Boolean Satisfiability
Lecture 4: The Exponential-Time Hypothesis

Edward A. Hirsch*

January 22, 2024

Lecture 4

In this lecture we address three questions:

- How far can we push the exponent for **3-SAT**? 1.3^n? $2^{n/5}$? $2^{n/100}$?...
- How do such exponents for ***k*-SAT** behave? What is the asymptotics as k goes to ∞?
- How do such exponents for other versions of **SAT** behave? What are the relations between them?

Contents

*Ariel University, http://edwardahirsch.github.io/edwardahirsch

1 The Exponential-Time Hypothesis and specific exponents

Given that better and better **3-SAT** algorithms appear, a natural question is: can we push this exponent further and further, or is there some limit where we will stop?

Informally, The Exponential-Time Hypothesis says — yes, we will stop. To formulate it more precisely, let us introduce parameterized **NP** problems (the classical **NP** class is not suitable here as we are talking about exponents w.r.t. some parameter, say, the number of variables, and not w.r.t. the input bit-size).

1.1 Parameterized problems and ETH

Definition 1 (Parameterized NP problems). Consider $L \in \mathbf{NP}$, that is, L is defined by some polynomial-time computable polynomially bounded relation L such that $x \in L \iff \exists y R(x, y)$. A **parameterized problem** (L, q) consists of L and a polynomial-time computable parameter $p\colon \{0,1\}^* \to \mathbb{N} \cup \{0\}$ that bounds the size of the shortest solution (according to R):

$$x \in L \iff \exists y(|y| \leqslant q(x) \wedge R(x, y)).$$

Example 1 (Parameterized problems).

- $(\boldsymbol{k}\textbf{-SAT}, n)$, where n is the number of variables,
- $(\boldsymbol{k}\textbf{-SAT}, m)$, where m is the number of clauses.

Remark 1. **Caution!** This is *not* the same notion as in the field of Parameterized Complexity.

Definition 2 (Subexponential-time decidable problems, SUBEXP). We say that a parameterized problem $(L, \boldsymbol{q}) \in$ **SUBEXP** if for every positive integer $\boldsymbol{t}$, the problem $x \in L$ is decidable in time $\widetilde{O}(\boldsymbol{2^{q(x)/t}})$.

Remark 2. Note that we are talking here about a series of algorithms, one for each t. It means that, for example, the constants and polynomials in $\widetilde{O}$ may be different for different values of t. (Think about the running time $2n2^n$, $2^2n^2 2^{n/2}$, $2^{2^2}n^4 2^{n/4}$, etc.) Also the design of each algorithm may be unique (it is designed by a mathematician), and we cannot provide such a machine description algorithmically, given t.

Alternatively, we could formulate **SUBEXP** instead as "for every small positive $\delta = 1/t$, there is an algorithm A_δ solving (L, q) in time $\widetilde{O}(2^{\delta q(x)})$.

An exponential time hypothesis for a problem (L, q) says that $(L, q) \notin$ **SUBEXP**.

The Exponential-Time Hypothesis, **ETH** is $(\textbf{3-SAT}, n) \notin \textbf{SUBEXP}$, that is, there is $t_* \in \mathbb{N}$ such that we will never be able to solve **3-SAT** in randomized time $\widetilde{O}(2^{n/t})$ for $t > t_*$.

Note that ETH $\Rightarrow$ $\mathbf{P} \neq \mathbf{NP}$, but the inverse is not necessarily true.

1.2 So many exponents!

So what are our limits of improvement? If we believe in ETH, we can introduce notation for this, that is, for a constant δ appearing in the exponent $2^{\delta n}$, and we can do this for various problems. (If we don't believe in ETH, then δ is simply zero.)

We will be using randomized one-sided bounded-error algorithms as our model of computation.

Definition 3. For a time-constructible[1] function τ, $L \in \mathbf{RTime}[\tau(n)]$ if there is a randomized algorithm A that stops in time $O(c|x|^c \cdot \tau(n) + c)$ for a certain constant c. For every input x,

$$\begin{aligned} x \notin L &\Rightarrow A(x) = 0, \\ x \in L &\Rightarrow \Pr\{A(x) = 1\} \geqslant \frac{1}{2}, \end{aligned}$$

This is just a "not-necessary-polynomial-time analogue" of **RP**.

