Toward Leaner Binary-Clause Reasoning in a Satisfiability Solver

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

Feb. 15, 2002

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 and unit 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, which is not easily checkable.

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]. (The omission of the unit-clause rule is intentional, as explained in Section B.) 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 backjumping, and learning. These techniques are compared in a recent paper [ZMMM01]. There are substantial difficulties in combining preorder and postorder techniques, and only one prototype has been reported [VG02].

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]. Similar techniques have been reported by del Val [dV01] for detecting strongly connected components in binary clauses.

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*. We also define the *tautologous clause* $\top$, which is true under any assignment.

3 Easily Checkable Proofs

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. In Appendix B we describe how to construct a refutation as a by-product of a DPLL search.

4 Implication Graph

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, $\neg p \rightarrow q$ and $\neg q \rightarrow p$. The symbol "$\rightarrow$" can be read as "implies." 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 $\neg v$.

Sharir recognized that strongly connected components 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. 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 $(q, \neg p)$.

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 enables us to find all the edges leaving p in G^T and all the edges leaving $\neg p$ in G.

5 Strongly Connected Components of the Implication Graph

Now we describe the algorithm for SCCs adapted for binary clauses. We assume familiarity with depth-first search (DFS), with three colors used for marking vertices, as described in several modern texts [BVG00, CLRS01]. 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.

1. Initially all vertices (literals) are white. Create an empty stack S.
2. For all vertices v in any order: If v is white, perform a depth-first search in G^T, rooted at v. (This comprises a DFSS on G^T.) Push each visited vertex onto the stack S at postorder time (also called finishing time) of its visit. 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.

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, then v is identified as the *leader* of its SCC and a DFS in G is performed, rooted at v. In addition $\neg v$ is pushed on the stack L. (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.) In other cases v is simply popped and not processed; if it is white, its color remains white.

   When a DFS of G is performed 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 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 $[\neg v, p]$.

5. 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 and a DFS in G is performed, rooted at v. Binary clauses are derived as described in step 4. (This step completes a DFSS on G that began 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 $\neg v$.

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 DFS 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 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. Consider this graph, with all edges directed downward or to the right:

```
x1      x2        ...    xn
|\      |\               |\
| \     | \              | \
| (-y1)-->(-y2)--> ... --->(-yn)
|   |   |   |            |   |
y1<-|---y2<-|---   ... <-yn  |
|   |   |   |            |   |
| (-z1)<--(-z2)<-- ... <---(-zn)
|   |   |   |            |   |
z1--|-->z2--|-->   ... ->zn  |
\   |   \   |             \  |
 \  |    \  |              \ |
  -x1     -x2              -xn
```

If vertices are examined in sequence, the DFS from x_1 will visit all the z_i and $\neg y_i$ vertices, so that later DFS's from x_2, ..., x_n will not find their complements. The path to $\neg x_1$ will be found, but possibly not before $\Theta(|G|)$ work has been done. To find all the derivable unit clauses takes $\Theta(n|G|)$ work. This is the same order as performing a separate DFS from each vertex.

Breadth-first search, aborted as soon as the complement of the root vertex is found, does better on the above example, but has its own drawbacks, including the fact that there is no efficient test to see if a specific vertex (other than the root) is an ancestor.

The next example is even more extreme in that a complete DFSS of the graph beginning with DFS's rooted at x_0 and x_1 discover *no* unit-clause derivations.

```
x0      x1      x2      x3        ...    xn
|        \      |\      |\               |\
|         \     | \     | \              | \
|          ------>(-y2)-->(-y3)--> ... --->(-yn)
|               |   |   |   |            |   |
|       --------y2<-|---y3<-|---   ... <-yn  |
\      /        |   |   |   |            |   |
 \    /      -----(-z2)<--(-z3)<-- ... <---(-zn)
  \ /       /   |   |   |   |            |   |
   ------------>z2--|-->z3--|-->   ... ->zn  |
   /      /     \   |   \   |             \  |
  /      /       \  |    \  |              \ |
-x1     -x0       -x2     -x3              -xn
```

Note that breadth-first search also fails if the first two trees are rooted at x_0 and x_1.

