[24] I. Schiermeyer. Solving 3-satisfiability in less than 1.579^n steps. In E. Borger, G. Jager, H. Kleine Buning, and S. Martini, editors, *Computer Science Logic 6th Workshop (CSL '92)*, volume 702 of *Lecture Notes in Computer Science*, pages 379–394. Springer–Verlag, 1993.

[25] I. Schiermeyer. Pure literal look ahead: An $o(1.497^n)$ 3-satisfiability algorithm. presented at the Satisfiability Workshop, Sienna, 1996.

[26] I. Schiermeyer. Pure literal look ahead: An $o(1.497^n)$ 3-satisfiability algorithm. Technical Report 96-230, University of Köln, 1996. Workshop on the Satisfiability Problem, Technical Report, Siena, April 29 – May 3, 1996.

[27] Roman Smolensky. Algebraic methods in the theory of lower bounds for Boolean circuit complexity. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, pages 77–82, New York City, 25–27 May 1987.

[28] M. Szegedy. What are the hardest instances of max clique? manuscript, 1997.

[29] Leslie G. Valiant. Graph-theoretic arguments in low-level complexity. In J. Gruska, editor, *Mathematical Foundations of Computer Science, 6th Symposium*, volume 53 of *Lecture Notes in Computer Science*, pages 162–176, Tatranská Lomnica, Czechoslovakia, 5–9 September 1977. Springer–Verlag.

[30] V.N. Vapnik and A. Ya Chervonenkis. On the uniform convergence of relative frequencies of events to their probabilities. *Theory of Probability Applications*, pages 264–280, 1971.

[31] Andrew Chi-Chih Yao. Separating the polynomial-time hierarchy by oracles (preliminary version). In *26th Annual Symposium on Foundations of Computer Science*, pages 1–10, Portland, Oregon, 21–23 October 1985. IEEE.

[32] Wenhui Zhang. Number of models and satisfiability of sets of clauses. *Theoretical Computer Science*, 155(1):277–288, 26 February 1996. Note.

[11] Johan Håstad, Stasys Jukna, and Pavel Pudlák. Top-down lower bounds for depth-three circuits. *Computational Complexity*, 5(2):99–112, 1995.

[12] Edward A. Hirsch. Two new upper bounds for SAT. In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 521–530, San Francisco, California, 25–27 January 1998.

[13] R. Impagliazzo, R. Paturi, and F. Zane. Which problems have strongly exponential complexity? manuscript, 1998.

[14] L. G. Kraft. A device for quantizing, grouping, and coding amplitude modulated pulses. Master's thesis, MIT, Cambridge, MA, 1949.

[15] O. Kullman. Worst-case analysis, 3-sat decision and lower bounds: approaches for improved sat algorithms. In D. Du, J. Gu, and P.M. Pardalos, editors, *Satisfiability Problem: Theory and Applications*, pages 261–313. American Mathematical Society, 1997.

[16] L.A. Levin. Universal sorting problems. *Problemy Peredaci Informacii*, 9:115–116, 1973. English translation in *Problems of Information Transmission* Vol. 9, pp. 265-266.

[17] J.H. Van Lint. *Introduction to Coding Theory*. Springer–Verlag, second edition, 1992.

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

[19] R. Paturi, P. Pudlák, M.E. Saks, and F. Zane. An improved exponential-time algorithm for k-sat. manuscript, 1998.

[20] Ramamohan Paturi, Pavel Pudlák, and Francis Zane. Satisfiability coding lemma. In *38th Annual Symposium on Foundations of Computer Science*, pages 566–574, Miami Beach, Florida, 20–22 October 1997. IEEE.

[21] Ramamohan Paturi, Michael E. Saks, and Francis Zane. Exponential lower bounds for depth 3 Boolean circuits. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, pages 86–91, El Paso, Texas, 4–6 May 1997.

[22] A.A. Razborov. Lower bounds on the size of bounded depth networks over a complete basis with logical addition. *Mathematische Zametki*, 41:598–607, 1986. English translation in *Mathematical Notes of the Academy of Sciences of the USSR* Vol. 41, pp. 333–338.

[23] N. Sauer. On the density of families of sets. *Journal of Combinatorial Theory, series A*, 13:145–147, 1972.

Bibliography

[1] M. Ajtái. σ_1^1-formulae on finite structures. *Annals of Pure and Applied Logic*, 24:1–48, 1983.

[2] N. Alon, J. Spencer, and P. Erdös. *The Probabilistic Method.* John Wiley & Sons, Inc., 1992.

[3] Richard Beigel and David Eppstein. 3-coloring in time $O(1.3446^n)$: a no-MIS algorithm. In *36th Annual Symposium on Foundations of Computer Science*, pages 444–452, Milwaukee, Wisconsin, 23–25 October 1995. IEEE.

[4] R.B. Boppana and M. Sipser. The complexity of finite functions. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol A.*, pages 759–804. Elsevier Science Publishers, 1990.

[5] Stephen A. Cook. The complexity of theorem-proving procedures. In *Conference Record of Third Annual ACM Symposium on Theory of Computing*, pages 151–158, Shaker Heights, Ohio, 3–5 May 1971.

[6] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5:394–397, 1962.

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

[8] Merrick Furst, James B. Saxe, and Michael Sipser. Parity, circuits, and the polynomial-time hierarchy. *Mathematical Systems Theory*, 17(1):13–27, April 1984.

[9] Jun Gu, P.W. Purdom, J. Franco, and B.W. Wah. Algorithms for the satisfiability (sat) problem: a survey. In D. Du, J. Gu, and P.M. Pardalos, editors, *Satisfiability Problem: Theory and Applications*, pages 19–51. American Mathematical Society, 1997.

[10] Johan Hastad. Almost optimal lower bounds for small depth circuits. In *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, pages 6–20, Berkeley, California, 28–30 May 1986.

```
%                                          2     3
%                     2 (-1 + 3 #1 - 3 #1  + #1 )
%
%                  2       3                 2       3       4
%         1 - 3 #1  + 2 #1  + Sqrt[1 - 6 #1  + 8 #1  - 3 #1 ]
%{P -> (------------------------------------------------------ & )}
%                                          2     3
%                     2 (-1 + 3 #1 - 3 #1  + #1 )


firstpartr = zeros(1,rbins);
firstpartr(1:ceil(2*rbins/3)) = ones(1,ceil(2*rbins/3));

innerterm = 1 - 6 * lowr.^2 + 8 * lowr.^3 - 3*lowr.^4;
num = 1 - 3 * lowr.^2 + 2* lowr.^3 - sqrt(innerterm);
denom = 2*(-1+3*lowr - 3*lowr.^2 + lowr.^3);

forcedeep = firstpartr .* (num ./ denom);

% the rest are forced with probability 1
forcedeep(ceil(2*rbins/3)+1:rbins) = ones(1,rbins-ceil(2*rbins/3));

% Indicator function for places where shallow trees are best:
indvec = forceshallow >= forcedeep;
% Use the worse tree.
forcetotal = forcedeep .* indvec + forceshallow .* (1-indvec);


% Probability of forcing =
%      sum_i (1/rbins * prob of forcing in bin i)
f = sum(forcetotal)/rbins;
```

```
% Shift the cummulative distribution over one.
% (ie, make the first bin 0)
% to account for "all things before this bin"
% For when we need an underestimate
lowH = zeros(1,rbins);
lowH(2:rbins) = highH(1:rbins-1);

% Compute, for each bin, the probability of being forced
% .* is pointwise multiplication of vectors

% In the 3-leaf tree, there are a number of ways of being forced:
% all three leaves come in bins before v
forceshallow = lowH.*lowH.*lowH;
% two leaves come before this bin, one comes in this bin
% three ways this can happen,
% probability 1/2 of forcing when it does
forceshallow = forceshallow + 1.5 * lowH.*lowH.*h;
% two leaves in this bin, one comes before
% three ways to happen, prob 1/3 when it does
forceshallow = forceshallow + lowH.*h.*h;
% all leaves in same bin as root, 1/4 chance of forcing
forceshallow = forceshallow +  1/4*h.*h.*h;

% In the infinite tree

% Now, generate vectors representing r's upper and lower bounds
% highr = 1/rbins, 2/rbins, ..., 1
highr = cumsum(ones(size(h)))/rbins;
% lowr = 0, 1/rbins, 2/rbins, ..., 1-1/rbins
lowr = zeros(1,rbins);
lowr(2:rbins) = highr(1:rbins-1);

% Now, there are three functions:  According to mathematica:
%In[3]:= DSolve[P[r]==(r+(1-r)P[r])^3,P,r]
%
%Out[3]= {{P -> (1 & )},
%
%
%                2       3                 2       3       4
%        1 - 3 #1  + 2 #1  - Sqrt[1 - 6 #1  + 8 #1  - 3 #1 ]
%{P -> ------------------------------------------------------ & )},
```

```
forceshallow = forceshallow +  1/3*h.*h;

% In the infinite tree

% Now, generate vectors representing r's upper and lower bounds
% highr = 1/rbins, 2/rbins, ..., 1
highr = cumsum(ones(size(h)))/rbins;
% lowr = 0, 1/rbins, 2/rbins, ..., 1-1/rbins
lowr = zeros(1,rbins);
lowr(2:rbins) = highr(1:rbins-1);

% Use (r/(1-r))^2 for first half, 1 for second half
% Made complicated by the fact that we don't want a
% divide by 0.
firsthalfr = zeros(1,rbins);
firsthalfr(1:ceil(rbins/2)) = lowr(1:ceil(rbins/2));
% We know that the worst point in the interval is the left edge
forcedeep = (firsthalfr ./ (1-firsthalfr)).^2;
forcedeep(ceil(rbins/2)+1:rbins) = ones(1,rbins-ceil(rbins/2));

% Indicator function for places where shallow trees are best:
indvec = forceshallow >= forcedeep;
% Use the worse tree.
forcetotal = forcedeep .* indvec + forceshallow .* (1-indvec);


% Probability of forcing =
%      sum_i (1/rbins * prob of forcing in bin i)
f = sum(forcetotal)/rbins;
```

FORCEPROB for $k = 4$

```
% Compute the probability of forcing given
% h = histogram for the desired probability distribution
function f=forceprob(h)

rbins = size(h,2);

% Compute cummulative distribution from pointwise
% This includes things in the current bin, so it is an
% overestimate
highH = cumsum(h);
```

```
% Compute running time due to setting nondefined variables
% Since this is better with more defining variables,
% use low end of interval on dfrac
% forceprob(x) computes the probability of being forced
% given that the permutation is x--good
2 = (1-forceprob(x)).*(1-dfrac);
% Sum these terms
tmpf = tmpf1+tmpf2;

% For each value of r, take the worst bound
f= max(tmpf);
```

FORCEPROB for $k = 3$

```
% Compute the probability of forcing given
% h = histogram for the desired probability distribution
function f=forceprob(h)

rbins = size(h,2);

% Compute cummulative distribution from pointwise
% This includes things in the current bin, so it is an
% overestimate
highH = cumsum(h);

% Shift the cummulative distribution over one.
% (ie, make the first bin 0)
% to account for "all things before this bin"
% For when we need an underestimate
lowH = zeros(1,rbins);
lowH(2:rbins) = highH(1:rbins-1);

% Compute, for each bin, the probability of being forced
% .* is pointwise multiplication of vectors

% In the 2-leaf tree, there are a number of ways of being forced:
% both leaves come in bins before v
forceshallow = lowH.*lowH;
% one leaf comes before this bin, one comes in this bin
% two ways this can happen, probability 1/2 of forcing when it does
forceshallow = forceshallow + lowH.*h;
% both leaves in same bin as root, 1/3 chance of forcing
```

```
function f=wrapfun(h,dfrac,d)

% k = length of h
k=size(h,2);

% extend y by 1.  Last element of y = 1 - sum(elts in h)
y=zeros(1,k+1);
y(1:k)=h(1:k);
y(k+1)=1-sum(h);

% y is a valid probability measure if it is > 0 everywhere
if min(y) < 0
% if not, penalize it
% for y<0, this is > 1, which is a bad running time
% also, the penalty function needs to increase as constraints are
% violated to allow minimizer to find the valid region
f=2^(-min(y));
else
% otherwise, just compute the running time
f = runtime(y,dfrac,d);
end
```

Runtime

```
% Compute the running time for the algorithm
%   x = vector of variables (defining x bins)
%   dfrac = fraction of defining variables
function f=runtime(x,dfrac,d)

% step size for fraction of defining variables
% for correct bounds, we need to know where both
% the left and right edges of the interval are
% so we can use the worst-case value
derror = 1/d;

% Compute running time due to the permutation
% using entropy method
% Since this is worse with more defining variables,
% use high end of interval on dfrac
% ent(x) computes the entropy of the vector x
tmpf1 = (dfrac+derror).*(log2(size(x,2))-sum(ent(x)));
```

```
% initialize guess for H(r) to uniform
startingH=ones(1,x-1)/x;

% iterate over the steps in the number of defining variables
for i=startd-1:-1:0;

% find one that minimizes the running time from wrapper function
% i/d = fraction of defining variables, d passed as argument so the
%      procedures know the step size
y=fmins('wrapfun',startingH,options,[],i/d,d);

% store results in output matrices
% matlab matrices start at 1
permdata(i+1,1:x)=extend(y);
timedata(i+1)=runtime(extend(y),i/d,d);

% checkpoint in case of crash
save permdata;
save timedata;

% output current state
[i,timedata(i+1)]

% use current value as starting point
tmpfstartingH = y;

end

% output worst-case running time
max(timedata)
```

Wrapfun

```
% This is just a wrapper for the real computation of the
% running time to add in a penalty function for not having a
% valid probability measure.
%   h     = vector defining the function h
%   dfrac = fraction of defining variables
%   d     = number of defining variables
%           (for calculating step sizes)
```

Appendix A

Matlab Code For Proof Search

The code which optimizes the parameters of the proof in Section 2.5.3 consists of several procedures: DOCOMPUTE is the outer loop, running over the discrete values of Δ searching for the worst-case value. It calls WRAPFUN, which is a wrapper on the real objective function (the running time of the algorithm), adding a penalty for having a invalid distribution function where the components of $\vec{h}$ do not sum to 1. RUNTIME computes the running time in terms of the probability that the placement is $\vec{H}$–good and FORCEPROB. Two versions of FORCEPROB given, one each for $k = 3, 4$.

DOCOMPUTE

```
% Do a complete run
%    x = number of bins (x-1 = number of variables)
%    d = number of values for fraction of defining variables
%    startd = starting value for d

function [f,val]=docompute(x,d,startd)

% matlab specific: increase number of iterations allowed
options = foptions;
options(14)=1000*x;

% create storage for all intermediate results for checkpointing
% initialize matrix for storing the permutation data
permdata=zeros(d+1,x);
% initialize vector for storing the running time exponents
timedata=ones(1,d+1);
```

Since nearly all degree 2 GF(2) polynomials have $O(n^2)$ monomials, by Lemma 4.2, with high probability, P requires Σ_3^k circuits of size at least $2^{(1-\epsilon)n}$ for every $k = o(\log \log n)$.

Applying Lemma 4.10 in the contrapositive direction with $l = 2$, this implies that there do not exist circuits computing Parity on the range of G_P with size less than $2^{2k}2^{(1-\epsilon)n} \leq 2^{n-o(n)}$.

■

As noted earlier, this is nearly tight, since there are only 2^n inputs in the domain of the map, so there exists a depth-2 circuit of size at most 2^n computing any function on the range of the map.

Lemma 4.10 *Let P be a $GF(2)$ polynomial of degree l on n input variables. If there exists a Σ_3^k circuit of size s computing* Parity *on the range of g_P, then there exists a Σ_3^{kl} circuit of size $O(2^{kl}s)$ computing P.*

Proof: Let C be the circuit computing Parity on the range of g_P. Each bottom level gate is an OR involving at most k variables of the range or their negations. Thus, it is some function of at most kl variables of the domain, and thus can be expressed as a kl–CNF of size at most 2^{kl}. By doing so and collapsing adjacent levels of ANDs, we obtain a Σ_3^{kl} circuit using the inputs from the domain of g_P. This circuit computes the parity of the monomials of P evaluated at a point of the domain of g_P, which is simply the value of the polynomial P at the input. The size can increase by a factor of at most 2^{kl}. ■

By applying this lemma in the contrapositive direction, a lower bound on the size of circuits computing P yields a lower bound on the size of circuits computing Parity on the range of g_P. Without the Sparsification Lemma, however, this would not be enough to obtain a generator of hard subsets for Parity: Finding a polynomial hard for Σ_3^k circuits using a straightforward counting argument would require using polynomials of degree greater than k, and applying Lemma 4.10 in the needed direction requires choosing $l < 1$. With the Sparsification Lemma, we obtain the following result.

Theorem 4.2 *There is a family of functions G each mapping n bits to $O(n^2)$ bits indexed by $O(n^2)$ bits so that with probability approaching 1, for a randomly chosen $g \in G$, computing parity on the range of g requires Σ_3^k circuits of size at least $2^{n-o(n)}$ for $k = o(\log\log n)$.*

Proof: Let $\mathcal{P}$ be the set of degree 2 GF(2) polynomials with $O(n^2)$ monomials. Let G consist of the maps G_P defined by each $P \in \mathcal{P}$. Finally, let $\epsilon > 0$ be a constant. We will show that for every such ϵ, for a randomly chosen G_P there are no circuits of size $2^{(1-\epsilon)n}$.

the substitution. Furthermore, these choices are independent of those made for P_2. From this, we can conclude that the probability that P restricted to A' is identically 1 is at most $2^{-\binom{\delta n}{2}}$.

We now use this upper bound on the probability that a given sparse k–CNF accepts a large subset of $P^{-1}(1)$ (rejecting all of $P^{-1}(0)$) to bound the probability that *any* sparse k–CNF accepts such a set. Since the number of k–CNFs with Dn clauses is at most

$$\binom{(2n)^k}{Dn}$$

this implies that the probability that there exists such a sparse k–CNF accepting a large subset of $P^{-1}(1)$ is at most

$$2^{-\binom{\delta n}{2}}\binom{(2n)^k}{Dn} \leq 2^{-\binom{\delta n}{2}+Dnk\log(2n)}$$

This bound on the probability that there exists some sparse k–CNF accepting a large subset of $P^{-1}(1)$ approaches 0, since $\delta = \theta(1) = \omega(\sqrt{\log n/n})$. For a randomly chosen P, with high probability there does not exist a sparse k–CNF accepting a large subset of $P^{-1}(1)$ for this P, and thus no small Σ_3^k circuit C computing P exists. ■

4.2.2 Constructing Hard Subsets

There is a natural association between computing Parity and computing GF(2) polynomials: Viewing the monomials as variables, the polynomial is simply the parity of these new input variables. This will allow us to construct generators of hard subsets for Parity using the results in the previous section.

Definition 4.9 *Given a GF(2) polynomial P with m monomials, let $g_P : \{0,1\}^n \to \{0,1\}^m$ be the map defined by evaluating an n bit input on each of the m monomials of P.*

remaining variables, we obtain a nonzero polynomial of degree at most 2 on the two variables x, y. This polynomial on two variables can be 0 on at most 3 of the 4 inputs. From this, we can conclude that P is 0 on at most $1/4$ of the 2^n inputs.

Applying Lemma 3.1, we obtain a depth-2 subcircuit of C' which rejects all of $P^{-1}(0)$ and accepts at least $2^{2\epsilon n-2} \geq 2^{\epsilon n}$ points from $P^{-1}(1)$. Moreover, this k–CNF subcircuit is sparse. We now show that this is not possible.

Fix such a sparse k–CNF B and let A be the set of inputs accepted by B. Again applying the VC-dimension arguments, Lemma 3.15 implies that there exists a set of variables V shattered by A. From this, we can conclude that there exists a set $A' \subseteq A$ in which every partial assignment to V appears exactly once. Since $|A| \geq 2^{\epsilon n}$, $|V| \geq \delta n$ for some constant $\delta > 0$ depending on ϵ. We call the set V the free variables. Over the set A', every nonfree variable can be expressed as a function of the free variables in brute-force fashion using a GF(2) polynomial of degree δn.

We now show that with high probability, P is not identically 1 on A', which implies that the k–CNF B mistakenly accepts some input for which P is 0, producing the desired contradiction. We do so by showing that, with high probability, P restricted to A' contains some monomials of degree 2. Express P as $P_1 + P_2$, where P_1 consists of monomials involving only free variables, and P_2 consists of all other monomials. We now consider P as being generated by choosing P_1 and P_2 independently at random from their respective spaces. Given P, we can obtain its restriction to A' by substituting the degree δn polynomials for the nonfree variables described above. This clearly gives a polynomial equivalent to P on A', and such a representation must be unique. We first choose P_2 at random. Applying this substitution to P_2, we obtain some polynomial in the free variables. These substitutions have no effect on the terms from P_1, since they involve only free variables. Thus, P restricted to A' contains a degree 2 monomial unless the monomials chosen for P_1 exactly match those which exist in P_2 after

of such circuits computing Parity on such a set. First, one could compute Parity on the entire range; however, this requires a Σ_3^k circuit of size $2^{O(n^2/k)}$ which for small k is impractically large. Second, one can explicitly identify each point of the range of this map and compute Parity on the set using a depth-2 circuit of size poly$(n)2^n$. Our goal is to find methods of constructing such sets whose hardness approaches this second bound.

4.2.1 Lower Bounds on Random Polynomials

Our first step is to show that functions defined by random GF(2) polynomials of degree 2 require large Σ_3^k circuits for all $k = o(\log \log n)$. To do so, we make use of the same top-down argument applied in Chapter 3.

Definition 4.8 *G_n^2 is the space of degree 2 $GF(2)$ polynomials on n variables, including monomials of all degrees ≤ 2.*

Theorem 4.1 *Let P be selected randomly from G_n^2, and let $k = o(\log \log n)$. With high probability, every Σ_3^k circuit computing P has size at least $2^{n-o(n)}$.*

Proof:

Let $\epsilon > 0$, and assume that C is a Σ_3^k circuit of size $2^{(1-3\epsilon)n}$ computing P. By showing that this leads to a contradiction for every constant $\epsilon > 0$, we obtain the theorem.

Let $D = (2/\epsilon)^{O(k2^k)}$. Every depth–2 subcircuit of C is a k–CNF. Using the Sparsification Lemma for k–CNFs, we replace every such subcircuit with an OR of $2^{\epsilon n}$ k–CNFs, each with at most Dn clauses. By collapsing these ORs into the output gate of C, this yields a new Σ_3^k circuit C' of size $2^{(1-2\epsilon)n}$ which computes the same function as C.

We now wish to apply Lemma 3.1 to the circuit C' computing the function defined by P. To do so, we must first show that $P^{-1}(1)$ is not too small. Pick some degree 2 monomial xy in P with nonzero coefficient. For every setting to the

Proof: We use a straightforward reduction of k–SAT to k–Set Cover.

We define the instance of k–Set Cover as follows: For every $i \leq n$, add the trivial clause $(x_i \vee \bar{x}_i)$ to the formula. U is the set of $2n$ literals. For every clause, we create a family of sets $\mathcal{S}$. For each clause in this modified formula, we construct a set containing the literals appearing in that clause.

Because of the sets arising from the added trivial clauses, if the set cover has size n, then it corresponds to an assignment of the satisfiability problem, since it must choose exactly one of $\{x_i, \bar{x}_i\}$. Second, if the set of elements covers all of the sets arising from the clauses of the formula, then the corresponding assignment satisfies the formula. Also, it is easy to see that if the formula is satisfiable, then there exists a set cover of size n. Finally, given a k–Set Cover problem which includes these sets corresponding to the trivial clauses, we can construct a k–CNF which has a satisfying assignment if and only if there exists a set cover of size n, by reversing these arguments.

