Practical Algorithms for Unsatisfiability Proof and Core Generation in SAT solvers

Roberto Asín [a] Robert Nieuwenhuis [b]
Albert Oliveras [c]
Enric Rodríguez-Carbonell [d]

[a] *www.lsi.upc.edu/˜rasin*
[b] *www.lsi.upc.edu/˜roberto*
[c] *www.lsi.upc.edu/˜oliveras*
[d] *www.lsi.upc.edu/˜erodri*

Abstract:[*] Since Zhang and Malik's work in 2003 [ZM03], it is well-known that modern DPLL-based SAT solvers with learning can be instrumented to write a trace on disk from which, if the input is unsatisfiable, a resolution proof can be extracted (and checked), and hence also an *unsatisfiable core*: a (frequently small) unsatisfiable subset of the input clauses.

In this article we first give a new algorithmic approach for processing these (frequently huge) traces. It achieves the efficiency of a depth-first traversal, while preserving the property that memory usage remains upper bounded by that of the SAT solver that generated the trace.

The second part of this article is about in-memory algorithms for generating SAT proofs and cores, without writing traces to disk. We discuss advantages and disadvantages of this approach and investigate why the current SAT solvers with this feature still run out of memory on long SAT runs. We analyze several of these in-memory algorithms, based on carefully designed experiments with our implementation of each one of them, as well as with (our implementation of) a trace-based one. Then we describe a new in-memory algorithm which saves space by doing more bookkeeping to discard unnecessary information, and show that it can handle significantly more instances than the previously existing algorithms, at a negligible expense in time.

[*]Technical Univ. of Catalonia, Barcelona. Partially supported by Spanish Min. of Educ. and Science through the LogicTools-2 project, ref. TIN2007-68093-C02-01.

AI Communications
ISSN 0921-7126, IOS Press.

1. Introduction

More and more applications of DPLL-based ([DP60,DLL62]) (propositional) SAT solvers and their extensions keep emerging. For some of these applications, it suffices to obtain a yes/no answer, possibly with a model in case of satisfiability. For other applications, also in case of *un*satisfiability a more detailed answer is needed. For example, one may want to obtain a small (or even minimal, wrt. set inclusion) unsatisfiable subset of the initial set of clauses. Such subsets, called *unsatisfiable cores*, are obviously useful in applications like planning or routing for explaining why no feasible solution exists, but many other applications keep emerging, such as solving MAX-SAT problems [FM06,MSP08] or debugging software models [Jac02].

In addition, it is frequently helpful, or even necessary, to be able to check the unsatisfiability claims produced by a SAT solver, using some small and simple, independent, trusted checker for, e.g., resolution proofs. Note that, although for certain classes of formulas the minimal resolution proof is exponentially large [Hak85], for real-world problems the size tends to be manageable and frequently surprisingly small (as is the core).

Since Zhang and Malik's work in 2003 [ZM03], it is well-known that modern DPLL-based SAT solvers with learning can be instrumented to write a trace on disk from which, if the input is unsatisfiable, a resolution proof can be extracted (and checked), and hence also an *unsatisfiable core* a (frequently small) unsatisfiable subset of the input clauses explaining the reasons for the unsatisfiability.

Efficiently finding proofs and cores is important in many applications. The processing time becomes even more important when, for reducing the size of the core, one iteratively feeds it back into the SAT solver with the hope of generating a smaller one, until a fixpoint is reached (that may still not be minimal, so one can apply other methods for further reducing it, if desired). Efficiency is also important in other applications requiring features like the identification of all disjoint cores, i.e., all independent reasons for unsatisfiability or applications where cores are used for solving MAX-SAT problems [MSP08] or debugging software models [Jac02].

In Section 3 of this article we give a new algorithmic approach for processing the (frequently huge) traces generated in Zhang and Malik's approach. It achieves the efficiency of a depth-first traversal, while maintaing the property that memory usage remains upper bounded by the memory usage of the SAT solver that generated the trace.

However, for further enhancing efficiency, in-memory approaches are being developed. For instance the one implemented by Biere in PicoSAT [Bie08] essentially corresponds to storing the trace file of [ZM03] in main memory. In those cases where this is indeed feasible, i.e., if there is enough memory, this has several advantages over the trace file one. Not only does one avoid the inefficiencies caused by the use of external memory, but also, and more importantly, for retrieving the proof or the core one does not need to sequentially traverse the *whole* trace, but only those parts of it that appear in the proof. This gives one order of magnitude speedup in applications where cores or proofs have to be produced frequently [BKO+07,Bie08], and of course even more in the context of sophisticated (e.g., iterative) core/proof minimization techniques.

The second part of this article (Sections 4–6) is about such in-memory algorithms for generating SAT proofs and cores. We discuss advantages and disadvantages of the in-memory approach and investigate why the current SAT solvers with this feature still run out of memory on longer SAT runs. We analyze several of these in-memory algorithms, based on carefully designed experiments with our implementation of each one of them, as well as with (our implementation of) a trace-based one. Similar algorithms to the ones explained here were also applied in the context of first-order theorem proving, (e.g. [RV02] and [DS96]).

Our motivation for this work was that we also needed proofs and cores from longer runs. Moreover, we use SAT solvers inside other systems (e.g., for SAT Modulo Theories) where memory for the SAT solver is more limited. All this will become even more important if (multicore) processor performance grows faster than memory capacity.

Here we describe a new and better in-memory method that saves space by doing some bookkeeping to discard unnecessary information, and show that it can handle significantly more instances than the previously existing algorithms, at a negligible expense in time. We give a careful experimental comparison of it with the previous ones, which is non-trivial, since, for assessing different data structures and algorithms for SAT, it is crucial to develop implementations of each one of them, based on the same SAT solver, and *in such a way that the search performed by the SAT solver is always identical*. All software sources and benchmarks used here can be found at `www.lsi.upc.edu/~rasin`.

