Fighting Perebor: New and Improved Algorithms for Formula and QBF Satisfiability

Rahul Santhanam
School of Informatics
University of Edinburgh
Edinburgh, United Kingdom.
Email: rsanthan@inf.ed.ac.uk

***Abstract*—We investigate the possibility of finding satisfying assignments to Boolean formulae and testing validity of quantified Boolean formulae (QBF) asymptotically faster than a brute force search.**

Our first main result is a simple deterministic algorithm running in time $2^{n-\Omega(n)}$ for satisfiability of formulae of linear size in n, where n is the number of variables in the formula. This algorithm extends to exactly counting the number of satisfying assignments, within the same time bound.

Our second main result is a deterministic algorithm running in time $2^{n-\Omega(n/\log(n))}$ for solving QBFs in which the number of occurrences of any variable is bounded by a constant. For instances which are "structured", in a certain precise sense, the algorithm can be modified to run in time $2^{n-\Omega(n)}$.

To the best of our knowledge, no non-trivial algorithms were known for these problems before.

As a byproduct of the technique used to establish our first main result, we show that every function computable by linear-size formulae can be represented by decision trees of size $2^{n-\Omega(n)}$. As a consequence, we get strong superlinear *average-case* formula size lower bounds for the Parity function.

***Keywords*-Satisfiability algorithms; random restrictions; average case lower bounds; quantified Boolean formulae**

I. Introduction

The P vs NP problem is the central intractable problem in theoretical computer science. Though the resolution of this problem seems unlikely in the near future, it is widely believed that the two classes are different. Many areas of theoretical computer science have been shaped by this belief - it underlies all of cryptography, and the study of various techniques for "coping with NP-hardness" such as approximation algorithms and parameterized complexity is driven pragmatically by the lack of efficient algorithms for solving NP-hard problems exactly.

One of the basic intuitions for this belief is the metaphor of the "needle in the haystack" - it should not be possible to explore an exponentially large unstructured space in polynomial time. Even before the P vs NP problem rose to prominence in academic circles in the West, the seeming unavoidability of brute force searches in certain contexts had been recognized and investigated extensively by Russian cyberneticians and mathematicians. They termed the exhaustive search algorithm "perebor", and Yablonski even claimed a proof that perebor was inevitable for certain problems [Tra84]. This proof, if correct, would have solved the P vs NP problem before it had even been stated! In any case, the folk wisdom still is that *in general*, NP problems cannot be solved more efficiently than exhaustive search or something "morally equivalent" to exhaustive search.

Over the past couple of decades, work on the exact complexity of natural NP-complete problems has tested this wisdom, albeit in a gentle way, by showing various upper bounds of the form c^n for $c < 2$, where n is the witness size. This work is excellently surveyed by Woeginger [Woe01], [Woe08]. Some of the most impressive improvements have been on the complexity of the Satisfiability problem - determining whether a formula in CNF has a satisfying assignment or not. Probabilistic algorithms for 3SAT running in time approximately 1.33^n are known, based on work of Schoning [Sch99] and Paturi, Pudlak, Saks and Zane [PPSZ98], while k-SAT is known to have algorithms that run in time $O(2^{n-n/k}poly(m))$, where n is the number of variables and m the number of clauses [DH08]. The algorithms for SAT take advantage of the clausal representations of formulae, raising the question of whether it is this simplicity of representation which is key to the improvements over brute force search.

In this paper, we study the solvability of satisfiability of *general Boolean formulae*, rather than formulae in CNF. We also study the solvability of the much harder PSPACE-complete QBF satisfiability problem on formulae, which asks whether a fully quantified Boolean formula evaluates to true. For both these problems, we get significant improvements over brute force search on natural classes of instances for which the problems

remain complete.

We explain our contributions in more detail in the next subsection. Here, we further motivate our investigations.

The problems we consider are very natural problems, and it is interesting from a theoretical point of view to find the best algorithms we can for them. This is important both in the search for new tools for our algorithmic arsenal, and for our understanding of analytical techniques.

In the case of the satisfiability problem, however, the interest in new and better algorithms goes far beyond the theoretical. Because of its conceptual simplicity and because most NP problems can be very efficiently encoded into SAT, SAT is arguably the most fundamental of all NP-complete problems. It has extensive practical applications in areas such as software and hardware verification and testing, design automation, planning and automated reasoning. This has motivated a lot of application-oriented work in designing fast SAT solvers, and there is a vibrant conference devoted purely to the theory and applications of SAT. Malik and Zhang [MZ09] have an excellent overview of the practical significance of the problem and of the state of the art in SAT solvers.

Most SAT solvers require their input to be in CNF form. As discussed by Thiffault, Bacchus and Walsh [TBW04], this can be a significant disadvantage in contexts such as hardware verification where the problem more naturally has a formula or circuit structure. First, the conversion to CNF destroys important structural information about the original problem. Second, the conversion to CNF adds several new variables, blowing up the size of the search space. Therefore, it is of interest to try and solve satisfiability directly on general Boolean formulae and circuits.

While highly optimized SAT solvers do exist and are effective in various contexts, the situation is very different for QBF satisfiability. Efficient QBF solvers would be of great benefit in applications such as model checking and verification, and multi-agent planning [GIB09], however the effectiveness of QBF solvers lags far behind that of SAT solvers. Thus insights from theory have potentially much greater impact here. In this context even more so than for SAT, techniques for solving the problem directly on formulae would be useful [GIB09], [Zha06]. This is because solvers do not often succeed in pruning the search space by much, therefore the cost of adding new variables by encoding into CNF is greater.

