Satisfiability Testing with More Reasoning and Less Guessing

Allen Van Gelder   Yumi K. Tsuji*

Baskin Center for Computer Engineering and Information Sciences

University of California, Santa Cruz 95064

UCSC-CRL-95-34

avg@cs.ucsc.edu tsuji@cs.ucsc.edu

April 21, 1995

Abstract

A new algorithm for testing satisfiability of propositional formulas in conjunctive normal form (CNF) is described. It applies reasoning in the form of certain resolution operations, and identification of equivalent literals. Resolution produces growth in the size of the formula, but within a global quadratic bound; most previous methods avoid operations that produce any growth, and generally do not identify equivalent literals. Computational experience indicates that the method does substantially less "guessing" than previously reported algorithms, while keeping a polynomial time bound on the work done between guesses. Experiments indicate that, for larger problems, the time investment in reasoning returns a profit in reduced searching, and the profit increases with increasing problem size.

Experimental data compares six branching strategies of the proposed algorithm on a variety of problems, including several Dimacs benchmarks. These branching strategies were shown to perform differently with statistical significance. A new scheme based on Johnson's maximum satisfiability approximation algorithm was found to be the best overall.

Both satisfiable and unsatisfiable random 3-CNF formulas with 50–283 variables and 4.27 ratio of clauses to variables have been tested; this class is generally acknowledged to be relatively "hard" and required extensive backtracking by other algorithms. Unsatisfiable random problems were found to deviate from the easy-hard-easy pattern.

The new algorithm solves random formulas with surprisingly little backtracking: the average number of guesses was 3,267 for 200 variables at this ratio, and 57,503 for 283 variables. Larger unsatisfiable formulas from circuit-fault analysis, with 500–12,800 variables were solved with *no* backtracking in some cases. Extensive statistics on guesses and time are reported. Statistical and experimental techniques and traps are discussed. An exponential growth rate for random formulas is estimated.

Keywords: Satisfiability, Boolean formula, propositional formula, resolution, 2-satisfiability, k-closure.

To appear in *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge*, Johnson, D. S. and Trick, M. *eds.*, American Mathematical Society, 1996.

*Authors were supported in part by NSF grant CCR-8958590.

1 Overview

The problem of Boolean, or propositional, satisfiability is the fundamental NP-complete problem and has many practical applications as well. We assume the reader is generally familiar with it, and give definitions only when needed for clarity. In Section 2, we present the idea of k-closure, an operation on a propositional formula that is in conjunctive normal form (CNF). The method involves a limited use of resolution, called *k-limited resolution* (Definition 2.1). Starting from the given formula, derivation of clauses by k-limited resolution continues until no further such derivations are possible. Since the number of such clauses in a formula of n variables is $O(n^k)$, k-closure takes polynomial time. We describe an algorithm for satisfiability based on k-closure in Section 3.

The k-closure approach is compared with other approaches, and some of its properties are described. An earlier version of this algorithm (presented at the 1993 AAAI Spring Symposium) used 3-closure, but some operations were much too expensive as implemented. The new version reported here, called `2cl`, uses 2-closure, plus certain resolution operations on longer clauses (see Section 4). Two-closure is achievable in quadratic time per eliminated variable.

Section 5 presents the experimental data comparing the different branching strategies used by the proposed `2cl` algorithm on a variety of problems. The *Student's paired t-test* was applied to the number of guesses for these branching strategies to demonstrate the statistical significance of their differences. A new scheme based on Johnson's maximum satisfiability approximation algorithm was found to perform the best on random 3-CNF formulas, and was always close to the best on other varieties.

Section 7 addresses the question of whether the added time for reasoning that is occurred by `2cl` pays dividends in terms of reduced search space (as measured by the number of guesses). The time performance of `2cl` is compared to the traditional satisfiability algorithm of Davis, Putnam, Logemann and Loveland [DP60, DLL62] on families of increasingly large random formulas. Above 2000 literals (about 150 variables) `2cl` emerges as the clear winner.

Achieving acceptable performance depends strongly on efficient data structures. We found that memory was often more of a problem than time for large numbers of variables: for an n variable formula, it was not feasible to use an $n \times n$ array to remember the 2-closure results. Space-efficient data structures often require some overhead in time. Keeping that overhead low was the primary implementation challenge. We believe there is room for significant additional improvement without changing the basic structure of the algorithm.

Appendix A provides further discussion on the statistical traps and techniques for NP-hard problems.

Concerning experimental techniques generally, it was found that variable reordering (Section 6.2) has a significant impact on the performance of satisfiability testing of Boolean formulas associated with structured problems. We investigated the effect of variable reordering by shuffling the variables in the input formula, so that ties would be broken in a variety of ways; a seed for the random number generator used by this shuffling procedure can be changed at the invocation time. The effectiveness of this approach was experimentally confirmed for many of the Dimacs benchmark problems. Appendix B gives the results on the benchmark problems for the *Second DIMACS Challenge*.

2 CNF Formulas and Closures

In this paper we regard a CNF propositional formula in the standard way as a set of clauses, where each clause is regarded as a set of literals. Whether x is a positive or negative literal, $\tilde{x}$ denotes its complement. We may classify CNF formulas by their widest clause as 2-CNF, 3-CNF, etc. Sometimes the shortest clause length is of interest, too. The empty clause is written as $\emptyset$, and represents *false*.

Recall that clause C is said to *subsume* clause D, and D is *subsumed by* C if the literals of C are a subset of those of D. A *tautologous clause* is one that contains both a literal and its complement, can be thought of as *true*, and is considered to be subsumed by any clause. We assume familiarity with the terms *resolution*, *unit clause rule*, and *pure literal rule*.

A *Krom formula* (or 2-CNF formula) has an associated *implication graph*: its nodes are literals of the formula, and clause $\{x, y\}$ induces directed edges $\tilde{x} \rightarrow y$ and $\tilde{y} \rightarrow x$. The notation $a \rightarrow\!\!\rightarrow b$ means that there is a path from literal a to literal b in the implication graph.

We shall use n to represent the number of variables, m for the number of clauses, and L for the number of occurrences of literals, in a formula.

Definition 2.1: Several notions of closure for sets of clauses will be used.

1. A set of clauses S is said to be *closed under resolution*, or simply *closed*, if no clause of S is subsumed by a different clause of S, and the resolvent of each pair of resolvable clauses is implied by (subsumed by) some clause in S.
2. A *closure* of a CNF formula F is a CNF formula that derived from F by a series of resolutions (which add clauses) and subsumptions (which delete clauses), and is closed.
3. The operation of *k-limited resolution* is defined to be resolution in which the operands and the resolvent have at most k literals each.
4. A set of clauses S is said to be *k-closed* if no clause of S is subsumed by a different clause of S, and the resolvent of each possible k-limited resolution with operands in S is implied by (subsumed by) some clause in S.
5. A *k-closure* of a CNF formula F is a CNF formula that derived from F by a series of k-limited resolutions (which add clauses) and subsumptions (which delete clauses), and is k-closed.

It is easy to see that both closure and k-closure are unique. $\square$

By the completeness of resolution, a set of clauses that is closed under resolution is unsatisfiable if and only if it contains the empty clause, $\emptyset$.

The size of the closure of formula F with n propositional variables may be exponential in n. However, the k-closure has a size that is $O(n^k)$, and so can be computed in polynomial time if k is a constant. We shall be interested primarily in $k = 2$ or 3.

2.1 Two-Closure

For a 2-CNF formula, all resolvents have at most two literals, so its 2-closure is also its closure. It follows that computing the 2-closure provides a polynomial-time decision procedure for 2-SAT. Using the technique of strongly connected components, it is possible to *decide* 2-CNF formulas more efficiently by avoiding the explicit construction of the 2-closure [APT79]. However, this method does not compute all available inferences on satisfiable 2-CNF; such inferences are valuable when the 2-CNF is embedded in a larger formula.

Larrabee's algorithm [Lar92] (in effect) uses the 2-closure of the 2-CNF part of more general CNF formulas. This paper extends that idea by adding efficient subsumption resolution, and by incrementally updating the 2-closure and backtracking out of the updates when necessary. Pretolani [Pre95], as well as Jaumard et.al. [JSD95] presented satisfiability algorithms that exploit 2-SAT relaxation. Combining some resolution with searching was also reported by Billionnet and Sutter [BS92], but they do not give enough specifics to permit a detailed comparison.

