Electronic Colloquium on Computational Complexity, Report No. 27 (2013)

A Framework for Proving Proof Complexity Lower Bounds on Random CNFs Using Encoding Techniques

Luke Friedman

December 7, 2010

Abstract

Propositional proof complexity is an area of complexity theory that addresses the question of whether the class NP is closed under complement, and also provides a theoretical framework for studying practical applications such as SAT solving. Some of the most well-studied contradictions are random k-CNF formulas where each clause of the formula is chosen uniformly at random from all possible clauses. If the clause-to-variable ratio is chosen appropriately, then with high probability such a formula will be unsatisfiable, and the lack of structure in these formulas makes them natural candidates for proving lower bounds on the size of refutations in strong systems such as Frege systems where basically no nontrivial lower bounds are known.

In this work we prove exponential lower bounds on treelike resolution refutations of random 3-CNFs using new combinatorial techniques. The basic idea is to show that any formula with a short refutation can be compressed in a way that a random formula cannot be by algorithmically constructing such a short encoding. It is important to emphasize that the actual results in the work are not new; exponential lower bounds for random k-CNFs are already known for treelike resolution, and in fact for much stronger systems such as general resolution and the RES(k) system. However, it is our hope that these new techniques generalize in a way that the known combinatorial techniques based on size-width tradeoffs do not, and that we can use these encoding techniques to attack stronger systems such as constant depth Frege systems where the old techniques are not applicable.

1 Introduction

Propositional proof complexity is a sub-area of complexity theory concerned with the following type of question: given a proof system P and a propositional tautology σ, what is the shortest proof of σ in P? Here a propositional tautology is a formula made up of a finite set of boolean variables and logical connectives such that regardless of how the variables are set to TRUE or FALSE, the formula always evaluates to TRUE. For our purposes we can restrict attention to tautologies presented in DNF form.

A propositional proof system can be formally defined as a polynomial time function P such that

$$\forall \sigma \in \mathcal{T} \; \exists y[P(\sigma, y) = 1]$$
$$\forall \tau \notin \mathcal{T} \; \forall y[P(\tau, y) = 0]$$

ISSN 1433-8092

where $\mathcal{T}$ is the set of all propositional tautologies encoded as strings. For $\sigma \in \mathcal{T}$, we think of y as a proof that σ is a tautology. This definition captures the intuitive features of a proof system: the ability to prove anything that is true, the inability to prove anything that is false, and that verifying whether a proof is correct should be an efficient process.

Note that there is no restriction on the size of a proof y, only that checking whether y is correct can be done in polynomial time. If a proof system P has the additional feature that for any $\sigma \in T$, there exists a correct proof y of σ such that $|y| \leq p(|\sigma|)$ for some fixed polynomial p, we say that the proof system P is polynomially bounded. Cook and Reckhow introduced this abstract notion of a proof system, and they also established the following fundamental connection between propositional proof complexity and the standard computational complexity.

Theorem 1.1 *[CR79]* NP = coNP *if and only if there exists a polynomially bounded proof system.*

It is generally conjectured that NP $\neq$ coNP, in which case for every proof system P there exists certain families of tautologies for which the shortest proofs in P must be of superpolynomial size. This suggests a natural program: attempt to show that stronger and stronger proof systems P are not polynomially bounded as progress towards showing that NP $\neq$ coNP.

This program has practical implications as well, due to the importance of the satisfiability problem and its relationship to propositional tautologies. (A formula σ is a tautology if and only if its negation is unsatisfiable). Because many important optimization problems can be efficiently encoded as satisfiability instances, an enormous amount of research has been invested into developing SAT solvers that outperform the trivial exponential time brute force algorithm. Proof complexity lower bounds can be used to lower bound the runtime of certain classes of these solvers. For instance, many of the SAT solvers used in practice are variants of the DPLL algorithm [DLL62, DP60], a simple branching algorithm based on setting a variable to either TRUE or FALSE and then recursively considering the simplified formulas that result. It has been shown that from the execution of a DPLL based SAT solver on an unsatisfiable formula σ, one can derive a resolution refutation of σ that has size proportional to the runtime of the solver. This holds even for DPPL based solvers that use sophisticated heuristics to choose the order on which to branch on variables and "clause learning" to do more efficient backtracking [BKS04]. Thus, any resolution lower bounds immediately imply lower bounds for the runtime of these algorithms as well. By identifying a resolution refutation of σ as a proof of the tautology $\neg\sigma$, we can think of resolution as a proof system in the above framework. Resolution is one of the simplest and well-studied proof systems; there are exponential lower bounds on the size of resolution refutations of many classes of unsatisfiable formulas, which proves that any DPLL based SAT solver will have exponential runtime on these same classes of formulas. This is a general principle; anytime we derive lower bounds for a propositional proof system on a class of tautologies, as a byproduct we get lower bounds on the runtime of some class of co-nondeterministic SAT solving algorithms on the negation of those same instances.

In this paper we focus on randomly generated 3-CNF formulas, where each clause is independently and uniformly chosen at random from all possible clauses. If the clause to variable ratio is slightly above the satisfiability threshold, then with high probability such a random formula will be unsatisfiable. One main reason why random formulas are of interest is that they are candidates to be hard to refute for any proof system. No proof system is known to have polynomial size refutations of these instances, and intuitively they seem to be hard due to their lack of structure. For strong proof systems such as Frege systems, which have rules for manipulating arbitrary formulas

instead of just clauses, no superlinear lower bounds of any kind are known and there are few hard candidates, so random formulas are a natural choice for attacking such systems.

However, the lack of structure that makes random formulas potentially useful for proving lower bounds can be an impediment as well. Historically, lower bounds on random formulas have come only after lower bounds on explicit formulas have already been proven. For instance, the first resolution lower bounds were for classes of formulas capturing the idea of "counting" such as Tseitin formulas and propositional encodings of the pigeonhole principle [Tse68, Hak85]. Later, Szémeredi and Chvátal were able to adapt some of these techniques to work for random formulas as well [CS88]. In the case of constant depth Frege systems, which are restricted versions of Frege systems whose lines are constant depth formulas, we have exponential lower bounds for certain explicit families of formulas like the pigeonhole principle [Ajt94, PBI93], but no superpolynomial bounds are known for random formulas.

Another reason for focusing on random formulas in proof complexity is that they do not have an analogue in circuit complexity. Progress on propositional proof complexity has run somewhat parallel to circuit complexity, usually with breakthroughs in circuit complexity lower bounds coming first and informing new lower bounds for proof complexity. For instance, the combinatorial lower bounds for constant depth Frege systems rely on random restrictions and a switching lemma adapted from the AC^0 lower bounds of [FSS84, Has86]. Sometimes the connection to circuit complexity is more direct; in the case of the cutting planes system, an algebraic proof system that manipulates linear inequalities, the only known superpolynomial lower bounds for any class of formulas rely on a reduction to the monotone circuit lower bounds of Razborov [Raz85]. Presently, progress in propositional proof complexity lags slightly behind circuit complexity in the sense that we have some lower bounds for constant depth circuits with mod gates [Raz87, Smo87, Wil11], but no lower bounds for restrictions of Frege systems that work with such circuits. Proving circuit lower bounds is notoriously difficult, and a number of barriers such as relativization, natural proofs, and algebrization have been identified in that field [BGS75, RR97, AW08]. If a major breakthrough in proof complexity is to occur anytime soon, such as superpolynomial lower bounds for Frege systems or the even stronger Extended Frege systems, it is likely that there will have to be a departure from the paradigm of adapting circuit complexity techniques.

Random formulas are intriguing in this respect because they highlight a major difference between proof complexity and circuit complexity. In circuit complexity we are not interested in lower bounds on the circuit size of arbitrary functions, as a trivial counting argument shows that a randomly chosen function will have high circuit complexity. Instead, we try to prove lower bounds for *explicit* functions (usually defined as functions in some complexity class such as NP). However, it is not clear how to formulate the notion of a random function in a class like NP in a useful way. On the other hand, no such trivial counting argument shows that a random formula must have large proof complexity. This opens up the possibility of employing probabilistic methods and other nonconstructive techniques to prove lower bounds for random formulas that perhaps cannot be used in the circuit complexity setting.

In this paper we adopt this approach and prove treelike resolution lower bounds for random 3-CNF formulas using new encoding techniques. The high level idea is simple; we show that given a short treelike resolution refutation of an unsatisfiable formula, the description of the formula can be compressed more than one can compress a random formula, and therefore random formulas cannot have short refutations. In essence we are using a counting argument that is specifically tailored to work with random formulas.

The results themselves are not particularly interesting; treelike resolution is a weak variant of the resolution system and our bounds are weaker than those proven previously. (For the clause density we choose the best known bounds are of the form $2^{\Omega(n)}$, and we derive bounds of 2^{n^ϵ} for some ϵ). It is our hope however that the techniques introduced generalize better than current methods and will be useful for attacking stronger proof systems for which no good methods currently exist. Wigderson and Ben-Sasson showed that most of the known resolution lower bounds, including those for random formulas, could be explained in terms of size-width tradeoffs [BSW01]. In this framework, one shows that any short refutation can be transformed to another refutation where the width of all clauses in the refutation (defined as the number of literals in the clause) is small. Then one derives lower bounds on the size of refuting certain formulas by showing that any such refutation must contain a clause of large width. This approach is very elegant, but is hard to generalize since the notion of width is specific to clauses and does not apply to proof systems that have greater expressive power than resolution. The basic framework we establish in this paper can be applied to any proof system, and the lower bounds we derive for treelike resolution do not directly refer to clause width at all.