Finally, motivation for addressing these problems comes from complexity theory. This is not merely because SAT is NP-complete, but is due rather to the fascinating interplay between upper and lower bounds in this area. Just as is the case in the theory of pseudorandomness and in the study of optimization algorithms and the Unique Games conjecture, research on better upper bounds for SAT has gone hand-in-hand with developments in lower bound techniques. In papers of Impagliazzo, Paturi, Pudlak, Zane and others [PPZ97], [PPSZ98], [IP01], [IPZ01], [CIP09], structural properties of CNF formulas, such as the Satisfiability Coding Lemma and the Sparsification Lemma, are used *both* in designing improved algorithms for SAT and in proving close to exponential lower bounds for depth-three circuits.

The relationship between improved algorithms for satisfiability and lower bounds has been made more explicit in a recent paper of Williams [Wil10]. He shows that algorithms which perform even *slightly* better than brute force search for Circuit Satisfiability imply super-polynomial size circuit lower bounds for NEXP. He also shows similar results for satisfiability on more restricted kinds of circuits and formulae.

Our results in this paper provide yet another example of the synergy between upper bound and lower bound techniques. As we describe in more detail below, our algorithm for formula satisfiability is inspired by the random restriction method for proving superlinear lower bounds on formula size [B.A61], [Has98]. In analyzing our algorithm, we discover a new structural characterization of formulae, which can in turn be used to derive *average-case* lower bounds on formula size. Thus we have a situation where a lower bound approach inspires an upper bound approach, which in turn inspires a refined version of the original lower bound approach!

A. Our Contributions

We study the formula satisfiability and QBF satisfiability problems on formulae of size linear in the number of variables. Note that the Formula Satisfiability problem remains NP-complete and the QBF satisfiability problem PSPACE-complete [Pap94], [Heu99] even for formulae with at most *two occurrences* of each variable.

Our first main result is a simple deterministic algorithm for formula satisfiability on formulae of linear size which saves a constant factor in the exponent over brute force search.

Theorem 1. *There is a deterministic algorithm for FormulaSAT which runs in time* $2^{n-\Omega(n)}$ *on formulae of size* $O(n)$.

The algorithm of Theorem 1 is a simple and natural algorithm which follows the DPLL (Davis-Putnam-

Logemann-Loveland) paradigm of searching for a satisfying assignment by setting a certain variable, simplifying the resulting formulae and searching recursively on the simplified formulae. The keys are to choose a good simplification procedure and to be able to analyze the running time.

Our simplification procedure and part of our analysis are inspired by the random restriction method to prove formula size lower bounds [B.A61], [Has98]. The basic idea is that when one hits a formula of size L on n variables with a random restriction setting all but a fraction p of the variables, one expects the formula to shrink by more than a linear amount - to size approximately p^2L in Hastad's optimal analysis. Now, if one chooses a small enough p for $L = O(n)$, the expected size of the simplified formula is actually a constant factor *smaller* than the number of unset variables - this can be used to save over brute force search.

This basic idea does not work directly because the analyses of Subbotovskaya and Hastad only deal with the *expected* reduction in formula size. Since we are searching for a satisfying assignment, setting variables at random will not be sufficient for us. What we will need to do is prove a *concentration* version of the shrinkage result, and this is the main technical part of our work. A further point is that our algorithm is deterministic - we prove that it works to choose which variable to set in a greedy way rather than at random. We note that other SAT algorithms inspired by random restrictions [PPSZ98], [CIP09] are either intrinsically randomized or take some effort to derandomize.

Our proof gives a non-trivial (constructive) *decision tree* representation of functions computed by linear-size formulae, which might be of independent interest. We use this representation, together with the fact that average-case decision tree complexity of Parity is easily characterized, to derive strong superlinear average-case formula size lower bounds for Parity.

Theorem 2. *Any sequence of formulae of size $O(n)$ must err in computing Parity on at least a $1/2-1/2^{\Omega(n)}$ fraction of inputs of length n, for each n.*

Our algorithm for formula satisfiability also extends to counting satisfying assignments exactly, within the same bound.

Next we turn our attention to the much harder QBF satisfiability problem. We design algorithms for this problem on formulae with a bounded number of variable occurrences. Recall that the problem is PSPACE-complete even when the number of variable occurrences is bounded by 2. For this problem, we are not able to get a constant factor improvement in the exponent over brute force search, but we come close.

Theorem 3. *For any constant k, there is an algorithm for QBF satisfiability which runs in time $2^{n-\Omega(n/\log(n))}$ on formulae with at most k occurrences of each variable.*

The restriction idea we use for formula satisfiability does not work here, and we rely on a completely different technique based on memoization which we call the "bottleneck method".

We are able to save a constant factor in the exponent over brute force search when the instances are "structured" in a certain natural sense. As Malik and Zhang say in their survey on SAT [MZ09], "it is exactly the non-adversarial nature of practical instances that is exploited by SAT solvers". It is an important theoretical question how to best model this "non-adversarial nature", and to use this to provide a convincing explanation for the success of SAT solvers. Here we model the structuredness of instances in the *broadest* possible way - a class of instances is structured if each one has a succinct representation from which it can be extracted in polynomial time. Even for this very broad notion of structure, we show an improved algorithm for QBF satisfiability.

Theorem 4. *(Informal) For any constant k and any structured set S of inputs, there is an algorithm for QBF satisfiability which runs in time $2^{n-\Omega(n)}$ on formulae with at most k occurrences of each variable.*

Our best algorithms have running times of the form $2^{(1-\delta)n}$, where $\delta > 0$ is some small constant. One might wonder if this is necessary - could we hope to have algorithms with running time $2^{n/2}$ or even $2^{o(n)}$ for our problems? The Exponential Time Hypothesis (ETH) of Impagliazzo, Paturi and Zane [IPZ01] says that is impossible. ETH is a robust hypothesis ruling out algorithms with running time $2^{o(n)}$ for many NP-complete problems, and is closely connected to standard hypotheses in parameterized complexity. In fact, one of the implications of the work of Impagliazzo, Paturi and Zane is that $2^{o(n)}$ time algorithms for CNF SAT, $2^{o(m)}$ time algorithms for CNF SAT (where m is the number of clauses), and $2^{o(n)}$ time algorithms for FormulaSAT on linear size formulae are all *equivalent* to the negation of ETH.

