THE LOGIC IN COMPUTER SCIENCE COLUMN

BY

YURI GUREVICH

Microsoft Research
One Microsoft Way, Redmond WA 98052, USA
gurevich@microsoft.com

A SAT SOLVER PRIMER

David G. Mitchell[*]

Abstract

In not much more than a decade the practice of implementing programs for testing satisfiability of propositional formulas has progressed from being practically a curiosity to an active and competitive endeavor. Impressive performance improvements have made SAT solvers useful in some industrial applications, most notably in formal verification of hardware and software systems as the engine for bounded model checking (BMC) tools. Here we describe the main design and implementation features of the family of SAT solvers which represent the state of the art for solving of industrial instances, and includes the solvers Chaff, BerkMin and siege. We attempt to give a self-contained presentation in a uniform style, and in doing so hope to encourage readers to ask interesting questions, as well to tackle some problems we mention. It should also be possible to produce a creditable solver with little more at hand than this paper and a penchant for highly efficient implementation.

[*]Supported by NSERC. Address: Simon Fraser University, Burnaby, BC V5A 1S6, CANADA

Contents

1 Introduction

The implementation of programs for deciding satisfiability of formulas of propositional logic (SAT), once a bit of a novelty, is now a competitive activity, with widespread interest in academia and an important and growing base of industrial users. The 2004 SAT Solver Competition [26], held in association with the Seventh Annual Conference on Theory and Applications of Satisfiability Testing in May of 2004 [20], involved 55 solvers produced by 27 groups with authors in a dozen countries. The solvers were tested on a collection of 685 benchmark instances grouped in 77 source domains [1].

The method of representing an instance of a search problem as a propositional formula so that a satisfying assignment represents a solution, and then running a SAT solver to find such an assignment if there is one, has been found to be a practical and effective method for solving a number of problems. It has been used successfully in the electronic design automation (EDA) industry for a variety of tasks including microprocessor verification [43] and automated test generation [42] among many others [34]. Perhaps most notably, SAT-based bounded model checking [8] has become a widely used verification method, and these methods are being extended to un-bounded model checking [35]. SAT-based verification has been applied to software as well as hardware [10], and SAT solvers are used as computational engines in a variety of model checking tools such as NuSMV [11] and the analyzer for the Alloy software modeling language [22]. The AI planning tool BlackBOX [25], won the optimal plan category of the 2004 planning competition [21] powered by the SAT solver siege [40]. SAT solvers, or modifications of them, are used as the engines for tools using more expressive logics, including for problems that we expect are not in NP, such as answer set programming [28, 29], causal logics [16], quantified boolean formulas and modal logics [15], and even Presburger formulas [41] and restricted first order theorem proving [18].

This is in marked contrast with the situation a decade and a half ago, when the fact that every problem in NP can be reduced to SAT in polynomial time was widely regarded as being of only theoretical interest. Conventional wisdom was that a general purpose algorithm could not be expected to perform well on NP-hard search or decision problems: Such problems presumably require exponential time in the worst case, so any algorithm that would be effective in practice would have to take advantage of the particular structural properties of instances at hand, and thus would have to be carefully tailored by someone intimately familiar with the application and instances. The other side of this coin is that in the abstract framework good general heuristics can be more easily found, clarified and efficiently implemented. The best possible performance on a distribution of instances will certainly be obtained by an algorithm tuned to that distribution, and indeed the solvers we discuss have been tuned for their most significant application. Still, they generally work well on a wide variety of application instances, and their performance as general purpose solvers is often impressive. (This claim must be tempered by the observation that there are families of formulas for which they are not well suited, for example they are not the best for random formulas, or for other formulas constructed to require long resolution proofs [26].)

This change in status is primarily the result dramatically increased performance. In 1989, the first year in which this writer implemented a SAT solver, we were impressed at being able to solve formulas of a few hundred clauses on a research workstation. Today, a generic home PC running software down-loadable for free from the web can easily solve many industrial instances with hundreds of thousands of clauses. Certainly work on how to effectively represent problems as formulas was essential, but without the raw speed gains would not have paid off. Arguably the performance increase is due to three factors; improved algorithms, improved implementation techniques, and increased machine capacity. Each of these has individually contributed multiple orders of magnitude in speedup. Increases in memory size and speed allowed the introduction of so-called "clause learning" techniques which require storage of tens or hundreds of megabytes which must be accessed very efficiently. Increases in processor speed allow much larger searches, which in turn makes more computationally expensive heuristics pay off in terms of reduced search time.

Here we give an account of the elements that go into making the best current publicly known SAT solvers. We do not attempt a survey of SAT solver technologies, but rather try to give a reasonably detailed and self-contained description of a particular family of solvers, which includes Chaff [36, 46] BerkMin [14] and siege [40] among others. In taking this narrow approach, we necessarily ignore a great deal of interesting and important work in design and development of SAT solvers and their applications, including all complete methods that do not fit into our narrow mold and all randomized local search methods.

The paper is organized as follows. In Section 2 we introduce backtracking, its relation to resolution proofs, and the DPLL procedure, and in section 3 we cover so-called clause learning and conflict-directed backjumping. Section 4 addresses unit propagation in detail. We present decision heuristics in Section 5, restarts in Section 6 and pre-processing briefly in Section 7. Finally, Section 8 mentions a few problems.

