On Computing k-CNF Formula Properties

Ryan Williams*

Computer Science Department
Carnegie Mellon University

Abstract. The latest generation of SAT solvers (e.g. [9, 5]) generally have three key features: randomization of variable selection, backtracking search, and some form of clause learning. We present a simple algorithm with these three features and prove that for instances with constant Δ (where Δ is the clause-to-variable ratio) the algorithm indeed has good worst-case performance, not only for computing SAT/UNSAT but more general properties as well, such as maximum satisfiability and counting the number of satisfying assignments. In general, the algorithm can determine any property that is computable via *self-reductions* on the formula.

One corollary of our findings is that for all fixed Δ and $k \geq 3$, *Max-k-SAT* is solvable in $O(c^n)$ expected time for some $c < 2$, partially resolving a long-standing open problem in improved exponential time algorithms. For example, when $\Delta = 4.2$ and $k = 3$, *Max-k-SAT* is solvable in $O(1.8932^n)$ expected time. We also improve the known time bounds for exact solution of $\#2SAT$ and $\#3SAT$, and the bounds for k-SAT when $k \geq 5$.

1 Introduction/Background

Exponential time algorithms for SAT with improved performance have been theoretically studied for over 20 years. Beginning in 1979, Monien and Speckenmeyer [7] gave a $\tilde{O}(1.618^n)$ algorithm for 3-SAT [7]. Reviewing the literature, it appears that studies in improved worst-case time bounds for SAT were mostly dormant for many years, until a resurgence in the late 1990s (e.g. [10, 4, 1]). The first improvements used DPLL-style variants, where variables were repeatedly chosen in some way, and the algorithm recursed on both possible values for the variables. The improved time bounds came about due to clever case analysis about the number of variables or the number of clauses removed from consideration in each of these recursive branches. In 1999, Schöning [13] gave a $\tilde{O}(1.3333^n)$ algorithm for 3-SAT that is essentially the WalkSAT algorithm [14]; this was followed by a $\tilde{O}(1.3303^n)$ improvement a couple of years later [6].

The work on *Max-k-SAT* has been less successful than that for k-SAT: it has been open whether or not *Max-k-SAT* can be solved in c^n steps for $c < 2$. In this work, we will resolve the question in the affirmative, when the clause density is constant. Further,

* Supported in part by an NSF Graduate Research Fellowship and the NSF ALADDIN Center (`www.aladdin.cs.cmu.edu`). Email address: `ryanw@cs.cmu.edu`.

there has been strong recent progress in counting satisfying assignments [2]: #2SAT and #3SAT are solvable in 1.3247^n and 1.6894^n time, respectively. Our approach supplants these bounds, having 1.2923^n and 1.4461^n expected time. Also, for large k, our algorithm outperforms any other in the SAT case. For example, for $k = 20$, Schöning's random walk algorithm runs in $\tilde{O}(1.9^n)$ whereas ours runs in $\tilde{O}(1.8054^n)$. This bound improvement occurs for all $k \geq 5$. It is important to stress that our randomized method is of the Las Vegas variety and thus *complete*, unlike the previous randomized algorithms for these problems [10, 13, 6] which are Monte Carlo (with one-sided error).

One disadvantage of some improved exponential time algorithms is their limited applicability: often, an improved algorithm for one variant of SAT yields little or no insight about other SAT variants. Here, our strategy can in general be applied to determine most interesting hard-to-compute properties of an arbitrary k-CNF formula that have been considered, under conditions that we will formally specify. We deliberately make our approach as abstract as possible, so that perhaps its ideas may be useful in other areas as well.

2 Notation

Let $T(n)$ be super-polynomial and $p(n)$ be a polynomial. We will express runtime bounds of the form $T(n) \cdot p(n)$ as $\tilde{O}(T(n))$, the tilde meaning that we are suppressing polynomial factors.

Boolean variables will be denoted as $x_i \in \{true, false\}$. Literals (negated or non-negated variables) will be denoted by $l_i \in \{x_i, \overline{x_i}\}$. F will denote a Boolean formula in conjunctive normal form over variables $x_1, \ldots, x_n$. We represent F as a family of subsets over $\{x_1, \overline{x_1}, \ldots, x_n, \overline{x_n}\}$. The sets of F are called clauses. We will implicitly assume that F has no trivial clauses containing both x_i and $\overline{x_i}$ for some i. The number of clauses in F is denoted by $m(F)$, the number of variables is $n(F)$, and the density of F is $\Delta(F) = m(F)/n(F)$. Typically we will just call these n, m, and Δ when the formula F under consideration is clear. Two special kinds of formulas are $\top$ and $\bot$. $\top$ is the empty formula $\varnothing$, or trivially true formula. $\bot := \{\varnothing\}$, the formula with a single, empty constraint, a trivially false formula.

