On Computing k-CNF Formula Properties

Ryan Williams*

Computer Science Department, Carnegie Mellon University
Pittsburgh, PA 15213 USA

Abstract. The latest generation of SAT solvers (e.g. [11, 8]) 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 $k \geq 2$, *Max-k-SAT* is solvable in $O(c^n)$ expected time for some $c < 2$ (where n is the number of variables), 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 $O(1.8932^n)$ worst-case expected time. We also improve the known time bounds for exact solution of $\#2SAT$ in general, and the bounds on k-SAT for $k \geq 5$ when Δ is constant.

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 [9] gave a $\tilde{O}(1.618^n)$ worst-case time algorithm for 3-SAT [9].[1] 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.* [12, 5, 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 [15] gave a $\tilde{O}(1.3333^n)$ algorithm for 3-SAT that is essentially the WalkSAT algorithm [16]; this was followed by a $\tilde{O}(1.3303^n)$ improvement a couple of years later [7].

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,

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

[1] Note: All time bounds in this paper will be *worst-case*.

we will resolve the question in the affirmative, when the clause density Δ is constant. Further, there has been strong recent progress in counting satisfying assignments [3]: #2SAT and #3SAT [2] 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 enough k, our algorithm outperforms the previous SAT algorithms, assuming $\Delta = O(1)$. 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 [12, 15, 7] 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:*

[2] The #3SAT bound holds provided that Δ is upper-bounded by some fixed constant; see Section 4.5.

(1) $\forall i \in \mathbb{N}$, $f(\top, i)$ 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 [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	$\max\{o(x, F) + v_1, o(\overline{x}, F) + v_2\}$	0	0	c^n ($c < 2$)	
#SAT	$v_1 + v_2$	2^i	0	1.2923^n	1.4461^n

(We define $o(l, F)$ 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 [14, 17]. The three major differences here are the use of randomness, the manner in which dynamic programming is employed, and the tighter analysis that results from analyzing k-CNF formulas.

Roughly speaking, the algorithm first chooses a random subset of δn variables, then does a standard depth-first branching on these variables, 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 major point is that for suitable δ (depending on k and Δ), the space usage necessary is small, and the expected runtime is greatly reduced asymptotically.

[3] The bounds for $k = 3$ assume $\Delta = O(1)$; that is, the clause density m/n (of 3-CNF formulas given as input) is upper-bounded by some fixed constant.

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. Let F be a k-CNF formula with n variables. Choose a random permutation $\sigma : [n] \to [n]$. (We will naturally think of σ as a permutation on the variables of F.) Define $F_{cover} \subseteq F$ as:

$$F_{cover} := \{c \in F \mid \forall\, l_i \in c.\ \ l_i \in \{x_{\sigma(1)}, \overline{x_{\sigma(1)}}, x_{\sigma(2)}, \overline{x_{\sigma(2)}}, \ldots, x_{\sigma(\delta n)}, \overline{x_{\sigma(\delta n)}}\}\}.$$

That is, F_{cover} is the subset of clauses c such that all k variables of c are contained in $\{x_{\sigma(1)}, \ldots, x_{\sigma(\delta n)}\}$. Define $F_{rem} := F - F_{cover}$. (F_{cover} is the set of clauses "covered" by the first δn variables of σ, and F_{rem} is the set that might possibly "remain", when the first δn variables of σ are set to values.)

2. Define $\leq_c$ to be a lexicographic (total) ordering on the clauses of F_{rem}, 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 f be some feasibly self-reducible function we wish to compute for F. Let V be the co-domain of f. (Typically, V is either $\{true, false\}$ or $\mathbb{N}$.) Initialize the set *Learned* $\subseteq \{0,1\}^{m(F_{rem})} \times \{1, \ldots, n\} \times V$ 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 $F_r = \bot$ or $F_r = \top$; take step 2.]

2. If $F_r = \bot$ or $F_r = \top$, return $f(\top, i)$ or $f(\bot, i)$, 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 clauses that *remain* (possibly $< k$-)clauses in F_r. (We say a clause of F *remains* in F_r if it (a) has not been satisfied by the partial assignment that reduced F to F_r, and (b) has also not been *falsified*; that is, at least one of its literals has not been set false.) It follows that $F_r^k \subseteq F_{rem}$; see the analysis in the following subsection.

