Generalizations of Watched Literals for Backtracking Search

Allen Van Gelder
237 B.E., University of California, Santa Cruz, CA 95064
E-mail avg@cs.ucsc.edu.

October 3, 2001

Abstract

The technique of watching two literals per clause to determine when a clause becomes a unit clause was introduced recently in the Chaff satisfiability program. That program does not perform either equivalent-literal detection or binary-clause detection. We describe a generalization of the technique that handles the tracking necessary to support equivalent-literal detection and binary-clause detection. Both generalizations are designed to preserve the important property of the original method that unbinding a variable is very efficient. We decribe a data-structure technique called "lazy deletion" that permits several operations to be performed in amortized constant time, where the more straightforward implementation might require linear time per operation. The overall efficiency of generalized technique is analyzed. Preliminary implementation results are reported.

1 Introduction

A new technique for *boolean constraint propagation* was reported recently, as part of the satisfiability solver named *Chaff2* [MMZ$^+$01]. The previously favored technique was published by Dalal and Etherington [DE92], and involved maintaining an exact count of how many literals in each clause were *not* currently contradicted. The new technique, which may be called *watching two literals*, only keeps track of whether at least two literals in the clause are not currently contradicted. In the Chaff2 environment, this is enough information because it only processes unit clauses. This note investigates how to generalize the idea (efficiently) to environments where (a) binary clauses need to be detected, and/or (b) equivalent-literal identification is being carried out. The main problem is that in the Chaff2 environment (essentially a DPLL environment [DLL62]) a variable v has three possible states: free, *true*, or *false*, whereas in the presence of equivalent-literal identification there are $2(n - 1)$ additional states: equivalent-to $\pm u$, where u is any other variable in the formula.

A major point made by the authors of Chaff2 is that the bookkeeping techniques can make an order-of-magnitude difference in the program's speed. In particular, they point out that watching two literals requires "no" work when a literal binding is retracted during backtracking. Another point they make is that the cache is used very efficiently, compared to earlier techniques. The generalizations we propose try to preserve these properties. For that reason, we would like to avoid straightforward methods such as replacing equivalent literals throughout the formula, or setting all literals in an equivalence class to true when one of them is set to true.

First we review the basic method (filling in some details omitted from the Chaff2 paper), then we describe the generalizations to accommodate equivalent-literal identification. We note that equivalent-literal identification has been found to be effective in earlier studies [Pre95, VGT96, Li00]. In this note we do not study *how to detect* equivalent literals, but only how to utilize the information, once they are detected.

`clOffsets`

	0	5	9	9	15	...	5000	
0	1	2	3	4	5		1001	

`clLits`

1	3	25	–14	33	2	2	25	–33	2	4	–25	–19	15	–16		...		
0					5				9						15		5000	

Figure 1: Basic clause arrays for a formula with 3000 literals and clauses numbered 1–1000. Clause 1 contains three literals (25, –14, 33) and one watched literal (25). Clause 2 contains two literals (25, –33) and two watched literals. Clause 3 has been deleted. Clause 4 contains four literals (–25, –19, 15, –16) and two watched literals (–25, –19).

Nearly all complete satisfiability solvers are in the DPLL family (for Davis, Putnam, Loveland, and Logemann [DLL62]). They search for a satisfying assignment by fixing variables one by one and backtracking when an assignment forces the formula to be *false*. The procedure is not very effective in its original form, but it has been enhanced with various techniques to reduce the search space. Techniques to choose the branch variable are a separate topic, not treated here. Reasoning techniques can be broadly classified as *preorder* and *postorder*. Preorder techniques are applied as the search goes forward, and include unit-clause propagation (also called boolean constraint propagation), binary-clause reasoning, equivalent-literal identification, and others [BS92, Pre95, VGT96, Li00].

Postorder techniques are applied when the search is about to backtrack, because a "conflict" has been discovered [SS96, Zha97, BS97, MMZ$^+$01]. Postorder techniques are variously called non-chronological backtracking, conflict-directed back-jumping, and learning. The "learning" label is based on the fact that new clauses can be added to the formula; we call these clauses *postorder lemmas*. A postorder lemma consists of the negations of the literals that make up a conflict set. In none of these works have preorder and postorder techniques been combined (except for unit-clause propagation). A companion paper (also submitted to AMAI-02) addresses this issue. For this note we do not study *how to derive* postorder lemmas, but we assume they will be derived and might have hundreds or thousands of literals in one clause. It is in such long clauses that watching only two literals pays major dividends.

