Incomplete Thoughts about Incomplete Satisfiability Procedures

(extended abstract)

Allen Van Gelder and Yumi K. Tsuji
University of California, Santa Cruz
E-mail avg@cs.ucsc.edu, tsuji@cs.ucsc.edu.

August 1, 1993

Abstract

Recent successes with incomplete satisfiability procedures raise numerous issues about their uses and evaluation. Most such procedures search for "yes" answers, and give up if they do not find one. The need for "no" answers as well is addressed. It is shown that a recent approximation algorithm of Yannakakis can produce "no" answers only in somewhat trivial cases. The possibility of salvaging information from a "yes" program that is giving up, and using it to find a "no" answer is discussed.

Two incomplete procedures were studied experimentally. One (based on "easy" resolutions) produces "no" answers. The other (based on Johnson's maximum satisfiability approximation algorithm) produces "yes" answers. Both are deterministic, polynomial time algorithms, quadratic with careful implementation. Combined, they solved 83% of the formulas generated from a circuit-based application, ranging from 400 to over 10000 variables.

Key Words

Satisfiability, Boolean formula, propositional formula, approximation algorithms, resolution, 2-satisfiability, constant-width random formula.

1 Overview

The problem of Boolean, or propositional, satisfiability hardly needs another review. We assume the reader is generally familiar with it, as well as the *maximum satisfiability* problem, and give definitions only when needed for clarity.

Recently, incomplete procedures have enjoyed success on this NP-complete problem. They employ heuristics, and often randomness, to search for a satisfying assignment. They may be orders of magnitude faster than complete procedures in some cases. But what happens when the program does not succeed quickly? Subsequent sections discuss various issues related to this question. This is frankly a workshop paper. Issues are discussed rather than resolved, and we bounce around a lot.

We show that it is difficult for an approximate maximum satisfiability algorithm to give useful information on unsatisfiable cases. This is contrasted with the situation for many NP-hard optimization problems.

We describe an implementation of Johnson's approximate maximum satisfiability algorithm, but it is oriented toward finding complete assignments rather than simply satisfying a lot of clauses. We present a useful companion algorithm that solves some unsatisfiable instances. These algorithms have a certain comfort level lacking in many incomplete procedures. With many such procedures, you don't have a clear idea when to give up. Will the next iteration finally succeed? Should we make a fresh start altogether at a new starting point? Is the problem unsolvable? Our programs have no parameters to twiddle, and they stop when they are done, win, lose or draw.

A method to generate random formulas that are satisfiable, but not excessively easy is outlined. Possibilities to salvage information when a procedure gives up are explored. Certificates of unsatisfiability are discussed.

This paper does not mention CPU time because, for the algorithms examined, it is not an issue. They are deterministic, and have no parameters to twiddle. They are implemented suboptimally, but are fast enough that it is a secondary issue. For example, `MaxSat` described later processed Dimacs benchmark formulas (families "ii16" and "ii32") at 24,000 to 54,000 literals per CPU minute on an old Sun4/110 (pre-sparc). More important is the question, what are their capabilities?

2 CNF Formulas

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 $\tilde{x} \to y$ and $\tilde{y} \to x$. The notation $a \to\!\!\to b$ 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 and use m to represent the number of clauses in a formula.

3 Approximation Algorithms and "No" Answers

When an NP-complete problem is based on a natural optimization problem, approximation algorithms may yield both "yes" and "no" answers to the decision form of the problem. An interesting question is whether approximation algorithms for *maximum satisfiability* can help significantly with "no" instances of satisfiability.

To illustrate the first case, consider bin packing. Let K denote the target number of bins (all of unit size, for simplicity here). David Johnson described and analyzed a first fit decreasing procedure that is guaranteed to find a solution within 11/9 of the optimum [Joh74]. For a given set of packages, if this algorithm finds a solution using 100 bins, then a "no" answer is implied for values of $K \leq 81$, and some of these cases can be nontrivial in the sense that the sum of the package sizes can be less than K. Similar results are known for vertex cover, subset cover, etc.

