Enhancing Davis Putnam with Extended Binary Clause Reasoning

Fahiem Bacchus
Dept. Of Computer Science
University Of Toronto
Toronto, Ontario
Canada, M5S 1A4
fbacchus@cs.toronto.edu

Abstract

The backtracking based Davis Putnam (DPLL) procedure remains the dominant method for deciding the satisfiability of a CNF formula. In recent years there has been much work on improving the basic procedure by adding features like improved heuristics and data structures, intelligent backtracking, clause learning, etc. Reasoning with binary clauses in DPLL has been a much discussed possibility for achieving improved performance, but to date solvers based on this idea have not been competitive with the best unit propagation based DPLL solvers. In this paper we experiment with a DPLL solver called 2CLS+EQ that makes more extensive use of binary clause reasoning than has been tried before. The results are very encouraging—2CLS+EQ is competitive with the very best DPLL solvers. The techniques it uses also open up a number of other possibilities for increasing our ability to solve SAT problems.

Introduction

Many interesting problems can be encoded as satisfiability problems, e.g., planning problems (Kautz & Selman 1992), probabilistic planning problems (Majercik & Littman 1998), verification problems (Biere *et al.* 1999), etc. For many of these problems a decision procedure is required, e.g., in many verification problems one wants to prove that no model of a buggy configuration exists.

In the realm of decision procedures, the backtracking based Davis Putnam (DPLL) procedure dominates.[1] Over the past ten years the basic procedure has been significantly improved with better heuristics (Li & Anbulagan 1997), data structures (Zhang 1997), intelligent backtracking (Bayardo & Schrag 1997), clause learning (Moskewicz *et al.* 2001), equality reasoning (Li 2000), etc. These improvements have had such an impact on performance, that converting a problem to SAT and using DPLL can often be the fastest solution technique. For example, in the verification community it has been shown that for many problems DPLL can be faster than other techniques (or at least just as fast and much easier to use) (Copty *et al.* 2001).

With one notable exception (Van Gelder 2001) current DPLL solvers utilize unit clause reasoning, i.e., unit propagation (UP). The advantage of using UP is that it can be implemented very efficiently, while the disadvantage is that it has limited pruning power and thus DPLL might explore a large number of nodes during its search for a solution. It is also possible to use binary clause reasoning in DPLL. This yields the dual: it is significantly less efficient to implement, but it can also cause DPLL to explore many fewer nodes due to its greater pruning power. This tradeoff between more reasoning at each node of a backtracking tree and the exploration of fewer nodes is well known. And, as we shall see, on some problems this tradeoff does not favor binary clause reasoning. However, we will also see that an extremely competitive DPLL solver based on binary clause reasoning can be built.

In the sequel we first introduce DPLL, UP and the various types of binary clause reasoning that we employ, then a number of formal properties concerning the relative power of these various forms of reasoning are presented and related previous work is discussed. This leads to the 2CLS+EQ sat solver that employs binary clause reasoning. The system uses two additional features to improve its performance—intelligent backtracking and a technique called pruneback. These are discussed next. A number of empirical results are presented to give a picture of the performance that 2CLS+EQ achieves, and finally, we close with some concluding remarks.

UP and Binary Clause reasoning in DPLL

DPLL takes as input a CNF formula which is a conjunctive set of clauses, with each clause being a disjunctive set of literals, and each literal being either a propositional variable v or its negation $\bar{v}$. DPLL decides whether or not there exists a truth assignment that satisfies the formula.

DPLL is most easily presented as a recursive algorithm, in which case we can view its inputs as being a pair (F, A), where F is a CNF formula, and A is a set of literals that have already been assigned TRUE by previous invocations. Initially DPLL is called with the input formula $\mathcal{F}$ and $A = \emptyset$, i.e., with $(\mathcal{F}, \emptyset)$.

[1] The original Davis Putnam procedure used directed resolution not backtracking. The backtracking version was developed by Davis, Logemann and Loveland (Cook & Mitchell 1997). Hence our use of "DPLL" instead of "DP".