Represent F_r as a pair $(b(F_r^k), i)$, where $b(F_r^k)$ is a vector of $m(F_{rem})$ bits: for $j = 1, \ldots, m(F_{rem})$,

$$b(F_r^k)[j] := 1 \iff c_j \in F_{rem} \text{ (the } j\text{th clause in } \leq_c) \text{ remains in } F_r.$$

(The analysis subsection will further explain why this uniquely represents F_r.[4])

5. If $(b(F_r), i, v) \in \mathit{Learned}$ for some v, then return v.

6. Else, let b_t and b_f be the bit vector representations of $F[x_{\sigma(n-i+1)} = \textit{true}]$ and $F[x_{\sigma(n-i+1)} = \textit{false}]$, respectively.

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

Set $v := g(x_{\sigma(n-i+1)}, v_t, v_f)$.

Update *Learned* := *Learned* $\cup \{(b(F_r), i, v)\}$, and return v.

4.3 Details of 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$(F_r[x_{\sigma(i)} = \mathit{true}], i-1)$ and Compute-f$(F_r[x_{\sigma(i)} = \mathit{false}], i-1)$ return correct answers.

- i always equals the number of variables that have not been set to values by the algorithm; the proof is a simple induction. Hence if $i < n - \delta n$, then the first δn variables have all been set in F_r, so letting $F_r^k \subseteq F$ be the set of original k-clauses in F that correspond to the clauses of F_r, $F_r^k \subseteq F_{rem}$ follows from the definition of F_{rem}: any clause $c \in F_r^k$ *cannot* be in F_{cover} (if $i < n - \delta n$, then the first δn variables have been set, hence by definition by F_{cover}, every literal in $c \in F_{cover}$ has been set, so $c \notin F_r^k$), hence $c \in F_{rem}$.

- In Steps 4 and 5, notice the representation $(b(F_r^k), i)$ tells us two things: (a) which clauses of F_{rem} have been either satisfied or falsified (and which have not) to yield F_r, and (b) which variables have been set to values in F_r (those variables that have been set are just those $x_{\sigma(j)}$ where $j < i$).

Thus, if literals of these variables appear in the (un-satisfied and un-falsified) clauses specified by $b(F_r^k)$, we may infer that these literals are *false*, as in the example of

[4] To illustrate with an example, suppose $F_{rem} = \{\{a, z\}, \{\overline{a}, b\}, \{\overline{a}, y\}, \{x, y\}\}$ and $F_r = F[a = \textit{true}, y = \textit{false}] = \{\{b\}, \{x\}\}$. Then $F_r^k = \{\{\overline{a}, b\}, \{x, y\}\}$: $\{\overline{a}, y\}$ is not included because it's falsified, and $\{a, z\}$ is not included because it's satisfied. If the ordering $\leq_c$ is given by $\{a, z\} \leq_c \{\overline{a}, b\} \leq_c \{\overline{a}, y\} \leq_c \{x, y\}$, then $b(F_r^k) =$ [0 1 0 1]. Observe that, given $b(F_r^k)$ and the knowledge that a and y have been set to some values, this is enough to reconstruct F_r: the presence of y in $\{x, y\} \in F_r$ implies that y was set *false*, and the presence of a in $\{\overline{a}, b\} \in F_r$ implies that a was set *true*.

the footnote on the preceding page. Therefore we can reconstruct F_r given the pair $(b(F_r^k), i)$: $b(F_r^k)$ tells us F_r^k, which is the set of clauses in F that remain in F_r, and i tells us which literals in those clauses of F_r^k do not appear in F_r (*i.e.* are set to *false*). Hence the map $F_r \mapsto (b(F_r), i)$ is 1-1, and it is semantically correct to return v for $f(F_r)$ if $(b(F_r), i, v) \in$ *Learned* in Step 5.

For every f-value computed, it is stored 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 $m(F_{rem})$ 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 $\tilde{O}(2^{E[m(F_{rem})]})$ expected time for the learned values phase, where $E[m(F_{rem})]$ is the expected number of clauses in F_{rem} over the choice of random σ. Hence in total, the expected runtime is $\tilde{O}(2^{E[m(F_{rem})]} + 2^{\delta n})$, and the optimal choice of δ to minimize this expression will make $E[m(F_{rem})] = \delta n$.

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. It will *first* (a) construct the *Learned* set of f-values recursively, saving each discovered value as it goes along. Then it will (b) run the branching phase until depth δn, in which case it simply refers to the corresponding stored value in *Learned*.

It is clear that if the runtime of (a) is bounded by T, then the runtime of this procedure is $\tilde{O}(2^{\delta n} + T)$, as looking up an f-value in *Learned* takes only polynomial time. Thus it suffices for us to prove that (a) takes $\tilde{O}(2^{E[m(F_{rem})]})$ expected worst-case 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] $= \tilde{O}(2^{E[m(F_{rem})]})$,

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

Let us specify the procedure for constructing *Learned* more formally. Start with $(\bot, i)$ and $(\top, i)$ for every $i = 1, \ldots, n$, and put $(\bot, i, f(\bot, i))$ and $(\top, i, f(\top, i))$ in *Learned*.