2. Short Overview on DPLL Algorithms for SAT

For self-containedness of the paper, here we give a short overview on DPLL based on the abstract presentation of [NOT06]. Let P be a fixed finite set of propositional symbols. If $p \in P$, then p is an *atom* and p and $\neg p$ are *literals* of P. The *negation* of a literal l, written $\neg l$ or $\bar{l}$, denotes $\neg p$ if l is p, and p if l is $\neg p$. A *clause* is a disjunction of literals $l_1 \vee \ldots \vee l_n$. A *unit clause* is a clause consisting of a single literal. The *empty clause* is a clause that has no literals. A (CNF) *formula* is a conjunction of zero or more clauses $C_1 \wedge \ldots \wedge C_n$. When it leads to no ambiguities, we will sometimes also write such a formula in set notation $\{C_1, \ldots, C_n\}$, or simply replace the $\wedge$ connectives by commas. A (partial truth) *assignment* M is a set of literals such that $\{p, \neg p\} \subseteq M$ for no p. M will be written as a sequence (also seen as a set when convenient) of (possibly annotated) literals with Ml meaning the concatenation of M with l. A literal l is *true* in M if $l \in M$, is *false* in M if $\neg l \in M$, and is *undefined* in M otherwise. A literal is *defined* in M if it is either true or false in M. A clause C is true in M if at least one of its literals is true in M. It is false in M if all its literals are false in M, and it is undefined in M otherwise, the empty clause is always false. A formula F is true in M, or *satisfied* by M, denoted $M \models F$, if all its clauses are true in M. In that case, M is a *model* of F. If F has no models then it is *unsatisfiable*. Then, any formula F containing the empty clause is unsatisfiable. If

Fig. 1. Set of rules that model a DPLL procedure

UnitPropagate :

$$M \parallel F,\, C \vee l \implies M\ l \parallel F,\, C \vee l \quad \textbf{if} \begin{cases} M \models \neg C \\ l \text{ is undefined in } M \end{cases}$$

Decide :

$$M \parallel F \implies M\ l^{\mathsf{d}} \parallel F \quad \textbf{if} \begin{cases} l \text{ or } \neg l \text{ occurs in a clause of } F \\ l \text{ is undefined in } M \end{cases}$$

Fail :

$$M \parallel F,\, C \implies \mathit{Fail} \quad \textbf{if} \begin{cases} M \models \neg C \\ M \text{ contains no decision literals} \end{cases}$$

Backjump :

$$M\ l^{\mathsf{d}}\ N \parallel F,\, C \implies M\ l' \parallel F,\, C \quad \textbf{if} \begin{cases} M\ l^{\mathsf{d}}\ N \models \neg C, \text{ and there is} \\ \text{some clause } C' \vee l' \text{ such that:} \\ \quad F, C \models C' \vee l' \text{ and } M \models \neg C', \\ \quad l' \text{ is undefined in } M, \text{ and} \\ \quad l' \text{ or } \neg l' \text{ occurs in } F \text{ or in } M\ l^{\mathsf{d}}\ N \end{cases}$$

Learn :

$$M \parallel F \implies M \parallel F,\, C \quad \textbf{if} \begin{cases} \text{each atom of } C \text{ occurs in } F \text{ or in } M \\ F \models C \end{cases}$$

Forget :

$$M \parallel F,\, C \implies M \parallel F \quad \textbf{if} \ \{ F \models C$$

F and F' are formulas, we write $F \models F'$ if F' is true in all models of F. Then we say that F' *is entailed by* F, or is a *logical consequence* of F. If C is a clause $l_1 \vee \ldots \vee l_n$, we write $\neg C$ to denote the formula $\neg l_1 \wedge \ldots \wedge \neg l_n$.

A *state* of the DPLL procedure is a pair of the form $M \parallel F$, where F corresponds to a (CNF) formula, and M is, essentially, a (partial) assignment. A literal l may be annotated as a *decision literal* (see below), writing it as l^{d}. We say that a state M *is at decision level* n if in M there are n literals marked as decisions. A clause C is *conflicting* in a state $M \parallel F, C$ if $M \models \neg C$. A DPLL procedure can be modeled by a set of rules over such states (see figure 1).

- The Decide rule represents a case split: an undefined literal l is chosen and added to the model, annotated as a decision literal.
- UnitPropagate forces a literal l to be true if there is a clause $C \vee l$ in F whose part C is false in M.
- By Learn one can add any entailed clause to F. Learned clauses prevent repeated work in *similar* conflicts, which frequently occur in industrial problems having some regular structure.
- Since a lemma is aimed at preventing future similar conflicts, it can be removed using Forget, when such conflicts are not very likely to be found again. In practice, a lemma is removed when its *relevance* (see, e.g., [BS97]) or its *activity* level drops below a certain threshold; the activity can be, e.g., the number of times it becomes a unit or a conflicting clause [GN02].
- Fail rule applies only when a conflicting clause C is detected and M contains no decision literals (i.e. when there is a clause at decision level zero). FailState state is then produced and search ends.
- On the other hand, if there is some decision literal in M and an entailed conflicting clause, then one can always find (and Learn) a *backjump clause*, an entailed clause of the form $C \vee l'$, such that Backjump using $C \vee l'$ applies. Good backjump clauses can be found by *conflict analysis* of the conflicting clause [MSS99,ZMMM01]. To better understand how the Backjump and Learn rules work we refer to the example of Section 3.

Modern DPLL-based solvers also frequently *restart* the search. This is somewhat orthogonal to the subject of this paper and we refer to [NOT06] for futher details on this and DPLL in general.

For deciding the satisfiability of an input formula F, one can generate an arbitrary derivation

$\emptyset \parallel F \implies \ldots \implies S_n$, where S_n is a final state (no rule applies). Under simple conditions, this always terminates. Moreover, for every derivation like the above ending in a final state S_n, (i) F is unsatisfiable if, and only if, S_n is *Fail*, and (ii) if S_n is of the form $M \parallel F$ then M is a model of F (see [NOT06] for all details).

Also for self-containedness of the paper, we define *resolution* between two clauses and give the concept of a *resolution proof* of some entailed clause.

The *(binary) Resolution* rule is the following inference rule with two clauses as premises and another clause as conclusion:

$$\frac{x \vee C \qquad \neg x \vee D}{C \vee D} \qquad \textit{Resolution}$$

Let F be a set of clauses and let C be a clause. A *Resolution proof* of C from F is a directed acyclic graph where:

- each vertex is (labeled by) a clause
- C is its single sink vertex (i.e., a vertex with no outgoing edges)
- the source vertices (no incoming edges) are clauses from S
- every non-source vertex has two incoming edges from clauses from which it can be obtained in one Resolution step.