The exponents for *k*-SAT. We can now define the exponent for ***k*-SAT**:

$$s_k \quad = \quad \inf\{\delta \geqslant 0 \,|\, \boldsymbol{k}\textbf{-SAT} \in \mathbf{RTime}[2^{\delta n}]\},$$

Note that we take the infimum, because the existence of a limit is not guaranteed. Now ETH can be reformulated as $\boxed{\textbf{ETH} : s_3 > 0.}$

Where do these constants go when $k \to \infty$? Define

$$s_\infty \quad = \quad \lim_{k \to \infty} s_k.$$

Recall that **SAT** is decidable in time $\widetilde{O}(2^{n(1-1/O(\log(m/n)))})$, but currently we do not know a $\widetilde{O}(2^{(1-1/\mathrm{const})n})$-time algorithm for it. **Strong Exponential-Time Hypothesis** states that we will never know such an efficient algorithm and, moreover, even our ***k*-SAT** algorithms are doomed to become closer and closer to 2^n-time as k grows: $\boxed{\textbf{SETH} : s_\infty = 1.}$

Other versions of SAT. For $f \in \mathbb{N}$, problems **SAT-*f*** and ***k*-SAT-*f*** are defined as subproblems limited to formulas such that the frequency (the number of occurrences) of each variable is bounded by f. (This was not formulated in the lecture, but it is used in these lecture notes for the ease of presentation.) For example, an instance of **3-SAT**-3 is a 3-CNF that contains at most three occurrences of every variable.

If the number of occurrences of every variable is bounded by a constant, then, of course, the number clauses m is bounded by a linear function in n. (We call such formulas *sparse*, and we call the ratio m/n the *density* of a formula.)

---

[1] That is, one can compute $\tau(n)$ within time $O(\tau(n))$ on input 1^n. This is a standard thing when one defines a complexity class.

Define the following constants:

$$
\begin{aligned}
s_k^{\text{freq}.f} &= \inf\{\delta \geqslant 0 \,|\, \boldsymbol{k}\textbf{-SAT}\text{-}f \in \mathbf{RTime}[2^{\delta n}]\}, \\
s_k^{\text{dens}.d} &= \inf\{\delta \geqslant 0 \,|\, \boldsymbol{k}\textbf{-SAT} \text{ for CNFs with at most } dn \text{ clauses} \in \mathbf{RTime}[2^{\delta n}]\}, \\
s^{\text{freq}.f} &= \inf\{\delta \geqslant 0 \,|\, \textbf{SAT}\text{-}f \in \mathbf{RTime}[2^{\delta n}]\}, \\
s^{\text{dens}.d} &= \inf\{\delta \geqslant 0 \,|\, \textbf{SAT} \text{ for CNFs with at most } dn \text{ clauses} \in \mathbf{RTime}[2^{\delta n}]\}, \\
\sigma_k &= \inf\{\delta \geqslant 0 \,|\, \text{Unique } \boldsymbol{k}\textbf{-SAT} \in \mathbf{RTime}[2^{\delta n}]\},
\end{aligned}
$$

Define their limits:

$$
\begin{aligned}
s^{\text{freq}.\infty} &= \lim_{f\to\infty} s^{\text{freq}.f}, \\
s^{\text{dens}.\infty} &= \lim_{d\to\infty} s^{\text{dens}.d}, \\
\sigma_\infty &= \lim_{k\to\infty} \sigma_k.
\end{aligned}
$$

How are all these constants related to each other?

2 Subexponential reducibilities

In order to relate the complexities of parameterized problems, and to do it up to an arbitrary *subexponential* speedup, we need a different type of reductions than just polynomial-time reductions.

2.1 SERF reductions.

Recall that an oracle Turing machine $T^\bullet$ has a special "oracle" state where it queries some blackbox (oracle) function (let us call it $\mathcal{C}$) about some string z (in particular, $\mathcal{C}$ may be a Boolean function, that is, a language). The query is answered by $\mathcal{C}$ in a single step, by providing T with $\mathcal{C}(z)$, so it almost does not waste the time.

When we use $T^\bullet$ with a specific $\mathcal{C}$, we write $T^{\mathcal{C}}$, and when we have none specified, we write just $T^\bullet$.

Note that $\mathcal{C}$ can be replaced by some Turing machine M computing $\mathcal{C}$, and then M can used as a subroutine (so the running time of the combined machine T^M will be $\text{time}_T(x)$ plus the sum of all the running times of M on the queries asked by T when running on input x).

Definition 4 (Subexponential reduction family, SERF). For two parameterized **NP** problems, (A, p) and (B, q), a subexponential reduction family, **SERF**, from (A, p) to (B, q) is a series $\{T_t^\bullet\}_{t\in\mathbb{N}}$ of oracle Turing machines such that

- $T_t^B(F)$ solves the problem $F \overset{?}{\in} A$ in time $\widetilde{O}(2^{p(F)/t})$,
- it asks the oracle about $G \overset{?}{\in} B$ with $q(G) = O(p(F))$ and $|G| = |F|^{O(1)}$.