The formula $F[x_i = v]$ is the formula that results when value $v \in \{true, false\}$ is substituted for variable x_i in F.

3 Self-Reducible Properties

Let us formalize the sort of formula properties that are computable by the algorithm we will describe. Intuitively, they are those properties that may be described due to *self-reducibility*. For example, satisfiability of a formula F is a self-reducible property, since satisfiability of F may be deduced by testing satisfiability on the smaller formulas $F[x = true]$ and $F[x = false]$.

Definition 1. *Let f be a function from k-CNF formulas and natural numbers to a set V. f computes a* **feasibly self-reducible property** *iff:*

(1) $\forall i$, $f(\top, i)$ and $f(\bot, i)$ are polytime computable.

(2) There exists a polytime computable function g such that $f(F, n) = g(x, f(F[x = true], n-1), f(F[x = false], n-1))$, for all formulas F and variables x.

In English, this means we can easily compute f on F using g, provided we are given f's values when some variable is true, and when it is false.

To motivate this definition, we demonstrate that interesting (e.g. NP and $\#P$ complete) properties normally determined of SAT instances are feasibly self-reducible, provided we begin our computation of $f(F, n)$ with $n = n(F)$. The following table shows some of the properties that fall under our framework, given g and f's definition on the trivially true and trivially false formula. We also provide our algorithm's expected time bounds when $k = 2$ and $k = 3$ for these various properties. For the first two rows of the table, the v_i are truth values; for the second two rows, they are natural numbers.

f	$g(x, v_1, v_2)$	$f(\top, i)$	$f(\bot, i)$	$k = 2$	$k = 3$
SAT	$v_1 \vee v_2$	true	false	trivial	1.4461^n
UNSAT	$v_1 \wedge v_2$	false	true	trivial	1.4461^n
Max-SAT	$\max\{n(x_1, F) + v_1, n(\overline{x_1}, F) + v_2\}$	0	0	c^n $(c < 2)$ if $\Delta = O(1)$	
#SAT	$v_1 + v_2$	2^i	0	1.2923^n	1.4461^n

(We define $n(l, F)$ to be the number of occurrences of literal l in F.) $\#SAT(F)$ is the number of satisfying assignments. *Max-SAT*(F) is the maximum number of clauses satisfied by any assignment. (A simple modification of the algorithm will be able to extract an assignment satisfying the maximum number, with no increase in asymptotic runtime.) We remark that *Max-SAT* is the only function above that uses the variable x in the specification for g.

4 Algorithm

We now present a way to compute any feasibly self-reducible f on k-CNF formulas with density Δ. The methodology is quite similar in nature to previous improved exponential time algorithms using dynamic programming [12, 15]. The three major differences here are the use of randomness, the manner in which dynamic programming, and the tighter analysis that results from analyzing k-CNF formulas.

Roughly speaking, the algorithm chooses a random ordering on the variables, then does a standard depth-first branching on the first δn variables in this ordering, for some calculated $\delta \in (0, 1)$. After depth δn has been reached, the algorithm continues branching, but saves the computed f-values of all formulas considered after this point. The key is that for suitable δ (depending on k and Δ), the space usage necessary is small, and the expected runtime is greatly reduced asymptotically.

4.1 Preliminary initialization

Before the main portion of the algorithm is executed, a few preliminary steps are taken to set up the relevant data structures.

0. Let $\Delta = m/n$, and δ be the smallest root of the polynomial $\Delta\delta^k + \delta - \Delta$ over the interval $(0, 1)$. (Existence of such a root will be proven later.) Since k is constant, one can numerically compute this root to a suitable precision in polynomial time.

1. Choose a random permutation $\sigma : \mathbb{N} \to \mathbb{N}$. Let $F_{cover} \subseteq F$ be the subset of clauses that have each of their k variables in $\{x_{\sigma(1)}, \ldots, x_{\sigma(\delta n)}\}$.