We mentioned before the recent work of Williams [Wil10] deriving lower bounds from improved algorithms for satisfiability. Can his results be used in conjunction with ours to derive formula size lower bounds? Not quite, but his results do show that our results are nearly tight, modulo the development of new

lower bound techniques. For instance, implicit in his work is the following result: there is a universal constant d such that if FormulaSAT can be solved on formulae of size n^c in time $2^n/n^{\omega(1)}$, then E^{NP} does not have formulae of size $n^{c/d}$. Thus, if we could extend our algorithms to work on polynomial-size formulae with a large enough exponent, we would get new formula size lower bounds! Maybe it is no accident that our algorithmic technique is so closely connected to known lower bound techniques for formula size...

B. Related Work

As mentioned before, there has been a lot of work on algorithms for 3SAT and CNF SAT, of which we can't give an exhaustive survey here. Some of the highlights are the work of Monien and Speckenmeyer [MS85] on analyzing DPLL-type algorithms for SAT, the work of Schoning and others [Sch99], [DGH02] on local search algorithms, and the work of Paturi, Pudlak, Saks and Zane [PPZ97], [PPSZ98] on randomized splitting algorithms. Dantsin, Hirsch, Ivanov and Vsemirnov have a survey [DHIV01] that is slightly out-of-date now, and Dantsin and Hirsch [DH08] have a newer survey.

There is no general result solving CNF SAT in time c^n for some $c < 2$ if the clause size is not bounded. Wahlstrom [Wah05] gives a deterministic algorithm with a running time of this form for CNF formulae of linear length - our result is stronger because we get a running time of this form for formulae of linear length which are not restricted to be in CNF.

There has been little theoretical work on general formula satisfiability or QBF satisfiability. There are two papers that relate somewhat to ours. The first is an algorithm due to Calabro, Impagliazzo and Paturi [CIP09] which has a running time of the form $c^n, c < 2$ for satisfiability of *constant-depth* Boolean *circuits* of linear size. This result is incomparable with ours because we have no restriction on depth, however we can only deal with formulae. Unlike their algorithm, ours is deterministic and the improvement over brute force search has a nicer dependence on the formula size.

Williams [Wil02] has some algorithms for QBF satisfiability. However, he considers QBF satisfiability on CNF formulae, and his analysis is in terms of the number of clauses m rather than the number of variables n. He obtains algorithms with running time $c^m, c < 2$ for various different constants c depending on the maximum clause size in the CNF formula.

II. Preliminaries

We describe here some of the fundamental concepts and notation we use.

The model of computation we use is the standard random access Turing machine model. Such a machine has a read-only input tape along with sequential and random-access work tapes. The random-access work tapes are accessed through address tapes which themselves are sequential.

Two kinds of representations for Boolean functions are central in this paper - the *formula* representation and the *decision tree* representation.

A formula is a binary tree with internal nodes labelled with binary connectives and leaves labelled by literals, i.e., variables or complements of variables, or constants, i.e. 0 or 1. A formula corresponds to a Boolean function in the obvious manner - leaves correspond to the functions they're labelled with, and a node labelled with connective h and with children corresponding to functions f_1 and f_2 respectively corresponds to the function $h(f_1, f_2)$. The function computed by a formula is the function corresponding to its root.

When we refer to formulae, by default we mean De Morgan formulae in which the connectives are AND and OR. When we use different connectives, we explicitly state that this is the case. We call a formula balanced if it is the complete binary tree. The size of a formula is the number of leaves. Occasionally we refer to the size of a formula ϕ as $|\phi|$. We distinguish the *size* of a formula from its *length* - the length refers to the number of bits when representing the formula in binary. Using a standard encoding, formulae of size s have length $O(s \log(s))$.

Given a formula ϕ, the Formula Satisfiability (FormulaSAT) problem asks if there is a setting of Boolean values to the variables of ϕ for which the formula evaluates to 1. The #FormulaSAT problem is the counting version of this- it asks for the exact number of satisfying assignments of a Boolean formula.

A *quantified Boolean formula* (QBF) in prenex normal form is an expression of the form $Q_1 x_1 Q_2 x_2 \ldots Q_n x_n \phi(x_1, x_2 \ldots x_n)$, where each $Q_i, 1 \leqslant i \leqslant n$ is either $\exists$ or $\forall$ and ϕ is a formula. When we refer to QBFs, by default we mean QBFs in prenex normal form. The semantics of QBFs, i.e., when a QBF evaluates to true, is defined inductively in the standard way. The QBF satisfiability problem asks whether an input QBF evaluates to true. Note that the Formula Satisfiability problem is a special case of QBF Satisfiability, where all quantifiers are $\exists$.

The other kind of representation for Boolean functions we're interested in is the decision tree representation. A decision tree is a binary tree whose internal nodes are labelled with variables and leaves with constants, i.e., 0 or 1. A decision tree implicitly represents

a Boolean function f as follows. Each input x defines a path from the root to the leaf, where at a node labelled wlog by x_i (the i'th input variable of x) the path continues through the left child of the node if $x_i = 0$ and through the right child if $x_i = 1$. The value of f at x is 0 if the path ends in a leaf labelled 0, and 1 otherwise.

