Using CSP Look-Back Techniques to Solve Real-World SAT Instances

Roberto J. Bayardo Jr.
The University of Texas at Austin
Department of Computer Sciences (C0500)
Austin, TX 78712 USA
bayardo@cs.utexas.edu
http://www.cs.utexas.edu/users/bayardo

Abstract
We report on the performance of an enhanced version of the "Davis-Putnam" (DP) proof procedure for propositional satisfiability (SAT) on large instances derived from real-world problems in planning, scheduling, and circuit diagnosis and synthesis. Our results show that incorporating CSP look-back techniques -- especially the relatively new technique of relevance-bounded learning -- renders easy many problems which otherwise are beyond DP's reach. Frequently they make DP, a systematic algorithm, perform as well or better than stochastic SAT algorithms such as GSAT or WSAT. We recommend that such techniques be included as options in implementations of DP, just as they are in systematic algorithms for the more general constraint satisfaction problem.

Introduction
While CNF propositional satisfiability (SAT) is a specific kind constraint satisfaction problem (CSP), until recently there has been little application of popular CSP look-back techniques in SAT algorithms. In previous work [Bayardo & Schrag 96] we demonstrated that a look-back-enhanced version of the Tableau algorithm for 3SAT instances [Crawford and Auton 96] can solve easily many instances which without look-back are "exceptionally hard" -- orders of magnitude harder than other instances with the same surface characteristics. In this work the instances were artificially generated. Here, we demonstrate the practical utility of CSP look-back techniques by using a look-back-enhanced algorithm related to Tableau to solve large SAT instances derived from real-world problems in planning, scheduling, and circuit diagnosis and synthesis. Kautz and Selman [96] had found unenhanced Tableau inadequate to solve several planning-derived instances and resorted to using a stochastic algorithm, WSAT (also known as WalkSAT) [Selman et al. 94]; our results show that look-back enhancements make this recourse unnecessary.

Given the usual framework of backtrack search for systematic solution of the finite-domained constraint satisfaction problem (CSP), techniques intended to improve efficiency can be divided into two classes: look-ahead techniques, which exploit information about the remaining search space, and look-back techniques, which exploit information about search which has already taken place. The former class includes variable ordering heuristics, value ordering heuristics, and dynamic consistency enforce-

Stochastic algorithms outperform systematic ones dramatically on satisfiable instances from the phase transition region of random problem spaces, such as Random 3SAT [Selman et al. 92]. Instances in this region are on average most difficult for widely differing algorithms; they have come to be used frequently as benchmarks for SAT algorithm performance. At the same time, it is widely recognized that they have very different underlying structures from SAT instances one would expect to arise naturally in real-world problems of interest.

Stochastic algorithms also outperform systematic algorithms such as Tableau on some real-world problems. Several SAT-encoded planning problems described by Kautz and Selman [96] are infeasible for Tableau (given 10 hours) but solved easily by WSAT (given around 10 minutes). Our
look-back enhanced version of DP is competitive with WSAT in identifying feasible plans using the same instances. Furthermore, look-back-enhanced DP proves the nonexistence of shorter plans in 1 to 3 minutes on instances which Tableau did not solve in 10 hours; this task is impossible for WSAT because of its incompleteness. The innovative work of Kautz and Selman [96] was "pushing the envelope" of feasibility for planning problems; this lays a foundation where our look-back-enhanced DP slips in neatly as a key component in a planning system at the state of the art.

Definitions
A propositional logic variable ranges over the domain \{true, false\}. An assignment is a mapping of these values to variables. A literal is the occurrence of a variable, e.g., \(x\), or its negation, e.g., \(\neg x\); a positive literal \(x\) is satisfied when the variable \(x\) is assigned true, and a negative literal \(\neg x\) is satisfied when \(x\) is assigned false. A clause is a simple disjunction of literals, e.g., \((x \lor y \lor \neg z)\); a clause is satisfied when one or more of its literals is satisfied. A unit clause contains exactly one variable, and a binary clause contains exactly two. The empty clause \((\)\) signals a contradiction (seen in the interpretation, "choose one or more literals to be true from among none"). A conjunctive normal form (CNF) is a conjunction of clauses (e.g., \((a \lor b \lor \neg c)\)); a CNF is satisfied if all of its clauses are satisfied.