Why do we have many Turing machines and not just one? Because we want **SUBEXP** to be closed under SERF reductions.

In order for the reductions to be useful, it is desirable that

- they are transitive (that is, a composition of two SERFs is a SERF) — an easy exercise,
- the class **SUBEXP** is closed under them, this is also easy, but let us check it now.

Lemma 1. *If $(B, q) \in$ **SUBEXP** and (A, p) reduces by SERF to (B, q), then $(A, p) \in$ **SUBEXP** as well.*

Proof. Since $(B, q) \in$ **SUBEXP**, for every s, there is an $\widetilde{O}(2^{q(G)/s})$-time machine M_s solving (B, q). Since we have a SERF-reduction, for each t, we have $T_t^\bullet$ as in the definition.

We prove that for each t', we can solve $F \overset{?}{\in} A$ in time $\widetilde{O}(2^{p(F)/t})$. The algorithm is obvious:

Algorithm $\mathcal{A}(F, t')$:
`-- Run the combined machine` $T_t^{M_s}(F)$ `using` M_s `as a subroutine.`

The questions are: what t are we using for T, and what s are we using for $M_{B,s}$.

By definition, T queries its oracle about G_i's with $q(G_i) \leqslant cp(F) + c$ for some constant $c \geqslant 0$. Let $t = 2t'$, $s = 4ct'$.

Since constant-parameter queries can be answered in polynomial time and thus we can assume $p(F) \geqslant 1$, the total time running time of A, up to a polynomial factor ($\widetilde{O}(\ldots)$), does not exceed

$$2^{p(F)/t} + \sum_i 2^{q(G_i)/s} \leqslant 2^{p(F)/t} \cdot \max_i 2^{q(G_i)/s} \leqslant 2^{p(F)/t} \cdot 2^{(cp(F)+c)/s} \leqslant 2^{p(F)/(2t')} \cdot 2^{p(F)/(2t')} \leqslant 2^{p(F)/t'}. \qquad \square$$

To have a specific interesting question about SERF, ask

Does $(\mathbf{3\text{-}SAT}, n)$ SERF-reduce to $(\boldsymbol{k}\mathbf{\text{-}SAT}, m)$?

Indeed, $(\boldsymbol{k}\mathbf{\text{-}SAT}, m)$ could be an easier problem: typically $m \geqslant n$, and indeed $\widetilde{O}(a^m)$-time algorithms exist for it for better a than we currently have for $(\boldsymbol{k}\mathbf{\text{-}SAT}, n)$. We will address this question in Section 3.

2.2 SERF-completeness

Do we have complete problems under SERF reductions? In order to speak about it, we need to tell for what class they are complete (and then provide a complete problem — no surprise that it will be $(\mathbf{3\text{-}SAT}, n)$).

We will prove that it is complete for a parameterized version of **SNP**. The material of this subsection is indeed difficult for non-logicians and is **not needed for the exam**.

Parameterized **SNP** and completeness

Definition 5 (parameterized version of SNP). $(L, q) \in$ **SNP** if L can be defined as

$$(R_1, \ldots, R_u) \in L \iff \exists f_1, \ldots, f_s \; \forall i_1, \ldots, i_t \; \Phi(i_1, \ldots, i_t),$$

where

- f_j's and R_j's are logical relations (arity k_j) over a finite domain $[1\,..\,d]$,
- $i_j \in [1\,..\,d]$,
- formula Φ applies f_j's and R_j's to i_j's.

The parameter q is the total bit-size of f_j's, that is, $\sum_j d^{k_j}$.

Theorem 1. *$(\boldsymbol{k}$-**SAT**$, n)$ is SERF-complete for* ***SNP****.*

Proof. 1. To show $(\boldsymbol{k}$-**SAT**$, n)$ is in **SNP**, let us describe how do we represent $(\boldsymbol{k}$-**SAT**$, n)$ in terms of **SNP**.

