Toward Leaner Binary-Clause Reasoning in a Satisfiability Solver

Allen Van Gelder
Computer Science Dept., SOE, University of California, Santa Cruz, CA 95064
E-mail avg@cs.ucsc.edu.

March 11, 2003

Abstract

Binary-clause reasoning has been shown to reduce the size of the search space on many satisfiability problems, but has often been so expensive that run-time was higher than that of a simpler procedure that explored a larger space. The method of Sharir for detecting strongly connected components in a directed graph can be adapted to performing "lean" resolution on a set of binary clauses. Beyond simply detecting unsatisfiability, the goal is to find implied equivalent literals, implied unit clauses, and implied binary clauses.

1 Introduction

We assume the reader is familiar with the *satisfiability problem*, which seeks to determine if any assignment to the propositional variables of a Boolean formula causes it to evaluate to *true*. In recent research, planning problems, hardware and software verification problems and others have been encoded as satisfiability problems. There is a substantial difference among these types of problems, however. For planning problems, the successful outcome is a satisfying assignment, which describes the plan, and is easily checkable. For verification problems, the successful outcome is the *lack of* a satisfying assignment. But the single bit "no" is not easily checkable.

If an important decision is to be taken based on claims that certain statements have been formally verified, there is a need to be able to verify the verifier. It is probably impractical to prove that the solver is bug-free, but if it produces an easily checkable proof, then that *proof* can be verified without addressing the issue of whether the program is bug-free.

Propositional resolution proofs are very easy to check, independently of the program that produced the proof. The proof can be presented as a sequence of "records". The m input clauses are indexed 1 through m in this sequence. After that, each record consists of its index in the sequence, the clashing literal, the two operand clauses (i.e., their indexes), and the resolvent clause. The correctness of the proof can be established merely by applying the definition of the resolution operation to each record in isolation. In theoretical terms, the checking problem is in logspace, a very easy complexity class.

These considerations motivate our study of the problems connected with extracting proofs from the runs of satisfiability solvers. Elsewhere, we describe how to construct a refutation as a by-product of a DPLL search that incorporates pre-order binary-clause reasoning, extending earlier work of Lee and Plaisted [VG02b, LP92].

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 binary-clause reasoning, equivalent-literal identification, and other efficient reasoning steps whose goal is to show that certain variable bindings cannot lead to a satisfying assignment [BS92, Pre95, VGT96, Li00, Bac02]. (The unit-clause rule can be absorbed into the policy for selecting the branching literal, so it is not classified as preorder reasoning.)

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. These techniques are compared in a recent paper [ZMMM01]. There are substantial difficulties in combining preorder and postorder techniques, and only two attempts have been reported [VG02b, Bac02].

This paper describes a more efficient procedure for binary-clause reasoning than those previously used. Using depth-first search as the foundation, the procedure derives a "lean" set of binary clauses that justify equivalent literal replacements, as well as certain implied unit clauses. In many cases the set of derived clauses is much smaller than would be derived by two-closure [VGT96, Bac02]. Similar techniques have been reported by del Val [dV01] for detecting strongly connected components in binary clauses.

Although techniques for binary-clause reasoning might seem at first to be independent of techniques for proof production and verification, we show that there is a close connection in Section 5. By paying attention to the proof requirements, we arrive at a deduction strategy that would not naturally be chosen for ease of deductions alone. This strategy permits proofs for equivalence of literals to be generated with only an increase in the constant factor over inferring their equivalence without proof. Previously known methods might require quadratically many proof steps.

A new technique is introduced to "approximate" the two-closure of the given binary clauses. Using the idea of *active intervals* from depth-first search, a data structure is built in linear time that permits querying in constant time for an implied binary clause. The method is sound, but incomplete: a binary clause might be implied by the given clauses but not known by the query system.

One application for the "approximate" two-closure that we explore is that it allows us to avoid checking for *some* unit-clause inferences. These are inferences that cannot possibly be present in the implication graph, after it is condensed, based on equivalent literals, into a directed acyclic graph (DAG). The "approximate" two-closure shows that certain inferences cannot possibly be present.

The effectiveness of the approximation and its ability to eliminate searches for impossible inferences is explored experimentally with random 2-CNF formulas. The data shows a substantial gain in efficiency for finding all inferable unit-clauses. However, we are not able to show that the worst case in below $O(nm)$, which was achieved with previously reported methods for two-closure. A preliminary version of this work was presented at a symposium [VG02a].

2 Notation

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 $\neg x$. If $q = \neg x$ is a negative literal, $\neg q$ is considered to be its complement, 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*.