0. Initialize $j := n - 1$.

1. Repeat steps 2-3 until $j = \delta n$:

2. Set $\mathcal{F} := \{F_r \cup \{c \in F | x_{\sigma(j)} \in c \ \vee \ \overline{x_{\sigma(j)}} \in c\} \ \mid \ \exists v.\ (b(F_r), j+1, v) \in \mathit{Learned}\}$. That is, $\mathcal{F}$ represents the class of all formulas that currently have values

in *Learned*, each one being unioned with the clauses that contain $x_{\sigma(j)}$. (Observe that successive members of this set can be generated in order, with polynomial time delay.)

3. For all $F_r \in \mathcal{F}$,
Find v_t and v_f such that $(b(F_r[x_{\sigma(j)} = true]), i+1, v_t)$ and
$(b(F_r[x_{\sigma(i)} = false]), i+1, v_t)$ in *Learned*, using a search tree.
Put $(b(F_r), i, g(x_{\sigma(i)}, v_t, v_f))$ 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 $n2^{m(F_{rem})}$ values in *Learned*, in total. Each iteration of the for-loop for a fixed F_r takes polynomial time. The number of possible F_r in $\mathcal{F}$ is at most $2^{m(F_{rem})}$ (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 $\tilde{O}(2^{E[m(F_{rem})]})$ expected time, in the worst case.

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 $\tilde{O}(c^n)$ expected (worst-case) 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 $\delta^k[1 - o(1)]$. (So the probability that $c_i \in F_{cover}$ is this quantity.) 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 $c_i \in F_{rem} = F - F_{cover}$ 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 if $c_i \in F_{rem}$, and 0 otherwise. Then the expected number of clauses in F_{rem} is

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

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

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

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 $m(F_{rem})$ 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, while $p(0) = -\Delta$. Since $p(1) > 0$ and $p(0) < 0$, it follows that there is an $r \in (0, 1)$ satisfying $p(r) = 0$. □

A remark on the tightness of this result. We have empirically observed that as either Δ or k increase, the relevant root of $p(x)$ approaches 1. Thus, if either k or Δ are unbounded functions in terms of n, we cannot (using the above analysis) guarantee any $\delta < 1$ such that the aforementioned procedure takes at most $2^{\delta n}$ time.

4.4 Max-k-SAT solution

Ever since Monien and Speckenmeyer [10] 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 exact algorithms for this problem have runtimes of the form $O(c^m)$, with c decreasing slowly over time (*e.g.* [10, 1, 5]). One prior algorithm was very close to a c^n time bound: a $(1-\epsilon)$ approximation scheme was given by Hirsch [6] that runs in $(c_\epsilon)^n$ for some c_ϵ; however, c_ϵ approaches 2 as ϵ approaches 0.

A corollary of our above theorem is that when k and the clause density Δ are constant, there exists a less-than-2^n algorithm. 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 constant 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 the 60's [4] 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, we 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_{rem}, rather than requiring *all* k variables of c to be assigned values in the first δn variables, only $k-1$ of the variables need to be assigned: if one of them makes c true, c is no longer present; if $k-1$ literals are *false* in c then the unit clause rule applies.

This gives us a slightly 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 at least $1-\delta^k-k\delta^{k-1}$, the third term coming from the fact that there are k ways to choose $k-1$ of the variables in c that appear. As might be expected, this equation yields better time bounds. There is no longer a strict dependence on Δ, and we obtain bounds such as the following:

Theorem 2. *#3SAT is solvable in $\tilde{O}(1.4461^n)$ expected time, for any constant Δ.*

Proof. (Sketch) Assume Δ is fixed. We wish to compute the largest possible $\delta \in (0,1)$ s.t. $\delta = (1-\delta^3-3\delta^2)\Delta$, *i.e.*

$$\Delta\delta^3 + 3\Delta\delta^2 + \delta - \Delta = 0.$$

This cubic equation has three solutions; the only one that is non-negative for $\Delta > 0$ is:

$$\delta(\Delta) = -\frac{F^{1/3}}{12\Delta} + \frac{3\Delta-1}{F^{1/3}} - 1 + i\frac{\sqrt{3}}{2}\left(\frac{F^{1/3}}{6\Delta} + \frac{2-6\Delta}{F^{1/3}}\right),$$

where

$$F = \left(-108\Delta + 108 + 12\sqrt{\frac{12}{\Delta} - 27 + 162\Delta - 243\Delta^2}\right)\Delta^2.$$

When $\Delta = 1$, it is straightforward to calculate that $\delta(1) = \sqrt{2}-1 \approx 0.414\ldots$.