Now, we apply the Sparsification Lemma for k–Set Cover to this problem, and obtain $t = 2^{\epsilon n}$ sparse set cover problems $\mathcal{T}_1, \ldots, \mathcal{T}_t$ with $\sigma(\mathcal{S}) = \cup_{i=1}^{t} \mathcal{T}_i$. For each restriction $\mathcal{T}_i$ output by the algorithm, we can add the trivial sets $\{x_i, \bar{x}_i\}$ without changing the set of set covers, since these trivial sets belonged to the original family $\mathcal{S}$. As noted above, we can then construct a formula Ψ_i whose set of satisfying assignments is in one-to-one correspondence with the set covers of size n, which are in turn in correspondence with the satisfying assignments of Φ. Thus, $\mathrm{Sat}(\Phi) = \cup_{i=1}^{t} \mathrm{Sat}(\Psi_i)$, and $\Phi = \vee_{i=1}^{t} \Psi_i$. ■

4.2 Hard Subsets of Parity

Using this Sparsification Lemma we will now construct a "pseudo-random" generator of hard subsets of the parity function. Given some advice bits, this generator will map n bits to $O(n^2)$ bits so that computing parity on the range of this map requires large Σ_3^k circuits. There are two obvious upper bounds on the size

Proof: To bound the number of families output, we bound the number of distinct paths from the root to a leaf in the recursion tree. Each path is uniquely identified by the set of decisions (adding heart or petals) taken along that path. Since any set added is of size at most $k-1$, Lemma 4.6 implies that the number of sets added along any path is at most $\beta_{k-1}n$. By Lemma 4.7, the number of steps in which petals are added is at most kn/α. Thus, the total number of paths is at most

$$\sum_{r=0}^{nk/\alpha} \binom{\beta_{k-1}n}{r}$$

We bound this quantity using the binary entropy function $h(\delta) = \delta \log_2 \delta + (1-\delta)\log_2(1-\delta)$. Specifically, we use the bound

$$\binom{n}{\delta n} \leq 2^{(h(\delta)+o(1))n}$$

In our case, this yields an upper bound on the number of families output of

$$\begin{aligned}
2^{\beta_{k-1}nh(k/(\alpha\beta_{k-1}))} &\leq 2^{(2k/\alpha)n\log(\alpha\beta_{k-1}/k)} \text{ since the first term in } h \text{ is larger} \\
&\leq 2^{(2k/\alpha)n\log(\alpha(4\alpha)^{2^{k-2}-1})} \text{ by the definition of } \beta \\
&\leq 2^{(2k/\alpha)n(2^{k-2})\log 4\alpha} \\
&\leq 2^{\epsilon n} \text{ by the definition of } \alpha
\end{aligned}$$

■

Having shown this lemma for the k–Set Cover problem, we return to our original k–SAT problem.

Lemma 4.9 (Sparsification (k–CNF)) *Let Φ be a k–CNF on n variables. For every $\epsilon > 0$, there exists a constant D so that Φ can be expressed as $\Phi = \vee_{i=1}^{t} \Psi_i$, where $t \leq 2^{\epsilon n}$, and each Ψ_i is a k–CNF with at most Cn clauses. Furthermore, this representation can be produced by an algorithm running in time poly$(n)2^{\epsilon n}$.*

heart or petal, and any set eliminated because of S must be eliminated in this call to `Simplify`. ■

Lemma 4.6 *The total number of sets of size at most i that are added along any path is at most $\beta_i n$.*

Proof: The proof is by induction on i. When $i = 1$, any new set added is an element. Since the calls to `Simplify` ensure that a set can be added at most once, there are at most $n = \beta_1 n$ such additions. Now, we assume that at most $\beta_{i-1} n$ sets of size at most $i-1$ are added along any path. Every i–set added is either present in the family $\mathcal{T}$ output by `Reduce`, or it is eliminated by another, smaller set. By Lemma 4.3, there are at most $\theta_{i-1} n$ i–sets in the output. Any i–set eliminated is removed by a set of size at most $i-1$ added during the execution. By Lemma 4.5, each such smaller set can eliminate at most $2\theta_{i-1}$ i–sets, and by the induction hypothesis, there are at most $\beta_{i-1} n$ such smaller sets. Thus, the total number of sets of size at most i added is at most

$$\theta_{i-1} n + 2\beta_{i-1}\theta_{i-1} \leq 4\beta_{i-1}\theta_{i-1} n = (4\alpha\beta_{i-1}^2) n = \beta_i n$$

■

We now show that the number of steps in which the petals of a sunflower are added is significantly smaller.

Lemma 4.7 *At most kn/α families along any path are created by adding petals.*

Proof: By Lemma 4.6, the number of i–sets added along any path is at most $\beta_i n$. If a family is created by adding petals of size i, then at least θ_i such petals are added. Therefore, there are at most $\beta_i n/\theta_i = n/\alpha$ steps which add petals of size i, and at most kn/α steps which add petals of any size. ■

Lemma 4.8 *The number of families output by reduce is at most $2^{\epsilon n}$.*

$\mathcal{T}$ is at most Dn, with $D = \sum_i \alpha\beta_i = (2/\epsilon)^{O(2^k)}$, proving another of the conditions of the lemma.

We now must show that the number of sets which are created and then removed by `Simplify` is not too large. To do so, we bound the number of added sets which are removed because they are supersets of a given set S.

Lemma 4.4 *If $\mathcal{T}$ is a family such that an i–set has been added to it or any of its predecessors in the path, then no element u appears in more than $2\theta_{i-1}$ i–sets in $\mathcal{T}$.*

Proof: If this statement were false, it would be false in some step in which an i–set was added, so it suffices to consider such steps. Let $\mathcal{T}'$ be the family to which some i–sets were added. There must be some weak sunflower F in $\mathcal{T}'$ which gave rise to these new sets, either by adding the heart or the petals of F to $\mathcal{T}'$. If an i–set is added, then there is no sunflower of j–sets with $j \leq i$. In particular, every element u is contained in less than θ_{i-1} i–sets.

Now, we consider the two branches. If the heart is added, at most one i–set is added to the less than θ_{i-1} which already contain u. Since $\theta_{i-1} \geq 1$, this is less than $2\theta_{i-1}$ i–sets. If the petals are added, we can add at most θ_{i-1} i–sets containing u. Otherwise, the i–sets added form a weak sunflower with petal size less than i, which would have been chosen instead of F. Thus, the total number of i–sets containing u is at most $2\theta_{i-1}$. ■

Lemma 4.5 *At most $2\theta_{i-1}$ of the added i–sets can be eliminated because of a single set.*

Proof: Let S be an i–set added to a family $\mathcal{T}$. By Lemma 4.4, no element u appears in more than $2\theta_{i-1}$ i–sets. This implies that in the next call to `Simplify`, at most $2\theta_{i-1}$ i–sets can be eliminated, since any set eliminated must contain every $u \in S$. After this, no set strictly contains S. Thus, S never again appears as a

Let $T \in \sigma(\mathcal{T})$. Since `Simplify` only removes sets which are supersets of other sets, we likewise have $T \in \sigma(\texttt{Simplify}(\mathcal{T}))$. If a weak sunflower $\{S_1, \ldots, S_c\}$ is found during the execution of $\texttt{Reduce}(\mathcal{T})$, then either $T \cap H \neq \emptyset$, or for every $1 \leq l \leq c$, $T \cap (S_l - H) \neq \emptyset$, since T is a cover. T is therefore a cover for at least one of the two families produced in this call. By a similar argument, no new covers are added to $\sigma(\mathcal{T})$ by this process. ■

To show that the number of restrictions output by `Reduce` is not too large, we examine a path through the recursion tree that leads to any family $\mathcal{T}$ output by `Reduce`. We show that such a path has length rn for some constant r, but the number of branches where the petals were chosen is at most some sufficiently small fraction of n. This will imply that the total number of paths, and thus of restrictions output, is sufficiently small. To bound the length of these paths, we bound the number of sets which are added to the family along this path, since each recursive call adds some number of sets. Any set added is either present in the output, or is removed by a call to `Simplify` before `Reduce` terminates. By accounting for these two cases, we obtain the desired bounds on the depth of the recursion.

First, we will need to recall our definitions. Let α be an integer such that $\alpha / \log(4\alpha) > 4k2^k/\epsilon$. For $1 \leq i \leq k-1$, let $\theta_0 = 1$, $\beta_i = (4\alpha)^{2^{i-1}-1}$, and $\theta_i = \alpha\beta_i$.

Lemma 4.3 *For all $1 \leq i \leq k$, every family $\mathcal{T}$ output by $\texttt{Reduce}(\mathcal{S})$ has at most $(\theta_{i-1} - 1)n$ sets of size i.*

Proof: Suppose that a family $\mathcal{T}$ output by `Reduce` contained more than $(\theta_{i-1} - 1)n$ sets of size i. By the pigeonhole principle, there must be some element u belonging to at least θ_{i-1} of them. These sets containing u then form a weak sunflower with θ_{i-1} petals of size at most $i-1$. However, if such a family exists, the algorithm cannot terminate. ■

This establishes that the total number of sets in any $\mathcal{T}$ output by `Reduce` is at most $\sum_{i=1}^{k} \theta_i n$. By the definition of θ, this implies that the number of sets in

DLL procedures discussed earlier. Here, however, we allow branching on the more general condition on a set of elements, rather than a single one.

During the course of the algorithm, sets are added to an existing family during the recursive calls, and removed by `Simplify` when it finds one set contained in another.

Lemma 4.1 (Sparsification-Set Cover) *Let $\mathcal{S}$ be a family of sets of size k over a universe of n elements. For every $\epsilon > 0$, there exists a constant C such that*

- `Reduce`$(\mathcal{S})$ *outputs a list of $t \leq 2^{\epsilon n}$ restrictions $\mathcal{T}_1, \ldots, \mathcal{T}_t$ of $\mathcal{S}$*
- $\sigma(\mathcal{S}) = \cup_{i=1}^{t} \sigma(\mathcal{T})$
- *For every i, $|\mathcal{T}_i| \leq Dn$*
- *As a function of k, $C = (2/\epsilon)^{O(2^k)}$.*
- `Reduce`$(\mathcal{S})$ *runs in time poly$(n)2^{\epsilon n}$*

The proof of this lemma will require a number of intermediate steps

Lemma 4.2 `Reduce`$(\mathcal{S})$ *returns a set $\{\mathcal{T}_1, \ldots, \mathcal{T}_t\}$ of restrictions of $\mathcal{S}$ with $\sigma(\mathcal{S}) = \cup_{l=1}^{t} \sigma(\mathcal{T}_l)$.*

Proof: For any family $\mathcal{T}$, every set added to a family by the recursive calls during `Reduce`$(\mathcal{T})$ is a subset of some $T \in \mathcal{T}$. Thus, every family examined by the algorithm is a restriction of $\mathcal{S}$.

We now examine the recursion by taking the families already output by `Reduce` together with the families passed as arguments to outstanding recursive calls. This collection $\{\mathcal{T}_1, \ldots, \mathcal{T}_c\}$ represents the state of the algorithm. The output of the algorithm is simply the state at the time the algorithm terminates.

We show that if the statement of the lemma holds (that is, $\sigma(\mathcal{S}) = \cup_{i=1}^{t} \sigma(\mathcal{T}_i)$) at the beginning of the execution of a call to `Reduce`, then it remains true afterwards. The lemma then follows by induction.

Definition 4.6 *A* j–set *is a set of size* j.

Definition 4.7 *A collection of* j*–sets* $S_1, \ldots, S_c$ *is a* weak sunflower *of* j*–sets if* $\cap_{i=1}^{c} S_i \neq \emptyset$. *We call* $H = \cap_{i=1}^{c} S_i$ *the* heart *of the sunflower, and the collection* $\{S_1 - H, \ldots, S_c - H\}$ *the* petals. *The* petal size *of the sunflower is* $j - |H|$, *the size of a petal.*

The recursive algorithm `Reduce` effects this transformation, turning an instance of k–Set Cover into several sparse instances. To do so, it requires some global parameters θ. These parameters will be examined more closely in the proof, but can be computed from the expressions $\alpha / \log(4\alpha) > 4k2^k/\epsilon$, and $\theta_0 = 1$, $\beta_i = (4\alpha)^{2^{i-1}-1}$, and $\theta_i = \alpha\beta_i$, for $1 \leq i \leq k-1$.

It also makes use of a procedure `Simplify`. `Simplify`($\mathcal{S}$) removes any $S \in \mathcal{S}$ for which there exists $S' \in \mathcal{S}$ with $S' \subset S$.

```
Reduce(collection of sets S)
  S = Simplify(S)
  for j = 2 to k
    for i = 1 to j - 1
      if there is a weak sunflower {S_1, ..., S_c} of j–sets
        with at least θ_i petals of size i
      then
        Reduce(S ∪ {H})
        Reduce(S ∪ {S_1 - H, ..., S_c - H})
        return
  output S
  return
```

`Reduce` outputs a collection $\mathcal{T}_1, \ldots, \mathcal{T}_t$ of families of sets by recursively considering such families. We can view this recursion as a tree which has nodes corresponding to each call to `Reduce`. With each node, we associate the family of sets passed as the argument to that call. The children of a node are the two recursive calls made, one in which H is added to $\mathcal{S}$, and another in which the sets $\{S_1 - H, \ldots, S_c - H\}$ are added. This backtracking search is similar to the

as a subset of the space of possible inputs of length m, and ask how large a Σ_3^k circuit is required to compute Parity on this subset, allowing it to return arbitrary values on inputs outside this subset. We show that this subset is hard, meaning that for computing Parity correctly on this small subset of inputs is essentially as hard as computing correctly on all inputs of length m. While the existence of such sets can be shown using the Switching Lemma [10], this gives a more constructive method for producing such sets.

4.1 Sparsification Lemma

In this section, we state and prove the Sparsification Lemma, and give some of its algorithmic consequences. To simplify the presentation, we actually prove the lemma for the k–Set Cover Problem rather than k–SAT, and then derive the lemma for k–SAT as a consequence.

> k–SET COVER
>
> INSTANCE: Universe U of elements, collection $\mathcal{S}$ of subsets of U. For all $S \in \mathcal{S}$, $S \subseteq U$ with $|S| \leq k$.
>
> PROBLEM: Find the minimal set cover; that is, find a collection $U' \subseteq U$ such that for all $S \in \mathcal{S}$, there exists $u \in U'$ with $u \in S$.

We will represent an instance of k–SAT as a k–Set Cover problem in a straightforward way, by choosing the universe to be the set of literals $\{x_i, \bar{x}_i\}$ and defining sets which correspond to the clauses of the formula. For now, however, we restrict our attention to the set cover problem.

Definition 4.3 *An instance of k–Set Cover is sparse if the number of sets is $O(n)$.*

Definition 4.4 *$\sigma(\mathcal{S})$ denotes the family of set covers of $\mathcal{S}$*

Definition 4.5 *If $\mathcal{S}$ and $\mathcal{T}$ are families of sets, $\mathcal{T}$ is a* restriction *of $\mathcal{S}$ if for every $S \in \mathcal{S}$, there exists $T \in \mathcal{T}$ with $T \subseteq S$.*

sparse k–CNF is only linear in the number of variables, the number of variables in the resulting 3–CNF is still $O(n)$. Therefore, the hypothesized 3–SAT algorithm runs in time which is still subexponential in n, and since there only subexponentially many 3–CNFs to be checked, the whole process runs in subexponential time.

As in the case of the satisfiability algorithms analyzed earlier, the properties of CNFs we use to analyze algorithms actually expose limitations on the expressive power of these formulae. The Sparsification Lemma states that sets accepted by a k–CNFs also have the property that they can be decomposed into sets accepted by sparse k–CNFs in an efficient way. Using this characterization, we return to the problem of constructing hard functions, that is, functions which require large circuits. From the top-down argument used in the previous section, we know that to prove lower bounds on depth–3 circuits with bottom fanin k, we can simply find a function f such that no large subset of $f^{-1}(1)$ can be accepted by a k–CNF. Using the Sparsification Lemma, we turn a Σ_3^k circuit into a slightly larger Σ_3^k circuit which has the additional property that every depth-2 CNF subcircuit is sparse. Following this top-down argument, we see that there must exist some sparse k–CNF which accepts a large subset of $f^{-1}(1)$. Ideally, to show that f is hard, we would then present an explicit function for which this cannot be done. However, we are only able to show something more modest: there exists a small family of very simple functions such that nearly all of them are sufficiently hard. While this does not directly give a lower bound, it suggests a direction for future work.

However, this family of functions is quite simple and has a nice algebraic structure. By exploiting this, we obtain a 'pseudorandom' generator of sets of inputs for Σ_3^k circuits computing parity. This generator takes as advice the description of a function in this family. Using this advice, it maps binary strings of length n to ones of length m, with $n < m$. We then view the range of this map

k–CNFs, as long as k is not too large. This will give us a reduction (more specifically, a Turing-reduction) between arbitrary k–CNFs and sparse ones. When these reductions are used for NP-completeness, in order to make the relationships between reduced problems meaningful, we require that such reductions are efficient in a polynomial sense. For this reduction, that would mean that only polynomially many sparse formula are needed, and each such formula can be output in polynomial time. In our current setting, we are investigating relationships between problems solved by exponential time algorithms, rather than polynomial time ones. Thus, we can relax our notion of an efficient reduction to include reductions which expand the problem by a subexponential factor.

Definition 4.2 *$f(n)$ is subexponential if, for every constant $\epsilon > 0$, $f \leq 2^{\epsilon n}$ for sufficiently large n.*

Representing a k–CNF as a subexponential union of sparse k–CNFs in subexponential time has several consequences for satisfiability algorithms. First, this gives us justification for our intuition that sparse instances are hard. Given any other instance F, we first express it as $F = \vee F_i$, with each F_i sparse and the number of F_i subexponential in n. By checking each F_i for satisfiability, we can determine whether F is satisfiable. Thus, if we had a subexponential time algorithm for k–SAT which solved all sparse instances, we would have such an algorithm for any instance. In addition, this result allows us to relate the complexity of k–SAT to 3–SAT. The standard reduction between the two adds additional variables *per clause*; thus a 4–CNF on n variables could become a 3–CNF on $O(n^4)$ variables. Because of this, even a subexponential 3–SAT algorithm could yield a very poor 4–SAT algorithm and an even worse one for k–SAT for larger k. Using the Sparsification Lemma, if we had a subexponential time algorithm for 3–SAT, we obtain a subexponential time algorithm for k–SAT for all constant k: First, we rewrite the input k–CNF using subexponentially many sparse k–CNFs. Then, we apply the standard reduction to 3–SAT to each formula. Since the number of clauses in each

Chapter 4

CNFs

In our search for algorithms with better worst-case running times, finding one with a polynomial running time seems unlikely, since it would violate widely believed complexity assumptions ($P \neq NP$, or in the case of our randomized algorithms, $RP \neq NP$). If we believe such assumptions, then some instances of satisfiability must be hard. On the other hand, many instances are quite easy: If the formula contains a simple contradiction involving only a few variables, it is easy to determine that it is unsatisfiable. Distinguishing between the easy and hard instances would be useful, enabling us to focus our attention on the hard instances in the way that example of the formula Unique from Section 2.4 led us to the `ResolveSAT` algorithm.

In this section, we will provide some insight regarding these hard instances by showing that they must lie in a much smaller subclass of k–CNF formulae.

Definition 4.1 *A CNF formula F on n variables is* sparse *if the number of clauses in F is $O(n)$.*

Sparse k–CNFs are natural candidates for hard instances, since the standard reduction of a generic NP computation to satisfiability uses such instances.

We first prove a technical lemma (the *Sparsification Lemma*) which says that *any* k–CNF can efficiently be written as the union (OR) of several sparse

inputs. By Lemma 3.16, C_i accepts projection of dimension greater than $\sqrt{n/2}$. Since C_i accepts a subset of an error-correcting code of distance $\sqrt{2n}$, this contradicts Lemma 3.17. ■

For every partial assignment α to the variables in X', $F|_\alpha$ is satisfiable because there exists some $a \in A$ with that setting. Thus, by Lemma 3.14, $\mathrm{Sat}(F)$ contains a projection of dimension $\log(|A|)/\log n$. ■

With this characterization in hand, we can obtain a lower bound on the size of Σ_3^2 circuits accepting the codewords of an error-correcting code.

Definition 3.16 *If P is a projection, the* parts P_i *of P are the sets of variables v such that for every v, the part containing v consists of all v' such that either $v = v'$ or $\bar{v} = v'$.*

Lemma 3.17 *If A is the set of codewords of an error-correcting code with distance d, then A does not contain any projections of dimension greater than n/d.*

Proof: For the purposes of contradiction, let $P \subseteq A$ be a projection of dimension greater than n/d. There are at most n variables which are not set to constants on P. Thus, there is some part P_i such that P_i is not equal to a constant, and P_i involves less than d variables.

For $a \in A$, let a' be obtained from a by changing the value of every variable in the part P_i. By the definition of parts, the only constraints imposed by the projection which involve these variables are those equating literals among these variables. If a satisfies such a relation $w_1 = w_2$, then a' does as well. a' is a member of P, and thus of A.

From this, we obtain two points $a, a' \in A$ with Hamming distance $(a, a') < d$, contradicting the assumption about the minimum distance of d. ■

Theorem 3.4 *If C is a Σ_3^2 circuit computing $\mathrm{Ecc}_{\sqrt{2n}}$, then the size of C is at least $2^{n-\sqrt{n}\log n}$.*

Proof: To obtain a contradiction, we assume there exists a circuit C of size less than $2^{n-\sqrt{n}\log n}$. Since the number of codewords of $E_{\sqrt{2n}}$ is at least $2^{n-\sqrt{n/2}\log n}$, this implies that there exists 2–CNF subcircuit C_i which accepts more than $2^{\sqrt{n/2}\log n}$

To do so, we utilize some results relating to Vapnik-Chernovenkis dimension [30].

Definition 3.15 *Let A be a set of assignments over a set X of n variables. A set $X' \subseteq X$ is* shattered *if A restricted to X' has size $2^{|X'|}$; that is, all possible settings to X' appear in A. The set A has* VC-dimension *d if the largest shattered set of variables has size d.*

We now apply Sauer's Lemma [23].

Lemma 3.15 (Sauer) *If A is a family of subsets of an n element set, and A has VC-dimension at most d, then:*

$$|A| \leq \sum_{i=0}^{d} \binom{n}{i}.$$

By showing that there is a large shattered set of variables, we can show that the projection contained in A is large.

Lemma 3.16 *If F is a 2–CNF accepting a set A, then F accepts a projection of dimension at least*

$$\frac{\log(|A|)}{\log n}$$

.

Proof: If F accepts the set A, then by Lemma 3.15 there exists a set $X' \subset X$ of variables which are shattered by A. Furthermore, by turning the statement about $|A|$ into one about d, we see that $|X'|$ is at least as large as the largest value of d with

$$|A| \geq \sum_{i=0}^{d} \binom{n}{i}$$

In particular, this inequality is satisfied by $d = \log(|A|)/\log n$, so

$$|X'| \geq \frac{\log |A|}{\log n}$$

either the nodes t form cycles, contradicting the definition of H_F, or there is a path connecting some $s_1, s_2 \in S$, contradicting the fact that all settings to S yield satisfying assignments. Thus, the algorithm sets all $t \notin S$ without setting any $s \in S$.

After all nodes except S have been set, the remaining graph H consists of the nodes S and their complements, and by the above observation, there are no edges. Thus, every setting to these nodes of H produces a satisfying assignment. For each node $b \in H$, we include the constraints that define setting a node; that is, we include constraints equating all literals in $H(b)$. Prior to this, some nodes $t \notin S$ were set to 0 or 1. We also include those constraints in the definition of the projection P.

Since these constraints define a projection, and there are at least $2^{|S|}$ assignments which satisfy the constraints, then P is a projection of dimension at least $|S|$. ■

Lemma 3.14 *If A is a set of variables with the property that for every partial assignment α to A, F after α is substituted for A is satisfiable, then F is satisfiable on a projection of dimension $|A|$.*

Proof: If $w_1, w_2 \in H(b)$ for some node b, then by Lemma 3.9, the formula implies the constraint $w_1 = w_2$. Thus, for each node $b \in H$, $H(b)$ contains at most one variable in A, since otherwise not all assignments to A would satisfy the formula. This implies that there exists a set of nodes S with $|S| = |A|$ and with F satisfiable on every setting to S. The lemma follows by applying the previous lemma. ■

From this, we obtain a projection on which F accepts. However, this occurs trivially if F has only one satisfying assignment, which gives rise to a projection of dimension 0, so we need a stronger condition. We now show that if F accepts many inputs, the resulting projection has large dimension.