There are 2 basic transformations that DPLL can perform on its input prior to invoking itself recursively. First, it can compute the reduction of its input by a literal ℓ (also called *forcing* ℓ), denoted by $(F, A)[\ell]$. $(F, A)[\ell] = (F', A')$ where F' is generated from F by removing from F all clauses containing ℓ and then removing $\bar{\ell}$ from all the remaining clauses, and A' is simply $A \cup \{\ell\}$. For example, $(\{(a, b, \bar{c}), (\bar{a}, d), (e, f)\}, [g, \bar{h}])[a] = (\{(d), (e, f)\}, [a, g, \bar{h}])$

Second, it can perform various modifications on the formula component of its input. The most important of these is unit clause reduction. Unit clause reduction simply selects a clause from the input formula that has length 1 (a unit clause) and performs a reduction of the input by the literal in that clause. For example, reduction of the unit clause (a) converts the input $(\{(a), (\bar{a}, b), (a, c)\}, [d])$ to $(\{(b)\}, [a, d])$. A unit clause reduction can generate new unit clauses. *Unit Propagation* (UP) is the iterative process of doing unit clause reductions until either (a) a contradiction is achieved, or (b) there are no more unit clauses in the input. The order in which the unit clause reductions occur is not important to the correctness of the algorithm. An *contradiction* is achieved when the set of assigned literals, A, contains both ℓ and $\bar{\ell}$ for some literal ℓ. For example, UP$(\{(a), (\bar{a}), (b, c, d), (b)\}, [d]) = (\{(b, c, d), (b)\}, [d, a, \bar{a}])$ where a contradiction has been discovered.

With these two components, the basic implementation of DPLL is to first perform unit propagation on the input formula. If the resulting formula is empty, i.e., all clauses have been satisfied, then A is a satisfying truth assignment (any variable not assigned a value in A is free to take any value) and we return TRUE to the calling invocation. Otherwise, if A contains a contradiction then this particular collection of assigned literals cannot be extended to a solution, and we backtrack by returning FALSE. Otherwise, DPLL recursively searches for a satisfying truth assignment containing ℓ and if none exist, for one that contains $\bar{\ell}$. If neither extension succeeds DPLL backtracks by returning false to the calling invocation. Note that DPLL is doing a depth-first search of a tree with each DPLL invocation being a new node visited in the tree, and each return being a backtrack in the tree search.

```
DPLL(T,A)
1. (T',A')=UP(T,A)
2. if A' contains a contradiction
3.    return(FALSE)
4. elseif T' is empty
5.    return(TRUE)
6. l := selectVarNotInA(T,A)
7. if(DPLL((T,A)[l])
8.    return(TRUE)
9. else
10.   return(DPL((T,A)[l̄]))
```

Except for the 2clsVER system of (Van Gelder 2001), current DPLL solvers use this basic algorithm (along with other orthogonal improvements). In particular, unit propagation is all that is used in the formula reduction phase.

The input formula might also contain many binary clauses, and it is possible to do various kinds of reductions of the input formula by reasoning with these clauses as well, as is done by the system we describe in this paper. All of these reductions are done in conjunction with UP.

The first is to perform all possible resolutions of pairs of binary clauses. Such resolutions yield only new binary clauses or new unit clauses. We denote by BinRes the transformation of the input that consists of repeatedly (a) adding to the formula all new binary or unit clauses producible by resolving pairs of binary clauses, and (b) performing UP on any new unit clauses that appear (which in turn might produce more binary clauses causing another iteration of (a)), until either (1) a contradiction is achieved, or (2) nothing new can be added by a step of (a) or (b). For example, BinRes$(\{(a, b), (\bar{a}, c), (\bar{b}, c)\}, [])=(\{(a, b)\}, [c])$: resolving the binary clauses produces the new binary clauses (b, c), (a, c), and $(c, c) = (c)$. Then unit propagation yields the final reduction.

This example shows that BinRes can produce more literal assignments than can UP, as UP applied to this input formula yields no changes. This also means that BinRes can produce a contradiction in situations where UP cannot: e.g., if we enhance the above example so that BinRes also produces the clause $(\bar{c})$ we will get a contradiction. BinRes gives us the obvious algorithm DPLL-BinRes in which we substitute BinRes for UP in line 1.

OBSERVATION 1 *DPLL-BinRes has the potential to explore exponentially fewer nodes than DPLL-UP.*

At a particular node DPLL-BinRes might detect a contradiction while DPLL-UP might descend trying other literals and exploring an exponentially sized subtree. However, this is only a potential savings: the subtrees that DPLL-BinRes avoids may turn out to be easy for DPLL-UP to refute.

Another common technique used in DPLL solvers is failed literal detection (Freeman 1995). Failed literal detection is a one-step lookahead with UP. Say we force literal ℓ and then perform UP. If this process yields a contradiction then we know that $\bar{\ell}$ is in fact entailed by the current input and we can force it (and then perform UP). DPLL solvers often perform failed literal detection on a set of likely literals at each node. The SATZ system (Li & Anbulagan 1997) was the first to show that very aggressive failed literal detection can pay off. Failed literal detection and binary resolution are related.