- $L = \boldsymbol{k}$-**SAT**, parameter n (the number of variables);
- $\boldsymbol{i_1}, \ldots, \boldsymbol{i_t} \in [1\,..\,n]$, these are counters for **variables indices**,
- input relations R_i: clauses incidence, such as
  $R_{+-+--}(i_1, i_2, i_3, i_4, i_5) = \texttt{True} \iff (x_{i_1} \vee \overline{x}_{i_2} \vee x_{i_3} \vee \overline{x}_{i_4} \vee \overline{x}_{i_5}) \in F$;
- solution relation $\boldsymbol{f}$ (just unary!): **a satisfying assignment**, such as
  $f(1) = \texttt{True}$, $f(2) = \texttt{False}, \ldots,$ $f(n) = \texttt{False}$;
  note that we need exactly n bits (our parameter) to represent it;
- define $f^+(i) = f(i)$ and $f^-(i) = \overline{f(i)}$ (this is just a notation);
- formula $\Phi(i_1, \ldots, i_t)$:

$$\exists \text{ assignment } f \;\; \forall \text{ indices } i_1, \ldots, i_t \in [1\,..\,n]$$
$$\bigwedge_{s_1, \ldots, s_t \in \{+,-\}} (R_{s_1, \ldots, s_t}(i_1, \ldots, i_t) \Rightarrow (f^{s_1}(i_1) \vee f^{s_2}(i_2) \vee \ldots \vee f^{s_t}(i_t)))$$

  That is, if R says that a clause of this type with these variables is present in the formula, then we must satisfy it using f.

Example 2. Classical form of a 2-CNF:

$$(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3) \wedge (\overline{x_1} \vee \overline{x_3}) \wedge (\overline{x_3} \vee \overline{x_2})$$

SNP input relations:

$$\begin{aligned}
R_{++}(i_1, i_2) &: \texttt{False} \\
R_{+-}(i_1, i_2) &: \text{if } i_1 = 1 \text{ and } i_2 = 2 \text{ then } \texttt{True} \text{ else } \texttt{False} \\
R_{-+}(i_1, i_2) &: \text{if } i_1 = 1 \text{ and } i_2 = 3 \text{ then } \texttt{True} \text{ else } \texttt{False} \\
R_{--}(i_1, i_2) &: \text{if } (i_1 = 1 \text{ and } i_2 = 3) \text{ or } (i_1 = 3 \text{ and } i_3 = 2) \text{ then } \texttt{True} \text{ else } \texttt{False}
\end{aligned}$$

SNP solution relations:
just one, an assignment $f\colon \{1,2,3\} \to \{\texttt{False}, \texttt{True}\}$.

SNP Formula Φ:

$\exists$ assignment $f\colon \{1,2,3\} \to \{\texttt{False}, \texttt{True}\}$
$\forall$ indices $i_1, i_2 \in \{1,2,3\}$

$$\begin{aligned}
&\Big(R_{+,+}(i_1, i_2) \Rightarrow (f(i_1) \vee f(i_2))\Big) \wedge && // \text{ nothing} \\
&\Big(R_{-,+}(i_1, i_2) \Rightarrow (\overline{f(i_1)} \vee f(i_2))\Big) \wedge && // \ (\neg x_1 \vee x_3) \\
&\Big(R_{+,-}(i_1, i_2) \Rightarrow (f(i_1) \vee \overline{f(i_2)})\Big) \wedge && // \ (x_1 \vee \neg x_2) \\
&\Big(R_{-,-}(i_1, i_2) \Rightarrow (\overline{f(i_1)} \vee \overline{f(i_2)})\Big) && // \ (\overline{x_1} \vee \overline{x_3}),\ (\overline{x_3} \vee \overline{x_2})
\end{aligned}$$

2. We now show that every **SNP** problem is SERF-reducible to $(\boldsymbol{k}\textbf{-SAT}, n)$. Take an **SNP** problem from the definition, and

- introduce d^t Boolean variables for $f_j(i_1, \ldots, i_t)$, such as variable $x_{10;\,1,8,5}$ for the expression $f_{10}(1,8,5)$.
- let k be the number of occurrences of f_j's in Φ (note that Φ is a fixed formula, so it has a constant size),
- for specific $i_1, i_2, \ldots$, we can compute R_j's (for specific arguments, they are Boolean constants!) and write Φ as k-CNF $\Phi'_{i_1,i_2\ldots}$ of $\leqslant 2^k$ clauses involving our Boolean variables,
- instead of the original
$$\forall i_1, i_2 \ldots \Phi(i_1, i_2, \ldots),$$
write
$$\bigwedge_{i_1,i_2\ldots\in[1..d]} \Phi'_{i_1,i_2,\ldots}.$$

Alternatively, write the $\boldsymbol{k}\textbf{-SAT}$ instance directly using assignment $f\colon [1\,..\,d^t] \to \{\texttt{False}, \texttt{True}\}$ and specify the clauses of Φ' as we did in showing $(\boldsymbol{k}\textbf{-SAT}, n) \in \textbf{SNP}$. $\square$

3 Sparsification

In order to relate the exponents for ***k*-SAT** and for linear-size **SAT** (in particular, to show that $s_\infty \leqslant s^{\text{dens}.\infty}$), we provide a subexponential reduction that shortens formulas (makes them *sparse*). We will provide a reduction in the other direction as well.