3 Implication Graphs

Apsvall, Plass, and Tarjan recognized that binary clauses can be interpreted as a directed graph $G = (V, E)$, called the *implication graph* [APT79]. The vertices are the literals and the binary clause $[p, q]$ is interpreted as two directed edges, $\neg p \to q$ and $\neg q \to p$. The symbol "$\to$" can be read as "implies." Their main result was that the binary clauses are unsatisfiable if and only some literal and its complement occur in the same strongly connected component (SCC) of G. An implication graph has this important symmetry property: There is a path from v to w if and only if there is a path from $\neg w$ to $\neg v$.

For a set of binary clauses, $[p, q]$ can be interpreted as two directed edges in G^T, $(p, \neg q)$ and $(q, \neg p)$. Notice that "q implies $\neg p$" would be an incorrect interpretation of $(q, \neg p)$; therefore we avoid an arrow notation for edges of G^T.

For data structures, we assume that the program in which the SCC routine is embedded maintains lists of (references or pointers to) binary clauses in which each variable occurs. Thus the list for literal p contains precisely (references to) all of the binary clauses of the form $[p, q_i]$, which enables us to find all the edges leaving p in G^T (they go to the literals $\neg q_i$) and all the edges leaving $\neg p$ in G (they go to the q_i).

For example, if clauses containing literal p are $[p, q]$, $[p, r]$, $[p, s]$, then exactly these clauses are referenced from the list for p, and the edges leaving $\neg p$ in G are $\neg p \to q$, $\neg p \to r$, and $\neg p \to s$, while the edges leaving p in G^T are $(p, \neg q)$, $(p, \neg r)$, $(p, \neg s)$.

4 Sharir Algorithm

Sharir recognized that strongly connected components (SCCs) of G can be detected by performing depth-first search with the "correct" vertex ordering [Sha81]. The correct ordering can be discovered by performing depth-first search (with any vertex ordering) on G^T the *transpose graph* of G, in which all the edges of G have their directions reversed.

Sharir's algorithm is described with the terminology used in this paper in several modern texts [BVG00, CLRS01]. It is sketched here for self-containment. The paper assumes that the reader is familiar with depth-first search (DFS), with three colors used for marking vertices, as described in these sources.

A *depth-first search series* (DFSS) visits every vertex in the graph, using one or more DFS's. Each DFS constructs a single *DFS tree*, at least implicitly. Three colors are used for marking vertices, *white* for undiscovered, *gray* for discovered but not finished (also called "open"), and *black* for finished (also called "closed"). During the DFS only white vertices are visited. Each vertex is turned gray at preorder time (also called discovery time) of its visit and is turned black at postorder time of its visit.

Sharir's algorithm proceeds in two main phases. Phase one consists of steps 1 and 2 below, while phase two consists of steps 3 and 4.

1. Initially color all vertices white. Create an empty stack S.
2. (This step comprises a DFSS on G^T.) For all vertices v in any order:

   If v is white:

      Perform a depth-first search (DFS) rooted at v in G^T.

      During this DFS, at the postorder time of each visited vertex w (also called finishing time), push w onto the stack S.

3. Reset all vertices to white.
4. For all vertices v in the order in which they are popped from the stack S:

   If v is white:

      (v is identified as the *leader* of its SCC.)

      Perform a DFS rooted at v in G.

      Precisely those vertices visited during this DFS (and in the DFS tree rooted at v) comprise the SCC of which v is leader.

5 Strongly Connected Components of the Implication Graph

Although Sharir's algorithm, sketched in Section 4, can be used directly to determine the *satisfiability* of a set of binary clauses, we are interested in obtaining much more information. If the set is unsatisfiable, we want to extract a resolution proof of that fact. If the set is satisfiable, all literals in a single SCC are "equivalent" (i.e., they must be equal in any satisfying assignment); we want to extract resolution proofs for these equivalences. Eventually, we want to discover any unit clauses that are implied and extract their resolution proofs, too.

Now we describe the algorithm for SCCs adapted for binary clauses and extraction of resolution proofs. As before, phase one consists of steps 1 and 2 and phase two consists of steps 3 and 4. There is also a phase three, consisting of step 5.

1. Initially color all vertices white. Create an empty stack S.

2. (This step comprises a DFSS on G^T.) For all vertices v in any order:

   If v is white:

   Perform a depth-first search (DFS) rooted at v in G^T.

   During this DFS, at the postorder time of each visited vertex w (also called finishing time), push w onto the stack S.

3. Reset all vertices to white (this might be done by redefining the value of white).

   Create an empty stack L.

4. For all vertices v in the order in which they are popped from the stack S:

   If v is white and $\neg v$ is also white:

   (v is identified as the *leader* of its SCC.)

   Push $\neg v$ onto the stack L.