2 Backtracking, Resolution and DPLL

Most solvers take their input in conjunctive normal form (CNF), and henceforth by "formula" we mean one in this form: a conjunction of disjunctions of literals, or equivalently a set of clauses each of which is a set of literals, a literal being a propositional atom p or its negation $\neg p$. We assume the literals in a clause all involve distinct variables. We use ϕ to denote a formula, and α an assignment – possibly partial – to the variables of ϕ. We take α to be a sequence of literals, containing just those literals which are mapped to *true*, in chronological order as the values were assigned by the algorithm at hand. For literal l and assignment α, we denote the extention of α by l as αl. For formula ϕ and (partial) assignment α, we denote by $\phi|\alpha$ the formula obtained by deleting from ϕ every clause that is satisfied by α, and deleting from each clause of ϕ every literal that α makes false. We call this a "restricted" formula.

Most modern solvers aimed at industrial instance solving are based on the basic backtracking algorithm – albeit with important improvements.

```
procedure BT( $\phi$, $\alpha$ )
    (SAT)           if $\phi|\alpha$ is empty return SATISFIABLE
    (Conflict)      if $\phi|\alpha$ contains an empty clause return UNSATISFIABLE
```

```
(Branch)    Otherwise, let $p$ be a literal of $\phi|\alpha$
            if BT( $\phi$, $\alpha p$ ) returns SATISFIABLE
                return SATISFIABLE
            else
                return BT( $\phi$, $\alpha\neg p$ )
```

To check if ϕ is satisfiable, we execute BT(ϕ,$\emptyset$). The procedure tries to extend α to a satisfying assignment, terminating if one is found and backtracking if a conflict condition is reached. When a conflict occurs, corresponding to the empty clause in $\phi|\alpha$ is a clause C of ϕ such that α makes every literal of C false. We call C a conflicting clause.

The search tree for execution of BT(ϕ,$\emptyset$) has a node for each call to BT and edges from each node to the two nodes representing the recursive calls. The root corresponds to BT(ϕ,$\emptyset$) and leaves correspond to calls which return without recursion. Each internal node has an associated "branching variable" x, and the edges to its children are labeled with x and $\neg x$, so the sequence of labels on the path from the root to node v is exactly the assignment α passed to the invocation of BT corresponding to v.

The backtracking algorithm and refinements of it used in the solvers we consider can be viewed as constructing resolution proofs. The propositional resolution rule allows us to derive the clause $(C \vee D)$ from two clauses $(x \vee C)$ and $(\neg x \vee D)$. We say we resolve them on x, and that $(C \vee D)$ is the resolvent. A resolution derivation of C from ϕ is a sequence of clauses $\{C_1, \ldots, C_s = C\}$ in which each C_i is either a clause of ϕ or is the resolvent of some C_j and C_k with $j, k < i$. A derivation of the empty clause () from ϕ is called a refutation of ϕ, and ϕ is unsatisfiable if and only if it has a refutation.

The graph of derivation π of C from ϕ is a directed acyclic graph with clauses of π as nodes. Each internal node is a derived clause and has edges to the two clauses from which it was derived. C is a source, and sinks are clauses of ϕ. A derivation is called tree-like if its graph is a tree, or equivalently if any derived clause is used at most once to derive further clauses. It is called regular if on any path from C to a leaf no variable is resolved on more than once.

We may label the search tree for the execution of BT on an unsatisfiable formula ϕ with clauses as follows. (If ϕ is satisfiable, the tree can be so labeled with the exception of the last branch.) Each leaf is labeled with a clause that is made false by the assignment on the path to that leaf. For an internal node v, let C and D be the two clauses labeling its children, and x the branching variable. If C and D both mention the variable x, then one contains x and the other $\neg x$, and we label v with the resolvent of C and D. Otherwise, we label v with one of C or D which does not mention x. In this scheme, the clause labeling each node is made false by the the partial assignment on the path to that node, and in particular the root node is labeled with the empty clause. It is easy to check that the clauses labeling this tree form a regular tree-like resolution refutation of ϕ. Moreover, any regular tree-like refutation which is minimal, in that its graph is connected, can be constructed by executing BT with suitable choices of branching variables.

Davis, Putnam, Logemann and Loveland [13, 12] proposed a refined version of backtracking with a few simple but effective heuristics. A unit clause is a clause of size one, and a literal p is pure in ϕ if there is no occurrence of its negation $\neg p$ in ϕ. The algorithm is:

```
procedure DPLL( $\phi$, $\alpha$ )
    (SAT)            if $\phi|\alpha$ is empty return SATISFIABLE
    (Conflict)       if $\phi|\alpha$ contains an empty clause return UNSATISFIABLE
    (Unit Clause)    if $\phi|\alpha$ contains a unit clause $\{p\}$, return DPLL( $\phi,\alpha p$)
    (Pure Literal)   if $\phi|\alpha$ has a pure literal $p$ return DPLL( $\phi,\alpha p$)
    (Branch)         Let $p$ be a literal from a minimum size clause of $\phi|\alpha$
                     if DPLL( $\phi$, $\alpha p$ ) returns SATISFIABLE
                          return SATISFIABLE
                     else
                          return  DPLL( $\phi$, $\alpha\neg p$ )
```

Some authors use the term DPLL only for this exact algorithm, while others use it to refer to any refinement of the backtracking algorithm. This algorithm is also often called the Davis-Putnam algorithm, which ignores the contribution of two authors of [12] where it was presented.

The Pure Literal rule is omitted in most solvers, being considered too computationally expensive for the benefit obtained, especially since the combination of backjumping and decision heuristics in current algorithms is likely to simply ignore pure literals. The branching heuristic of selecting a variable from a short clause was influential, and branching heuristics in DPLL-based solvers continued to be refinements of this strategy until a very different approach was found more useful in clause learning solvers, as discussed in Section 5.

