Combining Preorder and Postorder Resolution in a Satisfiability Solver

Allen Van Gelder

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

Abstract

Recently, the classical backtracking-search satisfiability algorithm of Davis, Putnam, Loveland and Logemann (DPLL) has been enhanced in two distinct directions: with more advanced reasoning methods, or with conflict-directed back-jumping (CBJ). Previous methods used one idea or the other, but not both (except that CBJ has been combined with the unit-clause rule). The difficulty is that more advanced reasoning derives new clauses, but does not identify which backtrackable assumptions were relevant for the derivation. However CBJ needs this information. The linear-time decision procedure for binary-clause formulas is a good example of this phenomenon.

This paper describes an efficient method to integrate several popular reasoning methods with CBJ. It includes binary-clause reasoning, equivalent-literal identification, variable-elimination resolution, and others. The main idea is to exploit the fact that DPLL can be viewed as an algorithm to construct a resolution refutation, rather than as backtracking-search for a model. Conflict sets in CBJ correspond to clauses in the resolution refutation. Incorporating advanced reasoning into this dual view of DPLL shows the way to combine it soundly with CBJ.

Experiments show that the search space is reduced, but that memory has to be managed extremely carefully.

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.

Conflict-directed back-jumping (CBJ) has been implemented recently in several satisfiability solvers [SS96, Zha97, BS97]. These 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 idea of CBJ is to avoid trying the opposite assignment to a variable when the first assignment did not contribute to the formula becoming *false*. The variables whose assignments *do* contribute to the formula becoming *false* are termed a *conflict set*.

A different way to reduce the amount of searching is to use reasoning operations before branching on a variable assignment. The unit-clause rule is a classical example. Several methods have been reported that implement binary-clause reasoning, as well [BS92, Pre96, VGT96]. Because this reasoning occurs prior to a branching step, we call it *preorder reasoning*. While experience has shown that these methods greatly reduce the search space, the reasoning introduces a substantial overhead per search step, and the trade-off is not always beneficial.

This paper describes a method to combine CBJ with resolution-based reasoning, which includes equivalent-literal identification. The difficulty to be overcome is that derived clauses depend on the assumptions (guessed assignments) in a nontransparent way. If the only reasoning is unit resolution, then each derived clause corresponds to one original clause, and the relevant assumptions can be traced through this association [LP92, SS96, VGO99]. With non-unit resolution, a derived clause may be associated with an arbitrary number of original clauses. The procedure constructs a directed acyclic graph to maintain the association.

A key to understanding the correctness of the procedure is that the DPLL algorithm can be viewed dually as a procedure to construct a resolution refutation. The refutation is constructed in a post-order fashion. Conflict-directed back-jumping falls out naturally from this dual view. This view is then enhanced with resolutions performed during the search.

Implementation of the method in C posed challenges in memory management. The auxiliary data structure for the directed acyclic graph can be built with only a constant amount of overhead per operation, but during backtracking large amounts of the structure become garbage. Preliminary experimental results are reported.

1.1 Relationship to Constraint Satisfaction

Constraint satisfaction problems (CSP) have been studied extensively in the recent literature, where variables range over finite domains, which are usually non-boolean [Gin93, Pro93, Pro95, BR96, Bes99, BMFL99, CGU00] (see also bibliographies in these papers). While their are some general similarities between the problems and recent techniques in constraint satisfaction and those in satisfiability testing, when we look closely, there are significant differences. For example, most work in constraint satisfaction is restricted to constraints on two variables. In the satisfiability arena, this corresponds to binary clauses, which by themselves are easy. Also, one of the techniques for reducing search is maintaining arc consistency (MAC). Since they work on binary constraints, MAC procedures have a superficial resemblance to resolution on binary clauses. In actuality, MAC cannot detect unsatisfiability in binary clauses alone, and is more akin to unit-clause resolution.

Previous satisfiability testers that use CBJ limit their preorder reasoning to unit-clause resolution [SS96, Zha97, BS97] and thus are closer to a combination of MAC + CBJ than the program reported here, which relies heavily on binary-clause resolution and related operations for its preorder reasoning.

