Today: In the previous two lectures, we mentioned ETH and SETH, about the time complexity of SAT. Today we want to talk about the best known theoretical algorithms and techniques for solving SAT! (In practice, SAT is often solved very efficiently. But there are instances in practice that make SAT solvers trip up.)

1 Randomized reduction: a first start

We'll start with an algorithm that beats 2^n in a special case. It will illustrate some useful principles in some algorithms to come.

Theorem 1.1 3-SAT *can be solved in* $O^*((3/2)^t)$ *time on formulas with* $\leq t$ 3*-clauses.*

(3-clause = clause with three literals)

Proof. Here's an algorithm, then we'll explain it.

ALG(F)
Repeat for $20 * (3/2)^m$ times:
    For every 3-clause $C = (L_1 \vee L_2 \vee L_3)$ in F,
        Randomly choose an L_i, remove it from C forming new 2-clause C'; add C' to F.
    End for
    Solve the remaining 2-SAT instance F'. If satisfiable, return "SAT".
End repeat
Return "UNSAT".

Now for the analysis. Suppose A is a satisfying assignment to F.

Claim 1.1 *For all clauses* C*,* $Pr[A$ *satisfies* $C'] \geq 2/3$*.*

Proof of Claim: In the worst case, exactly one literal in C is satisfied by A. This literal is removed with probability $1/3$. If you remove any other literal, the remaining clause is still satisfied.
QED

Since each literal removed is an independent choice:

$$Pr[F' \text{ is SAT } by A] \geq (2/3)^t.$$

We repeat the inner loop for $r = 20 \times (3/2)^t$ times.

$$Pr[\text{No } F' \text{ is SAT by } A \text{ over all times}] \leq (1-(2/3)^t)^r = (1-(2/3)^t)^{20(3/2)^t} \leq \exp -20(2/3)^t \cdot (3/2)^t = \exp(-20).$$

(Here we used the useful inequality $(1 - x) \leq \exp(-x)$.)

Therefore, when the algorithm reports "UNSAT", the probability it is wrong is less than $e^{-20} < 10^{-9}$.

□

2 Algorithms which beat 2^n

First, we give some simple improvements over exhaustive search. We'll start with branching (a.k.a. backtracking) algorithms, as they are very natural. The idea is that you try to cleverly pick variables to assign, when you assign them they get removed and their removal simplifies the formula. Then you recurse on the simplified formula. The branching paradigm is most like actual real-life SAT solvers. (In the 1990s, local search was the fastest, but in the 2000s folks highly engineered branching algorithms, and they've been the fastest ever since.)

Theorem 2.1 *k-SAT on n vars is in $2^{n-n/O(k2^k)}$ time.*

Proof. Backtracking/branching.

Alg A(F): // F is a k-CNF formula
If F has no clauses, return SAT. If F has an "empty" clause (clause set false), return UNSAT.
Take the shortest clause in F, call it $(x_1 \vee \ldots \vee x_L)$.
For all $2^L - 1$ satisfying assignments a to the clause,
    Call A($F((x_1, \ldots, x_L) = a)$) ($a$ is plugged in – has L fewer variables!)
If some call above returns SAT then return SAT else return UNSAT.

Analysis: In the worst case, the shortest clause is always of length k. Then, our running time recurrence is $T(n) \leq (2^k - 1)T(n/k) + O(\text{poly}(n))$. This easily solves to $T(n) \leq (2^k - 1)^{n/k}\text{poly}(n)$. Note that (using $1 - x \leq e^{-x}$):

$$(2^k - 1)^{n/k} = 2^n(1 - 1/2^k)^{n/k} \leq 2^n e^{-n/(k2^k)},$$

and we are done. □

We can improve slightly:

Theorem 2.2 *k-SAT on n vars is in $2^(n - n/O(2^k))$ time.*

Proof. Branching/backtracking algorithm of Monien-Speckenmeyer 1986.