2.2 Three-Closure

For a 3-CNF formula F, it might happen that its 3-closure is also its closure, in which case absence or presence of the empty clause $\emptyset$ immediately decides satisfiability. In general, of course, the 3-closure is not the closure, as the resolvent of two 3-clauses is normally a 4-clause, which is not in the 3-closure. However, the 3-closure might contain the empty clause $\emptyset$ anyway, demonstrating unsatisfiability.

Definition 2.2: Suppose a set of clauses S contains a variable v such that some resolvent involving v as the annihilated variable is not subsumed by any clause in S. Then v is said to have *implicit information* associated with it. □

If *no* variable has implicit information, then the set S is closed and the satisfiability question is trivial (look for $\emptyset$). It may also be possible to exploit the fact that one variable has no implicit information.

Theorem 2.1: If a variable x contains no implicit information in a CNF formula, then all clauses containing that variable can be deleted without changing the satisfiability of the formula.
Proof: A valid step of the original (resolution-based) Davis Putnam procedure is to perform resolution with x as the annihilated variable, then delete all clauses containing x. But, by Definition 2.2, all resolvents of this operation are already in the formula, or are subsumed by clauses already in the formula. ∎

Unfortunately, testing for implicit information is very expensive. Nevertheless the concept can still be useful. A 2-closed set of binary clauses has no implicit information, which leads to one of our principal guidelines:

Fact It is never necessary to guess an assignment to a variable that occurs only in binary clauses.

Although we do not want to pay to verify the presence of implicit information before choosing a variable to branch on, we can restrict the choice to those that most probably do have such implicit information.

3 The "Ideal" Algorithm

The above considerations suggest an algorithm that performs 3-closure on the current formula, checks for $\emptyset$, then checks whether there is any variable with implicit information. If so, then "guess" an assignment to that variable. If this guess leads to an unsatisfiable formula, then backtrack and "guess" the complementary assignment. In both cases the guessed assignment is added to the current formula as a new unit clause, and the algorithm is called recursively.

It is interesting to compare the above approach with the older algorithms for CNF satisfiability. The first algorithm oriented toward CNF formulas was due to Davis and Putnam [DP60], and was based on variable elimination through resolution. It used the unit clause rule and pure literal rule as heuristics for choosing a variable, but when those were not applicable, it specified to choose a variable from a *shortest clause*. All possible resolutions with this variable are done, and clauses containing this variable (or its complement) are eliminated. There is no "guessing" or branching, but longer and more numerous clauses may be created. Subsumption checking is an option to reduce the size of the formula.

What is often referred to in current literature as the Davis-Putnam algorithm is actually a modification due to Davis, Logemann, and Loveland [DLL62], and will be called the DPLL algorithm here. Neither longer clauses nor new clauses are created by DPLL. The DPLL algorithm uses the unit clause rule (a special case of resolution), and the pure literal rule, but when these are inapplicable, it chooses a variable from a shortest clause and "guesses" an assignment, essentially by adding that literal to the current formula as a new unit clause and recursively calling the algorithm. If this guess leads to an unsatisfiable formula, the complement is guessed and the algorithm again called recursively; this step is called backtracking. If both guesses create unsatisfiable formulas, then the current formula is reported as unsatisfiable. Section 7 presents performance results for DPLL.

However, the chosen variable may or may not contain implicit information. For example, if the current formula contains $(v \vee a)$, $(\tilde{v} \vee b \vee c)$, $(a \vee b \vee c)$, and other clauses, but no more occurrences of v or $\tilde{v}$, then v contains no implicit information, but might be chosen by DPLL as the branching or guessing variable. Several variants of this procedure have been proposed, but all have the same property that the branching variable may have no implicit information. The motivation for choosing a variable in a shortest clause is probably to create more unit clauses.

The 3-closure method combines the ideas of resolution and assignment guessing. By only doing resolution when the resolvent is three or fewer literals, we are assured of a polynomial bound (per guess) on this operation. By always choosing a variable with implicit information for assignment guessing, we bring the formula closer to one whose 3-closure is its closure. We add two further optimizations to reduce formula size:

1. The pure literal rule: requires no comment.

2. Equivalent literal recognition: if $(\tilde{a} \vee b)$ and $(\tilde{b} \vee a)$ are present, then all occurrences of b and $\tilde{b}$ may be replaced by a and $\tilde{a}$, respectively. This reduces the number of variables and usually causes many clauses to be subsumed.

Definition 3.1: In the context of a k-closure algorithm, we say that a k-CNF formula is *k-stable* if it is k-closed, does not contain $\emptyset$, has no pure literals, and has no equivalent literals, as described above. □

Casual experimentation shows that it is quite difficult to construct a 3-stable formula that also contains binary clauses. One example is shown below.

An unusual feature of our "ideal" algorithm is that it may, and normally does, detect satisfiability without constructing a satisfying assignment. Consider this formula

$$(a \vee b),\ (x \vee y),\ (\tilde{a} \vee \tilde{x} \vee \tilde{y}),\ (\tilde{y} \vee \tilde{b} \vee \tilde{a}),\ (b \vee \tilde{x} \vee \tilde{y}),\ (x \vee \tilde{b} \vee \tilde{a}),$$

It is seen to be 3-stable, and is in fact the smallest 3-stable formula containing binary clauses we have been able to construct. Because it is closed, and does not contain the empty clause, it must be satisfiable.

Some applications require an explicit satisfying assignment to be produced. Fortunately, this can be done efficiently by relying on the following observation:

Theorem 3.1: If literal x occurs in a closed formula S, then S has a model in which x is true.
Proof: The formula S has no implicit information, so $\tilde{x}$ cannot be a logical consequence of S by the completeness of resolution. ∎

So a satisfying assignment for a closed k-stable formula S can be found recursively by choosing any literal x in a non-unit clause of S, and then finding a satisfying assignment for the closure of $(S \cup (x))$.

4 The Practical Algorithm

As mentioned before, to determine whether a variable contains implicit information it is necessary first that the formula be 3-closed, and then all possible resolutions must be tried to see if any produce an *unsubsumed* 4-clause. We programmed a short-cut that chooses variable that is likely to have implicit information.

Once we abandon strict implicit information, there is little motivation for performing complete 3-closure. However certain of these operations on "long" clauses (3 or more literals) are efficient and very valuable.

1. *Krom subsumption resolution* removes one literal from a long clause: $\{a, x\}$ and $\{\tilde{a}, x, y, \ldots\}$ resolve to $\{x, y, \ldots\}$, which subsumes the original $\{\tilde{a}, x, y, \ldots\}$.

2. *Simple subsumption* removes a long clause when a binary clause or unit clause implies it. This reduces the "non-information" in long clauses, so branching choices will be more pertinent.

In addition, the following reduction might gain efficiency.

Corollary 4.1: If a variable x (including $\tilde{x}$) occurs only in binary clauses in a 2-closed CNF formula (possibly containing longer clauses), then all clauses containing that variable can be deleted without changing the satisfiability of the formula.
Proof: Variable x has no implicit information, so Theorem 2.1 applies. ▮

While removing such clauses is an option in our implementation, the situation seems to arise rarely, and we have not observed clear-cut benefits in practice.

Our `2cl` algorithm is a modification of DPLL. Between guesses, DPLL repeatedly performs unit clause simplification and pure literal simplification until no further simplifications are possible. Between guesses, `2cl` repeatedly performs 2-closure (which includes unit clause simplification), pure literal simplification, equivalent literal simplification, Krom subsumption resolution, and simple subsumption, until a 2-stable formula (Definition 3.1) is attained.

The implementation for which we report experimental results maintains explicit 2-closure; that is, every derivable, unsubsumed binary clause is represented explicitly in the data structure. Each literal is associated with the sorted list of clauses containing it, so that it is reasonably efficient to locate all long clauses containing two specified literals by the intersection operation. This secondary index supports Krom subsumption resolution and simple subsumption.

Recall that the set of 2-clauses can be regarded as edges in an implication graph. Explicit 2-closure amounts to maintaining its transitive closure.

The important information that may be present in the transitive closure that cannot be detected from the strong component analysis is a path from a literal to its complement, $x \rightarrow\rightarrow \tilde{x}$, from which the unit clause $\tilde{x}$ can be inferred.