Definition 3.13 *A* projection *is a set of assignments defined by equations of the form*

$$w_i = 0, w_i = 1, w_i = w_j$$

where the w_i are literals.

Since these equations are linear if we view the Boolean-valued variables as elements of $GF(2)$, we can view a projection as the solution to a system of linear equations over $GF(2)^n$.

Definition 3.14 *The* dimension *of a projection $P(X)$ is the dimension of the set as an affine $GF(2)^n$ subspace.*

Because of this linear structure, this implies that the dimension is also equal to $\log |A|$, where A is the set of assignments in the projection.

Now, our analysis of the `2SATAlg` will locate a projection contained in the set of satisfying assignments of a satisfiable formula.

Lemma 3.13 *Let S be a set of nodes of H_F with the property that for all settings of S, F is satisfiable. Then, F is satisfiable on a projection of dimension $|S|$.*

Proof: The algorithm `2SATAlg` does not specify how the algorithm should choose among the nodes of outdegree 0 in H_F. Here, we will constrain those choices, not setting any node in S until all other nodes have been set. By monitoring the execution of `2SATAlg`, we obtain a set of constraints which define our projection P.

If H_F had any edges connecting $s_1, s_2 \in S$, then there would be some setting of S on which the formula was false. Thus, there are no such edges. We want the algorithm to set all nodes except those in S. Suppose that the algorithm reaches a state where it is unable to set a node not in S. It must be that for all $t \notin S$, t does not have outdegree 0. This would imply that all such t also have indegree greater than 0, by the duality of H_F. From this, we can conclude that

Proof: By Lemma 3.10, if the algorithm exits in the first stage, then F is unsatisfiable, so the algorithm is correct. If the algorithm does not exit there, there is no $b \in H_F$ with $v, \bar{v} \in H(b)$. We now show that with this assumption, each time the algorithm sets some $b \in H_F$, it does not falsify any clause of F. Since at the end of the algorithm, there are no unset variables, the assignment produced must satisfy F.

First, because `2SATAlg` did not terminate in the first stage, we know that setting b and $\bar{b}$ is consistent, meaning that no variable is set to both true and false.

Because of the self-dual nature of G_F and thus H_F, $H(b)$ and $H(\bar{b})$ involve the same set of variables. Each clause C of F has 0, 1, or 2 variables in common with $H(b)$. If it has none, then it cannot be falsified by setting b. If it has one, we use the fact that b has outdegree 0. If there were any clauses of the form $(\bar{w}_1 \vee w_2)$ with $w_1 \in H(b)$ and $w_2 \notin H(b)$, then there would be an edge $w_1 \rightarrow \bar{w}_2$ in G_F, and thus H_F could not have outdegree 0. By symmetry, $\bar{b}$ has indegree 0, and the same argument holds. Thus, every clause with one literal in $H(b)$ is made true by setting b and $\bar{b}$. Finally, in the case that both variables in the clause C appear in $H(b)$, the only clauses that can be made false are of the form $\bar{w}_1 \vee \bar{w}_2$. If such a clause existed, then this would yield a implications from w_1 to $\bar{w}_2$ and from w_2 to $\bar{w}_1$ in G_F. Since $w_1, w_2 \in H(b)$ and $\bar{w}_1, \bar{w}_2 \in H(\bar{b})$, this implies that $\bar{w}_1 \in H(b)$, which would have caused the algorithm to exit in the first stage.

■

As an immediate consequence of the fact that this algorithm finds a satisfying assignment if one exists, we obtain

Lemma 3.12 *A 2–CNF F is satisfiable if and only if for all variable v and nodes $b \in H_F$, $H(b)$ does not contain both v and $\bar{v}$.*

We now show that the behavior of this algorithm also gives rise to certain structured subsets of satisfying assignments.

Because of the way in which the edges of G_F are created, the graph has a specific, symmetric structure.

Definition 3.10 *Given a graph G whose nodes are labeled by literals, the* dual *of G is obtained from G by, for all $i \leq n$, switching the labels of the nodes x_i and $\bar{x}_i$. If the dual of G is identical to G, G is* self-dual.

G_F is clearly self dual, since its edges are created in pairs, and interchanging x_i and $\bar{x}_i$ preserves each pair of edges. H_F also inherits this self-dual structure from G_F, and we will want to identify the "dual" node associated with each $b \in H_F$.

Definition 3.11 *If $b \in H_F$, let $\bar{b} \in H_F$ with $H(\bar{b}) = \{\bar{v} | v \in H(b)\}$.*

We now express our operations on variables in terms of operations on the nodes of H_F.

Definition 3.12 *A node $b \in H_F$ is* set *to a Boolean value y by a partial assignment to the variables which makes all literals in $H(b)$ have the truth value y.*

Then we can find satisfying assignments of 2–CNFs with the following algorithm:

```
2SATAlg(2–CNF formula F)
  if there exists a b ∈ H_F and variable v
    such that v, v̄ ∈ H(b)
    then output ("unsatisfiable"); exit
  H = H_F
  while H is nonempty
    let b ∈ H be a node of outdegree 0
    set b to TRUE
    set b̄ to FALSE
    delete b and b̄ from H
```

Lemma 3.11 *If F is satisfiable,* `2SATAlg` *produces a satisfying assignment.*

Definition 3.6 *Let $scc(l)$ be the set of literals which are strongly connected to the literal l in the graph G.*

Lemma 3.9 *If two literals v_1 and v_2 lie in the same strongly connected component of G_F, then F implies that $v_1 = v_2$.*

Proof: By the definitions of G_F and strong connectedness, there exist two sequences of clauses:

$$(v_1 \vee \bar{a}_1), (a_1 \vee \bar{a}_2), \ldots, (a_i \vee \bar{v}_2)$$

and

$$(v_2 \vee \bar{b}_1), (b_1 \vee \bar{b}_2), \ldots, (b_i \vee \bar{v}_1)$$

which imply the clauses $v_1 \vee \bar{v}_2$ and $v_2 \vee \bar{v}_1$, and thus $v_1 = v_2$, by resolution.

■

From this, we immediately obtain

Lemma 3.10 *If for any variable v, $\bar{v} \in scc(v)$, then F is unsatisfiable.*

In the other direction, if we have a clause graph G_F where no strong component contains both a variable and its negation, then we can easily find a satisfying assignment of F.

Definition 3.7 *Let $TC(G)$ denote the transitive closure of the graph G.*

Definition 3.8 *Let* $\text{imply}(l)$ *be the set of literals which imply a literal l in the graph $TC(G_F)$. Let* $\text{impliedby}(l)$ *be the set of literals which are implied by a literal l in the graph $TC(G_F)$.*

Definition 3.9 *Let H_F be the graph whose nodes correspond to strongly connected components of G_F. For each $b \in H_F$, let $H(b)$ be the set of literals in the strongly connected component of G_F corresponding to b. There is an edge $(b_1 \rightarrow b_2)$ in H_F if and only if there exists $v_1 \in H(b_1)$ and $v_2 \in H(b_2)$ with $v_2 \in \text{imply}(v_1)$.*

Lemma 3.8 *There exists a Σ_3 circuit computing* Parity *of size* $(1+o(1))n^{1/4}2^{\sqrt{n}}$.

Proof: Parity on m inputs can be computed by Σ_2 and Π_2 circuits of size $2^{m-1}+1$. Partition the inputs into $\sqrt{n}+\frac{1}{4}\log n+o(1)$ groups of size $\sqrt{n}-\frac{1}{4}\log n$, and compute the parity within each group using a Π_2 circuit. This requires $n^{-1/4}2^{\sqrt{n}-1}+1$ gates per group, or at most $(\frac{1}{2}-o(1))n^{1/4}2^{\sqrt{n}}$ gates in all.

Then, compute the parity of these groups using a Σ_2 circuit. Since the number of inputs is now $\sqrt{n}+\frac{1}{4}\log n+o(1)$, the size of this circuit is also $(\frac{1}{2}+o(1))n^{1/4}2^{\sqrt{n}}$.

This yields a depth 4 circuit, but since the middle two levels of the circuit consist of AND gates, these two levels can be collapsed by increasing the fanin of the AND gates. This yields a depth 3 circuit of size $(1+o(1))n^{1/4}2^{\sqrt{n}}$ ∎

3.3 Bottom Fanin 2

Following our intuition that fast algorithms for satisfiability problems exist because of limitations on the expressive power of the formula, the fact that 2–SAT can be solved in polynomial time suggests that the algorithms which do so capture the limitations of 2–CNFs very effectively. By studying these algorithms, we expose a property of sets accepted by 2–CNFs, which we will exploit to prove strong lower bounds on Σ_3^2 circuits. First, however, we develop some ideas necessary to present a polynomial time algorithm for 2–SAT.

Definition 3.5 *For a 2–CNF F, the associated* clause graph *G_F is a directed graph defined as follows:*

1. *For every variable x in F, G_F contains two nodes $x, \bar{x}$.*
2. *For every clause $(a \vee b)$ in F, G_F contains the two directed edges $\bar{a} \rightarrow b$ and $\bar{b} \rightarrow a$.*

Let

$$T_a = \sum_{l=1}^{n} \frac{N_l(a)}{n} \frac{2^l}{l}$$

denote the inner summation.

Examining T_a, we see a familiar pattern: Let $C_a(v)$ be the length of the critical clause for v at a, and $L(r) = 2^r/r$. T_a is then the expectation over variables of the function $L(C_x(v))$. Since this function $L(r)$ is concave, we can lower bound the expectation of the function by the function of the expectation.

Define

$$\delta_a = \sum_{l=1}^{n} \frac{lN_l(a)}{n}$$

to be the expectation of $C_a(v)$. From the discussion above, we have $T_a \geq 2^{\delta_a}/\delta_a$.

Now, by the Schwartz Inequality,

$$\left(\sum_{l=1}^{n} N_l(a)/l\right)\left(\sum_{l=1}^{n} lN_l(a)\right) \geq \left(\sum_{l=1}^{n} N_l(a)\right)^2 = n^2$$

Since the first term is $\mathrm{wt}(a)$ and $a \in S_2$, the first term is at most τ. Thus, the second term $\sum_{l=1}^{n} lN_l(a) \geq n^2/\tau$, and $\delta_a \geq n/\tau$.

Since the function $2^l/l$ is monotone for $l \geq 2$, the inner sum $T_a \geq \tau 2^{n/\tau}/n$ for sufficiently large n. Thus, the the number of level one OR gates must be at least $|S_2|\tau 2^{-n+n/\tau}$.

The total number of gates is at least

$$\begin{aligned} |S_1|2^{\tau-n} + |S_2|\tau 2^{-n+n/\tau} &= 2^{-n}(|S_1|n^{1/4}2^{\sqrt{n}} + |S_2|(\sqrt{n} + \frac{1}{4}\log n)n^{-1/4}2^{\sqrt{n}}) \\ &\geq 2^{-n}(|S_1|n^{1/4}2^{\sqrt{n}} + |S_2|n^{1/4}(1+o(1))2^{\sqrt{n}}) \end{aligned}$$

Finally, from the constraint $|S_1| + |S_2| = 2^{n-1}$, we obtain the lower bound:

$$\frac{1}{2}n^{1/4}2^{\sqrt{n}}(1+o(1))$$

■

This lower bound is quite close to the upper bound for the natural divide-and-conquer construction

since d is sufficiently small. Now, applying the fact that $|S| = |S_1| + |S_2|$, we obtain the desired lower bound. ■

We now use the same technique to prove a lower bound on Parity. Since our goal is to obtain a tight bound, we will need to modify the analysis above slightly. Previously, we worked with the critical clause trees constructed for different variables for a given satisfying assignment and were able to argue that the tree for one of the variables contained at least one long clause. Here, we will work directly with the simpler critical clauses, and be able to consider whether the critical clause for each variable is long or short, giving us much tighter bounds.

Lemma 3.7 *If C is a Σ_3 circuit computing* Parity*, then the size of C is at least* $\frac{1}{2}(1 + o(1))n^{1/4}2^{\sqrt{n}}$.

Proof:

As before, we partition S into S_1 and S_2 depending on $\mathrm{wt}(a)$. Here, however, we choose $\tau = \sqrt{n} + \frac{1}{4}\log n$. Repeating the argument above, the number of CNFs required, and thus the number of gates at level 2 in C, is at least $|S_1|2^{\tau - n}$.

In the other case, we now must be more careful. For each a, we fix one critical clause per variable, and ignore the contributions from all other critical clauses. Let $N_l(a)$ be the number of critical clauses for a of length l. Because a is n–isolated and because of our simplification, $\sum_{l=1}^{n} N_l(a) = n$ for every $a \in S$. Now, a clause of length l is critical for at most $l2^{n-l}$ *pairs* (a, i) with $a \in S$ and $i \in [1, \ldots, n]$ indicating a specific variable. This implies that the number of clauses is at least

$$\begin{aligned}\sum_{l=1}^{n}\sum_{a\in S_2}\frac{N_l(a)}{l2^{n-l}} &= \sum_{a\in S_2}\sum_{l=1}^{n}\frac{N_l(a)}{l2^{n-l}} \\ &= \sum_{a\in S_2} n2^{-n}\sum_{l=1}^{n}\frac{N_l(a)}{n}\frac{2^l}{l}\end{aligned}$$

with small Hamming distance from a. Since a is distance–$\log n$ isolated from all other satisfying assignments, C_a is false on y, and $\texttt{ConstructTree}(C_a, a, v, s)$ finds some clause falsified by y and makes use of it in the resolution steps. If every such falsified clause had length at most τ, then we would have

$$\Pr[v \in \text{Forced}(C'_a, \pi, a)] \geq R_\tau - o(1) = \frac{\pi^2}{6}\frac{1}{\tau} - o(1) = \tau/n - o(1)$$

by Lemma 2.24, Lemma 2.23, and the definition of τ. This would imply that $\text{wt}(a) \leq \tau$. Thus, if $\text{wt}(a) > \tau$, then there must exist at least one variable v such that $\texttt{ConstructTree}(C_a, a, v, s)$ considers some clause D which has length greater than τ. We call such clauses *long*.

We now argue that each long clause can only appear in a small number of critical clause trees produced by `ConstructTree`. If a clause C_i is used in $\texttt{ConstructTree}(C_a, a, v, d)$, then C_i is false on some point within Hamming distance d of a. Given a and the specific set of at most d variables, this determines the orientation of all variables in C_i. This implies that if $|C_i| = l$, then C_i is appears in at most

$$\binom{l}{d}2^{n-l}$$

trees. Since $l \geq \tau$, this is at most

$$\binom{n}{d}2^{n-\tau}$$

Since there a long clause for each $a \in S_2$, the circuit C must contain at least

$$\frac{|S_2|}{\binom{n}{d}2^{n-\tau}}$$

clauses (ie, gates at level one).

From this, we derive that the total size of the circuit is at least

$$|S_1|2^{\tau-n} + \frac{|S_2|}{\binom{n}{d}2^{n-\tau}} = |S_1|2^{\tau-n} + |S_2|2^{-n+\tau} - 2^{o(n)}$$

Now, we generalize these results for Σ_3^k circuits to Σ_3 circuits of arbitrary fanin. To do so, we make use of a standard observation: If the fanin of an OR-gate is large, then it almost always outputs 1, and so only affects the computation on a small number of inputs. Here, however, we reverse our usual order and prove the result for $\mathrm{Ecc}_{\log n}$ first. The reason for this is that for Parity, we will attempt to prove very tight bounds, which will require more complicated accounting. In proving the bound on $\mathrm{Ecc}_{\log n}$, we will introduce the technique we will use in proving the bound on Parity.

Theorem 3.3 *If C is a Σ_3 circuit computing $\mathrm{Ecc}_{\log n}$, then the size of C is at least $2^{1.282\sqrt{n}}$.*

Proof:

Let C be a Σ_3 circuit computing $\mathrm{Ecc}_{\log n}$, let $C = \vee C_i$, and let S be the set of inputs accepted by C. For each input a accepted by C, we associate a CNF subcircuit $C_a \in \{C_i\}$. We then define the $\mathrm{wt}(a)$ to be the weight of a with respect to the CNF C_a.

Let $c = \sqrt{\frac{\pi^2}{6}} \geq 1.282$, and let $\tau = c\sqrt{n}$

We partition S into two sets: $S_1 = \{a | \mathrm{wt}(a) \leq \tau\}$ and $S_2 = \{a | \mathrm{wt}(a) > \tau\}$. We now show that if S_1 is large, there are many CNFs, and thus many gates at level 2, and that if S_2 is large, there are many clauses among the $\{C_i\}$, and thus many gates at level 1. Since $|S| = |S_1| + |S_2|$, at least one of these must occur. By Lemma 3.4, each CNF can accept at most $2^{n-\tau}$ inputs with weight at most τ. This implies that the number of CNFs required, and thus the number of gates at level 2 in C, is at least $|S_1|2^{\tau - n}$.

Now, in the other direction, we must argue that if S_2 is large, then there must be many clauses. Consider an input a accepted by the circuit, let $s = \tau^{\log n}$, and let $C'_a = \texttt{Resolve}(C_a, s)$. We now examine the execution of $\texttt{ConstructTree}(C_a, a, v, s)$. During each iteration i, the procedure changes the values of the variables along a path from root to leaf, producing an assignment y

Lemma 3.5 *If F is a k–CNF which accepts a subset of* Parity*, then F accepts at most $2^{(1-1/k)n}$ points.*

Proof: If F accepts a subset of Parity, then every $a \in \mathrm{Sat}(F)$ is n–isolated. Therefore, by Lemma 2.7, $\mathrm{wt}(a) \geq n/k$. By Lemma, 3.4, this implies that $|\mathrm{Sat}(F)| \leq 2^{(1-1/k)n}$. ■

Theorem 3.1 *If C is a Σ_3^k circuit computing* Parity*, then the size of C is at least $\frac{1}{2}2^{n/k}$.*

Proof: Assume for the purposes of contradiction that C computes Parity with size less than $\frac{1}{2}2^{n/k}$. For Parity, $|f^{-1}(1)| = 2^{n-1}$. Thus, by Lemma 3.1, there is a k–CNF accepting a subset of Parity of size greater than $2^{(1-1/k)n}$, contradicting Lemma 3.5. ■

Lemma 3.6 *If F is a k–CNF which accepts a subset of* $\mathrm{Ecc}_{\log n}$*, then F accepts at most $2^{(1-R_k+o(1))n}$ points.*

Proof: If F accepts a subset of $\mathrm{Ecc}_{\log n}$, then every $a \in \mathrm{Sat}(F)$ is distance–$\log n$ isolated with respect to $\mathrm{Sat}(F)$. Therefore, by Lemma 2.25, $\mathrm{wt}(a) \geq (R_k - o(1))n$. By Lemma 3.4, this implies that $|\mathrm{Sat}(F)| \leq 2^{(1-R_k+o(1))n}$. ■

Theorem 3.2 *If C is a Σ_3^k circuit computing* $\mathrm{Ecc}_{\log n}$*, then the size of C is at least $2^{(R_k-o(1))n}$.*

Proof: Assume for the purposes of contradiction that C computes $\mathrm{Ecc}_{\log n}$ with size less than $2^{(1-R_k+o(1))n}$. By the definition of $\mathrm{Ecc}_{\log n}$, $|f^{-1}(1)| = 2^{(1-o(1))n}$. Thus, by Lemma 3.1, there is a k–CNF accepting a subset of $\mathrm{Ecc}_{\log n}$ of size greater than $2^{(1-R_k+o(1))n}$, contradicting Lemma 3.6. ■

As a function of k, R_k is greater than $1/k$, and by Lemma 2.23 approaches $\frac{\pi^2}{6(k-1)}$ for large k. Thus, these lower bounds on the size of Σ_3^k circuits for $\mathrm{Ecc}_{\log n}$ are stronger than those for Parity.

as there exists a non-leaf node b with $L(b) > 0$ and children b_1, b_2, increase the values of $L(b_1)$ and $L(b_2)$ by $L(b)/2$ and set $L(b) = 0$. This process conserves the sum of the $L(b)$ and terminates with $L(b) \geq 2^{-\mathrm{depth}(b)}$ if b is a leaf, and 0 otherwise. Thus, we can conclude that

$$\sum_{a \in S} 2^{-l_a} \leq 1$$

Then, the result follows by algebraic manipulation:

$$\begin{aligned} l - \log |S| &= \sum_{a \in S} \frac{1}{|S|}(l_a - \log |S|) \\ &= -\sum_{a \in S} \frac{1}{|S|}(\log 2^{-l_a} + \log |S|) \\ &= -\sum_{a \in S} \frac{1}{|S|} \log(|S| 2^{-l_a}) \\ &\geq -\log(\sum_{a \in S} 2^{-l_a}) \\ &\geq 0. \end{aligned}$$

where the penultimate inequality follows from the convexity of the logarithm function. Hence, $|S| \leq 2^l$. ∎

By combining this with the previous lemma, we obtain a relationship between the number and weight of the satisfying assignments of F.

Lemma 3.4 *For any CNF F, the number of satisfying assignments a with* $\mathrm{wt}(a) \geq \tau$ *is at most* $2^{n-\tau}$.

Proof: Let S be the set of satisfying assignments a with $\mathrm{wt}(a) \geq \tau$. By Lemma 3.2, there is a prefix-free encoding Φ_F such that

$$\mathbf{E}_{a \in S}[|\Phi_F(a)|] \leq n - \tau$$

.

By Lemma 3.3, this implies that $|S| \leq 2^{n-\tau}$. ∎

The lower bounds on Σ_3^k circuits follow immediately.

For each $a \in S$, we define $\Phi_F(a)$ by listing the values of the variables not in Forced(F, π, a) in the order in which they occur in π. To show that this is an encoding (ie, a one-to-one function), we show that it can be decoded uniquely. Given $\Phi(a)$, we can retrieve a by running `Modify`(F, π, b) for an arbitrary b. However, whenever `Modify` examines some bit of b, we substitute the next unread bit from $\Phi(a)$. This produces exactly the same execution as `Modify`(F, π, a), and thus outputs a.

Since the expected number of forced variables is at least wt(a), we have

$$\mathbf{E}_{a \in S}[|\Phi_F(a)|] \leq n - W$$

■

From the fact that each satisfying assignment can be encoded using a small number of bits, we wish to conclude that the set of satisfying assignments is not too large. This is captured by a standard inequality used in information theory due to Kraft [14].

Lemma 3.3 (Kraft) *If $\Phi : S \to \{0,1\}^*$ is a prefix free encoding with average code length l, then $|S| \leq 2^l$.*

For completeness, we include a proof of this fact.

Proof: Let $l_a = |\Phi(a)|$. By definition,

$$l = \frac{1}{|S|} \sum_{a \in S} l_a$$

Since Φ is one-to-one and prefix free, we can associate with Φ a binary tree T so that the leaves of T correspond to the codewords $\Phi(a), a \in S$. Furthermore, if for each node we label the edge to its left child with 0 and its right child with 1, then we can ensure that the path from the root to a leaf a follows the encoding $\Phi(a)$.

We first prove that $\sum_{a \in S} 2^{-l_a} \leq 1$. To do so, consider the following process: Begin with $L(b) = 1$ if b is the root node, and 0 otherwise. Then, as long

circuits with widely varying bottom fanin, we will define the weight of a satisfying assignment, which generalizes the probability that a variable is forced.

Definition 3.4 *Given a CNF F and $a \in \mathrm{Sat}(F)$, let*

$$\mathrm{wt}(a) = \mathbf{E}_\pi[|Forced(F,\pi,a)|] = \sum_{i=1}^{n} \Pr_\pi[x_i \in \mathrm{Forced}(F,\pi,a)]$$

For the case of distance–1 isolated satisfying assignments of k–CNFs, this is simply the familiar $(1-1/k)$ quantity which occurs in the analysis of `RandomUC`.

We now make precise the intuition that if the algorithm makes few decisions, but identifies all satisfying assignments, then the number of satisfying assignments is small. To do so, we view the decisions made by the algorithm as an encoding of the satisfying assignment.