The *size* of a decision tree is the number of leaves of the tree. Every function on n bits can be represented by a decision tree with 2^n leaves, and in some cases this is tight, eg., for the Parity function.

We also need the notion of time-bounded Kolmogorov complexity of a string. Fix a universal Turing machine U with an output tape. The Kolmogorov complexity of a string x with respect to U, $K_U(x)$ is the length of the smallest program q such that $U(q) = x$. If we have a certain universal machine in mind, we drop the prefix - Kolmogorov complexity is robust with respect to the choice of universal machine, up to an additive constant. We are interested in *time-bounded* Kolmogorov complexity, where the time taken by the program to output the string is also taken into account. Given a polynomial p, $C^p(x)$ is the length of the smallest program q such that $U(q) = x$, and q halts in time less than $p(|x|)$. For any string x and polynomial p that is at least linear, $C^p(x) \leqslant |x| + O(1)$; on the other hand, by a counting argument, there is an *incompressible* string, i.e., a string x for which $C^p(x) \geqslant |x|$.

Occasionally we use the "O*" notation for function growth. Given functions $f(n)$ and $g(n)$, we say that $g(n) = O^*(f(n))$ iff $g(n) = O(f(n)\mathsf{polylog}(f(n)))$.

The rest of this paper is organized as follows. In Section III, we give our algorithm for formula satisfiability. We also discuss our average-case lower bound for Parity there. In Section IV, we sketch our results for QBF satisfiability.

III. The Algorithm for Formula Satisfiability

In this section, we describe our main result: an improved algorithm for Formula Satisfiability when the formula is of length linear in the number of variables.

The basic idea is to set variables greedily so that the size of the reduced formula is substantially smaller than the size of the original formula. What we are aiming for is a sublinear scaling of the formula size with the number of unset variables, so that when a certain constant fraction α of variables have been set, the size of the reduced formula is less than $(1-\alpha)\beta n$, for some constant $\beta < 1$. This would imply, of course, that the reduced formula can depend on at most $(1-\alpha)\beta n$ variables, and thus we save an additive term of $(1-\alpha)(1-\beta)$ in the exponent over brute force search.

Hastad's work [Has98] on the shrinkage exponent of de Morgan formulae does imply that there is a sublinear scaling of formula size with unset variables *on average*, i.e., for a random setting of variables. This suffices for Hastad's purposes, since he is aiming to derive a lower bound on formula size for an explicit Boolean function, but not for ours - we also need to analyze what happens in a worst-case setting of variables. In such a case, a sublinear scaling does not occur in general, and there are examples of formulae for which this is the case. What we do show is that if that variables are set in a certain fashion, the probability that a sublinear scaling does not occur is exponentially small, and this suffices for us to save over brute-force search. Implicitly, what we are doing is proving a concentration version of the results on shrinkage exponent. We don't show a concentration version of Hastad's rather technical result, but instead of Subbotovskaya's result [B.A61] that the shrinkage exponent is at least $3/2$. Since we are interested in an algorithm for satisfiability, we also need a procedure to set the variables, and we show that a simple greedy procedure works.

The algorithm is specified below.

EvalFormula (ϕ: Formula; n: integer)

1) If ϕ has no literals, return "yes" if ϕ evaluates to 1 else return "no"
2) Let r be such that x_r is the variable occurring the maximum number of times in ϕ
3) Let $\phi_0 \leftarrow Simplify(\phi\mid_{x_r=0})$
4) Let $\phi_1 \leftarrow Simplify(\phi\mid_{x_r=1})$
5) EvalFormula(ϕ_0, $n-1$)
6) EvalFormula(ϕ_1, $n-1$)

The procedure Simplify reduces the size of a formula by applying rules to eliminate constants and redundant literals. These are the same rules of simplification used by Hastad [Has98] in his proof that the shrinkage exponent of formulae is 2.

Simplify(ϕ: Formula)

Repeat the following until there is no decrease in size of ϕ:

- If $0 \vee \psi$ or $\psi \vee 0$ occurs as a subformula, where ψ is any formula, replace this subformula by ψ

- If $0 \wedge \psi$ or $\psi \wedge 0$ occurs as a subformula, where ψ is any formula, replace this subformula by 0
- If $1 \wedge \psi$ or $\psi \wedge 1$ occurs as a subformula, where ψ is any formula, replace this subformula by ψ
- If $1 \vee \psi$ or $\psi \vee 1$ occurs as a subformula, where ψ is any formula, replace this subformula by 1
- If $y \vee \psi$ or $\psi \vee y$ occurs as a subformula, where ψ is a formula and y is a literal, then replace all occurrences of y in ψ by 0 and all occurrences of $\bar{y}$ by 1
- If $y \wedge \psi$ or $\psi \wedge y$ occurs as a subformula, where ψ is a formula and y is a literal, then replace all occurences of y in ψ by 1 and all occurrences of $\bar{y}$ by 0

We term the first two rules 0-simplification rules, the next two 1-simplification rules and the final two variable simplification rules. It is obvious that the 0-simplification and the 1-simplification rules preserve the set of satisfying assignments of the formula. We clarify why the first variable simplification rule results in an equivalent formula; the analysis of the second one is dual. Let ϕ' be a subformula of the form $y \wedge \psi$, where y is a literal and ψ is a formula. When $y = 0$, this is clearly equivalent to the simplification of ψ where y has been set to 0 and $\bar{y}$ to 1 in ψ. When $y = 1$, this forces ϕ' to 1 and therefore the evaluation of ψ is irrelevant. Thus, in either case, the simplified formula is equivalent to the original one.

Note that the procedure Simplify runs in time polynomial in the size of ϕ: checking whether a rule applies, and actually applying the rule, can be done in polynomial time, and there are only a polynomially bounded number of applications of rules because each application decreases the size of the formula.