2. Define $\leq_c$ to be a lexicographic (total) ordering on the clauses of F_{cover}, where the ordering is obtained from the variable indices. For instance, given $i_1 < j_1 < k_1$ and $i_2 < j_2 < k_2$, $\{x_{i_1}, x_{j_1}, x_{k_1}\} \leq_c \{x_{i_2}, x_{j_2}, x_{k_2}\}$ iff either $i_1 < i_2$ or ($i_1 = i_2$ and $j_1 < j_2$) or ($i_1 = i_2$ and $j_1 = j_2$ and $k_1 \leq k_2$). Define c_i to be the ith clause w.r.t. the ordering $\leq_c$.

3. Let V be the co-domain of f. (Typically, V is either $\{true, false\}$ or $\mathbb{N}$.) Initialize the set *Learned* $\subseteq \{0,1\}^{m(F_{cover})} \times \{1, \ldots, n\} \times V$ of learned f-values as empty.

4.2 Search

The search portion of the algorithm recurses on a formula F_r and integer i, which are initially F and n, respectively.

Compute-f(F_r, i):

1. [If $i = 0$ then either $F_r = \perp$ or $F_r = \top$; take step 2.]

2. If $F_r = \perp$ or $F_r = \top$, return $f(\top, i)$ or $f(\perp, i)$, respectively.

3. (*Branching phase*) If $i \geq n - \delta n$, then return:
$g(x_{\sigma(n-i+1)}$,Compute-f$(F_r[x_{\sigma(n-i+1)} = true], i-1)$,Compute-f$(F_r[x_{\sigma(n-i+1)} = false], i-1))$.

4. (*Learned values phase*) Else, let $F_r^k \subseteq F$ be the set of original k-clauses in F that correspond to the remaining (possibly $< k$-)clauses of F_r. It follows that $F_r^k \subseteq F_{cover}$. Represent F_r as a pair $(b(F_r), i)$, where $b(F_r)$ is a vector of $m(F_{cover}) = |F_{cover}|$ bits: $b(F_r)[j] := 1$ iff c_j (the jth clause in $\leq_c$) has not yet been satisfied in F_r.

5. If $(b(F_r), i, v) \in \mathit{Learned}$, then return v. Let b_t and b_f be the bit vector representations of $F[x_{\sigma(n-i+1)} = true]$ and $F[x_{\sigma(n-i+1)} = false]$, respectively.

5a. Set $v_t :=$ Compute-f$(f(F[x_{\sigma(n-i+1)} = true]), i+1)$ and
$v_f :=$ Compute-f$(f(F[x_{\sigma(n-i+1)} = false]), i+1)$.

5b. Update *Learned* := *Learned* $\cup \{(b_t, i, v_t), (b_f, i, v_f)\}$. Return $g(x_{\sigma(i+1)}, v_t, v_f)$.

6. Otherwise, branch as before; that is, return
$g(x_{\sigma(n-i+1)}$,Compute-f$(F_r[x_{\sigma(n-i+1)} = true], i-1)$,Compute-f$(F_r[x_{\sigma(n-i+1)} = false], i-1))$.

4.3 Analysis

Sketch of correctness Here, we assume the choice of δ is suitable and defer its justification until later. We consider each step in the above algorithm one by one.

• Steps 1 and 2, the base cases, are clear. Step 3 is obvious assuming Compute-f$(F_r[x_{\sigma(i)} = true], i+1)$ and Compute-f$(F_r[x_{\sigma(i)} = false], i+1)$ return correct answers.

• i always equals the number of variables that have not been set by the algorithm; the proof is a simple induction. Hence when $i > \delta n$, then the first δn variables have been set in F_r, so letting $F_r^k \subseteq F$ be the set of original k-clauses in F corresponding to the (possibly $< k$) clauses of F_r, $F_r^k \subseteq F_{cover}$ follows from the definition of F_{cover}.

• In Steps 4 and 5, notice the representation $(b(F_r), i)$ tells us two things: which clauses of F_{cover} have not (yet) been satisfied, and which variables have already been set (those $x_{\sigma(j)}$ where $j < i$). Thus, if literals of these variables appear in the clauses specified by $b(F_r)$, we may infer that these literals are *false*. Therefore we can reconstruct F_r given $(b(F_r), i)$, so the map $F_r \mapsto (b(F_r), i)$ is 1-1. Hence it is semantically correct to return v for $f(F_r)$ when we find $(b(F_r), i, v) \in$ *Learned* in Step 5.