A(F): // F is a k-CNF formula
If F has no clauses, return SAT. If F has an "empty" clause (clause set false), return UNSAT.
If F is 2-CNF, solve it in polytime and return the answer.
If there is a 1-CNF clause (x), call **A** on $F(x = 1)$.
Take shortest clause $(x_1 \vee \ldots \vee x_L)$.
For $i = 1, \ldots, k$,
    Call **A** on $F(x_1 = 1), F(x_1 = 0, x_2 = 1), \ldots, F(x_1 = 0, x_2 = 0, ..., x_k = 1)$.
If any of the k calls says SAT, then return SAT; else return UNSAT.

Recurrence for the running time: $T(n) \leq \sum_{i=1}^{k} T(n - i) + O(\text{poly}(n))$ (*).

How to solve it? Take $T(n) = 2^{\alpha n}$, try to solve for $\alpha > 0$.

Inductively, we want

$$T(n) \leq \sum_{i=1}^{k} T(n - i) \leq \sum_{i=1}^{k} 2^{\alpha(n-i)} \leq 2^{\alpha n}$$

to be true.

That is, we want $\sum_{i=1}^{k} 2^{\alpha(n-i)} \leq 2^{\alpha n}$.

Dividing by $2^{\alpha n}$, this is $\sum_{i=1}^{k} 2^{-\alpha i} \leq 1$.

By the usual expression for the sum of a geometric series,

$$(1 - 2^{-\alpha(k+1)})/(1 - 2^{-\alpha}) - 1 \leq 1.$$

So we want to know α s.t. $(1 - 2^{-\alpha(k+1)})/(1 - 2^{-\alpha}) \leq 2$, i.e. $1 - 2^{-\alpha(k+1)} \leq 2 - 2^{1-\alpha}$, and hence

$$2^{1-\alpha} - 2^{-\alpha(k+1)} \leq 1.$$

Try $\alpha = 1 - \log(e)/(10 \cdot 2^k)$.

Then $2^{1-\alpha} - 2^{-\alpha(k+1)} = 2^{\log(e)/(10 \cdot 2^k)} - 2^{-(k+1)(1-\log(e)/(10 \cdot 2^k))}$ which equals

$$e^{1/(10 \cdot 2^k)} - 2^{-(k+1)(1-\log(e)/(10 \cdot 2^k))} \leq 1 + 1/(10 \cdot 2^k) - 2^{-(k+1)(1-\log(e)/(10 \cdot 2^k))} < 1.$$

□

In general, the following is useful for analyzing backtracking algorithms:

Theorem 2.3 *Every recurrence of the form*

$$T(n) \leq T(n - k_1) + T(n - k_2) + \cdots + T(n - k_i) + O(\textit{poly}(n))$$

has as a solution

$$T(n) = O(r(k_1, \ldots, k_i)^n \cdot \textit{poly}(n)),$$

where $r(k_1, \ldots, k_i)$ is the smallest positive root of the expression $1 - \sum_{j=1}^{i} x^{-j}$.

For example, consider:

$$T(n) \leq T(n - 1) + T(n - 2) + O(\text{poly}(n)).$$

The respective expression to solve is $1 - 1/x - 1/x^2 = 0$.

Hence we get $x^2 - x - 1 = 0$ which is the same as $x(x - 1) = 1$. Solutions for x are $x = 1.618\ldots, -.618033\ldots$, and hence $T(n) \leq O(1.618^n)$.

3 Improved algorithms for k-SAT.

We will prove the following theorem:

Theorem 3.1 *k-SAT on n vars is in $2^{n - n/O(k)}$ time.*

Improving on the exponent of $n(1 - 1/O(k))$ is a major open problem! For example, is there an algorithm with exponent $n(1 - \log(k)/O(k))$?