For a given CNF, we represent an assignment notationally as a set of literals each of which is satisfied. A nogood is a partial assignment which will not satisfy a given CNF. The clause \((a \lor b \lor \neg c)\) encodes the nogood \(\{\neg a, -b, c\}\). We call such a nogood-encoding clause a reason. Resolution is the operation of combining two input clauses mentioning a given literal and its negation, respectively, deriving an implied clause which mentions all other literals besides these. For example, \((a \lor \neg b)\) resolves with \((b \lor c)\) to produce \((a \lor c)\).

Basic Algorithm Description
The Davis-Putnam proof procedure (DP) is represented below in pseudo-code. As classically stated, SAT is a decision problem, though frequently we also are interested in exhibiting a satisfying truth assignment \(\sigma\), which is empty upon initial top-level entry to the recursive, call-by-value procedure DP.

\[
\text{DP}(F, \sigma) = \\
\text{UNIT-PROPAGATE}(F, \sigma) \\
\text{if } ( ) \text{ in } F \text{ then return} \\
\text{if } F = \emptyset \text{ then exit-with}(\sigma) \\
\alpha \leftarrow \text{SELECT-BRANCH-VARIABLE}(F) \\
\text{DP}(F \cup \{\alpha\}, \sigma \cup \{\alpha\}) \\
\text{DP}(F \cup \{-\alpha\}, \sigma \cup \{-\alpha\}) \\
\text{return}
\]

The CNF \(F\) and the truth assignment \(\sigma\) are modified in calls by name to UNIT-PROPAGATE. If \(F\) contains a contradiction, this is failure and backtracking is necessary. If all of its clauses have been simplified away, then the current assignment satisfies the CNF. SELECT-BRANCH-VARIABLE is a heuristic function returning the next variable to value in the developing search tree. If neither truth value works, this also is failure.

\[
\text{UNIT-PROPAGATE}(F, \sigma) = \\
\text{while} (\exists \omega \in F \text{ where } \omega = (\lambda)) \\
\sigma \leftarrow \sigma \cup \{\lambda\} \\
F \leftarrow \text{SIMPLIFY}(F)
\]

UNIT PROPAGATE adds the single literal \(\lambda\) from a unit clause \(\omega\) to the literal set \(\sigma\), then it simplifies the CNF by removing any clauses in which lambda occurs, and shortening any clauses in which \(\neg \lambda\) occurs through resolution.

Modern variants of DP including POSIT and Tableau incorporate highly optimized unit propagators and sophisticated branch-variable selection heuristics. The branch-variable selection heuristic used by our implementation is inspired by the heuristics of POSIT and Tableau, though is somewhat simpler to reduce implementation burdens.

Details of branch-variable selection are as follows. If there are no binary clauses, select a branch variable at random. Otherwise, assign each variable \(\gamma\) appearing in some binary clause a score of \(\neg \gamma \cdot \text{pos}(\gamma) + \neg \gamma \cdot \text{pos}(\gamma)\) where \(\text{pos}(\gamma)\) and \(\neg \gamma\) are the numbers of occurrences of \(\gamma\) and \(\neg \gamma\) in all binary clauses, respectively. Gather all variables within 20% of the best score into a candidate set. If there are more than 10 candidates, remove variables at random until there are exactly 10. If there is only one candidate, return it as the branch variable. Otherwise, each candidate is re-scored as follows. For a candidate \(\gamma\), compute \(\text{pos}(\gamma)\) and \(\neg \gamma\) as the number of variables valued by UNIT-PROPAGATE after making the assignment \(\{\gamma\}\) and \(\{-\gamma\}\) respectively. Should either unit propagation lead to a contradiction, immediately return \(\gamma\) as the next branch variable and pursue the assignment for this variable which led to the contradiction. Otherwise, score \(\gamma\) using the same function as above. Should every candidate be scored without finding a contradiction, select a branch variable at random from those candidates within 10% of the best (newly computed) score.

Except in the cases of contradiction noted above, the truth value first assigned to a branch variable is selected at random. We have applied the described randomizations only where additional heuristics were not found to substantially improve performance across several instances.