Turning to maximum satisfiability, David Johnson also gave an algorithm that guarantees to find an assignment within a fraction of $(1-1/2^k)$ of the optimum, where k is the length of the shortest clause in the CNF formula [Joh74]. Recently, Yannakakis improved on this with a sophisticated algorithm that guarantees to achieve 3/4 of the optimum when $k = 1$, solving several max flow problems along the way [Yan93].

Example 3.1: Let the formula be $\{a\}$, $\{b\}$, $\{\tilde{a}, \tilde{b}\}$. Both Johnson's and Yannakakis' algorithms find assignments that satisfy 2 clauses. But Yannakakis promised to satisfy 3/4 of the optimum, so it carries the additional information that the optimum is strictly less than 3. Consequently, it implies a "no" answer to the satisfiability question. $\square$

Looking at Johnson's algorithm, we find that it guarantees to satisfy $(1 - 1/2^k)$ of *all* clauses, and *a fortiori* that fraction of the optimum. Thus it never helps (directly) to find a "no" answer. As just illustrated, Yannakakis algorithm can help with "no" answers at least in some cases. Unfortunately, as we shall now show, Yannakakis' algorithm cannot help "significantly" with "no" answers.

Unit resolution is one of the oldest restrictions on resolution that was known to be fast and incomplete. It requires one clause of any resolution operation to be a unit clause. Unit resolution can be implemented to run in linear time with the correct data structures, despite published "experimental evidence" that it requires quadratic time.

Theorem 3.1: If Yannakakis' algorithm satisfies less than 3/4 of the total number of clauses, then the formula can be refuted by unit resolution.

Proof: (Sketch) Inspection of Yannakakis' algorithm reveals that, if the formula is *not* refutable by unit resolution, then the max flow is zero in all of the intermediate flow problems of the algorithm. It then follows that H'' of his Lemma 10 is the original formula, while G' and H' are empty. ∎

The import is that unit resolution is a faster and more thorough way to get "no" answers to the satisfiability decision problem. It can also deliver better approximations than Yannakakis' algorithm *when* it does not detect a contradiction: simply make all the literals true that were forced to be true by unit resolution, leaving a remainder with no unit clauses, and then apply Johnson's algorithm. Yannakakis' algorithm wins when unit resolution detects a contradiction, by finding a good choice among the contradictory clauses.

4 Incomplete Procedures as Filters

We became involved with incomplete procedures while composing a set of satisfiability problems for the Dimacs Challenge, which formulas had their origins in circuit fault detection applications [Lar92, LT92]. The application program generated too many formulas. How could we systematically screen them to retain the "interesting" (i.e., somewhat challenging) ones? A criterion of the form "algorithm X took more than Y seconds (or guesses) on this problem" left something to be desired.

This application generates many Krom clauses (1- and 2-clauses) in its formulas. Our initial filter was, "do the Krom clauses alone demonstrate unsatisfiability?" This can be checked in linear time [APT79]. But we wanted to screen out "easily satisfiable" formulas as well.

4.1 Easily Satisfiable

We decided to implement a version of Johnson's algorithm [Joh74]. This algorithm weights each clause according to its probability of being falsified by a random assignment (with equal probabilities) to the remaining unbound variables. A 3-clause begins with weight 1/8; if a literal is made false, the clause weight doubles to 1/4; if a literal is made true, the weight becomes zero. (Yannakakis exploited the possibility of basing weights on unequal probabilities of true and false, which varied from variable to variable.) A successful assignment reduces the formula weight to zero.

Because the formulas contained unit clauses, it was not evident that Johnson's algorithm would accomplish much: it only promised to satisfy 1/2 the clauses, and we were only interested in the cases where it satisfied *all* of them. However, we were interested in discarding very easy formulas, not in achieving the highest possible success rate. This approach had the advantage of having no searching, no randomness, and known polynomial running time (probably linear with the best implementation, but not with *our* implementation).