OBSERVATION 2 *If BinRes forces the literal ℓ, then failed literal detection on $\bar{\ell}$ would also detect that ℓ is entailed.*

For example, BinRes forces c in the formula $(\{(a, b), (\bar{a}, c), (\bar{b}, c)\}, [])$. On the other hand if $\bar{c}$ is forced UP will generate a contradiction: i.e., that c is entailed will be detected by the failed literal test. This observation can be proved by examining the implication graph representation of binary clauses.

OBSERVATION 3 *Failed literal detection is able to detect entailed literals that cannot be detected by BinRes.*

For example, if we test c with failed literal detection in the formula $(\{(\bar{c}, a), (\bar{c}, b), (\bar{c}, d), (\bar{a}, \bar{b}, \bar{d})\}, [])$, we will detect a contradiction and thus that $\bar{c}$ is entailed. BinRes on the other hand, does not force $\bar{c}$. So we see that failed literal detection

is strictly stronger than BinRes.

This example indicates that a weakness of BinRes is that it does not consider the non-binary clauses. This weakness can be addressed by hyper-resolution.

Hyper-resolution is a resolution step that involves more than two clauses. Here we define a hyper-resolution step to take as input *one* n-ary clause ($n \geq 2$) $(l_1, l_2, ..., l_n)$ and $n-1$ binary clauses each of the form $(\bar{l}_i, \ell)$ ($i = 1, \ldots, n-1$). It produces as output the new binary clause (ℓ, l_n). For example, using hyper-resolution on the inputs (a, b, c, d), $(h, \bar{a})$, $(h, \bar{c})$, and $(h, \bar{d})$, produces the new binary clause (h, b). Note that the standard resolution of two binary clauses is covered by this definition (with $n = 2$).

With this hyper-resolution step we define a more powerful form of formula reduction. We denote by HypBinRes the transformation of the input that is exactly like BinRes except that it performs the above hyper-resolution instead of simply resolving binary clauses. DPLL-HypBinRes is then defined to be DPLL with HypBinRes substituted for UP.

OBSERVATION 4 *HypBinRes detects the same set of forced literals as repeatedly (a) doing a failed literal test on* ***all*** *literals, and (b) performing UP on all detected entailed literals, until either (1) a contradiction is achieved, or (2) no more entailed literals are detected.*

This observation can be proved by showing that HypBinRes forces a literal if and only if it can be detected by the failed literal test. The requirement for repeating the failed literal test follows from fact that HypBinRes is run to closure. Note that simply performing the failed literal test on every literal is not as powerful as HypBinRes. One would have to retest every literal every time an entailed literal is unit propagated until no new entailed literals are detected. As a result it is much more efficient to perform HypBinRes rather than repeated failed literal tests—HypBinRes does not need to repeat work in the same way.

OBSERVATION 5 *DPLL-HypBinRes has the potential to explore exponentially fewer nodes than DPLL-BinRes.*

This follows from the fact that DPLL-HypBinRes can detect contradictions that DPLL-BinRes cannot.

HypBinRes is also very useful when it comes to computing heuristics. A very useful heuristic is to rank literals by the number of new literals they would force under UP. The idea is to split next on the literal that along with its negation will cause the largest reduction in the formula under UP—thus the two recursions will both have to deal with a smaller formula. This is the heuristic used by the SATZ (Li & Anbulagan 1997). However, this heuristic is costly to compute, and can usually only be estimated. SATZ for example evaluates this heuristic on some set of candidate literals by unit propagating each one and then counting the number of newly forced literals. (Failed literal detection is an important side effect of this process). HypBinRes has the following property:

OBSERVATION 6 *A literal ℓ will force a literal ℓ' under UP if and only if the binary clause $(\bar{\ell}, \ell')$ is in the formula after performing HypBinRes.*

Thus after performing HypBinRes a simple count of the number of binary clauses a literal's negation participates in yields the precise number of literals that would be forced by unit propagating that literal. This observation is a direct corollary of Observation 4.

Finally, there is one more type of binary clause reasoning that is useful, equality reduction. If a formula F contains $(\bar{a}, b)$ as well as $(a, \bar{b})$ (i.e,, $a \Rightarrow b$ as well as $b \Rightarrow a$), then we can form a new formula EqReduce(F) by equality reduction. Equality reduction involves (a) replacing all instances of b in F by a (or vice versa), (b) removing all clauses which now contain both a and $\bar{a}$, (c) removing all duplicate instances of a (or $\bar{a}$) from all clauses. This process might generate new binary clauses.