Lemma 3.2 *If F is a CNF and $S \subset \mathrm{Sat}(F)$ with*

$$\mathbf{E}_{a\in S}[\mathrm{wt}(a)] \geq W$$

then there exists a one-to-one prefix-free encoding function

$$\Phi_F : S \rightarrow \{0,1\}^*$$

such that

$$\mathbf{E}_{a\in S}[|\Phi_F(a)|] \leq n - W$$

.

Proof: By the definition of $\mathrm{wt}(a)$, we have that

$$\mathbf{E}_{a\in S}\mathbf{E}_\pi[|Forced(F,\pi,a)|] \geq W$$

Then, we interchange the expectations, and choose some π which achieves at least the expectation. With respect to this π, we have

$$\mathbf{E}_{a\in S}[|Forced(F,\pi,a)|] \geq W$$

f prevent any k–CNF from accepting a large subset of $f^{-1}(1)$. In our search for hard functions, error-correcting codes will play a central role.

Definition 3.1 *$S \subseteq \{0,1\}^n$ is an* error-correcting code of length n with distance d *if every pair of elements $s_1, s_2 \in S, s_1 \neq s_2$ have Hamming distance $dist(s_1, s_2) \geq d$. The elements of S are called* codewords.

Such sets are important in communicating information over noisy channels. This distance property ensures that after a small number of bits of a codeword c have been altered, the resulting string is closer in Hamming distance to c than any other codeword, and thus can be uniquely decoded.

We will need two error-correcting codes with slightly different properties, which we will define now. First, however, we state a known result about a standard family of error-correcting codes, known as BCH codes.

Fact 3.1 *There exist error-correcting codes S of length n with distance d and*

$$|S| \geq 2^{n-(d-1)(\log(n+1))/2}$$

From this fact, we can show the existence of the two codes we will need.

Definition 3.2 *Let $E_{\log n}$ be an error-correcting code with distance $d = \log n$ and $2^{n-O(\log^2 n)}$ codewords. Let $\mathrm{Ecc}_{\log n}$ be the function with $\mathrm{Ecc}_{\log n}(a) = 1$ if and only if a is a codeword of $E_{\log n}$.*

Definition 3.3 *Let $E_{\sqrt{2n}}$ be an error-correcting code with distance $d = \sqrt{2n}$ and $2^{n-\sqrt{n/2}\log n}$ codewords. Let $\mathrm{Ecc}_{\sqrt{2n}}$ be the function with $\mathrm{Ecc}_{\sqrt{2n}}(a) = 1$ if and only if a is a codeword of $E_{\sqrt{2n}}$.*

3.2 Applying the Algorithms

Here, we make use of the machinery developed in analyzing our satisfiability algorithms for k–CNFs. Because we will later want to consider more general

as making a sequence of decisions in order to arrive at a satisfying assignment. Instead of counting all assignments accepted by the CNF, we instead find a subset of the accepted assignments with useful properties. We first identify a set of decisions made by the algorithm so that no matter what choice the algorithm makes on those steps, it is still possible for it to find a satisfying assignment. By examining the algorithm closely, we will show that this set of accepted properties has a very regular structure. Furthermore, we relate this structure to the number of satisfying assignments, showing that if the formula has many satisfying assignments, then this set of decisions is large. From this, we conclude that if the formula has many satisfying assignments, then there exists a large structured subset of satisfying assignments. By selecting a function f with the property that f^{-1} contains no large structured subset, we obtain lower bounds on the circuit size of f.

3.1 Basics

The basic argument outlined above is called *top-down* since it works down (towards the inputs) from the output gate. The heart of this argument is the following lemma.

Lemma 3.1 *If C is a Σ_3^k circuit of size s computing f, then there exists a k–CNF formula F such that F accepts a subset of $f^{-1}(1)$ of size at least $\frac{|f^{-1}(1)|}{s}$.*

Proof: Since C is a Σ_3^k circuit, we can express C as $C = \bigvee C_i$, where each C_i is a k–CNF. Since the size of C is at most s, there are at most s subcircuits C_i.

If C computes the function f, each C_i accepts a subset of $f^{-1}(1)$, since otherwise C outputs 1 on some point in $f^{-1}(0)$. Also, for every $x \in f^{-1}(1)$, there exists at least one subcircuit C_i which accepts x. By averaging, this implies that some C_i accepts a subset of $f^{-1}(1)$ of size at least $\frac{f^{-1}(1)}{s}$. ∎

In addition to devising characterizations of the sets accepted by k–CNFs, we will need to find functions f which are hard, in the sense that the properties of

assignments with certain properties, like isolation, and related this to the number of satisfying assignments. Here, we make use of a similar approach. We make use of two different approaches for bounding the size of such a subset by examining the properties of satisfiability algorithms.

We begin by showing how the analysis of the algorithms `RandomUC` and `ResolveSAT` can be applied in this fashion to yield lower bounds on the size of Σ_3^k circuits computing parity and certain error-correcting codes, respectively. Then, we generalize this to obtain improved lower bounds on Σ_3^k circuits of arbitrary bottom fanin. These bounds use the following high-level argument.

We can view the procedure `Modify`(F, π, b) as identifying a satisfying assignment by making a series of decisions: When it reads a bit from b, it 'decides' whether to set that variable to true or false; when the value is set by a unit clause, no decision is necessary. Varying over the set of possible decisions, `Modify` is capable of generating every satisfying assignment of F. In analyzing `RandomUC` and `ResolveSAT`, we proved that if a satisfying assignment a has certain isolation properties, then the number of variables forced is large, and thus the number of decisions `Modify` makes in finding a is small. If either algorithm reaches all sufficiently isolated satisfying assignments, but does so making few decisions, then there simply cannot be many isolated satisfying assignments. From this, we will be able to conclude that no k–CNF accepts many inputs that are well-isolated from other points in $f^{-1}(1)$. By considering functions f where every true input has this isolation property, we obtain a lower bound on the size of Σ_3^k circuits computing f.

The other lower bound makes use of the properties of 2–CNFs. Polynomial time algorithms for the 2–SAT problem are known. Now, we will use ideas from these algorithms to derive strong exponential lower bounds on the size of Σ_3^2 circuits computing certain functions based on error-correcting codes. This proof makes use of a somewhat different approach. Once again, we view the algorithm

Chapter 3

Circuits

In proving upper bounds on the running time of `RandomUC` and `ResolveSAT`, we made use of relationships between a k–CNF formula F and the set of satisfying assignments $\mathrm{Sat}(F)$. These relationships imply some limitations on sets which can be accepted by k–CNFs. In this section, we exploit these limitations to prove lower bounds on the size of depth–3 circuits computing certain explicit functions. To do so, we associate the integers 1 and 0 with the Boolean values true and false, respectively.

The proofs will follow a simple framework: A Σ_3^k circuit is the OR of a set of k–CNF formulae. Each such subcircuit accepts a subset of the points on which the circuit outputs 1, and every point on which the circuit outputs 1 is covered by some subcircuit. If the circuit is not too large, then by averaging, there must exist a CNF which accepts a large subset of the points accepted by the circuit. For a Boolean function f, we will first show that no k–CNF accepts a large subset of $f^{-1}(1)$, and then conclude that there is no small Σ_3^k circuit computing f.

To carry out this plan, we must have a method for obtaining upper bounds on the maximum size of the subset of $f^{-1}(1)$ which can be accepted by a k–CNF. It is here that techniques from the analysis of satisfiability algorithms will be useful. In our analysis of `RandomUC` and `ResolveSAT`, we considered sets of satisfying

k	Previous	`RandomUC`	`ResolveSAT`(analytic)	`ResolveSAT`(computational)
3	$2^{0.582n}$	$2^{0.667n}$	$2^{0.533n}$	$2^{0.446n}$
4	$2^{0.879n}$	$2^{0.750n}$	$2^{0.581n}$	$2^{0.562n}$
5	$2^{0.947n}$	$2^{0.800n}$	$2^{0.649n}$	
6	$2^{0.975n}$	$2^{0.833n}$	$2^{0.711n}$	

$H_i \leq H_{i+1}$, and the last $H_i = 1$. We can also define an analogous probability distribution vector $\vec{h}$ by $h_i = H_i - H_{i-1}$. $h(v_i)$ then represents the probability that a defining variable lands in the ith bin, and the distribution within each bin is uniform.

Given such a function, $\beta(H)$ is easily determined using an entropy calculation similar to the one used in Lemma 2.28. For each discrete value of r, we can compute the probability that a nondefining variable is forced in each of the two worst-case cases in Lemma 2.35, given that the placement is H–good. By taking the smaller of these two values, we obtain a lower bound for the probability that a nondefining variable is forced, given that it is placed in a given bin. Summing over all bins (the analogue of integrating over all values of r) yields the probability that this variable is forced. From this, by accounting for the effects of discretization on our previous calculations, we can obtain an upper bound on the running time of `ResolveSAT`.

Finally, we treat the vector $\vec{h}$ simply as a list of parameters to the function described above which estimates the running time of `ResolveSAT`. Then, we apply numerical optimization routines available in the Matlab Optimization Toolkit to find good value of these parameters. The Matlab code for this search is included in the Appendix.

From this search, we obtain the second half of our results on `ResolveSAT`.

Theorem 2.5 *For satisfiable 3–CNFs F,* `ResolveSAT`(F, s, I) *with* $I = 2^{0.446n}$ *finds a satisfying assignment with high probability in time* $2^{0.446n+o(n)}$.

For satisfiable 4–CNFs F, `ResolveSAT`(F, s, I) *with* $I = 2^{0.562n}$ *finds a satisfying assignment with high probability in time* $2^{0.562n+o(n)}$.

Our results on satisfiability algorithms are summarized in the following table. All running times given are modulo factors of $2^{o(n)}$.

function H, we compute the probability that σ is H–good, and a lower bound on the probability that a nondefining variable is forced given this fact. From this, we obtain a lower bound on μ in the same way as before. We then perform a computer search which chooses a distribution function from this class with the goal of optimizing the bound on μ that it yields.

Recall the definition of T^*: T^* is a tree of depth 1 where the root is labeled by v and has $k-1$ children labeled by defining nodes.

Lemma 2.35 *Let $T = \texttt{NewConstructTree}(F, a, v, d)$, and let H be a distribution function. For every $r \in [0,1]$, $Q_T \geq \min(R_k - \varepsilon, Q_{T^*})$, where all probabilities are conditioned on the event that σ is $H(r)$–good.*

Proof:

If $H(r) \leq R_k(r)$, then consider the following rule:

Let b be a leaf labeled by a defining variable. Relabel b with a nondefining variable, and give b $k-1$ children labeled by defining variables. Before, the probability that this subtree was weakly cut was $H(r)$; after, it is $r+(1-r)H(r)^{k-1}$. This implies that $Q_T(r)$ is not increased as long as $f_k^r(H(r)) = (r+(1-r)H(r))^{k-1} - H(r) \leq 0$. Since $R_k(r)$ is the minimal root of that expression, and $f_k^r(x)$ is increasing in x, this is true if $H(r) \leq R_k(r)$. By repeating this process, we obtain a tree that has no defining variables at depth d or less.

If $H(r) \geq R_k(r)$, then we run the process in reverse. Beginning with the tree T^*, we apply the transformation rule repeatedly to obtain a tree isomorphic to T. In this case, the transformation rule does not decrease the probability of T^*, so $Q_T \geq Q_{T^*}$.

■

With this lemma, our search can be described simply: First, r and Δ are discretized uniformly from $[0,1]$. In our searches, we used 100 steps for r and 500 for Δ, which we also think of as bins representing ranges for each quantity. A piecewise linear distribution function H is represented by a vector $\vec{H}$ with $0 \leq H_i \leq 1$,

By Lemma 2.11

$$\mu(F_s) \geq \min_{a \in \mathrm{Sat}(F_s)} \mu_a(F_s|P(a))$$

Both β_{H^*} and R_k approach 0 as k increases, by Lemma 2.34 and Proposition 2.1, respectively. Thus, if $\chi \geq 0$ for some k, then this is true for all $k' > k$. Looking at the table, this is true for $k = 5$. Then, by Lemma 2.32, this implies that for all $a \in \mathrm{Sat}(F)$,

$$\mu_a(F_s|P(a)) \geq 2^{-(1-R_k)n+o(n)}$$

■

For $k \geq 5$, we obtain the same results as in the well-isolated case, which represent the best we can hope for given this technique, since the formula could have exactly one satisfying assignment. For $k = 3, 4$, however, the bounds we obtain here are weaker. Because $\chi < 0$, we make use of a tradeoff between the bounds of Lemma 2.31 and 2.26. The first conditions on σ having a very specific property (being $H^*(r)$–good), while the second relies only on the number of defining variables and makes no assumptions about σ. Thus, in trying to improve the bounds for the cases $k = 3, 4$, we investigate a broader range of distribution functions H to allow for intermediate levels of conditioning on σ. We then search for distribution functions H which give rise to better bounds. Lacking an explicit characterization of the properties we require, we do this in a computational fashion.

In order to prove bounds for the various distribution functions we consider, we first require a method that allows us, for an arbitrary distribution function H, to compute Q_T given that σ is $H(r)$-good. To do this, we prove a lemma which reduces the computation of $Q_T(r)$ for an arbitrary tree constructed by `NewConstructTree` to one of two worst case trees, depending on the properties of $H(r)$. Then, we discretize the values of $r \in [0, 1]$ and consider the class of distribution functions which are linear on each discrete interval. For each distribution

$$\begin{aligned} A &= (k-1)g_k - 2Hf_k \\ &= (k-1) + (k-3)H + (k-5)H^2 + \ldots + -(k-3)H^{k-2} \\ &= (k-1) + (k-3)(H - H^{k-2}) + (k-5)(H^2 - H^{k-3} + \ldots \\ &\quad \ldots + (k - 2\lfloor k/2 \rfloor + 1)(H^{\lfloor k/2 \rfloor} - H^{\lceil k/2 \rceil}) \\ &\geq k-1 \end{aligned}$$

while $B \leq k-1$ follows from

$$(k-1)g_k \geq g_k^2 \geq f_k \geq f_k H^k$$

Thus $A \geq B$, $h_k(H) - h_{k+1}(H) \geq 0$, and the lemma holds. ■

From this, we can calculate the value of the bound obtained in Lemma 2.32 for small values of k. Let coeff be the logarithm of the bound obtained in the lemma, divided by $-n$. This represents the coefficient in front of n in the exponent of the running time, using our standard relationship between the running time and $1/\mu$.

k	β_{H^*}	R_k	χ	coeff	
3	1.115	0.614	-0.729	0.533	$(= 0.387 + 0.729\Delta')$
4	0.666	0.445	-0.111	0.581	$(= 0.556 + 0.111\Delta')$
5	0.478	0.350	0.172	0.649	
6	0.373	0.288	0.339	0.711	

Since for $k = 5, 6$, $\chi > 0$, the monotonicity of β_{H^*} and R_k allow us to conclude that for all $k \geq 5$, `ResolveSAT` obtains the same bounds in the uniquely satisfiable case. We can state our main result for $k \geq 5$ now:

Theorem 2.4 *Let F be a satisfiable k–CNF formula. If $k \geq 5$ and $I \geq 2^{(R_k + o(1))n}$, then* `ResolveSAT`(F, s, I) *finds a satisfying assignment of F with high probability.*

Proof:

From Lemma 2.2 and the definition of `ResolveSAT`, it suffices to show that $\mu(F_s)$ is at least $2^{-(R_k + o(1))n}$.

obtained in Lemma 2.33. We will show that $\forall H \in (0,1), h_k(H) \geq h_{k+1}(H)$. From that, the lemma follows by noting that $\log x$ is an increasing function of x and then integrating.

First, define

$$\begin{aligned} f_k = f_k(H) &= 1 + 2H + 3H^2 + \ldots + (k-2)H^{k-3} \\ g_k = g_k(H) &= 1 + H + H^2 + \ldots + H^{k-2} \end{aligned}$$

With these definitions

$$h = \frac{(1-H)^2 g_k^2}{(1-H)^2 f_k} = \frac{g_k^2}{f_k}$$

Thus, we can show that $h_k \geq h_{k+1}$ by showing that $f_{k+1}g_k^2 - f_k g_{k+1}^2 \geq 0$

$$\begin{aligned} f_{k+1}g_k^2 - f_k g_{k+1}^2 &= f_{k+1}g_k^2 - f_k g_k^2 - f_k g_{k+1}^2 + f_k g_k^2 \\ &= g_k^2(f_{k+1} - f_k) - f_k(g_{k+1} - g_k^2) \\ &= (H^{k-2}g_k)((k-1)g_k - f_k(2h + \frac{H^k}{g_k})) \\ &= (H^{k-2}g_k)(A - B) \end{aligned}$$

where we define

$$\begin{aligned} A &= (k-1)g_k - 2Hf_k \\ B &= \frac{f_k H^k}{g_k} \end{aligned}$$

Since $(H^{k-2}g_k) > 0$ for $H \in (0,1)$, we need only show that $A \geq B$ to prove the lemma. To do so, we lower bound A and upper bound B by $k-1$.

Proof:

We once again integrate by first changing the variable of integration to simplify the computation. To do so, we invert H and express r as a function of H.

We will need the following equality: By the definition of H^* and $R_k(r)$, $H = H^*(r) = r + (1-r)R_k(r) = R_k(r)^{\frac{1}{k-1}}$.

$$\begin{aligned} r &= \frac{H - H^{k-1}}{1 - H^{k-1}} \\ h = \frac{dH}{dr} &= \frac{(1-H^{k-1})^2}{1 + (k-2)H^{k-1} - (k-1)H^{k-2}} \end{aligned}$$

where the second equality follows from the first by differentiating with respect to r and simplifying.

We can now write $h(r)dr$ as $\frac{dH}{dr}dr = dH$.

$$\begin{aligned} \beta_{H^*} &= \int_0^1 h(r)\log_2 h(r)dr \\ &= \int_0^1 \log_2 h(r)(h(r)dr) \\ &= \int_0^1 \log_2 h(H)dH \\ &= \int_0^1 \log_2 \frac{(1-H^{k-1})^2}{1 + (k-2)H^{k-1} - (k-1)H^{k-2}}dH \end{aligned}$$

■

With some additional calculations, we can also show the following

Lemma 2.34 *β_{H^*} decreases monotonically with k.*

Proof:

To do show, we show that $\forall r \in (0,1)$, $h_k(r) > h_{k+1}(r)$, using the expression

$$h_k(H) = \frac{(1-H^{k-1})^2}{1 + (k-2)H^{k-1} - (k-1)H^{k-2}}$$

and note that the worst case is when the quantities in the exponents are equal, which occurs when $\Delta = \Delta'$.

By combining this observation with previous lemmas, we obtain the following:

Lemma 2.32 *Let F be a k-CNF formula, a be a satisfying assignment, P a partition of* $\mathrm{Sat}(F)$ *into unique subcubes.*

Let d be $\omega(1)$ and $o(n/\log n)$.

1. *If $\Delta = o(1/\log n)$:*
$$\mu(F_s) \geq 2^{-(1-R_k)n+o(n)}$$

2. *If $\Delta = \Omega(1/\log n)$, then for any distribution function H satisfying $H(r) \geq r + (1-r)R_k(r)$ we have:*
$$\mu_a(F_s|P(a)) \geq 2^{-(1-R_k)n+o(n)} \text{ if } \chi \geq 0$$
and
$$\mu_a(F_s|P(a)) \geq 2^{-((1-R_k)-\chi\Delta'-\delta)n} \text{ if } \chi < 0$$

Now, we have reduced the problem of computing bounds on $\mu_a(F_s|P(a))$ to that of computing the quantities χ and β for our chosen distribution function H^*. We first show how β_{H^*} can be calculated. Then, we show that β_{H^*} is monotonically decreasing in k. Since R_k decreases in a similar fashion, this implies that once χ becomes nonnegative, we can use the bound which is independent of Δ' for all larger values of k.

The following expression for β_{H^*} can be derived:

Lemma 2.33
$$\beta_{H^*} = \int_0^1 \log_2 \frac{(1-H^{k-1})^2}{1+(k-2)H^{k-1}-(k-1)H^{k-2}} dH$$

Thus, the expected number of forced variables is at least $(R_k - \epsilon/4)(1 - \Delta)n$, and by concavity of the exponential function,

$$\mathbf{E}[2^{|\text{Forced}(F_s,\pi,a)|}] \geq 2^{\mathbf{E}[|\text{Forced}(F_s,\pi,a)|]}$$

where both expectations are conditioned on the event $\sigma \in , _{H^*,D}$. Thus, by applying Lemma 2.12, we obtain

$$\mu_a(G|P(a)) \geq 2^{-(1-\Delta)n+(R_k-\epsilon/4)(1-\Delta)n}$$

likewise conditioned on the event $\sigma \in , _{H^*,D}$. Now, we multiply this by the probability that $\sigma \in , _{H^*,D}$ and simplify.

$$\begin{aligned} \mu_a(F_s|P(a)) &= 2^{(1-\Delta)n+(R_k-\epsilon/4)(1-\Delta)n} \\ &\geq 2^{-((1-R_k)-(1-R_k-\beta_{H^*})\Delta-\epsilon)n} \end{aligned}$$

■

The coefficient of Δ in this expression is important in determining the bound on μ.

Definition 2.34 $\chi = (1 - R_k - \beta_{H^*})$

If $\chi \geq 0$, then Lemma 2.31 yields a bound independent of Δ; otherwise, the bound becomes worse as Δ increases. At the same time, the bound in Lemma 2.26 improves as Δ increases.

We define

Definition 2.35

$$\Delta' = \frac{R_k - 1/k}{\beta_H + R_k - 1/k}$$

sum over all such events of the difference of their probabilities is $o(1)$. Thus, the probability $Q_{k,d}(r)$ likewise changes by at most an additive $o(1)$ term. The lemma follows by integrating with respect to r. ■

Lemma 2.30 *For every $\epsilon > 0$, there exists some integer d such that if $s = s_{k,d} = o(|D|)$, then*

$$\Pr_{\sigma}[v \in \text{Forced}(F_s, \pi, a) | \sigma \in , _{H^*,D}] \geq R_k - \epsilon$$

Proof: Let $T = \texttt{NewConstructTree}(F, a, v, s)$. By 2.29, if we condition on the event $\sigma \in , _{H^*,D}$, then for all r, $Q_T(r) \geq Q_{k,d}(r) - o(1)$. Integrating these quantities, we obtain $Q_T \geq Q_{k,d} - o(1)$.

Now, choose d sufficiently large such that Lemma 2.21 implies that $Q_{k,d} \geq R_k - \epsilon/2$. Thus,

$$\Pr_{\sigma}[v \in \text{Forced}(F_s, \pi, a) | \sigma \in , _{H^*,D}] \geq Q_{k,d} - o(1) \geq R_k - \epsilon$$

■

Lemma 2.31 *If $\Delta = \Omega(1/\log n)$ and $d = o(\log n / \log\log n)$, then for every $\epsilon > 0$, there exists a integer d*

$$\mu_a(F_s, a | P(a)) = 2^{-((1-R_k)-(1-R_k-\beta_{H^*})\Delta-\epsilon)n}$$

Proof: By Lemma 2.28, for sufficiently large n:

$$\Pr[\sigma \in , _{H^*,D}] \geq 2^{(-\beta_{H^*}\Delta-\epsilon/4)n}$$

again, for sufficiently large d.

If $\Delta = \Omega(1/\log n)$ and $d = o(\log n / \log\log n)$, then $|D| = \Omega(n/\log n)$ and the total number of nodes in the tree, at most $(k-1)^d$, is $o(|D|)$. Thus, we can apply Lemma 2.30 to obtain

$$\Pr_{\sigma}[v \in \text{Forced}(F_s, \pi, a) | \sigma \in , _{H^*,D}] \geq R_k - \epsilon/4$$

for v. Now, however, we must also account for the fact that we are conditioning on $\sigma \in$, $_{H,D}$, rather than choosing σ uniformly.

This approach suggests a natural choice for $H(r)$: $H^*(r) = r + (1 - r)R_k(r)$. Then, in the critical clause tree for v, a leaf labeled by a defining variable will have the same probability of being weakly cut as a nondefining node with a large, full subtree below it.