For $\Delta > 1$, the solution for $\delta(\Delta) > \sqrt{2}-1$. F is dominated by $c_1 \cdot i \cdot \Delta^3 - c_2 \cdot \Delta^3$ for some constants c_1, c_2. Note that every term in the expression for δ involving F cancels out these Δ^3 terms (by taking a cube root and then dividing by Δ). Hence the expression for δ approaches a certain constant as Δ increases.

Our numerical experiments show that $\delta(\Delta) \to 0.53208\ldots$ as $\Delta \to \infty$. Therefore $2^{0.53208n} \leq 1.4461^n$ is the bound. □

For $k \geq 5$, even an improvement in SAT (over previous algorithms) is observed, using similar reasoning. The best known algorithm in that case has been that of Paturi, Pudlak, Saks, and Zane [13], 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 2. *5-SAT and #5-SAT are solvable in $\tilde{O}(1.5678^n)$ expected time for any constant Δ; 6-SAT and #6-SAT are solvable in $\tilde{O}(1.6065^n)$ for any constant Δ.*

Proof. (Sketch) Numerical solution of the equations $\delta^5 + 5\Delta\delta^4 + \delta - 1 = 0$ and $\delta^6 + 6\Delta\delta^5 + \delta - 1 = 0$ show that we can upper bound δ by $\delta \leq 0.6486\ldots$ and $\delta \leq 0.6839\ldots$, respectively. These δ-values yield the time bounds of the corollary. □

A sharper improvement can be given 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 to a value in one branch, while at least 2Δ variables are assigned in another.

Theorem 3. *#2SAT is solvable in $\tilde{O}(1.2923^n)$ expected time.*

Proof. (Sketch) Let c be a constant to be fixed later. We consider the following algorithm for computing #2SAT.

Given a k-CNF F,

(0) If $\{x\} \in F$ (resp. $\{\overline{x}\} \in F$), recursively compute the number of SAT assignments A for $F[x = true]$ (resp. $F[x = false]$), and return A.

(1) If $\Delta \geq c$, then we claim the average number of occurrences per variable in the 2-CNF F is at least $2c$. (Let o be the average; then $o \cdot n = 2m$, where m is the number of clauses.) Therefore there is at least one variable x that occurs in $2c$ clauses. Recursively compute the number of SAT assignments $A_{x=t}$ for $F[x = true]$. Then compute the number of SAT assignments $A_{x=f}$ for $F[x = false]$. Return $A_{x=t} + A_{x=f}$.

(2) If $\Delta < c$, then return Compute-f(F, n) for $f = \#SAT$, *i.e.* the dynamic programming algorithm for counting satisfying assignments.

Now we analyze the algorithm. Case (0) is just the unit clause rule.

Case (1) analysis: It can be shown that the worst case is when x appears either all positively or all negatively. In this case, if $x = true$ then only one variable (x) is removed from F in one recursive branch. When $x = false$, all of the variables in clauses with x are set to true by case (0); hence at least $2c + 1$ variables are removed from F in the other branch.

The analysis of Case (2) is simply that from previous sections.

This gives an upper bound on the time recurrence in terms of n:

$$T(n) \leq \max\{T(n-1) + T(n-2c),\ 2^{\delta n}\},$$

where $\delta \in (0, 1)$ is the smallest value such that $\delta < (1 - \delta^2 - 2\delta)c$. (The first term in the $\max$ corresponds to Case (1), the second term to Case (2).)

We want an integer c such that the runtime is minimized (so the two terms in the $\max$ are roughly equal). To do this, let $\lambda(A, B)$ be $r \in (1, 2)$ satisfying $1 - 1/r^A - 1/r^B = 0$. (Note that r is unique, and $O(\lambda(A, B)^n)$ is an upper bound on the recurrence $T(n) = T(n - A) + T(n - B), T(1) = 1$.)

Observe that as c increases, $\lambda(1, 2c)$ decreases while $\delta \in (0, 1)$ such that $\delta = (1 - \delta^2 - 2\delta)c$ increase. Thus, we wish to choose c such that both $\lambda(1, 2c)^n$ and $2^{\delta n}$ s.t. $\delta = (1 - \delta^2 - 2\delta)c$ are minimized.

