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

Robert C. Schrag
Information Extraction and Transport, Inc.
1730 North Lynn Street, Suite 502
Arlington, VA 22209 USA
schrag@iet.com
http://www.iet.com/users/schrag

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 Walk-SAT) [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 enforcement schemes such as forward checking. The latter class includes schemes for backjumping (also known as intelligent backtracking) and learning (also known as nogood or constraint recording). In CSP algorithms, techniques from both classes are popular; for instance, one common combination of techniques is forward checking, conflict-directed backjumping, and an ordering heuristic preferring variables with the smallest domains.

SAT is a specific kind of CSP in which every variable ranges over the values {true, false}. For SAT, the most popular systematic algorithms are variants of the Davis-Putnam procedure (DP) [Davis et al. 62]. In CSP terms, DP is equivalent to backtrack search with forward checking and an ordering heuristic favoring unit-domained variables. Two effective modern implementations of it are Tableau [Crawford and Auton 96] and POSIT [Freeman 95]; both are highly optimized and include carefully selected variable ordering heuristics. Neither of these implementations (in their published descriptions) include look-back enhancements like we describe.

Systematic, or global search algorithms traverse a search space systematically to ensure that no part of it goes unexplored. They are complete: given enough running time, if a solution exists they will find it; if no solution exists they will report this. Alternative to systematic algorithms for SAT are stochastic, or local search algorithms such as WSAT and GSAT [Selman et al. 92]. Stochastic algorithms explore a search space randomly by making local perturbations to a working assignment without memory of where they have been. They are incomplete: they are not guaranteed to find a solution if one exists; they cannot report that no solution exists if they do not find one.

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 $\{\text{true}, \text{false}\}$. 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. $(x \vee y \vee \neg z)$; 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 formula* (CNF) is a conjunction of clauses (e.g. $(a \vee b) \wedge (x \vee y \vee \neg z)$); 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 $(a \vee b \vee \neg c)$ encodes the nogood $\{\neg a, \neg b, c\}$. 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, $(a \vee \neg b)$ resolves with $(b \vee c)$ to produce $(a \vee c)$.

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 σ, which is empty upon initial top-level entry to the recursive, call-by-value procedure DP.

```
DP(F, σ)
   UNIT-PROPAGATE(F, σ)
   if ( ) in F then return
   if F = ∅ then exit-with(σ)
   α ← SELECT-BRANCH-VARIABLE(F)
   DP(F ∪ {(α)}, σ ∪ {α})
   DP(F ∪ {(¬α)}, σ ∪ {¬α})
   return
```

The CNF F and the truth assignment σ 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.

```
UNIT-PROPAGATE(F, σ)
   while (exists ω in F where ω = (λ))
      σ ← σ ∪ {λ}
      F ← SIMPLIFY(F)
```

UNIT-PROPAGATE adds the single literal λ from a unit clause ω to the literal set σ, 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 γ appearing in some binary clause a score of $\text{neg}(\gamma) \cdot \text{pos}(\gamma) + \text{neg}(\gamma) + \text{pos}(\gamma)$ where $\text{pos}(\gamma)$ and $\text{neg}(\gamma)$ are the numbers of occurrences of γ 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 γ, compute $\text{pos}(\gamma)$ and $\text{neg}(\gamma)$ as the number of variables valued by UNIT-PROPAGATE after making the assignment $\{\gamma\}$ and $\{\neg\gamma\}$ respectively. Should either unit propagation lead to a contradiction, immediately return γ as the next branch variable and pursue the assignment for this variable which led to the contradiction. Otherwise, score γ 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 $\{\neg a, \neg b\}$ is part of the current assignment, the input clause $(a \vee b \vee x)$ 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 reasons; then it backs up to the most recently assigned variable δ in C. Suppose $\{\gamma\}$ was the most recent assignment of variable γ. If $\{\neg\gamma\}$ is excluded by a reason D, then we create a new working reason E by resolving C and D and back up to the most recently assigned variable in E. Otherwise, we install C as the reason for excluding $\{\gamma\}$, change the current assignment to include $\{\neg\gamma\}$, and proceed with DP.