So we store every f-value computed in *Learned* and search for it before recomputing. The *Learned* set used in step 5 can be implemented using a binary search tree, where the keys are pairs containing (a) the $|F_{cover}|$ bit vector representations of the F_rs and (b) the variable index i. The relevant operations (insert and find) take only polynomial time.

Runtime analysis We claim the algorithm devotes $\tilde{O}(2^{\delta n})$ time for the branching phase (when $i \leq \delta n$) and a separate count of $\tilde{O}(2^{E[m(F_{cover})]})$ expected time for the learned values phase, where $E[m(F_{cover})]$ is the expected number of clauses in F_{cover} over the choice of random σ. (Hence in total, the expected runtime is $\tilde{O}(2^{E[m(F_{cover})]} + 2^{\delta n})$, and the optimal choice of δ to minimize the this will make $E[m(F_{cover}] = \delta n$.)

To simplify the analysis, we consider an "unnatural" procedure, for which our algorithm has runtime no worse than it. The procedure will perform the phases of the algorithm described above, but in the opposite order. First, it (a) determines all of the possible f-values in *Learned* recursively, saving each discovered value as it goes along. Then it (b) runs the branching phase until depth δn, in which case it simply refers to the stored values in *Learned*.

It is clear that if the runtime of (a) is bounded by T, then the runtime of this procedure is $\tilde{O}(2^{\delta n} + T)$. So it suffices for us to prove that (b) takes $\tilde{O}(2^{E[m(F_{cover})]})$ expected time. Each $(b(F_r), i)$ pair's f-value in *Learned* is computed at most once, and is determined in polynomial time using g and assuming the f-values for smaller F_r are given. (We defer the cost of computing the f-values for smaller F_r to those smaller formulas). Moreover, the base cases $f(\top, i)$ and $f(\bot, i)$ are polytime computable by self-reducibility.

Thus the total time used by the learned formula phase will be at most

$poly(n)\cdot$ [number of possible $(b(F_r), i)$ pairs] $= \tilde{O}(2^{E[m(F_{cover})]})$,

since the total number of pairs possible in *Learned* is at most $n \cdot 2^{m(F_{cover})}$.

Let us specify the procedure more formally. [We can omit this part from the final version if the above sketch is convincing.] Start with $(\perp, i)$ and $(\top, i)$ for every i, and put $(\perp, i, f(\perp, i))$ and $(\top, i, f(\top, i))$ in *Learned*.

0. Initialize $i := n - 1$.
1. Repeat steps 2-5 until $i = \delta$:
2. Set $\mathcal{F} := \{F_r \cup \{c \in F | x_{\sigma(i)} \in c \vee \overline{x_{\sigma(i)}} \in c\} \mid \exists v.(b(F_r), i+1, v) \in Learned\}$.
3. For all $F_r \in \mathcal{F}$,
4. Find v_1 and v_2 such that $(b(F_r[x_{\sigma(i)} = true]), i + 1, v_t)$ and $(b(F_r[x_{\sigma(i)} = false]), i + 1, v_t)$ in *Learned*, using a search tree.
5. Put $(b(F_r), i, g(v_t, v_f))$ in *Learned*, and set $i := i - 1$.

Notice we are always placing a value for a new pair in *Learned*. Hence we place at most $n2^{m(F_{cover})}$ values in *Learned*. Each iteration of the for-loop for a fixed F_r takes polynomial time. The number of possible F_r in F is at most $2^{m(F_{cover})}$ (though it will be much less in most cases). There are at most $n - \delta n$ repetitions of the repeat loop, hence this procedure takes $\tilde{O}(2^{E[m(F_{cover})]})$ expected time.

Note that while our procedure takes exponential space, as in [12, 15], a tradeoff may be exhibited between time and space usage, by varying δ. In other words, for larger values of δ, less space is required at the cost of a longer runtime.

Theorem 1. *For every k and Δ, there exists a constant $c < 2$ such that any feasibly self-reducible f on k-CNF Boolean formulas with density Δ is computable in $O(c^n)$ expected time.*