Incorporating CBJ and Learning
The pseudo-code version of DP above performs naive backtracking mediated by the recursive function stack. Conflict directed backjumping (CBJ) [Prosser 93] backs up through this abstract stack in a non-sequential manner, skipping stack frames where possible for efficiency's sake. Its mechanics involve examining assignments made by UNIT-PROPAGATE, not just assignments to DP branch variables, so it is more complicated than the DP pseudo-code would represent. We forego CBJ pseudo-code in this short paper.
We implement CBJ by having UNIT-PROPAGATE maintain a pointer to the clause in the (unsimplified) input CNF which serves as the reason for excluding a particular assignment from consideration. For instance, when \( \{ -a, -b \} \) is part of the current assignment, the input clause \( (a \lor b \lor \neg x) \) is the reason for excluding the assignment \( \{ \neg x \} \). Whenever a contradiction is derived, we know some variable has both truth values excluded. CBJ constructs a working reason \( C \) for this failure by resolving the two respective assignments; then it back up to the most recently assigned variable \( x \). Other-\( \ldots \) 

We use three separate test suites to compare the performance of look-back enhanced DP with other algorithms whose performance has been reported for the same instances: SAT-encoded planning instances from Kautz and Selman\(^1\); selected circuit diagnosis and planning instances from the DIMACS Challenge directory associated with the 1993 SAT competition\(^2\); and planning, scheduling, and circuit synthesis instances from the 1996 Beijing SAT competition\(^3\).

**TABLE 1. Kautz and Selman’s planning instances.**

<table>
<thead>
<tr>
<th>instance</th>
<th>vars</th>
<th>clauses</th>
<th>sat</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>log_gp.b</td>
<td>2,069</td>
<td>29,508</td>
<td>Y</td>
<td>planning</td>
</tr>
<tr>
<td>log_gp.c</td>
<td>2,809</td>
<td>48,920</td>
<td>Y</td>
<td>planning</td>
</tr>
<tr>
<td>log_dir.a</td>
<td>828</td>
<td>6,718</td>
<td>Y</td>
<td>planning</td>
</tr>
<tr>
<td>log_dir.b</td>
<td>843</td>
<td>7,301</td>
<td>Y</td>
<td>planning</td>
</tr>
<tr>
<td>log_dir.c</td>
<td>1,141</td>
<td>10,719</td>
<td>Y</td>
<td>planning</td>
</tr>
<tr>
<td>log_un.b</td>
<td>1,729</td>
<td>21,943</td>
<td>N</td>
<td>planning</td>
</tr>
<tr>
<td>log_un.c</td>
<td>2,353</td>
<td>37,121</td>
<td>N</td>
<td>planning</td>
</tr>
<tr>
<td>hw_dir.c</td>
<td>3,016</td>
<td>50,457</td>
<td>Y</td>
<td>planning</td>
</tr>
<tr>
<td>hw_dir.d</td>
<td>6,325</td>
<td>131,973</td>
<td>Y</td>
<td>planning</td>
</tr>
</tbody>
</table>

Selected SAT-encoding planning instances constructed by Kautz and Selman\(^6\) (the hardest of these instances which were available to us) are listed in Table 1. The “log” instances correspond to planning problems in logistics; the “bw” instances are for blocks worlds -- not “real” worlds -- but they are nonetheless hard. The “gp” instances are Graphplan encodings, the “dir” instances direct encodings (state-based for the logistics instances, linear for blocks world), and the “un” instances are unsatisfiable Graphplan encodings used to demonstrate the infeasibility of shorter plans. (See cited paper for more details.)

**TABLE 2. DIMACS instances.**

<table>
<thead>
<tr>
<th>instance</th>
<th>vars</th>
<th>clauses</th>
<th>sat</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ssa2670-141</td>
<td>986</td>
<td>2,315</td>
<td>N</td>
<td>diagnosis</td>
</tr>
<tr>
<td>bf1355-075</td>
<td>2,180</td>
<td>6,778</td>
<td>N</td>
<td>diagnosis</td>
</tr>
<tr>
<td>hanoi4</td>
<td>718</td>
<td>4,932</td>
<td>Y</td>
<td>planning</td>
</tr>
<tr>
<td>hanoi5</td>
<td>1,931</td>
<td>14,468</td>
<td>Y</td>
<td>planning</td>
</tr>
</tbody>
</table>

In the DIMACS suite, we looked at Van Gelder and Tsuji’s “ssa” (single-stuck-at) and “bf” (bridge-fault) circuit diagnosis instances, and Selman’s tower of hanoi planning instances also using linear encoding. For brevity, we report on only the hardest, for all algorithms investigated, of the single-stuck-at and bridge-fault instances (shown in Table 2).