The examples show that complete searches of G via DFSS maybe 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.

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. Everything in $\overline{A}$ connects to everything in $\overline{C}$. Everything in $\overline{B}$ connects to everything in $\overline{D}$. Everything in $\overline{C}$ connects to everything in D. Everything in $\overline{D}$ connects to everything in C. Everything in D connects to everything in B. Everything in C connects to everything in A. 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]` = $union_w(d_w) \cup \{v\}$. The union is over all w adjacent to v. With reasonable assumptions about the data structures it is possible to compute $union_w(d_w)$ 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.

8 Implementation Status

The method is not implemented yet. We hope to have it ready for the solver competition. We will test it on some large formulas from planning applications, pigeon-hole formulas, and random 3CNF formulas. The random and pigeon-hole formulas provide families of formulas with similar characteristics and varying sizes, to measure how program performance scales.

The basic program (`2cl`) [VGT96] is a DPLL-style solver with 2-closure reasoning, which means that all binary and/or unit clauses that can be derived from other binary and/or unit clauses are materialized. Equivalent-literal identification is also carried out, and variable-elimination resolution (VER) is used as well. The methods described in this paper will replace 2-closure, so that not all derivable binary clauses are materialized.

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 $\Theta(|G|)$.

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.

[BP97] S. Buss and T. Pitassi. Resolution and the weak pigeonhole principle. In *Computer Science Logic*, pages 23–29, Aarhus, Denmark, 1997. Springer-Verlag.

[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.

[BSW01] E. Ben-Sasson and A. Wigderson. Short proofs are narrow-resolution made simple. *Journal of the ACM*, 48(2):149–169, 2001.

[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.

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

[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.

[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.

[Urq87] A. Urquhart. Hard examples for resolution. *Journal of the Association for Computing Machinery*, 34(1):209–219, January 1987.

[VG02] 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.

Appendix A Notation for Appendices

Definition A.1: (strengthened formula) Let $\mathcal{A}$ be a partial assignment for formula $\mathcal{F}$. The clause $C|\mathcal{A}$, read "C strengthened by $\mathcal{A}$", and the formula $\mathcal{F}|\mathcal{A}$, read "$\mathcal{F}$ strengthened by $\mathcal{A}$", are defined as follows.

1. $C|\mathcal{A} = \top$, if C contains any literal that occurs in $\mathcal{A}$.
2. $C|\mathcal{A} = C - \{q \mid q \in C \text{ and } \neg q \in \mathcal{A}\}$, if C does not contain any literal that occurs in $\mathcal{A}$. This might be the empty clause.
3. $\mathcal{F}|\mathcal{A} = \left\{C|\mathcal{A} \;\middle|\; C \in \mathcal{F}\right\}$; i.e., apply strengthening to each clause in $\mathcal{F}$.

   Usually, occurrences of $\top$ (produced by part (1)) are deleted in $\mathcal{F}|\mathcal{A}$.

The operation $\mathcal{F}|p$ (i.e., $\mathcal{F}|\,\{[p]\}$) is sometimes called "unit simplification". □

The resolution operation is denoted by res(q, C_0, C_1); it returns the resolvent of clauses C_0 and C_1 with clashing literal q. The resolvent is $(C_0 - [q]) \cup (C_1 - [\neg q])$.

Appendix B DPLL as Construction of a Refutation

Normally, the classical branching algorithm of Davis, Putnam, Loveland, and Logemann (DPLL) [DP60, DLL62] is viewed as a backtracking search for a satisfying assignment for a Boolean CNF formula. It can be sketched as a recursive procedure with parameters $\mathcal{F}$ and $\mathcal{A}$, the formula and the assumptions (guessed assignments, represented as a set of literals):

DPLL($\mathcal{F}$, $\mathcal{A}$)

If $\mathcal{F}|\mathcal{A}$ has no clauses:
    output "sat by $\mathcal{A}$" and **terminate**.
If $\mathcal{F}|\mathcal{A}$ contains an empty clause:
    return "unsat".