Local search and random walks. Schoening'1999 obtained an improved solution of k-SAT using an entirely different strategy. Based on an earlier local search / random walk algorithm for solving 2-SAT, due to Papadimitriou (1991):

Theorem 3.2 *There is a randomized algorithm for 2-SAT running in poly(n) time.*

The algorithm is as follows:

LS(F):
Let A be a random assignment to the n vars of F.
Repeat for $100n^2$ times:
    If F is SAT by A, return "SAT"
    Else pick a clause c that A falsifies
    and pick a variable v in c at random.
    Flip the value of v in A.
End repeat
Return "UNSAT".

Let us prove this algorithm works with good probability. Clearly if LS returns "SAT" then F is SAT.

Claim 3.1 *Suppose F is SAT. Pr[LS returns "UNSAT"] $< 1/10$.*

Let us attempt to prove the above claim. We think of the local search algorithm as a *random walk* on a line graph.

Let A^* be a satisfying assignment to F. Consider a line graph on $n+1$ nodes labeled by $\{0, \ldots, n\}$.

*** PICTURE OF LINE GRAPH ***

We associate each node on the line with a collection of n-variable assignments. Node i corresponds to the set of assignments A' such that $h(A^*, A') = i$, where

$$h(x, y) = \text{ Hamming distance between } x \text{ and } y, \text{ number of bits in which they differ.}$$

Thus:

Node 0 corresponds to A^*, Node 1 corresponds to the n assignments that are like A^*, except have one bit flipped, etc.

How does this graph correspond to the algorithm? At any step, we have an internal assignment that's associated with some node i on the line. When we are not at node 0, we try to "move towards" node 0. We pick a clause that isn't satisfied, and pick a variable, and flip its value. Each clause has two variables, and at least one of them is satisfied by A^*. So we have probability $\geq 1/2$ of moving one bit closer to A^*, and probability $\leq 1/2$ of moving one bit *away* from A^*.

Therefore we can think of the LS algorithm as executing a random walk on this line graph: one step towards node 0 with probability $\geq 1/2$, one step away with probability $\leq 1/2$.

Now we want to know how long it will take for us to reach node 0, after randomly walking back and forth for $10n^2$ steps.

Claim 3.2 *The "cover time" for a random walk on a line graph on n nodes is at most $2n^2$ steps. That is, the expected number of steps needed to visit all nodes on the line is at most $2n^2$.*

Here we'll only give intuition for the proof; we'll go in more detail for the case of 3-SAT.

Proof Intuition: Can use lower bounds on tail of binomial distribution. Suppose we start at any node. Think of our random walk as flipping a coin that comes up heads or tails: if heads we move left, if tails we move right. We want to know how long before we can expect that the number of heads exceeds the number of tails by n. (Then we definitely reached node 0 at some point.)

Define a random var X_i which is 0 if move right in step i, and 1 if move left. Define $X_M = \sum_{i=1}^{M} X_i$. Then, $E[X_M] = M/2$, Stddev$(X_M) = \Theta(\sqrt{M})$.

Claim 3.3 *There are $c, d > 0$ s.t. Pr[for some $M = 1, \ldots, dn^2, X_M \geq n] > c$.*

The idea is that for $M = \Theta(n^2)$, our Stddev becomes $> n$, and with non-zero constant probability, we can show that X exceeds n at some point in the run of M steps. This uses fact that $\sum_{i=0}^{M/2-\sqrt{M}} \binom{M}{i}/2^M > c \ldots$

Why doesn't this algorithm work for 3-SAT?

Well, it does, in a sense, but the random walk probabilities get screwed up! We only have prob $1/3$ of moving towards node 0, and probability $2/3$ of moving away, so we are much more likely to move away...

Nevertheless Schoening'99 found an adaptation of the algorithm which leads to a decent running time for 3-SAT.

4 Schoening's algorithm and its generalization.

We will begin with the generalization of Schoening's algorithm that works for k-SAT.