The published algorithm left the choice of variable at each step open: *any* choice would achieve the bounds. We shall refer to our implementation as `MaxSat`, to distinguish it from Johnson's generic algorithm. We tried to make choices that reflected our goal of achieving 100%, that is, driving the formula weight to zero. Thus we refused (as long as possible) to choose a variable with the property that one assignment increased the weight and the other assignment made a unit clause false (or created a unit clause contradicting an existing unit clause). We always preferred a pure literal; that was a "free ride". Next in priority came an assignment that both satisfied a unit clause and reduced overall weight. Otherwise we chose the variable that reduced the formula weight the most.

`MaxSat` was not very successful on the initial formulas. It seems that 1/2 was just too far from 1. However, we were separately testing for "easily unsatisfiable" by doing certain "easy" resolutions (described below) that reduced the formula to one without unit clauses, and fewer variables. So we tried `MaxSat` on the reduced formulas instead. Now the algorithm guaranteed to satisfy 3/4 of the clauses. In fact, it often satisfied them all. (see Table 1).

Circuit	Number of Formulas	Known Sat	Number Solved	Fraction Solved
0432.bf	143	1	0	.00
2670.bf	581	28	12	.43
2670.ssa	141	59	59	1.00
7552.ssa	163	68	28	.41
total		156	99	.63

Table 1: `MaxSat` was able satisfy 100% of the clauses in 99 out of 156 (known) satisfiable circuit-based problems.

Circuit	Number of Formulas	Known Unsat	Number Solved	Fraction Solved
0432.bf	143	134	122	.91
0432.ssa	16	16	9	.56
1355.bf	770	638	621	.97
1355.ssa	8	8	8	1.00
2670.bf	581	525	516	.98
2670.ssa	141	71	70	.99
6288.ssa	51	51	48	.94
7552.ssa	163	55	55	1.00
total		1498	1449	.97

Table 2: `2clNG` refuted 1449 out of 1498 (known) unsatisfiable circuit-based problems. The status of several hundred more was not known at the time of the screening run.

4.2 Easily unsatisfiable

To screen out "easily unsatisfiable" formulas, we implemented `2clNG` (2-closure with *N*o *G*uessing), which performed 2-closure, pure literal elimination, subsumption, and subsumption resolution. These are all performable in quadratic time with the right data structures, but our implementation has a cubic time component. These operations reduce the formula size while maintaining its satisfiability status. If the formula did not reduce to a trivial one, the final reduced version was written out for `MaxSat`, as mentioned. Results are summarized in Table 2.

Two-closure draws all available inferences from Krom clauses, and is closely related to the well-known 2SAT linear time algorithm [APT79]. E.g., if $a \twoheadrightarrow b$ and $b \twoheadrightarrow a$ in the implication graph, all occurrences of b are replaced by a in the formula, and attendant reductions are performed. If $\tilde{a} \twoheadrightarrow a$, then a is concluded to be true, etc.

Subsumption resolution applies when the resolvent is a subclause of one of the operands; unit resolution is a special case. Examples are: $(\{\tilde{a}\}, \{a, b, c\}) \rightarrow \{b, c\}$, $(\{\tilde{a}, b\}, \{a, b, c\}) \rightarrow \{b, c\}$, and $(\{\tilde{a}, b, c\}, \{a, b, c\}) \rightarrow$

Clauses	Variables	Satisfiable Formulas	Solved by `MaxSat`	Percent Solved	% Clauses Satisfied Ave.	Min	Max
300	100	100	95	95%	99.98	99.67	100.00
400	100	93	19	20%	99.53	98.50	100.00
427	100	61	4	7%	99.29	97.89	100.00
450	100	17	1	6%	98.85	97.33	100.00
550	100	0	0		97.55	94.00	98.91
800	200	98	8	8%	97.56	97.38	100.00

Table 3: `MaxSat` performance on random 3-CNF formulas (constant width model). Each sample contained 100 independent formulas.

Family	Clauses Range (Ave.)	Satisfiable Formulas	Solved by `MaxSat`	Percent Solved	% Clauses Satisfied Ave.	Min	Max
ii8	186–8214 (3899)	14	2	14%	99.88	96.77	100.00
ii16	7825–24792 (16479)	10	0	0%	99.56	98.10	99.85
ii32	1186–20862 (6378)	17	2	12%	98.29	96.90	100.00

Table 4: `MaxSat` performance on "inductive inference" Dimacs Benchmark formulas, all of which are satisfiable. The small number of samples and wide range of sizes prevents any firm conclusions.