Bessiere and Regin have observed empirically that CBJ is often of little value in combination with MAC [BR96]. Due to the relationship of unit-clause resolution to MAC, we might expect that this observation would carry over to satisfiability testing. However, based on the satisfiability papers cited above, this appears not to be the case: CBJ was found to be extremely helpful.

Our purpose is to investigate further the effect of CBJ in a satisfiability tester with more powerful preorder reasoning than just unit-clause resolution. The first step is to overcome the implementation difficulties. This paper reports on progress in that direction.

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 $\neg x$. 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.

Definition 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) $\mathcal{F}|\mathcal{A} = \{C|\mathcal{A} \mid C \in \mathcal{F}\}$; 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 $\mathrm{res}(q, C_0, C_1)$; it returns the resolvent of clauses C_0 and C_1 with clashing literal q. The resolvent is $(C_0-[q]) \cup (C_1-[\neg q])$.

3 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(𝓕, 𝓐)
--------------------------------------------
If 𝓕|𝓐 has no clauses:
    output "sat by 𝓐" and terminate.
If 𝓕|𝓐 contains an empty clause:
    return "unsat".

(Otherwise) Choose a splitting literal q.
Call DPLL(𝓕, {𝓐, ¬q}).
Call DPLL(𝓕, {𝓐, q}).
Return "unsat".
--------------------------------------------
```

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

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.

```
Refute(𝓕, 𝓐)
--------------------------------------------
If 𝓕|𝓐 has no clauses:
    output "sat by 𝓐" and terminate.
If 𝓕|𝓐 contains an empty clause:
    return a clause of 𝓕 that became empty.

