    - Reached bound on number of conflicts:
                        cancelUntil(root_level)
                        - force a restart
                        return ⊥
                else
                    - New variable decision:
                        lit p = lit(order.select())
                        - may have heuristic for polarity here
                        assume(p)
                        - cannot return false
```

Fig. 13. Search method. Assumes and propagates until a conflict is found, from which a conflict clause is learnt and backtracking performed until search can continue. **Precondition:** root_level == decisionLevel().
4.8 Top-level solver

The method implementing MINISAT’s top-level strategy can be found in Figure 16. It is responsible for making the incremental assumptions and setting the root level. Furthermore, it completes the simple backtracking search with restarts, which are performed less and less frequently. After each restart, the number of allowed learnt clauses is increased.

The code contains a number of hand-tuned constants that have shown to perform reasonable on our applications [ES03,CS03]. The top-level strategy, however, is a productive target for improvements (possibly application dependent). In SATZOO, the top-level strategy contains an initial phase where a static variable ordering is used.

5 Conclusions and Related Work

By this paper, we have provided a minimal reference implementation of a modern conflict-driven SAT-solver. Despite the abstraction layer for boolean constraints, and the lack of more sophisticated heuristics, the performance of MINISAT is comparable to state-of-the-art SAT-solvers. We have tested MINISAT against ZCHAFF and BERKMIN 5.61 on 177 SAT-instances.\(^3\) At a time-out of 10 minutes, MINISAT solved 160 instances, while ZCHAFF solved 147 instances and BERKMIN 157 instances.

Another approach to incremental SAT and non-clausal constraints was presented by Aloul, Ramani, Marilkov, and Sakallah in their work on SATIRE and PBS [WKS01,ARMS02]. Our implementation differs in that it has a simpler notion of incrementality, and that it contains a well documented interface for non-clausal constraints.

Finally, a set of reference implementations of modern SAT-techniques is present in the OPENSAT project.\(^4\) However, the project aim for completeness rather than minimal exposition, as we have chosen in this paper.

\(^3\) These instances were used to tune SATZOO for the SAT 2003 Competition.

\(^4\) http://www.opensat.org
void Solver.reduceDB()
    int i, j
    double lim = clause / learnts.size()
    sortOnActivity(learnts)
    for (i=j=0; i < learnts.size()/2; i++)
        if (!learnts[i].locked(this))
            learnts[i].remove(this)
        else
            learnts[i++ ] = learnts[i]
    for (; i < learnts.size(); i++)
        if (!learnts[i].locked(this)
            && learnts[i].activity() < lim)
            learnts[i].remove(this)
        else
            learnts[i++ ] = learnts[i]
    learnts.shrink(i - j)

def Solver.simplifyDB()
    if (propagate() != NULL)
        return FALSE
    for (int i = 0; i < learnts.size(); i++)
        if (cs[i].simplify(this))
            cs[i].remove(this)
        else
            cs[i++ ] = cs[i]
    if (cs.size()-j)
        return TRUE

Fig. 15. reduceDB(): Remove half of the learnt clauses minus some locked clauses. A locked clause is a clause that is reason to a current assignment. Clauses below a certain lower bound activity are also be removed. simplifyDB(): Top-level simplify of constraint database. Will remove any satisfied constraint and simplify remaining constraints under current (partial) assignment. If a top-level conflict is found, False is returned. Pre-condition: Decision level must be zero. Post-condition: Propagation queue is empty.

bool Solver.solve( Vec(lit) assumes)
    SearchParams params(0.95, 0.99)
    double nconflicts = 100
    double learnts = nConstraints()/3
    bool status = ⊥
    - Push incremental assumptions:
        for (int i = 0; i < assumes.size(); i++)
            if (assumes[i] != NULL)
                cancelUntil(0)
        return FALSE
    rootLevel = decisionLevel()
    - Solve:
        while (status == ⊥)
            status = search((int)nconflicts, (int)learnts, params)
            nconflicts * = 1.5
            learnts * = 1.1
            cancelUntil(0)
        return status == TRUE

Fig. 16. Main solve method. Pre-condition: If assumptions are used, simplifyDB() must be called right before using this method. If not, a top-level conflict (resulting in a non-usuable internal state) cannot be distinguished from a conflict under assumptions.
6 Exercises

1. Write the code for an AtMost constraint. The constraint is satisfied if at most \( n \) out of \( m \) specified literals are true.

2. Implement a generator for (generalized) pigeon-hole formulas using the new constraints. The generator should take three arguments: number of pigeons, number of holes, and hole capacity. Each pigeon must reside in some pigeon-hole. No hole may contain more pigeons than its capacity.

3. Make an incremental version that adds one pigeon to the problem at a time until the problem becomes unsatisfiable.

References