$\{b, c\}$. Because the resolvent subsumes one of the operands, the formula shrinks. Subsumption deletes a clause when a proper subclause of it exists in the formula.

In summary, the circuit test pattern generator `Nemesis` generated 1873 formulas, of which 1548 were solved by the above deterministic polynomial time methods. The methods detected both satisfiable and unsatisfiable formulas. Those that passed through this screening can be presumed to have a certain objective level of difficulty, and were contributed to the Dimacs Challenge. They ranged from 427 to 10410 variables, and from 1004 to 34238 clauses.

5 More on MaxSat

We were intrigued by the success of `MaxSat` on the circuit-based formulas. We planned to do a number of experiments, but the student who was supposed to work on this disappeared. However, we can report limited results, both positive and negative, at this time.

On the positive side, the algorithm consistently satisfies well above its guaranteed fraction. On formulas only having 3-clauses, it promises 87.5%, but Table 3 shows that it scores in the high 90's. Previous experiments have shown that the clause/variable ratio of 4.27 was hardest for several algorithms [MSL92, LT92]. The ratio of 3.00 was found to be easy, and `MaxSat` was very successful at that ratio, *as a satisfiability tester*. For the more difficult ratios, its performance fell off. The ratio 5.50 was again easy for `DPLL` variants, but almost no satisfiable formulas occur in this range.

Clauses	Variables	% Solved by MaxSat	% Clauses Satisfied Ave.	Min	Max
400	100	19%	99.51	98.25	100.00
800	200	9%	99.56	98.88	100.00
1600	400	1%	99.55	98.88	100.00
2400	600	0%	99.57	99.00	99.92
3200	800	0%	99.55	99.16	99.97
4000	1600	0%	99.56	99.18	99.80
450	100	3%	98.89	97.33	100.00
550	100	0%	97.38	95.45	99.27
750	100	0%	94.65	90.40	96.53

Table 5: `MaxSat` performance on "forced satisfiable" random 3-CNF formulas, generated by the method described in Section 6. Each sample contained 100 independent formulas.

Additional runs are reported in Table 4. These formulas have a large majority of 2-clauses. The guarantee is 75%, but again, performance in the high 90's was observed. However, totally satisfying assignments were rarely achieved. Also Table 5 shows results for larger random formulas: the percent of clauses satisfied on average looks stable, but the percent of formulas solved completely drops rapidly.

The negatives are less quantitative. We hoped that `MaxSat` preserved satisfiability for many assignments, so that an initially satisfiable formula remained so through most of the processing. Perhaps only the last few assignments created unsatisfiability. If this were true, we could stop `MaxSat` part way, and spit out a much smaller formula that was still satisfiable, then run a complete method on this smaller formula. Alas, in our limited attempts along these lines, the smaller formula was found to be unsatisfiable even though the original was satisfiable.

There is room for more work along these lines; we may have been too greedy, trying to reduce the formula by 90% within `MaxSat` before switching to the complete method. For 400 to 450 clauses, contradictory unit clauses tend to show up after about 365 clauses have been satisfied; the formula may have been unsatisfiable even earlier.

A possible application for `MaxSat` is to provide a starting point for a local search algorithm. Experiments are needed to determine whether it is better to start with 99% of the clauses satisfied deterministically, or with 88% satisfied randomly.

6 Generating Satisfiable Formulas

A known procedure to generate randomly a CNF formula and know it is satisfiable is to make up a "hidden assignment" and discard any generated clause that is inconsistent with it. This was found to produce excessively easy formulas [MSL92]. This section explains why and offers an alternative. The implementation in C was contributed to the Dimacs Challenge as `cnfgen`.