However, updating the transitive closure is potentially expensive: following a guess (a backtrackable variable assignment), one new inferred edge can generate $O(n^2)$ secondary updates. Moreover, they all need to be "retracted" upon backtracking. While efficient transitive closure has been much studied, we are not aware of any work that considers the need to backtrack out of updates to the graph. We use a straightforward method in which the transitively closed graph is an array (indexed by variable) of adjacency lists. Each new edge that is added is recorded in a "journal", and upon backtracking, the journal is used to "roll back" the updates, as is common in database systems.

5 Experimental Results

This section presents experimental data on the proposed `2cl` algorithm, with attention to variations that all have the same reasoning component. See Section 7 for comparisons with DPLL, and discussion of the trade-off between reasoning and guessing.

Six branching strategies were tested on a variety of problems, including several Dimacs benchmarks. These branching strategies were shown to perform differently with statistical significance. A new strategy based on Johnson's maximum satisfiability approximation algorithm proved the best for the class of random 3-CNF formulas. However, for the circuit fault-detection formulas the performance of the branching strategies displayed a less clear pattern. *Student's paired t-test* was used to test for the significantly different means of the number of guesses for these branching strategies. Such statistical tests must be applied with care; further discussions on the statistical traps and techniques for NP-complete problems are found in Appendix A.

5.1 Typical Questions

There are numerous satisfiability algorithms and corresponding implementations. We consider the following typical questions that arise in comparative evaluations of satisfiability programs:

1. Are there significant differences among the observed performances of the given set of programs on the same distribution of formulas?
2. Is the relative goodness of programs affected if we use different distributions of formulas?
3. Is performance of an algorithm sensitive to the *presentation* of the formula? By "presentation" we mean choice of variable numbering, clause order, etc.

5.2 Performance Measures

Our principal measure of resource usage, other than CPU time, is the number of "guesses", or branches. A "guess" is a variable assignment that may change the satisfiability of the formula, so its complement may have to be considered (or was considered earlier). The number of guesses has the advantage of being a reproducible measure, so it is used for most presentations. See Appendix A for further discussion.

For the random formulas reported upon here, we have found by regression analysis that CPU time (for SunSS10/41) for `2cl` is modeled quite accurately by the equation

$$cpusec = -.264 + g(.00000304L + .00000000451L^2)$$

where g is the number of guesses and L is the number of literal occurrences. The root mean square error of the model was 3.7 on data whose standard deviation was 111, so, informally, it explains 97% of the variation observed. The regression data included all branching strategies, all tested clause/variable ratios, and most tested formula sizes.

For structured formulas, no adequate regression equation was found. Statistics on guesses and CPU time for both random and structured formulas are given in various tables and figures, as discussed throughout the paper.

5.3 Branching Strategies

To provide some answers to the above questions, we conducted satisfiability experiments on various classes of formulas; we also compared six variants of our implementation of the `2cl` algorithm described in Section 4. These variants correspond to the six different selection criteria for the branching variables.

For all criteria, a *positive* score and a *negative* score are computed for each variable. The variable's final score is the product of the positive and negative components. The eligible variable (eligibility is defined by each branching rule) with the maximum final score is chosen for branching.

Our use of the product, instead of the sum, appears to be unique in the literature. It is motivated by the desire to reduce the size of each subproblem substantially. For example, if branching on x achieves reductions

Circuit Family	statistics	Number of Branches maxscore	minlen	minlen23	maxlen	maxlen23	dsj
ssa0432	Mean	184	527	184	511	184	268
	StdDev	86	263	86	194	86	136
0 sat	Min	70	220	70	278	70	80
7 unsat	Median	202	494	202	496	202	320
	Max	272	920	272	858	272	400
bf0432	Mean	1398	5066	1398	1853	1270	1407
	StdDev	2641	8216	2641	1980	2233	2429
1 sat	Min	6	12	6	14	6	6
20 unsat	Median	250	1296	250	851	250	240
	Max	10968	31214	10968	6310	8756	8412
ssa7552	Mean	25	3446	25	22	23	23
	StdDev	5	6438	5	7	7	6
80 sat	Min	12	7	12	8	9	7
0 unsat	Median	25	31	25	25	24	22
	Max	39	18040	42	37	39	34

Circuit Family	statistics	CPU seconds for SunSS10/41 maxscore	minlen	minlen23	maxlen	maxlen23	dsj
ssa0432	Mean	0.68	1.27	0.73	1.07	0.68	0.77
	Max	0.77	1.69	0.86	1.38	0.77	0.91
bf0432	Mean	6.67	20.33	6.81	11.07	6.41	5.85
	Max	30.22	103.69	32.30	33.85	26.73	22.30
ssa7552	Mean	2.00	2.69	2.02	2.02	2.00	1.93
	Max	2.86	6.04	2.86	2.81	2.87	2.77

Figure 1: The statistics for the number of guesses and the running time by the six branching strategies of `2cl` on circuit fault-detection formulas. More information on the number of variables and literals for these groups of formulas can be found in Figure 7.

of 4 and 18, while branching on y achieves reductions of 10 and 10, we prefer y, with the higher product, rather than higher sum.

maxscore All variables are eligible; score is sum of occurrences in all clauses.

minlen Variables that occur in a minimum length clause are eligible; score is sum of occurrences in minimum length clauses.

minlen23 Variables that occur in a minimum length clause are eligible; score is sum of occurrences in all clauses.

maxlen Variables that occur in a clause of length at least 3 are eligible; score is sum of occurrences in clauses of length at least 3. (If there are only binary clauses, then this reverts to `minlen`.)

maxlen23 Variables that occur in a clause of length at least 3 are eligible; score is sum of occurrences in all clauses. (If there are only binary clauses, then this reverts to `minlen`.)

dsj All variables are eligible; score is the *weighted* sum of occurrences in all clauses. Binary clauses count twice as much as 3-clauses, and 3-clauses count twice as much as the clauses of longer length. This is the modified version of the weighting used in D. S. Johnson's maximum satisfiability approximation algorithm [Joh74].

clauses	samples	`maxscore`	`minlen`	`minlen23`	`maxlen`	`maxlen23`	`dsj`
300	100	16.78	15.91	16.61	15.31	16.83	17.20
		± 3.18	± 3.17	± 3.12	± 2.79	± 3.21	± 3.28
400	100	40.45	32.30	40.39	105.51	40.45	34.82
		± 43.67	± 33.59	± 40.90	± 148.54	± 43.64	± 33.42
427	100	92.00	75.70	85.95	226.43	92.00	70.30
		± 70.17	± 56.71	± 65.33	± 189.58	± 70.17	± 51.33
450	100	101.84	77.33	91.74	230.79	101.84	78.47
		± 47.85	± 32.63	± 41.72	± 129.46	± 47.85	± 32.54
550	100	45.36	31.50	39.72	74.60	45.36	36.64
		± 14.35	± 9.06	± 12.59	± 28.84	± 14.35	± 10.34

Figure 2: The average number of guesses by the six branching strategies of `2cl` on random 3-CNF formulas (constant width model) with 100 variables and varying numbers of clauses. Standard deviations are prefixed by "±".

> Jeroslow and Wang [JW90] have reported on a satisfiability algorithm that uses a somewhat similar branching strategy, except they maximize over positive and negative components, where we multiply. Their motivation is "most likely to satisfy".
>
> Hooker and Vinay have challenged the "most likely to satisfy" explanation (see also Section 8), and have proposed the "2-sided Jeroslow-Wang" rule, with the motivation "maximize size reduction" [HV94]. This rule is like `dsj`, except that it adds, where we multiply.

There may be multiple candidates for the choice of a branching variable. A tie between any two such variables is broken by selecting a lower or higher variable number depending on the parity of their sum. We provide further randomization by shuffling the variable numbers at the preprocessing stage.

5.4 Observations on the Branching Strategies

The six branching strategies were run on some classes of circuit fault-detection formulas. Results appear in Figure 1. We observed evident weakness of `minlen` strategy on the 432.bf family. Due to the small sample sizes and large variance for these classes of formulas we have not been successful in establishing significance of the differences in the average number of guesses. Figure 7 presents additional results for `dsj` only on a larger set of circuit fault-detection formulas.

The six branching strategies were also run on random 3-CNF formulas, which were generated according to the "constant width" model: each triple of distinct variables is equally likely to be selected for a clause, each variable is signed plus or minus with equal probability, and all clauses are drawn independently "with replacement".

Results appear in Figure 2. The `dsj` and `minlen` branching strategies had the lowest overall averages. Because of the poor performance of the `minlen` strategy on circuit fault-detection formulas, we chose `dsj` as the best candidate. To determine the statistical significance of the results, the `dsj` strategy was compared against each of the other five on five samples of varying clause/variable ratio, making 25 cases in all.