One other way in which the methods here seem to be qualitatively different than previous methods is that they are algorithmic in nature. At the most abstract level, in order to establish that random formulas do not have short refutations, one must identify a property Q that random formulas have but that formulas with short refutations do not have. The previous resolution lower bounds use static properties Q based on the idea of expansion. Given a CNF formula, we can form a bipartite graph where one set of vertices are all the clauses of the formula and the other set of vertices are all the variables of the formula, and there is an edge between a clause and a variable if the variable appears in the clause. For a random formula the resulting bipartite graph will have some type of explictly statable expansion properties that formulas with short resolution refutations provably cannot have. On the other hand, in our lower bound proof the property Q is intrisically tied to the main coding algorithm of the proof. Our Q also is capturing some idea related to expansion, but it seems impossible to extract it from the main algorithm and to state it explicitly as a static property. Perhaps the ability to use these implicit properties will be useful in dealing with more powerful proof systems.

2 Random Formulas and the General Framework

In this section we define random formulas and prove two simple lemmas that establish our general framework applicable to any proof system.

Definition 2.1 A boolean variable is a variable that takes only the values TRUE or FALSE; we will associate the value 1 with TRUE and the value 0 with FALSE. For a boolean variable x, the ***positive literal*** $\boldsymbol{x^1}$ evaluates to 1 if and only if $x = 1$ and the ***negative literal*** $\boldsymbol{x^0}$ evaluates to 1 if and only if $x = 0$. A ***clause*** C is a disjunction of literals, i.e $C = x_{i_1}^{j_1} \vee \cdots \vee x_{i_r}^{j_r}$, where each $j_l \in \{0, 1\}$ and each x_{i_l} is a boolean variable. We say that a clause C is ***simple*** if no variable appears twice in C, and we say that two simple clauses C_1 and C_2 are equal if they contain the same literals. $\boldsymbol{lits(C)}$ denotes the set of literals in C, and $\boldsymbol{vars(C)}$ denotes the set of variables in C, ignoring the sign of the literal. A ***k-CNF formula*** φ is a conjunction of clauses, i.e. $\varphi = C_1 \wedge \cdots \wedge C_r$, where each C_i is a clause that contains at most k literals. For a k-CNF formula φ, the ***size*** of φ, denoted by $|\varphi|$, is the number of clauses in φ. We write $\boldsymbol{\phi \subseteq \varphi}$ if ϕ is a subset of the clauses of φ.

Definition 2.2 Let $x_1, \ldots x_n$ be a finite set of boolean variables. We define a ***random k-CNF formula with n variables and clause density Δ*** to be a formula chosen uniformly at random from all k-CNF formulas $\mathcal{F}$ such that $|\mathcal{F}| = \Delta n$, each clause of $\mathcal{F}$ is simple and contains exactly k literals, and no two clauses of $\mathcal{F}$ are equal. [1]

It is known that there exists a constant $\Delta^* \leq 4.596$ such that for large n, a random 3-CNF formula with n variables and clause density $\Delta > \Delta^*$ will be unsatisfiable with high probability [JSV00].

Our first lemma establishes a simple counting method for proving that random formulas do not have short refutations in a proof system.

Lemma 2.3 *Let $\mathcal{P}$ be an arbitrary proof system, and c be a constant such that $c > \Delta^*$. Let Φ be the set of all 3-CNF formulas with at most n variables and exactly cn distinct clauses such that each clause is simple and contains exactly three literals. Let $\Psi \subseteq \Phi$ be the set of all such formulas that are unsatisfiable and have refutations of size at most $t(n)$ in $\mathcal{P}$. Let $\{0,1\}^{\leq k}$ be the set of all binary strings of length at most k. If there exists an onto function $\eta : \{0,1\}^{\leq k} \to \Psi$ with*

$$k = \log \left(\binom{8\binom{n}{3}}{cn} \right) - \omega(1)$$

then with high probability a random 3-CNF formula $\mathcal{F}$ with n variables and clause density c will be unsatisfiable and have no $\mathcal{P}$ refutation of size at most $t(n)$. [2]

Proof: Because $c > \Delta^*$, with high probability $\mathcal{F}$ will be unsatisfiable.

We have that

$$\log |\Phi| = \log \left(\binom{8\binom{n}{3}}{cn} \right)$$

Therefore, given the assumption, $|\Psi| \ll |\Phi|$, so with high probability $\mathcal{F}$ will be unsatisfiable and have no $\mathcal{P}$ refutation of size at most $t(n)$.

∎

Lemma 2.3 shows that if we can "compress" any formula ψ that has a short refutation, then we can get lower bounds on the size of refutations of random formulas. Our next lemma says that we can focus our attention on finding some subset of the clauses of ψ that are compressible. Suppose we encode a 3-CNF formula with cn clauses by individually encoding each clause (in other words, writing down each of the three variables in the clause and the signs of the literals). Then altogether the encoding would be of size about $3cn \log n$, since writing down a variable takes about $\log n$ bits. Of course, by considering the clauses altogether instead of individually, we can save on

[1] This final condition, that no two clauses are equal, is not usually included in the definition of a random formula. However, for large n and constant clause density, if we omit this last condition with high probability a random formula will contain distinct clauses, and explicitly including this condition as part of our definition will help with what follows.

[2] Again, we are identifying a refutation of an unsatisfiable formula σ with a proof of the tautology $\neg\sigma$, so these lower bounds on the size of $\mathcal{P}$ refutations of unsatisfiable CNF formulas are equivalently lower bounds on the the size of $\mathcal{P}$ proofs of tautological DNF formulas

our encoding; as it turns out it is possible to encode an arbitary 3-CNF formula with cn clauses using about $2cn \log n$ bits, which is about $2 \log n$ bits per clause. The following lemma says that if given a formula ψ with a short refutation we are always able to find some subset of $m > 0$ clauses of ψ that can be encoded using $(2 - \epsilon)m \log n$ bits, then this suffices to get the lower bounds of Lemma 2.3.

Lemma 2.4 *Let $\mathcal{P}, c, n, \Phi, \Psi, \mathcal{F}$ be as in Lemma 2.3.*

Let Φ_m be the set of all sets of m clauses.

Suppose there exists a constant $\epsilon > 0$ and a family of functions $\{\rho_m\}, 1 \leq m \leq cn$, such that

1. *$\rho_m : \{0,1\}^{(2-\epsilon)m \log n} \to \Phi_m$*
2. *For all $\psi \in \Psi$ there exists $\varphi \subseteq \psi$ and ρ_m such that φ is in the range of ρ_m*

Then with high probability $\mathcal{F}$ will have no $\mathcal{P}$ refutation of size at most $t(n)$.

Proof: By Lemma 2.3, it suffices to construct an onto function $\eta : \{0,1\}^{\leq k} \to \Psi$ with

$$k = \log \left(\binom{8\binom{n}{3}}{cn} \right) - \omega(1)$$

Let $\psi \in \Psi$. By assumption there exists $1 \leq m \leq cn$ and $\varphi \subseteq \psi$ such that $\rho_m(x) = \varphi$ for some $x \in \{0,1\}^{(2-\epsilon)m \log n}$. Let ψ^- be the clauses of ψ that are not in φ. The total number of 3-CNF formulas with at most n variables and exactly $cn - m$ distinct clauses such that each clause is simple and contains exactly three literals is

$$\binom{8\binom{n}{3}}{cn - m}$$

Therefore ψ^- can be uniquely represented by a binary string y of size

$$\log \left(\binom{8\binom{n}{3}}{cn - m} \right)$$

We will define $\eta(x \circ y) = \psi$, where $\circ$ is the concatenation function. η is well defined; as it turns out, the larger m is the shorter the string $x \circ y$ will be, so the length of the string $x \circ y$ contains the information about where the delimitation between x and y occurs.