   Perform a DFS rooted at v in G.

   Precisely those vertices visited during this DFS (and in the DFS tree rooted at v) comprise the SCC of which v is leader.

   During the DFS of G rooted at v, for each visited vertex p a binary clause $[\neg v, p]$ is derived (unless $[\neg v, p]$ is already a clause in the formula, in which case $v \to p$ is an edge of G).
   Each derived clause requires only one additional resolution step: If visiting p from parent q, the clause $[\neg v, q]$ has already been derived and the edge $q \to p$ corresponds to the clause $[\neg q, p]$. The resolvent of these two clauses is $[\neg v, p]$.

   In other cases v is simply popped and not processed; if it is white, its color remains white.

   (If v is white and $\neg v$ is black and $\neg v$ is the leader of its SCC, then then v is also the *leader* of its SCC but it might not be safe to perform a DFS rooted at v at this point. That is why v was pushed onto L as $\neg v$ was processed earlier.)

5. (This step completes a DFSS on G that began in step 4.) For all vertices v in the order in which they are popped from the stack L:

   (v must be white.)

   (v is identified as the *leader* of its SCC.)

   Perform a DFS rooted at v in G.

   Binary clauses are derived as described in step 4.

Del Val correctly observes that step 5 is unnecessary to *identify* the SCCs [dV01]. However, we include it to develop the resolution proof that can be independently checked. Note that if v and $\neg v$ are leaders of their respective SCCs and if p is in the SCC of v, then clauses $[\neg v, p]$ and $[v, \neg p]$ have been derived. These formally justify the equality $p = v$. The clause $[v, \neg p]$ can be used to replace all occurrences of p in the formula by v, justified by resolution. Similarly all occurrences of $\neg p$ can be replaced by $\neg v$.