Theorem 4.1 *k-SAT can be solved in $2^{n-n/O(k)}$ time.*

We will give only the inner loop of the algorithm. Eventually we'll repeat it for some number of times. In general, if we have a procedure that always says "UNSAT" when F is unsatisfiable, and when F is satisfiable says "SAT" with probability P, then if we repeat this procedure $10/P$ times, and say "UNSAT" only if none of the trials returned "SAT", then the new algorithm will be correct with probability $\geq 1 - \exp(-10)$, as we showed earlier. Thus, we will focus on giving procedures that are fast and are correct with exponentially low probability, but when we run them an exponential number of times, they become correct with constant probability. Here is such an algorithm for k-SAT:

LS-S(F):
Choose random assignment A.
Repeat for n/k times:
    If A satisfies F return "SAT"
    If A does not satisfy F, then
        Let C be a falsified clause of F.
        Pick a random variable in C. Flip its value in A.
End repeat.
Return "UNSAT".

Useful Inequalities. For our analyses below, we will use the following inequalities.

1. $1 - x \leq e^{-x}$,
2. $\binom{n}{k} \leq (en/k)^k$,
3. $\binom{n}{\alpha n} \geq 2^{H(\alpha)n}/(n+1)$, where $H(a) = a\log_2(1/a) + (1-a)\log_2(1/(1-a))$ is the binary entropy function,
4. $(a+b)^n = \sum_{i=0}^{n} \binom{n}{i} a^i b^{n-i}$.

Analysis of algorithm. Let A^* be a SAT assignment to F, as before.

Claim 4.1 *$Pr[A^*$ found by algorithm$] \geq 1/(2^n \cdot e^{-n/k+n/k^2}) \geq 1/2^{n-n/ck}$ for some c.*

So if we repeat for $t = O(2^{n-n/ck})$ times, we'll have high probability of finding a SAT assignment.

Proof of Claim:

Let event E be: "Random assignment A is within n/k Hamming distance of A^*". Then $Pr[E] \geq \binom{n}{n/k}/2^n$. This is because there are $\binom{n}{n/k}$ different strings A' which have $h(A^*, A') = n/k$.

Pr[Local search for n/k steps finds $A^*|E] \geq 1/k^{n/k}$, because for each variable picked, there's a $1/k$ chance we have chosen a variable that is flipped to its correct value.

Thus $Pr[A^*$ is found by alg$] \geq \binom{n}{n/k}/2^n \cdot 1/k^{n/k}$.

We now claim that $\binom{n}{n/k}/2^n \cdot 1/k^{n/k} \geq 1/(2^n \cdot e^{-n/k+n/k^2})$.

Noting that $2^{H(1/k)n} = \Theta^*(k^{n/k} \cdot (1/(1-1/k))^{n-n/k})$, we have

$$\binom{n}{n/k}/2^n \cdot 1/k^{n/k} \geq \frac{k^{n/k} \cdot (1/(1-1/k))^{n-n/k}}{k^{n/k}2^n} \cdot \text{poly}(n) \geq 1/(2^n \cdot (1-1/k)^{n-n/k}) \geq 1/(2^n \cdot e^{-n/k+n/k^2}),$$

where we used $1 - x \leq e^{-x}$. QED

The key to the algorithm is this: instead of taking $O^*(\binom{n}{n/k})$ time to try all possible assignments within Hamming distance n/k of a given assignment, the random walk algorithm takes only $O^*(k^{n/k})$ time instead. So over the whole space of assignments, we are "dividing out" $\binom{n}{n/k}$ and replacing it with $k^{n/k}$. This gives the $2^{n/O(k)}$ speedup.

Derandomization. We can derandomize this algorithm by using a deterministic $k^{n/k}$ time algorithm for the local search (branch on all k choices of a literal to flip, for recursion depth of n/k), and using a subset $S \subseteq 0,1^n$ of size $O^*(2^n/\binom{n}{n/k})$, with the property that every n-bit string is within n/k Hamming distance of some string in S (called a covering code).