3. Zhang and Malik's trace-based method

Since Zhang and Malik's work in 2003 [ZM03], it is well-known that modern DPLL-based solvers with learning can be instrumented to write a trace on disk from which a resolution proof can be extracted and checked.

In this approach, essentially, each learned clause generates a line in the trace with only the list of its parents' identifiers (ID's), i.e., the ID's of the clauses involved in the conflict analysis, which is a sequence of resolution steps (see the example below).

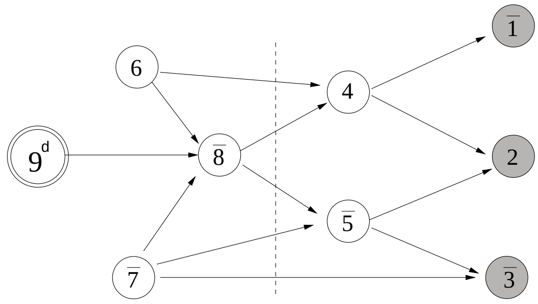
When unsatisfiability is detected, that is, a conflict with no decision literals in the current assignment, it provides a last line in the trace corresponding to the parents list of the empty clause. By processing the trace file backwards from this last line one can hence reconstruct a resolution proof and find the subset of the clauses in the original formula F that is used in it.

We explain the technique by means of the following example (see [NOT06] Section 2 for details). Consider, among others, a set of clauses:

$$\begin{array}{llll} \overline{9}\vee\overline{6}\vee7\vee\overline{8} & 8\vee7\vee\overline{5} & \overline{6}\vee8\vee4 & \overline{4}\vee\overline{1} \\ \overline{4}\vee5\vee2 & 5\vee7\vee\overline{3} & 1\vee\overline{2}\vee3 & \end{array}$$

and a state of the DPLL procedure where the partial assignment M is of the form: $\ldots 6 \ldots \overline{7} \ldots 9^{\mathsf{d}}\ \overline{8}\ \overline{5}\ 4\ \overline{1}\ 2\ \overline{3}$. It is easy to see that this state can be reached after the last *decision* 9^{d} by six *unit propagation* steps with these clauses (in the given order).

For example, $\overline{8}$ is implied by 9, 6, and $\overline{7}$ because of the first clause. Now, the clause $1\vee\overline{2}\vee3$ is *conflicting* (it is false in the current assignment), and working backwards from it we get an *implication graph*:



where the so-called *1UIP cut* (the dotted line, see [MSS99,MMZ+01]) gives us the *backjump clause* $8 \vee 7 \vee \overline{6}$ that is *learned* as a lemma. For those who are more familiar with resolution, this is simply a backwards resolution proof on the conflicting clause, resolving away the literals $3, \overline{2}, 1, \overline{4}$ and 5, in the reverse order their negations were propagated, with the respective clauses that caused the propagations:

$$\dfrac{8\vee7\vee\overline{\mathbf{5}} \qquad \dfrac{\overline{6}\vee8\vee\mathbf{4} \qquad \dfrac{\overline{4}\vee\overline{\mathbf{1}} \qquad \dfrac{\overline{4}\vee5\vee\mathbf{2} \qquad \dfrac{5\vee7\vee\overline{\mathbf{3}} \qquad 1\vee\overline{2}\vee\mathbf{3}}{5\vee7\vee1\vee\overline{\mathbf{2}}}}{\overline{4}\vee5\vee7\vee\mathbf{1}}}{5\vee7\vee\overline{\mathbf{4}}}}{\overline{6}\vee8\vee7\vee\mathbf{5}}}{8\vee7\vee\overline{6}}$$

until reaching a clause with only one literal of the current decision level (here, literal 8). This clause $8\vee7\vee\overline{6}$ will be learned, adding it as a new clause (a lemma). It allows one to *backjump* to the state $\ldots 6 \ldots \overline{7}\, 8$, as if it had been used on $\ldots 6 \ldots \overline{7}$ for unit propagation.

It is easy to see that this kind of linear resolution proofs can be reconstructed with as only information the ordered list of clauses that are resolved. This is true because the proof has a linear structure, and because in each binary resolution step the two given clauses have only one literal that can be resolved upon (otherwise the conclusion would be a tautology, which is never the case in this context).

Now assume that an input clause's ID is simply its line number in the input CNF file, and that a lemma's ID is its line number in the trace file, i.e., the line containing the ordered list of its parents' ID's (some additional prefix can be used to distinguish lemma ID's from input clauses ID's).

Then it is clear that one can reconstruct and check the whole resolution proof from (i) the input clauses file and (ii) the ordered list of parent ID's at each line of the trace file.

The overhead in time for producing the trace file is usually small (typically around 10 per cent, [ZM03], see Section 6), but the traces quickly become large (hundreds of MB from a few minutes run, and several GB from long runs). Therefore, extracting from it the proof or the core, i.e., the leaves of the proof DAG, may be expensive.

This is especially the case since usually the trace does not fit into memory, and hence in [ZM03] a breadth-first processing of the trace is proposed that is guaranteed not to exceed the memory usage of the SAT solver that generated the trace. Here we propose a simple implementation idea that achieves the same property, but with an algorithm that is as efficient as the depth-first one, and hence, according to the experimental results of [ZM03] much more efficient than the breadth-first approach (which we did not re-implement ourselves).

The first step is to reverse the whole trace file line-wise, i.e., in an N-line trace, line i becomes line $N - i + 1$ in the reversed trace. The linux `tac` (the reverse `cat`) command does this very efficiently (in time negligible w.r.t. the rest of our algorithm).

In [GN03] an alternative method to check unsatisfiability (and also extract the core) from a trace is presented. This method, nevertheless, is more time consuming than the one presented below (specially for the core extraction process).

3.1. Processing reversed traces: cores

Let us first consider unsatisfiable core extraction. The first line of the reversed trace will consist of the ID's of the parents of the empty clause. These ID's will become *active*, meaning that they participate in the proof of the empty clause. These active ID's are stored in some datastructure (e.g., a hash table) where one can check in constant time whether a given ID is active or not. While traversing the reversed trace, at each line one can hence determine whether the given line number corresponds to an active ID or to a passive one. If the current line is active, all its ID's are set to active too (note that some may already be active beforehand since the resolution proof is a DAG rather than a tree).