3.1 The sparsification procedure

Recall the Clause Shortening algorithm from the previous lecture.

$$F = G \wedge (\ell_1 \vee \ldots \vee \ell_k \vee \ldots\ldots\ldots)$$

shorten: $G \wedge (\ell_1 \vee \ldots \vee \ell_k)$ — substitute: $F[\overline{\ell}_1, \overline{\ell}_2, \ldots, \overline{\ell}_k]$

shorten — substitute — shorten — substitute

What if there are many clauses containing $(\ell_1 \vee \ldots \vee \ell_k)$? Then we get rid of all of them in the left branch simultaneously by the subsumption rule! This is exactly our goal now (instead of clause shortening): to get rid of many clauses without doing much work.

We thus cut a "weak sunflower" into pieces (see the picture below: we leave the heart H in one branch, and we leave the petals in the other branch). This terminology is motivated by the somewhat similarly looking Erdös–Rado sunflower lemma.

What if there are no or very few clauses that contain the same sub-clause? Perhaps, it means that there are only a few clauses left (this can be proved rigorously, but first think about $k = 1$: then we are talking about the number of occurrences of a literal).

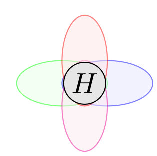
This intuition leads to the following procedure.

Algorithm 1 (The sparsification procedure).

`Input:` k-CNF formula F, small ε.

`Output:` linear-size k-CNF formulas $F_1, \ldots, F_L$ including a satisfiable formula iff F is satisfiable.

`Parameters:` integers θ_i.



1. `For` $c = 1, \ldots, n$ *`// the shorter, the better`*
   `For` $h = c, \ldots, 1$ *`// the larger, the better`*
   `If` F `contains` c`-clauses` $C_1, \ldots, C_d$ $(d \geqslant \theta_{c-h})$ `such that` $|\bigcap_{i=1}^{d} C_i| = h$
   `Then let` $H = \bigcap_{i=1}^{d} C_i$ `and proceed to step 2`
   `If nothing is found, then print` F `and exit.`
2. `Make a recursive call for` $F \setminus \{C_1, \ldots, C_d\} \cup \{H\}$.
3. `Make a recursive call for` $F[\,\overline{H}\,]$.

Every printed formula contains at most $(\theta_{c-1}-1)\cdot 2n$ c-clauses (otherwise, there would be a literal that occurs in θ_{c-1} c-clauses, and the formula would not be printed). In total, every literal occurs in at most $\sum_{c=1}^{k}(\theta_{c-1}-1)$ clauses, and there are at most $\sum_{c=1}^{k}(\theta_{c-1}-1)\cdot 2n$ clauses in the formula.

We did not say what are the parameters θ_i. In fact, in order to show that the number of printed formulas is at most $\underline{\boldsymbol{2^{\varepsilon n}}}$ one can use $\theta_i = O\left(\left(\frac{k^2}{\varepsilon}\log\frac{k}{\varepsilon}\right)^{i+1}\right)$ when $k\to\infty$. This is a technical combinatorial statement that **we do not prove** in this course.

Let $c_{k,\varepsilon} := (k/\varepsilon)^{3k}$. Given such θ_i's, the number of occurrences of any literal is at most

$$\sum_{c=1}^{k}(\theta_{c-1}-1)\leqslant\sum_{c=1}^{k}\left(\frac{k^2}{\varepsilon}\log\frac{k}{\varepsilon}\right)^{c}\leqslant c_{k,\varepsilon},$$

and the number of clauses is $O(c_{k,\varepsilon}n)$.

Now, if we believe that the number of formulas is bounded by $2^{\varepsilon n}$, then we can design a SERF reduction by querying the oracle instead of printing the formulas, and returning "yes" if for some formula the oracle answered "yes": for given t, let $\varepsilon = 1/t$, then the running time of this reduction is $\widetilde{O}(\#\text{leaves})$, and the queries have the number of clauses at most $c_{k,\varepsilon}n$.

Repeating what we said earlier about SERF reductions, we see now that given an $\widetilde{O}(2^{\boldsymbol{m}/s})$-time $\boldsymbol{k}$**-SAT** algorithm, for every t', we can solve $\boldsymbol{k}$**-SAT** in time $\widetilde{O}(2^{\boldsymbol{n}/t'})$: taking $\varepsilon = 1/t$, $s = (\varepsilon + c_{k,\varepsilon})t'$ we get the running time $\widetilde{O}(2^{\varepsilon n}\cdot 2^{c_{k,\varepsilon}n/s}) = \widetilde{O}(2^{n/t'})$.

