Exploring the Computational Tradeoff of more Reasoning and Less Searching

Fahiem Bacchus
Dept. Of Computer Science
University Of Toronto
Toronto, Ontario
Canada, M5S 1A4
fbacchus@cs.toronto.edu

February 5, 2002

1 Introduction

When doing backtracking tree search, as in the Davis-Putnam-Logemann-Loveland [4, 3] procedure, there is a well known tradeoff between doing more reasoning at each node of the search tree and exploring more nodes. At the extreme one could invoke a complete reasoner at each node, backtracking if the reasoner says the formula at that node is unsatisfiable. Such a procedure would explore zero nodes for unsatisfiable problems and at most n nodes for satisfiable problems with n variables. However, such a procedure has no potential for improving problem solving efficiency.

In the constraints literature a large collection of polynomial time reasoning procedures that can be executed at each node of a backtracking search have been defined under the name "local propagation" (e.g., [5]). These procedures have two properties: they are able to detect some forms of inconsistency which allows the search to backtrack immediately without having to search the subtree under the inconsistent node, and they are able to simplify the problem so that even if inconsistency is not detected useful work can still be accomplished. Furthermore, there is a large family of propagation techniques with the property that one can monotonically gain inconsistency detection power for a monotonic increase in computational cost [7]. As a result a large amount of research in the Constraints field has focused on identifying the local propagation procedures that offer the best tradeoff between reducing the number of nodes searched and the time it takes at each node to run the procedure. The aim, of course, being to reduce the total time required by the backtracking procedure.

In DPLL solvers, on the other hand, unit propagation has dominated, and to date other more expensive forms of reasoning at each node has not proved to be cost effective.[1] For example, the 2clsVER system of van Gelder [13] generally explores many fewer nodes than its unit propagating alternatives, but it is generally slower. In this case the additional reasoning being done (computing a closure over the binary clauses) is not achieving a good tradeoff between time spent per node and the reduction in the number of nodes searched.

This result would seem to indicate that less reasoning is better. However, that is not necessarily the case. There are two factors to consider. First, additional polynomial time reasoning has the potential to allow the search to avoid searching subtrees that might take it exponential time to explore. Thus it is always possible to contrive examples where even very expensive polynomial time reasoning pays off because it allows the search to avoid the exponential cost of searching a particularly expensive subtree. The issue in practice is how often this happens and how much work is saved (e.g., the extra reasoning might only avoid subtrees that are easy to refute with simpler reasoning). Second, additional reasoning might sufficiently simplify the theory that the bulk of the nodes, i.e., those at the deeper levels of the tree, involve very simple formulas and can be searched very quickly. This might have the effect that the net nodes/second search rate is actually increased by doing more reasoning.

In this paper we demonstrate that both of these phenomena can occur. In particular, we show that using an even more expensive form of reasoning than performed by 2clsVER can achieve a better tradeoff between reasoning time and search reduction and thus achieve significantly better performance. In some cases, we will also see that the net rate of nodes/seconds can also be improved due to the additional simplification being done at the top of the search tree. In the sequel we explain the reasoning that we employ, and show some empirical results.

2 Hyper Resolution with Binary Clauses in DPLL

We will view the DPLL procedure as being a recursive algorithm taking as input a *pair* (F, A), where F is a CNF formula, and A is a set of literals that have already been assigned TRUE by previous invocations. Initially DPLL is called with the input formula $\mathcal{F}$ and $A = \emptyset$, i.e., with $(\mathcal{F}, \emptyset)$.

DPLL performs various reductions on the formula component of its input prior to invoking itself recursively. The most fundamental of these is to compute the reduction of its input by a literal ℓ (also called *forcing* ℓ), denoted by $(F, A)[\ell]$. $(F, A)[\ell] = (F', A')$ where F' is generated from F by removing from F all clauses containing ℓ and then removing $\overline{\ell}$ from all the remaining clauses, and A' is simply $A \cup \{\ell\}$.