Improving the above algorithm to get $(4/3)^n$ for 3-SAT. Schoening's algorithm for 3-SAT is as follows:

LS-S2(F):

Repeat $10(n+1)(4/3)^n$ times:
    Choose random assignment A.
    Repeat for $3n$ times:
        If A satisfies F return "SAT"
        If A does not satisfy F, then
            Let C be a falsified clause of F.
            Pick a random variable in C. Flip its value in A.
            [$(1/3)$ probability of choosing "correct" literal to flip]
    End repeat
End repeat
Return "UNSAT".

Theorem 4.2 *Suppose F is SAT. Then, Pr[LS-S2(F) returns "UNSAT"] $< \exp(-10)$.*

Let A^* be a satisfying assignment to F.

Claim 4.2 Pr[*Inner loop returns A^*, starting from A*] $\geq (1/2)^{h(A,A^*)}/(n+1)$.

Recall that $h(x,y) =$ hamming distance between x and y, number of bits in which they differ.

Using the argument from our k-SAT algorithm, we could just say that the above probability is at least $(1/3)^{h(A,A^*)}$: imagine that the algorithm chooses the correct literal each time. It does this with prob $\geq 1/3$, and it only has to do this for $h(A,A^*)$ times.

But in fact the prob can be much higher, because the algorithm could make some mistakes in variable flips, and end up correcting them later.

Assume for now that Claim 4.2 holds.

Claim 4.3 $Pr_A[h(A, A^*) = k] = \binom{n}{k}/2^n$.

We already had this claim in our proof of our k-SAT algorithm.

Now, assuming Claim 4.2,

$$Pr[\text{One repetition returns "SAT"}] \geq \sum_{A \in \{0,1\}^n} Pr[\text{Inner loop returns } A^*, \text{ starting from } A] \cdot (1/2^n)$$

$$= \sum_{k=0}^{n} Pr_A[h(A, A^*) = k] \cdot Pr[\text{Inner loop returns } A^*, \text{ starting from } A] = \sum_{k=0}^{n} \binom{n}{k}/2^n \cdot (1/2)^k/(n+1).$$

The statements above follow from Claim 4.2 and Claim 4.3. The probability equals:

$$1/2^n \cdot (1 + 1/2)^n/(n+1)(\text{by the binomial theorem}) = (3/4)^n/(n+1).$$

Therefore, after $10(n+1)(4/3)^n$ repetitions, we have probability $< \exp(-10)$ of returning "UNSAT" when F is satisfiable.

Proof of Claim 4.2:

Let $t = h(A, A^*)$.

Consider the following event E:

Over a $3t$ step walk, we walk for t steps to the "right" on the hamming distance line (make t "bad" choices of which variable to flip), and and $2t$ steps to the "left" (make $2t$ "good" choices of which variable to flip).

Note that: $Pr[E] \geq (2/3)^t(1/3)^{2t}\binom{3t}{t}$.

Each time we step to the right, it costs probability $(2/3)$, when we step to the left it's prob $1/3$, and there are $\binom{3t}{t}$ possible ways to step t times to the right and $2t$ times to the left.

By introducing another 2^t factor, can rewrite this as: $1/2^t(1/3)^t(2/3)^{2t}\binom{3t}{t}$.

Let $A = (1/3)^t(2/3)^{2t}$ and $B = \binom{3t}{t}$. Now it suffices to show that $A \cdot B \geq 1/(n+1)$. This is basically a combinatorial exercise.