Proof. It suffices to show that the optimal choice of δ is always less than 1. Let c_i be a k-CNF clause. For a randomly chosen σ, the probability that a particular variable v is among the first δn variables is δ. Hence the probability that every variable in c_i is among the first δn variables designated by σ is at least $\delta^k(1 - o(1))$. More precisely, the probability is

$$\prod_{i=0}^{k-1} \frac{\delta n - i}{n - i} \geq \delta^k \prod_{i=0}^{k-1} \left(1 - \frac{i}{n}\right) \geq \delta^k \left(1 - \frac{d}{n}\right),$$

for some constant $d > 0$. Thus the probability that $c_i \in F_{cover}$ is at most $1 - \delta^k(1 - o(1))$. For each clause $c_i \in F$, define an indicator variable X_i that is 1 iff $c_i \in F_{cover}$. Then the expected number of clauses in F_{cover} is

$$E[F_{cover}] = \sum_{i=1}^{m} E[X_i] = m \cdot [1 - \delta^k(1 - d/n)],$$

by linearity of expectation. Hence the expected time for the learned value phase is (modulo polynomial factors)

$$2^{[1-\delta^k](1-d/n)\Delta n} = 2^{[1-\delta^k]\Delta n - d\cdot\Delta\cdot[1-\delta^k]} \in \tilde{O}(2^{[1-\delta^k]\Delta n}),$$

and the optimal choice of δ satisfies the equation

$$\delta = (1-\delta^k)\Delta \implies \Delta\delta^k + \delta - \Delta = 0.$$

Notice that the variance in $m(F_{cover})$ will be small in general (more precisely, susceptible to Chernoff bounds), thus our expectation is not a mathematical misnomer; we will not analyze it in detail here.

We now show that for $k > 0$ and $\Delta > 0$, the polynomial $p(x) = \Delta x^k + x - \Delta$ has at least one root $x_0 \in (0, 1)$; the theorem will follow.

First, $p(x)$ has at least one real root r. Note $p(1) = 1$ for all k and Δ, so $r \neq 1$. If $r > 1$, then $\Delta r^k > \Delta$, a contradiction. Hence $r < 1$. If k is even, then $p(x)$ has at least one positive root r, so $r \in (0, 1)$. On the other hand, if k is odd and $r < 0$, then all three terms in $p(r)$ are negative; hence $r \in (0, 1)$. □

We have empirically observed that as either Δ or k increase, the relevant root of $p(x)$ approaches 1.

4.4 Max-*k*-SAT solution

Ever since Monien and Speckenmeyer [8] showed in 1980 that there exists an algorithm for *Max-3-SAT* running in $\tilde{O}(2^{m/3})$, it has been a well-studied open problem as to whether *Max-k-SAT* could actually be solved in $O(c^n)$ time for $c < 2$. All previous proposals towards answering this question have given algorithms of the form $O(c^m)$, with c decreasing slowly over time (e.g. [8, 1, 4]).

A corollary of the above result is that the answer is *yes*, when the clause density Δ is constant. While this is probably the more relevant situation for applications, it remains open whether *Max-k-SAT* can be solved when Δ is an unbounded function of n.

Corollary 1. *For every k and Δ, there exists a constant $c < 2$ such that Max-k-SAT on formulas of density Δ is solvable in $\tilde{O}(c^n)$ expected time.*

4.5 Improvements on Counting and SAT for high *k*

If the property we seek is some function on the satisfying assignments of F, then a better runtime bound can be achieved; we will outline our modified approach here. For instance, if we wish to count the number of satisfying assignments or determine satisfiability, then we can use the unit clause rule in branching. The unit clause rule has been used since [3] for reducing SAT instances.

Rule 1 *(Unit clause) If $\{l_j\} \in F$ then set $F := F[l_j = true]$.*

For feasibly self-reducible f on satisfying assignments, let us incorporate the unit clause rule into the previous algorithm, between Steps 2 and 3. Now we observe that, in order to say that a clause $c \in F$ is not in F_{cover}, rather than requiring *all* k lvariables of c to be assigned values in the first δn variables, now we only need $k-1$ of the variables to be assigned. For if one of them made c true, c is no longer present, and if all $k-1$ of them were $false$ in c then the unit clause rule applies.

This leads us to a better equation for δ, namely $\delta = (1 - \delta^k - k\delta^{k-1})\Delta$, since the probability that at least $k-1$ variables of any clause c appear in the first δn variables of σ is $1 - \delta^k - k\delta^{k-1}$, the third term coming from the fact that there are k ways to choose $k-1$ of the variables in c that do not appear.