Clearly η is onto, so all that remains is to show that

$$|x \circ y| \leq \log \left(\binom{8\binom{n}{3}}{cn} \right) - \omega(1)$$

We have that

$$\begin{aligned} |x \circ y| &= \log \left(\binom{8\binom{n}{3}}{cn - m} \right) + (2 - \epsilon)m \log n \\ &= \log \left(\binom{8\binom{n}{3}}{cn - m} \right) + 2m \log n - \epsilon m \log n \end{aligned}$$

Therefore, it suffices to show that

$$\log\left(\binom{8\binom{n}{3}}{cn}\right) - \log\left(\binom{8\binom{n}{3}}{cn-m}\right) \geq 2m\log n - o(m\log n)$$

Let $(n)_k$ denote the falling factorial. We have that

$$\begin{aligned}
&\log\left(\binom{8\binom{n}{3}}{cn}\right) - \log\left(\binom{8\binom{n}{3}}{cn-m}\right) \\
&= \log\left(\frac{((4/3)(n)_3)!}{(cn)!((4/3)(n)_3 - cn)!}\right) - \log\left(\frac{((4/3)(n)_3)!}{(cn-m)!((4/3)(n)_3 - cn + m)!}\right) \\
&= \log((cn-m)!) + \log(((4/3)(n)_3 - cn + m)!) - \log((cn)!) - \log(((4/3)(n)_3 - cn)!) \\
&= -\sum_{i=cn-m+1}^{cn} \log i + \sum_{i=(4/3)(n)_3-cn+1}^{(4/3)(n)_3-cn+m} \log i \\
&\geq -m\log(cn) + m\log((4/3)(n)_3 - cn + 1) \\
&= -m\log n + 3m\log n - o(m\log n) \\
&= 2m\log n - o(m\log n)
\end{aligned}$$

∎

3 Resolution and the DPLL system

Resolution is one of the simplest and well-studied proof systems; even so, proving resolution lower bounds has been a difficult task.

A resolution refutation of an unsatisfiable CNF formula φ is a sequence of clauses $\mathcal{C} = C_1, C_2, \ldots C_t$, where $C_t = \emptyset$, the empty clause, and each C_i is either a clause from φ or is derived from two clauses C_j, C_k, with $j, k < i$, using the following resolution rule

$$\frac{A \vee x^1, B \vee x^0}{A \vee B}$$

Here $C_j = A \vee x^1$, $C_k = B \vee x^0$, $C_i = A \vee B$, A and B are arbitrary sets of literals, and x is an arbitrary boolean variable. Because the resolution rule is sound and the empty clause is trivially unsatisfiable, such a sequence of clauses is a proof that φ is unsatisfiable. One can also show that resolution is complete; any unsatisfiable 3-CNF formula has a resolution refutation of size at most 2^n, where the size of a resolution refutation is defined to be the number of clauses in the sequence.

It is natural to represent a resolution refutation π of a formula φ as a rooted directed acyclic graph π with nodes in π corresponding to the clauses in $\mathcal{C}$. In this representation, the root of π is the clause $\emptyset$, the leaves of π are clauses of φ, and there are edges going from C_j and C_k into C_i if C_i was derived from C_j and C_k. (The graph π cannot necessarily be uniquely determined from $\mathcal{C}$, since it is possible that a clause from $\mathcal{C}$ can be legally derived in multiple ways). If there exists a π corresponding to $\mathcal{C}$ such that the outdegree of every node in π other than the root is one, then

we say that $\mathcal{C}$ is a ***treelike*** refutation. In some cases the shortest treelike resolution refutation of a formula φ can be exponentially larger than the shortest general resolution refutation of φ [BSIW00].

Tseitsin proved the first superpolynomial resolution lower bounds for a restricted form of resolution called regular resolution (a system strictly more powerful than treelike resolution) in the late 1960's, using a class of contradictions based on the fact that the sums of degrees of an undirected graph must be even [Tse68]. Almost two decades passed before Haken proved the first superpolynomial bounds for general resolution, using a class of contradictions based on the pigeonhole principle [Hak85]. Soon after, Szémeredi and Chvátal proved the first resolution lower bounds for random CNFs using techniques adapted from Haken's proof [CS88]. Later, Ben-Sasson and Wigderson unified all of these previous results under the size-width tradeoff paradigm [BSW01]. The best lower bounds for random 3-CNFs with n variables and clause density $c > \Delta^*$ are of the form $2^{\Omega(n)}$, which is tight up to the multiplicative factor in the exponent. There also exist exponential lower bounds for random formulas when the clause density is non-constant, and for the proof system RES(k), which is a generalization of resolution whose lines are k-DNFs instead of just clauses [BSW01, Ale05]. (A clause is a 1-DNF and resolution is equivalent to RES(1). RES(k) for constant $k > 1$ is strictly stronger than resolution, and strictly weaker than constant depth Frege systems).

We now describe the DPLL proof system.

Definition 3.1 A DPLL refutation π of an unsatisfiable 3-CNF formula φ is a binary rooted tree. Each interior node of π is labeled with a variable x, and the two edges entering such a node are respectively labeled with x^0 and x^1. For a given node a in π, we associate with a a set $\rho(a)$ that consists of the literals labeling the path from the root to a. Every leaf node b is labeled with a clause from φ whose literals are a subset of $\rho(b)$. [3] The size of π is the number of nodes in π.

We will make a couple of assumptions about the structure of a DPPL refutation.

Definition 3.2 Let π be a DPLL refutation of a formula φ. For a node a in π, let $\boldsymbol{tree(a)}$ be the subtree of π with root a.

Suppose an interior node a is labeled with the variable x. Then $\boldsymbol{child_1(a)}$ denotes the node in π connected to a by an edge labeled with x^1, and $\boldsymbol{child_0(a)}$ denotes the node in π connected to a by an edge labeled with x^0. Also, $\boldsymbol{tree_1(a)}$ denotes $tree(child_1(a))$, the subtree rooted at $child_1(a)$, and $\boldsymbol{tree_0(a)}$ denotes $tree(child_0(a))$, the subtree rooted at $child_0(a)$.

We say that π is in ***normal form*** if the following conditions are satisfied:

1. Along any path from the root of π to a leaf node, no two nodes are labeled with the same variable

2. Let a be an interior node in π labeled by a variable x. Then there exists some leaf node in $tree_1(a)$ labeled by a clause containing the literal x^1, and there exists some leaf node in $tree_0(a)$ labeled by a clause containing the literal x^0.

Lemma 3.3 *If there exists a DPLL refutation of a formula φ of size s, then there exists a normal form DPLL refutation of φ of size at most s.*

[3] In the usual setup a leaf node b is labeled with a clause that only contains literals whose negations are in $\rho(b)$. This corresponds more closely to the DPLL algorithm, where one branches on variables and backtracks after finding a clause that is falsified by the current restriction of the variables. Our definition is equivalent and will ease notation.

Proof: Let π be a DPLL refutation of a formula φ that is not in normal form. Suppose that condition 1 is violated, so that along some path from the root to a leaf node there are two nodes a and b, each labeled with a variable x. Without loss of generality, assume that a is above b in π, and that b is in $tree_1(a)$. Then if we modify π by replacing the subtree $tree(b)$ with the subtree $tree_1(b)$, π will still be a valid DPLL refutation of φ, and the size of π will decrease. We can iterate this process until condition 1 is satisfied.

Now suppose that condition 2 is violated. Let a be an interior node in π labeled by a variable x, and without loss of generality suppose that there is no leaf node in $tree_1(a)$ that contains the literal x^1. Then if we modify π by replacing the subtree $tree(a)$ with the subtree $tree_1(a)$, π will still be a valid DPLL refutation of φ and the size of π will decrease. We can iterate this process until condition 2 is satisfied. ∎

Our interest in the DPLL system stems from the following connection to treelike resolution. (See [Kra95] for instance)

Theorem 3.4 *An unsatisfiable k-CNF formula φ has a DPLL refutation of size s if and only if it has a treelike resolution refutation of size s.*

Proof sketch: Given a graph π corresponding to a treelike refutation $\mathcal{C}$ of φ, we can create a normal form DPLL refutation π' of φ with the exact same node and edge structure as π. The leaf nodes of π' will be labeled the same as in π. Suppose an interior node a of π has edges entering it from nodes b and c, and that a, b, and c correspond to the clauses C_a, C_b, and C_c in $\mathcal{C}$ respectively. Furthermore, suppose that in $\mathcal{C}$ C_a is derived from C_b and C_c by resolving on the variable x, and that x^1 is in C_b and x^0 is in C_c. Then in π', a will be labeled with the variable x, the edge going from b to a will be labeled with x^1 and the edge going from c to a will be labeled with x^0. If we form π' this way it will be a valid normal form DPPL refutation of φ.

Similarly, if π' is a normal form DPLL refutation of φ, we can create a treelike resolution refutation $\mathcal{C}$ of φ such that $|\mathcal{C}| = |\pi'|$. We will build $\mathcal{C}$ inductively using the tree π'. For each leaf node in π' labeled with a clause C, we add C to $\mathcal{C}$. Suppose an interior node a in π' has edges going into it from nodes b and c, and that C_b and C_c are the clauses already in $\mathcal{C}$ corresponding to the nodes b and c. Furthermore, suppose a is labeled with the variable x. Then we add C_a to $\mathcal{C}$ where C_a is the clause derived from C_b and C_c by resolving on x. If we form $\mathcal{C}$ this way, it will be a valid treelike resolution refutation that has a corresponding graph π with the same node and edge structure as π'.
∎

4 Statement of the Main Theorem and the Coding Algorithm

We are now ready to state our main theorem.