(Otherwise) Choose a splitting literal q.
P0 = Refute(𝓕, {𝓐, ¬q}).
P1 = Refute(𝓕, {𝓐, q}).
Return the tree for res(q, root(P0), root(P1)).
--------------------------------------------
```

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

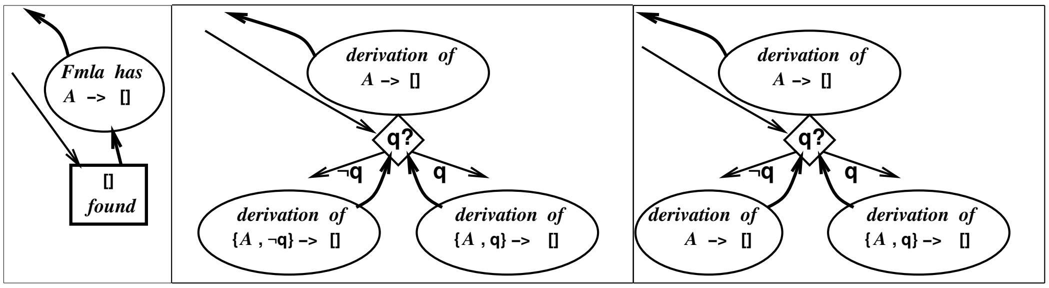

Fig. 1. Extracting a resolution refutation from DPLL (as Refute). Note that the implication $\mathcal{A} \rightarrow []$ is equivalent to a disjunctive clause. In general, the antecedents are *subsets* of A, $\{A, \neg q\}$, and $\{A, q\}$, rather than the entire sets. Left panel is the base case; middle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

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.

4 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.

The program `2cl` has extensive preorder reasoning, much of it based on binary clauses, as previously reported [VGT96]. Its performance has been improved recently through more efficient data structures. The next section describes a significant new feature that has been incorporated since the cited paper.

5 Variable-Elimination Resolution

The operation that we call Variable-Elimination Resolution (VER) was used in the DavisPutnam60 Davis-Putnam *resolution* procedure, where it was called "elimination of an atom" [DP60]. If x is the variable to be eliminated, all resolutions involving clauses containing x or $\neg x$ are performed; then all clauses containing x or $\neg x$ are eliminated from the new formula. It is easy to show that $\mathcal{F}$ is satisfiable if and only if VER$(x, \mathcal{F})$ is satisfiable, for any choice of x. Note that the pure literal rule is a special case of VER in which the set of resolvents is empty.

The program `2clVER` is `2cl` enhanced with VER. VER is applied according to an heuristic formula that evaluates the trade-off between reducing the number of variables and increasing the overall formula length. At the point where a variable would be selected to branch on, `2clVER` considers both branching and performing VER. Both choices lead to eliminating one free variable from the formula. Branching results in two subproblems with shorter formulas in each. VER results in one subproblem, often with a longer formula.

The trade-off is estimated with a simple cost model. For formulas with the same number of variables, we assume the cost of solution is proportional to $e^{\alpha L}$, where L is formula length. The estimate uses these parameters, where q is the literal that would be selected for branching, and x is the literal that would be selected for VER.

R = increase in formula length after VER on x;
P = decrease in formula length after binding q to be positive;
N = decrease in formula length after binding q to be negative.

Increases and decreases are estimations and R might be negative. VER should be more efficient than branching when $e^{\alpha(L+R)} < e^{\alpha(L-P)} + e^{\alpha(L-N)}$. After dividing out $e^{\alpha L}$, we assume the exponents are small and use one term of Taylor expansion, rather than evaluate transcendental functions: $1 + \alpha R < 1 - \alpha P + 1 - \alpha N$. Finally, the rule is:

$$\text{Select VER when } R < (1/\alpha) - (P + N).$$

Based on experience, the default value of α is 0.0039 and $1/\alpha = 256$. This means the program chooses to let the formula grow by some 200 literals rather than split it into two subproblems. The program user can change the value on the command line.

6 Finding Conflict Sets in Practice

Section 4 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 the literal is stored in this DAG node. Source nodes have no ancestors.

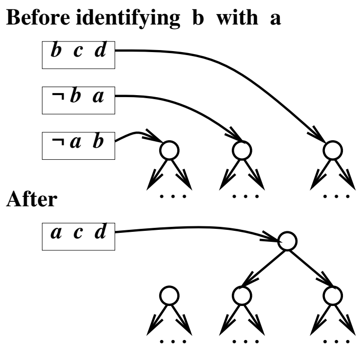

Fig. 2. Resolution-DAG nodes need to be separate from their clauses because the clauses can disappear.

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 nonsource 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 $\{\mathcal{F}, \mathcal{D}\}\,|\mathcal{A}$.

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.