Recall

$$H(\alpha) = \alpha \log_2(1/\alpha) + (1-\alpha)\log_2(1/(1-\alpha)).$$

Hence

$$1/A = 3^t \cdot (3/2)^{2t} = (1/(1/3))^{1/3(3t)} \cdot (1/(2/3))^{2/3(3t)} = 2^{H(1/3)\cdot 3t}.$$

Further recall

$$\binom{n}{\alpha n} \geq 2^{H(\alpha)n}/(n+1) = (1/\alpha)^{\alpha n} \cdot (1/(1-\alpha))^{(1-\alpha)n}/(n+1).$$

Hence $A \cdot B \geq 1/(2^{H(1/3)\cdot 3t}) \cdot 2^{H(1/3)\cdot 3t}/(n+1) = 1/(n+1)$. QED

Some further improvements. De-randomizing Schoening: Moser and Scheder: "A full derandomization of Schoening", STOC 2011, gets $(4/3 + \varepsilon)^n$ time, for all $\varepsilon > 0$. This is the fastest known det alg. for 3-SAT.

The best known randomized algorithm for 3-SAT is due to Hertli in FOCS'11: $O(1.308^n)$ time. Hertli analyzed the PPSZ Algorithm, also known as the Randomized Assignment Algorithm [PPSZ'98]. This algorithm randomly sets variables to values, aggressively check if any variables are implied.

Imagine an algorithm Simplify(F) which given a k-SAT formula, tries to simplify the formula as follows:

1. If there is a clause that contains both x and $\neg x$, remove it, as it is always satisfied.

2. If there is a 1-Clause (ℓ), plug in $\ell = 1$, remove any satisfied clauses, and remove occurrences of $\neg\ell$ from the rest of the clauses. If any clause is empty, return "UNSAT".

3. Generalizing above, go through all subsets S of ≤ 100 clauses and all satisfying assignments to S. If there is some variable y for which all satisfying assignments to S assign y to a fixed $v \in \{0,1\}$, then set $y = 1$, and simplify the formula as above.

4. Several other simple rules.

Then, the PPSZ algorithm looks like this:

PPSZ(F):
Repeat until all vars are assigned:
    Until no more simplifications are possible, repeat: Simplify(F)
    Pick a random unassigned variable x.
    Set x to a random value.
End repeat

The surprising theorem is:

Theorem 4.3 *Pr[PPSZ Algorithm returns a SAT assignment $\mid$ F is satisfiable$] \geq 1/(1.308)^n$.*

Paturi, Pudlak, Saks and Zane (PPSZ) originally proved the above theorem for formulas F that have a unique satisfying assignment. Hertli proved it for all F.

It is known that the PPSZ algorithm requires $\Omega(2^{n-n\log(k)/k})$ time on some instances, as shown by [Pudlak, Scheder, Talebanfard '17].

OPEN PROBLEM: But is that tight? Maybe this algorithm solves k-SAT faster for large k, perhaps in $O^*(2^{n-n\log(k)/k})$ time!

5 An equivalent version of SETH.

Recall that SETH states that for all $\varepsilon > 0$, there is a k such that k-SAT on n vars cannot be solved in $O(2^{(1-\varepsilon)n})$ time.

Consider the following More-Believable SETH (MBSETH): For all $\varepsilon > 0$, there is a c such that CNF-SAT on n variables and cn clauses cannot be solved in $O(2^{(1-\varepsilon)n})$ time.

MBSETH looks more believable as due to the Sparsification Lemma, SETH implies MBSETH. We show that SETH and MBSETH are actually equivalent, a result by Calabro, Impagliazzo and Paturi'06. To show this, we will show the reverse direction, that MBSETH implies SETH:

Theorem 5.1 *Suppose there is a $\delta < 1$ such that for all constant k, k-SAT is in $O(2^{\delta n})$ time (SETH is false). Then there is a $\gamma < 1$ such that for every c, CNF-SAT with cn clauses is in $O(2^{\gamma n})$ time.*

Proof. Suppose there is a $\delta < 1$ such that for all constant k, k-SAT is in $O(2^{\delta n})$ time.

Let F be a CNF Formula with n variables and cn clauses. We will show how to use the $O(2^{\delta n})$ time k-SAT algorithm to also solve SAT for F.