We now prove formally that EvalFormula gives an improvement over brute-force search for formula satisfiability. Before proving the result, we need a technical lemma, whose proof we omit. Intuitively, what the lemma says is that if there is a telescoping product of numbers between 0 and 1 where consecutive terms are close, then the product of a large random subset of terms is not likely to be much larger than the full product. We do need a strong concentration bound, i.e., "not likely" should mean the probability is exponentially small.

Lemma 5. *Let $\{a_i\}_{i=0}^{N-1}$, be a sequence, where $a_i = 1 - 1/(N-i)$. Then for any positive constant ϵ there is a positive constant $c < 1/2$, such that if S is a random subset of $[1 \ldots (1-c)N]$ of size at least $N/4$, the probability that $\Pi_{i \in S} a_i \geqslant \epsilon^{1/8}$ is at most $2^{-\Omega(N)}$ (where the Omega term depends on ϵ).*

We attempt to give some intuition for how the lemma will be used in the proof of the main result. The numbers a_i represent the superlinear component of the reduction in formula size at a reduction step. This expected decrease might not occur in a majority of reduction steps, but we will show that even if it occurs in a constant fraction, say one quarter, of reduction steps, the consequent decrease in formula size is substantial enough that we obtain significant savings over brute force search. We will separately give an exponentially small upper bound on the probability that the expected decrease does not occur often enough. Now for the details.

The following is a restatement of Theorem 1.

Theorem 6. *There is a deterministic algorithm for FormulaSAT which runs in time $2^{n-\Omega(n)}$ on formulae of length $O(n)$, where n is the number of variables in the formula.*

Proof:

We prove that the algorithm EvalFormula runs in time $2^{n-\Omega(n)}$ on formulae of size at most cn, where c is any constant. We will not try to optimize the savings over brute force search as a function of c, but our proof does imply a running time of $2^{n-n/c^k}$ for some constant k.

We define a notion of "computation tree" corresponding to the execution of EvalFormula on a formula ϕ. The computation tree is a binary tree whose internal nodes are labelled by pairs $< \psi, y >$ where ψ is a formula and y is the variable with the maximum number of occurrences in ψ. We call ψ the formula label of the node and y its variable label. The left child of a node labelled by $< \psi, y >$ has formula label $Simplify(\psi \mid_{y=0})$ and the right child has formula label $Simplify(\psi \mid_{y=1})$. The leaves of the computation tree are labelled with constants, i.e., 0 or 1.

We will assume that the computation tree is a complete binary tree of depth n by padding it - if there is a node at depth less than n which has already simplified to a constant, we label it by that constant and an arbitrary variable.

We will analyze the computation tree truncated at depth $d = (1-c')n$, where $c' < 1/2$ is a constant depending on c which we will fix later. We name nodes in this tree with binary strings in the obvious manner - the root is identified with the empty string and nodes at depth l are identified with strings of length l. Given a node v, let ϕ_v denote the formula label of v and x_v its variable label.

We define a notion of "goodness" for nodes in the truncated tree, and then extend that notion to paths in the tree. Fix a node v. If the label formula of v is a constant,

we label the left child "good" and the right child "bad". In this case, we call v a trivial node and any path from the root through v a trivial path. Otherwise, let r_1 be the number of times x_v occurs positively in ϕ_v as the child of an OR node plus the number of times it occurs negatively as the child of an AND node. Let r_0 be the number of times it occurs negatively as the child of an OR node plus the number of times it occurs positively as the child of an AND node . Since x_v is the most frequently occurring variable, we have that $r_0 + r_1 \geqslant |\phi_v|/(n - |v|)$. Here $|v|$ is the length of v as a string, and hence the depth of the node v in the tree.

If $r_0 \geqslant r_1$, we call the left child of v "good" and the right child "bad", otherwise we call the left child bad and the right child good. We now analyze how the sizes of the formulae ϕ_{v0} and ϕ_{v1} shrink relative to the size of ϕ_v. Note that ϕ_v has been simplified prior to EvalFormula being called on it. In particular, the variable simplication rules have been applied, and so we are guaranteed that for no leaf labelled with a literal does the literal or its complement occur in the sibling sub-tree of the leaf. Thus, in ϕ_v, for no gate is it the case that x_v or its complement is one child of the gate and the other child is a formula containing x_v or its complement. If $x_v \leftarrow 0$, the size of ϕ_v decreases by at least $2r_0 + r_1$ once the resulting formula has been simplified, and thus $|\phi_{v0}| \leqslant |\phi_v| - 2r_0 - r_1$. Similarly, $|\phi_{v1}| \leqslant |\phi_v| - 2r_1 - r_0$. This implies that for the good child of v, the decrease is at least $(r_0 + r_1)3/2 \geqslant 3|\phi_v|/(2(n - |v|))$. Thus, for a good child w of v, $|\phi_w| \leqslant |\phi_v|(1 - 3/(2(n - |v|)) \leqslant |\phi_v|(1 - 1/(n - |v|))^{3/2}$. Note that even for the bad child w' of v, $|\phi'_w| \leqslant |\phi_v|(1 - 1/(n - |v|))$.

Now, we call a path of length d starting from the root "good" if there are at least $n/4$ good nodes on the path and bad otherwise. Since each node has one good child and one bad child, the probability that a path is bad is at most $\Sigma_{i=1}^{n/4}\binom{d}{i}$, which by using Hoeffding's inequality for tail bounds on the binomial distribution, is at most $e^{-2(d/2 - n/4)^2/n}$. Since $d = (1 - c')n$ for $c' > 1/2$, this is $2^{-\Omega(n)}$.