(Otherwise) Choose a splitting literal q.
Call DPLL($\mathcal{F}$, $\{\mathcal{A}, \neg q\}$).
Call DPLL($\mathcal{F}$, $\{\mathcal{A}, q\}$).
Return "unsat".

The top-level call is DPLL($\mathcal{F}$, $\emptyset$). For implementation efficiency, the first parameter is normally $\mathcal{F}|\mathcal{A}$, rather than $\mathcal{F}$. Two important observations are:

1. *Unit-Clause Rule*: Note that the unit-clause rule is incorporated in the above sketch by choosing q to be a unit clause if $\mathcal{F}|\mathcal{A}$ contains any such.
2. *Pure-Literal Rule*: Note that the pure-literal rule is incorporated in the above sketch by *not* choosing q if it is a pure literal (unless all remaining variables are pure literals).

Thus the procedure incorporates all of DPLL.

If the formula is satisfiable, the pseudocode outputs and terminates, rather than returning back out of the recursion. This is not recommended for actual implementation, but it simplifies the presentation to focus on the processing of unsatisfiable formulas.

A dual view of this procedure is that it is constructing a resolution refutation. The procedure returns a *resolution tree* whose *root* contains the clause derived by the tree and whose *leaves* are clauses in $\mathcal{F}$. Resolution trees are denoted by P_0 and P_1. Each internal node contains the resolvent clause of its two children.

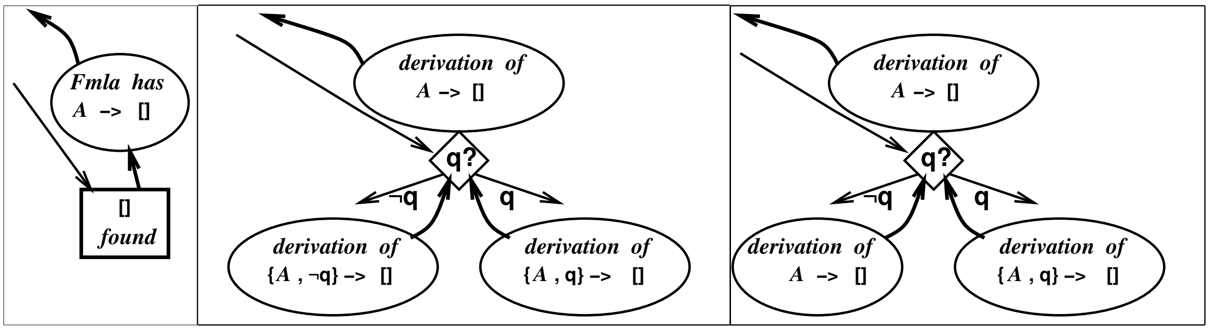

Figure 1: Extracting a resolution refutation from DPLL (as Refute). Note that the implication $\mathcal{A} \rightarrow []$ is equivalent to a disjunctive clause. In general, the antecedents are *subsets* of A, $\{A, \neg q\}$, and $\{A, q\}$, rather than the entire sets. Left panel is the base case; middle panel shows resolution, the usual case; right panel applies when a missing clashing literal prevents resolution.

Refute($\mathcal{F}$, $\mathcal{A}$)

If $\mathcal{F}|\mathcal{A}$ has no clauses:
    output "sat by $\mathcal{A}$" and **terminate**.
If $\mathcal{F}|\mathcal{A}$ contains an empty clause:
    return a clause of $\mathcal{F}$ that became empty.

(Otherwise) Choose a splitting literal q.
P_0 = Refute($\mathcal{F}$, $\{\mathcal{A}, \neg q\}$).
P_1 = Refute($\mathcal{F}$, $\{\mathcal{A}, q\}$).
Return the tree for res(q, *root*(P_0), *root*(P_1)).