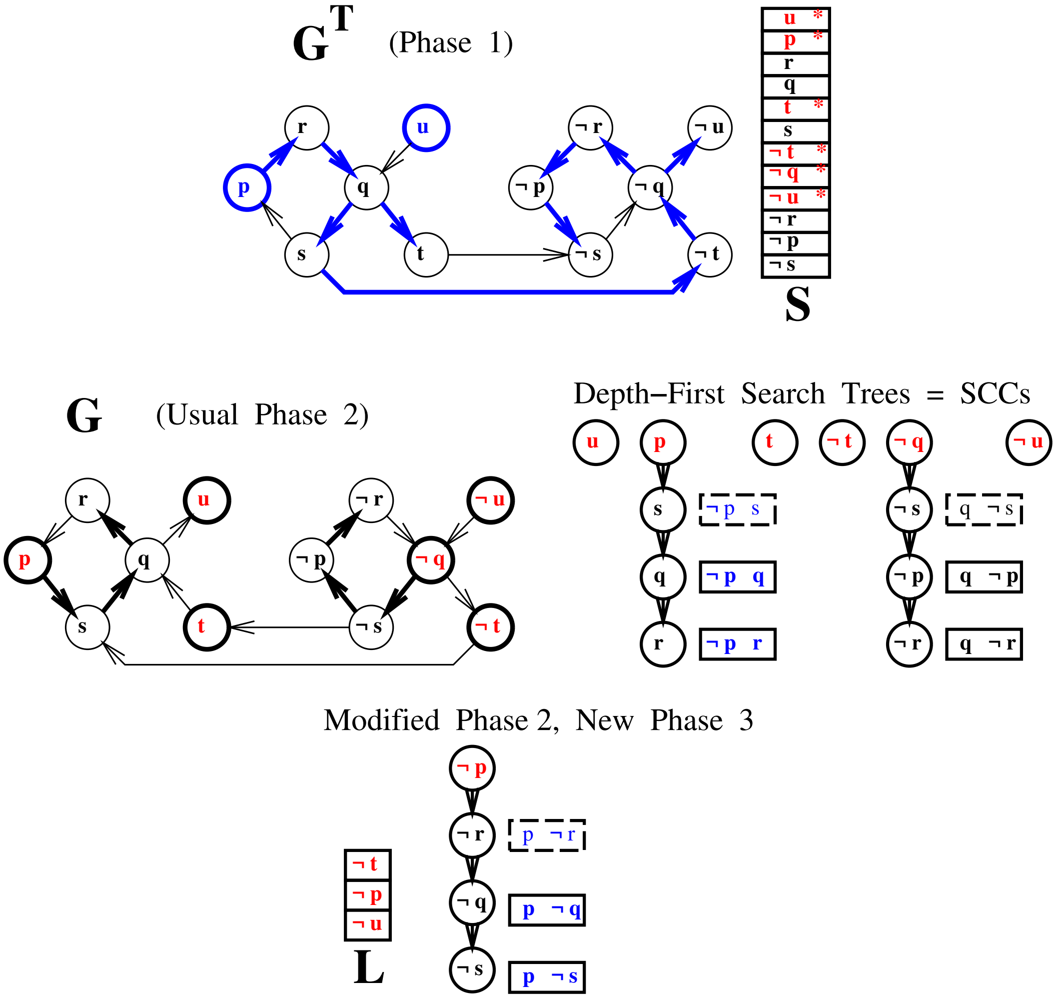


Figure 1: The modified phase two and new phase three for Sharir's SCC algorithm. Heavy lines denote roots of DFS trees and tree edges. Starred vertices in S will be roots of DFS trees in the usual phase 2. Dashed boxes alongside vertices are existing clauses, while solid boxes show derived clauses.

To understand why steps 4 and 5 use the stack L as they do, we recall the underlying reason why Sharir's algorithm works correctly. If there is a path from u to v in G, and u and v are leaders of their SCCs, then v is higher than u in the stack S. This ensures that step 4 will construct DFS trees that span a single SCC. By the symmetry of the implication graph, there is a path from u to v if and only if there is a path from $\neg v$ to $\neg u$. But step 4 ensures that $\neg u$ is higher than $\neg v$ in the stack L. Therefore step 5 will construct DFS trees that span a single SCC.

The combination of steps 4 and 5 ensures that v is the root of a DFS tree if and only if $\neg v$ is the root of a DFS tree. This structure permits the lean (i.e., linear in the number of distinct literals) derivation of binary clauses that establish literal equivalences for each DFS tree, as explained in the previous paragraph, and illustrated in the following example.

Example 5.1: See Figure 1 for an example that illustrates the main points of the modified SCC algorithm. Phase 1 begins with a DFS on G^T, rooted at p, that reaches all vertices except u. Vertices are pushed onto stack S at their finishing times. A second DFS discovers just u, which is pushed last.

Phase 2 begins by popping u from S and beginning a DFS on G at u. This terminates immediately, identifying u as a singleton SCC. Then p is popped, and the second DFS proceeds to s, q, and r. These vertices comprise the second

SCC. (Of course, u is not visited, as it is now colored black—the essential point of Sharir's algorithm.) During this DFS, clauses $[\neg p, s]$ and $[\neg s, q]$ are in the formula and are resolved to derive $[\neg p, q]$ as q is visited. Then this newly derived clause is resolved with $[\neg q, r]$ to derive $[\neg p, r]$ as r is visited.

Then r and q are popped, but are already black. Next, t is popped and makes a singleton DFS tree. Then s is popped, but is already black.

We are now at a critical point because the vertices about to be popped are complementary to those already popped. Next to be popped is $\neg t$, which makes a singleton tree. However, $\neg q$, which is popped next, will cause trouble if it is used as the root of a new DFS tree: the correct vertices will be visited, but the desired clauses will not be derived. We need clauses $[p, \neg s]$, $[p, \neg s]$, and $[p, \neg r]$. Instead we get the clauses shown alongside the tree rooted at $\neg q$.

The problem is corrected by pushing the complement of each DFS root onto a new stack L, provided the complement is white; if the complement is black the tree is not constructed. Thus u was popped first from S, so $\neg u$ is pushed onto L. Then p is popped, p is white, and $\neg p$ is white, so $\neg p$ is pushed onto L and p roots a new DFS tree. Later t is popped and $\neg t$ is pushed onto L. Still later, when $\neg t$, $\neg q$, and $\neg u$ are popped from S, they do *not* root new trees because their complements are black, i.e., have been placed in SCCs. Thus the modified phase 2 completes with DFS trees for exactly half of the SCCs, rooted at u, p and t.

Finally, in the new phase 3, vertices are popped from L and used to root the remaining DFS trees. The essential difference is that $\neg p$ becomes the leader of its SCC and the desired clauses containing p are derived as its tree is traversed. $\square$

Additional processing would be needed if v and $\neg v$ are in the same SCC, to derive the contradiction. However, we show how this situation can be avoided.

With a little extra bookkeeping during the phase 1 DFSS on G^T, it is possible to anticipate that a contradiction will occur. Suppose σ is a contradictory SCC. Then for each $q \in \sigma$, $\neg q \in \sigma$, also. If v is the earliest vertex in σ to be discovered during the DFSS on G^T, it is known by the *White Path Theorem* [CLRS01] that $\neg v$ will be a descendant of v in that search. This means that, at the time $\neg v$ is discovered, v will be a gray vertex, and the procedure can infer that there is a path *in* G from $\neg v$ to v. This path allows the derivation of the unit clause $[v]$ before steps 4 and 5 even start.