The Unit Clause rule is logically equivalent to always choosing a branching variable in a unit clause if there is one, since one of the recursive calls on such a variable will always return immediately without further recursion, but efficient execution of the rule is so important to good performance that implementers never think of it this way. It may also be seen as an operation that takes pair (ϕ,α) to $(\phi,\alpha p)$ when there is a clause $C \in \phi$ for which $C|\alpha = (p)$. Unit Propagation (UP)[1]

[1] We prefer “unit propagation” to the other terms sometimes used, "unit resolution" and "boolean constraint propagation" (BCP), as the former is a misnomer, and the latter is sometimes used in a more general setting.

is the repeated application of this operation until either a conflict is reached (α makes some clause of ϕ false) or a fixed point is reached without conflict. Notice that conflicts are *only* reached during unit propagation. Section 4 addresses the importance and implementation of UP.

3 CDCL: Adding Backjumping and Learning

The first SAT solver competition of which the author is aware was held in 1992 at the University of Paderborn [9]. One of the conclusions of that competition was that the best solvers were DPLL solvers. This algorithm remained dominant among complete methods until the introduction of so-called clause learning solvers in 1996 [6, 33]. This new method is a variation on DPLL suggested two observations about the backtracking algorithm and corresponding refutation. The first is that, if we actually derive the clauses labeling the search tree, we can add some of them to the formula. If later in the execution the assignment at some node falsifies one of these clauses, the search below that node is avoided with possible time savings. This technique is variously called “clause learning” and “clause recording” or just “learning”.

The second observation is that at a node v with branching variable x, we first make one recursive call during which we derive a clause C and then make a second recursive call during which a clause D is derived. C and D are resolved to produce the clause to label v. However, if the clause C does not mention the variable x, there is no need to make the second recursive call, since the clause C suffices to label v. In some cases this may save a great deal of work. This is the propositional version of the method called “conflict directed back-jumping” (CBJ) in the constraint satisfaction literature [39], from where the idea came. The version used in most SAT solvers is a bit smarter than this, as described below.

3.1 The CDCL algorithm

The algorithm used in GRASP, Chaff, BerkMin, siege and many other recent solvers is sometimes called “conflict driven clause learning” (CDCL). It is possible to describe recursively, but the following iterative version is clearer, and better reflects how it is implemented.

We categorize assigned variables as decision variables or propagation variables. The decision level of variable x is the number of decision variables in the assignment α that were assigned no later than x (i.e., do not come after x in the sequence α). Assignments made by unit propagation before any decision variable is assigned are at level zero. We call the level at which a conflict occurs the conflict level. The algorithm is:

```
procedure CDCL( $\phi$ )
    $\Gamma = \phi$,  $\alpha = \emptyset$,  $level = 0$
    repeat:
        execute UP on $(\Gamma,\alpha)$
        if a conflict was reached
            if $level = 0$ return UNSATISFIABLE
            $C$ = the derived conflict clause
            $p$ = the sole literal of $C$ set at the conflict level
            $level = \max\{ \text{level}(x) : x \in C - p\}$
            $\alpha = \alpha$ less all assignments made at levels greater than $level$
            $( \Gamma, \alpha ) = ( \Gamma \cup \{C\}, \alpha p )$
        else
            if $\alpha$ is total return SATISFIABLE
            choose a decision literal $p$ occurring in $\Gamma|\alpha$
            $\alpha = \alpha p$
            increment $level$
        end if
```

Assuming that the initial unit propagation does not reach a conflict, the algorithm repeats the steps of choosing a decision literal to set and then executing unit propagation, until either the assignment is total and satisfies ϕ, or a conflict is reached. When a conflict is reached, a "conflict clause" is derived and is added to the formula. This clause must be made false by α, must be logically implied by ϕ and must be a so-called "asserting clause", which means that it contains exactly one literal p whose value was assigned at the conflict level. Backtracking is carried out by deleting from α all literals assigned at levels greater than the greatest level of a literal in $C\backslash p$. Further details are given in the next subsection. This algorithm is also sometimes described as the "failure driven assertion" (FDA) loop, the failure being our conflict, and the assertion being the literal p.

Correctness of CDCL is less obvious than that of BT and DPLL. Certainly if CDCL returns SATISFIABLE, α satisfies ϕ. If CDCL returns UNSATISFIABLE, there was a conflict at level zero. Since every variable set at level zero was the result of unit propagation from ϕ or deriving a unit conflict clause, these values are logically implied by ϕ, so ϕ must be unsatisfiable. It remains to prove termination, which we do along the lines used in [48]. Let $\alpha[i]$ denote the number of variables in assignment α that were set at level i. Let $\prec$ be an ordering on partial assignments for ϕ, such that $\alpha \prec \beta$ if and only if for some i, $\alpha[i] \prec \beta[i]$ and for all $j < i$, $\alpha[j] = \beta[j]$. The minimum assignment is $\emptyset$, every assignment satisfies $\sum_i \alpha[i] \leq n$, and the maximum assignment has $\alpha[0] = n$. Initially $\alpha = \emptyset$, so it is enough to verify that the assignments at the top of the CDCL loop are monotonically increasing according to this order. Clearly $\alpha \prec \alpha p$, so UP and the else condition satisfy this. For the if condition, we remove values from α and then immediately extend it with p at the new last level, which increases α on the ordering.