We confine this discussion to $\mathcal{F}$ being unsatisfiable. The objective of Refute($\mathcal{F}$, $\mathcal{A}$) is to return the derivation of a clause C such that $\neg C \subseteq \mathcal{A}$. (We are being loose about notation, regarding $\neg C$ as a set of literals and using $\{\mathcal{A}, q\}$ to denote the addition of q to the set $\mathcal{A}$.) If Refute($\mathcal{F}$, $\mathcal{A}$) always achieves its objective, then, since $\mathcal{A}$ is empty in the top level call, the value returned to top level is a derivation of the empty clause.

It is clear that Refute($\mathcal{F}$, $\mathcal{A}$) *does* achieve its objective in the nonrecursive case, where it encounters an empty clause. Otherwise, if both recursive calls meet *their* objectives, then $\neg$*root*$(P_0) \subseteq \{\mathcal{A}, \neg q\}$ and $\neg$*root*$(P_1) \subseteq \{\mathcal{A}, q\}$. By the definition of resolution, $\neg$ res(q, *root*(P_0), *root*(P_1)) $\subseteq \mathcal{A}$.

The only gap in the above argument is that possibly *root*(P_0) does not contain q, so that resolution with q as the clashing literal is not defined. But then $\neg$*root*$(P_1) \subseteq \mathcal{A}$, so Refute($\mathcal{F}$, $\mathcal{A}$) can simply return P_0 and meet its objective. Similarly, if *root*(P_1) does not contain $\neg q$, then Refute($\mathcal{F}$, $\mathcal{A}$) can return P_1. With these added details the algorithm is correct. Figure 1 illustrates the ideas.

Now we observe that if *root*(P_0) does not contain q, then the call Refute($\mathcal{F}$, $\{\mathcal{A}, q\}$) is unnecessary. The right half of the refutation tree can be pruned and the left half becomes the whole tree. Thus conflict-directed back-jumping is built into this algorithm!

In other words, $\neg$*root*(P_0) is a conflict set for $\mathcal{F}|\{\mathcal{A}, \neg q\}$. If *root*$(P_0)$ does not contain q, then $\neg$*root*(P_0) is also a conflict set for $\mathcal{F}|\mathcal{A}$, and the assumption $\neg q$ was irrelevant to the former formula being unsatisfiable.

Appendix B.1 Incorporating Resolution on the Way Down

We now consider an enhanced version of Refute($\mathcal{F}$, $\mathcal{A}$) in which some resolution steps may be carried out prior to the recursive call. The new procedure is Refute($\mathcal{F}$, $\mathcal{D}$, $\mathcal{A}$), where $\mathcal{D}$ denotes a set of derived clauses. Also, Δ will denote a set of clauses derived after Refute($\mathcal{F}$, $\mathcal{D}$, $\mathcal{A}$) begins.

Refute($\mathcal{F}$, $\mathcal{D}$, $\mathcal{A}$)

If $\mathcal{F}|\mathcal{A}$ has no clauses:
    output "sat by $\mathcal{A}$" and **terminate**.
If $\mathcal{F}|\mathcal{A}$ contains an empty clause:
    return a clause of $\mathcal{F}$ that became empty.
If $\mathcal{D}|\mathcal{A}$ contains an empty clause:
    return a proof tree for a clause of $\mathcal{D}$ that became empty.
(Otherwise) Derive additional clauses Δ.
If $\Delta|\mathcal{A}$ contains an empty clause:
    return a proof for a clause of Δ that became empty.

(Otherwise) Choose a splitting literal q.
P_0 = Refute($\mathcal{F}$, $\{\mathcal{D}, \Delta\}$, $\{\mathcal{A}, \neg q\}$).
If q is not in *root*(P_0):
    return P_0.
(Otherwise) P_1 = Refute($\mathcal{F}$, $\{\mathcal{D}, \Delta\}$, $\{\mathcal{A}, q\}$).
If $\neg q$ is not in *root*(P_1):
    return P_1.
(Otherwise) Return the tree for res(q, *root*(P_0), *root*(P_1)).

As long as each clause in $\mathcal{D}$ or Δ is derived by resolution, its derivation can be represented by a resolution DAG (directed acyclic graph) whose edges point to earlier-derived clauses or original clauses. Such DAGs are returned in the nonrecursive case of Refute($\mathcal{F}$, $\mathcal{D}$, $\mathcal{A}$). Thus the structure returned is still a valid resolution proof, but it is not necessarily a tree.