Generally, derivation of unit clauses is beneficial, so it would probably not be a terrible policy to interrupt the DFS whenever a path from a vertex to its complement is noticed, and perform the unit-clause derivation. However, there is a risk that redundant derivations might be carried out. This topic also arises after all the SCCs are identified, so we defer further discussion until Section 6.

6 Unit Clause Derivation

Whenever there is a path from $\neg v$ to v in G, the clauses on the path can be resolved in either order to produce a derivation of $[v]$. If G is acyclic, this is the only way a unit clause might be derivable.

If a DFS tree contains a path from $\neg v$ to v, this fact can be detected in constant time. At the beginning of the visit to v, simply check whether $\neg v$ is a gray vertex. If the search is in G, then $[v]$ can be derived and if the search is in G^T, then $[\neg v]$ can be derived.

As argued in Section 5, if the set of binary clauses is unsatisfiable, then it is necessarily true that some DFS tree constructed during the DFSS on G^T (see Section 5, step 2) contains a path from $\neg v$ to v for *some* v that is in a contradictory SCC. Immediately upon discovering this path (at the beginning of the visit to v) the unit clause $[\neg v]$ can be derived.

Once a unit clause, say $[\neg v]$, has been derived in the DFSS of G^T, the DFS in progress can be suspended and a DFS rooted at $\neg v$ can be performed on G. All vertices reachable from $\neg v$ are also derived as unit clauses. If $\neg v$ is in a contradictory SCC, then $[v]$ will be derived, the empty clause will be derived, and processing can stop (the DFS data structures need to be cleaned up).

If $\neg v$ is *not* in a contradictory SCC, then some number of unit clauses are derived, and the interrupted DFS on G^T resumes. However, all binary clauses containing a derived literal are now subsumed, so the DFS procedure should be

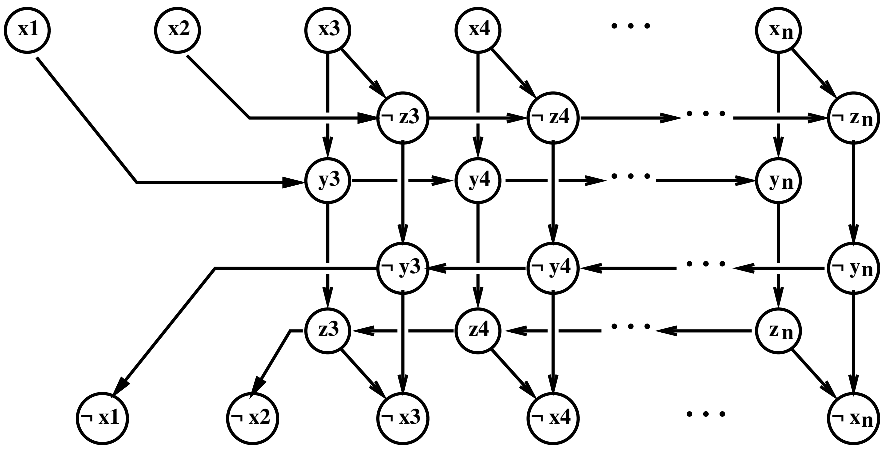


Figure 2: One search of the implication graph can miss unit clauses.

adapted to ignore such clauses. If w is about to be visited and $[\neg w]$ has been derived, then do not visit w. The visit of v, which caused the interrupt, is also aborted. If w is backtracked into and $[w]$ has been derived, then do not visit any other vertices from w; instead, backtrack to its DFS parent. By making these adaptations, it is unnecessary to alter the data structures that define G and G^T in the middle of the DFS.

With this strategy for recognizing unit-clause derivations, we will only complete the SCC algorithm when the binary-clause set is satisfiable. After equivalent literals have been replaced by their SCC leaders, G can be reformulated as an acyclic graph. To have complete unit-clause inference, it is necessary to detect any paths of the nature $\neg v$ to v in this graph. Unfortunately, one DFSS of G is not guaranteed to find all such paths.

Example 6.1: Consider the graph in Figure 2. If vertices are examined in sequence, the DFS from x_1 will visit all the y_i and z_i vertices, then DFS from x_2 will visit all the $\neg z_i$ and $\neg y_i$ vertices, so that later DFS's from $x_3, \ldots, x_n$ will not find their complements. Unless a fortunate vertex order is used, finding all the derivable unit clauses takes $\Theta(n|G|)$ work. This is the same order as performing a separate DFS from each vertex. $\square$

This example shows that complete searches of G via DFSS may be counterproductive. An early tree might contain (and color black) vertices that are needed to derive unit clauses. On the other hand, beginning a new DFS (with all vertices white) at each vertex can be highly redundant. Fortunately, we can exploit the symmetry of the implication graph G to avoid some useless work.