3.2 Deriving the Conflict Clause

We begin by describing a particular choice of conflict clause (called the First UIP clause), and then discuss some variations. To help in constructing the clause we extend the information stored in the assignment α to include, for each assigned literal, a "reason" for the assignment. This reason is either the fact that it was a decision literal or is the index of the clause which became unit and forced the literal to be assigned true during unit propagation. Now suppose UP reaches a conflict with partial assignment α and conflicting clause C. The clause C must be of the form $\{p \vee q \vee C'\}$, where $\neg p$ and $\neg q$ were added to α at the conflict level, since otherwise C would have been a unit clause at the previous decision level. Assume p was assigned after q. We derive a sequence of resolvents C_i as follows. The reason for p must be a clause $B_1 = \{\neg p \vee r \vee B_1'\}$ (where r could possibly be the same as q). We resolve C and B_1 to obtain $C_1 = \{q \vee r \vee C' \vee B_1'\}$. If C_i contains only one literal whose value was set at the conflict decision level, we stop and let C_i be our conflict clause. Otherwise we select the literal $l \in C_i$ whose value was set last, let B_{i+1} be the reason for the assignment to l, and resolve C_i with B_{i+1} to obtain C_{i+1}. The process must terminate, as eventually all literals of the current level will be resolved out except for the decision literal. It is often the case, though, that the one conflict level literal remaining was set by UP.

Conflict clause construction is often described in terms of the implication graph [32] in which nodes are literals of α, and there is a directed edge from u to v if $\neg u$ occurs in the reason clause for v. Decision literals have in-degree zero. If q was the last literal set false in the conflicting clause C, we include q as well as $\neg q$ in the graph, and we consider C to be the reason for q. We call q and $\neg q$ conflict literals. Now consider a cut in this graph which has the conflict literals on one side, called the conflict side, and all decision literals on the other side, called the reason side, and such that every node on the conflict side has a path to a conflict literal. If we take the set of literals on the reason side which have at least one edge to the conflict side, we obtain a clause which can be derived by the process described above – though possibly choosing literals to resolve out in a different order, and possibly using a different termination condition. The implication graph and its relationship to resolution derivations is most clearly described in [7].

Suppose that p is the conflict level decision literal. A vertex u through which every path from p to the conflict literals passes is called a unique implication point (UIP). The UIP nearest to the conflict literals is called the First UIP. The cut which has the First UIP on the reason side and all literals assigned after it on the conflict side gives the conflict clause called the First UIP conflict clause. This is also the clause produced by the resolution process described above, and is the one used by siege and by early versions of Chaff. Other choices of conflict clause that have been tried include, for example:

RelSAT scheme Use the derivation process described, but continue until the only remaining conflict level literal is the decision literal (i.e., derive the clause corresponding to the last UIP).

Decision Scheme Derive the RelSAT clause, and then continue the process of resolving out literals by resolving out all literals set at other levels until only decision literals remain.

All-UIP Scheme Proceed as in the Decision Scheme, but for each level stop resolving out literals at that level the first time there is only one literal from the level remaining.

Zhang et al [46] investigated the performance of these and other schemes in the Chaff solver, and found that the FirstUIP clause is the most useful single clause to record. GRASP and recent versions of Chaff record more than one conflict clause, although it is probably too early so say which combinations are best.

This conflict derivation process makes it clear that in implementation of UP, not only speed but the exact order of propagation steps may matter, since this affects which conflict clauses are derived and stored. We remark further on this in Section 4.

3.3 Backjumping

Having derived our asserting clause C, we implement backjumping as follows. Assuming C has at least two literals, let p be the unique literal in C assigned at the conflict level, q be the last literal assigned other than p, and l the level at which q was assigned. Since C is our conflict clause we add it to our clause set. We revise our assignment α by removing all assignments made at decision levels greater than l. Notice that this new assignment makes all literals in C false except for p. Thus, we can simply pick-up unit propagation where it finished off after the level l decision, immediately setting p as a propagation step, and possibly continuing with further propagation. If $C = \{p\}$ has only one literal, then backjumping is to just before the start of level one, so the value p is set at level zero and can never be changed again.

3.4 Clause Deletion

The number of clauses derived and cached by the procedure just described is, on large hard formulas, far too large to keep them all indefinitely. Even if sufficient memory is available for storage, the time to manage the clauses – in particular the time to execute unit propagation – becomes impractical, and ultimately reduces performance. All effective solvers that implement clause learning also implement a clause “forgetting” strategy, to keep the size of the clause store reasonable. Generally speaking, the strategy is to periodically delete all those learned clauses which are very large and which have not been used recently or frequently in deriving new clauses. Siege periodically deletes a random selection of clauses that are larger than some threshold size. In Chaff and siege very large clauses are deleted when backtracking leaves them with many (say 100 or 200) literals unassigned, at which point it seems likely they will never get used.

4 Unit Propagation (a.k.a. BCP)

In competitive solvers approximately 80 to 95 per cent of execution time is spend performing unit propagation (UP), so it is essential that it be very efficiently implemented. The main work is to identify clauses which have effectively become unit clauses, because all but one of their literals have been set false. Under any scheme designed so far this requires visiting many clauses each time a variable is set.

To implement unit propagation, we keep a queue of variable assignments which have been set, but which have not yet been propagated. At each stage, we take an assignment p off of the queue, and check whether extending the current assignment with p produces any additional unit clauses. In the very first execution of unit propagation, at level zero, the queue initially contains all unit clauses of the input formula. At the start of each decision level, the queue is initialized with the decision literal for that level. After each backjump the queue is initialized with the assertion literal.

4.1 Watched Literals

Early designs typically had two counters for each clause which kept track of the number of true and false literals. These were used to detect when a clause became unit (or satisfied). This required visiting every clause mentioning variable x whenever x was assigned a value. The Chaff solver introduced the two-watched-literal scheme [36], a variant of an earlier scheme used in the solver Sato [45], to reduce the number of clause visits. Under this scheme, two non-false literals are chosen to watch in each clause. A clause could only become unit if one of these is set false, so we need only visit a clause when one of its watched literals is set false. When a literal p is assigned true, we visit each clause in which $\neg p$ is a watched literal, and search its literals for another that has not been set false to use as a watch. If there is none, we inspect the other watched literal for the clause. If it is assigned true, the clause is satisfied and we ignore it; if it is false we have reached a conflict; if it is unassigned the clause is now a unit clause, and we add that literal to the UP queue.