2 Notation and Basic Data Structures

In CNF, the formula is a conjunction of clauses and each clause is a disjunction of literals; each literal is a propositional variable x or its negation $\neg x$. We denote a clause as $[q_1, q_2, \ldots, q_k]$ and a formula as $\{C_1, C_2, \ldots, C_m\}$. An empty formula is *true* and [], the empty clause, is *false*. We also define the *tautologous clause* $\top$, which is true under any assignment. The formula has n variables, m clauses, and L (occurrences of) literals; m and L can vary if clauses are added and deleted. We use L for the length of the formula.

The variables of the formula are represented by positive integers, $1, \ldots, n$; negative literals are negative integers. Two arrays of integers record the clauses in the formula, as suggested in Figure 1. The `clOffsets` array permits any clause to located in constant time, given its clause number, which never changes. The clause contents are stored in `clLits`; if this array is compacted after some clauses are deleted, `clOffsets` is updated simultaneously. Note that the clause information is a sequence of integers interpreted as follows: `nWatched`, `clLength`, `lit[1]`, $\ldots$, `lit[clLength]`. The number of watched literals is `nWatched`. We assume `clLength` ≥ 1.

3 The Two-Watched-Literal strategy

The basic idea of the *watched-literal strategy* is that a watched literal is either free or bound to *true*. When a watched literal becomes bound to *false*, it ceases to be watched.

For the *two-watched-literal strategy*, an array of lists, called `watchedIn`, is maintained; it is indexed by literals from $-n$ to n. The list of integers `watchedIn[q]` (usually) specifies the numbers of the clauses in which literal q is a watched literal. This is true as long as q has not been bound to *false*.

For a "normal" clause, the first two literals are its watched literals, and `nWatched` = 2. As variables are bound during the search, this property is maintained, if possible.

Using Figure 1 as an example, suppose variable 25 is being bound to *true*. The list `watchedIn[-25]` will contain 4, as well as other clauses where –25 is watched. Clauses in this list need to be updated, as –25 ceases to be a watched literal. We search to the right for a literal that is not *false*. We first find –19, so we swap –25 with –19. But this just puts –25 in the second position, where it still should be watched, so we search to right again for a literal that is not *false*. Suppose 15 is *false*, but –16 is either free or *true*. We swap –25 with –16, restoring the watched-literal property for the the first two literals of the rearranged clause. To complete the bookkeeping, we delete clause 4 from the list for literal –25, and insert clause 4 in the list for –16. The new configuration (including `nWatched` and `clLength`) is (2, 4, –19, –16, 15, –25).

Now we turn to the exception. Again suppose variable 25 is being bound to *true*. This time, assume that 15 and –16 are both *false*. We still swap –25 with –19. But the second search fails. To record this, we reduce `nWatched` to 1. This denotes that clause 4 is effectively a unit clause, and it implies the binding –19 = *true*. The configuration is: (1, 4, –19, –25, –16, 15).

It might seem logical now to delete clause 4 from the list for –25, because it is no longer watched, but here is the trick: By leaving this clause *in* the list, when we are finished updating clauses, the list `watchedIn[-25]` will contain precisely those clauses that became unit clauses as a result of binding 25 to *true*. In all such clauses –25 appears as the second literal. Therefore, if 25 is unbound by backtracking, the list `watchedIn[-25]` contains exactly those clauses that are no longer effectively unit clauses, and –25 is "instantly" their second watched literal. This is the basis for the claim by the Chaff2 authors that variable unbinding can be done in constant time.

However, as we have presented the procedure, when a variable binding (say 25 = *true*) is retracted, to preserve the semantics of the data structure, it is necessary to visit all the clauses in `watchedIn[-25]` and add 1 to `nWatched`. A lazy alternative is discussed later.

Also, depending on the implementation, `nWatched` might be reduced to 0 when a contradiction is discovered, before conflict processing begins. Continuing with the previous example, the binding 25 = *true* made clause 4 into the (effectively) unit clause [–19]. It might also produce a unit clause [19] elsewhere, and the processing of the binding 19 = *true* might subtract 1 again from `nWatched` in clause 4. Keeping clause 4 on the list `watchedIn[-19]` avoids having to rebuild the list when the binding of 19 is retracted.