Second, it can use various forms of reasoning to further reduce its input formula. The most important of these is unit clause reduction. Unit clause reduction selects a unit clause from the

[1] We are not considering here specialized reasoning techniques that can be implemented efficiently for specific types of sub-theories. Such specialized techniques can be cost effective as shown, e.g., by the EQSAT system [8]. Specialized reasoners is also heavily utilized in the constraints field. The most well known probably being the specialized reasoning employed in the all-difference constraint [11].

input formula and performs a reduction of the input by the literal in that clause. Unit clause reduction can generate new unit clauses and *unit Propagation* (UP) is the iterative process of doing unit clause reductions until either (a) a contradiction is achieved, or (b) there are no more unit clauses in the input. A *contradiction* is achieved when the set of assigned literals, A, contains both ℓ and $\overline{\ell}$ for some literal ℓ.

The basic implementation of DPLL is to first perform unit propagation on the input formula. If the resulting formula is empty, i.e., all clauses have been satisfied, then A is a satisfying truth assignment. Otherwise, if A contains a contradiction then this particular collection of assigned literals cannot be extended to a solution, and we backtrack. Otherwise, DPLL chooses a literal ℓ to split on and recursively searches for a satisfying truth assignment containing ℓ and if none exist, for one that contains $\overline{\ell}$. If neither extension succeeds DPLL backtracks.

```
 DPLL(T,A)
1. (T',A')=UP(T,A)
2. if A' contains a contradiction
3.    return(FALSE)
4. elseif T' is empty
5.    return(TRUE)
6. l := selectVarNotInA(T,A)
7. if(DPLL((T,A)[l])
8.    return(TRUE)
9. else
10.   return(DPL((T,A)[$\overline{l}$]))
```

Except for the 2clsVER system of [13], most current DPLL solvers use this basic algorithm (along with other orthogonal improvements). In particular, unit propagation is all that is used in the formula reduction phase.

In many problems the initial formula can contain many binary clauses, and even if it doesn't many binary clauses will be created as DPLL reduces the formula during its splitting process. 2clsVER performs all possible resolutions of pairs of binary clauses. Such resolutions yield only new binary clauses or new unit clauses.[2] We denote by BinRes the transformation of the input that consists of repeatedly (a) adding to the formula all new binary or unit clauses producible by resolving pairs of binary clauses, and (b) performing UP on any new unit clauses that appear (which in turn might produce more binary clauses causing another iteration of (a)), until either (1) a contradiction is achieved, or (2) nothing new can be added by a step of (a) or (b).

Another common technique used in DPLL solvers is failed literal detection [6]. Failed literal detection is a one-step lookahead with UP. If forcing literal ℓ and then performing UP yields a contradiction then $\overline{\ell}$ is in fact entailed by the current input and we can force it (and then perform UP). DPLL solvers often perform failed literal detection on a set of likely literals at each node.

[2]2clsVER also has the option of resolving away a variable rather than splitting on it, but this additional form of reasoning is still not sufficient to make it competitive with state of the art UP based DPLL solvers.

OBSERVATION 1 *If BinRes forces the literal ℓ, then failed literal detection on $\bar{\ell}$ would also detect that ℓ is entailed.*

This observation can be proved by examining the implication graph representation of the binary clauses.

OBSERVATION 2 *Failed literal detection is able to detect entailed literals that cannot be detected by BinRes.*

For example, if we test c with failed literal detection in the formula $(\{(\bar{c}, a), (\bar{c}, b), (\bar{c}, d), (\bar{a}, \bar{b}, \bar{d})\}, [])$, we will detect a contradiction and thus that $\bar{c}$ is entailed. BinRes on the other hand, does not force $\bar{c}$: BinRes does not consider the non-binary clauses.

We have found in our experiments that ignoring the non-binary clauses tends to produce a binary clause sub-theory that is relatively isolated from the rest of the theory. Hence, computing its closure produces fewer contradictions than might be expected.