Suppose for simplicity that the hidden assignment is "all true". Thus any clause without a positive literal is rejected. The 7 acceptable patterns for 3-clauses have an average of 12/21 positive literals, as $(-,-,-)$ is rejected. Thus on average a variable will occur positively in 4/7 of its occurrences, biasing it toward the hidden assignment. (The same reasoning works for any hidden assignment, not just "all true".)

The solution is to reject any generated clause that has an *even number* of literals that match the hidden assignment. Also, clauses are required to have at least 2 literals. In particular, zero is even, so inconsistent clauses are rejected. But there is no bias toward the hidden assignment among the accepted clauses. To see why, again assume the hidden assignment is "all true". The number of positive literals among acceptable patterns for a clause of length $k \geq 2$ is

$$1\binom{k}{1} + 3\binom{k}{3} + 5\binom{k}{5} + \ldots$$

But this is

$$k\left[\binom{k-1}{0} + \binom{k-1}{2} + \binom{k-1}{4} + \ldots\right] = k\,2^{k-2}$$

Since the total number of "positions" in the acceptable patterns is $k\,2^{k-1}$, exactly half are positive. (Note that the argument breaks down for $k = 1$.)

Indications are that this family of formulas is no easier than the satisfiable subset of the usual constant-clause-width family. For example, compare Table 3 and 5 for 400 clauses, 100 variables and for 800 clauses, 200 variables. `MaxSat` was able to solve about the same percentages of the (satisfiable) formulas generated by each method.

High clause-to-variable ratios may not produce easier problems for this generation method. The last three lines of Table 5 show the percent satisfied declining as the number of clauses increases. It will be interesting to see whether this behavior carries over to iterative local search methods.

7 Salvaging Operations

Suppose an approximate maximum satisfiability procedure, such as one of the many "hill-climbing" methods, runs for a long time without satisfying all clauses on a formula whose satisfiability status is unknown. There is a presumption that the formula is unsatisfiable, but can anything useful be salvaged to help prove this?

It seems to occur frequently in practice that all clauses but one are satisfied by the approximation procedure. (We observed this frequently with `MaxSat`.) In this case, the lone falsified clause must belong to a minimal unsatisfiable subset, if the formula is indeed unsatisfiable.

There are several restrictions of resolution proof procedures that are guaranteed to succeed if the initial or "top" clause is part of a minimal unsatisfiable subset. Without having such a "top" clause to begin a proof, one could potentially be forced to try each clause of the formula as a top clause before discovering a refutation or concluding that one did not exist. Therefore, the lone falsified clause from a procedure that is giving up may be very valuable salvage.

It is quite possible that the minimal unsatisfiable subset is a very small portion of the whole formula. A proof of unsatisfiability using only these clauses exists. However, there is no way to guarantee that the proof search remains within this unknown subset. "Linear" proof procedures always use the most recently derived clause as one operand of the next resolution; thus a proof is like a path through a graph in which clauses are nodes and complementary literals induce edges. A breadth-first search or best-first search can be conducted. This topic is treated extensively by Loveland and co-authors [Lov69, FLSY74, Lov78], but with emphasis on the first order case. Experiments are needed to evaluate this approach on propositional formulas.

8 Certificates of Unsatisfiability

We have a new, much shorter, proof of the four-color theorem. We negated the theorem statement and asked our program if that formula was satisfiable. The output was "No". QED.

Readers cannot be accused of undue skepticism if they ask for a few more details. This is a serious problem with many "satisfiability testers" such as that of Davis, Putnam, Logemann, and Loveland [DLL62], which we shall call `DPLL`. They can produce an easily verified certificate of their "yes" answers, but for the "no" answers, it's "trust me". If costly design decisions or safety considerations depend on the accuracy of the program, "trust me" will not be good enough.

For a CNF formula, a resolution refutation is probably the best known certificate. That is, each step can be validated by a simple program. (A *refutation* is a derivation of $\emptyset$.) The certificate is exponentially long in the worst case, but nothing better is available in practice.