7 Asserting Postorder Lemmas

If S is a conflict set derived as in the previous sections (regarded as a conjunction of literals), then $\neg S$ is a clause that has been derived from $\mathcal{F}$ and may be added to $\mathcal{F}$ without changing the set of satisfying assignments for $\mathcal{F}$. We call $\neg S$ a *postorder lemma* to distinguish it from clauses derived while the search is "going forward". In principle, conflict sets can be used for conflict-directed back-jumping whether or not their clause is asserted permanently; whether it is practical to do so might depend on the implementation.

The reported satisfiability solvers that use CBJ and have been successful on large problems all have the capability to assert postorder lemmas [SS96, Zha97, BS97]. The policy for asserting these lemmas is an heuristic under some control by the user. The program reported in this paper does not yet have that capability implemented, but we plan to implement it in the near future.

8 Experimental Results

We tested a preliminary implementation of the method described in this paper on some random 3CNF formulas, pigeon-hole formulas, and large formulas from circuit-fault detection applications and planning applications. The random and pigeon-hole formulas provide families of formulas with similar characteristics and varying sizes, to measure how program performance scales. These experiments are in the nature of a proof-of-concept because, as mentioned in the previous section, the program does not yet have the ability to assert postorder lemmas. Thus these results indicate what is achievable with CBJ alone.

All CPU times are based on a Sun Ultra 10 with 440 MHz clock; times are seconds unless indicated otherwise. Tables use abbreviations as follows:

br	branches (i.e., guessed variable assignments, first and alternate)
pr	alternate branches pruned by CBJ (after first guess was unsat)
k	thousand

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. The program now has an option to apply variable-elimination resolution (VER), described in Section 5. Our main question is how CBJ affects the performance of these two programs.

For reference purposes we also tested several other programs that use CBJ: Sato3.2.1 [Zha97], Relsat2 [BS97], and Chaff2 (Moscewicz *et al.*, this workshop). For sato we used the default settings; for relsat we used learn-order 4 and fudge-factor 0.9, as recommended; for chaff we used the default settings, except on the random formulas, where we used alternative settings suggested by one of the program authors. The somewhat older version of Grasp [SS96] that we have ran considerably slower. All of the programs have several runtime parameters and it is possible that different parameter setting would produce much different results.

Figure 3 shows the growth rates on random 3CNF formulas with clause-

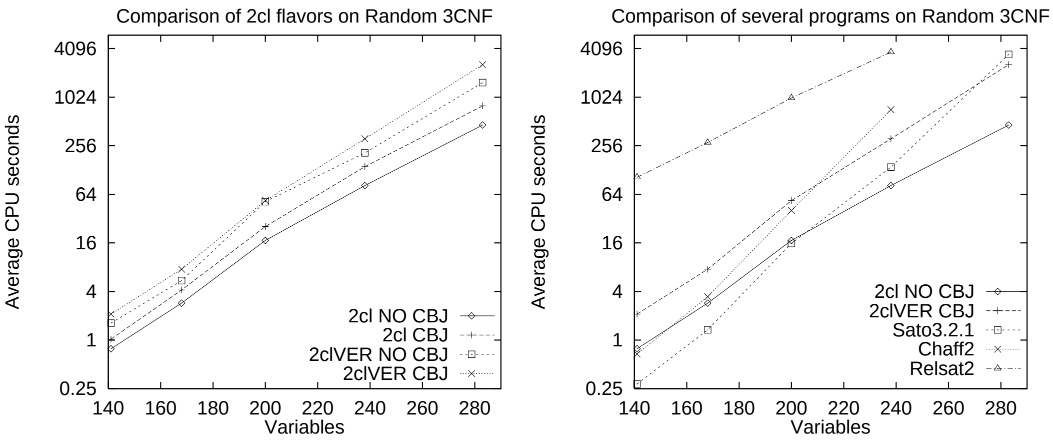

Fig. 3. Growth rates for CPU time on 2cl flavors (left) and for various programs with CBJ (right).

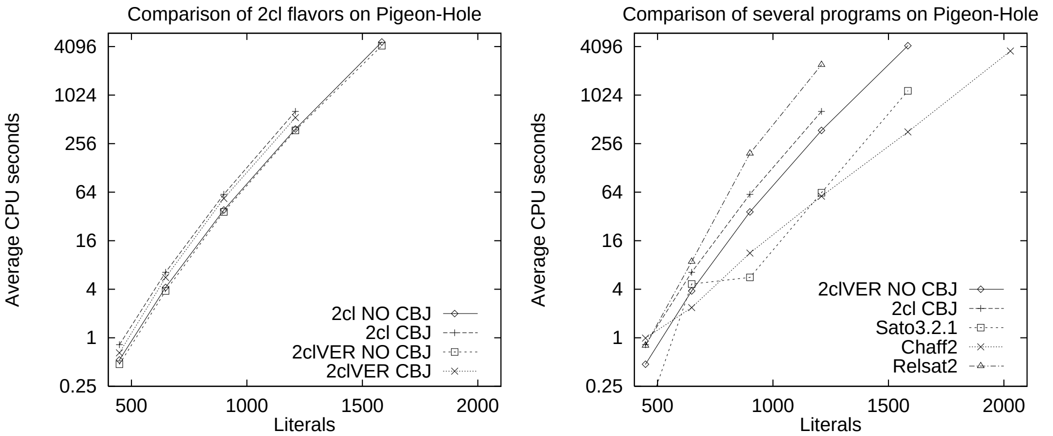

Fig. 4. Growth rates for CPU time on 2cl flavors (left) and for various programs with CBJ (right).

variable ratio of 4.27, which is believed to be close to the hardest ratio. On semi-log scales, straight lines with slope α represent functions proportional to 2^α. On the left of the figure we see that both variable-elimination resolution (VER) and CBJ increase the constant factor for 2cl but do not noticeably change the growth rate. On the right we plot the best and worst versions of 2cl, together with sato3.2.1, relsat2, and chaff2. We observe that both sato and chaff are much faster for small formula sizes but grow considerably faster than 2cl and relsat.