Thus the probability that a path is bad is exponentially small. We will show that when a path is good, the probability that the formula label of the endpoint of the path at depth d has size at most $c'n/2$ is $1 - 2^{-\Omega(n)}$. To show this we apply Lemma 5. We've seen that irrespective of whether a nontrivial node v is good or bad, the formula size shrinks by at least a factor of $(1 - 1/(n - |v|))$ (unless the formula size is already down to 1). If a nontrivial node is good, there is an additional shrinkage factor of at least $\sqrt{(1 - 1/(n - |v|))}$. Applying Lemma 5, with the subset S interpreted as the set of depths of good nodes in a good path, for any constant $\epsilon > 0$, the probability that either the good path is trivial or the additional shrinkage factor is at least $\epsilon^{1/16}$ is at least $1 - 2^{-\Omega(n)}$. By setting $\epsilon = 1/(2c)^{16}$ and setting $c' = \epsilon/4$ as in the proof of Lemma 5, we get that with probability at least $1 - 2^{-\Omega(n)}$, the formula label at the end of a good path has size at most $cc'n/2c = c'n/2$, as desired.

The point is that we're saving over brute force search on all but an exponentially small fraction of good paths. Indeed, the cumulative time spent by the algorithm exploring good paths ending in a formula label of size at most $c'n/2$ and subtrees below those paths is at most $2^{1 - c'n + c'n/2} = 2^{n - \Omega(n)}$. The fraction of paths which are bad or are good but do not end in a small formula label is $2^{-\Omega(n)}$, and hence the total time spent by the algorithm on such paths is $2^{n - \Omega(n)}$. Thus the algorithm halts in time $2^{n - \Omega(n)}$ overall. ■

We note that by making a minor modification to Algorithm EvalFormula, we can exactly count the number of satisfying assignments within the same time bound.

Theorem 7. *There is a deterministic algorithm for #FormulaSAT which runs in time $2^{n - \Omega(n)}$ on formulae of length $O(n)$.*

Theorem 6 gives an algorithmic application of a lower bound method - namely, the restriction method used to prove lower bounds on formula size. Now, we turn this on its head and use our algorithmic method to give a strong *average-case* lower bound on the formula size of the Parity function. We are not aware of any such formula size lower bounds in the literature. A strong motivation for showing average-case lower bounds is as a step towards constructing pseudo-random generators. Pseudo-random generators with non-trivial seed size for formulae are unknown; our result raises optimism that such generators might be constructible with current techniques.

Our average-case lower bound goes via a characterization of formulae in terms of decision trees, which might be of independent interest.

Lemma 8. *For any constant $c > 0$, there is a constant $\delta > 0$ such that any function computed by a formula of size at most cn has decision trees of size at most $2^{(1-\delta)n}$*

We omit the proof - indeed Lemma 8 is implicit in the proof of Theorem 6.

The proofs of Theorems 6 and 7 actually show a constructive version of Lemma 8, where the decision tree can be constructed from the formula in time quasi-linear in the size of the decision tree. This constructive

version immediately yields Theorems 6 and 7 - given a decision tree, counting the number of inputs accepted by the decision tree (and hence deciding if there is an input it accepts) can be done easily in quasilinear time. However, storing the decision tree explicitly would mean usage of an exponential amount of memory - by evaluating the decision tree implicitly as in Algorithm EvalFormula, we keep the storage requirements down to polynomial.

A natural question is whether the bounds of Lemma 8 are tight. Could it be the case that all formulae of linear size have decision trees of size $2^{o(n)}$. The answer is no: consider the CNF formula $(x_1 \vee x_2 \vee x_3) \wedge (x_4 \vee x_5 \vee x_6) \ldots (x_{n-2} \vee x_{n-1} \vee x_n)$. It is easy to see that this formula requires DNFs of size $3^{n/3}$, and since every function with decision trees of size s has can be represented by DNFs with s terms, this also gives a lower bound on decision tree complexity. We suspect that the Ω term in the exponent of the decision tree size must tend to 1 in the worst case for formulae of size cn as c tends to ∞ - the methods of Miltersen, Radhakrishnan and Wegener [MRW05], who study possible blowups when converting CNFs to DNFs, might be helpful in establishing this.

The reason why a representation in terms of decision trees is useful is that decision tree complexity is easy to analyze for many functions. In particular, for the Parity function, we can get tight bounds not just on the worst-case decision tree complexity but also on the average-case complexity.

We begin by defining the notion of *advantage* of a decision tree in computing a function on a given set of inputs.

Definition 9. *Let f be a Boolean function on n bits, and T be a decision tree. Let $S \subseteq \{0,1\}^n$ be a set of inputs. The advantage of T in computing f on S is $|S|(Pr_{x \in S}[f(x) = T(x)] - Pr_{x \in S}[f(x) \neq T(x)])$.*

First we observe that decision trees of small size have low advantage in computing Parity on the Boolean cube.

Lemma 10. *No decision tree of size at most s has advantage more than s in computing Parity on the Boolean cube.*

Proof: let T be a decision tree of size s. Every leaf t of T corresponds to a subcube S_t of the Boolean cube - namely the subcube for which the variables along the path to t are set as specified in the tree and the other variables are free. The subcubes S_t are all disjoint, therefore the advantage of T in computing Parity over $\{0,1\}^n$ is the sum of the advantages of T in computing Parity over S_t, t a leaf of T. Now, if the leaf t is at depth less than n, T has zero advantage in computing Parity over S_t, since the number of inputs in S_t with even parity is equal to the number with odd parity. If t is at depth n, T has advantage at most 1 in computing Parity over S_t, since $|S_t| = 1$. Thus the total advantage of T over the Boolean cube is at most s. ■

Lemma 10 is close to tight - for each $s, 1 \leqslant s \leqslant 2^n$, there is a decision tree with at most $s+n$ leaves which has advantage s in computing Parity on $\{0,1\}^n$.