Lemma 2.29 *Let* $T =$ `NewConstructTree`(F, a, v, s). *If* $s = s_{k,d} = o(\sqrt{|D|})$, *then the probability* $Q_T(r)$ *given that* $\sigma \in$, $_{H^*,D}$ *is at least* $Q_{k,d} - o(1)$ *as a function of* $|D|$.

Proof: We first consider the case that the probability that for all $v \in \mathrm{Def}(P, a)$, $\Pr[\sigma(v) < r] \geq H^*(r)$, as one would expect after conditioning on $\sigma \in$, $_{H^*,D}$. However, this is not true when the tree contains more than one defining variable, since learning that $\sigma(v) < r$ decreases slightly the probability that $\sigma(v') < r$ for subsequent defining variables. After dealing with this simpler case, we return to address this issue.

We begin by showing that, as in the case of Lemma 2.19, the probability that the tree is cut can be lower bounded inductively. Unlike the previous case, we must now account for the possibility that one of the subtrees being considered consists of a single defining node. Since $R_k(r) \geq Q_{k,d}(r)$ for all $r \in [0, 1]$, the same induction holds. Thus, the value of $Q_{k,d}(r)$ remains unchanged, and all subsequent results using that value still hold.

All that remains is to deal with the negative correlation between the events $\sigma(v) < r$ and $\sigma(v') < r$ for $v, v' \in \mathrm{Def}(P, a)$ which results from our definition of $H(r)$–good. Consider two cases: the first is our idealized one in which each such event has probability $H(r)$, and the second in the real one in which we account for this negative correlation. Since the total number of defining variables in the entire tree is $o(\sqrt{|D|})$, then the probability of any such event changes by at most $o(1/\sqrt{|D|})$. Since there are only $o(\sqrt{|D|})$ such events for a fixed value of r, the

Therefore, removing the floors from all M terms introduces an error term of size at most $\frac{M}{D}\log_2\frac{e}{D} = o(D)$.

After removing the floors, we can lower bound the the probability that σ is approximately $H(r)$–good by

$$\begin{aligned} 2^{-o(D)}2^{D\log_2 D - D\log_2 M - \sum_i g_i D\log_2 g_i D} &= 2^{-o(D)}2^{-D\log_2 M - \sum_i g_i D\log_2 g_i} \\ &= 2^{-o(D)}2^{-D\sum_i \frac{h_i}{M}\log_2 h_i} \end{aligned}$$

This final summation can be bounded for large D and M using calculus. Note that $h_i = \frac{g_i}{1/M}$ is the standard approximation to the slope at the point m_i. Considering increasing values of M, the sequence $\{\frac{1}{M}\sum_{i=0}^{M-1} h_i \log_2 h_i\}$ approaches the lower integral $\int_0^1 h(x)\log_2 h(x)dx$. Since h is bounded and nonzero except at finitely many points, then so is $h(x)\log_2 h(x)$. The integral of the function $h(x)\log_2 h(x)$ therefore exists and the sequence converges to this limit.

$$\lim_{M\to\infty}\frac{1}{M}\sum_i h_i \log_2 h_i = \int_0^1 H'(m_i)\log_2 H'(m_i)$$

From this, we obtain that

$$\Pr[\sigma \in \ , _{H,D}] \geq 2^{-\beta_H |D| - o(|D|)}$$

for sufficiently large D. ■

Our approach has three parts: First, we choose a distribution function $H(r)$. Second, we lower bound the probability that $v \in \text{Forced}(F_s, \pi, a)$ under the assumption that the underlying placement σ is chosen from an $H(r)$–good distribution for $\text{Def}(P, a)$. Finally, we compute the probability that $\sigma \in \ , _{H,D}$. From these quantities, we lower bound the probability $\mu_a(F|P(a))$.

To accomplish the second step of this process, we will need to prove an analogue of Lemma 2.19, which allowed us to relate the probability that a variable v is forced to an easily computed quantity of the the critical clause tree constructed

since there are at most M excess variables which must be accounted for and placed in the first bin.

Expressing each factorial using Stirling's approximation and simplifying, we obtain a lower bound of

$$M^{-M}\frac{\sqrt{2\pi D}}{\prod_i \sqrt{2\pi g_i D}} 2^{D\log_2 D - D\log_2 M - \sum_i \lfloor g_i D\rfloor \log_2 \lfloor g_i D\rfloor}$$

where the sums and products are taken over $0 \le i < M$, defining $0\log_2 0 = 0$. The lower order terms can be lower bounded by $M^{-M}\sqrt{2\pi D}\left(\frac{M}{2\pi D}\right)^{M/2}$, and since $M = \lfloor\sqrt{D}\rfloor$, this is $2^{-o(D)}$.

Next, we remove the floors from the quantities $\lfloor g_i D\rfloor \log\lfloor g_i D\rfloor$. First, we reexpress the summation in a more convenient form

$$\begin{aligned}\sum_i \lfloor g_i D\rfloor \log_2\lfloor g_i D\rfloor &= D\sum_i \frac{\lfloor g_i D\rfloor}{D}\log_2\frac{\lfloor g_i D\rfloor}{D} \\ &= D\sum_i \frac{\lfloor g_i D\rfloor}{D}\log_2 D + D\sum_i \frac{\lfloor g_i D\rfloor}{D}\log_2\frac{\lfloor g_i D\rfloor}{D}\end{aligned}$$

The first summation is at least $D\log_2 D - O(M\log_2 D)$. To bound the second, we define $\delta_i = g_i - \frac{\lfloor g_i D\rfloor}{D}$. We must show that the difference between $g_i\log_2 g_i$ and $(g_i-\delta_i)\log_2(g_i-\delta_i)$ is not too large. Let c satisfy $g_i = c\delta_i$. Next, note that $c \ge 1$: if $g_i \le 1/D$, then $g_i = \delta_i$ and $c = 1$. Otherwise, there exists some $g_i' < g_i$ with $g_i' = \delta_i$, implying that $c > 1$.

$$\begin{aligned}g_i\log_2 g_i - (g_i-\delta_i)\log_2(g_i-\delta_i) &= \log_2\frac{g_i^{g_i}}{(g_i-\delta_i)^{g_i-\delta_i}} \\ &= \log_2\frac{(c\delta_i)^{c\delta_i}}{((c-1)\delta_i)^{(c-1)\delta_i}} \\ &= \delta_i\log_2 c\delta_i\frac{1}{(1-1/c)^{c-1}}\end{aligned}$$

Now, $\delta_i \le c\delta_i = g_i \le 1$, by definition, and $1 \le \frac{1}{(1-1/c)^{c-1}} \le e$. Thus, the magnitude of this quantity is at most

$$\delta_i\log_2\delta_i \le \frac{1}{D}\log_2\frac{e}{D}$$

Definition 2.33

$$\begin{aligned} \beta_H &= \int_0^1 (h(r)) \log_2(h(r)) dr \\ \gamma_H(k) &= \int_0^1 H(r)^{k-1} dr \end{aligned}$$

It is not hard to derive a lower bound on the probability that a random placement is in , $_{H,D}$:

Lemma 2.28 *Let H be a distribution function and let h be its derivative. Then*

$$\Pr[\sigma \in , _{H,D}] \geq 2^{-\beta_H |D| - o(|D|)}$$

for sufficiently large D.

Proof: To simplify the expressions in this proof, we let $D = |D|$ in an abuse of notation. The idea of the proof is to fix some integer M, then discretize quantities in units of $1/M$. We show that as long as D is sufficiently large compared to M, as we increase M we obtain lower bounds approaching a quantity which is $2^{-\beta_H |D| - o(|D|)}$. The lemma then follows: to obtain an appropriately strong lower bound, we choose some large value of M, and the proof goes through for D sufficiently large compared to that value of M.

Consider a partition of $[0,1]$ into M equal-sized intervals, separated by the points $m_i = i/M$, $0 \leq i < M$. Also, define $g_i = H(m_{i+1}) - H(m_i)$ and $h_i = g_i M$ for $0 \leq i < M$. We also require that $D \geq M^2$.

With respect to this partition, we say that a placement σ is *approximately H–good for D* if for all i, the number of variables placed in bin i is $\lfloor g_i D \rfloor$, except that all excess variables are placed in the first bin.

We can easily lower bound the probability that σ is approximately $H(r)$–good.

$$M^{-M} \frac{D!}{\lfloor g_0 D \rfloor ! \cdots \lfloor g_{M-1} D \rfloor !} \left(\frac{1}{M}\right)^D$$

by Lemma 2.24.

$$\mathbf{E}[|\mathrm{Forced}(F_s, \pi, a)||E] \geq (R_k - \epsilon/4)|\mathrm{NDef}(P, a)| \geq (R_k - \epsilon/2)n$$

by linearity of expectation and the fact that Δ is small for sufficiently large n. Now, we apply the concavity of the exponential function to obtain

$$\mathbf{E}[2^{|\mathrm{Forced}(F_s,\pi,a)|}|E] \geq 2^{\mathbf{E}[|\mathrm{Forced}(F_s,\pi,a)||E]}$$

Then, we obtain

$$\mu_a(F_s|P(a)) \geq \mu_a(F_s|P(a), E)\Pr[E] \geq 2^{-(1-R_k+\epsilon/2)n}2^{-o(n)} \geq 2^{-(1-R_k+\epsilon)n}$$

for sufficiently large n.

■

If Δ is a constant, however, the probability that the π has all defining variables first is sufficiently small that we can not hope to obtain good bounds by this method. Instead, we must settle for something weaker: *many* defining variables happen *early* in the permutation. We first need some definitions which allow us to quantify this statement.

Definition 2.30 *$H(r)$ is a* distribution function *if H is continuous, nondecreasing function on $[0,1]$ with $H(0) = 0$ and $H(1) = 1$, is piecewise differentiable, and except at finitely many points, its derivative nonzero.*

Definition 2.31 *A placement σ is* $H(r)$–good for $\mathrm{Def}(P, a)$ *if, for all $r \in [0,1]$, the number of defining variables v with $\sigma(v) \leq r$ is at least $\lfloor H(r)|\mathrm{Def}(P, a)| \rfloor$.*

Definition 2.32 *, $_{H,D}$ is the set of all placements that are H–good on the set D of defining variables.*

A distribution function naturally defines a probability density function $h(r) = \frac{d}{dr}H(r)$. We define a constant β_H and a sequence $\gamma_H(k)$ associated with the function H,

tiplying by the probability we computed assuming this property by the probability that σ_d has this property, we obtain our bound on μ.

Following this approach, we prove the argument sketched above. If the number of defining variables is sufficiently small, then π will have this property will some nonnegligible probability. Then, by applying our analysis from the previous section, we obtain a good lower bound on μ.

Definition 2.29 $\Delta = |\mathrm{Def}(P,a)|/n$

Lemma 2.27 *If $\Delta = o(1/\log n)$, then for every $\epsilon > 0$, there exists an integer d so that*

$$\mu_a(F_s|P(a)) \geq 2^{-(1-R_k+\epsilon)n}$$

Proof: Let E_d be the event that $\forall v \in \mathrm{Def}(P,a)$, $\sigma(v) < \Delta$, and let E_n be the event that $\forall w \in \mathrm{NDef}(P,a)$, $\sigma(w) > \Delta$. Let $E = E_d \wedge E_n$.

$$\Pr[E_d] \geq n^{-\Delta n} \geq 2^{-o(n)}$$

since $\Delta \geq 1/n$, and

$$\Pr[E_n] \geq (1-\Delta)^n \geq 2^{-o(n)}$$

since $\Delta = o(1/\log n)$

Now, suppose we set variables according to a in the order determined by σ. Once a Δ–fraction of the variables have been set, all defining variables are set, but no nondefining variable has been set. a is then the unique satisfying assignment of F_s under this restriction. Furthermore, when we condition on E, the placement on the remaining variables remains uniformly distributed. Thus, we have returned to the case of a formula with only one satisfying assignment.

Let $v \in \mathrm{NDef}(P,a)$. Then, by choosing d sufficiently large and setting $s = s_{k,d}$, we obtain

$$\Pr[v \in \mathrm{Forced}(F_s,\pi,a)|E] \geq R_k - \epsilon/4$$

One case in which we do obtain a better bound is when the number of nondefining variables is sufficiently small (and thus, the number of defining variables is sufficiently large). In this case, Lemma 2.26 yields a better lower bound on μ, and thus a better upper bound on the running time of the algorithm.

Conversely, if the number of defining variables is very small, we expect that we should be close to the analysis which assumed a large distance. For example, if the number of defining variables is zero, then there is only one satisfying assignment, and the previous analysis holds. However, if there is more than one satisfying assignment, it is not clear how to identify and exploit this isolation.

Recall the proofs of Lemmas 2.8 and 2.12, where we proved statements of the form

$$2^{-|\mathrm{NDef}(P,a)|}\mathbf{E}_\pi[2^{|\mathrm{Forced}_a(G,\pi,a)|}] \geq 2^{-|\mathrm{NDef}(P,a)|+\mathbf{E}_\pi[|\mathrm{Forced}_a(G,\pi,a)|]}$$

using the concavity of the exponential function to conclude $\mathbf{E}[2^x] \geq 2^{\mathbf{E}[x]}$. This concavity bound is tight when x is a constant, and loose when x is far from its expectation with some probability.

If there are few defining variables, their position in the ordering π could be crucial. If all defining variables occur at the beginning of the ordering π, then after they are assigned values the formula has only one satisfying assignment, allowing us to apply our stronger analysis of `ResolveSAT` in this case. When this happens, the number of forced variables can be much better than its expectation, which implies that the bound $\mathbf{E}[2^x] \geq 2^{\mathbf{E}[x]}$ is not tight.

To overcome this, we will be more careful in using this expectation. We divide our placements σ (the events from which we generate permutations) into two parts: σ_d and σ_n, which are placements on the defining and nondefining variables, respectively. We first condition on the event that σ has the following property: Given $0 \leq r \leq 1$, for any defining variable v, $\sigma(v) < r$, and for any nondefining variable w, $\sigma(w) > r$. With this assumption on σ, we obtain sufficiently good bounds that we can interchange the expectation and the exponentiation. By mul-

$a \oplus L(B) \in \mathrm{Sat}(F)$, giving rise to a dead node in T_{i+1}.

To simplify our analysis in this section, we modify `ConstructTree` so that when we reach this uncertain situation, we do not expand any further.

`NewConstructTree`(k–CNF formula F, assignment a, variable v, integer d)
  Let P be a partition of $\mathrm{Sat}(F)$ into unique subcubes
  $T =$ `ConstructTree`(F, a, v, d)
  for each node $t \in T$ in breadth-first fashion
    if t is marked dead
      remove t and the subtree rooted at t from T
    if $L(t) \in \mathrm{Def}(P, a)$
      mark all children of t dead.

Proposition 2.2 *1.* `NewConstructTree` *outputs a critical clause tree for* F_s

2. If b *is a leaf node of depth less than* d, $L(b) \in \mathrm{Def}(P, a)$

Our analysis can be motivated by a simple example: Suppose that the procedure `NewConstructTree` outputs the following tree T^*

Definition 2.28 *Let* v *be a nondefining variable of* $P(a)$. T^* *is a tree of depth 1 where the root is labeled by* v *and has* $k-1$ *children labeled by distinct defining nodes.*

Note that if $v \in \mathrm{NDef}(P, a)$, then v must have at least one critical clause at a in the original formula F, which by Lemma 2.8 implies the following.

Lemma 2.26

$$\mu_a(F_s|P(a)) \geq 2^{-(1-1/k)|\mathrm{NDef}(P,a)|}$$

At the same time, since all the children in T^* are labeled by defining nodes, then `NewConstructTree` will be unable to produce a larger tree with more cuts. From this, it is not clear that the bound we obtain for `ResolveSAT` is better than we had in the case of `RandomUC`.

Theorem 2.3 *Let F be a formula with exactly one satisfying assignment a. For any $\epsilon > 0$, there exists a sufficiently large integer s so that* `ResolveSAT`$(F, s, 2^{(R_k+\epsilon)n}$ *outputs a with high probability.*

Proof: Follow from Lemma 2.25, 2.8, and 2.2. ■

In particular, for $k = 3$ we obtain a running time of $2^{0.387n}$, significantly improving over the $2^{0.667n}$ running time of `RandomUC`.

2.5.3 General k–SAT

In trying to generalize these results to arbitrary k–CNF formula, it is helpful to recall which parts of our argument relied on this isolation property. Even if a is not distance–d isolated, then $T =$ `ConstructTree`(F, a, v, d) will be a critical clause tree. However it may not be a full critical clause tree because it may happen that $a \oplus L(P_i) \in \mathrm{Sat}(F)$ for some path P_i, giving rise to dead nodes. Q_T is then still a lower bound on the probability that v is forced, but we can no longer apply Lemma 2.19 to obtain a good lower bound on Q_T for all admissible trees T.

As in the analysis of `RandomUC`, we make use of a partition P into unique subcubes in an attempt to capture some of the isolation properties which exist in $\mathrm{Sat}(F)$. Having constructed P by Lemma 2.10, we then appeal to Lemma 2.11: If we show that $\mu_a(F_s|P(a))$ is large regardless of the number of variables in $\mathrm{Def}(P, a)$, then this will imply a lower bound on $\mu(F_s)$. This captures our intuition about isolation and density: Having few satisfying assignments implies large subcubes in P with one satisfying assignment each, which implies isolation.

As long as T_i constructed during `ConstructTree`(F, a, v, d) has all live nodes and only labels nodes with nondefining variables, then T_{i+1} will not contain dead nodes, since every assignment tested by `ConstructTree` lies inside the subcube $P(a)$. If T_i contains some leaf node b with $L(b) \in \mathrm{Def}(P, a)$, and B is the path from the root to b, then $a \oplus L(B) \notin P(a)$. In this case, it is possible that

Lemma 2.23 *For sufficiently large k, R_k approaches $\frac{\pi^2}{6(k-1)}$.*

Proof: For large k

$$\sum_{j=1}^{\infty} \frac{1}{j(j+\frac{1}{k-1})}$$

approaches

$$\frac{1}{k-1}\sum_{j=1}^{\infty}\frac{1}{j^2} = \frac{1}{k-1}(\frac{\pi^2}{6})$$

■

Lemma 2.24 *Let F be a k–CNF, and let v be a variable. For every $\epsilon > 0$, there exists a sufficiently large integer d such that if $s = s_{k,d}$, $F_s = \texttt{Resolve}(F, s)$, and $a \in \mathrm{Sat}(F)$ is distance–d isolated, then*

$$\Pr[v \in \mathrm{Forced}(F_s, \pi, a)] \geq R_k - \epsilon$$

.

Proof: For any variable v,

$$\Pr[v \in \mathrm{Forced}(F_s, \pi, a)] \geq Q_T \geq Q_{k,d} \geq R_k - \epsilon$$

by applying Lemma 2.17,2.19, and 2.22, respectively. ■

Lemma 2.25 *Let F be a k–CNF, and let v be a variable. For every $\epsilon > 0$, there exists a sufficiently large integer d such that if $s = s_{k,d}$, $F_s = \texttt{Resolve}(F, s)$, and $a \in \mathrm{Sat}(F)$ is distance–d isolated,*

$$\mathbf{E}[|\mathrm{Forced}(F_s, \pi, a)|] \geq (R_k - \epsilon)n$$

.

One easy application of Lemmas 2.24 and 2.25 is to uniquely satisfiable formulae, that is, formulae with exactly one satisfying assignment. Since there is only one satisfying assignment, it is distance–d isolated for all values of d, and we can apply the lemmas to obtain good bounds on μ in this case.

$$R_3 = \int_0^1 R_3(r)dr = 2 - 2\ln 2 \geq 0.6137$$

.

For $k > 3$, our results will be more complicated.

Lemma 2.22

$$R_k = \frac{1}{k-1}\sum_{j=1}^{\infty} \frac{1}{j(j+\frac{1}{k-1})}$$

Proof:

As we noted, $R_k(r)$ is a continuous strictly increasing function on $[0, r_k]$ and is 1 on $[r_k, 1]$. Therefore we can define a unique function $S_k(t)$ on the interval $[0, 1)$ which is an inverse for $R_k(r)$ on the interval $[0, r_k]$, and $\int_0^1 R_k(r)dr = \int_0^1 (1 - S_k(t))dt$

Now $S_k(t)$ is the unique value of r satisfying $t = (r + (1-r)t)^{k-1}$, which is

$$S_k(t) = \frac{t^{\frac{1}{k-1}} - t}{1-t}$$

and

$$1 - S_k(t) = \frac{1 - t^{\frac{1}{k-1}}}{1-t} = \sum_{i=0}^{\infty} t^i - t^{i+\frac{1}{k-1}}$$

Integrating $1 - S_k(t)$ from 0 to 1 yields

$$\begin{aligned} R_k &= \int_0^1 (1 - S_k(t))dt = \sum_{i=0}^{\infty} \frac{1}{i+1} - \frac{1}{i+1+\frac{1}{k-1}} \\ &= \sum_{j=1}^{\infty} \frac{1}{j} - \frac{1}{j+\frac{1}{k-1}} = \frac{1}{k-1}\sum_{j=1}^{\infty} \frac{1}{j(j+\frac{1}{k-1})} \end{aligned}$$

completing the proof. ■

placement σ, in the tree of depth d where every label is distinct, there is some cut C such that every $c \in C$ has $\sigma(c) < r$. This can be shown easily using the same induction as Lemma 2.19. Thus, $Q_{k,d+1}(r) \geq Q_{k,d}(r)$, with equality only when they are both equal to 0 or 1. Since $Q_{k,d}(r)$ is bounded by 1 and monotonic in d, it must have a limit. The operation of increasing the depth of the tree by one changes $Q_{k,d}(r)$ from x to $f_k^r(x)$, so the limit value x must satisfy $f_k^r(x) - x = 0$. Thus, we conclude that this limit is $R_k(r)$.

Since [0,1] is compact, $Q_{k,d}(r)$ is increasing with d, and $Q_{k,d}(r)$ converges pointwise to $R_k(r)$, $Q_{k,d}(r)$ converges to $R_k(r)$ uniformly, and thus the integrals $Q_{k,d}$ converge to R_k.

For the last property, we recall the the function $S(x)$ defined in the proof of the first property. We define $S_k(x)$ and $S_{k'}(x)$ with $k' > k$ by substituting different values of k into S. For $x \in (0,1)$, $S_{k'}(x) > S_k(x)$. These functions are the inverses of $R_{k'}(r)$ and $R_k(r)$, respectively

Now, if $R_{k'}(r) \geq R_k(r)$ for some $r \in (0,1)$, we obtain

$$r = S_{k'}(R_{k'}(r)) > S_k(R_{k'}(r)) \geq S_k(R_k(r)) = r$$

using the monotonicity of S_k. Thus, for all $r \in (0,1)$, $R_k(r)$ and R_k are monotonically decreasing functions of k. ■

These convergence properties allow us to make the earlier intuitive argument regarding limits precise.

Lemma 2.21 *For every $\epsilon > 0$, there exists an integer d such that $Q_{k,d} \geq R_k - \epsilon$.*

Proof: Follows from the definition of convergence. ■

For $k = 3$, we can explicitly solve for $R_3(r)$ to get

$$R_3(r) = \begin{cases} (\frac{r}{1-r})^2 & r < 1/2 \\ 1 & r \geq 1/2 \end{cases}$$

and

Proposition 2.1 *Let $k \geq 3$.*

1. *$R_k(r)$ is continuous and strictly increasing on the interval $[0, r_k]$ and $R_k(r) = 1$ on the interval $[r_k, 1]$*
2. *For each fixed d, $Q_{k,d}(r)$ is a continuous, nondecreasing function of r on $[0,1]$, with $Q_{k,d}(0) = 0$ and $Q_{k,d}(1) = 1$.*
3. *For each fixed r, $Q_{k,d}(r)$ is nondecreasing in d, and strictly increasing if $r \in (0,1)$.*
4. *For each fixed r, $(Q_{k,d}(r) : d \geq 0)$ converges to $R_k(r)$.*
5. *$(Q_{k,d} : d \geq 0)$ converges to R_k.*
6. *$R_k(r)$ and R_k are monotonically decreasing functions of k.*

Proof: For the first property, we define the function

$$S(x) = \frac{x^{1/(k-1)} - x}{1 - x}$$

where $x^{1/(k-1)}$ denotes the real, positive $(k-1)$-th root of x. This continuous function has the property that $S(R_k(r)) = r$ for $r \in [0,1)$. Furthermore, because all other k–th roots of x yield negative or complex values of $S(x)$, for $x \in [0,1)$, $S(x)$ is the only real, nonnegative value of r satisfying the equation $f_k^r(x) - x = 0$. We extend S by defining $S(1) = (1 - 1/k) = r_k$; using L'Hopitals' rule, this can be show to maintain the continuity of S. S one-to-one on the domain $[0,1]$, since it is the inverse of the function $R_k(r)$ on $[0, r_k]$. Thus, S is strictly monotonic on $[0,1]$, and its inverse $R_k(r)$ is also continuous and strictly monotonic on $[0, r_k]$.