The application of the Student's paired t-test [WEC91], using a published C-language implementation [PTVF92], (reviewed in Appendix A) yields the conclusion that `dsj` branching strategy outperforms others with statistical significance (at level .02) in 16 cases, underperforms significantly in 3 cases, and is not significantly different in 6 cases. The details, including exact probability values, are given in Figure 3.

clauses	samples	`maxscore`	`minlen`	`minlen23`	`maxlen`	`maxlen23`
300	100	0.2817	$\downarrow *$ 0.0017	0.1188	$\downarrow *$ 0.0000	0.3463
400	100	$*$ 0.0091	0.2654	$*$ 0.0139	$*$ 0.0000	$*$ 0.0090
427	100	$*$ 0.0000	0.0997	$*$ 0.0000	$*$ 0.0000	$*$ 0.0000
450	100	$*$ 0.0000	0.5760	$*$ 0.0000	$*$ 0.0000	$*$ 0.0000
550	100	$*$ 0.0000	$\downarrow *$ 0.0000	$*$ 0.0000	$*$ 0.0000	$*$ 0.0000

Figure 3: Student's paired t-test significance probabilities of the mean difference from the `dsj` branching strategy on random 3-CNF formulas (constant width model) with 100 variables. We indicate the significance level of 0.02 for the paired t-test by "$*$". For each entry with a significant mean difference, `dsj` performed better unless it is also marked with "$\downarrow$".

		satisfiable			unsatisfiable		
clauses	variables	samples	branches	time	samples	branches	time
300	100	100	17.20	0.07	0	NA	NA
400	100	93	27.74	0.33	7	128.86	1.46
427	100	61	41.31	0.52	39	115.64	1.49
450	100	17	32.18	0.43	83	87.95	1.19
550	100	0	NA	NA	100	36.64	0.58

Figure 4: The different difficulty patterns of satisfiable and unsatisfiable random 3-CNF formulas. Time is in seconds for SunSS10/41.

These tests convinced us that the new `dsj` strategy was the best among the alternatives considered for `2cl`. Most subsequently reported experiments use this strategy exclusively.

5.5 Satisfiable vs. Unsatisfiable Random Formulas

We have tested both satisfiable and unsatisfiable random 3-CNF formulas with 100 variables and from 300 to 550 clauses, including 427 clauses, believed to be the hardest point in the spectrum [MSL92, LT92]. The result for the `dsj` branching strategy appears in Figure 4. This figure shows:

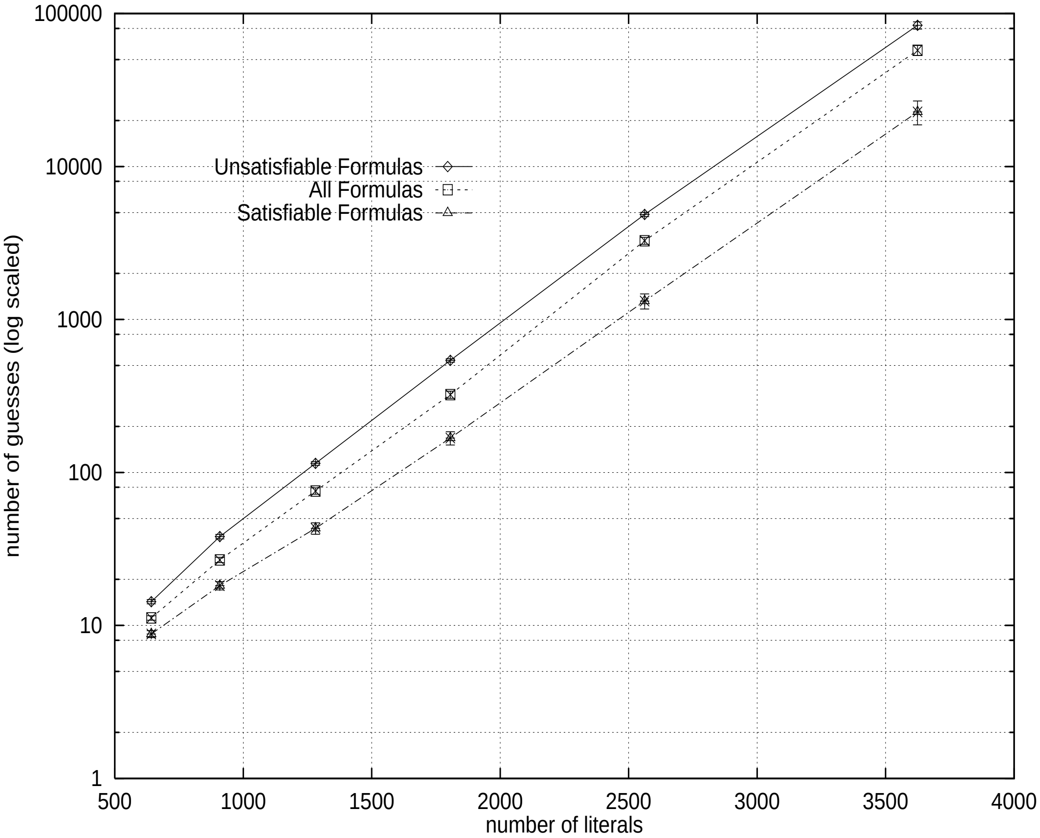
1. Unsatisfiable random problems do not follow the easy-hard-easy pattern.

2. Satisfiable problems are much easier on balance.

5.6 Growth Rate Function on Random 3-CNF Formulas

We have also tested a range from 50 to 283 variables maintaining the 4.27 clause-to-variable ratio. The `2cl` algorithm has solved them with surprisingly little backtracking: for 200 variables, the average number of guesses was 3,267; for 283 variables, the average number of guesses was 57,503. For 283 variables the number of guesses ranged from 105 to 127,538 for satisfiable, and from 25,834 to 182,006 for unsatisfiable. CPU times are reported in Section 7. Figure 5 summarizes the results.

Growth rate was analyzed separately for unsatisfiable and satisfiable formulas, since the satisfiable are 3 to 4 times easier, and the fraction of satisfiable formulas at a fixed ratio changes with the number of variables.



Result	Vars	Total literals	Sample Size	Branches Ave	Stderr	Stddev	Min	Median	Max
unsat	50	642	87	14	0	4	6	14	30
	71	909	87	38	1	12	18	36	72
	100	1281	91	115	3	31	48	116	212
	141	1806	83	540	14	132	208	524	944
	200	2562	110	4860	159	1671	2164	4454	9836
	283	3624	57	83706	4505	34010	25834	75780	182006
sat	50	642	113	9	0	5	3	8	29
	71	909	113	18	1	12	4	14	58
	100	1281	109	43	4	39	6	27	175
	141	1806	117	168	17	181	10	105	1151
	200	2562	90	1320	149	1409	18	835	7002
	283	3624	43	22768	4057	26601	105	10618	127538

Figure 5: Growth rate of `2cl` on random 3-CNF formulas using the `dsj` option. The average number of guesses is plotted against the number of literals on a logscale. The formulas generated all have the clauses-to-variables ratio of 4.27.

Family	Fmlas	Solved	Cpu secs (SunSS10/41) / Branches Ave	Stddev	Min	Median	Max
ii8	14	7	203.88	283.01	0.07	108.14	779.27
			31534	45902	8	22185	132210
ii16	10	2	1839.54	2476.61	88.31	88.31	3590.77
			10674	14854	170	170	21177
ii32	17	16	251.91	571.64	0.36	1.78	1985.41
			2753	6634	67	142	26215

Figure 6: Performance of `2cl` with `dsj` branching strategy on "inductive inference" Dimacs benchmark formulas.