For example, EqReduce$\big(\{(a, \bar{b}), (\bar{a}, b), (a, \bar{b}, c), (b, \bar{d}), (a, b, d)\}, [e]\big) = \big(\{(a, d), (a, \bar{d})\}, [e]\big)$. Clearly EqReduce$(F)$ will have a satisfying truth assignment if and only if F does. Furthermore, any truth assignment for EqReduce(F) can be extended to one for F by assigning b the same value as a.

EqReduce can be added to HypBinRes (HypBinRes+eq) by repeatedly doing (a) equality reduction, (b) hyper-resolution, and (c) unit propagation, until nothing new is added or a contradiction is found. DPLL-HypBinRes+eq is then defined to be DPLL using HypBinRes+eq instead of UP

OBSERVATION 7 *HypBinRes+eq detects the same set of forced literals as HypBinRes.*

The two binary clauses $(a, \bar{b})$ and $(\bar{a}, b)$ allow HypBinRes to deduce everything that HypBinRes+eq can.

This means that modulo changes in the heuristic choices it might make, DPLL-HypBinRes+eq does not have the potential to produce exponential savings over DPLL-HypBinRes. The benefit of using equality reduction is that for many problems equality reduction significantly simplifies the formula. This can have a dramatic effect on the time it takes to compute HypBinRes.

We have implemented DPLL-HypBinRes+eq along with two key extra improvements, which we describe in the next section. But first we relate DPLL-HypBinRes+eq to the closest previous work.

EQSAT (Li 2000) is a system specialized for equality reasoning. It does much more with equalities than the simple equality reduction described above. In particular, it represents ternary equalities with a special non-causal formula and does more extensive reasoning with these higher order equalities. As a result it is the only sat solver capable of solving the parity-32 family of problems which involve large numbers of equalities. However, it is not particularly successful at solving other types of theories. See (Simon & Chatalic 2001) for detailed empirical results on EQSAT and other solvers.

2clsVER (Van Gelder 2001) is a DPLL solver that does binary clause reasoning. However, it does only BinRes not HypBinRes. On the other hand, it does extensive subsumption checking which our system does not do. Nevertheless, its performance lags significantly behind our system.

2-simplify (Brafman 2001) is a preprocessor that uses binary clause reasoning to simplify the formula. It does a limited version of HypBinRes+eq. In particular, it does

BinRes as well as EqReduce but only a limited form of hyper-resolution (a step called "derive shared implications" in (Brafman 2001)). In addition it does not repeat these reductions until closure is achieved, but rather only performs these reductions once. Achieving closure can make a tremendous difference in practice.

DCDR (Rish & Dechter 2000) is a solver that can choose to resolve away a variable during search rather than split on it (the more recent 2clsVER system also does this). The resolution steps are restricted so that they never generate a resolvant larger than k literals, for some small k. Although it worked well on some problems, the system was not competitive with UP based solvers. Interestingly, we can unroll the hyper-resolution step we have defined here into a sequence of ordinary resolution steps. However, these steps would produce intermediary resolvants of arbitrary size. Hyper-resolution allows us to avoid generating these large intermediate resolvants, moving instead directly to the small final resolvant. In one sense, hyper-resolution acts as macro that allows us to capture specific useful sequences of resolution steps.

The Sat Solver 2CLS+EQ

We have implemented DPLL-HypBinRes+eq in a system called 2CLS+EQ. 2CLS+EQ also contains two key extra improvements, intelligent backtracking and a simplified form of clause learning that is closely related to the pruneback techniques described in (Bacchus 2000). In this section we discuss these two improvements. The implementation itself has a number of interesting features, but space precludes discussing them.

Intelligent backtracking has been found to be essential for solving the more realistic structured problems that arise in, e.g., AI planning problems and hardware verification problems. Intelligent backtracking in DPLL was first implemented in (Bayardo & Schrag 1997) and it is an essential component of what is probably the most powerful current DPLL solver ZCHAFF (Moskewicz *et al.* 2001).