To remedy this we have investigated hyper-resolution. Hyper-resolution is a resolution step that involves more than two clauses. Here we define a hyper-resolution step to take as input *one* n-ary clause ($n \geq 2$) $(l_1, l_2, ..., l_n)$ and $n-1$ binary clauses each of the form $(\bar{l_i}, \ell)$ $(i = 1, \ldots, n-1)$. It produces as output the new binary clause (ℓ, l_n). For example, using hyper-resolution on the inputs (a, b, c, d), $(h, \bar{a})$, $(h, \bar{c})$, and $(h, \bar{d})$, produces the new binary clause (h, b). Note that the standard resolution of two binary clauses is covered by this definition (with $n = 2$).

We denote by HypBinRes the transformation of the input that is exactly like BinRes except that it performs the above hyper-resolution instead of simply resolving binary clauses. DPLL-HypBinRes is then defined to be DPLL with HypBinRes substituted for UP, similarly DPLL-BinRes is DPLL with BinRes substituted for UP.

OBSERVATION 3 *HypBinRes detects the same set of forced literals as repeatedly (a) doing a failed literal test on* ***all*** *literals, and (b) performing UP on all detected entailed literals, until either (1) a contradiction is achieved, or (2) no more entailed literals are detected.*

This observation can be proved by showing that every literal that HypBinRes forces can be detected by the failed literal test. The requirement for repeating the failed literal test follows from fact that HypBinRes is run to closure. Note that simply performing the failed literal test on every literal is not as powerful as HypBinRes. One would have to retest every literal every time an entailed literal is unit propagated until no new entailed literals are detected. As a result it is much more efficient to perform HypBinRes rather than repeated failed literal tests—HypBinRes does not need to repeat work in the same way.

HypBinRes is also very useful when it comes to computing heuristics. A very useful heuristic is to rank literals by the number of new literals they would force under UP. This is the heuristic used by SATZ [9]. However, this heuristic is costly to compute, and can usually only be estimated. SATZ for example evaluates this heuristic on some set of candidate literals by unit propagating each one and then counting the number of newly forced literals. (Failed literal detection is an important side effect of this process). HypBinRes has the following property:

OBSERVATION 4 *A literal ℓ will force a literal ℓ' under UP if and only if the binary clause $(\bar{\ell}, \ell')$*

is in the formula after performing HypBinRes.

Thus after performing HypBinRes a simple count of the number of binary clauses a literal's negation participates in yields the precise number of literals that would be forced by unit propagating that literal. This observation is a direct corollary of Observation 3.

Finally, if one is reasoning with binary clauses, equality reduction can be performed. If a formula F contains $(\bar{a}, b)$ as well as $(a, \bar{b})$, then we can form a new formula $\text{EqReduce}(F)$ by equality reduction. Equality reduction involves (a) replacing all instances of b in F by a (or vice versa), (b) removing all clauses which now contain both a and $\bar{a}$, (c) removing all duplicate instances of a (or $\bar{a}$) from all clauses. This process might generate new binary clauses.

EqReduce can be added to HypBinRes (HypBinRes+eq) by repeatedly doing (a) equality reduction, (b) hyper-resolution, and (c) unit propagation, until nothing new is added or a contradiction is found. DPLL-HypBinRes+eq is then defined to be DPLL using HypBinRes+eq instead of UP

OBSERVATION 5 *HypBinRes+eq detects the same set of forced literals as HypBinRes.*

The two binary clauses $(a, \bar{b})$ and $(\bar{a}, b)$ allow HypBinRes to deduce everything that HypBinRes+eq can.

This means that modulo changes in the heuristic choices it might make DPLL-HypBinRes+eq does not have the potential to produce exponential savings over DPLL-HypBinRes since it cannot detect any further inconsistencies.[3] The benefit of using equality reduction is that for many problems equality reduction significantly simplifies the formula. This can have a dramatic effect on the time it takes to compute HypBinRes.