The scheme has two additional benefits; we do not need to keep track of satisfied clauses, and backtracking is essentially free. When we backtrack we remove assigned values in reverse of the order they were assigned, so any watched literal before backtracking remains a valid literal to watch after backtracking. Therefore, to backtrack the only work is to change the location of the stack pointer on the assignment. In contrast, most other schemes require doing work, such as revising counter values, when backtracking.

4.2 Cache-aware implementation

The introduction in Chaff of cache-aware implementation is as important as the introduction of the watched literal scheme. When solving challenging formulas the size of the clause store may easily grow to tens or hundreds of thousands of clauses, many of which contain hundreds of literals. A current i86-family processor has 1MB of L2 cache, so only a fraction of these clauses can be in the cache at any time. Each time part of a clause is accessed that is not in the L2 cache – a cache miss – it must be read from main memory. The time penalty for this is large: 50 to 250 instructions to read from main memory, versus 5 to 10 for reading from L2 cache (and one or two instructions to read from L1 cache).

A cache aware implementation attempts to minimize cache misses by, for example, reducing the memory footprint and preferring arrays to pointer-based data structures. Whenever possible data is stored so that sequences of memory accesses are likely to involve contiguous memory locations. Zhang and Malik [47] report experiments that show that cache-aware solvers like Chaff and BerkMin have a speedup on the order of a factor of three over earlier designs like RelSAT, GRASP and Sato purely due to number of cache misses. The difference between performance for the same algorithm using counter-based unit propagation versus the two-watched-literal scheme was on average a factor of eight, which must partly be due to number of clause visits and partly to attendant cache misses.

Current implementations store the clauses in one array as a sequence of literals with sentinel values delimiting clauses. More complex data structures have been tested, including tries [44] and ZBDDS [2]. Effects of cache misses may partly explain why these have not been widely adopted, despite some advantages.

The cache miss rate is further reduced in siege [40] by using 21 bits for each literal (including one bit to flag watched literals), rather than the usual 32. This allows three literals per 64-bit word rather than two, and so a much reduced memory footprint. Siege has the fastest unit propagation of any solver of which we are aware, but this has to be weighed against the resulting limitation to formulas with $2^{20} = 524,288$ variables. Few current public benchmarks have this many variables, but it is an unreasonable limitation for many industrial users.

4.3 Handling Short Clauses

Pilarski and Hu [37, 38] observe that hardware verification instances typically have a large number of binary (size 2) clauses, and that it makes little sense to execute the standard unit propagation algorithm for them. Fractions of binary clauses range from 55 to 90 per cent in various benchmark collections. One need only keep a list for each literal p of all literals q for which there is a binary clause $(\neg p \vee q)$, and scan this list upon assigning p true. This reduces memory footprint as well as number of operations to execute. Ryan [40] extends the idea to special data structures for clauses of size three as follows. For each literal appearing in a ternary clause, there is a list of all pairs of literals occurring together with it in ternary clauses. There are three such pairs, on three different lists, for each ternary clause. Each of these pairs has a pointer to the other two pairs representing this clause. For algorithm details see [40].

Propagation in these structures is much faster than in the scheme for general clauses, so it makes sense to maximize their use. In siege, propagation always is done first through the binary clauses. When this is complete, propagation through ternary clauses begins, but each time a unit clause is found and a literal set true, propagation is done again through the binary clauses before continuing the process of propagating through ternary clauses. Finally, after ternary clause propagation is complete, propagation begins through longer clauses, once again with a return to the short clause propagation process each time a variable is assigned.

4.4 Resolution Strategy

At any point during unit propagation there may be more than one unit clause and thus a choice of which order to assign literals and propagate further. For simplifying the formula and detecting existence of a conflict only speed matters and this order is of little concern. However, the particular order chosen does affect which conflicting clause is identified, and which clauses are resolved against it to produce the conflict clause. This has received little attention so far in the literature but is almost certainly important.

The strategy described in the previous subsection for propagation through short clauses amounts also to a strategy for deriving conflict clauses. Roughly, we give preference to resolving the current clause against binary or ternary clauses when possible. This produces a distinct pattern in derivations of conflict clauses in siege. In most cases each such derivation involves only one or two long clauses, but a long sequence of steps resolving these with short – mostly binary – clauses. Typically the resulting conflict clause is not much longer than the original long clause in the sequence.

We may view this as carrying out a more general strategy to allow the size of derived clauses to grow, but only slowly. We know by results on resolution proof complexity that hard-to-refute formulas require deriving long clauses, so our solvers must allow derivation of such clauses. On the other hand, long clauses are hard to manage and, because there are many of them, a priori any particular long clause is likely not to be useful. Therefore, it makes sense to allow clause length to grow only strategically.

We described unit propagation above using a breadth-first search scheme by keeping a queue of assignments for propagating. Ryan [40] reports that a depth first scheme has a slightly more efficient implementation, but results in slightly poorer performance. Presumably this is because the depth first scheme is more likely to involve longer sequences, and thus to produce larger derived clauses.

5 Decision Heuristics

DPLL branching heuristics have been the subject of numerous studies. The most successful are based roughly on making choices that make the resulting restricted formulas as "simple" as possible (see, e.g., [19]). The original DPLL scheme of choosing a variable in a minimum length clause is an inexpensive-to-compute version of this, as it will tend to result in unit propagation and thus reduce the number of variables and clauses in the formula. As machine speeds increased larger and harder instances could be solved and the tradeoff between time spent choosing branching variables and time saved by a good choice shifted in favour of more complex heuristics. Some representative examples, of increasing complexity, include:

MOMS Choose the literal with the Maximum number of Occurrences in Minimum Size clauses.