The set of assigned literals A can be divided into two subsets. The set of choice literals (those literals that were assigned as a result of being split on by DPLL), and the other literals that were forced by whatever reduction process is being used. When a contradiction is found A will contain a literal ℓ and its negation $\bar{\ell}$. Intelligent backtracking is based on identifying a subset of the choice literals that was responsible for forcing ℓ and a subset responsible for forcing $\bar{\ell}$. The union of these two sets $\bar{C} = \{l_1, \ldots, l_k\}$ is a set of choice literals that cannot be simultaneously made true. That is, $\mathcal{F} \models \neg(l_1 \wedge \cdots \wedge l_k)$, or in clausal form $\mathcal{F} \models C = (\bar{l}_1, \cdots, \bar{l}_k)$. This new clause allows DPLL to backtrack to the deepest level where one of these literals was assigned (i.e., the shallowest level they were all assigned), say literal l_k at level i, perhaps skipping many intermediate levels, to try the alternate branch ($\bar{l}_k$) at that level—the clause gives a proof that no solutions exist in the subtree under the assignment l_k. If after trying $\bar{l}_k$ in the alternate branch DPLL again backtracks to level i, it would in a similar manner have discovered another new clause $C' = (\bar{l}'_1, \ldots, \bar{l}'_h, l_k)$ this time containing l_k (the negation of the current assignment at level i). C and C' can thus be resolved to yield a new clause $C_1 = (\bar{l}_1, \ldots, \bar{l}_{k-1}, \bar{l}'_1, \ldots, \bar{l}'_h)$ that allows DPLL to now backtrack to the deepest assignment in C_1. This backtrack might also skip a number of levels.

Notice that the kind of reasoning used to infer ℓ and $\bar{\ell}$ from the set of choice literals is irrelevant to this process. As long as we can identify a subset of choice literals responsible for forcing a literal, we can then compute a valid new clause by unioning the subset that forced ℓ with that which forced $\bar{\ell}$. Intelligent backtracking can then be performed. In DPLL-UP this identification process is relatively simple. A literal ℓ can only be forced by an original clause that has became unit. Hence the choice literals that forced ℓ is simply the union of the choice literals that forced the negations of the other literals in clause, and this set can be computed by simply backtracing through the clauses that forced each literal. In other words, all the bookkeeping that is required is to associate a clause with each forced literal.

With HypBinRes+eq the bookkeeping required is considerably more complex. Since a literal could be forced by the resolution of two binary clauses, we must also keep track of the (potentially complex) reasons the binary clauses came into existence. Nevertheless, it is possible to implement the necessary record structures to support backtracing from a literal to a set of choice literals that entailed it. An initial implementation had been completed when we discovered the approach described in (Van Gelder 2001) which presented a much cleaner record structure supporting greater structure sharing. 2CLS+EQ implements Van Gelder's method.[2]

The other technique we implemented is related to the backpruning techniques described in (Bacchus 2000). It is best described by a example. Say that a contradiction is discovered at level 30 in the search tree, and we compute that new clause $C = (\bar{a}, \bar{b}, \bar{c}, \bar{d})$ with a having been assigned at level 1, b at level 5, c at level 10, and d at level 25. This clause allows us to backtrack to level 25 and there try $\bar{d}$. Suppose we subsequently backtrack from the subtree rooted by $\bar{d}$ to level 23 to try an alternate branch. It is quite possible that in this new branch we could again try to assign d. But we know from the clause C that d cannot hold while a, b and c are still true—which they are since we have not as yet backtracked that far. Assigning d would violate the newly discovered clause C. In fact, $\bar{d}$ must be forced at all levels until we backtrack to level 10 to unassign c.

We have not as yet implemented clause learning, and if we had the forcing of $\bar{d}$ would be taken care of by the presence of the new clause C. Instead, we use a simpler approach of "backpruning" d to the next highest level in C at the time we discover C. We then discard C. Backpruning has the effect of disallowing d (i.e., forcing $\bar{d}$) at every level until we backtrack to level 10. A related backprune is that we also know from C that the binary clause $(\bar{c}, \bar{d})$ must hold between levels 5 (while b and a are still assigned) and 10

[2] We also had to solve the problem of garbage collecting these record structures on backtrack. This problem was mentioned as open in (Van Gelder 2001). We were able to allocate off a stack and reduce garbage collection to a single move of the stack pointer.

(below level 10 $\bar{d}$ will be forced and this binary clause will be subsumed). Using a similar approach we force this binary clause at these levels. In this way we get significant use out of the clause C without having to record it.[3]

Empirical Results

In our empirical tests we made use of the wonderful Sat-Ex resource. We ran 2CLS+EQ on a variety of local machines and then normalized all of our timings to the Sat-Ex machine standard (71.62 using the Dimacs machine scale program). As a check that this normalization was reasonable we also ran ZCHAFF on our local machines on a number of problems and compared its normalized times to those reported in Sat-Ex, the timings were accurate to 5%. Hence, the reader can reasonably compare the timings we report here to those available on-line at the Sat-Ex site.[4]