Circuit Family	# of Fmlas	# of Sat Fmlas	Mean # of Vars	Mean # of Lits	Number of Branches Ave	Stddev	Min	Median	Max
ssa0432	7	0	501	2481	268	137	80	320	400
ssa2670	12	0	1530	7835	520449	260399	99642	507782	866328
ssa6288	3	0	12836	92216	0	0	0	0	0
ssa7552	80	80	1626	8208	23	6	7	22	34
bf0432	21	1	1183	8296	1407	2430	6	240	8412
bf1355	149	0	2829	19319	13254	9038	2	12742	31194
bf2670	53	16	1531	8366	109347	576585	2	2428	4174892
					CPU seconds for SunSS10/41 Ave	Stddev	Min	Median	Max
ssa0432	7	0	501	2481	0.77	0.12	0.61	0.77	0.91
ssa2670	12	0	1530	7835	1539.60	639.34	177.16	1483.45	2424.46
ssa6288	3	0	12836	92216	39.45	6.10	32.41	42.95	43.00
ssa7552	80	80	1626	8208	1.93	0.28	1.47	1.92	2.77
bf0432	21	1	1183	8296	5.85	6.48	0.75	3.04	22.30
bf1355	149	0	2829	19319	41.81	20.85	2.94	37.70	86.29
bf2670	53	16	1531	8366	439.01	2436.85	0.79	14.18	17724.57

Figure 7: Performance of `2cl` with `dsj` branching strategy on the Circuit formulas.

However, no significant difference in growth rate was observed between these two groups. See Figure 5.

Our experiments for the `dsj` branching strategy indicate an exponential growth rate of $2^{.0039L}$, where L is the number of *occurrences of* literals in the formula. The estimated coefficient, $C = .0039$, was obtained by fitting the equation

$$mean\ branches = A2^{C \cdot L}$$

to the observed values of 57,503 for $L = 3624$ and 3,267 for $L = 2562$. Based on the standard errors in Figure 5, the standard error of C is about .0001. At this ratio variables occur an average of 12.81 times, so the growth rate in terms of n variables appears to be $2^{.050n}$; this coefficient's standard error is about .001.

5.7 Observations on the "Inductive Inference" Formulas

The Dimacs "inductive inference" benchmark formulas were attempted with spotty results. Results appear in Figure 6. With a limit of 1 hour CPU time imposed, the algorithm only solved 25 out of 41 problems. All of the tests given in the result were done with the same variable shuffling algorithm. Without the variable shuffling we observed that one more formula was solved within the given time limit; however, one formula was no longer solved with the net result being very similar.

5.8 Observations on the Circuits Formulas

Figure 7 shows the `2cl` result on all the "circuits" formulas that have been submitted to the DIMACS database of *cnf* formulas. Perhaps the most remarkable result in this group is that the three unsatisfiable `ssa6288` formulas, with an average of 12,836 variables, were solved with no guessing at all. We are not aware of any other algorithm that has succeeded solving these formulas. All formulas in the group were solved, although one required 5 CPU hours.

6 Experimental Techniques

This section discusses two techniques used in our experiments: the first one assures the randomness of the formulas while maintaining the repeatability of the experiment; the second provides information about the sensitivity of an algorithm to accidents in the presentation of the formula.

6.1 Seed Conversion

The basic assumption in the experiments on the randomly generated formulas is that the input formulas are chosen independently at random from a specified parent population and distribution. Such randomness is typically simulated by a formula generator that uses a reliable "random number generator" whose initial seed value is either produced internally or provided at invocation time by the user. In either case, unless one remembers all the initial seeds, the repeatability of the experiment is lost. This becomes an unreasonable burden for the experimenter, since easily remembered set of seeds can compromise the randomness factor. For example, if the same initial seed was used to generate a formula of 400 clauses and one with 500 clauses, both with the same number of variables, then there is a dependence between those two formulas; in fact, one is a prefix of the other.

To avoid this problem, for those experiments in Sections 5.4, 5.6 and 7, we used a script that converts a user specified seed in an easily remembered range, such as 1–200, into a unique actual seed to be used by the formula generator via the following formula:

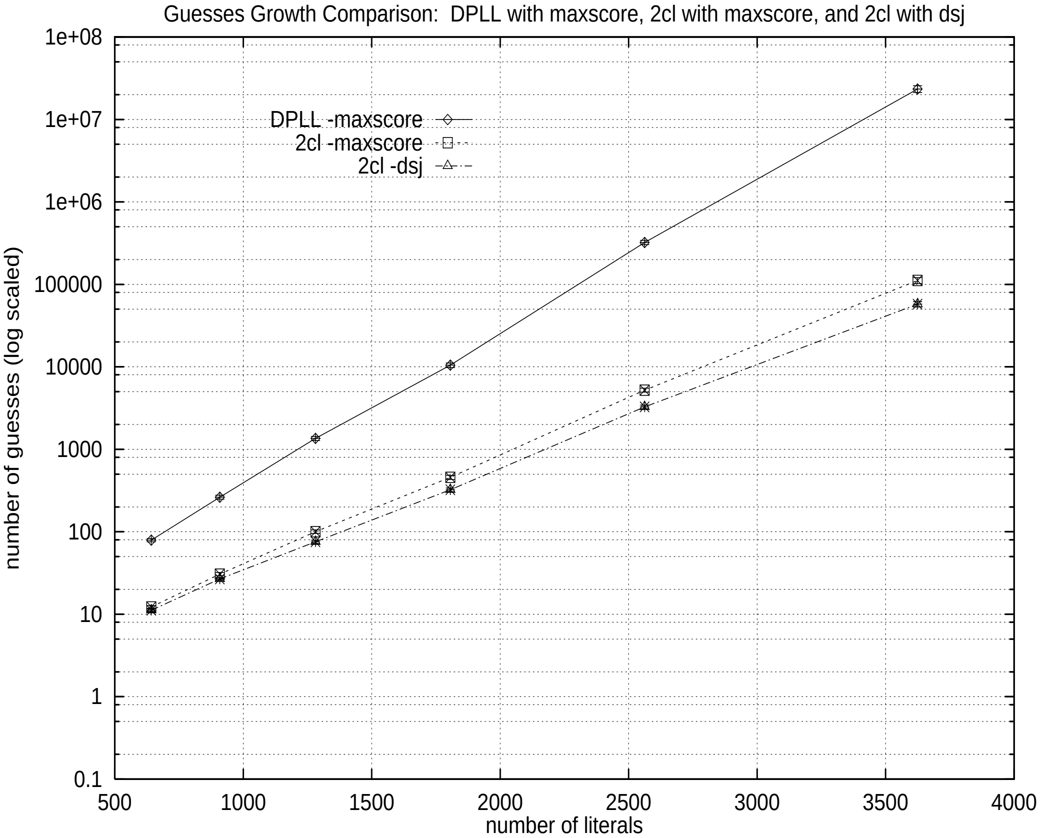
$$((v + 59 * c) * 123 + i) \bmod 1000000$$

where v, c, and i are the parameter values corresponding to the number of variables, the number of clauses, and the user specified seed, respectively. The same range, say 1–200, can be used for a variety of values of v and c without incurring any dependence.

6.2 Internal Variable Shuffling

To determine sensitivity to the formula presentation, we provide the option to shuffle internal variable numbers at the preprocessing stage. By default, variables are internally numbered in order of first appearance, and these numbers may determine how ties are broken. This shuffling does not change the formula in an essential way. It is also referred to as variable reordering.

The programs we used in this experiment have an execution time option to shuffle the variable numbers in the input formula at the preprocessing stage. The default seed value is 0 to indicate no shuffling, but