A *source vertex* is one with no incoming edges. Vertex $\neg v$ is a source vertex if and only if v has no outgoing edges. A vertex w is said to be *reachable from* v if there is a path from v to w or $w = v$.

Lemma 6.1: If $\neg v$ is a source vertex of G and a new DFS rooted at $\neg v$ visits both p and $\neg p$, then it also visits v.

Proof: The DFS visits all vertices that are reachable from $\neg v$, by the White Path Theorem. Since there is a path from $\neg v$ to $\neg p$, by the symmetry of implication graphs, there must be a path from p to v. Concatenate this to the path from $\neg v$ to p. ▮

Corollary 6.2: If $\neg v$ is a source vertex of G and a new DFS rooted at $\neg v$ does *not* visit v, then for every literal p that was reached from $\neg v$, there is no path in G from p to $\neg p$.

Proof: If there were such a path, concatenate it to the path from $\neg v$ to p, providing a path from $\neg v$ to $\neg p$. By the lemma, $\neg v$ would have been visited. ▮

Corollary 6.3: If $\neg v$ is a source vertex of G and a new DFS rooted at $\neg v$ visits p, and there is a path from p to $\neg p$ in G, then (1) $\neg p$ is visited; (2) the visit of $\neg p$ finishes before the visit of p; (3) v is visited. ▮

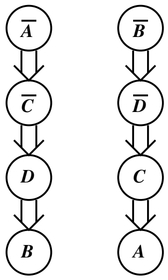
These statements permit us to weed out some vertices p for which there cannot be a path from p to $\neg p$. If the DFS rooted at $\neg v$ does not visit v, then all visited vertices can be discarded as starting points for new DFS's.

If the DFS rooted at $\neg v$ *does* visit v, then vertex p remains in consideration only if $\neg p$ was also visited and finished earlier than p. Furthermore, if $\neg p$ was discovered later than p it would have been noticed that p was gray, and the unit clause $[\neg p]$ would have been derived immediately, and all vertices reachable from $\neg p$ would also become unit clauses, as described at the beginning of this section. If p remains in consideration, then $\neg p$ can be discarded. Therefore at least half of the vertices visited from $\neg v$ can be discarded in this case.

If the DFS rooted at $\neg v$ reaches v, then $[v]$ is derived and vertices $\neg v$ and v, at least, disappear from G. Vertices p that were successors of $\neg v$ and such that $\neg p$ was also visited become new sources. The *last* of these to finish its visit in the previous DFS is the source chosen for the next DFS.

Discarding vertices is a limited benefit. We do not need to begin a DFS at a discarded vertex, but a DFS begun elsewhere might still visit the discarded vertex. A $\Theta(n^3)$ case for the strategy just described can be still be constructed, where n is the number of variables occurring in the binary clauses.

Example 6.2: To construct a bad case, put the n negative literals in four equal groups, $\overline{A}$, $\overline{B}$, $\overline{C}$, $\overline{D}$. The corresponding positive literals go in A, B, C, D. Groups $\overline{A}$ and $\overline{B}$ are sources. All vertices in one group connect to all vertices in a certain other group as shown by this picture:



There will be $n/2$ DFS's and each handles about $3n^2/64$ edges. We see that all the non-sources might be discarded after two searches, but that does not exclude them from future searches. □

7 Two-Closure from the Acyclic Implication Graph

Seeing that it is not easy to find all the derivable unit clauses, we might compute the transitive closure of G after it has been reduced to an acyclic graph by finding all the SCCs. The clause $[\neg p, q]$ is derivable if and only if there is a path from p to q. The *two-closure* of a set of binary clauses is the set of all derivable clauses. DFSS can be used, together with a *set* abstract data type that has the *union* operation.

Perform a complete DFSS on G. Each DFS will compute the set of descendants in G of the root of the DFS. During the DFS rooted at v, when edge $v \to w$ goes to a white vertex, perform a recursive DFS on w, then retrieve its descendants as d_w = `desc[w]`. When edge $v \to w$ goes to a black vertex, just retrieve d_w = `desc[w]`. At postorder time compute `desc[v]` = $\mathit{union}_w(d_w) \cup \{v\}$. The union is over all w adjacent to v. With reasonable assumptions about the data structures it is possible to compute $\mathit{union}_w(d_w)$ in time $\Theta(\sum_w |d_w|)$, but this sum can be $\Theta(n^2)$ if G has $2n$ vertices. Therefore the worst-case time for the whole two-closure is $\Theta(n^3)$.