For use as a certificate, a resolution derivation can be formalized as a sequence of "tuples". Each tuple has an `Id`, a `Clause`, and two `Parents`. The `Id` and `Parents` are integers. The first m tuples are the clauses of the original formula, with `Ids` 1 through m, and 0 for `Parents`. Subsequent tuples have successively higher `Ids`, $m+1$, $m+2$, ..., and their `Parents` are integers smaller than their `Ids`. (In a *linear* derivation, the first `Parent` of tuple with `Id` $= k$ must be $(k-1)$, so can be suppressed in the presentation.) The semantics is that the resolution of the two parents produces the current `Clause`.

Let us sketch how to extract such a certificate from `DPLL`, where only the splitting rule is used. Extensions to include the pure literal and unit clause rules are straightforward. Recall that, when `DPLL` decides a formula F is unsatisfiable, it has first found $F[x=1]$ to be unsatisfiable, then found $F[\tilde{x}=1]$ to be unsatisfiable.

As the base case, assume each branch takes one step, for a total of 2 steps. Then F contains $\{x\}$ and $\{\tilde{x}\}$. So the proof is $(1, \{x\}, 0, 0)$, $(2, \{\tilde{x}\}, 0, 0)$, $(3, \emptyset, 1, 2)$.

For searches of more than two steps, assume that both branches have produced refutations P_1 and P_0 of formulas $F[x=1]$ and $F[\tilde{x}=1]$, respectively. First transform the refutation P_1 based on $F[x=1]$ into a derivation of $\{\tilde{x}\}$ from F. Do so by inserting $\tilde{x}$ into each clause of P_1 that is or depends transitively on a clause of $F[x=1]$ that was obtained from F by removing literal $\tilde{x}$. For example, $\{\tilde{x}, y, z\}$ is in F, and produces $\{y, z\}$ in $F[x=1]$. If clause A in P_1 depends on $\{y, z\}$, transform it to $\{\tilde{x}\} \cup A$. In particular, $\emptyset$ is transformed into $\{\tilde{x}\}$. Similarly, transform P_0, based on $F[\tilde{x}=1]$ into a derivation of $\{x\}$.

Concatenate the two transformed proofs with renumbering to make the `Ids` unique. Append to that the empty clause, with the `Ids` of $\{\tilde{x}\}$ and $\{x\}$ as `Parents`.

The idea sketched here is based on the proof-of-completeness technique of Anderson and Bledsoe, and can be extended to provide a *linear* resolution refutation [AB70]. It is "folklore" that `DPLL`'s execution can somehow be associated with a resolution proof, but we have not seen it spelled out. We do so here to assist those implementers who may wish to provide certificates of their programs' greatest successes.

9 Conclusion and Future Work

Clearly there are few conclusions and much remaining work.

Acknowledgements

We are grateful to Tracy Larrabee for making her `Nemesis` software available, and her help in plumbing its depths for the purpose of printing out the formulas it generated internally. 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

[AB70] R. Anderson and W. W. Bledsoe. A linear format for resolution with merging and a new technique for establishing completeness. *Journal of the Association for Computing Machinery*, 17:525–534, 1970.

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

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

[FLSY74] S. Fleisig, D. W. Loveland, A. K. Smiley, and D. L. Yarmush. An implementation of the model elimination proof procedure. *JACM*, 21(1):124–139, 1974.

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

[Lar92] T. Larrabee. Test pattern generation using boolean satisfiability. *IEEE Transactions on Computer-Aided Design*, pages 6–22, January 1992.

[Lov69] D. W. Loveland. A simplified format for the model elimination theorem-proving procedure. *JACM*, 16(3):349–363, 1969.

[Lov78] D. W. Loveland. *Automated Theorem Proving: A Logical Basis.* North-Holland, Amsterdam, 1978.

[LT92] T. Larrabee and Y. Tsuji. Evidence for a satisfiability threshold for random 3CNF formulas. Technical Report UCSC–CRL–92–42, UC Santa Cruz, October 1992. (Presented at AAAI 1993 Spring Symposium).

[MSL92] D. Mitchell, B. Selman, and H. Levesque. Hard and easy distributions of SAT problems. In *Proceedings of AAAI*, 1992.

[Yan93] M. Yannakakis. On the approximation of maximum satisfiability. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 1–9, 1993.