In addition, if a line corresponding to an active ID is reached, then this ID can be removed from the datastructure of active ID's, since no more uses of it will be found in the (reversed) trace. As a consequence, this datastructure will never contain more ID's than the maximum number of clauses simultaneously kept by the SAT solver that generated the trace.

Each time an input clause's ID is set to active, this means that this input clause belongs to the unsatisfiable core. The process terminates (possibly before the whole reversed trace is scanned) when the set of active ID's becomes empty.

3.2. Processing reversed traces: proofs

We now describe an extension of the previous core-extraction method for generating and/or on-the-fly checking the resolution proof. Note that this involves actually determining the literals of the intermediate clauses and, if desired, checking all resolution steps for deriving the empty clause.

The **first stage** of the algorithm is similar to the aforementioned method for core extraction from the reversed trace, but where instead of keeping an active/passive bit, a counter is kept that is increased each time another use of an ID is detected. In this way for all clauses appearing in the proof its total number of uses is computed.

The **second stage** of the algorithm is a forward pass over the original unreversed trace. Each time an active clause's line is visited, one can infer its list of literals by reconstructing its linear resolution proof from the (ordered) list of parent clauses that are being resolved.

Each time a parent clause is used in such a step, its counter is decremented; when the counter reaches zero, this means that this parent clause has no more children and hence there is no need to store its literal list any longer. Note that in this way no more clauses are ever stored than in the SAT solver generating this trace. In fact significanty less clauses are stored, since here it is known (from the information collected during the first stage) which ones participate in the proof and only these ones are kept.

This counting method for deleting lemmas that are not longer needed is also used in the breadth-first algorithm of [ZM03], but there also the lemmas that do *not* participate in the proof are generated and kept (while they have any "active" children). On the other hand, in our approach between the two stages we need to keep a counter for *every* clause that participates in the proof, and not only for the ones that are active at any point of the second stage; but note that even for proofs from extremely long SAT runs, with many millions of new clauses (i.e., conflicts), keeping these counters still causes no important additional memory consumption. The use of *reference counters* is also briefly mentioned in [Bie08]. This same idea of active-children counting will be used later on in this article in the in-memory algorithm introduced in Section 6.2.

4. In-Memory Algorithms

To overcome the inefficiencies of the trace file approach, in what follows we study four alternative in-memory algorithms for generating unsatisfiability proofs and cores using DPLL-based propositional SAT solvers. Here we first give a short description of each one of them.

The first algorithm is based on adding one distinct new *initial ancestor (IA)* marker literal to each initial clause. These literals are set to false from the beginning. Then the solver is run without ever removing these false IA-marker literals from clauses, and the empty clause manifests itself as a clause built solely from IA-marker literals, each one of which identifies one initial ancestor, that is, one clause of the unsatisfiable core. This idea appears to be quite widely applied (e.g., in SAT Modulo Theories). As far as we know, it stems from the Minisat group (around 2002, Eén, Sörensson, Claessen). It requires little implementation effort, but here, in Subsection 5.1 we give experimental evidence showing that it is extremely inefficient in solver time and memory and explain why.

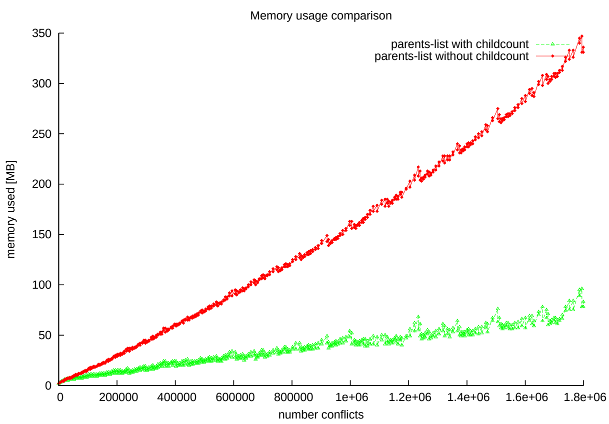
Our second algorithm, given in Subsection 5.2, tries to overcome these shortcomings by storing initial ancestor information at the meta level along with the clauses: each clause has an attached list with the ID's of its initial ancestors. This reduces part of the overhead of the first algorithm. However, our experiments reveal that also this method is still far too expensive in memory, especially in combination with certain clause simplification methods, which on the other hand, when turned off, slow down the solver too much.

The third algorithm (Section 6.1) stores the immediate parents list along with each clause. The problem with this approach is that if a low-activity clause is deleted (as usual in modern SAT solvers), its associated parent information can be removed only if this clause has generated no children (the *literals* of deleted clauses need not be kept, though).

This approach, implemented by Biere in PicoSAT [Bie08], essentially corresponds to storing the trace file of [ZM03] in main memory. As said, in those cases where this is indeed feasible, i.e., if there is enough memory, this has several advantages over the trace file one. One not only avoids the inefficiencies caused by the use of external memory, but also, and more importantly, for retrieving the proof or the core one does not need to sequentially traverse the *whole* trace, but only those parts of it that appear in the proof. According to [BKO+07,Bie08] this gives an order of magnitude speedup in applications where cores or proofs have to be produced frequently, and, as a consequence, even more in the context of certain sophisticated core/proof minimization techniques such as iteration (e.g., to fixpoint) as in [ZM03].

Our new in-memory algorithm, described in Section 6.2, keeps only the potentially needed parent information. The idea is to keep for each clause also a counter of how many of its children do have some active descendant. If it becomes zero the parent information can be removed. Here we show that (i) when implemented carefully, the overhead on the SAT solver time is still essentially negligible

Fig. 2. goldb-heqc-rotmul memory usage of Biere and Childcount methods



(around 5 per cent, similar to Biere's approach) and (ii) the memory usage frequently grows significantly slower.

As figure 2 shows, and as expected, in Biere's approach (parents-list without childcount) memory usage always grows linearly in the number of conflicts (or more, since parents lists get longer in longer runs). In our ChildCount approach (parents-list with childcount), performing exactly the same search on this instance (`goldb-heqc-rotmul`; cf. Section 6.3 for many more experimental results), one can see in the figure that on this particular example memory usage grows much slower. Skews in the plot correspond to clause deletion phases of the solver.