Theorem 4.1 *Let $\mathcal{F}$ be a random 3-CNF with n variables and clause density $c > \Delta^*$ for some constant c. There exists a δ such that with high probability $\mathcal{F}$ does not have a DPLL refutation of size at most 2^{n^δ}.*

Corollary 4.2 *Let $\mathcal{F}$ be a random 3-CNF with n variables and clause density $c > \Delta^*$ for some constant c. There exists a δ such that with high probability $\mathcal{F}$ does not have a treelike resolution refutation of size at most 2^{n^δ}.*

Proof of Theorem 4.1: By Lemma 2.4, we can reduce the proof to the following problem. We are given some unsatisfiable 3-CNF formula φ that has at most n variables and exactly cn distinct clauses where each clause is simple and contains exactly 3 literals. Furthermore, we are guaranteed that φ has a DPLL refutation of size at most 2^{n^δ}. We must show that we can find some subset of m clauses of φ, for $1 \leq m \leq cn$, such that we can encode these m clauses using at most $(2-\epsilon)m \log n$ bits for some $\epsilon > 0$. Our code must be independent of φ in the sense that we must use the same code for any such input to our problem.

Let π be the lexicographically first normal form DPLL refutation of φ of size at most 2^{n^δ}. We have no time constraints, so finding π is not an issue.

Our solution will be to design a coding algorithm ENCODE that traverses the tree π and always outputs a transcript *code* that is of size at most $(2-\epsilon)m \log n$ bits and encodes $m > 0$ clauses of φ. ENCODE will work by repeating the following steps:

ENCODE(π)
1 Get next clause C
2 Encode C
3 Make cache deletion decisions
4 Check halting condition

Let us first informally discuss the basic ideas behind ENCODE.

As part of the algorithm, each time ENCODE reaches line 1 it must produce a new clause to encode. In order to do this, ENCODE will start at the root and perform a depth-first traversal of π. Whenever there is a choice of whether to move to the left child or the right child, the traversal will go in the direction of the subtree of smaller size, breaking ties arbitrarily. When the traversal reaches a leaf node labeled by a clause C, if C has not yet been encoded previously during the algorithm ENCODE will return this clause C (and continue the depth-first traversal from this point the next time it must get a clause). Otherwise, ENCODE backtracks and continues the depth-first traversal until a clause C' is found that has not yet been encoded previously.

Once a clause C has been found that has not previously been encoded, ENCODE must encode C in line 2. Throughout its execution, ENCODE maintains a transcript *code* that will be the output of the algorithm. ENCODE also maintains a "cache" of variables that starts off empty and will be used to encode clauses. The following three types of operations are recorded in *code*:

1. Adding a new variable to the cache
2. Encoding a clause C of φ.
3. Deleting a variable from the cache

In order for ENCODE to encode the clause C, it first must ensure that all the variables in $var(C)$ are in the cache. Let $X \subseteq var(C)$ be the variables that are not currently in the cache.

ENCODE adds each $x \in X$ to the cache one at a time and records this in *code*. To record entering a variable x in the cache, ENCODE must write $\log n$ bits to *code* to describe x. [4]

Once all the variable in $var(C)$ are in the cache, ENCODE encodes the clause C by indexing the three variables in $var(C)$ in the cache and using three extra bits to specify whether each variable appears as a positive or negative literal in C. To do this ENCODE must write $3 \log |cache| + 3$ bits to *code*, where $|cache|$ is the current size of the cache.

Next, in step 3, ENCODE makes a decision about whether or not to delete some variables from the cache. If ENCODE decides to delete a variable x from the cache, it does this and then records this operation in *code* by writing $\log |cache|$ bits to *code* specifying x.

Finally, in step 4, ENCODE decides whether or not to halt. If it halts then the algorithm ends and ENCODE outputs *code*. Otherwise, the algorithm continues again at step 1.

Note that at any time during the execution of ENCODE, the current state of *code* implicitly defines *cache*, the variables that are in the cache at the current time, and *clauses*, the set of clauses in φ that have already been encoded.

Let us now give some intuition behind how ENCODE can compress the description of a set of clauses of φ. Suppose ENCODE only halts after it has traversed the entire tree π. Also, suppose we are able to ensure that the size of *cache* stays small throughout the execution of ENCODE (let us say $|cache| \leq O(\log n)$), and we can ensure that whenever a variable x is entered into the cache, it is accessed twice during an encode operation before it is deleted from the cache.

Then, since every time a clause is encoded exactly three variables are accessed from the cache, we get that the number of times a variable is added to the cache is at most $(3/2)m$, where m is the number of clauses that are encoded. Also, because the cache never grows to be bigger than $O(\log n)$, recording all the encode and deletion operations to *code* will together take at most $O(m \log \log n) = o(m \log n)$ bits. Therefore, including the add variable operations, when ENCODE halts and outputs *code* it will have length at most $(3/2)m \log n + o \log(n)$ bits, which is at most $(2 - \epsilon)m \log n$ bits for an appropriately chosen ϵ. The assumptions we make here are unrealistic, but this will be the intuition underlying how we define the cache deletion decisions and halting condition of ENCODE.

In order to describe further details of ENCODE we will need some more definitions.

Definition 4.3 Let a be an interior node of π. If during the depth-first traversal of ENCODE a has not yet been reached we say that a is ***unreached***. If ENCODE is in the process of traversing either $tree_0(a)$ or $tree_1(a)$ but has not yet traversed the other subtree, we say that a is ***in phase 1***. If ENCODE is in the process of traversing either $tree_0(a)$ or $tree_1(a)$ and has already finished traversing the other subtree, we say that a is ***in phase 2***. If ENCODE has finished traversing all of $tree(a)$, then we say that a is ***finished***.

Let a be an interior node of π labeled by a variable x. Then $\boldsymbol{edge_1(a)}$ denotes the edge labeled by x^1 entering a, $\boldsymbol{edge_0(a)}$ denotes the edge labeled by x^0 entering a, and we say that $edge_1(a)$ and $edge_0(a)$ are ***partner*** edges.

Let $leaf$ be a leaf node in π labeled by a clause C. Suppose $x^i \in lits(C)$. Then along the path from the root of π to $leaf$ there must be some node a labeled with the variable x. If at some point during the execution of ENCODE the depth-first traversal reaches $leaf$ and causes ENCODE to encode C, we say that $edge_i(a)$ was ***covered*** by C, and from that point on we say that the edge

[4] Also, whenever an operation is recorded in *code* the type of operation must be specified. But this will only take a constant number of extra bits per clause encoded and will not affect our analysis

$edge_i(a)$ has been ***covered***. For some node b, if $leaf$ is in $tree(b)$, we say that $edge_i(a)$ was ***covered during the traversal of tree(b)***. (Note that this does not imply that $edge_i(a)$ is in $tree(b)$.)

As a first try, let us define the cache deletion decisions of ENCODE as follows. Suppose ENCODE has just encoded a clause C in step 2 containing the literals x^i, y^j, z^k that labels a leaf node $leaf$ in π. In step 3 ENCODE now has to decide which if any variables to delete from the cache. The only variables that ENCODE will consider deleting are the variables x, y, and z. To decide whether to delete each of the variables, ENCODE will consider them one at time, using the following protocol.

Suppose ENCODE is deciding whether to delete the variable x. Because $x^i \in lits(C)$, there must be a node a on the path from the root to $leaf$ that is labeled by the variable x. Because ENCODE just encoded the clause C, the depth-first traversal is in the process of traversing $tree_i(a)$, so a is in phase 1 or in phase 2. If a is in phase 2 then delete x from the cache, and if a is in phase 1 then do not delete x from the cache.

The logic behind such a protocol is as follows. We know that because π is in normal form (see Definition 3.2), there is some leaf node $leaf_1$ in $tree_1(a)$ labeled by a clause C_1 containing the literal x^1. Similarly, there is some leaf node $leaf_0$ in $tree_0(a)$ labeled by a different clause C_0 containing the literal x^0. The hope is that if we wait to delete x from the cache until after both C_0 and C_1 have been encoded, then we can ensure that every time a variable is added to the cache it is accessed twice during an encode operation before it is deleted from the cache like in our hypothetical situation from before.

The problem with this strategy is that although we have the guarantee that $leaf_0$ and $leaf_1$ exist in $tree(a)$, we do *not* have the guarantee that when ENCODE begins traversing $tree(a)$ both C_0 and C_1 have not been encoded previously during the algorithm. Indeed, it may be the case using this cache deletion decision protocol that x is entered into the cache while a is in phase 1 but no clause containing x is ever encoded while a is in phase 2 or vice versa, in which case x may only be accessed once for an encode operation before it is deleted from the cache. However, in order for this to be the case, it must be that x was entered into the cache previously during the algorithm and then removed. This suggests a strategy for dealing with these bad situations: if we can predict when such a situation will occur, we should *not* delete x from the cache the first time around, so that it will still be waiting there when we need to access it again.

This motivates our second try at defining the cache deletion decisions of ENCODE. This time the protocol will be more complex. As part of the cache deletion decisions ENCODE will color edges of π in order to keep track of information that will be needed to make optimal decisions (and which will be used in our analysis of the algorithm). ENCODE will also maintain a counter *count* that starts at 0 and will be used to define a new halting condition.