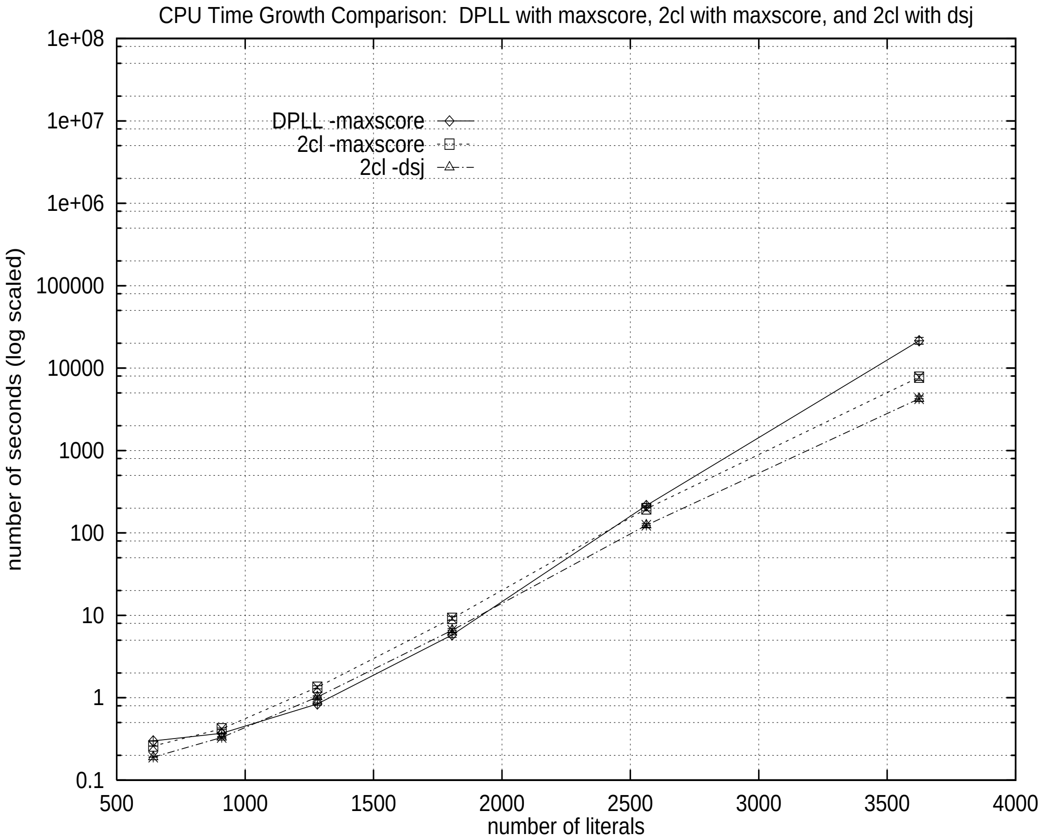


number of variables	number of literals	Number of guesses: DPLL maxscore Mean	DPLL maxscore StdErr	2cl maxscore Mean	2cl maxscore StdErr	2cl dsj Mean	2cl dsj StdErr
50	642	79	4	12	0	11	0
71	909	263	14	31	1	27	1
100	1281	1354	79	100	5	76	4
141	1806	10474	672	457	26	322	17
200	2562	322426	19625	5216	282	3267	166
283	3624	23420708	2232070	111590	9233	57503	4329

Figure 8: Growth rate comparison of `2cl` and `DPLL` on random 3-CNF formulas. The average number of guesses is plotted against the number of literals on a logscale. The formulas generated all have the clauses-to-variables ratio of 4.27.

user may specify any integer as the seed for the variable shuffling. This technique was used as a tool in investigating the sensitivity of the algorithmic performance to the formula presentation. As indicated in Section 5.7 and in Appendix B, variable shuffling by `2cl` could make a big performance difference in some structured formulas. However, it made very little difference in random formulas.

7 Time Trade-Offs for Reasoning vs. Guessing



number of variables	number of literals	Mean CPU seconds for SunSS10/41 DPLL maxscore	2cl maxscore	2cl dsj
50	642	0.30	0.26	0.19
71	909	0.37	0.42	0.33
100	1281	0.84	1.34	1.02
141	1806	5.80	9.21	6.60
200	2562	214.65	195.96	123.84
283	3624	21500.14	7743.29	4256.60

Figure 9: Growth rate comparison of `2cl` and `DPLL` on random 3-CNF formulas. The average number of CPU seconds (Sun equivalent, as explained in text) is plotted against the number of literals on a logscale. The formulas generated all have the clauses-to-variables ratio of 4.27.

The theme of `2cl` is to reduce the size of the search space, as measured by the number of guesses, by incorporating an efficient reasoning component into the search. A natural question is whether the time invested in reasoning pays dividends in terms of reduced search. To address this question we compared `2cl` with an efficient implementation of DPLL. The same samples of random formulas reported in Section 5.6 were used, ranging from 50 to 283 variables at the 4.27 ratio; all sizes had 200 samples, except that 283 had 100 samples. Besides relative magnitudes, we were especially interested in growth rates.

Because the theme of DPLL is to guess as fast as possible, it is not consistent to spend time maintaining the information needed to implement the `dsj` strategy. However, the `maxscore` strategy *can* be implemented with little overhead, and proved to be far superior to the published strategy of choosing "any variable in a minimum-length clause" (on random formulas, at least). Results are based on the `maxscore` strategy for DPLL. We ran `2cl` with both the `maxscore` and `dsj` strategies.

Figure 8 shows the growth rates in terms of guesses for both algorithms. Unsurprisingly, `2cl` does less guessing. The important observation is that exponential growth rate for guessing is much lower for `2cl` than for DPLL, as shown by the diverging lines. Assuming these rates persist to larger problems, it is *inevitable* that `2cl` will eventually require less time, because it spends only a polynomial factor greater time per guess than DPLL.

Using the method of Section 5.6, where `dsj` was found to grow at the rate $2^{.0039L}$, here we find that `2cl` with `maxscore` grows at $2^{.0042L}$, while DPLL grows at $2^{.0058L}$. (The standard errors of all of these coefficients is about .0001.)

Figure 9 shows the growth rates in terms of CPU time for both algorithms. DPLL was executed on SunSS10/41 while `2cl` was executed on a slower DECStation 5000/240. The CPU times measured on DECStation 5000/24 were transformed to the SunSS10/41 equivalent times by using the conversion factor of 0.5813. This graph confirms that "the future is now". For the `dsj` branching strategy, the extra reasoning begins showing a profit around 150 variables (2000 literals), and the margin widens rapidly for larger sizes. Roughly speaking, by considering time instead of guesses, the DPLL curve translates downward relative to `2cl`, but still has the greater slope. To show that the performance difference between `2cl` and DPLL is attributable to reasoning, and not a difference in branching strategies, we also ran `2cl` with the less effective `maxscore` strategy, which is the strategy that worked best for DPLL. We see the similar situation in this case, although the cross-over point with the DPLL is higher, about 200 variables (2600 literals).

8 Conclusion and Future Work

The combination of reasoning and guessing seems to be beneficial. Careful implementation is needed to keep the cost of reasoning down. Further investigation is needed to determine whether the expensive 3-closure operations can be done more efficiently and put back into service. Analysis of the asymptotic complexity, both of the worst case, and of the average case on random formulas, is another avenue of investigation. There are some known results on the worst-case performance of satisfiability on CNF formulas using algorithms that add various forms of reasoning to the basic DPLL framework [MS85, VG88, Sch93, Kul94, KL95]. For random CNF formulas of length L at the 4.27 ratio studied here, the smallest known worst case bound is $2^{.046L}$. This is still far above our observed growth rate of $2^{.0039L}$.

Statistical techniques, including nonparametric methods [GC92], for the comparative evaluation of different strategies is another topic that can be further pursued. Distributions for NP-hard problems are typically highly skewed. For most of the structured, i.e., nonrandom formulas, we observed that the maximum number of guesses was 2 to 3 times the second largest number of guesses in the same sample. Consequently, it is important to report a variety of statistics to give a good picture of the algorithm's behavior. Besides the average and standard deviation, we have also reported the minimum, median, and maximum.

Hooker has advocated an empirical approach to algorithm analysis that involves hypothesis formation and prediction, in analogy with natural sciences [Hoo94]. Many heuristics are proposed in the literature, with informal explanations of why they work well. Hooker believes that the explanations should be examined more critically, and proposes a framework for doing so: if the explanation is "correct", then we should be able to make new predictions and verify them. If, on the other hand, the predictions are not borne out, then that indicates that the intuitive explanation has somehow not really captured the reason for the heuristic's observed success. In the latter case, we should begin looking for a better explanation. Hooker and Vinay have reported on an application of this technique, as discussed in Section 5.3 [HV94]. Gent and Walsh indirectly applied the technique to arrive at negative conclusions about the importance of greediness and randomness in local search [GW93]. This technique seems to offer great promise for discovery of better algorithms for intractable problems.

Acknowledgements

We thank the anonymous referee for many suggestions on improving the paper. Many of the experiments were facilitated by equipment that was donated by Sun Microsystems, Inc. Both authors were supported in part by NSF grant CCR-8958590.

References

[APT79] B. Aspvall, M. Plass, and R. Tarjan. A linear-time algorithm for testing the truth of certain quantified Boolean formulas. *Information Processing Letters*, 8(3):121–123, March 1979.

[BS92] A. Billionnet and A. Sutter. An efficient algorithm for the 3-satisfiability problem. *Operations Research Letters*, 12:29–36, July 1992.

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

[DP60] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the Association for Computing Machinery*, 7:201–215, 1960.

[GC92] J. D. Gibbons and S. Chakraborti. *Nonparametric Statistical Inference.* Marcel Dekker, Inc., 1992.