3.2 Relating s_∞ to $s^{\mathbf{dens.}\infty}$

The sparsification procedure gives us

$$s_k \leqslant s_k^{\text{freq.}2((k/\varepsilon)^{3k})} + \varepsilon \leqslant s^{\text{freq.}2((k/\varepsilon)^{3k})} + \varepsilon.$$

Let $\varepsilon = 1/k$ and let $k\to\infty$, then we get

$$s_\infty \leqslant s^{\text{freq.}\infty} \leqslant s^{\text{dens.}\infty}$$

(the last inequality is due to the fact that the number of clauses does not exceed the number of variables multiplied by the frequency bound).

What about the opposite inequality?

Recall that the Clause Shortening **SAT** algorithm runs in time

$$\widetilde{O}(2^{c_k\boldsymbol{n}+4\boldsymbol{m}/2^{c_k k}}),$$

where c_k is such that we can solve $\boldsymbol{k}$**-SAT** in time $\widetilde{O}(2^{c_k n})$

By the definition of s_k, we can solve $\boldsymbol{k}$**-SAT** in time $\widetilde{O}(2^{(s_k+\varepsilon)n})$-time (where ε is here because we take the infimum in the definition), so $c_k = s_k + \varepsilon$.

$$s^{\text{dens}.d} \leqslant s_k + \varepsilon + \frac{4d}{2^{k(s_k+\varepsilon)}} \underset{\varepsilon\to 0}{\leqslant} s_k + \frac{4d}{2^{ks_k}}.$$

Take $k = k(d)$ growing fast enough for $\frac{4d}{2^{ks_k}} \leqslant s_\infty - s_k$, and take $d \to \infty$; then we get

$$s^{\text{dens}.\infty} \leqslant s_\infty.$$

The only problem is that we assumed here that $s_\infty - s_k > 0$ (otherwise how do we define $k(d)$?). This is the main result of Section 4: we show that $s_\infty - s_k > 0$ and even show the asymptotics for this difference.

4 Trading the size of clauses for the number of variables

In this chapter we prove that the sequence s_k strictly increases infinitely often. We can even give some bounds on how much it increases. This is shown in the proof of the following theorem.

Note that the proof of this theorem **will not be asked as a theory question** during the final exam, but there is a homework problem related to it, and the intuition acquired from studying this proof may be useful as well.

Theorem 2. $s_k \leqslant (1 - \Omega(k^{-1}))s_\infty$.

Proof (Sketch, some details are left as exercises).

1. The Unique-k-SAT case.

(a). Sparsification. We start with applying the sparsification procedure to ensure that each variable occurs at most c times, where c is a constant. (It is clear that this procedure does not add more satisfying assignments.)

(b). Counting forced variables in a chosen subset. Recall from Lectures 2 and 3 that for a d-isolated assignment and a random order of variables the expected number of variables that the IPZ algorithm gets for free by unit clause elimination (let us call them **forced** variables) is at least n/d, and that we can construct a small permutation space such that one of the permutations guarantees that.

Consider the unique satisfying assignment S. Let us do a somewhat similar thing: split the variables into two non-intersecting sets $\{x_1, \ldots, x_n\} = V \sqcup W$, where variables in V are set to their proper values under S, and we hope that many variables in W are forced by this. Select each variable for W at random with probability $1/k$. For each variable x that has a critical clause of size k, the chances that x is forced are $\frac{1}{k} \cdot (1 - 1/k)^{k-1} \geqslant 1/(k \cdot \boldsymbol{e})$ (this is the probability that x is selected for W and all other variables in its critical clause are selected for V). The expectation of the number of such variables is thus at least $n/(k \cdot \boldsymbol{e})$. We can derandomize this procedure using a small sample space similarly to how we did for the corresponding part of the PPZ algorithm, so from now on let us assume that our variables are split into V and W so that W contains at least $n/(k \cdot \boldsymbol{e})$ forced variables.

(c). Expressing the fact that x is forced using a DNF formula. Let us write the proposition that x is forced (by the assignment S) as a Boolean formula G_x. Let us enumerate all the possibilities and connect them by the disjunction sign. Every such case (possibility) is described by the term $\overline{\ell}_1 \wedge \ldots \wedge \overline{\ell}_s$, where there is an x-critical clause of the form $(\ell_1 \vee \ldots \vee \overline{\ell}_s \vee x)$ or $(\ell_1 \vee \ldots \vee \overline{\ell}_s \vee \overline{x})$ and the corresponding variables belong to V (note that $S[\ell_1] = \ldots = S[\ell_s] = 0$).