The coloring is based on a case analysis and may seem arbitrary; it will be helpful to remember that the purpose of the coloring is to keep track of those situations where a variable may be entered into the cache and only accessed once for an encode operation before being deleted from the cache. In general, the colors have the following meaning:

- All edges start off colored ***black***. If an edge is black, it means that it has not yet been covered during ENCODE.
- When an edge is covered during ENCODE, it is colored ***green*** if its partner edge has not been covered yet and would never be covered during ENCODE, even if ENCODE was allowed

to traverse all of π (i.e. ENCODE did not stop in the middle of the traversal because of a halting condition)

- When an edge is covered during ENCODE, it is colored ***white*** if it has not been covered previously, and its partner edge has already been covered or would be covered during ENCODE at some point if ENCODE was allowed to traverse all of π.
- When an edge is covered during ENCODE, if it is then colored ***blue*** it means the edge had already been covered previously during ENCODE.

Every time an edge is colored (or recolored) either green or blue, ENCODE will increment *count*. The new halting condition will be to halt if $count \geq 12n^{\delta}$, or if ENCODE finishes traversing all of π, whichever comes first. (This halting condition prevents the cache from getting too large, as we will see later in the analysis).

We now give the new cache deletion decisions protocol in full detail. Again the setup is as follows: we are assuming that ENCODE has just encoded a clause C containing the literal x^i that labels a leaf node $leaf$. ENCODE is now deciding whether or not to delete the variable x from the cache. We know there exists a node a on the path from the root of π to $leaf$ labeled with the variable x, and that a is either in phase 1 or phase 2. The edge $edge_i(a)$ has just been covered.

We will break up the protocol into two cases, depending on whether a is in phase 1 or phase 2.

First suppose a is in phase 1. We now consider the following subcases, which are listed in priority ordering since they are not completely disjoint. (For instance, both subcase 1 and subcase 2 could occur, in which case ENCODE follows the rules of subcase 1).

1. Let $\mathcal{C}$ be the set of all clauses labeling a leaf node in $tree_{1-i}(a)$. Let $\mathcal{C}' \subseteq \mathcal{C}$ be the set of all such clauses that have not yet been encoded during ENCODE. Suppose $\forall D \in \mathcal{C}'$, $x \notin var(D)$. This means that when ENCODE later traverses $tree_{1-i}(a)$, it will not encode any clause containing x. (Note that every clause labeling a leaf node in $tree_i(a)$ that contains x contains the literal x^i, and every clause labeling a leaf node in $tree_{1-i}(a)$ that contains x contains the literal x^{1-i}, so for the rest of the time a is in phase 1 whether or not $\mathcal{C}'$ contains a clause with the variable x will not change.) In this case ENCODE colors (or recolors) $edge_i(a)$ green, increments *count*, and removes x from the cache.
2. $edge_i(a)$ is black. ENCODE colors $edge_i(a)$ white and leaves x in the cache.
3. $edge_i(a)$ is not black. ENCODE colors (or recolors) $edge_i(a)$ blue, increments *count*, and leaves x in the cache.

Now suppose a is in phase 2. We consider the following subcases (this time the subcases are disjoint). Note that in all these subcases x is always removed from the cache.

1. $edge_{1-i}(a)$ is black. Since ENCODE has already traversed all of $tree_{1-i}(a)$, this means that $edge_{1-i}(a)$ will never be covered during ENCODE. In this case ENCODE colors (or recolors) $edge_i(a)$ green, increments *count*, and removes x from the cache.
2. $edge_i(a)$ is black and $edge_{i-1}(a)$ is not black. ENCODE colors $edge_i(a)$ white and removes x from the cache.

3. $edge_i(a)$ is not black and $edge_{1-a}(a)$ is not black. ENCODE colors (or recolors) $edge_i(a)$ blue, increments *count*, and removes x from the cache.

We are not quite done yet. ENCODE now is keeping track of the information it needs, but it is not using this information to make optimal cache deletion decisions yet.

The actual coding algorithm we will use will be the following algorithm ENCODE′. ENCODE′ first runs ENCODE as we have defined it above, but throws away its output. Let ENCODE1 denote this first execution of ENCODE. Let ***colored*** be the set of edges in π that are colored either green or blue during ENCODE1. Let ***var-colored*** be the set of variables that label one of the edges in *colored* (ignoring the sign of the literal).

ENCODE′ then runs ENCODE again. Let ENCODE2 denote this second execution of ENCODE. The traversal, coloring, and halting condition will be the same as in ENCODE1. Therefore ENCODE2 will halt at the same point that ENCODE1 does. The only difference between ENCODE2 and ENCODE1 is that ENCODE2 has the following modification to the cache deletion decisions protocol of ENCODE.

Suppose that ENCODE2 has just encoded a clause containing the literal x^i and is deciding whether to delete x from the cache. If $x \in$ *var-colored*, and later on in ENCODE2 an edge from *colored* labeled with x (ignoring the sign of the literal) will be colored (or recolored) green or blue, then ENCODE2 leaves x in the cache *regardless* of what the cache deletion decisions protocol calls for. This is well-defined, since the coloring is the same in ENCODE1 and ENCODE2 and they halt at the same time and we have already run ENCODE1. Otherwise ENCODE2 makes cache deletion decisions according to the protocol of ENCODE. The output of ENCODE′ is the output of ENCODE2.

5 Analysis of the Algorithm

We continue the proof of Theorem 4.1 by analyzing ENCODE′. Our goal is to show that for some $\epsilon > 0$ the output of ENCODE2, *code*, has length at most $(2-\epsilon)m \log n$ bits, where m is the number of clauses encoded in *code*. (Note that from *code* we can reconstruct every clause that is encoded in *code*.) First we state a combinatorial lemma that we will need later on.

Let T be a rooted binary tree, where each edge is colored black or white. We call T properly colored if for every interior node a of the tree, the two edges entering a are the same color.

Consider the following one-player coloring game: T originally starts off colored all black, except for two edges lying on some path P from the root of T to a leaf node in T that are colored white and are called the ***initial edges*** of the game. On each turn the player is allowed to color three black edges in T white provided they all lie on the same path from the root of T to a leaf node in T. The player wins if he is able to properly color T.

Lemma 5.1 *It is impossible to win the coloring game*

We defer the proof of this lemma to the Appendix.

The main challenge will be in proving a lemma that states that the cache never grows to be too large during ENCODE1. From there it will be easy to show that the cache never grows to be too large during ENCODE2, which will be the key to proving that the output of ENCODE′ is sufficiently short. We now give some definitions and prove a few smaller lemmas leading up to that main lemma.

Definition 5.2 Let us break an execution of ENCODE into discrete time steps. We say that ENCODE is at time $t = 0$ at the beginning of the algorithm, and every time ENCODE performs an action such as taking a step in the depth-first traversal, adding a variable to the cache, encoding a clause, or deleting a variable from the cache, we increment t by 1. If ENCODE is at time t of its execution, then $\boldsymbol{node(t)}$ denotes the last node that was visited during the depth-first traversal of ENCODE, $\boldsymbol{\rho_t}$ denotes the set of nodes along the path from the root of π to $node(t)$, and $\boldsymbol{cache(t)}$ denotes the set of variables that are in the cache at time t. Also $\boldsymbol{var(\rho_t)}$ denotes the set of variables that label some node in ρ_t.

For a node $a \in \rho_t$ such that $a \neq node(t)$, we define $\boldsymbol{child_{\rho_t}(a)}$ to be the child node of a that is in ρ_t, $\boldsymbol{edge_{\rho_t}(a)}$ to be the edge connecting a and $child_{\rho_t}(a)$, and $\boldsymbol{tree_{\rho_t}(a)}$ to be $tree(child_{\rho_t}(a))$. Similarly, we define $\boldsymbol{child_{\bar{\rho}_t}(a)}$ to be the child node of a that is not in ρ_t, $\boldsymbol{edge_{\bar{\rho}_t}(a)}$ to be the edge connecting a and $child_{\bar{\rho}_t}$ and $\boldsymbol{tree_{\bar{\rho}_t}(a)}$ to be $tree(child_{\bar{\rho}_t}(a))$.

Let $\boldsymbol{phase1(t)} \subseteq \rho_t$ be the set of nodes in ρ_t that are in phase 1 at time t, and $\boldsymbol{phase2(t)} \subseteq \rho_t$ be the set of nodes in ρ_t that are in phase 2 at time t. (All nodes in ρ_t will either be in phase 1 or phase 2 at time t.)

Note that the following three lemmas refer to ENCODE1, the first execution of ENCODE by ENCODE$'$ whose output is ignored.