[GW93] I. P. Gent and T. Walsh. Towards an understanding of hill-climbing procedures for SAT. In *Proceedings of the Eleventh National Conference on Artificial Intelligence; AAAI-93 and IAAI-93 (Washington, DC, USA, 11-15 July 1993)*, pages 28–33. Menlo Park, CA, USA: AAAI Press, 1993.

[Hoo94] J. N. Hooker. Needed: An empirical science of algorithms. *Operations Research*, 42(2):201–12, March-April 1994.

[HV94] J. N. Hooker and V. Vinay. Branching rules for satisfiability. In *Third International Symposium on Artificial Intelligence and Mathematics, Fort Lauderdale, Florida*, 1994.

[Joh74] D. S. Johnson. Approximation algorithms for combinatorial problems. *Journal of Computer and System Sciences*, 9:256–278, 1974.

[JSD95] B. Jaumard, M. Stan, and J. Desrosiers. Tabu search and a quadratic relaxation for the satisfiability problem. In D. S. Johnson and M. Trick, editors, *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge.*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society, 1995.

[JW90] R. Jeroslow and J. Wang. Solving propositional satisfiability problems. *Annals of Mathematics and Artificial Intelligence*, 1:167–187, 1990.

[KL95] O. Kullmann and H. Luckhardt. Various complexity upper bounds for decisions on propositional tautology. *Information and Computation*, 1995. To apppear.

[Kul94] O. Kullmann. Methods for 3-SAT-decision in less than 1.5045^n steps. Technical report, University of Frankfurt, 1994.

[Lar92] T. Larrabee. Test pattern generation using Boolean satisfiability. *IEEE Transactions on Computer-Aided Design*, 11(1):6–22, January 1992.

[LT92] T. Larrabee and Y. Tsuji. Evidence for a satisfiability threshold for random 3CNF formulas. Technical Report UCSC–CRL–92–42, UC Santa Cruz, Santa Cruz, CA., October 1992.

[MS85] B. Monien and E. Speckenmeyer. Solving satisfiability in less than 2^n steps. *Discrete Applied Mathematics*, 10:287–295, 1985.

[MSL92] D. Mitchell, B. Selman, and H. Levesque. Hard and easy distributions of SAT problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92), San Jose, CA.*, pages 459–465, July 1992.

[Pre95] D. Pretolani. Efficiency and stability of hypergraph SAT algorithms. In D. S. Johnson and M. Trick, editors, *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge.*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society, 1995.

[PTVF92] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C.* Cambridge University Press, second edition, 1992.

[Sch93] I. Schiermeyer. Solving 3-satisfiability in less than 1.579^n steps. In *Computer Science Logic. 6th Workshop, CSL '92 (San Miniato, Italy, 28 Sept.-2 Oct. 1992) LN Comp. Sci. 702 (1993)*, pages 379–94. Berlin, Germany: Springer-Verlag, 1993.

[VG88] A. Van Gelder. A satisfiability tester for non-clausal propositional calculus. *Information and Control*, 79(1):1–21, October 1988.

[WEC91] J. Welkowitz, R. B. Ewen, and J. Cohen. *Introductory Statistics for the Behavioral Sciences.* Harcourt Brace Jovanovich College, fourth edition, 1991.

A Statistical Traps and Techniques for NP-Complete Problems

NP-hard problems can produce highly skewed distributions of certain performance measures. Care is needed to avoid erroneous interpretations and unjustified conclusions.

Consider the performance of 4 algorithms on randomly generated formulas from the same family. The table below shows averages for independent samples of 1000 runs, in cpu times and branching.

Algorithm	Average CPU Secs.	Average Branches
A	3.36	120.75
B	3.42	124.23
C	3.44	124.65
D	3.61	130.44

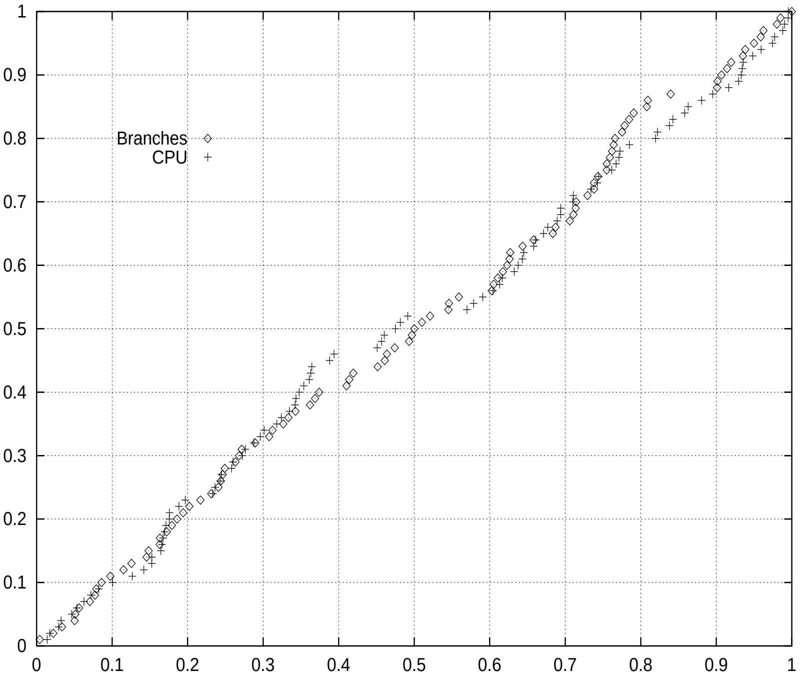


Figure 10: Cumulative distribution of 100 independent samples of the Student significance statistic S

Algorithms B and C look pretty indistinguishable. But with 1000 samples, clearly A is better than D, right?

Wrong! All four runs were made with the *same* deterministic algorithm, with independent randomly generated formulas. This data is not "cooked up". It is the summary of our only attempt to prove our point, not the worst case over many attempts.

Trap 1 Look only at the averages.

It is well known that standard deviations or other measures of dispersion are necessary to know the accuracy of averages, but we see a great many tables with just averages, and no measures of their accuracy. In the above data, the standard deviations were *greater than* the averages.

A fairly standard way to test for a significant difference between two average values is the *Student's t-test* [WEC91]. This test was derived assuming that both samples were drawn from Gaussian distributions with the same variance, but possibly different means (average values).

Trap 2 Use the Student t-test without knowing whether the actual distribution is approximately Gaussian.

The Student t-test has a reputation for being "robust", that is, not overly sensitive to the distribution assumptions. However, exponential time algorithms can stress those assumptions beyond the limits foreseen by traditional statisticians.

Is the Student t-test still reliable, although the distribution is decidedly non-Gaussian? Under ideal conditions, if two samples are drawn from the *same* distribution, the end result of their Student t-test, call it S, is a uniformly distributed random variable in the range 0 to 1. That is why, upon observing S to be 0.05 in an experiment, one can say, "If the two distributions were identical (the *null hypothesis*), the probability of getting an S this small or smaller is 0.05."

So construct some samples where you know the null hypothesis holds! Calculate the Student statistic (S, not t) over numerous independent pairs of such samples. These values of S should appear to be uniformly distributed. If not, the distribution is too skewed for the Student t-test to be accurate.

We carried out this experiment using our program with both the number of branches and the CPU time in seconds as the measures. We ran 100 pairs of samples, each consisting of 100 independent formulas (20000 runs in all). In every case, `2cl` with the `dsj` branching rule was run with a random input from the same family of formulas (100 variables, 400 clauses). This yielded 100 independent S values whose cumulative distribution is plotted in Figure 10. The slightly concave appearance suggests that the distribution is shifted a little to the high end, making it conservative. Because our program is designed to reduce branching, we suspect the skewness is lower than might be exhibited in other programs.

As mentioned, exponential time algorithms can stress the assumptions on underlying distributions beyond the limits foreseen by traditional statisticians.

Trap 3 Fail to distinguish between random variables with no known upper bounds or known variances and those for which these quantities are estimable and of reasonable size.

Informally, the problem is this: to know how accurate an estimate of the mean is, one must know the standard deviation. But again, only an estimate of the standard deviation is available, so how does one know how accurate *that* is?

Standard statistical methods handle this question by assuming that the underlying distribution is Gaussian. This is not realistic in this setting. The only constraint on the distribution that we can expect in practice is a firm upper bound.

Some examples having firm upper bounds include:

- Any probability is bounded between 0 and 1 and has standard deviation at most 0.5.
- The size of the optimum, for most optimization-based NP-complete problems is easily bounded by a low degree polynomial.