Appendix B.2 Finding Conflict Sets in Practice

Section B.1 gives us the theoretical basis for constructing a resolution refutation of a given formula $\mathcal{F}$. Each derived clause that is returned by Refute($\mathcal{F}$, $\mathcal{D}$, $\mathcal{A}$) is a conflict set for $\{\mathcal{F}, \mathcal{D}\}\,|\mathcal{A}$. These conflict sets can be used to prune unnecessary backtracking. A conflict set $\neg P$ can be represented very nicely as a descending-order list of the depths at which the literals of $\neg P$ were assumed or guessed. Once conflict sets are materialized, they can be combined in time that is linear in their combined length.

The main problem is materializing the conflict set for the resolution DAG returned in the nonrecursive case. Notice that this is a relatively simple matter for Refute($\mathcal{F}$, $\mathcal{A}$) because the returned value is always the trivial DAG consisting of a clause of $\mathcal{F}$, say C, such that $\neg C \subseteq \mathcal{A}$. For Refute($\mathcal{F}$, $\mathcal{D}$, $\mathcal{A}$) it would not be practical to materialize every derived clause in case it happened to be useful. Instead, if D is a conceptually derived clause, only $D|\mathcal{A}$ is materialized and two pointers are maintained to the ancestor clauses that were resolved to produce $D|\mathcal{A}$.

Let us describe the *dependency DAG*, which is the skeleton of the resolution DAG, in more detail. Every assumed literal has a DAG node that is a source, and the literal is stored in this DAG node. Source nodes have no ancestors. There is also a *true source*. Clauses of the original formula have the true source as their DAG node. Each derived clause has a DAG node with two ancestors, which identify how this clause was derived; i.e., the ancestors are the DAG nodes of the clauses that were resolved to obtain this clause. In the data structures pointers are from clauses to DAG nodes and from DAG nodes to their parents; i.e., toward the sources. There is no way to discover what clause a non-source DAG node corresponds to by following pointers. However, following pointers from a certain clause *does* lead to the assumed literals that were used to derive that clause. The idea is illustrated in Figure 2.

Suppose $D|\mathcal{A}$ is the empty clause. To reconstruct D it suffices to traverse the DAG rooted at $D|\mathcal{A}$ and collect all the reachable assumptions. This is accomplished efficiently with depth-first search. (Efficiently does not mean inexpensively.) The justification is that every literal that occurs in a clause (not an assumption) that participated in the derivation of D either survives in D or was resolved away during the derivation. If $\neg q$ is in $\mathcal{A}$ and is reached in the DAG, then q must be in an ancestor-clause of D and also in D. In fact, the ancestors collected comprise a conflict set for $\{\mathcal{F}, \mathcal{D}\}\,|\mathcal{A}$.

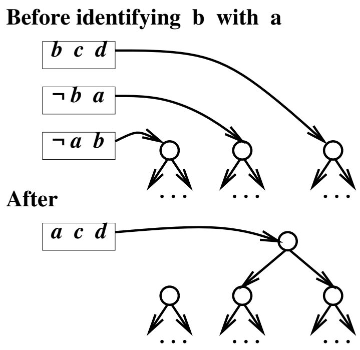

Figure 2: Resolution-DAG nodes need to be separate from their clauses because the clauses can disappear.

To include equivalent-literal identification in the reasoning scheme we proceed as follows. To replace b by a after $a = b$ has been derived, an old clause, say $[b, c, d]$ becomes a new clause $[a, c, d]$. The two pointers for $[a, c, d]$ point to $[b, c, d]$ and $[\neg b, a]$, as shown in Figure 2. The latter must exist because we have no way to derive $a = b$ without having that clause *if we limit our reasoning operations to resolution*. This is the method we implemented. If more sophisticated equivalent-literal identification schemes are used, then more complicated data structures might be needed to track the relevant assumptions.