3 The Sat Solver 2CLS+EQ

We have implemented DPLL-HypBinRes+eq in a system called 2CLS+EQ. And here we present some results comparing 2CLS+EQ when it uses hyper-resolution, when only BinRes is being used, and when equality reduction is turned on or off. In our tests we used some of the benchmark suites utilized on the Sat-Ex web site [12]. The details of which problems these suites contain can be found at this web site.

In Table 1 the first 5 problem suites are random 3-SAT problems drawn from the hard region (4.26 clause/variable ratio). The data indicates that equality processing is not helpful, that hyper-resolution saves about a factor 5 in nodes searched and a factor of 2 in time. More extensive data is provided in Table 3 in the appendix. This data indicates that a range of different behaviors are possible, but overall there is strong evidence for the superiority of using the defined hyper-resolution step over simply computing the closure of the binary sub-theory. Some of the key points are illustrated in Table 1.

- In the "bf" family of problems, hyper-resolution is able to save a factor of 1000 in nodes searched and a factor of 300 in time.

[3]Note that equality reduction when added to plain BinRes does add inconsistency detection power.

- In the ucsc-bf family equality processing allows nodes to be processed about 30 times as fast (when hyper-resolution is used). This is an example of more reasoning speeding up the rate at which nodes can be processed.

- In the (pigeon) hole problems neither equality reduction nor hyper-resolution pay off. Fortunately, it seems that the penalty paid when things are slower is significantly less than the gain received when things are faster. (See the data in Table 3 for further verification of this claim.)

Family (#problems)		**Binary**	**Binary+Eq**	**Hyper**	**Hyper+EQ**
3Sat 50 (100)	Fails	0	0	0	0
	Time	1.2	1.5	1.1	1.1
	Nodes	2,877	2,809	679	680
3Sat 100 (100)	Fails	0	0	0	0
	Time	12.7	13.0	8.9	8.8
	Nodes	20,308	20,277	3,926	3,918
3Sat 150 (100)	Fails	0	0	0	0
	Time	148.7	156.8	92.8	93.0
	Nodes	152,275	151,719	27,406	27,629
3Sat 200 (100)	Fails	0	0	0	0
	Time	1,590.5	1,639.5	858.8	854.1
	Nodes	1,016,073	1,018,201	157,026	158,865
3Sat 250 (100)	Fails	0	0	0	0
	Time	13,427	13,277.5	6,949.5	6,883.3
	Nodes	5,914,397	5,929,868	892,623.0	900,056
bf (4)	Fails	2	0	0	0
	Time	1321.2	346.5	11.6	3.3
	Nodes	2,318,380	3,255,878	1,138.0	2,648.0
hole (5)	Fails	0	0	0	0
	Time	98.7	108.5	123.0	130.2
	Nodes	4,519,272	4,519,272	2,887,456.0	2,887,456
ucsc-bf (223)	Fails	76	33	1	1
	Time	47,334.2	27,694	2,206.3	700.4
	Nodes	57,622,001	331,321,831	623,802	8,244,040

Table 1: Comparison of performance on various benchmarks. All problems were run with a time limit of 500 CPU Sec. Figures shown are totals over all problems in the family, counting the time (500 sec) for failed problems. Tests were on a 500MHz PIII.

Although these results are an improvement over simply computing the binary closure, they are still not competitive with the best UP DPLL solvers. However, there are two orthogonal improve-