The second property can be proven by induction on d. $Q_{k,d}(r)$ is a polynomial in r, and is some positive power of $(r + (1-r)Q_{k,d-1}(r))$, which by induction is a nondecreasing function of r and is 0 (resp. 1) at $r = 0$ ($r = 1$).

To show that $Q_{k,d}(r)$ is monotonic in d for fixed r, we appeal to the trees which led to the definition of Q. $Q_{k,d}(r)$ is the probability that for a random

For $k = 3$, approximate values of the quantities $1 - Q_{k,d}$ are calculated in the following table:

d	$1 - Q_{3,d}$
1	.667
2	.558
3	.505
4	.474
5	.454
6	.441
7	.431
8	.424
9	.419

When $d = 1$, a is an n–isolated satisfying assignment, and we obtain the same quantity $(1 - 1/k)$ as in the exponent of the bound in Lemma 2.9. Even for small d, we obtain substantial improvements with a very limited amount of resolution in the case where the formula contains some well-isolated satisfying assignment a. However, the incremental benefit from increasing d diminishes for large d.

Because the degree of $Q_{k,d}(r)$ increases rapidly with d, computing these expressions is quite computationally intensive. Intuitively, however, the probability that v is forced should continue to increase with d, since the larger tree potentially gives rise to more critical clauses. If this is true, rather than computing a $Q_{k,d}(r)$ for different values of d, we can look for a limit as d becomes large. If this limit exists, then if d is sufficiently large, $Q_{k,d}(r)$ will be close to this limit value.

To do so, we define

Definition 2.27 $f_k^r(x) = (r + (1 - r)x)^{k-1}$

$R_k(r)$ is the smallest nonnegative real root of the equation $f_k^r(x) - x = 0$; this is well-defined since 1 is a root.

$R_k = \int_0^1 R_k(r)dr$. r_k is the least r such that $R_k(r) = 1$; r_k is well defined because $R_k(1) = 1$.

By elementary analytic arguments one can show:

U'. Thus, each set W_i is monotone. It then follows from standard correlational inequalities (see Theorem 3.2 of [2]) that $\Pr[\bigwedge K_i] \geq \prod \Pr[K_i]$.

■

Definition 2.26 *Let $Q_{k,d}(r)$ be defined inductively by $Q_{k,0}(r) = 0$ and $Q_{k,d}(r) = (r + (1-r)Q_{k,d-1}(r))^{k-1}$, for all k. Let $Q_{k,d} = \int_0^1 Q_{k,d}(r)dr$.*

By computing these expressions $Q_{k,d}(r)$, we obtain lower bounds of $Q_T(r)$ for the critical clause tree constructed earlier.

Lemma 2.19 *If T is a full, admissible tree of degree at most $k-1$ and depth at least d, then for all $r \in (0,1)$, $Q_T(r) \geq Q_{k,d}(r)$, and $Q_T \geq Q_{k,d}$.*

Proof: The first inequality follows from Lemma 2.18 by induction on d, and the second by integrating both sides over $r \in [0,1]$. ■

With this, we can compute lower bounds on $\mu_a(F_s)$ in the case that a is at least distance–d isolated in terms of the integrals $Q_{k,d}$, thus obtaining an extension of Lemma 2.9.

Lemma 2.20 *Let a be a distance–d isolated satisfying assignment of F, $s = s_{k,d}$, and $F_s = \texttt{Resolve}(F, s)$, then $\mu_a(F_s) \geq 2^{-(1-Q_{k,d})n}$.*

Proof:

Let $T = \texttt{ConstructTree}(F, a, v, d)$.

For any variable v,

$$\Pr[v \in \text{Forced}(F_s, \pi, a)] \geq Q_T \geq Q_{k,d}$$

by Lemmas 2.17 and 2.19. By linearity of expectation, $\mathbf{E}_\pi[\text{Forced}(F_s, \pi, a)] \geq Q_{k,d}n$. The lemma then follows by Lemma 2.8. ■

Recall $\mu(F_s) \geq \mu_a(F_s)$, and that by Lemma 2.2, the running time of `RandomUC` on F_s (and thus of `ResolveSAT` on F) is (up to polynomial factors) $1/\mu(F_s)$. By Lemma 2.20, this is $1/\mu(F_s) = 2^{(1-Q_{k,d})n}$.

The second inequality is obtained from the first by integrating both sides where r goes from 0 to 1. Since the inequality holds for all values of $r \in [0,1]$, then it also holds when we integrate over r in this range. ■

Lemma 2.18 *Let T be a full, admissible tree with more than one node, t be the root node of T, and let $T_1, \ldots, T_l$ be the full, admissible subtrees rooted at the labeled children of t. Then $\forall r \in (0,1)$:*

$$Q_T(r) \geq \Pi_{i=1}^{l}(r + (1-r)Q_{T_i}(r))$$

where the empty product is interpreted as 1.

Proof:

If $l = 0$, then all children of T are unlabeled. The cut A which consists of these children has $L(A) = \emptyset$, so the event $Cut_T(r)$ happens with probability 1.

If $l > 0$, for each $1 \leq i \leq l$, let v_i be the variable labeling the root of T_i. Let K_i be the event that $Cut_{T_i}(r) \vee [\sigma(v_i) < r]$, that is, that there exists a weak cut A_i of T_i such that every $w \in L(A_i)$ has $\sigma(w) < r$. The event Cut_T can then be written as $\bigwedge_{i=1}^{l} K_i$, because every path from root to leaf in T passes through one subtree T_i.

Because T is admissible, v_i does not appear as a label anywhere in $L(T_i)$ except at the root. Thus, the events $Cut_{T_i}(r)$ and $[\sigma(v_i) < r]$ are independent, and $\Pr[K_i] = r + (1-r)Cut_{T_i}(r)$.

Now, we must compute the probability of $\bigwedge K_i$. This is complicated by the fact that a variable can appear as a label in more than one T_i, so these events are not independent. However, the events K_i are not negatively correlated, and by showing this, we will still be able to lower bound $\Pr[\cap K_i]$ by $\prod \Pr[K_i]$.

To formalize this intuition, we define $W(\sigma, r) = \{v | \sigma(v) < r\}$. The event K_i is the event that $W(\sigma, r)$ includes a weak cut of T_i. We then define W_i to be the set of subsets of variables U such that if $U = W(\sigma, r)$ then the event K_i occurs. If $U \in W_i$, and $U \subseteq U'$, then $U' \in W_i$, since if U contains a weak cut of T_i, so does

is one-to-one with probability 1, the associated permutation π is well-defined with probability 1 as well. If σ is chosen uniformly from the set of placements, then π is a uniformly chosen permutation. Now, if $\sigma(v) = r$, then any other variable has probability r of occurring before v, independent of the placement of all other variables. For the remainder of this section, unless otherwise noted, all events will be taken over the space of placements, rather than permutations.

With this notation, we will compute $\Pr[v \text{ is forced} | \sigma(v) = r]$, and then integrate over r to calculate $\Pr[v \text{ is forced}]$.

Lemma 2.16 *For a CNF formula G with $a \in \mathrm{Sat}(G)$,*

$$\Pr_{\sigma}[v \in \mathrm{Forced}(G, \pi, a)] = \int_0^1 \Pr_{\sigma}[v \in \mathrm{Forced}(G, \pi, a) | \sigma(v) = r] dr$$

Proof: The value of $\sigma(v)$ is uniformly distributed in $[0, 1]$. ∎

We now focus on computing this probability.

Definition 2.25 *For a critical clause tree T whose root is labeled by v, let $Cut_T(r)$ be the event that there is a cut A of T such that all variables in $L(A)$ occur before r in σ. Let $Q_T(r) = \Pr[Cut_T(r)]$ and $Q_T = \int_0^1 Q_T(r)$.*

Lemma 2.17 *If T is a critical clause tree for v for a formula G,*

$$\Pr[v \in \mathrm{Forced}(G, \pi, a) | \sigma(v) = r] \geq Q_T(r)$$

and

$$\Pr[v \in \mathrm{Forced}(G, \pi, a)] \geq Q_T$$

Proof: For the first part, let $B = \{b_i\}$ be the set of nodes such that $\sigma(L(b_i)) < r$. If B contains some cut A of T, then there exists an associated critical clause $C(A)$. If every variable $x \in L(A)$ has $\sigma(x) < \sigma(v)$, then in the associated permutation π, v occurs after every $x \in L(A)$, and thus occurs last among the variables in the critical clause $C(A)$. Thus, when v is examined in `Modify`, there is a unit clause which forces v.

Lemma 2.15 *If a is distance–d isolated in* Sat(F), *then for any variable v, the critical clause tree* `ConstructTree`(F, a, v, d) *contains no dead nodes. Furthermore, any critical clause guaranteed by the tree can be derived using $s_{k,d}$–bounded resolution.*

Proof: For the first statement, note that a dead node is produced only if, for some assignment $a \oplus L(P_i)$, no clause is made false. If $|P_i|$ is at most d, then this assignment must be false, so there must exist a falsified clause. The second statement follows from the fact that the size of the tree is at most $s_{k,d}$. ■

Given a variable v and a distance–d isolated satisfying assignment, we can construct a full critical clause tree of depth d. Now, we show that this critical clause tree implies that v is forced with sufficiently large probability. To do so, we break the problem into two parts. First, we compute the probability that v is forced given that it occurs at a particular position in the ordering π. This will be some function of the position. If v is first, it will not be forced unless the formula contains a unit clause for v; if v is last, it will certainly be forced. Once we obtain an expression for the probability that v is forced given that v appears at a given position, we can compute the probability that v is forced (without any conditions) by using the fact that v itself is positioned uniformly.

However, this introduces some complications. Because there are exactly n variables, if we determine that some variable occurs before v or after v, then this affects the probabilities of these events for all other variables, albeit only slightly. To simplify these calculations, we will produce the permutation π by a somewhat roundabout method.

Definition 2.24 *A* placement *is a function σ on the set of variables with $\sigma(v) \in [0, 1]$. A one-to-one placement σ* determines *a permutation π such that $\pi(u) < \pi(v) \iff \sigma(u) < \sigma(v)$.*

Instead of considering random permutations, we consider random placements, and use the permutations determined by them in the algorithms. Since σ

literals, and C' contains at least one variable. For each variable $c_j \in \text{vars}(C')$, we can associate a node $n_j \in P_i$ and a critical clause $C(A_j)$ for the cut $A' \cup n_j$. As noted above, each critical clause contains v, some negated variables from $L(A')$, and exactly one negated variable from $L(P_i)$, namely c_j. Thus, each $C(A_j)$ and C_i conflict only on the variable c_j. We set $D_0 = C_i$, and produce D_{j+1} by resolving D_j with $C(A_j)$ on the variable c_j. Note that each D_{j+1} has one less positive literal than D_j and conflicts with each remaining $C(A_j)$ on exactly one variable. At the end of this process, we obtain a clause with no positive literals except v, which is a critical clause for v at a. Furthermore, the only variables which appear in this clause are elements of either $L(N_i)$ or $L(A')$, and thus of $L(A)$.

Every clause used in this resolution process is certainly bounded by the size of the tree, since all variables are labels of some node in the tree, so by induction this new critical clause can be derived from F using s–bounded resolution.

■

2.5.2 Isolated Satisfying Assignments

Our goal now is to prove an analogue of Lemma 2.6, by showing that the existence of a full critical clause tree of large depth for a variable implies that the variable is forced with high probability. As in that lemma, however, in order to ensure that this occurs we will need to make some assumption about the isolation of the satisfying assignment a. However, here we will need a stronger notion of isolation.

Definition 2.23 *A satisfying assignment a is distance–d isolated if, for all $b \neq a$ with Hamming distance at most d from a, b is not a satisfying assignment.*

This extends our previous definition of isolation: n–isolation is the same as distance–1 isolation. From this, we obtain

For T_0, the statement is vacuously true, since there are no cuts in T_0. For T_1, the tree has only one cut, consisting of all leaf nodes. T_1 is constructed from a clause C which is false on $a \oplus v$, which is a critical clause for v at a. vars(C) consists of v together with the labels of the children of T_1.

Now, we consider $i \geq 2$ and assume that the lemma is true for T_{i-1}. T_i consists of T_{i-1} together with the set N_i of newly created children of b_i. Let A be a cut of T_i, and let $A' = A - N_i$. Since the path P_i from the root to b_i does not include these newly created children, for any $p \in P_i$, the set of nodes $A_p = A' \cup p$ is contained in the tree T_{i-1}. Furthermore, A_p forms a cut of T_{i-1}; by the inductive hypothesis, this implies that there is a critical clause $C(A_p)$ for v at a in F_s with vars$(C(A_p)) \subseteq L(A_p) \cup v$.

We first dispose of several simple cases where we can find an already derived critical clause C with vars$(C) \subseteq L(A') \cup v$, so vars$(C) \subseteq L(A) \cup v$, and we can take C as the critical clause for A. If A does not contain any nodes from N_i, then A is a cut of T_{i-1} and the inductive hypothesis guarantees a critical clause $C(A)$ with vars$(C(A)) \subseteq L(A) \cup v$. If $L(p) \notin C(A_p)$, then vars$(C(A_p)) \subseteq L(A') \cup v$. Similarly, if there is $p' \in P_i, p' \neq p$ with $L(p) \in$ vars$(C(A'_p))$, then $L(p) \in L(A'_p)$. This implies that there is a node $n_p \in A'$ with $L(n_p) = L(p)$. Thus, $L(A_p) \subseteq L(A')$ and vars$(C(A_p)) \subseteq L(A')$.

The remaining case is that for every $p \in P_i$, $C(A_{p'})$ contains $L(p)$ but not $L(p')$ for any $p' \in P_i, p' \neq p$. Since $C(A_p)$ is a critical clause for the all-true assignment a, v occurs positively in $C(A_p)$, while all other variables in the clause appear negated. In this case, we can write each clause $C(A_p)$ as $v \vee \overline{L(p)} \vee B_p$, where B_p is the disjunction of negated variables from $L(A')$.

We return to the clause C_i, and express it as $C_i = C' \vee C''$, with vars$(C') \subseteq L(P_i)$ and vars$(C'') =$ vars$(C_i) - L(P_i)$. By definition, C_i is false on the assignment $a \oplus L(P_i)$, but true on a. Using our simplifying assumption that a is the all-true assignment, this implies that C' consists of positive literals, C'' consists of negated

- *The root label is* v .
- *For any cut* A *of the tree, if every element of* A *is live then* G *has a critical clause* $C(A)$ *for* v *at* z *such that* $\text{vars}(C(A)) \subseteq L(A) \cup \{v\}$.

Definition 2.21 *A labeled tree is* full *if it contains no dead nodes.*

Given a k-CNF formula F, a variable v, and a satisfying assignment $z \in \text{Sat}(F)$, we now construct a critical clause tree for $F_s = \texttt{Resolve}(F, s)$.

```
ConstructTree(k–CNF formula F, assignment a, variable v, integer d)
  T_0 = tree consisting of one node r labeled v
  repeat
    if no labeled leaf node of depth at most d exists
      then exit
    b_i = a labeled leaf node of minimum depth
    P_i = the set of nodes on the path from the root to b_i, including the endpoints
    if there is a clause C_i ∈ F which is false on a ⊕ L(P_i)
      if vars(C_i) − L(P_i) = ∅
        T_{i+1} = T_i ∪ t, where t is a new unlabeled child of b_i.
      else
        for each variable w ∈ vars(C_i) − L(P_i)
          T_{i+1} = T_i ∪ t, where t is a new child of b_i with L(t) = w.
    else
      T_{i+1} = T_i ∪ t, where t is a new dead child of b_i.
    i = i + 1
```

Definition 2.22 $s_{k,d} = \sum_{j=0}^{d+1} (k-1)^j$

Lemma 2.14 *Let* $s = s_{k,d}$. *Every tree* T_i *constructed in* $\texttt{ConstructTree}(F, a, v, d)$ *is a critical clause tree for* $F_s = \texttt{Resolve}(F, s)$ *for* v *at* a.

Proof: To prove this, we must show that if A is a set of live nodes forming a cut, then there is a critical clause $C(A)$ for v at a in F_s. The proof is by induction on i.

Without loss of generality, we assume that a is the all-true assignment. Any other case can be translated to this one by interchanging the roles of x_i and $\bar{x}_i$ in the proof for every variable on which a is false.

variables in common, introducing dependencies between the events that v occurs last in one clause and v occurs last in another. To account for these contributions, we use the fact that these clauses are produced using resolution, and represent these new critical clauses using a labeled tree.

Definition 2.17 *The degree of a node in a rooted tree is the number of its children. The depth of a node is its distance from the root. The depth of the tree is the maximum depth of any leaf.*

A subset A of nodes is a cut *if it does not include the root, and every path from the root to a leaf includes a node of A. If we remove the requirement that A does not include the root, then A is a* weak cut*.*

If A is a set of nodes, write $L(A)$ for the set of variables that appear as labels of nodes of A.

Definition 2.18 *A rooted tree is said to be* admissible *with respect to a given set of Boolean variables if it has the following properties:*

- *The root is labeled by a variable.*
- *Each node in the tree is either dead, unlabeled, or labeled by a variable.*
- *Any dead or unlabeled node is a leaf of the tree.*
- *For any path P from the root to a leaf, no two nodes have the same label. In other words, if node a is an ancestor of node b and both are labeled, then they have different labels.*

Definition 2.19 *A node if* live *if it is not dead; that is, it is either unlabeled or labeled by a variable.*

Definition 2.20 *A tree is said to be a* critical clause tree *for variable v, formula G and satisfying assignment z if it is admissible and in addition satisfies*

`Resolve`(F, s) runs in time at most $O(\text{poly}(n)|F|n^{2s})$, and adds at most $O(n^s)$ clauses to F. If $s = o(n/\log n)$, then execution of `Resolve`(F, s) adds only a subexponential term to the running time of `ResolveSAT`(F, s, I). If we consider values of s with this property, then the running time of `ResolveSAT` will be determined up to a subexponential factor by the number of iterations I.

The approach for analyzing `ResolveSAT` is similar to the one taken previously for `RandomUC`. However, to account for the multiple critical clauses we hope to find in F_s we will need some additional machinery. First, we describe a procedure which, given a satisfying assignment a and a variable v, constructs a tree whose nodes are labeled by variables. Furthermore, this tree has the property that if a set of nodes forms a cut of the tree, then there exists a critical clause which corresponds to the labels of that set. Using this, we show that if the satisfying assignment a is isolated in a strong sense, then for any v, the critical clauses implied by the tree constructed for v at a cause v to be forced with higher probability than we obtained using only one critical clause. From this, we conclude that if such a well-isolated a exists, `ResolveSAT` finds it quickly. The last step is to generalize this result to apply to sets of satisfying assignments which do not include such well-isolated points. As before, this relies on the intuition that the absence of isolated satisfying assignments implies the existence of many satisfying assignments.

2.5.1 Critical Clauses and Isolation

In order to show that `RandomUC` performs well on $F_s =$ `Resolve`(F, s), we must show that the new critical clauses added to F make the appearance of unit clauses in `Modify` more likely. In trying to account for the contributions of the new critical clauses, there are two main challenges. First, these critical clauses can be of varying lengths, which affects the probability that the critical variable occurs last among the variables in the clause. Second, these critical clauses may have

as $C_1 = v \vee D_1$ and $C_2 = \bar{v} \vee D_2$ for some clauses D_1, D_2. If v is true, then any satisfying assignment of F must satisfy D_2; likewise, if $\bar{v}$ is true, then D_1 is satisfied. Since exactly one of v and $\bar{v}$ is true, the clause $D_1 \vee D_2$ must be true on any satisfying assignment. Therefore, adding this clause to F will not change the set $\mathrm{Sat}(F)$. If C_1 and C_2 conflict on another variable w, then $D_1 \vee D_2$ contains both w and $\bar{w}$, and so is trivially true.

Definition 2.15 *Two clauses are* resolvable *if they conflict on exactly one variable. The* resolvent *of two resolvable clauses $C_1 = v \vee D_1$ and $C_2 = \bar{v} \vee D_2$ is the clause $R(C_1, C_2) = D_1 \vee D_2$.*

This one rule is complete: applying it repeatedly is sufficient to obtain a contradiction if F is unsatisfiable. However, to do so, it may be necessary to generate many intermediate clauses. To see how easily the process can spiral out of control, notice that the resolvent of two length k clauses is a length $2(k-1)$ clause unless the clauses have other variables in common. Repeating this process quickly generates many large clauses. In our algorithm, we avoid this by not performing all possible resolution steps.

Definition 2.16 *A resolvable pair of clauses is* s–bounded *if their resolvent has at most s literals.*

Rather than producing all possible resolvents, we will confine ourselves to producing s–bounded ones. While this limited resolution is no longer capable of deriving all contradictions, it will produce enough new clauses to aid our search.

With this notation, `Resolve` simply closes the formula under resolution of s–bounded pairs of resolvable clauses.

`Resolve`(CNF Formula F, integer s)
  $F_s = F$.
  while F_s has an s-bounded resolvable pair C_1, C_2 with $R(C_1, C_2) \notin F_s$
    $F_s = F_s \wedge R(C_1, C_2)$.
  return (F_s).

the existence of multiple critical clauses for each variable. To do this, we turn to the analysis of the formula Unique used in Section 2.4. While this formula is hard for `RandomUC`, it looks quite easy to a human observer. Rather than setting variables blindly, a better strategy for satisfying this particular formula is to look ahead at the consequences of these partial assignments. With a little foresight, it becomes clear that for each variable, there is only one way which it can be set to satisfy the formula. Thus, the formula *implies* unit clauses for every variable, which guarantee forcing. `RandomUC` is unable to make use of this, however, because these implications are spread across a number of clauses, but `RandomUC` can only use the implications from the critical clauses present in the formula.

Our next algorithm, `ResolveSAT`, makes use of these two observations. First, it finds implications among sets of clauses in the formula and augments the formula by adding these implied clauses to the formula explicitly. Then, `RandomUC` is called on the new, augmented formula. These new clauses include new critical clauses. When this yields multiple critical clauses per variable, the variable is forced with higher probability, and `RandomUC` runs faster on the augmented formula than it did on the original.

This gives rise to a simple program:

`ResolveSAT`(CNF-formula F, integer s, positive integer I)
  $F_s =$ `Resolve`(F, s).
  `RandomUC`(F_s, I).

`Resolve`, the procedure for augmenting the formula, remains to be specified. As the name suggests, it makes use of techniques from the resolution proof system, so we review a few definitions:

Definition 2.14 *Two clauses C_1 and C_2* conflict on the variable v *if one contains v and the other contains $\bar{v}$.*

With this, we can explain the key idea of resolution: Given two clauses $C_1, C_2 \in F$ which conflict on v, without loss of generality, they can be written

Theorem 2.2 *For any* $0 \leq \epsilon < 1$, *if* $I \leq \epsilon 2^{(1-1/k)n}$, *then* `RandomUC`$(Unique, I)$ *fails with probability at least* $1 - \epsilon$.