We report on all instances in the Beijing suite, shown in Table 3. The planning instances (“blocks”) again use the linear encodings. The scheduling instances (“e”) encode Sadeh’s benchmarks as described in [Crawford and Baker 94]. The circuit synthesis instances (“bit”) were contributed by Bart Selman.

**Experimental Methodology**

Our algorithms are coded in C++ using less than 2000 lines including header files, blank lines, and comments.\(^4\) The implementation is flexible, with different look-back techniques and degrees installed by setting various com-

---

3. Available at http://www.cirl.edu/crawford/beijing.
pile-time and run-time parameters. We did not optimize the implementation extensively. We believe investing more attention in this regard, perhaps along the lines suggested by Freeman [95], should improve our performance by up to a factor of three. Freeman’s more sophisticated branch-vari-

able selection heuristics and instance preprocessing tech-
niques also should improve performance.

We experiment with several variants of our DP algo-

rithm. The version applying no look-back enhancements is
denoted “naivesat”, that applying only CBJ “cbjsat”, one
applying relevance-bounded learning of order $i$ “relsat($i$)
”, and one applying size-bounded learning of order $i$ “size-
sat($i$)”. We only use learn orders of 3 and 4, since higher
learn orders resulted in too high an overhead to be generally
useful, and lower learn orders had little effect.

Care must be taken when experimenting with real world
instances because the number of instances available for
experimentation is often limited. The experiment must
somehow allow for performance results on the limited
instance space to generalize to other similar instances. We
found the runtime variance of algorithms solving the same
instance to be extremely high given what seem to be insigni-

ficient differences in either value or variable ordering poli-

cies, whether or not the instance is satisfiable.

Kautz and Selman [96] address this issue by averaging
WSAT’s runtime over multiple runs. We take the same
approach and run our algorithms several times (100) on
each instance with a different random number seed for each
run to ensure different execution patterns. In order to deal
with runs which could take an inordinate amount of time, a
cutoff time was imposed (10 minutes unless otherwise
noted) after which the algorithm was to report failure. We
report the percentage of instances an algorithm failed to
solve within the cutoff time. We report the mean CPU time
required per run and sometimes the mean variable assign-
ments made per run, averaged over successful runs.

The experiments were performed on SPARC-10 worksta-
tions. Kautz and Selman [96] reported running times from a
110-MHz SGI Challenge. To “normalize” our running
times against theirs for the same instances, we solved a

selected set of instances and compared the mean “flips per
second” reported by WSAT, concluding their machine to
have been 1.6 times faster than our SPARC-10®. In the
experimental results that follow, we take the liberty of
reporting all run-times in “normalized SPARC-10™ CPU
seconds. Instead of normalizing run-times reported by
Kautz and Selman for Tableau (ntab), we repeat the experi-
ments on our machine, only using the newest available
version of Tableau, “ntab_back”, available at http://
www.cirl.uoregon.edu/crawford/ntab.tar. This version of
Tableau incorporates a backjumping scheme, and hence is
most similar to our “cbjsat” (though better optimized).
Because ntab_back incorporates no randomizations, the
runtime results for this algorithm are for a single run per

instance.

### Experimental Results

Table 4 displays performance data for relsat(4), WSAT, and
ntab_back on Kautz and Selman’s planning instances. Cutoff
time was 10 minutes for each instance except bw_dir.d,
for which it was 30 minutes. The times for WSAT are those
reported by Kautz and Selman [96], normalized to SPARC-
10 CPU seconds. Relsat(4) outperformed WSAT on most
instances. One exception where WSAT is clearly superior is
on Instance log_dir.c which caused relsat(4) to reach cutoff
22 times. Instance bw_dir.d caused relsat(4) to reach cutoff
18 times, but it still outperformed WSAT by several min-
utes even after averaging in 30 minutes for each relsat cut-
off. Though it is difficult to draw solid conclusions about
the performance of ntab_back since the times reported are
only for a single run, we can determine that relsat(4) is
more effective than ntab_back on the instances for which
relsat(4) never reached cutoff, yet ntab_back required sub-
stantially more than 10 minutes to solve. This includes all
log_gp and log_un instances.