Extending our example, suppose upon detecting failure we have the complementary reason $(a \vee b \vee \neg x)$, and that b was assigned after a. Resolution gives us the working reason $(a \vee b)$, so CBJ backs up to where the assignment $\{\neg b\}$ was made. If $\{b\}$ is excluded, then suppose the reason is $(\neg b \vee y)$. Resolution yields the new working reason $(a \vee y)$ and CBJ keeps backing up. If $\{b\}$ is not excluded (b was a branch variable), $(a \vee b)$ becomes the reason excluding $\{\neg b\}$, and $\{b\}$ replaces $\{\neg b\}$ in the current assignment before DP continues.

Learning schemes maintain derived reasons longer than does CBJ, which can discard them as soon as they are no longer denoting a value as excluded. Unrestricted learning records every derived reason exactly as if it was a clause from the underlying instance, allowing it to be used for the remainder of the search. Because the overhead of unrestricted learning is high, we apply only the restricted learning schemes as defined in [Bayardo & Miranker 96]. *Size-bounded* learning of order i retains indefinitely only those derived reasons containing i or fewer variables. For instance, the reason $(a \vee b)$ would be maintained by second-order size-bounded learning, but longer reasons would not. *Relevance-bounded* learning of order i maintains any reason that contains at most i variables whose assignments have changed since the reason was derived. For example, suppose we are performing second-order relevance-bounded learning, and we derive a reason $(a \vee b \vee y)$ where variables a, b, and y where assigned in the order they appear. This reason would be maintained by second-order relevance-bounded learning as long as a remains assigned as $\neg a$. As soon as a is re-assigned or un-assigned by a backup, the reason would be discarded.

Test Suites

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.

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

Selected SAT-encoding planning instances constructed by Kautz and Selman [96] (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.

instance	vars	clauses	sat	type
ssa2670-141	986	2,315	N	diagnosis
bf1355-075	2,180	6,778	N	diagnosis
hanoi4	718	4,932	Y	planning
hanoi5	1931	14,468	Y	planning

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 fewer 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-

1. Available at ftp://ftp.research.att.com/dist/ai/logistics.tar.Z and satplan.data.tar.Z.
2. Available at ftp://dimacs.rutgers.edu/pub/challenge/satisfiability.
3. Available at http://www.cirl.edu/crawford/beijing.
4. Source code available at http://www.cs.utexas.edu/users/bayardo.

TABLE 3. Beijing instances.

instance	vars	clauses	sat	type
e0-10-by-5-1	19,500	108,887	Y	scheduling
e0-10-by-5-4	19,500	104,527	Y	scheduling
en-10-by-5-1	20,700	111,567	Y	scheduling
en-10-by-5-8	20,700	113,729	Y	scheduling
ew-10-by-5-1	21,800	118,607	Y	scheduling
ew-10-by-5-8	22,500	123,329	Y	scheduling
3blocks	283	9,690	Y	planning
4blocksb	410	24,758	Y	planning
4blocks	758	47,820	Y	planning
2bitadd_10	590	1,422	N	synthesis
2bitadd_11	649	1,562	Y	synthesis
2bitadd_12	708	1,702	Y	synthesis
2bitcomp_5	125	310	Y	synthesis
2bitmax_6	252	766	Y	synthesis
3bitadd_31	8,432	31,310	Y	synthesis
3bitadd_32	8,704	32,316	Y	synthesis

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-variable selection heuristics and instance preprocessing techniques also should improve performance.

We experiment with several variants of our DP algorithm. 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 "sizesat(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 insignificant differences in either value or variable ordering policies, 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 assignments made per run, averaged over successful runs.

The experiments were performed on SPARC-10 workstations. 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[5]. 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 experiments 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 runtimes reported 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 minutes even after averaging in 30 minutes for each relsat cutoff. 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 substantially 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.

instance	relsat(4)	% fail	WSAT	ntab_back
log_gp.b	12.9	0%	75.2	2,621
log_gp.c	39.4	0%	419.2	11,144
log_dir.a	4.1	0%	4.3	369.7
log_dir.b	16.6	0%	2.6	161.4
log_dir.c	90.3	22%	3.0	> 12 hours
log_un.b	66.8	0%	--	12,225
log_un.c	192.5	0%	--	> 12 hours
bw_dir.c	119	0%	1072	16.9
bw_dir.d	813.3	18%	1499	> 12 hours