On the other hand, the CPU time required to solve a random problem in a given family normally has no known upper limit or known variance. As mentioned, without knowing something about the distribution, one cannot say how accurate is an *estimate* of the variance, calculated from a sample.

We strongly advocate measuring something in addition to CPU time, for performance evaluation. The alternate measure should preferably be strongly correlated to CPU time, but it should be a measure for which an upper bound is known for the whole sample space, *a priori*. For example, in simple DPLL, 2^n is an easy upper bound on the number of branches. Closer analysis can sharpen that. Therefore, some firm statistical statements can be made about the accuracy of an estimate of the average number of branches required over some sample space.

Trap 4 Use the Student t-test when the *pairs* test is applicable.

To compare two algorithms' performance, it is common sense to eliminate as much noise as possible, by running them on the same set of problems. But it is a common mistake to try to work with averages and standard deviations of the two samples, rather than all the data. The pairs test compares two algorithms, input by input, and computes the *difference* in the performance measure for each input. Thus the noise due to varying difficulty of inputs is eliminated. The standard deviation of the difference may well be an order of magnitude lower than the standard deviations of the samples themselves, and much higher discrimination results.

clauses	samples	maxscore	minlen	minlen23	maxlen	maxlen23
300	100	0.3589		0.1937		0.4205
400	100	$*$ 0.3071	0.5954	$*$ 0.2929		$*$ 0.3070
427	100		0.4811	$*$ 0.0611		
450	100		0.8049			
550	100			$*$ 0.0602		

Figure 11: Student's t-test significance probabilities of the mean difference from the `dsj` branching strategy on random 3-CNF formulas. Only those entries that are *not* significant at level 0.02 are shown. Those marked by "$*$" showed significance in Figure 3, based on the paired t-test for same data.

The phenomenon is illustrated with the data for comparison of `dsj` with five alternative strategies, from Section 5.4. Figure 11 shows the *insignificant* entries (at level 0.02) that would result from using the non-paired t-test. Of these, those that were significant in Figure 3 are marked with a "$*$". For this data and significance level, the paired t-test finds 5 out of 11 additional significant differences.

B Second DIMACS Challenge Satisfiability Benchmark Results

GENERAL INFORMATION

Authors: Allen Van Gelder and Yumi K. Tsuji (University of California, Santa Cruz)
Title: Satisfiability Testing with More Reasoning and Less Guessing
Name of Algorithm: 2cl
Brief Description of Algorithm: Complete: Combination of branching and limited resolution

Type of Machine: SunSS10/41
Compiler and flags used: gcc (version 2.4.5) -O2

MACHINE BENCHMARKS

User time for instances:

r100.5	r200.5	r300.5	r400.5	r500.5
0.04	0.93	8.05	49.06	189.29

ALGORITHM BENCHMARKS

Authors' Comments:

- Parameters: For this Appendix, the following invocation parameter values were used.
  - **Branching Strategy:** The program can use any of the six different branching strategies; we selected the one based on Johnson's maximum satisfiability approximation algorithm.
  - **Randomization:** The program shuffles the order of the variables; there is an invocation parameter to set a seed value for the random number generator used by this shuffling procedure. The multiple runs presented in the Appendix were produced by varying this initial seed.

- Additional columns in the Table:

  We have two additional columns in the Appendix table. The column named *RelStdErr* refers to the relative standard error of the mean. It is calculated as the percentage of the standard error of the mean with respect to the mean: $(Std.Dev * 100)/(Mean * \sqrt{n})$. The last column, *Median*, refers to the $int((n+1)/2)$-th value in the ordered list of the "cpu" values of the runs in each sample.

- Failed Runs:

  The failed runs given in the table had the maximum time limit set to 7 hours. For the ii32d3 and par32-2-c formulas, we have also tried 100 different seeds to do the variable scrambling with the time limit set to 10 minutes for each; they had all failed.

- Note on the Special sequences:

  We found that moving the first 52 clauses to the end of the formula permitted the program to succeed in 2 seconds with 2046 guesses on pret60_25 and pret60_75. These runs are not included in the table.

Results on Benchmark Instances:		Time				
Name	Runs(Fail)	Min	Avg	(StdDev)	Max	Result
aim-100-2_0-no-1	100	0.06	43.45	(22.31)	93.88	No
aim-100-2_0-no-2	100	0.09	24.19	(17.28)	65.61	No
aim-100-2_0-no-3	100	0.52	14.48	(7.35)	29.89	No
aim-100-2_0-no-4	100	0.03	25.64	(17.79)	69.16	No
aim-100-2_0-yes1-1	100	0.53	0.73	(0.09)	0.98	Yes
aim-100-2_0-yes1-2	100	0.04	0.50	(0.41)	1.25	Yes
aim-100-2_0-yes1-3	100	0.64	0.85	(0.11)	1.25	Yes
aim-100-2_0-yes1-4	100	0.04	0.22	(0.12)	0.58	Yes
bf0432-007	100	20.77	22.85	(1.59)	35.71	No
bf2670-001	100	4.03	4.43	(0.29)	6.57	No
dubois20	100	18.34	36.51	(11.42)	94.87	No
dubois21	100	23.60	64.27	(21.24)	151.00	No
f400	2 (1)		10869.85			Yes
f800	DNR					
f1600	DNR					
f3200	DNR					
f6400	DNR					
g125.17	1 (1)					
g125.18	1 (1)					
g250.15	DNR					
g250.29	DNR					
ii32b3	100	2.88	16.47	(32.31)	113.78	Yes
ii32c3	100	2.67	3.03	(0.35)	4.40	Yes
ii32d3	101 (101)					
ii32e3	100	2.34	2.53	(0.11)	2.98	Yes
par16-2-c	100	5.05	145.04	(95.65)	310.45	Yes
par16-4-c	100	2.42	145.28	(87.76)	352.39	Yes
par32-2-c	101 (101)					
par32-4-c	1 (1)					
par8-2-c	100	0.12	0.26	(0.07)	0.44	Yes
par8-4-c	100	0.15	0.23	(0.04)	0.35	Yes
pret150_25	1 (1)					
pret150_75	1 (1)					
pret60_25	100	4.73	50.82	(29.81)	169.44	No
pret60_75	100	5.18	49.83	(28.43)	140.98	No
ssa0432-003	100	0.45	0.55	(0.06)	0.79	No
ssa2670-141	100	148.70	164.58	(8.16)	193.66	No
ssa7552-038	100	1.76	1.85	(0.05)	2.04	Yes
ssa7552-158	100	1.05	1.14	(0.05)	1.30	Yes
ssa7552-159	100	1.07	1.14	(0.05)	1.38	Yes
ssa7552-160	100	1.36	1.44	(0.04)	1.57	Yes

Benchmark (cont.) Name	Avg Time	Rel.Std Err (%)	Median Time
aim-100-2_0-no-1	43.45	5.13	47.08
aim-100-2_0-no-2	24.19	7.14	22.01
aim-100-2_0-no-3	14.48	5.08	15.46
aim-100-2_0-no-4	25.64	6.94	25.06
aim-100-2_0-yes1-1	0.73	1.26	0.73
aim-100-2_0-yes1-2	0.50	8.22	0.17
aim-100-2_0-yes1-3	0.85	1.30	0.83
aim-100-2_0-yes1-4	0.22	5.68	0.18
bf0432-007	22.85	0.69	22.67
bf2670-001	4.43	0.64	4.38
dubois20	36.51	3.13	33.97
dubois21	64.27	3.30	62.54
f400	10869.85		
f800			
f1600			
f3200			
f6400			
g125.17			
g125.18			
g250.15			
g250.29			
ii32b3	16.47	19.62	3.09
ii32c3	3.03	1.14	2.87
ii32d3			
ii32e3	2.53	0.44	2.52
par16-2-c	145.04	6.59	191.52
par16-4-c	145.28	6.04	133.18
par32-2-c			
par32-4-c			
par8-2-c	0.26	2.75	0.25
par8-4-c	0.23	1.78	0.23
pret150_25			
pret150_75			
pret60_25	50.82	5.87	46.11
pret60_75	49.83	5.71	47.41
ssa0432-003	0.55	1.04	0.55
ssa2670-141	164.58	0.50	162.22
ssa7552-038	1.85	0.27	1.85
ssa7552-158	1.14	0.41	1.14
ssa7552-159	1.14	0.43	1.14
ssa7552-160	1.44	0.29	1.45