Likewise, define the proposition G_x^v saying that x is forced to the value v by taking the disjunctions of the terms corresponding only to the clauses that force x to this specific value v.

Note that if x is forced, then literally $G_x = G_x^0 \vee G_x^1$. Note also that these formulas contain at most $c(k-1)$ variables as there at at most c clauses (critical or not) containing x.

(d). Getting rid of forced variables. Our goal is to express forced variables via other variables and thus get rid of them, reducing the number of variables in the formula. The pay for it will be the increase in the size of clauses, we will get a k'-CNF out of k-CNF. We will express forced variables in W and rename the remaining variables in W.

First of all, split $W = W_1 \sqcup \ldots \sqcup W_p$ into subsets of a constant size g to be determined later. For each subset W_i, guess the number f_i of forced variables in W_i (we will enumerate all the relevant integer vectors $(f_1, \ldots, f_p)$).

Let us work with specific W_i. Let Y_i be the set of fresh new variables $y^{i_1}, y^{i_1+1}, \ldots, y^{i_1+(g-f_i)}$ where $i_1 = \sum_{i'<i}(g - f_i') + 1$. (We simply took variables $y_1, y_2, \ldots$ and distributed them to different Y_i's based on the information about f_i.) We will use these variables for renaming unforced variables of W. Note that if we rename x_j to $y_{j'}$ in the case it is unforced, then adding a CNF representation of

$$x_j \iff G_{x_j}^1 \vee (\overline{G_{x_j}^0} \wedge y_{j'})$$

to F does not change its satisfiability. However, we do not know a priori which variables are unforced and so we have to figure out j'. For this, we count the number of unforced variables z preceding x_j in W_i using the formulas G_z for them, so let

$$\beta_j := y_{i_1+q}, \text{ if exactly } q-1 \text{ of the formulas } G_z \text{ are true.}$$

This is a Boolean function on a constant number of variables, namely, $k' \leqslant ckg$, and so it can be expressed as a k'-CNF (as well as a k'-DNF) with at most $2^{k'}$ clauses (respectively, terms). Define

$$\Psi_j := G_{x_j}^1 \vee (\overline{G_{x_j}^0} \wedge \beta_j)$$

so that Ψ_j defines a formula that returns the correct value for x_j if it is forced, and returns the variable $y_{j'}$ otherwise.

We are ready to replace x_j by Ψ_j, so let $F_{\vec{f}}$ be F after we substitute all variables in W this way. What is the maximum clause size after this substitution? (We need to rewrite this formula as a CNF.) At most $k \cdot k'$. Why the index $\vec{f}$? Because we do not know what are the f_i's, and these numbers are used in the definition of Ψ_j (what is i_1 otherwise?!). Thus we consider all integer vectors $\vec{f} = (f_1, \ldots, f_p)$ such that $\sum_{i=1}^{p} f_i \geqslant n/(k \cdot \boldsymbol{e})$ as we know that the number of forced variables is at least $n/(k \cdot \boldsymbol{e})$. We thus ask the oracle about all these formulas $\boldsymbol{F}_{\vec{f}}$. How many of them are there, that is, how many vectors $\vec{f}$? It is easy to count: at most $(g+1)^{n/g}$. And this is where we choose g: if we choose it large enough (but still a constant!) so that $\log_2(g+1) \leqslant \varepsilon g$, then $(g+1)^{n/g} \leqslant 2^{\varepsilon n}$. All these formulas are k''-CNF for a constant k'', and they are larger than F at most by a constant factor.

And, yes, it contains only $n(1-1/(k\cdot e))$ variables, so the running time of a Unique-k''-SAT algorithm (designed for our specific ε) on it will be $\widetilde{O}(2^{(\sigma_{k''}+\varepsilon)(1-1/(k\cdot \boldsymbol{e}))n})$; given that we have only $2^{\varepsilon n}$ formulas to ask it about, we get $\sigma_k \leqslant \sigma_{k''}(1-1/(k\cdot \boldsymbol{e})) \leqslant \sigma_\infty(1-1/(k\cdot \boldsymbol{e}))$.

2. The general *k*-SAT case.

Recall the corresponding trick in PPZ. Let $\delta > 0$ (a constant) be small enough, so that the number of assignments of weight at most δn will be small enough (see below).

After checking these assignments, we can assume that all satisfying assignments are of weight at least δn, and so one of them (the "lightest" one, call it S_*) is δn-isolated.