4 Three Watched Literals

The generalization to three watched literals is straightforward. The value of `nWatched` ranges from 0 to 3 instead of 0 to 2. Now, when a binding, say q = *true*, causes a failing search for a replacement literal in a clause where $-q$ is watched and `nWatched` = 3, the clause remains on the list `watchedIn[-`q`]` and 1 is subtracted from `nWatched`. This clause would be put on a list of effectively binary clauses. Similarly, if `nWatched` < 3, we know a replacement cannot be found, so we subtract 1 from `nWatched`, leave the clause on the $-q$ list, and queue the clause for special processing according to its effective length. The complications arise when equivalent-literal identification is introduced.

0	1	2	3	4	5	6	7	8	9	10
	1	3	3	–10	5	6	7	–4	8	10

Figure 2: An in-tree without path compression to record equivalence classes. 3 is the leader of 2; 10 is the leader of –4, 8, and –9, etc.

5 Equivalent Literals and Watched Literals

Equivalent-literal identification is a procedure that identifies a set of literals E_ℓ that must have the same truth value or else a conflict (empty clause) results. Let's assume the result is a set statements of the form $q := \ell$, where q and ℓ are literals; ℓ called the *leader* of the set E_ℓ. Of course, there is also a set $E_{-\ell}$ containing $-q$. We are not concerned with the details of this procedure.

At any point in the computation, the literals of the formula are partitioned into equivalence classes, such as E_ℓ. Some equivalence classes contain a single literal. As the search goes forward, if the equation $p = q$ is discovered (presumably from analysis of the effectively binary clauses, but how does not matter), then the equivalence classes of p and q are merged. Upon backtracking, it becomes necessary to "unequate" p and q. That is, the present equivalence class containing them both has to be dissolved into the two former equivalence classes. Although very efficient data structures for going forward are known (under the names union-find, dynamic equivalence relations, path compression), we are not aware of corresponding results for backing up. An apparently satisfactory method is to simply record the previous state in the union-find data structure (essentially an in-tree embedded in an array; see Figure 2) every time a change to that structure is made going forward. Then backing out the changes should not cost more than installing them.

One possible way to incorporate the knowledge that $q = \ell$, where ℓ is the leader of q's equivalence class is to replace all occurrences of q by ℓ (and $-q$ by $-\ell$) throughout the formula. This is not in keeping with the spirit of the watched literal strategy, and is likely to be expensive, so we propose a different approach. Throughout this discussion we assume that we are normally watching three literals; obvious adjustments should be made for watching a different number.

To process the replacement, $q := \ell$, we examine each clause in the list `watchedIn[`q`]`. First, if none of the first three literals is equivalent to q, this is a stale entry in q's list and should be deleted; this is part of the lazy deletion technique discussed later. Next, we try to find a literal p in the clause that is free or is bound to *true*; also the leader of p must be different from the leader of any currently watched literal in the clause. If this succeeds, we swap q and p; we are done with this clause and delete it from q's list. If no such replacement can be found, we simply add this clause to `watchedIn[`ℓ`]` (leaving on q's list also). Finally, since q is not a leader, it is not considered for future branching.

Well, actually it is not that simple. The problem is that the clause might already be on ℓ's list, either because ℓ occurs in the clause or because some other literal in E_ℓ occurs in the clause. Since we want `nWatched` to reflect the number of watched literals *in different equivalence classes,* when this situation occurs, we subtract 1 from `nWatched` for this clause.

Only variables that are leaders are eligible to be bound to *true* or *false*. Say q is being bound to *false*, either because it was derived as a unit clause or it is being branched upon. Let us consider the updating procedure for the watched literals. Each clause in `watchedIn[`q`]` is visited. First, if none of the first three literals is equivalent to q, this is a stale entry in q's list and should be deleted; this is part of the lazy deletion technique discussed later. Now suppose that one or more of the first three literals *is* equivalent to q. As usual, if `nWatched` = 3, we try to find a new literal to watch and if we succeed we can delete this clause from q's list. Suppose this effort fails, or `nWatched` < 3 and we know it must fail. Then we subtract 1 from `nWatched` and leave the clause in q's list.

One of the themes of the literal-watching strategy is that the expensive operations are efficient in their cache usage

[MMZ$^+$01]. Our proposal tries to continue with this theme. Notice that a potentially expensive part of the update operations just described consists of searching through the clause literals for replacements, when the clause is very long, as is typical for postorder lemmas. However, because this searching uses contiguous memory, a high cache hit rate may be expected. Simultaneously, the search needs to check entries in the equivalence data structure, which is something like Figure 2. Here the cache usage is probably less favorable, but the entire structure is n integers, a fair portion of which might live in the cache. We do not presently have tools to measure cache usage.