5. Basic Algorithms, Only for Core Extraction

In this section we introduce and compare two basic algorithms that can be used for extracting unsatisfiable cores, but not unsatisfiability proofs.

5.1. First Algorithm: Marker Literals

As said, in this approach one adds to each initial clause C_i one distinct new *initial ancestor (IA)* marker literal, say, a positive literal y_i. These literals are set to false from the beginning, and hence the logical meaning of the clause set does not change.

Then the solver is run, but without applying to the y_i-literals the usual simplification technique of removing from all clauses the literals that are false at decision level zero (henceforth: *false literal deletion*). In every lemma that is generated, its subset of y_i-literals shows exactly the subset of the initial clauses it has been derived from. In such a run, unsatisfiability is then witnessed by the appearance of an "empty clause" built solely from y_i-literals, i.e., a clause of the form $y_{j_1} \vee \ldots \vee y_{j_k}$, indicating that $\{C_{j_1}, \ldots, C_{j_k}\}$ is an unsatisfiable core. Note that this technique can only be used for finding unsatisfiable cores, and not for generating a resolution proof, since the proof structure is lost.

The interesting aspect of this method is that it requires very little implementation effort. However, it leads to important inefficiencies in the SAT solver. Clauses can become extremely long, using large amounts of memory, and for clauses that without the y_i-literals would have been units or two-literal clauses this is no longer the case. This leads to an important loss of efficiency in, for instance, the unit propagation data structures and algorithms.

5.2. Second Algorithm: Initial Ancestor Lists

An obvious way for overcoming the shortcomings of the previous algorithm is by storing initial ancestor information at the meta level along with the clauses, instead of adding dummy literals for this. Therefore in this second algorithm each clause has an attached list with the ID's of its initial ancestors. This reduces part of the overhead of the first algorithm. For example, unit clauses are really treated as such, and false literal detection is not hindered by the additional IA literals.

In most DPLL-based SAT solvers, unit clauses and two-literal clauses are not explicitly stored as such. Units are usually simply set to true in the assignment at decision level zero, whereas binary clauses are typically kept in an adjacency list data structure, i.e., for each literal l there is a list of literals $l_1 \ldots l_n$, such that each $l \vee l_i$ is a binary clause. This is much faster and memory-efficient for unit propagation than the standard two-watched literal data structures that are used for longer clauses.

In the algorithm for core extraction given here, we also need to store the IA information for unit clauses and two-literal clauses. This is done here in a memory bank separate from the one of the other clauses. Since one- and two-literal clauses are never removed in our solver, neither is their IA information.

5.3. *Experiments: the First Two Algorithms vs Our Basic Solver*

In table 1 we compare a basic version of our own Barcelogic SAT solver without proof or core extraction (column **Basic**) with the two algorithms described in this section (**marker lits** and external **IAs**). Each one of these two algorithms is implemented on top of the basic version with the minimal amount of changes. In particular, binary clauses are still represented in their efficient special form and no unit propagation using longer clauses is done if there is any pending two-literal clause propagation.

As said, for the algorithm based on marker literals we had to turn off false literal deletion. For the IA algorithm, each time a clause $C \vee l$ with IA list L_1 gets simplified due the decision level zero literal $\neg l$ with IA list L_2, the new clause C gets the IA list $L_1 \cup L_2$. It turns out that the IA lists became long and memory consuming. Therefore for this first experiment also in the **IAs** algorithm we switched off false literal deletion, which slowed down the solver and also made it search differently with respect to the basic version, but it prevented memory outs. Also to prevent memory outs, we were doing very frequent clause deletion rounds: every 5000 conflicts we were deleting all zero-activity clauses. To make the comparison fairer, we also did this in the basic algorithm, for which this is not precisely its optimal setting.

Note that therefore all three versions of the solver perform a different search[1] and hence, due to "luck" a core-generating version could still be faster than the basic one on some particular benchmark. All experiments were run on a 2.66MHz Xeon X3230, giving each process a 1.8GB memory limit and a timeout limit of 90 minutes. Times are indicated in seconds, and time outs are marked here with TO. The table is split into two parts. The first part has the unsatisfiable problems from the qualification instance sets of the 2006 SAT Race (SAT-Race_TS_1 and 2, see `fmv.jku.at/sat-race-2006`) taking between 5 and 90 minutes in our basic solver. The second part has much easier ones. In all experiments the unsatisfiability of the extracted cores has been verified with independent SAT solvers.

[1]Below there is a version of the IA algorithm *with* false literal detection that does perform the same search as the basic version.

Table 1
Times for Basic, marker literals an IAs solvers

Runtimes (seconds)			
Instance	**Basic**	**marker lits**	**IAs**
manol-pipe-cha05-113	448	5035	786
manol-pipe-f7idw	546	2410	1181
6pipe	717	TO	1324
manol-pipe-g10idw	830	4171	2299
manol-pipe-c7idw	1534	TO	3701
manol-pipe-c10b	1938	TO	3926
manol-pipe-g10b	1969	TO	5365
manol-pipe-c6bid_i	2219	TO	4253
manol-pipe-g10ni	2419	TO	4412
manol-pipe-g10nid	2707	TO	TO
manol-pipe-c6nidw_i	2782	TO	TO
velev-dlx-uns-1.0-05	3306	1028	TO
goldb-heqc-frg2mul	3891	TO	TO
7pipe_q0_k	4184	TO	TO
manol-pipe-g10bidw	4650	TO	TO
goldb-heqc-i8mul	4911	TO	TO
hoons-vbmc-s04-06	TO	4543	TO
2dlx-cc-mc-ex-bp-f	1.81	2.91	1.35
3pipe-1-ooo	1.45	1.91	0.71
3pipe-3-ooo	1.92	3.53	1.59
4pipe-1-ooo	3.56	8.77	4.57
4pipe-3-ooo	5.38	11.67	5.36
4pipe-4-ooo	6.90	20.35	7.31
4pipe	8.15	33.64	14.82
5pipe-1-ooo	11.32	20.52	12.51
5pipe-2-ooo	10.31	18.98	14.33
5pipe-4-ooo	21.41	52.64	54.54
cache.inv14.ucl.sat. chaff.4.1.bryant	13.36	75.23	18.85
ooo.tag14.ucl.sat. chaff.4.1.bryant	7.05	6.78	7.96
s1841184384-of-bench-sat 04-984.used-as.sat04-992	2.07	4.62	1.97
s57793011-of-bench-sat 04-724.used-as.sat04-737	9.10	66.05	10.36
s376420895-of-bench-sat 04-984.used-as.sat04-1000	2.50	5.48	2.28

From the results of table 1 it follows that these techniques are not practical except for very simple problems.