As might be expected, this equation yields better time bounds. There is no longer a dependence on Δ, and we obtain bounds such as the following:

Corollary 2. *#3SAT is solvable in $\tilde{O}(1.4461^n)$ expected time.*

For $k \geq 5$, even an improvement in SAT (over previous algorithms) is observed. The best known algorithm in that case has been that of Paturi, Pudlak, Saks, and Zane [11], which has the bounds 1.5681^n and 1.6370^n for $k = 5$ and 6. We have found through numerical experiments that our algorithm does strictly better for $k \geq 5$. An example:

Corollary 3. *5-SAT and #5-SAT are solvable in $\tilde{O}(1.5678^n)$ expected time, while 6-SAT and #6-SAT are solvable in $\tilde{O}(1.6065^n)$.*

A sharper improvement can be made for #2SAT, since for large Δ, single variable branches can remove many variables due to the unit clause rule. Specifically, in the worst case, one variable is assigned in one branch, while at least 2Δ variables are assigned in another. We omit the analysis for space considerations, but can include it later if you like.

Theorem 2. *#2SAT is solvable in $\tilde{O}(1.2923^n)$ expected time.*

5 Conclusion

We have shown, in a very general manner, how various hard properties of k-CNF properties may be determined in less than 2^n steps. However, our procedure requires exponential space in order to achieve this. Therefore one obvious open problem is to find algorithms that can compute self-reducible formula properties in *polynomial* space. Another question (which we believe to be not so difficult, but did not work in time for submission) is how to derandomize the algorithm– i.e. convert it a deterministic one, without much loss in efficiency. A further direction is to use some clever properties of *Max-k-SAT* when $\Delta = \omega(1)$ to get an less-than-2^n algorithm for general *Max-k-SAT*.

Finally, it is worth exploring what other useful properties of CNF formulas can be expressed via our definition of self-reducible functions, to determine the full scope of the method we have described. One hard problem that probably *cannot* be computed with it is solving quantified Boolean formulas; this is because in QBFs, it seems crucial to maintain the fixed variable ordering given by the quantifiers. On the other hand, if we assume the number of quantifier alternations is small, this may permit one to use a variable-reordering approach of the form we have described.

References

1. N. Bansal and V. Raman. *Upper bounds for MaxSat: Further improved.* Proc. of the 10th ISAAC, 1999.
2. V. Dahllöf, P. Jonsson, Magnus Wahlstrm. *Counting Satisfying Assignments in 2-SAT and 3-SAT.* Proc. of COCOON, 535-543, 2002.
3. M. Davis and H. Putnam, *A computing procedure for quantification theory.* Journal of the ACM, 7(1):201-215, 1960.
4. E. A. Hirsch. *New worst-case upper bounds for SAT.* Journal of Automated Reasoning, Special Issue II on Satisfiability in Year 2000, 2000. A preliminary version appeared in Proceedings of SODA 98.
5. I. Lynce and J. Marques-Silva. *Complete unrestricted backtracking algorithms for satisfiability.* In Fifth International Symposium on the Theory and Applications of Satisfiability Testing, 2002.
6. T. Hofmeister, U. Schoening, R. Schuler, O. Watanabe. *A probabilistic 3-SAT algorithm further improved.* Proc. of STACS, 192202, 2002.
7. B. Monien, E. Speckenmeyer, *3-satisfiability is testable in $O(1.62^r)$ steps,* Bericht Nr. 3/1979, Reihe Theoretische Informatik, Universität-Gesamthochschule-Paderborn.
8. B. Monien and E. Speckenmeyer. *Upper bounds for covering problems.* Bericht Nr. 7/1980, Reihe Theoretische Informatik, Universität-Gesamthochschule-Paderborn.
9. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. *Chaff: Engineering an Efficient SAT Solver*. Proc. DAC-01, 2001.
10. R. Paturi, P. Pudlak, and F. Zane. *Satisfiability Coding Lemma.* Proc. of the 38th IEEE FOCS, 566-574, 1997.
11. R. Paturi, P. Pudlak, M. E. Saks, and F. Zane. *An improved exponential-time algorithm for k-SAT.* Proc. of the 39th IEEE FOCS, 628-637, 1998.
12. M. Robson. *Algorithms for maximum independent sets.* Journal of Algorithms, 7(3):425-440, 1986
13. U. Schoening. *A probabilistic algorithm for k-SAT and constraint satisfaction problems.* Proc. of the 40th IEEE FOCS, 410-414, 1999.
14. B. Selman, H. Kautz, and B. Cohen. *Local Search Strategies for Satisfiability Testing.* Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge, 1993.
15. R. Williams. *Algorithms for quantified Boolean formulas.* Proc. ACM-SIAM SODA, 299-307, 2002.