Proof: The formulae Unique clearly has only one satisfying assignment. Now, consider a run of `Modify`(Unique$, \pi, a$) for random π, a. Within each subformula Unique$_k$, only the last variable chosen can be forced, since only unit clauses can cause forcing. If any variable except the last from each group is set incorrectly, the entire run fails. Within each subformula, the probability of all variables being set correctly is $2^{-(k-1)}$, since every group has size k. Since the subformulae use disjoint sets of variables, these events are independent and the probability that all variables in all subformulae are correct is $2^{-(1-1/k)n}$. The probability that at least one of the $\epsilon 2^{(1-1/k)n}$ trials succeeds is at most ϵ. ■

2.5 ResolveSAT

In order to find satisfying assignments in time less than $2^{(1-1/k)n}$, we need to change the algorithm in some fashion. From the discussion on the limitations of `RandomUC`, we focus on one particular weakness of the analysis: Lemma 2.5 guarantees that if a satisfying assignment is isolated with respect to a variable, then there exists *at least one* critical clause for that variable. In accounting for the contributions of critical clauses, however, we obtained a lower bound on the probability of forcing a variable assuming *exactly one* critical clause. If a variable v has more critical clauses, it will be more likely that v is forced, since this will happen if v occurs last in any of its critical clauses. Suppose F contains two such critical clauses for x_1 at the all-true satisfying assignment, for example $(x_1 \vee \bar{x}_2 \vee \bar{x}_3)$ and $(x_1 \vee \bar{x}_4 \vee \bar{x}_5)$ with all of x_1, x_2, x_3, x_4, and x_5 distinct. In this case, the probability that a unit clause for x_1 occurs is $1/2$, rather than the $1/3$ obtained using only a single critical clause.

In order to benefit from this observation, we need to be able to prove

and we obtain

$$\mu(F) \geq 2^{-(1-1/k)|\mathrm{NDef}(P,a)|} \geq 2^{-(1-1/k)n}$$

since $|\mathrm{NDef}(P,a)| \leq n$. ■

Theorem 2.1 *On any satisfiable k–CNF formula F,* `RandomUC`$(F, n2^{(1-1/k)n})$ *outputs a satisfying assignment with high probability.*

Proof: By Lemma 2.13,

$$\mu(F) \geq 2^{(1-1/k)n}$$

By Lemma 2.2, this implies that `RandomUC`$(F, n2^{(1-1/k)n})$ outputs a satisfying assignment with high probability. ■

2.4 Limitations of `RandomUC`

Having shown that `RandomUC` provides an advantage over exhaustive search, we naturally wish to improve this advantage by showing better upper bounds on the running time of `RandomUC`. There are a number of places where our analysis used bounds that may not have been tight. The proof of Lemma 2.13 used the seemingly crude bound that $|\mathrm{NDef}(P,a)| \leq n$. Lemma 2.5 guarantees the existence of at least one critical clause per isolated variable; even if there is more than one critical clause, Lemma 2.9 makes use of only one. Finally, in partitioning $\mathrm{Sat}(F)$ into subcubes, we eliminated events where an assignment in one subcube produced a satisfying assignment in another.

However, a simple example shows that the bound in Theorem 2.1 is essentially tight.

Definition 2.13 *$Unique_k(x_1, \ldots, x_k)$ is the k–CNF formula on k inputs with $2^k - 1$ clauses which accepts if and only if all inputs are true. For n divisible by k, we define*

$$Unique(x_1, \ldots, x_n) = \bigwedge_{i=0}^{(n/k)-1} Unique_k(x_{1+in/k}, \ldots, x_{k+in/k})$$

.

We prove an analogue of Lemma 2.8 in this restricted case.

Lemma 2.12

$$\mu_a(G|P(a)) \geq 2^{-|\mathrm{NDef}(P,a)|+\mathbf{E}_\pi[|\mathrm{Forced}_a(G,\pi,a)|]}$$

Proof: Because we condition on the event that $y \in P(a)$, if $v \in \mathrm{Def}(P,a)$ then y and a agree on the value of v. By Lemma 2.3, $\texttt{Modify}(G,\pi,y) = a$ if every nondefining variable on which a and y differ is forced. As in Lemma 2.8, this implies that the probability that `Modify` succeeds is at least

$$2^{-|\mathrm{NDef}(P,a)|}\mathbf{E}_\pi[2^{|\mathrm{Forced}_a(G,\pi,a)|}] \geq 2^{-|\mathrm{NDef}(P,a)|+\mathbf{E}_\pi[|\mathrm{Forced}_a(G,\pi,a)|]}$$

■

Now, we use the relationship between the partition into unique subcubes and isolation to prove one of our key lemma.

Lemma 2.13 *For all satisfiable k–CNF formulae F,*

$$\mu(F) \geq 2^{-(1-1/k)n}$$

Proof: Let $P = P_{\mathrm{Sat}(F)}$ be the partition into unique subcubes whose existence is guaranteed by Lemma 2.10.

By Lemma 2.11

$$\mu(F) \geq \min_{a \in \mathrm{Sat}(F)} \mu_a(F|P(a))$$

And, by Lemma 2.12, for all $a \in \mathrm{Sat}(F)$,

$$\mu_a(F|P(a)) \geq 2^{-|\mathrm{NDef}(P,a)|+\mathbf{E}[|\mathrm{Forced}_a(F,\pi,a)|]}$$

Because a is isolated with respect to each nondefining variable,

$$\mathbf{E}[|\mathrm{Forced}_a(F,\pi,a)|] \geq |\mathrm{NDef}(P,a)|/k$$

then obtain P_A by taking the union of the parts in P_{A_T} and P_{A_F}, regarding each part as a subcube of the entire hypercube by setting v to the appropriate value. ■

Given a partition P into unique subcubes, we can view $P(a)$ as defining a neighborhood around a. Our hope is that if $b \in P(a)$, then it will be likely that $\texttt{Modify}(F, \pi, b) = a$ rather than some other satisfying assignment a'. With this motivation, we partition the events $\texttt{Modify}(G, \pi, y) = a$ according to which $p \in P$ contains y.

Definition 2.11 *Let z be a satisfying assignment of F.*

$$\mu_a(G|P(z)) = \Pr[\texttt{Modify}(G, \pi, y) = a | y \in P(z)]$$

$$\mu(G|P(z)) = \sum_{a \in \mathrm{Sat}(G)} \mu_a(G|P(z))$$

Lemma 2.11 *For any satisfiable formula G, let $P = P_{\mathrm{Sat}(G)}$ be a partition into unique subcubes. Then*

$$\mu(G) \geq \min_{a \in \mathrm{Sat}(G)} \mu_a(G|P(a))$$

Proof:

$$\begin{aligned}
\mu(G) &= \sum_{a \in \mathrm{Sat}(G)} \mu(G|P(a)) \Pr[y \in P(a)] \\
&\geq \sum_{a \in \mathrm{Sat}(G)} \mu_a(G|P(a)) \Pr[y \in P(a)] \\
&\geq \left(\min_{a \in \mathrm{Sat}(G)} \mu_a(G|P(a)) \right) \sum_{a \in \mathrm{Sat}(G)} \Pr[y \in P(a)] = \min_{a \in \mathrm{Sat}(G)} \mu_a(G|P(a))
\end{aligned}$$

■

Because we are conditioning on the event that y agrees with a on the defining variables, it will be helpful to restrict the definition of Forced to the nondefining variables. Given a partition P into unique subcubes, we define

Definition 2.12

$$\mathrm{Forced}_a(G, \pi, y) = \mathrm{Forced}(G, \pi, y) \cap \mathrm{NDef}(P, a)$$

Where it is clear from context that we are discussing a formula F, A refers to the set of satisfying assignments $\mathrm{Sat}(F)$ and $P = P_{\mathrm{Sat}(F)}$ is a partition into unique subcubes with respect to this set.

A partition into unique subcubes provides the needed link between the number and isolation of the satisfying assignments: Previously, we gave a sketch of an argument which had two cases: either there is some isolated $a \in \mathrm{Sat}(F)$, or $\mathrm{Sat}(F)$ is large. This argument naturally translates to statements about a partition into unique subcubes. Let $a \in \mathrm{Sat}(F)$, and P be a partition into unique subcubes. If $v \in \mathrm{NDef}(P, a)$, then a is isolated with respect to v, since $a \oplus v \notin \mathrm{Sat}(F)$; $a \oplus v$ also lies in the subcube $P(a)$, but a is the unique satisfying assignment in $P(a)$. If some $a \in \mathrm{Sat}(F)$ has many nondefining variables, then a will be isolated with respect to many variables. Otherwise, every $a \in \mathrm{Sat}(F)$ has many defining variables, which implies that the corresponding subcube $P(a)$ is small. To cover all assignments using small subcubes requires many subcubes in P, and thus many satisfying assignments a.

While this is the intuition behind our proof, we take a slightly different route. Rather than dividing into these two cases, we look at the extent to which each satisfying assignment is isolated.

First, we show that any nonempty set can be partitioned into unique subcubes.

Lemma 2.10 *For any nonempty subset A of the set of assignments, there exists a partition P_A into unique subcubes.*

Proof: The proof is by induction on n. If there is only one $a \in A$, then the partition consists of the entire cube. Otherwise, there exists some variable v such that there exist satisfying assignments both with $v = T$ and $v = F$. Let C_T and C_F be the subcubes obtained by restricting v in this fashion, and let A_T and A_F be A restricted to these subcubes. By induction, C_T and C_F can be partitioned into unique subcubes; let P_{A_T} and P_{A_F} be the partitions obtained in this fashion. We

2.3 Nonisolated solutions

Suppose that there is some satisfying assignment a which is n–isolated; that is, a is isolated. In this case, the proof is complete: From Lemma 2.9, we obtain $\mu_a(F) \geq 2^{-n+\frac{n}{k}}$. Since $\mu(F)$ is a sum of terms $\mu_a(F)$, $\mu(F) \geq \mu_a(F)$. Applying Lemma 2.2, we obtain that $I = O(n2^{(1-1/k)n})$ iterations of `RandomUC` suffice to find a with high probability.

In general, though, there need not be any isolated satisfying assignments. We show that in this case, there must be many satisfying assignments. If $a \in \mathrm{Sat}(F)$ is not isolated, then some of its neighbors must also be satisfying assignments. By applying this observation repeatedly, if no satisfying assignment is isolated, then the existence of one satisfying assignment implies the existence of many. From this, we will be able to conclude that even if no single term in $\mu(F) = \sum_{a \in \mathrm{Sat}(F)} \mu_a(F)$ is large, the sum is large because it has many terms corresponding to the many satisfying assignments of F.

First, we need a method for relating isolation to the set of satisfying assignments.

Definition 2.8 *A* subcube *is a set of assignments defined by fixing the values of some variables, which we call* defining *variables, and allowing all possible settings of the remaining* nondefining *variables.*

Definition 2.9 *Let A be a nonempty set of assignments. P_A is a* partition into unique subcubes *with respect to A if P_A is a partition of the set of assignments, every part $p \in P_A$ is a subcube,*

and every $p \in P_A$ contains exactly one $a \in A$. For $a \in A$, let $P(a) \in P_A$ denote the subcube containing A.

Definition 2.10 *Given a partition P_A into unique subcubes and an assignment a,* $\mathrm{Def}(P, a)$ *is the set of defining variables for the subcube $P(a)$ and* $\mathrm{NDef}(P, a)$ *is its complement.*

Proof: Because a is isolated with respect to v, there exists some critical clause C for v at a by Lemma 2.5. Since C is a clause of F, at most k variables appear in C. With respect to a randomly chosen permutation π, v occurs last among the variables in C probability at least $1/k$. By Lemma 2.4, this implies that v is forced with probability at least $1/k$. ■

Lemma 2.7 *If a is a j–isolated satisfying assignment of a k–CNF F, then*

$$\mathbf{E}_\pi[|Forced(F,\pi,a)|] \geq \frac{j}{k}$$

.

Proof: By Lemma 2.6, each variable v such that a is isolated with respect to v is forced with probability at least $1/k$. Since there are j such variables, the expected number of forced variables is at least j/k, by linearity of expectation. ■

By Lemma 2.3, if $\mathrm{Forced}(F,\pi,a) = l$, then there are 2^l assignments b for which $\texttt{Modify}(F,\pi,b)$ returns the satisfying assignment a. We can calculate the probability $\mu_a(F)$ as follows.

Lemma 2.8

$$\mu_a(F) \geq 2^{-n+\mathbf{E}_\pi[Forced(F,\pi,a)]}$$

Proof:

$$\begin{aligned}
\mu_a(F) &= \mathbf{E}_\pi[2^{-n}2^{|\mathrm{Forced}(F,\pi,a)|}] \\
&= 2^{-n}\mathbf{E}_\pi[2^{|\mathrm{Forced}(F,\pi,a)|}] \\
&\geq 2^{-n+\mathbf{E}_\pi[\mathrm{Forced}(F,\pi,a)|]}
\end{aligned}$$

where the last inequality follows from the concavity of the exponential function. ■

Combining Lemma 2.7 with Lemma 2.8, we obtain

Lemma 2.9 *If a is a j–isolated satisfying assignment of a k–CNF F,*

$$\mu_a(F) \geq 2^{-n+\frac{j}{k}}$$

V is a set of variables, $a \oplus V$ denotes the assignment obtained from a by changing the value of every $v \in V$.

Definition 2.7 *Let A be a set of assignments. For a variable v, an assignment $a \in A$ is* isolated with respect to v in A *if $a \oplus v \notin A$. a is j–*isolated *if there exist j distinct variables $v_1, \ldots, v_j$ such that, for all i, a is isolated with respect to v_i.*

These definitions follow from translating the graph theoretic notion of neighborhood to assignments by identifying the set of assignments with the hypercube in the natural way. In general, A will be the set of satisfying assignments of a formula F, and isolation will be understood to mean isolation with respect to the set $\mathrm{Sat}(F)$.

The crucial observation is this: *isolated satisfying assignments have critical clauses.* In fact, we show that this relationship is tight.

Lemma 2.5 *Let $a \in \mathrm{Sat}(F)$ and v be a variable. a is isolated with respect to v $\iff$ there exists a critical clause for v at a in F.*

Proof:

$\rightarrow$: If a is isolated with respect to v, then there is some clause C which is false on the assignment $a \oplus v$. Because $a \in \mathrm{Sat}(F)$, a literal corresponding to v must appear in C. From this, we conclude that this literal is true on a, while all other literals in C are false on a. Thus, C is a critical clause for v at a.

$\leftarrow$: If C is a critical clause for v at a, then v corresponds to the only true literal in C on the assignment a. Thus, C is false on $a \oplus v$, and $a \oplus v \notin \mathrm{Sat}(F)$.

■

Lemma 2.6 *Let F be a k–CNF, and let $a \in \mathrm{Sat}(F)$ be isolated with respect to v.*

$$\Pr_{\pi}[v \in \mathrm{Forced}(F, \pi, a)] \geq 1/k$$

.

Definition 2.4 *A clause C is a* critical clause *for a variable v at an assignment a if, when the variables are instantiated according to a, the only true literal in C corresponds to v. In this case, we call v the* critical variable *of the clause C.*

In these terms, we can precisely characterize when unit clauses occur when `Modify` is given a satisfying assignment as input.

Lemma 2.4 *Let v be a variable, and a be a satisfying assignment.*

$v \in \text{Forced}(F, \pi, a) \iff$ v occurs last according to π among the variables in a critical clause for v at a.

Proof:

$\rightarrow$: If v is forced, then there is some unit clause which causes it to be forced. Such a clause can only arise if there is a clause C in F where v appears in C, all other literals in C are set to false by a, and v occurs after all other variables in C in the ordering π. Finally, since C is true on a, the literal corresponding to v must be true on a, so we can conclude that C is a critical clause for v at a.

$\leftarrow$: If such a critical clause C exists and v occurs last among the variables in C, C will be a unit clause for v at the time v is set in `Modify`(F, π, a). ■

With this, we can define forcing independent of the procedure `Modify`.

Definition 2.5 Forced(F, π, a) *is the set of variables v such that v occurs last among the variables in some critical clause for v at a.*

By Lemma 2.4, the two definitions of Forced are equivalent.

Having established that critical clauses determine the appearance of unit clauses in `Modify`, we now show that these critical clauses are linked to the structure of the set Sat(F). This relationship will enable us to show that, under certain circumstances, the set Forced(F, π, a) is large.

Definition 2.6 *If a is an assignment and v is a variable, $a \oplus v$ denotes the assignment obtained from a by changing the value of v. If a is an assignment and*

Using this definition, we can identify other assignments b with the property that $\texttt{Modify}(F, \pi, b) = a$.

Lemma 2.3 *Let a be a satisfying assignment, and b be any assignment. If the set of variables on which a and b differ is contained in the set* $\text{Forced}(F, \pi, a)$*, then* $\texttt{Modify}(F, \pi, b)$ *outputs a.*

Proof: Let $z = \texttt{Modify}(F, \pi, a)$. During the execution of `Modify`, the variables in $\text{Forced}(F, \pi, a)$ are set by unit clauses, by definition. Therefore, those values of a are not examined in producing z, and thus any assignment which agrees with a on the complement of $\text{Forced}(F, \pi, a)$ will produce the output a. ■

2.2 Critical Clauses and Isolation

We now show that the set $\text{Forced}(F, \pi, a)$ is intimately related to the structure of $\text{Sat}(F)$. This relationship will enable us to show lower bounds on the size of the set $\text{Forced}(F, \pi, a)$. Once we know that $\text{Forced}(F, \pi, a)$ is large, Lemma 2.3 will allow us to conclude that there are many assignments b for which $\texttt{Modify}(F, \pi, b)$ will output a, and thus `RandomUC` is likely to find the satisfying assignment a quickly.

We show this relationship in two steps. First, we show that the condition "variable v is in $\text{Forced}(F, \pi, a)$" can be determined using properties of F and a in a way that does not explicitly refer to `Modify` or `RandomUC`. Then, we show that these properties are connected with the structure of the set $\text{Sat}(F)$, determining satisfiability in a neighborhood of assignments related to a.

We continue to focus our attention on the case when $\texttt{Modify}(F, \pi, a)$ is given a satisfying assignment a as input. To help us characterize which variables are forced, we define:

Lemma 2.2 *Let F be a satisfiable formula, and let $I = \Omega(n\frac{1}{\mu(F)})$.* `RandomUC`$(F, I)$ *outputs a satisfying assignment with probability $1 - e^{-n}$.*

Proof: If F is satisfiable, the probability that `Modify` returns "Failed" on any call made by `RandomUC` is $(1 - \mu(F))$. Thus, the probability that it fails on all I calls is $(1 - \mu(F))^I \leq e^{-I\mu(F)} \leq e^{-n}$. ■.

Lemma 2.1 implies that the running time of `RandomUC` will be determined up to a polynomial term by the number of calls made to `Modify`. In turn, Lemma 2.2 states that this number is determined by $\mu(F)$. To compute $\mu(F)$, we consider events where `Modify` not only succeeds in finding some satisfying assignment, but finds a specific satisfying assignment a.

Definition 2.2 *For any $a \in \mathrm{Sat}(G)$, let $\mu_a(G)$ be the probability that* `Modify`(G, π, b) *returns a when π and b are chosen uniformly at random.*

Since the events that `Modify` returns a and $b \neq a$ are disjoint events, it is easy to see that $\mu(F) = \sum_{a \in \mathrm{Sat}(F)} \mu_a(F)$.

Intuitively, any advantage that `Modify` has over random guessing must arise from the appearance of unit clauses which determine some variable, so a useful analysis of `Modify` must prove something about the occurrence of these unit clauses.

We first consider the execution of `Modify`(F, π, a) in the case when a is actually a satisfying assignment. In this case, `Modify` returns the assignment a: `Modify` sets variables according to the assignment given as input unless a unit clause dictates otherwise. If a unit clause required setting a variable differently from a, then that clause would be false on a, contradicting the assumption that $a \in \mathrm{Sat}(F)$.

With this in mind, we define:

Definition 2.3 $\mathrm{Forced}(F, \pi, a)$ *is the set of variables which are set because of unit clauses during the execution of* `Modify`(F, π, a).

2.1 RandomUC

We consider the following simple algorithm, consisting of two procedures. Given a formula, an ordering of the variables, and an initial assignment, `Modify` attempts to modify the assignment to satisfy the formula. `RandomUC` calls `Modify` repeatedly with different initial assignments and orderings of variables.

Procedure `Modify`(CNF formula $G(x_1, x_2, \ldots, x_n)$,
permutation π of $\{1, 2, \ldots, n\}$, assignment b)
returns assignment z
$G_1 = G$.
for $i = 1$ **to** n
if G_i contains the unit clause $x_{\pi(i)}$
then $z_{\pi(i)} = 1$
else if G_i contains the unit clause $\bar{x}_{\pi(i)}$
then $z_{\pi(i)} = 0$
else $z_{\pi(i)} = b_{\pi(i)}$
$G_{i+1} = G_i$ after substituting $x_{\pi(i)} = z_{\pi(i)}$
end /* end for loop */
if z satisfies F
then return(z);
return("Failed")

`RandomUC`(CNF-formula F, integer I)
repeat I times
π = uniformly random permutation of $1, \ldots, n$
a = uniformly random vector $\in \{T, F\}^n$
z = `Modify`(F, π, a);
if ($z \neq$ "Failed")
then output(z); exit();
end/* end repeat loop */
output("Unsatisfiable");

A few basic facts about this algorithm enable us to focus our attention:

Lemma 2.1 *The running time of* `Modify` *is poly*$(n)|F|$.

Definition 2.1 $\mu(G)$ *is the probability that* `Modify`(G, π, a) *succeeds (that is, it does not return "Failed") when* π *and* a *are chosen uniformly at random.*

Chapter 2

Satisfiability

In this chapter, we propose and analyze algorithms for the k–SAT problem. These algorithms will primarily be randomized versions of the DLL procedure [6], meaning that they operate by setting variables one at a time, simplifying the formula at each step. By "randomized", we mean that on *every* input, the algorithm succeeds with high probability over the random choices made by the algorithm. While a number of heuristics for satisfiability make use of randomness, this analysis bounding the success probability of such a scheme is novel.

For convenience, we first fix some basic notation: Throughout, F denotes the CNF formula given as input to the algorithm, k is the maximum width of a clause in F (ie, F is a k–CNF), and $x_1, \ldots, x_n$ are the variables on which F is defined. As noted above, we use $F(x_1, \ldots, x_n)$ to denote both the Boolean formula and the Boolean function it computes.

This chapter presents the algorithms `RandomUC` and `ResolveSAT` together with their analyses. We first consider the simpler algorithm `RandomUC`, and analyze its running time. In doing so, we develop many of the techniques used in the more involved analysis of `ResolveSAT`.

This result also allows us to relate algorithms for 3–SAT to algorithms for k–SAT for larger values of k. As noted earlier, any k–SAT problem can be transformed into a 3–SAT problem by introducing additional variables. However, additional variables are needed for each *clause* of the k–CNF; in the worst case, the number of variables increases dramatically. Because this reduction is inefficient in this regard, even a very efficient algorithm for 3–SAT does not immediately yield efficient an efficient algorithm for k–SAT for larger values of k.

Once again, this algorithmic characterization also has consequences for complexity. The approach we use to proving lower bounds on the size of Σ_3^k circuits has two parts: characterizing the sets accepted by k–CNFs, and identifying functions which are hard based on this characterization. In this second step, the goal is to find functions f which "defeat" all k–CNFs trying to make progress towards computing f. By showing that the class of all k–CNF formulae can be re-expressed using a much smaller subclass of formulae, we simplify the search for a hard function f, since f need only defeat this small subclass.

In particular, this Sparsification Lemma allows us to transform Σ_3^k circuits into ones whose depth-2 subcircuits are sparse k–CNFs with only a small increase in size. As a result, we are able to improve on standard counting arguments showing the existence of hard functions for Σ_3^k circuits. While we do not obtain an explicit hard function, we show that a random degree-2 GF(2) polynomial requires large Σ_3^k circuits. By exploiting the fact that this family of functions is itself quite simple to describe, we give a "pseudorandom" generator of hard subsets for Parity. This algorithm takes some advice, then produces a small set of assignments with the property that computing Parity correctly on this set is hard.

satisfiability algorithms, we then show that if f has certain properties, than no such k–CNF exists. From this, we are then able to conclude that f has no small Σ_3^k circuits.