It is well-known that DPLL-based SAT solvers are extremely sensitive in the sense that any small change (e.g., in the heuristic or in the order in which input clauses or their literals are given) causes the solver to search differently, which in turn can cause dramatic changes in the runtime on a given instance. Therefore, most changes in SAT solvers are hard to assess, as they can only be evaluated by running a statistically significant amount of problems and measuring aspects like runtime averages. For this reason, all experiments mentioned from now on in this paper have been designed in such a way that for each method for proof/core extraction our solver performs *exactly the same search* (which was impossible in the algorithm with marker literals). This allows us to measure precisely the overhead in runtime and memory consumption due to proof/core generation bookkeeping.

Table 2 compares our basic solver on the easy problems with the IAs method, in runtime and in memory consumption. Here MO denotes memory out (> 1.8 GB) after the indicated number of seconds. The difference in times with the previous table comes from the fact that here the setting of the solver is the standard one, with less frequent clause deletion phases, and with false literal deletion. As said, false literal deletion makes the IAs method even more memory consuming and also slower, as longer lists of parents have to be merged.

As we can see, usually only on the very simple problems the runtimes are comparable. As soon as more than few seconds are spent in the basic version, not only does the memory consumption explode, but also the runtime due to the bookkeeping (essentially, computing the union of long parents lists and copying them).

Table 2
Time and Memory for Basic and IAs method (same search)

Basic vs IAs (same search, **T**ime in seconds, **M**emory in MB)				
Instance	**T Basic**	**M Basic**	**T IAs**	**M IAs**
2dlx-cc-mc-...	1.64	3	4.17	298
3pipe-1-ooo	1.35	3	2.07	122
3pipe-3-ooo	1.78	5	3.99	215
4pipe-1-ooo	3.98	14	22.76	843
4pipe-3-ooo	4.88	13	30.08	1175
4pipe-4-ooo	7.14	19	36.14	MO
4pipe	11.35	47	32.80	1106
5pipe-1-ooo	10.52	24	55.53	MO
5pipe-2-ooo	10.30	23	50.92	MO
5pipe-4-ooo	33.08	65	42.87	MO
cache.inv14..	12.75	5	39.43	MO
ooo.tag14..	6.21	3	9.12	612
s1841184384..	1.83	1	1.86	51
s57793011-..	7.75	32	8.47	74
s376420895-..	1.99	1	2.37	89

6. Algorithms for Extracting Proofs and Cores

Here we analyze more advanced algorithms that are not only able to extract unsatisfiable cores, but also resolution proof traces, i.e., the part of the trace that corresponds to the resolution proof.

6.1. In-Memory Parent Information

We now consider the in-memory method, a simpler version of which is implemented in the PicoSAT solver [Bie08]. Here, along with each clause the following additional information is stored: its ID, its list of immediate parents' ID's, and what we call its *is-parent bit*, saying whether this clause has generated any children itself or not. The parents list is what one would write to the trace in the [ZM03] technique. Each time a new lemma is generated, it gets a new ID, its is-parent bit is initialized to false, the ID's of its parents are collected and attached to it, and the is-parent bit of each one of its parents is set to true. In this approach, the parent information of a deleted clause (by application of the Forget rule during the clause deletion phase of the SAT solver) is removed only if its is-parent bit is false.

Once the empty clause is generated (i.e., a conflict at level zero appears), one can recover the proof by working backwards from it (without the need of traversing the whole trace, and on disk, as in [ZM03]).

In our implementation of this method, *unlike what is done in PicoSAT*, we maintain the special-purpose two-literal clause adjacency-list representation also when the solver is in proof-generation mode. Hence the performance slowdown with respect to our basic reference solver corresponds *exactly* to the overhead due to the bookkeeping for proof generation. Our implementation treats all conflicts in a uniform way, including the one obtaining the empty clause. This in in contrast to what is done with the final decision-level-zero conflict in Zhang and Malik's trace format, which gets a non-uniform treatment in [ZM03] (in fact, the explanations given in section 3 correspond to our simplified uniform view where the empty clause has its conflict analysis like any other clause).

The parents lists of units and binary clauses are stored in a separate memory zone, as we also did for the IAs method. Unit and binary clauses are never deleted in our solver. Essentially, at SAT solving time (more precisely, during conflict analysis) what is required is a direct access to the ID of a given clause. For unit and binary clauses we do this by hashing (for the larger clauses this is not necessary, since the clause, along with all its information and literals, is already being accessed during conflict analysis). At proof extraction time, one needs direct access to the parent list corresponding to a given clause ID. This we do by another hash table that only exists during proof extraction. In [Gel02] another in-memory algorithm for extracting shorter proofs is presented; however, it is reported to need much more memory and time than the algorithms explained here.

6.2. Our New Method with Child Count

The idea we develop in this section is the following: instead of just an is-parent bit, we keep along with each clause a counter, called the *childcounter*, of how many of its children have some *active* descendant. Here a clause is considered active if it participates in the DPLL derivation rules that are implemented in the SAT solver. In our solver, that is the case if it has at most two literals (these clauses are never deleted in our solver) or if it has at least three literals and has not been removed by the Forget rule (i.e., it is being *watched* in the two-watched literal data structure for unit propagation [MMZ+01]).

If the childcounter becomes zero also the parent information can be removed, since this clause can never appear in a proof trace of the empty clause (obtained from active clauses only). Note that this is a recursive process: each time a clause C is selected for deletion, i.e., when C goes from active to non-active, if its childcounter is zero then a recursive `childcounter-update`(C) process starts:

For each parent clause PC of C,

1. Decrease by one the childcounter of PC.
2. If the childcounter now becomes zero and PC is non-active, then do `childcounter-update`(PC).
3. Delete all information of C.