Choosing $c = 3$, $\lambda(1, 2c) \approx 1.2555$ and $\delta \approx 0.36993$, so $2^\delta \approx 1.2923$. Hence the runtime of the procedure is upper bounded by $\tilde{O}(1.2923^n)$. □

A remark on applications. Consider the problem #Min2SAT, where we wish to count the number of satisfying assignments that have a minimum number of variables set *true*. (It is easy to check that #Min2SAT is feasibly self-reducible, by having the function f return a pair containing the *minimum number of* true*s in a satisfying assignment* and the *number of such assignments*. It follows that a simple generalization of the above procedure solves #Min2SAT.) There is a natural reduction from #Min2SAT to #Min-Vertex-Cover, which we will refrain from describing here. Quite interestingly, the existence of a good #Min-Vertex-Cover algorithm implies good algorithms for several other problems as well. We refer the reader to [2].

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. It is interesting to formulate our main result in the language of *strong backdoors* [18]. Informally, for a function f computing a property of F, a strong backdoor is a subset S of variables in F, defined with respect to some "subsolver" A that runs in polynomial time and solves special cases of the function f. A strong backdoor S has the property that systematically setting all possible assignments to the variables of S allows one to compute f on F, by using A to solve each reduced F that results from a variable assignment. Since each assignment takes only polynomial time to evaluate with A, the overall runtime is bounded by $2^{|S|}poly(n)$.

Our main result may be stated in terms of randomly chosen backdoors.

Theorem 4. *For any feasibly self-reducible f and k-CNF formula F with constant clause density Δ, there exists a $\delta \in (0, 1)$ s.t. a* random *subset S of δn variables is a strong backdoor for f with probability at least $1/2$, with respect to a subsolver A that runs with $2^{\delta n}$ preprocessing (and polynomial time on each assignment to variables in S).*

While the dynamic programming scheme used in this paper is very general, one obvious caveat is that the procedure requires exponential space in order to achieve this. Therefore one 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) is how to derandomize the algorithm– that is, 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.

6 Acknowledgements

Many thanks to the anonymous referees for their very helpful comments.

References

1. N. Bansal and V. Raman. *Upper bounds for MaxSat: Further improved.* Proc. of the 10th ISAAC, 247–258, 1999.
2. V. Dahllöf and P. Jonsson. *An algorithm for counting maximum weighted independent sets and its applications.* Proc. of ACM-SIAM SODA, 292–298, 2002.
3. V. Dahllöf, P. Jonsson, Magnus Wahlström. *Counting Satisfying Assignments in 2-SAT and 3-SAT.* Proc. of COCOON, 535-543, 2002.
4. M. Davis and H. Putnam, *A computing procedure for quantification theory.* Journal of the ACM, 7(1):201-215, 1960.
5. 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.
6. E. A. Hirsch. *Worst-case Time Bounds for MAX-k-SAT with respect to the Number of Variables Using Local Search.* ICALP Workshop on Approximation and Randomized Algorithms in Communication Networks, 69-76, 2000.
7. T. Hofmeister, U. Schöning, R. Schuler, O. Watanabe. *A probabilistic 3-SAT algorithm further improved.* Proc. of STACS, 192202, 2002.
8. 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.
9. B. Monien, E. Speckenmeyer, *3-satisfiability is testable in $O(1.62^r)$ steps*, Bericht Nr. 3/1979, Reihe Theoretische Informatik, Universität-Gesamthochschule-Paderborn.
10. B. Monien and E. Speckenmeyer. *Upper bounds for covering problems.* Bericht Nr. 7/1980, Reihe Theoretische Informatik, Universität-Gesamthochschule-Paderborn.
11. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. *Chaff: Engineering an Efficient SAT Solver*. Proc. DAC-01, 2001.
12. R. Paturi, P. Pudlak, and F. Zane. *Satisfiability Coding Lemma.* Proc. of the 38th IEEE FOCS, 566-574, 1997.
13. 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.
14. M. Robson. *Algorithms for maximum independent sets.* Journal of Algorithms, 7(3):425-440, 1986
15. U. Schöning. *A probabilistic algorithm for k-SAT and constraint satisfaction problems.* Proc. of the 40th IEEE FOCS, 410-414, 1999.
16. B. Selman, H. Kautz, and B. Cohen. *Local Search Strategies for Satisfiability Testing.* Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge, 1993.

17. R. Williams. *Algorithms for quantified Boolean formulas.* Proc. ACM-SIAM SODA, 299-307, 2002.
18. R. Williams, C. Gomes, and B. Selman. *Backdoors To Typical Case Complexity.* To appear in Proc. of IJCAI, 2003.