### Table 4. Performance of relsat(4) on Kautz and Selman’s
planning instances.

<table>
<thead>
<tr>
<th>instance</th>
<th>relsat(4)</th>
<th>% fail</th>
<th>WSAT</th>
<th>ntab_back</th>
</tr>
</thead>
<tbody>
<tr>
<td>log_gp.b</td>
<td>12.9</td>
<td>6%</td>
<td>73.2</td>
<td>2,621</td>
</tr>
<tr>
<td>log_gp.c</td>
<td>39.4</td>
<td>4%</td>
<td>419.2</td>
<td>11,144</td>
</tr>
<tr>
<td>log_dir.a</td>
<td>4.1</td>
<td>1%</td>
<td>4.3</td>
<td>369.7</td>
</tr>
<tr>
<td>log_dir.b</td>
<td>16.6</td>
<td>0%</td>
<td>2.6</td>
<td>161.4</td>
</tr>
<tr>
<td>log_dir.c</td>
<td>90.3</td>
<td>22%</td>
<td>3.0</td>
<td>&gt; 12 hours</td>
</tr>
<tr>
<td>log_un.b</td>
<td>66.8</td>
<td>0%</td>
<td>--</td>
<td>12.225</td>
</tr>
<tr>
<td>log_un.c</td>
<td>192.5</td>
<td>0%</td>
<td>--</td>
<td>&gt; 12 hours</td>
</tr>
<tr>
<td>bw_dir.c</td>
<td>119.0</td>
<td>0%</td>
<td>1072</td>
<td>16.9</td>
</tr>
<tr>
<td>bw_dir.d</td>
<td>813.3</td>
<td>18%</td>
<td>1499</td>
<td>&gt; 12 hours</td>
</tr>
</tbody>
</table>

Table 5 displays performance data for our several DP
variants on DIMACS instance bfi355-075 -- the hardest of
the bridge-fault instances. Freeman [95] reports that POSIT
requires 9.8 hours on a SPARC 10 to solve this instance,
and we found ntab_back to solve it in 17.05 seconds. Table
6 displays the same information for the DIMACS instance

5. We could not easily repeat the experiments of Kautz and Sel-
man on our machines due to the need to hand-tune the multiple
input parameters of WSAT.
TABLE 5. Performance on DIMACS bridge-fault instance
bf1355-075.

<table>
<thead>
<tr>
<th>algorithm</th>
<th>run-time</th>
<th>assgnmnts</th>
<th>% fail</th>
</tr>
</thead>
<tbody>
<tr>
<td>naivesat</td>
<td>--</td>
<td>--</td>
<td>100%</td>
</tr>
<tr>
<td>cbjsat</td>
<td>115</td>
<td>999,555</td>
<td>0%</td>
</tr>
<tr>
<td>sizesat(3)</td>
<td>2.5</td>
<td>18,754</td>
<td>0%</td>
</tr>
<tr>
<td>sizesat(4)</td>
<td>.5</td>
<td>3,914</td>
<td>0%</td>
</tr>
<tr>
<td>relsat(3)</td>
<td>3.6</td>
<td>23,107</td>
<td>0%</td>
</tr>
<tr>
<td>relsat(4)</td>
<td>.6</td>
<td>4,391</td>
<td>0%</td>
</tr>
</tbody>
</table>

TABLE 6. Performance on DIMACS single-stuck-at instance
ssa270-141.