Lemma 10, combined with Theorem 8, gives a strong superlinear-size average-case lower bound for Parity over the uniform distribution on inputs of length n.

The following is a restatement of Theorem 2.

Theorem 11. *No family of linear-sized formulae has advantage $2^{n-o(n)}$ in computing Parity on $\{0,1\}^n$.*

Another way of stating Theorem 11 is to say that every family of linear-sized formulae must err on a $1/2 - 1/2^{\Omega(n)}$ fraction of inputs of length n, for each n. This is a strong average-case lower bound.

IV. The Algorithm for QBF Satisfiability

In the QBF satisfiability problem for formulae, we are given a fully quantified Boolean formula ϕ on n variables in prenex normal form, and asked whether ϕ is true. This problem is solvable by brute force in time $O(2^n|\phi|)$, since it reduces to evaluating a fully balanced formula with 2^n leaves, namely the evaluations of ϕ on all possible truth assignments to the n Boolean variables. Here we are interested in whether there is an algorithm for interesting restricted cases of this problem which runs in worst-case time $o(2^n)$.

The restricted case we consider is formulae for which there is an a priori constant bound k on the number of occurrences of any variable. Given such a k, we call this the QBF satisfiability problem for k-bounded formulae. Even the QBF satisfiability problem for 2-bounded QBFs is PSPACE-hard.

A first idea is to apply the restriction method used in the proof of Theorem 6. However, a key element of that method was our ability to choose the variable ordering when setting variables. We could do this because existential quantifiers commute with each other - $\exists x_1 \exists x_2 \phi(x_1, x_2)$ is equivalent to $\exists x_2 \exists x_1 \phi(x_1, x_2)$. In contrast, existential quantifiers do not commute with universal quantifiers - there is an ordering of variables imposed by the ordering of quantifiers in a QBF formula, and setting variables out of order does not seem to help in determining the truth of a QBF formula. If we so set variables in order, in the worst case, the formula might be most heavily dependent on variables which have high quantification depth. In this case, setting the

outer variables need not significantly reduce the size of a formula.

Because of this, we are unable to tackle general formulae, and restrict our attention to formulae with bounded number of variable occurrences. If the number of variable occurrences is bounded by a constant k, we are guaranteed *some* sort of reduction in the size of a formula irrespective of the order in which variables are set - when all but l variables are set, the resulting formula can have size at most kl. However, this is only a linear scaling of the reduced formula size with number of unset variables. In our algorithm for formula satisfiability, we relied critically on a sub-linear scaling to save on the number of assignments to search over and get an improved algorithm. So it's still unclear how to beat brute force in the present case.

We need a completely different idea, which we call the "bottleneck method". The idea is that when enough variables are set, there is a global "bottleneck" in the computation, meaning that the cumulative number of sub-problems that still need to be solved becomes sub-exponential. Instead of carrying out a brute force search, we implement a pre-processing step which computes solutions to all "small" sub-problems, and store all these solutions in a lookup table. Then, when we are exploring the tree of assignments to the QBF formula, we do not continue the exploration below a certain depth which corresponds to a "small enough" sub-problem, and instead look up the solution to the induced sub-problem in our pre-computed lookup table. This enables us to save over brute force in terms of time, though now our space usage does go up since we need to store a large lookup table. We note that similar ideas have been used several times before in the study of satisfiability, eg., in Williams' work [Wil02] on QBFSAT.

The following is a restatement of Theorem 3. We omit the formal proof, due to lack of space.

Theorem 12. *For any positive integer k, there is a deterministic algorithm solving the satisfiability problem for k-bounded QBFs in time $2^{n-\Omega(n/\log(n))}$.*

Using the same proof, but choosing $t(n)$ differently, we get an improvement over brute force search for a broader class of formulae.

Theorem 13. *There is a deterministic algorithm solving k-bounded TQBF in time $2^{n-\omega(1)}$ for any $k = o(n/\log^2(n))$.*

The algorithm of Theorem 12 is not guaranteed to save a constant factor in the exponent. We can save a constant factor in the exponent for instances that are "structured" in a certain precise sense. In many practical applications of QBF solving, instances are not adversarial but are structured in some way. We model "structured" in the broadest possible way, namely an instance is structured if it has a succinct description from which the instance can be recovered in polynomial time.

Definition 14. *A set $S \subseteq \{0,1\}^*$ is structured if there is a polynomial $p(n)$ such that for any $x \in S$, $C^p(x) = o(|x|)$.*

This definition of "structured" captures in a robust and uniform way a wide variety of notions of structuredness, for example a graph being planar or having bounded treewidth, or a matrix being sparse.

Now, by modifying the proof of Theorem 12 and performing the pre-processing only on instances that are not too unstructured, we can save a constant factor in the exponent. The key observation is that given a structured QBF formula ϕ, the formula obtained by substituting values for a constant fraction of variables of ϕ is not too unstructured.

The following is a restatement of Theorem 4. We omit the formal proof, due to lack of space.

Theorem 15. *For any integer k and structured set S of instances, there is a deterministic algorithm solving k-bounded TQBF which runs in time $2^{n-\Omega(n)}$ on S.*

The algorithms in the proofs of Theorem 12 and 15 are not practical - they use an exponential amount of memory. However, they are interesting from a couple of different perspectives. First, they give an indication that improvement over brute force search is *possible* even for interesting versions of QBFSAT - this has been known for a long while for SAT, but there's been precious little progress on QBFs. Second, Theorem 15 attempts to exploit the *structure* in instances, which is an important aspect of the success of SAT solvers in the real world. It does this by modelling "structure" in a very broad way, but perhaps similar algorithms would give useful results for narrower notions of structure which are present in real-world instances.