Family (#problems)	2CLS+EQ	2CLS+EQ ranking	RELSAT-2000	SATO-v3.00	ZCHAFF
hfo4(40)	1,602.90	10th	**569.51**	33,785.92	6,506.10
eq-checking(34)	24.75	3rd	13.88	(1) 10,007.25	**2.45**
facts(15)	92.37	5th	75.46	**12.78**	13.45
quasigroup(22)	1996.20	6th	2,347.83	1,087.70	**845.89**
queueinvar(10)	139.59	4th	236.04	(2) 20,400.45	**15.39**
des-encryption(32)	(8) 80230.66	3rd	(8) 80,729.42	(8) 80,402.84	**(2) 22,726.43**
fvp(4)	(3) 30003.68	2nd	(3) 30,006.79	(3) 30,006.35	**1,224.86**
Beijing(16)	(8) 80773.51	12th	(2) 24024.57	(4) 44078.88	**(2) 20268.12**
barrel(8)	6415.63	3rd	(1) 11,872.80	(1) 10,417.70	**912.22**
longmult(16)	**3234.24**	1st	41,243.77	(1) 22,270.98	4,502.49
miters(25)	**737.44**	1st	(7) 86,670.25	(20) 209,123.33	(2) 21,289.77

Table 2: Results of the best general purpose Sat Solvers on various problem suites. Bracketed numbers indicate number of failures for that family. Ranking is with respect to all 23 solvers on Sat-Ex. The best times are in **bold** and times have been standardized to the Sat-Ex machine times.

ments that are useful in any DPLL solver: intelligent backtracking and clause recording. We have implemented intelligent backtracking in 2CLS+EQ, and a simplified (and low-overhead) version of clause recording by adapting the pruneback techniques described in [1].

With these two improvements 2CLS+EQ becomes one of the most powerful SAT solvers. For example, with these additions it can solve all 722 of the non-random problems in Tables 1 and 3 in a total of 494 sec. The performance of 2CLS+EQ against three of the best UP DPLL solvers on harder families of problems is given in Table 2.[4] In Table 2 all times have been standardized to the Sat-Ex machine times (using the Dimacs machine scale), and we use the Sat-Ex convention of counting 10,000 seconds for each failed run. All of the results quoted in this paper were generated using the same variable choice heuristic: count the number of binary clauses each value of the variable participates in, combine these scores using a multiplicative function, and then choose the top scoring variable.

The miters family is where 2CLS+EQ has its best performance. It is the only solver among those reported on in the Sat-Ex site that is able to solve all of these problems. In many ways 2CLS+EQ is superior to every other DPLL solver except for ZCHAFF [10]. On a number of problems ZCHAFF is in a class of its own. However, there is much room for improvement in 2CLS+EQ. In particular, the simplified version of clause recording we have implemented is not as powerful as normal clause learning: our technique forgets the learned clause as soon as we backtrack far enough that 3 of the clause's literals are unassigned. We expect that once we have implemented full clause learning 2CLS+EQ will be superior to ZCHAFF on a much larger set of problems, and more competitive with it on an even larger set. Additionally, ZCHAFF was very carefully engineered and highly

[4]These three solvers, RELSAT-2000 [2], SATO-v3.00 [14], and zchaff [10] were the only other solvers capable of solving all of the 722 "easy"problems mentioned in Tables 1 and 3.

optimized. Although we have put a fair effort into implementing 2CLS+EQ, there is considerable room for further optimization.

4 Conclusions

Our results indicate that the space of tradeoffs between more reasoning and less search is far from being fully explored. Unfortunately, the function relating the amount of reasoning and solving time is not monotonic. So considerable further experimentation will be required to learn more about this space.

References

[1] Fahiem Bacchus. Extending forward checking. In *Principles and Practice of Constraint Programming—CP2000*, number 1894 in Lecture Notes in Computer Science, pages 35–51. Springer-Verlag, New York, 2000.

[2] R. J. Bayardo and R. C. Schrag. Using csp look-back techniques to solve real-world sat instances. In *Proceedings of the AAAI National Conference (AAAI)*, pages 203–208, 1997.

[3] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 4:394–397, 1962.

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

[5] R. Debruyne and C. Bessière. Some practicable filtering techniques for the constraint satisfaction problem. In *Procceedings of the International Joint Conference on Artifical Intelligence (IJCAI)*, pages 412–417, 1997.