<table>
<thead>
<tr>
<th>algorithm</th>
<th>run-time</th>
<th>assgnmnts</th>
<th>% fail</th>
</tr>
</thead>
<tbody>
<tr>
<td>naivesat</td>
<td>--</td>
<td>--</td>
<td>100%</td>
</tr>
<tr>
<td>cbjsat</td>
<td>415</td>
<td>9.4 Million</td>
<td>23%</td>
</tr>
<tr>
<td>sizesat(3)</td>
<td>242</td>
<td>5.0 Million</td>
<td>0%</td>
</tr>
<tr>
<td>sizesat(4)</td>
<td>278</td>
<td>4.7 Million</td>
<td>4%</td>
</tr>
<tr>
<td>relsat(3)</td>
<td>71</td>
<td>1.2 Million</td>
<td>0%</td>
</tr>
<tr>
<td>relsat(4)</td>
<td>46</td>
<td>.62 Million</td>
<td>0%</td>
</tr>
</tbody>
</table>

ssa270-141 -- the hardest of the single-stuck-at instances. Freeman reports POSIT to require 50 seconds to solve this instance\(^6\), and we found ntab_back to solve it in 1,353 seconds. Both of these instances are unsatisfiable.

Naivesat was unable to solve either instance within 10 minutes in any of 100 runs. Adding CBJ resulted in the bridge fault instance being solved in all 100 runs, but the single-stuck-at instance still caused 23 failures. All the learning algorithms performed extremely well on the bridge-fault instance. For the single-stuck-at instance, relevance-bounded learning resulted in a significant speedup. Fourth-order size-bounded learning, while restricting the size of the search space more than third-order size-bounded learning, performed less well due to its higher overhead.

TABLE 7. Performance on DIMACS planning instance hanoi4.

<table>
<thead>
<tr>
<th>algorithm</th>
<th>run-time</th>
<th>assgnmnts</th>
<th>% fail</th>
</tr>
</thead>
<tbody>
<tr>
<td>naivesat</td>
<td>--</td>
<td>--</td>
<td>100%</td>
</tr>
<tr>
<td>cbjsat</td>
<td>325</td>
<td>3.5 Million</td>
<td>94%</td>
</tr>
<tr>
<td>sizesat(3)</td>
<td>214</td>
<td>1.7 Million</td>
<td>92%</td>
</tr>
<tr>
<td>sizesat(4)</td>
<td>227</td>
<td>1.4 Million</td>
<td>79%</td>
</tr>
<tr>
<td>relsat(3)</td>
<td>254</td>
<td>1.6 Million</td>
<td>13%</td>
</tr>
<tr>
<td>relsat(4)</td>
<td>183</td>
<td>.89 Million</td>
<td>1%</td>
</tr>
</tbody>
</table>

DIMACS instances hanoi4 and hanoi5 appear to contain very deep local minima; although they are satisfiable, they have not, to our knowledge, been solved by stochastic algorithms. Ntab_back solves hanoi4 in 2,877 seconds but was unable to solve hanoi5 within 12 hours. We are not aware of any SAT algorithm reported to have solved hanoi5. The results for our DP variants on hanoi4 appear in Table 7. Though sizesat(4) appears faster than relsat(3), its mean run-time is skewed by the fact that it only successfully solved the instance in 21% of its runs. Relsat(3) was successful in nearly all runs and relsat(4) in all but one. We ran the same set of DP variants on hanoi5. The only variant that successfully solved the instance at all was relsat(4), and it did so in only 4 out of the 100 attempts. The average run-time in these four successful runs was under three minutes.

Our DP variants performed relatively well on most of the Beijing instances. The general trend was that thus far illustrated -- the more look-back applied, the better the performance and the lower the probability of reaching cutoff. We were able to solve all the instances within this suite without significant difficulty using relsat(4) with the exception of the “3bit” circuit instances which were never solved by any of our DP variants. Interestingly, we found these instances were trivial for WSAT.

The “2bit” circuit instances were trivial (a fraction of a second mean solution time) even for cbjsat, with the exception of 2bitadd_10, their only unsatisfiable representative. This instance was not solvable by any of our algorithms within 10 minutes. After disabling cutoff, relsat(4) determined it unsatisfiable in 18 hours.

Relsat(4) solved 4 out of 6 scheduling instances with a 100% success rate. Two of the instances, e0-10-by-5-1 and en-10-by-5-1 resulted in failure rates of 21% and 18% respectively. Repeating the experiments for these two instances with a 30-minute cutoff reduced the failure rate to 3% and 1% respectively. Crawford and Baker [94] reported that ISAMP, a simple randomized algorithm, solved these types of instances more effectively than WSATl or Tableau. Our implementation of ISAMP solved these 6 instances an order of magnitude more quickly than relsat(4), and with a 100% success rate. We did not find ISAMP capable of solving any other instances considered in this paper.