Lemma 5.3 *At time t of ENCODE1, $|phase1(t)| \leq n^\delta$.*

Proof: Suppose for contradiction there are more than n^δ nodes in ρ_t in phase 1. Consider a node $a \in \rho_t$ that is in phase 1. Then, because the depth-first traversal always moves toward the subtree of smaller size, $size(tree_{\rho_t}(a)) \leq size(tree_{\bar{\rho}_t}(a))$. Therefore, because there are more than n^δ nodes in ρ_t in phase 1, π must have size greater than 2^{n^δ}, which contradicts the fact that π is supposed to have size at most 2^{n^δ}. ∎

Lemma 5.4 *Suppose ENCODE1 is at time t of its execution. Then*

$$y \in cache(t) \Rightarrow y \in var(\rho_t)$$

Proof: Let y be a variable such that $y \notin var(\rho_t)$. Suppose y is added to the cache at some time $t' < t$ in order for ENCODE1 to encode some clause C containing the literal y^i that labels a leaf node $leaf$. Then there exists a node a on the path from the root of π to $leaf$ that is labeled by y, and $a \notin \rho_t$ since $y \notin var(\rho_t)$. Examining the cache deletion decision protocol of ENCODE1, we see that if y is not deleted from the cache immediately after C is encoded, it is because a is in phase 1 and there is another clause C' containing the literal y^{1-i} labeling some leaf node $leaf'$ in $tree_{1-i}(a)$ that had not yet been encoded by time t'. (Remember that in ENCODE1 we use the the cache deletion decisions protocol of ENCODE without the extra modifications of ENCODE2). Because $a \notin \rho_t$, ENCODE1 will have traversed all of $tree_{1-i}(a)$ by time t. So at some time t'' such that $t' < t'' < t$, ENCODE1 will encode C'. At this time a will be in phase 2, so, according to the cache deletion decisions of ENCODE1, y will immediately afterwards be removed from the cache. This proves that if $y \notin var(\rho_t)$, then $y \notin cache(t)$ ∎

Definition 5.5 For an interior node b in π, let $\boldsymbol{var(b)}$ be the variable labeling b and $\boldsymbol{edges(b)}$ be the set of edges that lie on the path from the root of π to b.

Lemma 5.6 *Suppose ENCODE1 is at time t of its execution, and let $b \in phase2(t)$. Suppose $edge_{\bar{\rho}_t}(b)$ is covered at some point during the traversal of $tree_{\bar{\rho}_t}(b)$ by a clause C, and neither $edge_{\bar{\rho}_t}(b)$ nor any edge in $tree_{\bar{\rho}_t}(b)$ is colored green or blue during the traversal of $tree_{\bar{\rho}_t}(b)$. Then some edge in $edges(b)$ is covered during the traversal of $tree_{\bar{\rho}_t}(b)$.*

Proof:

Suppose for contradiction that $edge_{\bar{\rho}_t}(b)$ is covered at some point during the traversal of $tree_{\bar{\rho}_t}(b)$, that neither $edge_{\bar{\rho}_t}(b)$ nor any edge in $tree_{\bar{\rho}_t}(b)$ is colored green or blue during the traversal of $tree_{\bar{\rho}_t}(b)$, and that no edge in $edges(b)$ is covered during the traversal of $tree_{\bar{\rho}_t}(b)$.

Define an instance of the coloring game from Lemma 5.1 as follows. $tree_{\bar{\rho}_t}(b)$ will be the tree T of the coloring game. The other two edges that are covered by C during the traversal of $tree_{\bar{\rho}_t}(b)$ are the initial edges of the game. Since by assumption no edge in $edges(b)$ is covered during the traversal of $tree_{\bar{\rho}_t}(b)$, they will both be in T. Also, because any edges covered by C must lie on a single path, they will both lie on a single path as required by the definition of the coloring game.

Let the player's moves be defined by the clauses that are encoded during the traversal of $tree_{\bar{\rho}_t}(b)$ other than C. Because $b \in phase2(t)$, by time t ENCODE1 has already traversed all of $tree_{\bar{\rho}_t}(b)$. For every clause C' that is encoded, the player plays a move where he colors white the three edges of T that correspond to the three edges covered by C'. Note that every time the player colors three edges white they lie on a single path, and by the assumption that no edge in $edges(b)$ is covered during the traversal of $tree_{\bar{\rho}_t}(b)$ and that $edge_{\bar{\rho}_t}(b)$ is not colored green or blue, they also all lie within T.

Also, no edge in $tree_{\bar{\rho}_t}(b)$ is ever colored blue, so the player does not ever illegally color the same edge twice. Furthermore, the player wins this coloring game; if there were a bad node a in T after the player was done making all his moves, then, according to the cache deletion decisions protocol of ENCODE1, either $edge_0(a)$ or $edge_1(a)$ would have to have been colored green during ENCODE1, which contradicts the assumption that no edge in $tree_{\bar{\rho}_t}(b)$ is colored green.

But this is a contradiction, since by Lemma 5.1 it is impossible to win the coloring game.

∎

We are now ready to prove our main lemma.

Lemma 5.7 *During an execution of ENCODE1 the cache never contains more than $14n^\delta$ variables*

Proof: Suppose ENCODE1 is at time t of its execution.

The high level strategy for proving Lemma 5.7 is as follows. By Lemma 5.4 a variable can only be in the cache at time t if it is $var(\rho_t)$. We will define a set of nodes that we will call ***good***, and establish a one-to-one mapping η from good nodes to elements of $var(\rho_t)$ that cannot be in the cache at time t. Then, by lower bounding the number of good nodes, we also lower bound the number of elements of $var(\rho_t)$ that cannot be in the cache at time t, and thus we get an upperbound on the number of variables that can be in the cache at time t.

As part of this process we will need to define another one-to-one mapping ν from instances where an edge has been colored (or recolored) green or blue in ENCODE1 to nodes in π.

Now we give the details. Let $b \in phase2(t)$. We will consider a number of disjoint cases.

1. Suppose that $var(b)$ has not ever been entered into the cache by ENCODE1 during the traversal of $tree(b)$. In this case, we say b is a good node and define $\eta(b) = var(b)$.

2. Suppose that during the traversal of $tree_{\bar{\rho}_t}(b)$ an edge $edge_{\rho_t}(a) \in edges(b)$ was covered for some $a \in phase2(t)$ at time $t' < t$. Furthermore, suppose $edge_{\rho_t}(a)$ was black before it was covered. Then, because $a \in phase2(t)$, according to the cache deletion decisions protocol of ENCODE1, $var(a)$ will be removed from the cache, and it cannot be in the cache at time t. We will therefore call b a good node and define $\eta(b) = var(a)$. Note that once an edge is covered it will be colored something other than black and cannot ever become black again, and also that for two nodes $b, b' \in \rho_t$, the trees $tree_{\bar{\rho}_t}(b)$ and $tree_{\bar{\rho}_t}(b')$ are disjoint. Therefore η can only map a single node to $var(a)$, so the property that η is one-to-one is maintained.

   Suppose that in addition when $edge_{\rho_t}(a)$ was covered, it was colored from black to green. Then we will say that a is ***responsible for coloring an edge*** and define $\nu(edge_{\rho_t}(a), t') = a$. This is an important point and is part of why these definitions are so involved; notice that in this case both an edge is colored either green or blue *and* a variable is popped out of the cache. The calculations in our analysis will be delicate enough that we must take special care to associate these instances with two different nodes through η and ν, or else the math would not go through.

3. Suppose that during the traversal of $tree_{\bar{\rho}_t}(b)$ an edge $edge$ is covered at time $t' < t$, where $edge$ is either in $tree_{\bar{\rho}_t}(b)$ or $edge = edge_{\bar{\rho}_t}(b)$. Furthermore, suppose that after $edge$ is covered it is colored either green or blue. Then we say that b is responsible for coloring an edge and define $\nu(edge, t') = b$.

4. Suppose that during the traversal of $tree_{\bar{\rho}_t}(b)$ an edge $edge \in edges(b)$ was covered for some time $t' < t$. Furthermore, suppose $edge$ was not black before it was covered (i.e. it had been covered previously), so that after it was covered it was recolored either green or blue. Then we say that b is responsible for coloring an edge and define $\nu(edge, t') = b$.

5. Suppose that during the traversal of $tree_{\bar{\rho}_t}(b)$ an edge $edge_{\rho_t}(a) \in edges(b)$ was covered for some $a \in phase1(t)$ at time $t' < t$. Furthermore, suppose $edge_{\rho_t}(a)$ was black before it was covered, and then colored white immediately afterwards as part of the cache deletion decisions. Note that in this case, because a is in phase 1, the variable $var(a)$ will *not* be removed from the cache according to the cache deletion decisions protocol of ENCODE1. In this case we say that b is ***responsible for covering a phase 1 edge***.

Note that as we have defined η and ν, they are both one-to-one mappings.

Now let us attempt to count the number of good nodes. Let $b \in phase2(t)$. Suppose that b is not responsible for coloring an edge and is not responsible for covering a phase 1 edge.

If $var(b)$ has not ever been entered into the cache during the traversal of $tree_{\bar{\rho}_t}(b)$, then by item 1 above b is a good node. Otherwise the edge $edge_{\bar{\rho}_t}(b)$ must have been covered during the traversal of $tree_{\bar{\rho}_t}(b)$. To see this, note that if $edge_{\bar{\rho}_t}(b)$ were not covered during the traversal of $tree_{\bar{\rho}_t}(b)$, then $edge_{\rho_t}(b)$ would have to have been covered during the traversal of $tree_{\rho_t}(b)$ in order for $var(b)$ to be in the cache. But then, according to the cache deletion decisions protocol of ENCODE1, $edge_{\rho_t}(b)$ would be colored green, and so by item 2 above, b is responsible for coloring an edge, which contradicts our assumption.