In summary, it might be just as efficient to compute the entire two-closure, and get unit clauses as a by-product, as it is to find all derivable unit clauses. Some experimental results on random 2-CNF are reported in Section 8.

8 Experimental Results

The methods described in earlier sections are partially implemented in preprocessing simplifier programs. Experiments on random 2-CNF were conducted, with formulas ranging up to 5000 variables and clause-to-variable ratios 0.9, 1.0, and 1.1. We report only on formulas with 5000 variables, as smaller numbers do not show any interesting differences.

Original clauses	Satisfiable formulas	Simplified clauses average	Simplified clauses std.dev.	Two-closure clauses average	Two-closure clauses std.dev.	Two-closure clauses min.	Two-closure clauses max.	2clSimpL0 avg. CPU secs.
4500	197/200	4437	81	37130	8906	21628	82,002	5.93
5000	189/200	4690	257	73595	26844	22277	177,627	7.21
5500	68/200	4281	558	80553	55672	11733	259,241	7.07

Table 1: Statistics on satisfiable random 2-CNF formulas with 5000 variables. Subsequent tables refer to this set of formulas.

Although random 2-CNF formulas are unsatisfiable with probability approaching 1, for ratios greater 1.0, the behavior at 5000 variables and ratio 1.1 is not very close to the limit: 34% were satisfiable. This might serve as a warning about inferring a threshold for 3-CNF from formulas with a few hundred variables.

The simplifiers make extensive use of the implication graph, G. For a formula with n variables and m clauses, G has $2n$ vertices and $2m$ edges. By "linear" we mean proportional to $(n + m)$.

The basic simplifier program, `2clSimpL0`, first identifies sets of equivalent literals by SCC computation on the implication graph, G. Each SCC has a leader. Literals are replaced by their leaders, simplifying the formula to one that has an acyclic implication graph (which we still call G). In the second step, all derivable unit clauses are found, and the formula is further simplified, accordingly. The third step (which is not simplification!) is to derive all the implied binary clauses, which is equivalent to finding the transitive closure of the simplified acyclic implication graph. The set of given clauses plus implied clauses is called the two-closure of the formula.

Several variations on the basic program were examined. Both the second and third steps are potentially expensive, proportional to $n(n + m)$ in the worst case. The program `2clSimpX0` simply omits the third step. Other variations try to "approximate" the third step and also modify the second step.

One variation, `2clSimpX4`, tries to reduce the cost of the second step by performing DFS's in a "good" order. The order is found in stack L (before popping in phase 3), as described in Section 5. Take vertices first from the bottom of stack L to its top, then from the top to the bottom, complemented. The idea is to start searches from vertices that can reach many vertices. The order for Figure 1 would be $\neg u$, $\neg p$, $\neg t$, t, p, u. In this example, one DFS from $\neg u$ finds a path from $\neg u$ to u and also a path from $\neg p$ to p; thus both $[u]$ and $[p]$ are derived in one DFS. One complete DFSS is done using this order. Then individual DFS's are performed as necessary.

Another possibility, `2clSimpX3`, is to perform just one DFSS in a "good" order, and risk missing some derivable unit clauses.

Both `2clSimpX3` and `2clSimpX4` "approximate" the third step in linear time. Using depth-first search, they construct two arrays in linear time, called *dtime* and *ftime*, for "discovery time" and "finishing time." The *active interval* for a vertex v is from *dtime*$[v]$ to *ftime*$[v]$. If the active interval for w is contained within the active interval for v, then G has a path from v to w [BVG00, CLRS01], and the clause $[\neg v, w]$ can be derived. However, there may well be a path from v to w when the active interval for w is earlier than that for v. So this data structure gives sound, but incomplete, information about the transitive closure of G.

Table 1 shows some statistics for the satisfiable random 2-CNF formulas. Notice that the two-closed formulas have 10–20 times as many clauses as the simplified formulas. All CPU times are seconds on a Sun UltraSparc 10, 440 MHz. The CPU time for `2clSimpL0` is evidently not proportional to the size of the two-closed formula. Unsatisfiability is always detected in about 0.15 seconds.

Table 2 shows how many unit clauses are derived, as well as times, for several programs. `2clSimpL0` derives unit clauses naively, by beginning a new DFS at each literal, then performs two-closure. `2clSimpX0` derives unit clauses the same way as `2clSimpL0`, but omits the subsequent two-closure. We see that the two-closure cost is substantial, but not dominant.