Proceed as in the uniquely satisfiable case (but now we keep in mind the assignment S_* when defining what is a "forced" variable). The number of forced variables is now $\delta n/(k\cdot \boldsymbol{e})$), so k'' may change but will still be a constant.

The first phase of this algorithm takes time $\widetilde{O}(2^{(s_{k''}/2)\cdot n})$, and the second phase produces at most $2^{\varepsilon n}$ polynomial-size formulas in k''-CNF and their satisfiability will be checked using a k''-SAT algorithm in the total time $\widetilde{O}(2^{(s_{k''}+\varepsilon)(1-\delta/(k\cdot \boldsymbol{e}))n+\varepsilon n})$. Thus $s_k \leqslant s_{k''}(1-\delta/(k\cdot \boldsymbol{e}))$.

It remains to compute δ using the volume of a Hamming ball of radius δn and the requirement that the gain $s_{k''}\cdot\delta/(k\cdot \boldsymbol{e})$ dominates the extra $H(\delta)$ (ideally, by $\Omega(\frac{1}{k})$):

$$s_{k''}\cdot\delta/(k\cdot \boldsymbol{e}) > H(\delta).$$

Then δ can be computed from $s_{k''}$. We estimated a similar quantity in Problem HW1.4. An accurate calculation showing that $s_k \leqslant (1-\Omega(\frac{1}{k}))s_\infty$ (as claimed) is left as an exercise. □

5 Isolation

One other setting is Unique ***k*-SAT**. We defined the constants σ_k and their limit σ_∞ for this problem. Again, it looks like this is an easier problem, but yet the limit is the same as for the general case of ***k*-SAT**.

Lemma 2 (Isolation Lemma)**.** $\forall k\ \forall\varepsilon \in (0,\frac{1}{4})$ $s_k \leqslant \sigma_{k'} + O(H(\varepsilon))$*, where* $k' = \max\{k, \frac{1}{\varepsilon}\ln\frac{2}{\varepsilon}\}$.

Corollary 1. *For* $\varepsilon = \frac{2\ln k}{k}$ *the lemma gives* $s_k \leqslant \sigma_k + O(\frac{\ln^2 k}{k})$*. Thus* $s_\infty = \sigma_\infty$.

We only give some ideas about the reduction that is used for proving Lemma 2 (of course, **not for the exam**). It consists of two stages.

ISOLATION PROCEDURE FOR k-CNF

Phase 1. Concentration.

Concentrate the satisfying assignments in a ball of radius εn.

How?

Let $k' = \max\{k, \frac{1}{\varepsilon} \ln \frac{2}{\varepsilon}\}$ and $N = O(n)$.
Add N length-k' random xors (hyperplanes) $\bigoplus_{i \in R} a_{R,i} x_i = b_R$,
where R is a random subset of variables of size k'; take $a_{R,i}, b_R \in \{0, 1\}$ at random.

Phase 2. Isolation within a ball.

1. Guess random $S \subseteq [1\,..\,n]$: the set of $2\varepsilon n$ variables that still have different values in different satisfying assignments.
2. Guess an assignment for variables in S to make the satisfying assignment unique.

The proof that this reduction has a good probability to output a uniquely satisfiable formula, is non-trivial, and we do not give it in this course. An interested reader can find the details in [CIKP03].

6 Takeaway

We learned about ETH and SETH, which are common conjectures found in the literature.

We learned that many exponents go to the same limit, as the size of clauses or the density goes to the infinity:

$$s_\infty = \sigma_\infty = s^{\text{freq.}\infty} = s^{\text{dens.}\infty}.$$

(And SETH states that this limit is 1, that is, the complexity is of the order 2^n.)

We learned how to reduce ***k*-SAT** to the case of linear-size formulas.

Historical notes and further reading

This lecture is based on the chapter "Worst-Case Upper Bounds" from *Handbook of Satisfiability* and on the four articles listed in the references.

References

[IPZ01] Russell Impagliazzo, Ramamohan Paturi, Francis Zane:
Which Problems Have Strongly Exponential Complexity?,
Journal of Computer and System Sciences 63, 512–530 (2001)

[IP01] Russell Impagliazzo, Ramamohan Paturi:
On the Complexity of ***k-SAT***,
Journal of Computer and System Sciences 62, 367-375 (2001)

[CIKP03] Chris Calabro, Russell Impagliazzo, Valentine Kabanets, Ramamohan Paturi:
*The Complexity of Unique-**k-SAT**: An Isolation Lemma for k-CNFs*,
Proceedings of CCC-2003

[CIP06] Chris Calabro, Russell Impagliazzo, Ramamohan Paturi:
A duality between clause width and clause density for ***SAT***,
Proceedings of CCC-2006