Of the Beijing planning instances, relsat(3) and relsat(4) found 3blocks to be easy, solving it with 100% success in 6.0 and 6.4 seconds on average respectively. The 4blocks instance was also easy, with both relsat(3) and relsat(4) again achieving 100% success, though this time in 79 and 55 seconds respectively. The 4blocks instance was more difficult. The failure rate was 34% for relsat(3) and 17% for relsat(4), with average CPU seconds of 406 and 333 seconds respectively. Because the mean times were so close to the cutoff, we expect increasing the cutoff time should significantly reduce the failure rate as it did with the scheduling instances.

**Discussion**

Look-back enhancements clearly make DP a more capable algorithm. For almost every one of the instances tested here (selected for their difficulty), learning and CBJ were critical for good performance. We suspect the dramatic performance improvements resulting from the incorporation of look-back is in fact due to a synergy between the look-ahead and look-back techniques applied. Variable selection heuristics attempt to seek out the most-constrained areas of the search space to realize inevitable failures as quickly as possible. Learning schemes, through the recording of derived clauses, can create constrained search-sub-spaces for the variable selection heuristic to exploit.

Size-bounded learning is effective when instances have relatively many short nogoods which can be derived with-

---

6. Freeman also reports that POSIT exhibits high run-time variability on ssa2670-141, though the variance is not quantified.
out deep inference. Relevance-bounded learning is effective when many sub-problems corresponding to the current DP assignment also have this property. Our findings indicate that real-world instances often contain subproblems with short, easily derived nogoods. Phase transition instances from Random 3SAT tend to have very short nogoods [Schrag and Crawford 96], but these seem to require deep inference to derive, and look-back-enhanced DP provides little advantage on them [Bayardo & Schrag 96].

As we have noted, a few test instances were infeasible for look-back-enhanced DP but easy or even trivial for WSAT. Look-back for DP is not a "magic bullet", and good look-back techniques alone will not result in universally superior performance, just as alone the good look-ahead techniques included in Tableau and POSIT do not. The best algorithms, stochastic or systematic, are bound to be stymied by instances of sufficient size and complexity or adversarial structure. Nevertheless, combining good techniques for look-ahead and look-back is likely to give better performance across a broad range of problems.

Some researchers have attempted to exploit the distinct advantages of systematic and stochastic search in hybrid global/local search algorithms. Ginsberg and McAllester's [94] partial-order dynamic backtracking, which incorporates a form of relevance-bounded learning along with a scheme that relaxes the restrictions on changing past variable assignments, has been shown to perform better than Tableau on a random problem space with crystallographic structure. Mazure et al. [96] evaluated a hybrid algorithm with interleaved DP and local search execution using several instances from the DIMACS suite, showing that it frequently outperformed capable non-hybrid DP implementations. Because look-back enhanced DP is also effective at solving the DIMACS instances used by Mazure et al. and the crystallographic instances of Ginsberg and McAllester [Bayardo and Schrag 96], future work is required to see if and when these techniques are complementary to look-back.

Given the similarities between experimental results from this study and those from our previous study on randomly generated "exceptionally hard" instances [Bayardo and Schrag 96], we speculate that this random problem space may contain instances that better reflect computational difficulties arising in real-world instances than random spaces like Random 3SAT.

Conclusions

We have described CSP look-back enhancements for DP and demonstrated their significant advantages. We feel their performance warrants their being included as options in DP implementations more commonly. Where DP is used in a larger system (for planning, scheduling, circuit processing, knowledge representation, higher-order theorem proving, etc., or in a hybrid systematic/stochastic SAT algorithm), look-back-enhanced DP should probably replace unenhanced DP, where another SAT algorithm is used. DP should be given a new evaluation using look-back enhancements. Finally, look-back-enhanced DP should become a standard algorithm, along with unenhanced DP, against which other styles of SAT algorithm are compared.

References


7. A theoretical comparison of these two methods for restricting learning overhead appears in [Bayardo & Miranker 96].