2-sided Jerowslow-Wang [23] Let $\phi[l]$ be the set of clauses of ϕ that contain literal l. Define a score $jw(l)$ for literals of ϕ by $jw(l) = \sum_{C \in \phi[l]} 2^{-|C|}$. Choose the variable x with maximum $jw(x) + jw(\neg x)$.

Satz One of the best DPLL solvers produced is Satz [27]. It uses MOMS to select a set of promising variables, and for each literal l for these assigns l and carries out unit propagation. Literals are scored according to the size of the resulting formula. The size of the subset of selected variables depends on the level of the current search node, with all variables tried near the root as decisions at these levels are the most important.

Marques-Silva [31] compared use of several DPLL heuristics in the early CDCL solver GRASP [33], and found that none was a clear winner. More significantly, making a random choice was almost as good as the other strategies, strongly suggesting that DPLL-type heuristics are inappropriate for CDCL. The decision strategies which have been found to work in CDCL solvers appear to emphasize a kind of locality rather than formula simplification, favouring variables which have appeared in recently derived conflict clauses.

VMTF Introduced in [40], Variable Move To Front (VMTF) is easy to implement and very cheap to compute. It often performs better than the more expensive VSIDS heuristic introduced in Chaff, and almost as well as the much more complex heuristics used in BerkMin and recent versions of Chaff. The variables are stored in a list, initially ordered by non-increasing frequency of occurrence in the input formula. An occurrence count $\mathrm{c}(l)$ is maintained for every literal l. Each time a conflict clause C is derived occurrence counts for literals of C are incremented and some number of variables occurring in C, for example $\min(|C|, 8)$, are moved to the front of the list. When a decision literal must be chosen, the first unassigned variable v on the list is selected. If $\mathrm{c}(v)>\mathrm{c}(\neg v)$, v is set true; if $\mathrm{c}(v)>\mathrm{c}(\neg v)$, v is set false; ties are broken randomly.

VMTF chooses a variable from a recently derived clause (but not the last conflict clause, as its literals are all assigned), making it likely that the next conflict clause contains a variable occurring in a recent conflict. The choice of sign maximizes the number of existing clauses which have v with the opposite sign, and thus could be resolved against it.

VSIDS The first heuristic suited to CDCL solvers was VSIDS (Variable State Independent Decaying Sum), introduced in Chaff [36]. For each literal l, keep a score $\mathrm{s}(l)$ which initially is the number of occurrences of l in ϕ. Each time a conflict clause with l is added, increment $\mathrm{s}(l)$. Periodically (every 255 decisions) re-compute all literal scores as $\mathrm{s}(l) = \mathrm{r}(l) + \mathrm{s}(l)/2$, where $\mathrm{r}(l)$ is the number of occurrences of l in a conflict clause since the previous update. To choose a decision literal, pick the unassigned literal with the highest score. Dividing the scores provides an aging mechanism which emphasizes variables in relatively recently derived clauses. It is worth noting that computing VSIDS requires about 10% of

Chaff running time, whereas computing VMTF requires about 1% of siege running time.

BerkMin The strategy in the BerkMin solver [14] is approximately as follows. Each variable has a score, which is periodically aged as in VSIDS, except that division is by 4 rather than 2 to increase the recency emphasis. Scores are incremented for all variables used in the conflict clause derivation, not just those in the conflict clause. To choose a decision literal, let C be the most recently added clause that is not satisfied by the current assignment. If C is an input clause, pick the variable with highest score, choosing the sign based on an estimate of which sign will produce the most unit propagation. If C is a conflict clause, let l be the unassigned literal of C with the highest score. Let n_l be the total number of occurrences of l in conflict clauses so far, and $n_{\neg l}$ similarly for $\neg l$. Set l true if $n_l > n_{\neg l}$; set l false if $n_l < n_{\neg l}$; break ties randomly.

Chaff 2004 The most recent versions of Chaff [30], use a complex heuristic that combines a BerkMin-like method with a version of VSIDS plus a scheme for deleting part of the current assignment when a newly derived conflict clause is very large, as a means of trying to keep conflict clause sizes small.

In general, the development of CDCL heuristics so far has seen increased performance as a consequence of increased emphasis on literals in recently derived conflict clauses as a first order heuristic, and keeping conflict clause size moderate as a second order heuristic.

6 Restarts: Robustness *vs* Completeness

Solvers often take dramatically different running times on very similar instances, and even over instances which are identical other than re-ordering clauses or literals within clauses, or re-naming variables. Often one instance among such a set takes unreasonably long while the rest are manageable, but that one instance is solved in reasonable time if re-ordered. The intuitive explanation is along the lines that even for some relatively easy instances certain orders of search may take the algorithm into parts of the search space that do not produce useful conflict clauses, leaving it “floundering”.

Restarts were proposed in [17] as an approach to dealing with high variance in running times over similar instances. A restart is the operation of throwing away the current partial assignment (excluding assignments at decision level zero), and starting the search process from scratch. In CDCL the new search will normally be different, because the cache of learned clauses is retained. This cache is typically much larger than the input formula so after restarts these clauses dominate the choice of decision variables. Chaff, BerkMin and siege all implement frequent restart policies: BerkMin restarts after every 550 conflicts, siege after every 16,000 conflicts.

The effects of restart policies are not very well studied. A preliminary report is given in [5]. A more principled approach is taken in [24], but the experiments there use solvers that do not implement clause learning. Roughly speaking, using restarts only provides a modest improvement in typical performance, but substantially improves robustness. That is, they significantly reduce the variability in running time found over collections of similar instances, often allowing a solver to handle a compete set of benchmark instances where the version without restarts fails on some small number of them.