Also, neither $edge_{\bar{\rho}_t}(b)$ nor any edge in $tree_{\bar{\rho}_t}(b)$ was colored green or blue during the traversal of $tree_{\bar{\rho}_t}(b)$, otherwise by item 3 above b would be responsible for coloring an edge. Therefore, by Lemma 5.6, some $edge \in edges(b)$ was covered during the traversal of $tree_{\bar{\rho}_t}(b)$. $edge$ must

have been black before it was covered, or else by item 4 above b would have been responsible for coloring an edge. Then we know that $edge = edge_{\rho_t}(a)$ for some $a \in phase2(t)$, otherwise b would be responsible for covering a phase 1 edge, which contradicts our assumption. Therefore, by item 2 above, b is a good node in this case as well.

We have that the number of nodes that are responsible for coloring an edge can be at most $12n^\delta$, due to the halting condition of ENCODE. Also, by Lemma 5.3 we have that $|phase1(t)| \leq n^\delta$, so there can be at most n^δ nodes responsible for covering a phase 1 edge.

Therefore we get the following lower bound on the number of good nodes

$$\begin{aligned}(\#\text{ of good nodes }) &\geq |phase2(t)| - 12n^\delta - n^\delta \\ &= (|\rho_t| - |phase1(t)|) - 13n^\delta \\ &\geq |\rho_t| - 14n^\delta\end{aligned}$$

By Lemma 5.4 a variable can only be in the cache at time t if it is in ρ_t. Because η is a one-to-one mapping between good nodes and elements of ρ_t that cannot be in the cache at time t, we get that

$$\begin{aligned}|cache(t)| &\leq |\rho_t| - (|\rho_t| - 14n^\delta) \\ &= 14n^\delta\end{aligned}$$

∎

Corollary 5.8 *During an execution of ENCODE2 the cache never contains more than* $26n^\delta$ *variables*

Proof: The only difference between ENCODE1 and ENCODE2 is that variables from *var-colored* are potentially kept in the cache in ENCODE2 when they would be removed in ENCODE1. We have that $|var\text{-}colored| \leq |colored| \leq 12n^\delta$ due to the halting condition of ENCODE. Therefore the cache at time t in ENCODE2 can contain at most $12n^\delta$ more elements than the cache at time t in ENCODE1. ∎

Corollary 5.9 *Let endcache be the set of variables in the cache when ENCODE2 halts. Then* $|endcache| \leq 14n^\delta$.

Proof: The cache deletion decision protocol of ENCODE2 is such that if ENCODE2 halts at time t, its cache at time t is the same as the cache of ENCODE1 at time t. ∎

Suppose ENCODE2 has halted and outputted *code*. We are ready to bound $|code|$, the length of *code* in bits. Let $\mathcal{T} = t_1, t_2, \ldots t_l$ be a list of the instances during ENCODE2 in which a variable was written to the cache, ordered chronologically. Let $var(t_i)$ be the variable that was added to the cache at time t_i. Note that it is possible that $var(t_i) = var(t_j)$ for some $i \neq j$. Let $num(t_i)$ be

the number of times the variable $var(t_i)$ was accessed from the cache for an encode operation after time t_i before being removed from the cache (including the time $var(t_i)$ is accessed immediately after being added to the cache at time t_i) Let *num-colored* be the number of times that an edge is colored (or recolored) green or blue during ENCODE2.

Lemma 5.10

$$l \leq \frac{\sum_{i=1}^{l} num(t_i) - num\text{-}colored + |endcache|}{2}$$

Proof:

Suppose at time t of its execution ENCODE2 had just encoded a clause C containing the literal x^i that labels a leaf node $leaf$. Then there exists a node a on the path from the root of π to $leaf$ that is labeled with the variable x. After encoding C, during the cache deletion decisions when ENCODE2 decided whether to delete x from the cache, ENCODE2 colored the edge $edge_i(a)$ some color.

Suppose first that ENCODE2 colored $edge_i(a)$ blue. This means that $x \in var\text{-}colored$ and $edge_i(a)$ had been covered previously, so some clause $C' \neq C$ containing x^i had been encoded at some time $t' < t$. Then, according to the cache deletion decisions protocol of ENCODE2, when C' was encoded the variable x must have been left in the cache and would still be there at time t when C was encoded.

Similarly, suppose ENCODE2 colored $edge_i(a)$ green. Again, this implies that $x \in var\text{-}colored$. Also, because $edge_i(a)$ is colored green it means that at the time ENCODE2 began traversing $tree(a)$, $tree_{1-i}(a)$ did not contain any leaf node labeled with a clause C' containing the literal x^{1-i} that had not already been encoded. But because π is in normal form (see Definition 3.2), there must exist some leaf node in $tree_{1-i}(a)$ labeled with a clause C' containing the literal x^{1-i} that had already been encoded previously. Then, according to the cache deletion decisions protocol of ENCODE2, when C' was encoded the variable x was left in the the cache and would still be there at time t when C was encoded.

Therefore, if $edge_i(a)$ was colored green or blue, $num(t_i)$ was incremented for some t_i without adding any new element to $\mathcal{T}$.

Now suppose that $edge_i(a)$ was colored from black to white, but that $x \notin endcache$. Then there exists a clause C' labeling a leaf node $leaf$ in $tree_{1-i}(a)$ that contains the literal x^{1-a} and had not been encoded by time t (let $leaf$ be the first node labeled by such a C' that ENCODE2 would reach during the traversal of $tree_{1-i}$). It may be that in order to encode the clause C ENCODE2 had to add x to the cache, in which case a new t_j would have been added to the list $\mathcal{T}$. But according to the cache deletion decision protocol of ENCODE2, x would have remained in the cache until C' was encoded, and since $x \notin endcache$, it must be that C' was encoded before ENCODE2 halted. Therefore $num(t_j) \geq 2$.

Putting all this information together, we get that

$$\sum_{i=1}^{l} num(t_i) \geq 2l + num\text{-}colored - |endcache|$$

from which the claim follows. ∎

Let m be the total number of clauses encoded in *code*. Then $m = \sum_{i=1}^{l} num(t_i)/3$. The total length of *code* is the cost of documenting every addition and deletion of a variable to/from the cache, plus the cost of encoding the m clauses by indexing three elements of the cache per clause, plus a constant number of bits per clause that is encoded. There can be at most $3m$ variables added to the cache during ENCODE2, so altogether there can be at most $3m$ total deletion operations recorded in code. By Corollary 5.8, the cache never grows to be larger than $26n^{\delta}$ during ENCODE2, so altogether these deletion operations contribute at most $3\delta m \log n + O(m)$ bits to $|code|$. Similarly, the encode operations will together contribute at most $3\delta m \log n + O(m)$ bits to $|code|$ as well. Therefore in total, the deletion and encode operations will contribute at most $6\delta m \log n + O(m)$ bits to $|code|$.

Each time a variable is added to the cache it takes $\log n$ bits to document this, so the total number of bits needed to document all the cache insertions is $l \log n$. Therefore we have that $|code| \leq l \log n + 6\delta m \log n + O(m)$.

Now suppose that ENCODE traverses all of π before halting. In this case $|endcache| = 0$. Using Lemma 5.10,

$$\begin{aligned}
|code| &\leq l \log n + 6\delta m \log n + O(m) \\
&\leq \left(\frac{\sum_{i=1}^{l} num(t_i) - num\text{-}colored + |endcache|}{2} \right) \log n + 6\delta m \log n + O(m) \\
&\leq \frac{3}{2} m \log n + 6\delta m \log n + O(m) \\
&\leq \left(\frac{3 + 12\delta}{2} \right) m \log n + O(m)
\end{aligned}$$

Otherwise suppose that ENCODE halts before traversing all of π. In this case we have that $num\text{-}colored = 12n^{\delta}$ and, by Corollary 5.9, $|endcache| \leq 14n^{\delta}$. Therefore, using Lemma 5.10 and the fact that $m \geq num\text{-}colored/3 = 4n^{\delta}$ we get that

$$\begin{aligned}
|code| &\leq l \log n + 6\delta m \log n + O(m) \\
&\leq \left(\frac{\sum_{i=1}^{l} num(t_i) - num\text{-}colored + |endcache|}{2} \right) \log n + 6\delta m \log n + O(m) \\
&\leq \left(\frac{3m + 2n^{\delta}}{2} \right) \log n + 6\delta m \log n + O(m) \\
&\leq \left(\frac{7m}{4} \right) \log n + 6\delta m \log n + O(m)
\end{aligned}$$

Therefore,

$$|code| \leq max \left(\left(\frac{3 + 12\delta}{2} \right) m \log n + O(m), \left(\frac{7m}{4} \right) \log n + 6\delta m \log n + O(m) \right)$$

Choosing δ small enough we get that $|code| \leq (15/8)m \log n$, which is $(2 - \epsilon)m \log n$ for $\epsilon = 1/8$. ∎

6 Open Questions

The main open question is whether these methods can be extended to prove lower bounds for stronger systems than treelike resolution. Certainly the first step is to try to prove general resolution lower bounds. Although exponential lower bounds for general resolution refutations of random CNFs already exist, this would be an important step towards showing the validity of these new methods. The reason why the proof in this paper does not extend immediately to the general resolution case is that it relies crucially on the fact that every time ENCODE takes a step in its depth-first traversal, because it always moves towards the smaller subtree, it is "cutting off" at least half of the proof tree. This property is used to prove Lemma 5.3, which is subsequently used to prove the main lemma, Lemma 5.7, which says that the cache never grows to be too large. In the general resolution case, if we take a step in a depth-first traversal, it is possible that the majority of the proof graph is still reachable regardless of which child node we choose to go to, so there is no way to "cut off" at least half of the proof graph as before.