[6] J. W. Freeman. *Improvements to Propositional Satisfiability Search Algorithms*. PhD thesis, University of Pennsylvania, 1995.

[7] E. Freuder. A sufficient condition for backtrack-bounded search. *Journal of the ACM*, 32(4):755–761, 1985.

[8] Chu Min Li. Integrating equivalence reasoning into davis-putnam procedure. In *Proceedings of the AAAI National Conference (AAAI)*, pages 291–296, 2000.

[9] Chu Min Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *Proceedings of the International Joint Conference on Artifical Intelligence (IJCAI)*, pages 366–371, 1997.

[10] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient sat solver. In *Proc. of the Design Automation Conference (DAC)*, 2001.

[11] J.-C. Regin. A filtering algorithm for constraints of difference in CSP. In *Proceedings of the AAAI National Conference*, pages 362–367, Seattle, Wash., 1994.

[12] Laurent Simon and Philippe Chatalic. Satex: A web-based framework for SAT experimentation (http://www.lri.fr/~simon/satex/satex.php3). In Henry Kautz and Bart Selman, editors, *LICS 2001 Workshop on Theory and Applications of Satisfiability Testing (SAT 2001)*, volume 9 of *Electronic Notes in Discrete Mathematics*. Elsevier, 2001.

[13] Allen Van Gelder. Combining preorder and postorder resolution in a satisfiability solver. In Henry Kautz and Bart Selman, editors, *LICS 2001 Workshop on Theory and Applications of Satisfiability Testing (SAT 2001)*, volume 9 of *Electronic Notes in Discrete Mathematics*. Elsevier, 2001.

[14] H. Zhang. Sato: An efficient propositional prover. In *Proceedings of the Fourteenth International Conference on Automated Deduction (CADE)*, volume 1249 of *LNCS*, pages 272–275. Springer-Verlag, 1997.

Family (#problems)		**Binary**	**Binary+Eq**	**Hyper**	**Hyper+EQ**
aim (72)	Fails	12	12	7	7
	Time	7375.2	7353.5	3,540.0	3,538.7
	Nodes	375,292,163	371,062,385	57,372,302	57,514,005
ais (4)	Fails	0	0	0	0
	Time	0.1	0.2	26.1	35.6
	Nodes	48	40	17,266	31,690
blocksworld (7)	Fails	1	1	0	0
	Time	1,336.6	674.8	155.6	95.8
	Nodes	6,482	4,403	201	25
hole (5)	Fails	0	0	0	0
	Time	98.7	108.5	123.0	130.2
	Nodes	4,519,272	4,519,272	2,887,456.0	2,887,456
ii8/16 (24)	Fails	0	0	0	0
	Time	630.8	628.0	36.0	33.3
	Nodes	177,411	177,441	1,246	1,249
jhn (50)	Fails	0	0	0	0
	Time	2.0	2.1	2.0	1.9
	Nodes	973	972	121	128
morphed (200)	Fails	1	1	1	1
	Time	522.9	523.2	525.1	524.9
	Nodes	8,017,261	6,517,103	5,260,416	3,068,016
par8 (10)	Fails	0	0	0	0
	Time	0.5	0.5	0.7	0.5
	Nodes	192	84	47	50
ssa (8)	Fails	1	0	0	0
	Time	712.3	42.0	27.6	4.2
	Nodes	490,476	410,294	4,604	214
ucsc-ssa (102)	Fails	6	5	0	0
	Time	4,027.3	2,687.9	507.8	40.2
	Nodes	2,737,423	59,997,666	60,779	3,085
dubois (13)	Fails	6	1	6	1.0
	Time	3,717.8	505.8	3,879.8	506.0
	Nodes	335,081,656	19,388,444	287,940,696	19,130,396

Table 3: Comparison of performance on various benchmarks. All problems were run with a time limit of 500 CPU Sec. Figures shown are totals over all problems in the family, counting the time (500 sec) for failed problems. Tests were on a 500MHz PIII.