We now turn to the requirements for retracting a binding. Say the binding $q = \mathit{false}$ is being retracted. We know that all clauses on the list `watchedIn[`q`]` have `nWatched` < 3. For simplicity and safety we visit each and add 1 to `nWatched`. The list resumes its usual function of identifying those clauses in which q, or something equivalent to q, is being watched.

Say the equivalence binding $q := \ell$ is being retracted. Possibly the clauses in `watchedIn[`q`]` should no longer be in `watchedIn[`ℓ`]`. However, `watchedIn[`ℓ`]` can contain many clauses other than those that were added when $q := \ell$ was processed going forward. It would be quite inefficient to search for those that need to be deleted.

Instead we adopt a *lazy deletion* technique. We leave the incorrect clauses in `watchedIn[`ℓ`]` for the time being, and we wait until the next time we traverse the list to find them and delete them. Such incorrect entries in the list are called *stale*. Since ℓ must be free and a leader at the time $q := \ell$ was bound going forward, the same is true at the time it is being retracted. Thus the next time the list `watchedIn[`ℓ`]` is traversed, it will be for some kind of binding to ℓ going forward (either $\ell = \mathit{false}$ or $\ell :=$ some other leader). During this traversal stale list entries can be detected and deleted.

To summarize, the watched literal strategy in the presence of literal equivalences requires finding the leader of a literal in many situations. Roughly speaking, the costs of operations are multiplied by the cost of finding leaders, compared to an environment without literal equivalences. This cost can be held to a near constant per "find" by using the path compression strategy and in-trees, but the constant factor is substantial, possibly 5 to 50. For this to pay off, literal equivalences have to bring about large reductions in the search space. There is some evidence for this [VGT96, Li00], but it has to be re-evaluated in the context of postorder lemmas.

6 Conclusion

The procedures described have been implemented using a two-literal watching policy and equivalent literal processing. However, the clauses supervised under this policy are only those derived as postorder lemmas (i.e., as the result of conflicts). The original formula is handled by older methods.

The implementation is grafted onto an existing program, `2clVER` which is described in greater detail in a companion paper also submitted to AMAI-02. The situation in which watching only two literals pays greatest dividends is when very long clauses are present. Postorder lemmas tend to be long clauses.

The most important statement we can make at this time is that the method works correctly in hundreds of tests. The performance is satisfactory in terms of time, but there is nothing with which to make a direct comparison. The bottlenecks for the program are elsewhere: the memory use is too high due to keeping too many postorder lemmas, and the time is too great due to inefficiencies in binary clause processing.

One of the purposes of preparing this report is to assist other researchers to incorporate the literal-watching strategy in environments that perform equivalent-literal processing (e.g., [Li00]), hopefully starting from a better base that we did.

Acknowledgments

This work was supported in part by NSF grants CCR-9505036 and CCR-9503830.

References

[BS92] A. Billionnet and A. Sutter. An efficient algorithm for the 3-satisfiability problem. *Operations Research Letters*, 12:29–36, July 1992.

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

[DE92] M. Dalal and D. Etherington. A hierarchy of tractable satisfiability problems. *Information Processing Letters*, 44:173–180, December 1992.

[DLL62] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5:394–397, 1962.

[Li00] C. M. Li. Integrating equivalency reasoning into davis-putnam procedure. In *AAAI*, 2000.

[MMZ+01] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *39th Design Automation Conference*, June 2001.

[Pre95] D. Pretolani. Efficiency and stability of hypergraph SAT algorithms. In D. S. Johnson and M. Trick, editors, *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge.*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society, 1995.

[SS96] J. P. Silva and K. A. Sakallah. GRASP–a new search algorithm for satisfiability. In *Proc. IEEE/ACM Int'l Conf. on Computer-Aided Design*, pages 220–227. IEEE Comput. Soc. Press, 1996.

[VGT96] A. Van Gelder and Y. K. Tsuji. Satisfiability testing with more reasoning and less guessing. In D. S. Johnson and M. Trick, editors, *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge.*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society, 1996. (also at `ftp://ftp.cse.ucsc.edu/pub/avg/kclose-tr.ps.Z`).

[Zha97] H. Zhang. SATO: An efficient propositional prover. In *14th International Conference on Automated Deduction*, pages 272–275, 1997.