V. Acknowledgements

I'd like to thank Ryan Williams for interesting discussions, and for sending me a copy of his paper [Wil10].

References

[B.A61] B.A.Subbotovskaya. Realizations of linear functions by formulas using and, or, not. *Soviet Mathematics Doklady*, (2):110–112, 1961.

[CIP09] Chris Calabro, Russell Impagliazzo, and Ramamohan Paturi. The complexity of satisfiability of small depth circuits. In *Proceedings of 4th International Workshop on Parameterized and Exact Computation*, pages 75–85, 2009.

[CZ01] Hana Chockler and Uri Zwick. Which bases admit non-trivial shrinkage of formulae? *Computational Complexity*, 10(1):28–40, 2001.

[DGH02] Evgeny Dantsin, Andreas Goerdt, Edward Hirsch, Ravi Kannan, Jon Kleinberg, Christos Papadimitriou, Prabhakar Raghavan, and Uwe Schoning. A deterministic $(2 - 2/(k + 1))^n$ algorithm for k-sat based on local search. *Theoretical Computer Science*, 289(1):69–83, 2002.

[DH08] Evgeny Dantsin and Edward Hirsch. Worst-case upper bounds. In H.van Maaren A.Biere, M.Heule and T.Walsh, editors, *Handbook of Satisfiability*. 2008.

[DHIV01] Evgeny Dantsin, Edward Hirsch, Sergei Ivanov, and Maxim Vsemirnov. Algorithms for sat and upper bounds on their complexity. Technical report, Electronic Colloquium on Computational Complexity (ECCC), 8(11), 2001.

[GIB09] Alexandra Goultiaeva, Vicki Iverson, and Fahiem Bacchus. Beyond CNF: A circuit-based QBF solver. In *Theory and Applications of Satisfiability Testing (SAT 2009)*, pages 412–426, 2009.

[Has98] Johan Hastad. The shrinkage exponent of de morgan formulas is 2. *SIAM Journal on Computing*, 27(1):48–64, 1998.

[Heu99] Peter Heusch. The complexity of the falsifiability problem for pure implicational formulas. *Discrete Applied Mathematics*, 96:127–138, 1999.

[IP01] Russell Impagliazzo and Ramamohan Paturi. On the complexity of k-sat. *Journal of Computer and System Sciences*, 63(4):512–530, 2001.

[IPZ01] Russell Impagliazzo, Ramamohan Paturi, and Francis Zane. Which problems have strongly exponential complexity? *Journal of Computer and System Sciences*, 62(4):512–530, 2001.

[MRW05] Peter Bro Miltersen, Jaikumar Radhakrishnan, and Ingo Wegener. On converting cnf to dnf. *Theoretical Computer Science*, 347(1–2):325–335, 2005.

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

[MZ09] Sharad Malik and Lintao Zhang. Boolean satisfiability from theoretical hardness to practical success. *Communications of the ACM*, 52(8):76–82, 2009.

[NZ96] Noam Nisan and David Zuckerman. Randomness is linear in space. *Journal of Computer and System Sciences*, 52(1):43–52, 1996.

[Pap94] Christos Papadimitriou. *Computational Complexity*. Addison Wesley, 1994.

[PPSZ98] Ramamohan Paturi, Pavel Pudlak, Mike Saks, and Francis Zane. An improved exponential-time algorithm for k-sat. In *Proceedings of 39th International Symposium on Foundations of Computer Sciece (FOCS)*, pages 628–637, 1998.

[PPZ97] Ramamohan Paturi, Pavel Pudlak, and Francis Zane. Satisfiability coding lemma. In *Proceedings of 38th International Symposium on Foundations of Computer Science (FOCS)*, pages 566–574, 1997.

[Sch99] Uwe Schoning. A probabilistic algorithm for k-sat and constraint satisfaction problems. In *Proceedings of 40th Annual Symposium on Foundations of Computer Science*, pages 410–414, 1999.

[TBW04] Christian Thiffault, Fahiem Bacchus, and Toby Walsh. Solving non-clausal formulas with DPLL search. In *Tenth International Conference on Principles and Practice of Constraint Programming (CP 2004)*, pages 663–678, 2004.

[Tra84] Boris Trakhtenbrot. A survey of russian approaches to perebor (brute-force search) algorithms. *Annals of the History of Computing*, 6(4):384–399, 1984.

[vMS05] Dieter van Melkebeek and Rahul Santhanam. Holographic proofs and derandomization. *SIAM Journal on Computing*, 35(1):59–90, 2005.

[Wah05] Magnus Wahlstrom. An algorithm for the SAT problem for formulae of linear length. In *Proceedings of 13th Annual European Symposium on Algorithms*, pages 107–118, 2005.

[Wil02] Ryan Williams. Algorithms for quantified boolean formulas. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 299–307, 2002.

[Wil10] Ryan Williams. Improving exhaustive search implies superpolynomial lower bounds. In *Proceedings of the 42nd Annual ACM Symposium on Theory of Computing*, page To Appear, 2010.

[Woe01] Gerhard Woeginger. Exact algorithms for NP-hard problems: A survey. In *5th International Workshop on Combinatorial Optimization*, pages 185–208, 2001.

[Woe08] Gerhard Woeginger. Open problems around exact algorithms. *Discrete Applied Mathematics*, 156(3):397–405, 2008.

[Zha06] Lintao Zhang. Solving QBF by combining conjunctive and disjunctive normal forms. In *Proceedings of 21st National Conference on Artificial Intelligence and 18th Innovative Applications of Artificial Intelligence Conference (AAAI 2006)*, 2006.