Figure 4 shows the growth rates on pigeon-hole formulas with 8 to 13 pigeons, again on semi-log scales. On the left of the figure we see that CBJ increases the constant factor for 2cl but do not noticeably change the growth rate. VER helps slightly on the constant factor. On the right we plot the best and worst versions of 2cl, together with sato3.2.1, relsat2, and chaff2. We observe that both sato and relsat are growing somewhat faster than 2cl. However, chaff is growing much slower than the other programs, unlike its behavior on random formulas. Chaff is the only program that finished 13 pigeons in the

Table 1
Performance of `2cl` and `2clVER` on 325 circuit fault-detection formulas available from DIMACS, of types bridge-fault (bf) and single-stuck-at (ssa). Entries are totals for the groups.

Group/		2cl		2clVER		sato	relsat
how many		no CBJ	CBJ	no CBJ	CBJ	3.2.1	2
bf	br's	10112k	4799k	51k	21k		
223	pr's	—	92k	—	0.9k		
	CPU	4570	3500	116	83	246	41
ssa	br's	6223k	4446k	51k	44k		
102	pr's	—	66k	—	0.5k		
	CPU	2261	2061	29	35	466	53

time available.

So far, CBJ has only hindered 2cl. We mention again that it is not fully implemented yet. However, the partial implementation does show some benefits on some formulas based on applications.

Table 1 shows performance of four configurations of `2cl` on 325 circuit fault-detection formulas. For reference purposes we compare `2clVER` with CBJ to some other programs using CBJ in the two righmost columns of Table 1. We observe that CBJ produces about the same overall improvement factor for 2cl whether or not VER is also employed. We also notice that the bf group benefited more from CBJ than did the ssa group. We believe this is due to the fact that the ssa group has a preponderance of satisfiable formulas, whereas the bf group is mainly unsatisfiable.

Table 2 shows performance of four configurations of `2cl` on four planning problems generated by Satplan, a "SAT compiler" developed by Kautz and Selman [KS96]. We observe that CBJ gains by a substantial factor on one formula and loses by substantial factors on two others, as measured by CPU time. The fourth formula was easy for all configurations. The rightmost three columns show Sato, Relsat, and Chaff (presented at this workshop).

In terms of branching, CBJ reduced the branching by a factor of 10 on the one formula for which it produced a time gain, but it had minor effects on the branching in the other formulas.

We further analyzed the time in 2clVER that is attributable to CBJ for bw_large.c-13. The CBJ effort can be divided into two phases. In phase 1 the dependency DAG is constructed. This occurs at preorder time, i.e., on the way down the search tree. Phase 2 occurs at postorder time. At a leaf in the search tree the DAG is traversed from that leaf node to discover what variable assignments were relevant; these literals constitute the conflict set for the search leaf. These costs are incurred even though CBJ might yield little or no reduction in the search space, as was the case for this formula. To separate the costs of phase 1 and phase 2 we ran 2clVER with phase 1 enabled and phase 2 disabled. The time was 209, compared to 133 with phase 1 also

Table 2
Performance of `2cl` and `2clVER` on four Satplan formulas. The versions with shorter deadlines are unsatisfiable; those with longer deadlines are satisfiable. Formula sizes are after simplification.

Problem-Deadline/ Variables, Clauses		2cl no CBJ	2cl CBJ	2clVER no CBJ	2clVER CBJ	sato 3.2.1	relsat 2	chaff 2
logistics.c-12	br's	1111k	131k	8k	0.8k			
1482, 6974	pr's	—	22k	—	0.2k			
	CPU	2204	420	19	4	5	1	0.3
logistics.c-13	br's	0.1k	0.1k	0.1k	0.1k			
1596, 8178	pr's	—	0	—	0			
	CPU	0	0	0	0	0	1	0.2
bw_large.c-13	br's	0.3k	0.3k	0.3k	0.3k			
4405, 29280	pr's	—	6	—	6			
	CPU	133	205	133	203	3	9	1.2
bw_large.c-14	br's	1.7k	1.7k	1.7k	1.7k			
4723, 34369	pr's	—	10	—	10			
	CPU	634	976	502	976	8	18	2.5

disabled. Therefore it appears that the overhead of recording the dependency DAG is about 50 percent.