We have again implemented this method on top of our basic Barcelogic solver, and again we have done this in such a way that the search is not affected, i.e., again the additional runtime and memory consumption with respect to our basic solver correspond exactly to the overhead due to the bookkeeping for proof generation.

As before, during conflict analysis again we need to add the parent's ID's to the parent list of the new lemma, but now, in addition, the childcounters of these parents are increased. For this, as before, we use hashing to retrieve the ID of parent clauses with less than three literals. For the parent clauses with at least three literals this is not necessary, since these clauses, along with all their information and literals, are already being accessed during conflict analysis.

The main additional implementation issue is that now during the clause deletion phase, when doing `childcounter-update`(C), given the ID of an (active or non-active) clause, we may need access its information (their childcounters and parent lists). For this we use an additional hash table, which supposes only a negligible time overhead. Note that the clause deletion phase is not invoked very frequently and takes only a small fraction of the runtime of the SAT solver.

6.3. Experiments

We have run experiments with the same unsatisfiable instances as before (the harder ones): from the qualification instance sets of the 2006 SAT Race (SAT-Race_TS_1 and 2), the ones taking between 250 seconds and 90 minutes. Here again we run our solver in its standard settings, with false literal deletion and less frequent clause deletion phases.

In all experiments the correctness of the extracted proofs has been verified with the TraceCheck tool, see [Bie08] and `fmv.jku.at/tracecheck`, and a simple log information has been used to verify that indeed exactly the same SAT solving search was taking place in all versions.

Time consumption is analyzed in table 3 (where instances are ordered by runtime) which has the

Table 3
Times for Basic, Biere, Biere-b, Childcount and trace on disk (Same search)

Time (s)								
Instance	**basic**	**Biere**		**Biere-b**		**disk**	**Child**	
	solve	solve	slv+tr	solve	slv+tr	solve	solve	slv+tr
manol-pi-cha05-113	254	265	269	265	269	273	267	271
manol-pipe-f7idw	257	268	270	268	269	279	272	273
manol-pipe-c7idw	348	362	364	361	363	372	365	367
manol-pipe-g10idw	412	433	444	432	443	453	438	444
manol-pipe-c10b	527	550	561	546	558	567	555	564
goldb-heqc-i8mul	577	601	644	604	648	635	611	648
velev-dlx-uns-1.0-5	696	729	735	731	736	738	727	729
manol-pipe-c6bid_i	748	780	790	771	780	800	788	794
6pipe	785	846	858	844	856	850	854	861
velev-pipe-uns-1.1-7	829	928	948	930	949	941	931	940
manol-pipe-c6nidw_i	885	923	937	923	936	949	920	928
manol-pipe-g10nid	1030	1073	1080	1073	1079	1116	1071	1074
hoons-vbmc-s04-06	1053	1084	1099	1088	1103	1110	1107	1118
7pipe_q0_k	1551	1725	1776	1718	1764	1781	1751	1768
manol-pipe-g10bidw	1709	MO	MO	1774	1783	1856	1773	1777
manol-pipe-g10ni	1788	MO	MO	MO	MO	2075	2029	2033
manol-pipe-c7nidw	4059	MO	MO	MO	MO	4385	4209	4238
manol-pipe-c7bidw_i	4255	MO	MO	MO	MO	4646	4414	4445

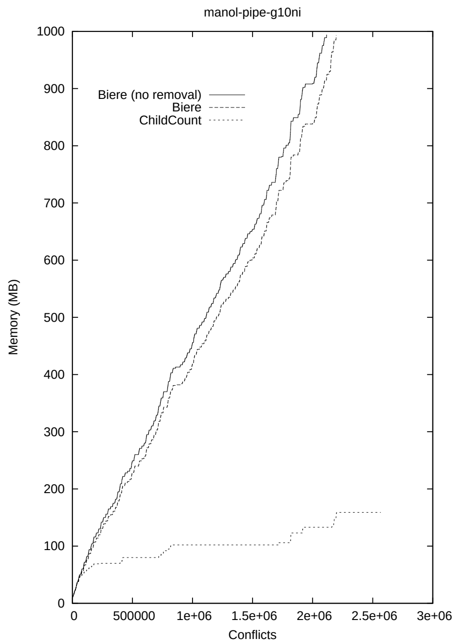
Table 4
Memory usage for Basic, Biere, Biere-b, Childcount and Trace on disk (Same search)

	Num.	**Time**	**Memory Usage** (MB)				**Trace** (MB)	
Instance	**cnflcts**	(s)	**Basic**	**Biere**	**Biere-b**	**Child**	**full**	**proof**
velev-dlx-uns-1.0-05	199390	696	129	239	234	229	226	30
manol-pipe-f7idw	333275	257	35	140	127	78	183	24
manol-pipe-cha5-113	336968	254	108	228	218	167	218	112
goldb-heqc-i8mul	397702	577	376	MO	972	947	1002	937
manol-pipe-g10idw	423079	412	137	434	412	285	550	191
manol-pipe-c10b	530022	527	128	347	330	240	452	258
manol-pipe-c7idw	536341	348	107	209	192	141	217	42
manol-pipe-c6bid-i	1123035	748	105	393	347	187	543	166
manol-pipe-c6nidw-i	1256752	885	117	488	436	252	671	226
hoons-vbmc-s04-06	1301190	1053	21	320	309	228	358	322
manol-pipe-g10nid	1327600	1030	75	613	557	144	986	82
6pipe	1377876	785	316	519	502	418	433	205
velev-pipe-uns-1.1-7	1761066	829	69	447	409	210	751	260
manol-pipe-g10bidw	2250890	1709	70	MO	892	146	1679	100
manol-pipe-g10ni	2566801	1788	85	MO	MO	159	2050	113
7pipe-q0-k	3146242	1551	81	810	753	342	1381	472
manol-pipe-c7nidw	3585110	4059	219	MO	MO	613	2388	692
manol-pipe-c7bidw-i	4011227	4255	117	MO	MO	643	2834	761

following columns: **basic**: our basic SAT solver without proof/core generation, **Biere**: the same solver extended with Biere's in-memory core generation, **Biere-b**: the same, also extended with is-parent bit, **disk**: the basic solver writing traces to disk, as in [ZM03], **Child**: our method with child count. Columns "solve" include just the solving time (all version performing exactly the same search), and "slv+tr" includes as well the time needed for traversing the in-memory data structures and writing to disk the part of the trace that contains the unsatisfiability proof. The entries labelled "MO" correspond to "Memory Out", which means more than 1.8GB.