Table 5 displays performance data for our several DP variants on DIMACS instance bf1355-075 -- 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 Selman 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.

algorithm	run-time	assgnmnts	% fail
naivesat	--	--	100%
cbjsat	115	999,555	0%
sizesat(3)	2.6	18,754	0%
sizesat(4)	.5	3,914	0%
relsat(3)	3.6	23,107	0%
relsat(4)	.6	4,391	0%

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

algorithm	run-time	assgnmnts	% fail
naivesat	--	--	100%
cbjsat	415	9.4 Million	23%
sizesat(3)	242	5.0 Million	0%
sizesat(4)	278	4.7 Million	4%
relsat(3)	71	1.2 Million	0%
relsat(4)	46	.62 Million	0%

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.

algorithm	run-time	assgnmnts	% fail
naivesat	--	--	100%
cbjsat	325	3.5 Million	94%
sizesat(3)	214	1.7 Million	92%
sizesat(4)	227	1.4 Million	79%
relsat(3)	254	1.6 Million	13%
relsat(4)	183	.89 Million	1%

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 WSAT 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 4blocksb 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.[7] 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.

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

References

Bayardo, R. J. and Miranker, D. P. 1996. A Complexity Analysis of Space-Bounded Learning Algorithms for the Constraint Satisfaction Problem. In *Proc. 13th Nat'l Conf. on Artificial Intelligence*, 558-562.

Bayardo, R. J. and Schrag, R. 1996. Using CSP Look-Back Techniques to Solve Exceptionally Hard SAT Instances. In *Proc. Second Int'l Conf. on Principles and Practice of Constraint Programming (Lecture Notes in Computer Science v. 1118)*, Springer, 46-60.

Crawford, J. M. and Auton, L. D. 1996. Experimental Results on the Crossover Point in Random 3SAT. *Artificial Intelligence* 81(1-2), 31-57.

Crawford, J. M. and Baker, A. B. 1994. Experimental Results on the Application of Satisfiability Algorithms to Scheduling Problems. In *Proc. Twelfth Nat'l Conf. on Artificial Intelligence*, 1097-1097.

Davis, M., Logemann, G. and Loveland, D. 1962. A Machine Program for Theorem Proving, *CACM* 5, 394-397.

Freeman, J. W. 1995. *Improvements to Propositional Satisfiability Search Algorithms.* Ph.D. Dissertation, U. Pennsylvania Dept. of Computer and Information Science.

Frost, D. and Dechter, R. 1994. Dead-End Driven Learning. In *Proc. of the Twelfth Nat'l Conf. on Artificial Intelligence*, 294-300.

Ginsberg, M. and McAllester, D. 1994. GSAT and Dynamic Backtracking, *Principles of Knowledge Representation and Reasoning: Proceedings of the Fourth Int'l Conf.*, 226-237.

Kautz, H. and Selman, B. 1996. Pushing the Envelope: Planning, Propositional Logic, and Stochastic Search. In *Proc. 13th Nat'l Conf. on Artificial Intelligence*, 558-562.

Mazure, M., Sais, L. and Gregoire, E. 1996. Detecting Logical Inconsistencies. In *Proc. of the Fourth Int'l Symposium on Artificial Intelligence and Mathematics*, 116-121.

Prosser, P. 1993. Hybrid Algorithms for the Constraint Satisfaction Problem. *Computational Intelligence* 9(3):268-299.

Schrag, R. and Crawford, J. M. Implicates and Prime Implicates in Random 3SAT. Artificial Intelligence 81(1-2), 199-222.

Selman, B., Kautz, H., and Cohen, B., 1994. Noise Strategies for Local Search. In *Proc. Twelfth Nat'l Conf. on Artificial Intelligence*, 337-343.

Selman, B., Levesque, H. and Mitchell, D. 1992. A New Method for Solving Hard Satisfiability Problems, In *Proc. Tenth Nat'l Conf. on Artificial Intelligence*, 440-446.

Stallman R. M. and. Sussman G. J., 1977. Forward Reasoning and Dependency-Directed Backtracking in a System for Computer-Aided Circuit Analysis. *Artificial Intelligence* 9, 135-196.