On the other hand, phase 2 saved a net 6 seconds even though it seemed to have a modest effect on the overall search. With CBJ the number of branches decreased from 268 to 260, due to 6 right branches being pruned. Each internal node of the search tree normally produces two branches and there were 134 internal nodes in this case. A phase-2 computation occurs at each internal node. The maximum depth in the search tree is only 16. This limits the size of each conflict set to 16. The small size of the search space probably explains why phase 2 had a negligible cost for this formula.

Again we compare `2clVER` with CBJ to some other programs using CBJ in the two rightmost columns of Table 2. This shows that bw_large.c-13 and bw_large.c-14 are not hard objectively. Indeed, humans can solve these problems (expressed in blocks-world language, not CNF!) in a few minutes. Thus the poor performance of CBJ in 2cl might be more related to how 2cl conducts its underlying computation than any properties of CBJ itself. At a branching rate on the order of once per second, a large amount of time was spent on preorder reasoning that was unproductive.

9 Conclusion

We have described a method to combine CBJ with resolution-based reasoning, as enhancements to the basic DPLL procedure. The method includes equivalent-literal identification and variable-elimination resolution, besides binary-clause resolution. Preliminary experiments indicate that the idea is feasible. Future work should implement a capability to assert postorder lemmas permanently, according to some "value" heuristic. (Lemmas with many literals are less likely to be useful and incur more overhead.) Then more extensive experimental studies should be undertaken.

Acknowledgments

This work was supported in part by NSF grants CCR-8958590 and CCR-9503830, by equipment donations from Sun Microsystems, Inc., and software donations from Quintus Computer Systems, Inc.

References

[Bes99] C. Bessiere. Non-binary constraints. In *Constraint Programming 99*, pages 24–27, 1999. Springer-Verlag.

[BMFL99] C. Bessiere, P. Meseguer, E. C. Freuder, and J. Larrosa. On forward checking for non-binary constraint satisfaction. In *Constraint Programming 99*, pages 88–102, Alexandria, VA, 1999. Springer-Verlag.

[BR96] C. Bessiere and J-C. Regin. MAC and combined heuristics: Two reasons to forsake FC (and CBJ?) on hard problems. In *Constraint Programming 96*, pages 61–75, Cambridge, MA, 1996. Springer-Verlag.

[BS92] A. Billionnet and A. Sutter. An efficient algorithm for the 3-satisfiability problem. *Op. Res. Let.*, 12:29–36, 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.

[CGU00] J. L. Caldwell, I. P. Gent, and J. Underwood. Search algorithms in type theory. *Theoretical Computer Science*, 232(1–2):55–90, 2000.

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

[DP60] M. Davis and H. Putnam. A computing procedure for quantification theory. *JACM*, 7:201–215, 1960.

[Gin93] M. L. Ginsberg. Dynamic backtracking. *JAIR*, 1:25–46, 1993.

[KS96] H. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *13th National Conference on Artificial Intelligence*, pages 1194–1201, 1996.

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

[Pre96] D. Pretolani. Efficiency and stability of hypergraph SAT algorithms. In *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge.*, Am. Math. Soc., 1996.

[Pro93] P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Comput. Intelligence*, 9:268–299, 1993.

[Pro95] P. Prosser. MAC-CBJ: Maintaining arc consistency with conflict-directed backjumping. TR 95/177, U. Strathclyde, Glasgow, 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.

[VGO99] A. Van Gelder and F. Okushi. Lemma and cut strategies for propositional model elimination. *Annals of Mathematics and Artificial Intelligence*, 26(1–4):113–132, 1999.

[VGT96] A. Van Gelder and Y. K. Tsuji. Satisfiability testing with more reasoning and less guessing. In *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge.*, Am. Math. Soc., 1996. (also at `ftp://ftp.cse.ucsc.edu/pub/avg/kclose-tr.ps.gz`).

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