The differences in runtime between our basic SAT solver without proof/core generation and its versions that do the necessary bookkeeping for in-memory proof/core generation are always very small, usually around five percent or less, and always less than the trace generation technique of [ZM03]. We conjecture that this is mainly because of the inefficiencies in writing to disk of the latter method (see below examples of the size of the traces that are written) since it requires less additional bookkeeping than the in-memory techniques. Note that our Childcount method in principle needs to do more work for generating the trace.

Much more important and interesting are the differences in memory usage. The plot we give below compares memory usage of three methods: (i) Biere's method without the is-parent bit (called "no removal" in the plot) i.e., where parent information is never deleted, (ii) Biere's method with the is-parent bit as explained here in Section 6.1, and (iii) our method with Childcount. We do this for one of the instances that generate many conflicts.



As we can see in table 4 (where "**Time**" refers to the runtime of our basic SAT solver, and column "**Biere-b**" is the one with is-parent bit), the benefits of our Childcount methods are less important on examples that are solved generating fewer conflicts. The is-parent bit of Biere's methods has only a very limited impact. In the last two columns we also show the size of the whole DPLL trace on disk (**"full"**) produced by the method of [ZM03], and the size of its subset corresponding to the just the *proof trace* (**"proof"**), i.e., the proof of the empty clause, as it is generated by the methods Biere, Biere-b, and Childcount (which all three produce exactly the same proof trace in our implementations). Since the entire DPLL trace is usually much larger than just the proof trace, the in-memory methods are also faster if one writes to disk the proof trace once the unsatisfiability has been detected (although for many applications, such as core minimization, this is not needed).

In these implementations we have not considered compression methods such as Biere's Delta Encoding, which compresses parents lists up to four times [Bie08], since this is a somewhat orthogonal issue that can be applied (or not) to both methods.

7. Conclusions and Future Work

We have shown that it is possible to handle trace files efficiently, without exceeding the memory usage of the SAT solver that generated the trace. Since the trace-based approach requires a smaller implementation effort than the in-memory approaches, and is not limited by the capacity of main memory, this may still be a good choice for some applications.

We have also carried out a systematic and careful implementation of different methods for in-memory unsatisfiable core and proof generation, which may be preferrable for applications where efficiency is a primary concern.

Regarding the two simpler methods for generating cores, our IA technique is indeed slightly more efficient than the one based on marker literals, but none of both is useful for instances on which our solver (using its default settings) takes more than few seconds. We have also shown that the techniques for generating cores and proofs explained in Section 6 are applicable to large SAT solving runs, and moreover allow one to keep the standard setting of the solver without a significant overhead in runtime.

Our experiments clearly show that our Child-count technique makes it possible to go significantly beyond previous in-memory techniques in terms of memory requirements. We plan to implement it in combination with Biere's Delta Encoding compression technique, which will make it possible to handle even longer DPLL runs or use even less memory. We also plan to use the basic underlying algorithms given here inside algorithms for core-minimization and for applications using cores (which are both outside the scope of this paper).

References

[Bie08] A. Biere. PicoSAT essentials. *Journal on Satisfiability, Boolean Modeling and Computation*, page 75, 2008.

[BKO+07] Randal E. Bryant, Daniel Kroening, Joël Ouaknine, Sanjit A. Seshia, Ofer Strichman, and Bryan A. Brady. Deciding bit-vector arithmetic with abstraction. In *Proccedings of 13th Tools and Algorithms for the Construction and Analysis of Systems, Conference, (TACAS)*, pages 358–372. Springer LNCS 4424, 2007.

[BS97] Roberto J. Jr. Bayardo and Robert C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI'97)*, pages 203–208, Providence, Rhode Island, 1997.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960.

[DS96] JRG DENZINGER and STEPHAN SCHULZ. Recording and analysing knowledge-based distributed deduction processes. *Journal of Symbolic Computation*, 21(4-6):523 – 541, 1996.

[FM06] Zhaohui Fu and Sharad Malik. On solving the partial MAX-SAT problem. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 252–265, 2006.

[Gel02] Allen Van Gelder. Extracting (easily) checkable proofs from a satisfiability solver that employs both preorder and postorder resolution. In *Seventh Int'l Symposium on AI and Mathematics (AMAI)*, 2002.

[GN02] E. Goldberg and Y. Novikov. BerkMin: A fast and robust SAT-solver. In *Design, Automation, and Test in Europe (DATE '02)*, pages 142–149, 2002.

[GN03] Evgueni Goldberg and Yakov Novikov. Verification of proofs of unsatisfiability for cnf formulas. In *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, page 10886, Washington, DC, USA, 2003. IEEE Computer Society.

[Hak85] Armin Haken. The intractability of resolution. *Theoretical Computer Science*, 39:297–308, 1985.

[Jac02] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):256–290, 2002.

[MMZ+01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proccedings of the 38th Design Automation Conference (DAC'01)*, 2001.

[MSP08] Joao Marques-Silva and Jordi Planes. Algorithms for maximum satisfiability using unsatisfiable cores. In *Proceedings of Design, Automation and Test in Europe (DATE 08)*, pages –, 2008.

[MSS99] Joao Marques-Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transaccions on Computer*, 48(5):506–521, may 1999.

[NOT06] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT Modulo Theories: from an Abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T). *Journal of the ACM*, xx(x):xx, 2006. "Accepted. To appear. Available from `www.lsi.upc.edu/~roberto`.".

[RV02] Alexandre Riazanov and Andrei Voronkov. The design and implementation of VAMPIRE. *AI Communications*, 15(91-110), 2002.

[ZM03] Lintao Zhang and Sharad Malik. Validating SAT Solvers Using an Independent Resolution-Based Checker: Practical Implementations and Other Applications. In *Proccedings of Design, Automation and Test in Europe Conference (DATE 2003)*, pages 10880–10885. IEEE Computer Society, 2003.

[ZMMM01] L. Zhang, C. F. Madigan, M. W. Moskewicz, and S. Malik. Efficient conflict driven learning in a Boolean satisfiability solver. In *International Conference on Computer-Aided Design (ICCAD'01)*, pages 279–285, 2001.