2CLS+EQ is intended to be a general purpose sat solver capable of solving a wide variety of sat problems. Hence, the first experiment we ran was to try to solve a large collection of problems that the best general purpose sat solvers find easy. We used the following benchmark families (named as in the Sat-Ex site): aim-100, aim-200, aim-50, ais, bf, blocksworld, dubois, hole, ii-16, ii-8, jnh, morphed-8-0, morphed-8-1, parity-8, ssa, ucsc-bf, and ucsc-ssa. These families contained a total of 722 problems. 2CLS+EQ was able to solve all of these problems in 16m 39s (999.52 sec). The fastest solver for these problems was ZCHAFF which completed in 4m 13s (253 sec) Second was SATO (v.3.00) which took 8m 43s (523 sec). Then RELSAT-2000 which took 31m 36s, and RELSAT (v.1.12) 34m 37s. The next fastest solver SATZ failed to solve 2 problems from the dubois family and one of the blocksworld problems within a 10,000 sec. timebound, EQSAT failed to solve 9 problems from the ucsc-bf family, and SATO-v.3.21 failed to solve three problems from the ii-16 family. Thus we see that 2CLS+EQ was third fastest on this suite of problems, was only 4 times as slow as ZCHAFF, and was one of only 4 solvers able to solve all problems.

In all of our experiments we used the same heuristic for variable selection. This heuristic first performs a restricted amount of lookahead to see if it can detect any binary clauses that are entailed by both a literal and its negation.[5] If any such clauses are found, they are added to the theory and HypBinRes+eq performed. Then to choose the next variable it simply counts the number of binary clauses each literal participates in. As pointed out in the previous section, under HypBinRes this is equivalent to the number of literals that it would forced under UP. These counts, for the literal (p) and its negation (n), are combined with a Freeman like function: $n+p+1+1024*n*p$. Ties are broken randomly.[6]

However, there are a few families for which this heuristic fails badly: par-16, ii-32, pret-60, and pret-150. For these families we found that replacing the final scoring stage with one that estimates the number of new binary clauses that would be forced by a literal under UP to be vastly superior (i.e., we estimate the number of new binary clauses rather count than the number of new unit clauses). With the standard heuristic, 2CLS+EQ required 30930.75 sec. to solve the 10 par-16 problems (ranking 20th fastest among the 24 sat solvers tested by Sat-Ex), 2229.66 sec. to solve the 17 ii-32 problems (ranking 10th), 48.97 sec. to solve the 4 pret-60 problems (ranking 15th), and *cannot solve* any of the pret-150 problems within a 10,000 sec. timebound (tied for 13th with 12 other solvers, including POSIT and SATZ). However, with scores based on estimating the number of binary clauses it required only 45.32 sec. to solve the par-16 family (ranking 4th), 495.73 sec. to solve ii-32 (ranking 7th), 0.82 sec. to solve pret-60 (ranking 10th), and 56.34 sec. to solve pret-150 (ranking 11th).

The first experiment indicates that the competitive solvers in the general purpose category are ZCHAFF, RELSAT-2000, and SATO-v3.00. (this is also confirmed by the solver ranking given on the Sat-Ex site). The next experiment examines 2CLS+EQ's performance on a collection of families that these three solvers find most difficult. We excluded the families parity-32, g, and f, as none of the general purpose solvers can solve these problems and neither can 2CLS+EQ. The results on the families we experimented with are shown in Table 1. (We used the standard heuristic of counting the number of new unit clauses and doing our simple lookahead, in all of these tests. In some cases our experiments were limited by our ability to locate the problem suites.) Included in the table are the results obtained by ZCHAFF, RELSAT-2000, and SATO-v3.00 as well as where 2CLS+EQ ranks (timewise) among all 23 solvers for which Sat-Ex has data. The table shows the total cpu time (in seconds) required to solve all members of the family. The number of problems in the family are indicated in brackets after the family name, and the number of problems that a solver failed on (with a 10,000 sec. timebound) are also indicated in brackets before the total time. We follow the Sat-Ex convention and count 10,000 as the increment in time for an unsolvable problem.[7]

The miters family is where 2CLS+EQ has its best performance. It is the only solver among the 23 cited on Sat-Ex that is able to solve all of these problems (all other solvers fail on 2 or more problems). ZCHAFF fails on the two problems c6288-s and c6288, and is unable to solve them even when given 172,800 sec. (48hrs) of CPU time. Interesting, 2CLS+EQ solves both of these problems without any search; i.e., an initial application of HypBinRes+eq suffices to show these problems unsatisfiable. Thus 2CLS+EQ has some potential as a preprocessor. In contrast, the 2Simplify (Brafman 2001) system does not reduce the difficulty of these

[3] In our context the complexity of hyper-resolution is heavily dependent on the number of clauses, so clause learning must be carefully implemented.