Let k be a parameter to set later, as a function of c.

Let's process F as follows:

If all clauses have length $\leq k$, then solve K-SAT using the $O(2^{\delta n})$ time k-SAT algorithm, and return the answer.
Else, take a clause with $> k$ literals, $C = (x_1 \vee \ldots \vee x_k \vee \ldots)$.
Recursively call F with C replaced with $(x_1 \vee \ldots \vee x_k)$, and also F with $x_1 = 0, \ldots, x_k = 0$ substituted.
If both are UNSAT then return UNSAT, else return SAT.

There are at most n/k branches where k variables are set, and at most cn branches where a clause is shortened. Once we get down to k-SAT, we then run our $O(2^{\delta n'})$ time k-SAT algorithm on the remaining n' variables. So the total running time is at most

$$\sum_{i=0}^{n/k} \binom{cn+i}{i} \cdot 2^{\delta(n-ik)} \leq 2^{\delta n} \cdot \sum_{i=0}^{n/k} \binom{cn+i}{i} 2^{-\delta ik}.$$

Now, $\binom{cn+i}{i}$ is maximized for the largest setting of i, $i = n/k$, and so the runtime is at most

$$2^{\delta n} \binom{cn+n/k}{n/k} \cdot \sum_{i=0}^{n/k} 2^{-\delta ik} \leq O(2^{\delta n} \binom{cn+n/k}{n/k}),$$

as the infinite geometric series converges. Now we apply $\binom{N}{K} \leq (eN/K)^K$ to obtain that the runtime is at most (asymptotically)

$$2^{\delta n} \left(\frac{cn+n/k}{n/k} \right)^{n/k} = 2^{\delta n}(ck+1)^{n/k} = 2^{\delta n} 2^{n \cdot \log(ck+1)/k}.$$

We want to set k so that $\log(kc+1)/k < \varepsilon$ for arbitrarily small $\varepsilon > 0$, so that $\delta + \varepsilon < 1$. Then it would suffice if $\log(2kc) < k\varepsilon$, and so $\log(k) + \log(2c) < k\varepsilon$ and $k\varepsilon - \log(k) > \log(2c)$.

If we set k large enough so that $k\varepsilon/2 \geq \log k$, then we'd just need $k\varepsilon > 2\log(2c)$.

So k needs to be $\geq \max\{2\log k/\varepsilon, 2\log(2c)/\varepsilon\}$. If we set $k = Z \cdot 2\log(2c)/\varepsilon$ for $Z \geq 1$, then $Z\log(2c) = \varepsilon k/2$, and $\log(k) = \log(2Z/\varepsilon) + \log\log(2c)$. Since we want $k\varepsilon/2 \geq \log k$, we thus want $Z\log(2c) \geq \log(2Z/\varepsilon) + \log\log(2c)$ and $Z\log(c) \geq \log(Z) + \log(1/\varepsilon) + \log\log(2c)$. For all $Z \geq 4$, $Z/2 \geq \log(Z)$, so if we set $Z \geq 4$, we would only need $Z\log(c) \geq 2(\log(1/\varepsilon) + \log\log(2c))$. Similarly, whenever c is a large enough constant, $\log(c) > 4\log\log(2c)$, so we only really need $Z\log(c) \geq 4\log(1/\varepsilon)$, so we can set $Z = 4\log(1/\varepsilon)$ and the inequality will be true.

Hence we are setting k proportional to $\log(1/\varepsilon)\log(c)/\varepsilon$, and with our conjectured $O(2^{\delta n})$ time algorithm for k-SAT, we obtain an $O(2^{(\delta+\varepsilon)n})$ time algorithm for CNF-SAT with cn clauses, for any small $\varepsilon > 0$. □

Note: Combining the reduction of this theorem with our k-SAT algorithm from before, this shows we can solve CNF-SAT with cn clauses in $2^{n-n/O(\log c)}$ time!