`2clSimpX3` omits two-closure and additionally attempts to short-cut the derivation of unit clauses, using only one DFSS. This is the only linear-time program in the table. Although it saves some time, a large fraction of the derivable unit clauses were missed. `2clSimpX4` improves on `2clSimpX3` by deriving all possible unit clauses at very little

Original	2clSimpL0		2clSimpX0		2clSimpX3		2clSimpX4	
clauses	unit cls.	CPU	unit cls.	CPU	unit cls.	CPU	unit cls.	CPU
4500	32	5.93	32	4.50	11	2.83	32	3.22
5000	156	7.21	156	4.73	45	3.05	156	3.36
5500	599	7.07	599	4.36	188	3.10	599	3.09

Table 2: Average unit clauses derived and times for several programs.

Original	Simplified	Two-closed	Inferred	Percent inferrable	
clauses	clauses	clauses	clauses	average	std.dev.
4500	4437	37130	16063	44	7
5000	4690	73595	21052	30	6
5500	4281	80553	18481	28	9

Table 3: Number and percent of two-closure clauses that `2clSimpX4` was able to infer by active-interval containment.

additional cost, using the methods described in Section 6. By comparing with `2clSimpX0` we see that optimizing the unit-clause step saves about as much as omitting the two-closure step.

Table 3 shows how well the active-interval analysis in `2clSimpX4` was able to infer clauses in the two-closure. If the implication graph consisted entirely of out-trees and in-trees, then 100 percent of the two-closure clauses could be inferred. Although many binary clauses are missed, the methodology does increase the number of known binary clauses by a factor of about four, at very little cost. The trade-offs between time to derive clauses and reduced search space are difficult to evaluate and probably vary widely among formulas.

9 Conclusion

A strategy to make all possible equivalent-literal and unit-clause inferences without performing a complete transitive closure on the implication graph G has been described. The procedure succeeds in detecting unsatisfiability of the binary clauses, as well as all equivalent literals in optimal time. However, when the binary clauses are satisfiable, then the detection of all possible unit-clause derivations might require n independent searches in the worst case, where n is the number of variables appearing in binary clauses. Each search might cost $\Theta(|G|)$. Experiments on random 2-CNF formulas suggest that this cost is not substantial in practice.

An heuristic procedure was described for "approximating" the two-closure by constructing active intervals in linear time. Experiments on random 2-CNF formulas suggest that 30 to 40 percent of the two-closure becomes known by this approximation. However, the new clauses are only "known" in the sense that a query can be answered (in constant time) for any desired clause. It would be expensive to materialize all the clauses. Bacchus has described a new hyper-resolution strategy with one long clause and several binary clauses [Bac02]. A topic for future study is whether this strategy can be implemented efficiently by querying for binary clauses.

In the context of backtracking search, it is desirable not to begin from scratch at each search node. Incremental processing of new binary clauses remains to be explored.

Acknowledgments

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

References

[APT79] B. Aspvall, M. Plass, and R. Tarjan. A linear-time algorithm for testing the truth of certain quantified Boolean formulas. *Information Processing Letters*, 8(3):121–123, March 1979.

[Bac02] F. Bacchus. Exploring the computational tradeoff of more reasoning and less searching. In *Symposium on the Theory and Applications of Satisfiability Testing*, pages 7–16, Cincinnati, OH, 2002.

[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.

[BVG00] S. Baase and A. Van Gelder. *Computer Algorithms: Introduction to Design and Analysis*. Addison-Wesley, 3rd edition, 2000.

[CLRS01] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. McGraw-Hill, 2nd edition, 2001.

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

[DP60] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the Association for Computing Machinery*, 7:201–215, 1960.

[dV01] A. del Val. Simplifying binary propositional theories into connected components twice as fast. In *8th Int'l Conf. on Logic for Programming, Artificial Intelligence and Reasoning*, pages 389–403. Springer-Verlag, 2001.

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

[LP92] S.-J. Lee and D. A. Plaisted. Eliminating duplication with the hyper-linking strategy. *Journal of Automated Reasoning*, 9(1):25–42, August 1992.

[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.

[Sha81] M. Sharir. A strong-connectivity algorithm and its application in data flow analysis. *Computers and Mathematics with Applications*, 7(1):67–72, 1981.

[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.

[VG02a] A. Van Gelder. Toward leaner binary-clause reasoning in a satisfiability solver. In *Symposium on the Theory and Applications of Satisfiability Testing*, pages 43–53, Cincinnati, OH, 2002.

[VG02b] Allen Van Gelder. Extracting (easily) checkable proofs from a satisfiability solver that employs both preorder and postorder resolution. In *Seventh Int'l Symposium on AI and Mathematics*, Ft. Lauderdale, FL, 2002. (Also available at `ftp://ftp.cse.ucsc.edu/pub/avg/CBJ/sat-pre-post.ps.gz`.).

[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.

[ZMMM01] L. Zhang, C. Madigan, M. Moskewicz, and S. Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *ICCAD*, Nov. 2001.