[4] The Sat-Ex data is quite accurate, and since it represents over 1 year of CPU time, difficult to reproduce locally.

[5] In particular, the procedure check if there are any pairs of 3-clauses of the form (ℓ, a, b) and $(\bar{\ell}, a, b)$ from which it can conclude the new binary clause (a, b).

[6] The random tie breaking turns out to be essential—it causes the search to cover different possibilities rather than get stuck repeatedly trying the same best scoring variable.

[7] There are many problems with this convention, but following it does allow our results to be more readily compared with those presented at the Sat-Ex site.

Family (#problems)	2CLS+EQ	2CLS+EQ ranking	RELSAT-2000	SATO-v3.00	ZCHAFF
hfo4(40)	2,622.95	13th	**569.51**	33,785.92	6,506.10
eq-checking(34)	32.53	3rd	13.88	(1) 10,007.25	**2.45**
facts(15)	95.24	5th	75.46	**12.78**	13.45
quasigroup(22)	6,760.12	8th	2,347.83	1,087.70	**845.89**
queueinvar(10)	273.93	5th	236.04	(2) 20,400.45	**15.39**
des-encryption(32)	(8) 80,258.05	3rd	(8) 80,729.42	(8) 80,402.84	**(2) 22,726.43**
fvp-unsat.1.0(4)	(2) 27,672.40	2nd	(3) 30,006.79	(3) 30,006.35	**1,224.86**
Beijing(16)	(2) 30,435.00	8th	(2) 24024.57	(4) 44078.88	**(2) 20268.12**
barrel(8)	4,793.61	3rd	(1) 11,872.80	(1) 10,417.70	**912.22**
longmult(16)	4,993.56	2nd	41,243.77	(1) 22,270.98	**4,502.49**
miters(25)	**419.80**	1st	(7) 86,670.25	(20) 209,123.33	(2) 21,289.77

Table 1: Results of the best general purpose Sat Solvers. Bracketed numbers indicate number of failures for that family. Ranking is with respect to all 23 solvers on Sat-Ex. The best times are in **bold**.

problems: the 2Simplified versions remain unsolvable by ZCHAFF (within 48hrs of CPU time).

The results demonstrate that the extended reasoning performed by 2CLS+EQ can be a competitive way of solving sat problems. In fact, the system is in many respects superior to all previous general purpose SAT solvers, except for ZCHAFF. The fvp-unsat.1.0 problems,[8] des-encryption, and some other tests we ran, show that ZCHAFF remains in a class of its own on many types of problems. Examining 2CLS+EQ's behavior on these problems indicates that 2CLS+EQ makes good progress in the search, but then becomes stuck in a part of the tree that is very difficult to exit from. A similar phenomenon occurs when using the wrong heuristic for the pret150 family. It is clear that restarts and clause learning are essential for improved performance on these types of problems, and that these features have to be implemented in 2CLS+EQ. We suspect that with these features 2CLS+EQ will be able to outperform ZCHAFF on many other problems.

The Beijing family indicates that 2CLS+EQ will, however, never be uniformly superior to UP based DPLL solvers. In this family of problems 2CLS+EQ actually performs quite well on the first 8 problems, and like the other 3 solvers is unable to solve 3bitadd_31 or 3bitadd_32. However, it takes a total of 8943.58 sec. to solve the 6 e?ddr2-10-by-5-? problems. Many other solvers find these problems very easy. 2CLS+EQ only needs to examine a total of 764 nodes during its search to solve all six problems. However, it requires an average of 11.7 seconds to examine each node. In these problems 90,058,925 binary clauses are added and retracted as we move up and down the search trees examining these 764 nodes. It seems that for these problems massive numbers of binary clauses can be generated with little or no pruning effect. In contrast, ZCHAFF solves all 6 of these problems in 18.14 sec, searching 37,690 nodes at the rate of 0.0005 seconds per node.

Finally, 2CLS+EQ utilizes a number of different features and a legitimate question is the relative impact of these features. We do not have the space to present a proper ablative study, but we can make the following general remarks. Hyper-resolution turns out to be essential in almost every case. Without it the binary sub-formula tends not to interact much with the rest of the formula and hence it forces relatively few literals.[9] Intelligent backtracking is also essential for most problems, e.g., without it 2CLS+EQ fails on 5 of the miters problems. Backpruning is less important, but without it, e.g., 2 of the miters problems cannot be solved. In some cases, however, intelligent backtracking can be a waste of time. For example, without it 2CLS+EQ can solve the barrel family in 1527.58 sec. For these problems intelligent backtracking does not decrease the number of nodes searched. Normally, however, the overhead for the record keeping intelligent backtracking requires is more like 25% rather than the 300% of this extreme example.