Also, the proof of Theorem 4.1 is messy at times, particularly in the description of the cache deletion decisions protocol of ENCODE and the proof of Lemma 5.7. Is it possible to simplify the coding algorithm or its analysis in the treelike resolution case?

References

[Ajt94] M. Ajtai. The complexity of the pigeonhole principle. *Combinatorics 14*, 4:417–433, 1994.

[Ale05] M. Alekhnovich. Lower bounds for k-DNF resolution on random 3CNF. In *Proceedings of the 37th Symposium on Theory of Computing*, pages 251–256, 2005.

[AW08] S. Aaronson and A Wigderson. Algebrization: A new barrier in complexity theory. In *Proc. ACM Symp. on Theory of Computing (STOC)*, pages 731–740. ACM, 2008.

[BGS75] T. Baker, J. Gill, and R. Solovay. Relativizations of the P =? NP question. *SIAM J. Comput.*, 4(4):431–442, 1975.

[BKS04] P. Beame, H. Kautz, and A. Sabharwal. Towards understanding and harnessing the potential of clause learning. *Journal of Artificial Intelligence Research*, 22:319–351, 2004.

[BSIW00] E. Ben-Sasson, R. Impagliazzo, and A. Wigderson. Near optimal separation of treelike and general resolution. In *Proceedings of SAT-2000: Third Workshop on the Satisfiability Problem*, pages 14–18, 2000.

[BSW01] E. Ben-Sasson and A. Wigderson. Short proofs are narrow — resolution made simple. *Journal of the ACM*, 48(2):149–169, 2001.

[CR79] S.A. Cook and R. Reckhow. The relative efficiency of propositional proof systems. *Journal of Symbolic Logic*, 44:36–50, 1979.

[CS88] V. Chvátal and E. Szémeredi. Many hard examples for resolution. *Journal of the ACM*, 35(4):759–768, 1988.

[DLL62] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 7:394–397, 1962.

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

[FSS84] M. Furst, J. Saxe, and M. Sipser. Parity, circuits, and the polynomial time hierarchy. *Mathematical Systems Theory*, 17:13–27, 1984.

[Hak85] A. Haken. The intractability of resolution. *Theoretical Computer Science*, 39:297–305, 1985.

[Has86] J. Hastad. Almost optimal lower bounds for small depth circuits. In *Proc. ACM Symp. on Theory of Computing (STOC)*, pages 6–20. ACM, 1986.

[JSV00] S. Janson, Y. C. Stamatiou, and M. Vamvakari. Bounding the unsatisfiability threshold of random 3-sat. *Random Structures and Algorithms*, 18:79–102, 2000.

[Kra95] J. Krajíček. *Bounded arithmetic, propositional logic and complexity theory.* Cambridge University Press, 1995.

[PBI93] T. Pitassi, P. Beame, and R. Impagliazzo. Exponential lower bounds for the pigeonhole principle. *Computational Complexity 3*, 2:97–140, 1993.

[Raz85] A. A. Razborov. Lower bounds on the monotone complexity of some boolean functions. *Dokl. Akad. Nauk. SSSR*, 281(4):798–801, 1985.

[Raz87] A. A. Razborov. Lower bounds on the size of bounded depth networks over a complete basis with logical addition. *Mathematical Notes of the Academy*, 41(4):598–607, 1987.

[RR97] A. A. Razborov and S. Rudich. Natural proofs. *J. Comput. Syst. Sci*, 55(1):24–35, 1997.

[Smo87] R. Smolensky. Algebraic methods in the theory of lower bounds for Boolean circuit complexity. In *Proc. ACM Symp. on Theory of Computing (STOC)*, pages 77–82. ACM, 1987.

[Tse68] G.S. Tseitin. On the complexity of derivations in the propositional calculus. In *Studies in Constructive Mathematics and Mathematical Logic*, Part II, pages 115–125. Consultants Bureau, New York and London, 1968.

[Wil11] R. Williams. Non-uniform ACC circuit lower bounds. In *Proc. IEEE Conf. on Computational Complexity*, 2011.

7 Appendix

In this appendix, we present the proof of Lemma 5.1.

Proof of Lemma 5.1: Call an interior node a in T bad if the two edges entering a are different colors. A proper coloring implies that there are no bad nodes in T. Let W be the set of black edges entering a bad node (every bad node has exactly one black edge entering it). We can assume without loss of generality that on each turn the player colors white some edge in W. To see this, let *edge* be an edge that is in W before move i. In order for the player to win, on some move $j \geq i$, *edge* must be colored white. Suppose the player wins, and let σ be the winning sequence of moves that the player plays. If the order of moves in σ is transposed in any way, it will still be a winning sequence of moves. Therefore we can exchange move j with move i, so that on move i *edge* is colored white.

We claim the following two invariants are maintained throughout the game:

1. Let $\{edge1, edge2\} \subseteq W$. Then there does not exist a path in T from the root to a leaf node that contains both $edge1$ and $edge2$.

2. For two edges $edge1$ and $edge2$, let $anc(edge1, edge2)$ denote the least ancestor of $edge1$ and $edge2$ (i.e. the farthest node from the root such that any path from the root that includes $edge1$ must pass through $anc(edge1, edge2)$ and any path from the root that includes $edge2$ must pass through $anc(edge1, edge2)$). Let $\{edge1, edge2\} \subseteq W$. Consider the two edges going into $anc(edge1, edge2)$. One of these two edges is colored white.

Let a and b denote the two bad nodes at the start of the game, and e and f the two elements of W at the start of the game. Because the two initial edges lie on the path P, invariant 1 will be satisfied at the beginning of the game. Also, $anc(e, f)$ will be either the node a or the node b, so invariant 2 will be satisfied at the beginning of the game as well.

Now suppose on some move the player colors white some $edge1 \in W$. By invariant 1, this move can only "fix" one bad node, so it will create two new bad nodes $bad1$ and $bad2$ in T. Let $bad1_B$ be the black edge entering $bad1$ after this move and $bad1_W$ the white edge entering $bad1$. Similarly let $bad2_B$ be the black edge entering $bad2$ after the move and $bad2_W$ the white edge entering $bad2$. ($bad1_B$ and $bad2_B$ will be the new members of W).

First consider the relationship between $bad1_B$ and $bad2_B$. Because there exists some path from the root to a leaf node in T that contains both $bad1_W$ and $bad2_W$, there will be no path from the root to a leaf node in T that contains both $bad1_B$ and $bad2_B$, so these two edges will not violate invariant 1 after the move. Also, because there exists some path from the root to a leaf node in T that contains both $bad1_W$ and $bad2_W$, either $bad1_W$ or $bad2_W$ will be going into $anc(bad1_B, bad2_B)$, so $bad1_B$ and $bad2_B$ will not violate invariant 2 after the move either.

Now consider some arbitrary edge $edge2 \in W$ that was in W before the move, and suppose for contradiction that after the move $edge2$ violates invariant 1 with one of the new members of W, say $bad1_B$. This means that $bad1_B$ and $edge2$ lie along the same path P'. $edge2$ cannot be above $bad1_B$ in P', because in this case $edge2$ and $edge1$ are both on some path, which would violate the fact that invariant 1 held before the move. Therefore $bad1_B$ is above $edge2$ in P'. We know that $edge1$ and $bad1_W$ lie on some path P'' since they were part of the same move. If $edge1$ is above $bad1_W$ in P'' then again $edge2$ and $edge1$ are both on some path, which would violate the fact that invariant 1 held before the move. Therefore $edge1$ is below $bad1_W$ in P''. But in this case $bad1$ is the node $anc(edge1, edge2)$, which is a contradiction since by invariant 2 one of the edges going into $anc(edge1, edge2)$ was already white before the move.

Now we show that $edge2$ and $bad1_B$ do not violate invariant 2 after the move either. Because every path from the root that goes through $edge1$ must pass through $anc(edge1, edge2)$, and $edge1$ and

$bad1_W$ lie along some path from the root to a leaf node since they are part of the same move, after the move there will be some path P' containing $bad1_W$ that goes through the node $anc(edge1, edge2)$. Suppose that $bad1_W$ is below $anc(edge1, edge2)$ in P'. Then $anc(edge2, bad1_B) = anc(edge1, edge2)$ so invariant 2 will not be violated by $edge2$ and $bad1_B$ after the move since invariant 2 held before the move. Otherwise suppose that $bad1_W$ is above $anc(edge1, edge2)$ in P'. Then $bad1_W$ is one of the edges going into $anc(edge2, bad1_B)$, so after the turn one of the edges going into $anc(edge2, bad1_B)$ is white and invariant 2 will not be violated by $edge2$ and $bad1_B$ in that case either.

By invariant 1, after every move $|W|$ increases by 2, so the player can never win. ∎

ECCC ISSN 1433-8092

http://eccc.hpi-web.de