Fixed-interval restart policies such as in BerkMin, siege and recent versions of Chaff, together with clause deletion, make these solvers incomplete because there is no mechanism to avoid repeating the same search over. Completeness can be assured by retaining all learned clauses, but a practical solver cannot possibly do this. Some solvers, such as GRASP and some versions of Chaff, retain completeness by gradually increasing the restart interval, so that eventually a search must complete with no restart. The utility of this is not clear, as the running time to reach this condition would generally be too large to be interesting.

Although proving unsatisfiability is very important, completeness *per se* is not. The reason is that all complete solvers will fail to terminate in reasonable time on many instances, and it does not matter if on some of these instances the solver would never halt. What does matter is that a solver halt in reasonable time on as many currently interesting instances as possible, and adding restarts typically improves performance by this measure.

7 Pre-Processing

The idea of pre-processing the input formula before running the main algorithm occurs regularly in the literature. The most usual ideas are to add certain derived clauses or to apply transformations that reduce the number of variables or clauses. The most successful example is that of Bacchus and Winter [3], based on a derivation rule HypBinRes. The rule permits deriving a binary clause $(p \vee l_n)$ from an n-ary clause $(l_1 \vee \dots l_n)$ and $n-1$ binary clauses $(p \vee \neg l_1), \dots (p \vee \neg l_{n-1})$. The primary work of the preprocessor is to compute a modified closure of the formula under this operation (importantly, without explicitly computing the closure under binary resolution, which often makes formulas too large to solve). Using the preprocessor together with BerkMin and Chaff in many cases reduced running times or allowed them to solve instances they could not without it. The preprocessor alone was also able to solve some instances that the solvers cannot handle.

That CDCL solvers add many more derived clauses than they start with suggests that executing something like this pre-processing step after deriving some clauses might be valuable. A CDCL solver that executed HypBinRes at every search node [4] performed well over a wide range of problems in the 2002 SAT Solver Competition, but the overhead of performing this computation so often made it uncompetitive overall with the best solvers. However, executing it at infrequent intervals, such as when clause deletion is done, seems likely to pay off.

8 Problems

1. Determine the power of CDCL as a proof system. We know that DPLL has the same power as tree-like resolution, and CDCL is no more powerful than unrestricted resolution. Beame et al [7] have studied the power of CDCL as a proof system, and shown that it is more powerful than regular and Davis Putnam resolution, which are already known to be stronger than DPLL. However, the question of whether it is as strong as general resolution remains open.

2. Design a data structure that supports an efficient unit propagation algorithm, while reducing cache misses. Using additional memory is acceptable. The amortized cost of UP must remain close to linear in the number of conflicts found.

3. Characterize classes of instances that CDCL solves well. Since resolution can simulate CDCL, any family of formulas hard for resolution will be hard for CDCL. Is there a CDCL strategy that efficiently handles instances with bounded tree width or other structural properties that guarantee existence of short refutations?

4. Identify a proof system more powerful than resolution which can be effectively used within the CDCL algorithm scheme. There are many formula families which require exponential size resolution refutations, and which must therefore require exponential time of DPLL and CDCL. It would be interesting as well as useful to have a practical algorithm not subject to these bounds. The branching nature of DPLL makes it hard to naturally take advantage of more powerful reasoning steps. However, CDCL seems somewhat more flexible. Any efficient scheme for propagation of values and derivation of conflict clauses can be used in place of UP and the conflict derivation scheme described, provided it produces a conflict “clause” (not necessarily a clause) satisfying the conditions mentioned in Section 3.1.

References

[1] 2004 SAT Solver Competition. http://www.lri.fr/~simon/contest/results/.

[2] F. Aloul, M. Mneimneh, and K. Sakallah. Backtrack search using ZBDDs. In *Proc., International Workshop on Logic Synthesis (IWLS'01)*, 2001.

[3] F. Bacchus and J. Winter. Effective preprocessing with hyper-resolution and equality reduction. In *Proc., Sixth International Symposium on Theory and Applications of Satisfiability Testing 2003*, pages 341–355, 2003.

[4] Fahiem Bacchus. Enhancing Davis Putnam with extended binary clause reasoning. In *Proceedings, AAAI 2002*, pages 613–619, 2002.

[5] L. Baptista and J.P. Marques-Silva. Using randomization and learning to solve hard real-world instances of satisfiability. In *Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming (CP'00)*, September 2000.

[6] Roberto J. Bayardo Jr. and Robert C. Schrag. Using CSP look-back techniques to solve exceptionally hard SAT instances. In *Proc., CP-96*, 1996.

[7] Paul Beame, Henry Kautz, and Ashish Sabharwal. Towards understanding and harnessing the potential of clause learning. *Journal of Artificial Intelligence Research*, 22:319–351, 2004.

[8] A. Biere, A. Cimatti, E.M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proc., Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99)*, 1999. LNCS Volume 1579.

[9] M. Buro and H.K. Büning. Report on a SAT competition. Technical Report 110, Universität Paderborn, 1992.

[10] Sagar Chaki, Edmund M. Clarke, Alex Groce, Somesh Jha, and Helmut Veith. Modular verification of software components in C. In *Proc., ICSE 2003*, pages 385–395, 2003.

[11] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *Proc., International Conference on Computer-Aided Verification (CAV 2002)*, 2002.

[12] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5:394–397, 1962.

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

[14] E.Goldberg and Y.Novikov. BerkMin: a fast and robust SAT-solver. In *Proc., DATE-2002*, pages 142–149, 2002.