Conclusions

We have presented a DPLL solver that is based on doing much more extensive reasoning in order to reduce the formula at each node. We have shown that with this can in fact pay off. The result is a very robust solver that is competitive with the best general purpose solvers. More interesting is that the solver is by no means as fast as it could be. More engineering effort could be put into making it faster, and most importantly restarts and clause learning could be added. The solver also has the potential of being used as a powerful preprocessor. The overall message is that for building faster SAT solvers we do not have to be confined by the recent trends of investigating faster ways of doing simple things, but rather we can continue to investigate more interesting and powerful reasoning techniques.

References

Bacchus, F. 2000. Extending forward checking. In *Principles and Practice of Constraint Programming—CP2000*,

[8] A range of interesting hardware verification problems, including the fvp-unsat.1.0 problem suite, have been generated by M.N. Velev and are available from http://www.ece.cmu.edu/~mvelev.

[9] This is one of the reasons van Gelder's 2clsVER system is not competitive with the best solvers.

number 1894 in Lecture Notes in Computer Science, 35–51. Springer-Verlag, New York.

Bayardo, R. J., and Schrag, R. C. 1997. Using csp look-back techniques to solve real-world sat instances. In *Proceedings of the AAAI National Conference (AAAI)*, 203–208.

Biere, A.; Cimatti, A.; Clarke, E.; Fujita, M.; and Zhu, Y. 1999. Symbolic model checking using sat procedures instead of bdds. In *Proc. 36th Design Automation Conference*, 317–320. IEEE Computer Society.

Brafman, R. I. 2001. A simplifier for propositional formulas with many binary clauses. In *Proceedings of the International Joint Conference on Artifical Intelligence (IJCAI)*, 515–522.

Cook, S. A., and Mitchell, D. G. 1997. Finding hard instances of the satisfiability problem: A survey. In Du, D.; Gu, J.; and Pardalos, P. M., eds., *Satisfiability Problem: Theory and Applications*, volume 35 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society. 1–18.

Copty, F.; Fix, L.; Fraer, R.; Giunchiglia, E.; Kamhi, G.; Tacchella, A.; and Vardi, M. Y. 2001. Benefits of bounded model checking at an industrial setting. In *Computer Aided Verification (CAV)*, 436–453.

Freeman, J. W. 1995. *Improvements to Propositional Satisfiability Search Algorithms*. Ph.D. Dissertation, University of Pennsylvania.

Kautz, H., and Selman, B. 1992. Planning as satisfiability. In *Proceedings of the European Conference on Artificial Intelligence (ECAI)*, 359–363.

Li, C. M., and Anbulagan. 1997. Heuristics based on unit propagation for satisfiability problems. In *Proceedings of the International Joint Conference on Artifical Intelligence (IJCAI)*, 366–371.

Li, C. M. 2000. Integrating equivalence reasoning into davis-putnam procedure. In *Proceedings of the AAAI National Conference (AAAI)*, 291–296.

Majercik, S. M., and Littman, M. L. 1998. Maxplan: A new approach to probabilistic planning. In *Proceedings of the International Conference on Artificial Intelligence Planning and Scheduling (AIPS)*, 86–93.

Moskewicz, M.; Madigan, C.; Zhao, Y.; Zhang, L.; and Malik, S. 2001. Chaff: Engineering an efficient sat solver. In *Proc. of the Design Automation Conference (DAC)*.

Rish, I., and Dechter, R. 2000. Resolution versus search: Two strategies for SAT. *Journal of Automated Reasoning* 24(1):225–275.

Simon, L., and Chatalic, P. 2001. Satex: A web-based framework for SAT experimentation (http://www.lri.fr/~simon/satex/satex.php3). In Kautz, H., and Selman, B., eds., *LICS 2001 Workshop on Theory and Applications of Satisfiability Testing (SAT 2001)*, volume 9 of *Electronic Notes in Discrete Mathematics*. Elsevier.

Van Gelder, A. 2001. Combining preorder and postorder resolution in a satisfiability solver. In Kautz, H., and Selman, B., eds., *LICS 2001 Workshop on Theory and Applications of Satisfiability Testing (SAT 2001)*, volume 9 of *Electronic Notes in Discrete Mathematics*. Elsevier.

Zhang, H. 1997. Sato: An efficient propositional prover. In *Proceedings of the Fourteenth International Conference on Automated Deduction (CADE)*, volume 1249 of *LNCS*, 272–275. Springer-Verlag.