Using this approach, we improve known lower bounds on depth–3 circuits. We present the results of [20, 19], which obtain a tight (up to constant factors) lower bound on the size of both Σ_3^k and Σ_3 circuits computing parity. By considering other functions based on error-correcting codes, we obtain even stronger lower bounds. Finally, using a related approach inspired by the satisfiability algorithms for 2–CNFs, we also present a result from [21], proving a strongly exponential lower bound of $2^{n-o(n)}$ on the size of Σ_3^2 circuits.

1.3 CNFs

At the center of both of these applications are questions about the expressive power of k–CNFs. In order to gain a better understanding of this power, we begin by trying to identify a hard subclass of k–CNF formulae. Our notion of hardness here is quite strong: By hard, we mean a subclass of k–CNF formulae with the property that good algorithms for solving satisfiability on this subclass would imply good algorithms for solving satisfiability for all k–CNFs. Here, we present some of the results of [13] which show that the subclass of *sparse* formulae (that is, formula for which the number of clauses is linear in the number of variables) has this property.

Our main tool in this endeavor is a Sparsification Lemma, which reduces arbitrary k–CNFs to sparse k–CNFs. More precisely, this lemma states that any k–CNF can be expressed as the union (that is, the OR) of several sparse k–CNFs. This confirms that such sparse formula are the hard instances of satisfiability: For any k–CNF given as a satisfiability problem, we can convert it into several sparse formulae. If we can solve satisfiability on sparse instances, by solving each of of these new problems we obtain an answer to our original problem.

the depth, however, causes the size of the circuit to grow dramatically, from $O(n)$ to $2^{O(n/\log\log n)}$ gates. At the same time, the construction also guarantees that the bottom fanin of the depth–3 circuit is at most n^ϵ for any $\epsilon > 0$. Thus, a sufficiently strong lower bound on the size of depth–3 circuits with this limited fanin would imply a nonlinear lower bound on logarithmic depth NC-style circuits, a longstanding open problem. If the original NC-style circuit has the additional property that it is a series-parallel construction (corresponding to a large class of natural constructions), then the resulting depth–3 circuit satisfies even stronger properties. If we allow it to have size $2^{n/2}$, then we can require that its bottom fanin be bounded by a constant. Therefore, strong lower bounds even for depth–3 circuits with constant bottom fanin would have important consequences.

From this, we can see the problem of proving lower bounds of $2^{\epsilon_k n/k}, \epsilon_k > 1$ on Σ_3^k circuits as an important challenge: As mentioned above, known techniques prove weak results of this form, with $\epsilon_k < 1$. It also provides a step towards resolving the open problems mentioned above, since both resolving the question of [11] and applying the circuit modification proposed by [29] will require proving bounds where this ϵ_k increases with k.

Motivated by this goal, we prove lower bounds on the size of Σ_3^k circuits computing certain explicit functions. To do so, we use a *top-down* argument (in this terminology, the output gate is at the top, and the inputs are at the bottom). Let f be the function for which we want to prove a lower bound. For each input accepted by the circuit, we can associate some Π_2^k subcircuit, or k–CNF, responsible for the decision to accept that input. If the circuit is not too large, then there must exist some k–CNF which is responsible for accepting many inputs x for which $f(x) = 1$. This reduces the problem of proving lower bounds on the size of Σ_3^k circuits to a question about the expressive power of k–CNFs: Is there a k–CNF which can be responsible for accepting many inputs for which $f(x) = 1$? Using the limitations on the expressive power of k–CNFs developed in analyzing the

With our focus on k–CNF formulae, a natural direction is to require the depth–2 subcircuits to be k–CNFs, rather than arbitrary CNF formulae. We define Σ_d^k and Π_d^k to be Σ_d and Π_d circuits with bottom fanin (ie, nearest to the inputs) at most k. With this notation, we will attempt to first prove lower bounds on Σ_3^k circuits using properties of k–CNF formulae and then extend them to arbitrary Σ_3 circuits by other means. Even with this restriction, however, our understanding is limited. When parameterized by k, known techniques, including the ones described above, yield lower bounds of the form $2^{\epsilon n/k}$, with $\epsilon < 1$. This limitation was explicitly recognized by [11], which posed as an open problem the problem of finding lower bounds of this form with $\epsilon = \omega(1)$ for any $k \leq \sqrt{n}$.

In addition to this technical challenge, there is a more direct reason for considering depth–3 circuits of limited bottom fanin. Valiant [29] demonstrated that sufficiently strong lower bounds on such circuits would have important consequences. Consider an alternative model of circuits, which we will refer to as NC–style circuits, in which the AND and OR gates are limited to at most 2 inputs. This limited fanin more naturally corresponds to the devices used in modern circuits or natural parallel computation. However, little is known about the limitations of such circuits. A nontrivial NC–style circuit must have $\Omega(n)$ gates and depth $\Omega(\log n)$, since otherwise the fanin limitation would imply that some input is not capable of reaching the output gate. It remains an open question to improve on this by showing a nonlinear lower bound on the size of an $O(\log n)$ depth NC–style circuit required to compute some explicit function.

What Valiant observed is that the graph of any linear-size, logarithmic-depth NC-style circuit must have a small set of edges which when removed separate the graph into components of small depth. By trying all possible values on edges crossing the cut, and verifying that these choices are consistent with the values of the other cut edges and with the inputs, he obtained a Σ_3 AC-style circuit computing the same function as the original NC-style circuit. This collapse of

Π_2 circuits. Σ_2 and Π_2 formulae are complete, meaning that any Boolean function can be represented in this fashion. Thus, depth–2 circuits already have as much expressive power as is possible.

However, in order to achieve this expressive power, the depth–2 circuits must have size exponential in the number of inputs. This can be shown non-constructively by counting arguments showing that the number of circuits of size $2^{n-o(n)}$ is much less than the number of functions, implying that most functions do not have small circuits. More importantly, there are specific functions, such as Parity, which can be shown to require large depth–2 circuits. Given this, a natural question is whether circuits of depth greater than 2 can provide the same expressive power with smaller size. Already, however, our understanding is limited. The same counting argument shows that unless the circuits have size $2^{n-o(n)}$, they cannot compute all functions, even if the depth is allowed to grow. Unlike the case of depth–2 where Parity provides an example of a hard function, however, there is no explicit function which is known to require truly exponential circuits of even depth 3. Our results are aimed at addressing this lack of knowledge by improving lower bounds on depth–3 circuits.

For AC-style circuits of constant depth, the best known results are obtained using the Switching Lemma [10], which builds on the results of [8, 31]. This result says that any small constant depth circuit can be made constant by setting many, but not all, of the input variables, which implies that it cannot compute Parity. Using this lemma, it was also shown in [10] that computing Parity requires depth d circuits of size $2^{\epsilon n^{1/(d-1)}}$ for some constant $\epsilon > 0$. For the case of depth 3 circuits, this yields a lower bound of $2^{\epsilon\sqrt{n}}$. Using different techniques, [11] improved this lower bound on depth–3 circuits to $2^{0.618\sqrt{n}}$ for Parity, and $2^{0.849\sqrt{n}}$ for the closely related Majority function. However, this is still quite far from the strongly exponential $2^{n-o(n)}$ lower bounds obtained by counting arguments.

Given this difficulty, our only recourse is to restrict the problem further.

respectively.

1.2 Circuits

In addition to providing a language for talking about problems, Boolean formulae also can be used directly to represent functions. In modeling computation, a more natural representation is Boolean circuits, which extend formulae by allowing for the reuse of intermediate results. The number of the ANDs and ORs (called gates in this context) in the formula or circuit offers a natural measure of the complexity of the circuit. Our goal is to show that certain functions are hard to compute by showing these functions require circuits of large size.

The model of circuits we consider here, which we call AC-style because of its similarities to the circuits used in the definition of the complexity class AC, are composed of unbounded fanin AND and OR gates. Each circuit operates on a fixed number of inputs and produces a Boolean output; a function is then computed by a family of such circuits, one for each input size. The circuits do not use NOT gates. Instead, we assume that the circuit has access to both positive and negative copies of the input variables.

The complexity of these circuit families are measured in terms of the number of gates in the circuit (the *size* of the circuit), and the length of the longest path from inputs to output (the *depth* of the circuit) as functions of the number of inputs. We will restrict our attention to circuits of at most a given depth, and then prove bounds on the size of such circuits required to compute the desired functions.

Because the gates in this model have unbounded fanin, we can assume that a circuit consists of alternating layers of AND and OR gates, since an AND of ANDs can simply be replaced by a single, larger AND. For a depth d circuit, we write Σ_d if the output gate is an OR, and Π_d otherwise. With this notation, CNF formula (without restrictions on the width of the clauses) correspond naturally to

partial assignment α assigning values to some subset X' is *autark* if every clause containing a variable from X' is satisfied by α, and thus removed when the variables are set according to α. For example, an assignment which sets one variable to true is autark if that variable does not appear negated in the formula.

If there is such an autark partial assignment α, then if the formula is satisfiable, there exists a satisfying assignment in which the variables in X' are assigned values according to α. In considering how to assign values to the variables in the clause C, either there exists an autark assignment, or not. If there is an autark assignment, we need only consider that assignment, avoiding the branching which leads to exponential growth in the running time. If there is not, though, there is still something to be salvaged: if a partial assignment is not autark, then setting variables according to it must cause some clause to become shorter. This is helpful to the algorithm, since branching on the $2^{k-1}-1$ ways of satisfying a length $k-1$ is better than branching on the 2^k-1 ways of satisfying a length k clause. Using this observation, they obtain an algorithm with an improved running time.

Following this, a sequence of papers [24, 15, 32] focussed on the case of 3–SAT. By careful choice of branching steps made by the algorithm, significantly improved running times were obtained, culminating in the $O(2^{0.589n})$ shown in [15] and the $O(2^{0.582n})$ claimed recently by [25]. These algorithms make use of more and more sophisticated and complex branching rules in order to obtain improved running times.

In this dissertation, we present a new approach for analyzing the performance of satisfiability algorithms, particularly randomized DLL variants. Using this, we present an analysis of `RandomUC`, a natural randomized algorithm for satisfiability. We show that on k–CNF formula with n variables, its running time is at most $2^{(1-1/k)n}$. With similar techniques, we show that `ResolveSAT`, which adds a preprocessing phase to `RandomUC`, obtains significantly better performance. These results on `RandomUC` and `ResolveSAT` originally appeared in [20] and [19],

performs basic simplifications which follow from setting a variable to a constant: for example, an OR containing a literal set to true can be replaced by the value true.

[6] also introduced a number of rules for organizing this search. One important rule we will make use of is the *unit clause rule*: After instantiating variables according to the current assignment, if there is some clause of length 1, set that variable to make the clause true. Clearly, when such a clause exists in the formula, there is only one way that the variable can be set which has any hope of satisfying the formula. Despite the simplicity of this description, analyzing these procedures has proved quite difficult.

As an example of such procedures, consider the following algorithm sketch.

```
Procedure Simple(F)
  if no nonconstant clauses remain
    if F is satisfied return true
    else return false
  let C = x_1 ∨ ... ∨ x_l be an arbitrary clause in F
  for each assignment α to x_1 ..., x_l except the one falsifying C
    F' = F after making the assignment α to {x_1, ..., x_l}
    if Simple(F') = T
      return true
  return false
```

Using induction on the number of variables, it is not difficult to show that in the worst case the algorithm runs in $\text{poly}(n)2^{cn}$ steps with $c = \frac{\log_2(2^k-1)}{k}$, something which certainly need not be true of arbitrary sets of assignments. Thus, this represents a limitation on the expressive power of such formula, which the algorithm exploits to direct its search for satisfying assignments. Our results will follow this philosophy, identifying places where the formula limits the possible set of satisfying assignments and using these limitations to direct the search.

The earliest results in this area are due to [18], who improve on this simple algorithm. Their algorithm makes use of the following observations: A

PROBLEM: Is there an assignment α satisfying C?

This begs the question: Can the k–SAT problem be solved quickly?

The NP-Completeness of k–SAT offers a partial negative answer: k–SAT can be solved in polynomial time only if every problem in NP can. This was shown in [5] and [16] by transforming the computation of a nondeterministic Turing machine M into a CNF formula F in a way that ensures that F is satisfiable if and only if the computation accepts the given input. If M runs in polynomial time in the input size, solving the problem in NP, then the number of variables is likewise polynomial, so the reduction is efficient. Then, using a standard trick, F can be transformed into a 3–CNF F' so that F is satisfiable if and only if F' is, and the number of variables is increased by only a polynomial factor.

In the other direction, an upper limit on the hardness of solving satisfiability is given by a simple brute-force algorithm: Generate all 2^n assignments to the n variables in F, then check each to see if it satisfies F. Such an algorithm clearly solves the satisfiability problem, and runs in time $O(\text{poly}(n)2^n)$. The trivial nature of this brute-force algorithm suggests that better results might be possible, and indeed, many proposals for improving it have been advanced. For an overview of this area, we refer the reader to the comprehensive survey [9]. These results include an array of heuristics which offer improved performance in various circumstances.

In keeping with our motivating example, however, here we will focus our attention on algorithms for which a bound on the running time of the algorithm in the worst case is known. Such algorithms largely fall into the framework of the Davis-Putnam [7] or Davis-Logemann-Loveland (DLL) [6] procedures, originally developed to study the closely related problem of proving unsatisfiability of formulae using the resolution proof system. An algorithm of this type picks a variable, recursively considers the formula where this variable is set to true, and then recursively considers the formulae where it is set to false. Along the way, it

that they are constructed in a simple fashion from subcircuits corresponding to CNFs. Our characterizations of the expressive power of such formula then allow us to conclude that no subcircuit is capable of making significant progress towards computing the desired function. Thus, many such subcircuits are required, and any circuit computing this function must have large size.

To motivate this study of the limitations of k–CNF formulae, we structure our presentation around satisfiability and circuit complexity, developing results about the formulae as required by the applications.

1.1 Satisfiability

Because of its central role in NP-Completeness, the satisfiability problem is a natural choice as a language for expressing many other problems. In this dissertation, we present new analyses and algorithms yielding running times which, while still exponential in the number of variables, improve substantially over known results.

First, however, we will need some basic definitions about Boolean formulae. A *literal* refers to a variable or the negation of a variable. A *clause* is the OR of a collection of literals. A Boolean formula is in *conjunctive normal form (CNF)* if it is the AND (conjunction) of a collection of clauses. If each clause contains at most k literals, we say that the formulae is a k–CNF. An *assignment* is a map from the variables to the set $\{T, F\}$, assigning a Boolean value to each variable. Finally, an assignment *a satisfies F* if F is made true by a. For CNFs, this means that every clause contains either a positive literal (ie, variable) assigned the value true or a negated literal assigned the value false.

For the sake of completeness, we define:

k–SAT

INSTANCE: Set $X = \{x_1, \ldots, x_n\}$ of variables, collection C of clauses over X such that each clause $c \in C$ has $|c| \leq k$.

k-CNF and 3-CNF formulae are natural languages for expressing other problems, and thus understanding the expressive power of these languages is an important problem.

This dissertation explores the limitations on the expressive power of these k-CNF Boolean formulae. By a variety of known techniques, one can show that not all problems can be expressed as the satisfiability of a k-CNF formula on a small number of variables, implying that such limits do exists. The main technical contributions of this dissertation sharpen our understanding of these limitations by providing specific, applicable situations where these limitations manifest themselves. We relate features of the formula to features of the set of satisfying assignments, showing that the existence of certain configurations of clauses determine certain properties of the set of satisfying assignments. In addition, we show that these characterizations are constructive, meaning that an algorithm can recognize and exploit situations in which they provide useful information in locating satisfying assignments.

The insights obtained in this way lead to improved results in two very distinct areas. In examining algorithms for satisfiability, we can view these characterizations as limiting the power of an adversary trying to construct formulae for which satisfiability is hard. Specifically, we show that in trying to construct such a formula, the adversary has two choices. If he constructs a formula with many satisfying assignments, then finding one will be easy. If the formula has few satisfying assignments, by relating the formula to this set of satisfying assignments, we show that the formula must contain hints which help an algorithm find these satisfying assignments. This insight enables us to prove upper bounds on the running time of satisfiability algorithms. In the opposite direction, these characterizations enable us to show that certain natural functions are hard by proving lower bounds on the size of circuits needed to compute them. We focus our attention on the important restricted class of depth-3 Boolean circuits. These circuits have the property

linear programming example, certain problems with a nonlinear structure cannot be expressed using only linear constraints.

This argument implies that such languages for expressing problems have some intrinsic tension: the problems they yield are solvable only to the extent that the expressive power of the language is limited. Because of this, studying the expressive power of the mathematical language is intimately related to understanding algorithms which solve problems expressed in this language. Again appealing to our linear programming example, the algorithms for solving this problem are closely related to insights about the expressive power of linear inequalities; for example, they rely heavily on the fact that the polytopes defined in this way are convex bodies.

In this dissertation, we will follow this approach of investigating the expressive power of a particular language for describing problems. Rather than the linear constraints used in the example above, we consider a natural class of simple Boolean formulae which play a central role in our understanding of complexity theory.

In complexity, this translation between problems is known as a *reduction*. Using these reductions, a large class of problems (called NP-Complete) are known to be efficiently expressible in terms of one another, and thus at some level are equivalent both in expressive power and in the difficulty of finding solutions. The starting point for these reductions is the observation that logical expressions can be used to model computations. Given any problem in NP, we express it using the language of Boolean formulae. A Boolean formula can be constructed which models the nondeterministic algorithm; the algorithm accepts the input if and only if the corresponding formula is satisfiable. Careful examination of this construction reveals that this can be accomplished using only a restricted subclass of Boolean formulae: k-CNF, or formulae in conjunctive normal form with at most k literals per clause; in fact, $k = 3$ suffices for this construction. Because of this generality,

Chapter 1

Introduction

Consider the following problem-solving paradigm: Fix some mathematical language for defining problems, such as some particular class of equations. Then, given some real problem, express it in this mathematical language by devising a mathematical problem whose solutions correspond to solutions to the natural problem. Then, if we apply some efficient algorithm for solving this mathematical problem, we obtain solutions to our real problem. For an example, many practical problems are attacked by writing systems of linear inequalities which capture the constraints of the problem, and then using well-known algorithms to quickly find feasible, even optimal, points satisfying these inequalities.

Now suppose that we begin with a provably difficult, or even unsolvable, problem. If we then follow this recipe, we obtain an efficient solution to this problem, contradicting our assumption that this problem was provably hard. One possibility is that the algorithm for solving the mathematical problem is in some way flawed: it is inefficient, or does not work all the time. If this is not the case, that is, there does exist a reliable, efficient algorithm for solving the mathematical problem, then only other possibility is that the difficulty lies in the mathematical language used to express the problem. This language is either incapable of expressing the problem, or can only do so in an inefficient manner. In the case of our

contribution of each subcircuit. Using this approach, we show tight lower bounds on the size of such circuits computing the parity function and even stronger lower bounds for certain functions based on error-correcting codes. In addition, inspired by ideas used in polynomial-time algorithms for the satisfiability of 2-CNFs, we obtain strongly exponential lower bounds on the size of depth-3 circuits with bottom fanin 2.

We conclude by examining k-CNFs directly in an attempt to identify the formulae which give rise to difficult satisfiability problems. We show that any k-CNF can be written as the union of several sparse k-CNFs, meaning k-CNFs with $O(n)$ clauses. This implies that such sparse formulae are the hard instances, since any other instances can be reduced to them. In addition, this technique allows us to restructure depth-3 circuits with limited bottom fanin, improving counting arguments showing the existence of hard functions.

ABSTRACT OF THE DISSERTATION

Circuits, CNFs, and Satisfiability

by

Francis Zane
Doctor of Philosophy in Computer Science
University of California, San Diego, 1998
Professor Ramamohan Paturi, Chair

This dissertation explores the connections between two seemingly unrelated problems: showing upper bounds on the running time of algorithms which solve the NP-complete satisfiability problem, and proving lower bounds on the complexity of Boolean circuits computing certain explicit hard functions. We show that both results follow from understanding the limitations of the expressive power of certain classes of Boolean formulae.

We begin by analyzing `RandomUC`, a natural randomized algorithm for finding satisfying assignments of a k-CNF formula F, and show that, if F is satisfiable, `RandomUC` finds a satisfying assignment in time $\mathrm{poly}(n)2^{(1-1/k)n}$, significantly smaller than the $\mathrm{poly}(n)2^n$ required for exhaustive search. We then propose a new algorithm `ResolveSAT`, which adds a preprocessing phase to `RandomUC`. While the running time of `ResolveSAT` is still exponential, we prove that its running time is smaller than previous algorithms for satisfiability. For the important case of 3-CNF formulae, we show that its running time is at most $2^{.446n}$.

Turning to circuit complexity, we prove lower bounds on the size of depth-3 circuits of unbounded fanin AND and OR gates which compute specific Boolean functions. Since such circuits are composed of depth-2 CNF subcircuits, the characterizations developed in analyzing satisfiability algorithms allow us to bound the

VITA

1993	B.S. Massachusets Institute of Technology
1998	Doctor of Philosophy University of California, San Diego

PUBLICATIONS

F. Zane, P. Marchand, R. Paturi and S. Esener. Scalable Network Architectures Using the Optical Transpose Interconnection System. In *Proceedings of the 3rd International Conference on Massively Parallel Processing Using Optical Interconnects (MPPOI)*, pages 114-121, Maui, Hawaii, October 1996.

R. Paturi, M.E. Saks, and F. Zane. Exponential Lower Bounds for Depth 3 Boolean Circuits, In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, pages 151–158, El Paso, Texas, 4–6 May 1997.

R. Paturi and P. Pudlák and F. Zane. Satisfiability Coding Lemma, In *Proceeding of the 38th Annual Symposium on Foundations of Computer Science*, pages 566–574, Miami Beach, Florida, 20–22 October 1997.

R. Paturi and F. Zane. Dimension of Projections in Boolean Functions. *Siam Journal of Discrete Mathematics.* To appear.

L.R. Matheson, S.G. Mitchell, T.G. Shamoon, R.E. Tarjan, and F.X. Zane. Security of Digital Watermarks In *Proceedings of Financial Cryptography '98*, Anguilla, BWI, 23-25 February 1998.

J. Kilian, F.T. Leighton, L.R. Matheson, T.G. Shamoon, R.E. Tarjan, and F. Zane. Resistance of Digital Fingerprints to Collusional Attacks. In *Proceedings of 1998 IEEE International Symposium on Information Theory* Cambridge, MA, 16-21 August 1998.

R. Impagliazzo and R. Paturi and F. Zane, Which Problems Have Strongly Exponential Complexity?, To appear in *Proceedings of the 39th Annual Symposium on Foundations of Computer Science*, Palo Alto, California, 8–11 November 1998.

R. Paturi and P. Pudlák and M.E. Saks and F. Zane, An Improved Exponential-time Algorithm for k-SAT, To appear in *Proceedings of the 39th Annual Symposium on Foundations of Computer Science*, Palo Alto, California, 8–11 November 1998.

ACKNOWLEDGEMENTS

First and foremost, I would like to thank Mohan, who has been a wonderful advisor, mentor, and research collaborator. I am grateful for having had the opportunity to work with him. I would also like to thank Russell, who has also been a source of guidance and insight during my time in San Diego. Much of this research was done in collaboration with Michael Saks and Pavel Pudlák. Their discussion, comments, and contributions have been invaluable in shaping this work.

I would also like to thank Sam Buss, Mike Freedman, and Sadik Esener. In addition to their role as members of my committee, they have all contributed greatly to my education here, through research, classes, and lectures. I will miss my time in the theory lab with Ted Carson, Giovanni Di Crescenzo, Markus Jakobson, Tassos Dimitriou, and many others, and I wish them all the best for the future.

Finally, I would like to thank my family for their support, and Donna, for making it all worthwhile.

TABLE OF CONTENTS

The dissertation of Francis Zane is approved, and it is acceptable in quality and form for publication on microfilm:

_____

_____

_____

_____

_____

Chair

University of California, San Diego

1998

UNIVERSITY OF CALIFORNIA, SAN DIEGO

Circuits, CNFs, and Satisfiability

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy
in Computer Science

by

Francis Zane

Professor Ramamohan Paturi, Chair

Professor Samuel Buss

Professor Sadik Esener

Professor Michael Freedman

Professor Russell Impagliazzo

1998