[15] E. Giunchiglia, F. Giunchiglia, and A. Tacchella. SAT-based decision procedures for classical modal logics. *Journal of Automated Reasoning*, 24, 2000.

[16] Enrico Giunchiglia, Joohyung Lee, Vladimir Lifschitz, Norman McCain, and Hudson Turner. Nonmonotonic causal theories. *Artificial Intelligence*, 153:49–104, 2004.

[17] C.P. Gomes, B. Selman, and H. Kautz. Boosting combinatorial search through randomization. In *Proc. of the National Conference on Artificial Intelligence (AAAI-98)*, July 1998.

[18] GrAnDe (ground and decide). `http://www.cs.miami.edu/~tptp/ATPSystems/GrAnDe/`.

[19] J.N. Hooker and V. Vinay. Branching rules for satisfiability. *Journal of Automated Reasoning*, 15:359–383, 1995.

[20] Holger H. Hoos and David G. Mitchell (eds). *Selected Papers from SAT 2004*. Springer (LNCS), 2005. To appear.

[21] ICAPS 2004 Planning Competition. `http://www-rcf.usc.edu/~skoenig/icaps/icaps04/planningcompetition.html`.

[22] Daniel Jackson, Ilya Shlyakhter, and Manu Sridharan. A micromodularity mechanism. In *Foundations of Software Engineering, Proceedings of the 8th European software engineering conference*, 2001.

[23] Robert E. Jeroslow and Jinchang Wang. Solving propositional satisfiability problems. *Annals of Mathematics and Artificial Intelligence*, 1:167–187, 1990.

[24] H. Kautz, E. Horvitz, Y. Ruan, C. Gomes, and B. Selman. Dynamic restart policies. In *Proceedings AAAI-2002*, 2002.

[25] Henry Kautz and Bart Selman. Unifying sat-based and graph-based planning. In *Proc. IJCAI-99*, 1999.

[26] Daniel Le Berre and Laurent Simon. Fifty-five solvers in Vancouver: The SAT 2004 competition. In *Selected Papers from SAT 2004*. Springer (LNCS), 2005. To appear.

[27] Chu Min Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *Proceedings, IJCAI'97*, pages 366–371, 1997.

[28] Y. Lierler and M. Maratea. Cmodels-2: SAT-based answer sets solver enhanced to non-tight programs. In *Proc., LPNMR-7 2004*, 2004.

[29] Fangzhen Lin and Yuting Zhao. ASSAT: computing answer sets of a logic program by SAT solvers. *Artificial Intelligence*, 157:115–137, 2004.

[30] Y.S. Mahajan, Z. Fu, and S. Malik. Zchaff2004: An efficient SAT solver. In *Selected Papers from SAT 2004*. Springer (LNCS), to appear.

[31] JoÃčo P. Marques-Silva. The impact of branching heuristics in propositional satisfiability algorithms. In *Proceedings of the 9th Portuguese Conference on Artificial Intelligence (EPIA)*, 1999.

[32] João P. Marques-Silva and Karem A. Sakallah. Conflict analysis in search algorithms for propositional satisfiability. In *Proc., IEEE Conference on Tools with Artificial Intelligence*, November 1996.

[33] João P. Marques-Silva and Karem A. Sakallah. GRASP: A new search algorithm for satisfiability. In *Proc., IEEE/ACM International Conference on Computer-Aided Design*, November 1996.

[34] João P. Marques-Silva and Karem A. Sakallah. Boolean satisfiability in electronic design automation. In *Proc., IEEE/ACM Design Automation Conference (DAC '00)*, June 2000.

[35] Ken L. McMillan. Applying SAT methods in unbounded symbolic model checking. In *Proc., CAV 2002*, pages 250–264, 2002. LNCS Volume 2404.

[36] M. Moskewicz, C. Madigan, Y. Zhao, and L. Zhang. Chaff: Engineering and efficient sat solver. In *Proc., 38th Design Automation Conference (DAC2001)*, June 2001.

[37] Slawomir Pilarski and Gracia Hu. SAT with partial clauses and back-leaps. In *Proc., DAC 2002*, pages 743–746, 2002.

[38] Slawomir Pilarski and Gracia Hu. Speeding up SAT for EDA. In *Proc., DATE 2002*, 2002.

[39] P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9(3):268–299, August 1993.

[40] Lawrence O. Ryan. Efficient algorithms for clause learning SAT solvers. Master's thesis, Simon Fraser University, Burnaby, Canada, 2004.

[41] Sanjit Seshia and Randal Bryant. Deciding quantifier-free Presburger formulas using parameterized solution bounds. In *Proc., LICS 2004*, 2004.

[42] P. Stephan, R. Brayton, and A. Sangiovanni-Vincentelli. Combinational test generation using satisfiability. *IEEE Transactions on CAD*, 1996.

[43] Miroslav N. Velev and Randal E. Bryant. Effective use of boolean satisfiability procedures in the formal verification of superscalar and VLIW microprocessors. In *Proc., IEEE/ACM Design Automation Conference (DAC '01)*, pages 226–231, June 2001.

[44] H. Zhang and M. Stickel. Implementing Davis-Putnam's method by tries. Technical report, University of Iowa, 1994.

[45] H. Zhang and M. Stickel. An efficient algorithm for unit-propagation. In *Proc., the Fourth International Symposium on Artificial Intelligence and Mathematics.*, 1996.

[46] L. Zhang, C. Madigan, M. Moskewicz, and S. Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *Proc., International Conference on Computer Aided Design (ICCAD2001)*, November 2001.

[47] L. Zhang and S. Malik. Cache performance of SAT solvers: A case study for efficient implementation of algorithms. In *Proc., Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT-2003)*, 2003.

[48] L. Zhang and S. Malik. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *Proc., DATE 2003*, March 2003.