CSE 599I Course Notes: Spring 2024

ETH, SETH and Fine-Grained Complexity, Lifting

Paul Beame

May 13, 2024

Overview

Exponential time hypotheses and fine-grain complexity: Beyond P vs NP Despite the best efforts of researchers for over 50 years since the P vs NP question was formulated, the best algorithms we have for SAT and other NP-complete problems are still exponential in the worst case and barely improve on brute force. If these are representative of the correct level of complexity, which seems a reasonable stronger conjecture than P≠NP, our usual ways of proving relationships between NP problems need to be rethought, both from a theoretical and practical point of view, and radically new relationships between problems emerge, a subject termed "fine-grain complexity". This yields surprising connections that have produced a web of problem-solving relationships well beyond the usual resource-focused complexity classes, for example, showing how improving existing polynomial-time algorithms for well-known problems is tied to improving exponential-time algorithms for SAT.

In this part of the course we will first discuss algorithms with different approaches to SAT and the best analysis with respect to their worst-case behavior. We will then consider the Exponential-Time Hypothesis (ETH) for SAT originally formulated by Impagliazzo and Paturi, analyze its robustness and implications for other NP problems.

After that we will then focus on a much more powerful Strong Exponential-Time Hypothesis (SETH) which, though it not as robust as ETH, has nearly the same level of evidence, and has wide-ranging implications that would allows to pinpoint the the complexity of many problems in P with a high degree of accuracy, proving that current algorithms are not far from the best possible. This is based on so-called fine-grained reductions and analysis of the complexity of these problems. (In general, fine-grained complexity encompasses both the consequences of SETH, but also of a small number of other key hypotheses, but in this course we will only focus on those that relate to SETH, or other closely related hypotheses.)

Depending on time we may also briefly discuss how even much smaller algorithmic improvements over brute force in solving satisfiability for restricted classes of circuits yield circuit lower bounds – an "ironic" consequence that algorithms imply lower bounds.

Lifting A number of longstanding problems in computational complexity have been resolved in the last decade by showing how simple forms of function composition let us convert hardness results proven in weak models of computation into hardness results for more powerful models of computation, a methodology that has been termed "lifting". We discuss lifting techniques and the some of the longstanding problems resolved using them. We then focus on a number of open problems and approaches to resolving them.

The main idea of lifting is conceptually simple: Given a Boolean function f or search problem (relation) R defined on $\{0,1\}^n$, one can take a simple Boolean function g (a "gadget") and define the composition $f \circ g^n$ or $R \circ g^n$, by $f(z_1, \ldots, z_n)$ where $z_i = g(x_i)$ or $z_i = g(x_i, y_i)$ depending on context where x_i and y_i may involve multiple bits.

The resulting function $f \circ g^n$ is called a "lift" of f. It can easily be computed in a composed manner also: Given a computation of f, whenever f needs to access z_i, the computation for the lift of f can access the x_i (possibly also y_i) and compute g instead to produce the value for z_i. The general idea is to translate the complexity of computing f to its lifted version but in a stronger computational model for example converting the complexity of computing f using queries (decision trees) to the complexity of the lift of f in a communication complexity model. The simple algorithm of computing a z_i each time it is needed yields a communication protocol that has communication complexity equal to the product of the query complexity of f and the communication complexity of g.

A key question is whether this can be improved. Lifting results have shown that under certain general conditions it cannot be improved. Applications of this to circuit complexity and proof complexity follow from the fact that lifting applies to search problems as well as to Boolean functions.

The worst-case complexity of SAT

We can ask this question for many different input formats, each a special case of CIRCUIT-SAT where the input is a circuit C defined on Boolean variables in $\{0,1\}^n$.

The obvious brute force algorithm for CIRCUIT-SAT has complexity $|C| \cdot 2^n$.

All of the special cases of CIRCUIT-SAT we will discuss are NP-complete. We first focus on k-SAT for $k \geq 3$. How much can we improve on this brute force algorithm?

1 Best current SAT algorithms

We review the ideas behind the best current algorithms.

The PPZ Algorithm

The first is an extremely simple randomized algorithm, the PPZ algorithm of Paturi, Pudlak, and Zane.

Algorithm 1 The PPZ-algorithm.
Input k-CNF formula F in n Boolean variables $x_1, \ldots, x_n$.

```
1:  function PPZ(F)
2:      repeat n^2 2^{n-n/k} times:
3:          α ← ∅
4:          F' ← F                                   ▷ The current partial assignment
5:          while F' ≠ ⊥ and F' has an unassigned variable do   ▷ F' is not identically false
6:              Choose an unassigned variable x_i uniformly at random
7:              if F' contains the unit clause x_i then
8:                  b ← 1                            ▷ x_i's value is forced to 1.
9:              else if F' contains the unit clause ¬x_i then
10:                 b ← 0                            ▷ x_i's value is forced to 0.
11:             else
12:                 Choose b ∈ {0,1} uniformly at random
13:             Add x_i = b to α
14:             F' ← F'|_{x_i ← b}                   ▷ Set x_i to b in F' and simplify
15:         if F' = ⊤ then Halt and output satisfying assignment α   ▷ F is satisfied
16:     until
```

Theorem 1.1. *If F is a satisfiable k-CNF formula, the PPZ algorithm finds a satisfying assignment for F with probability $1 - o(1)$.*

Before we prove this, we need a few definitions.

Definition 1.2. For $x \in \{0,1\}^n$, write $x^{\oplus i}$ for the element with the i-bit flipped. For a n-bit Boolean function f, $i \in [n]$, and $x \in \{0,1\}^n$, bit i is *sensitive for f at x* iff $f(x^{\oplus i}) \neq f(x)$. The *sensitivity of f at x* is the number of bits i that are sensitive for f at x, denoted $s_f(x)$. We extend this to circuits and formulas by applying the definition to the associated Boolean function.

Proposition 1.3. *Suppose that x is a satisfying assignment for CNF formula F. Then there are $s_F(x)$ distinct* critical *clauses of F, one clause $C_{x,i}$ for each sensitive bit i of F at x, such that $C_{x,i}(x) = 1$ but $C_{x,i}(x^{\oplus i}) = 0$. In particular, $C_{x,i}$ contains x_i or $\overline{x_i}$.*

Proof. The existence of critical clauses is immediate from the definition. The fact that the clauses must be distinct follows from the fact a clause containing both variables x_i and $x_{i'}$ for i and i' sensitive on x cannot be critical for either variable since the clause would have two true literals on input x. □

Proposition 1.4. *Let S be the set of satisfying assignments of F. Then $\sum_{x\in S} 2^{s_F(x)} \geq 2^n$.*

Proof. By induction on n. For $n = 0$ there is exactly one string in $\{0,1\}^n$ which has sensitivity 0 so the statement holds. Suppose it is true for $n-1$ and consider the subsets $S_b = \{x' \in \{0,1\} \mid x'b \in S\}$ for $b = 0, 1$. Then S_b is the set of satisfying assignments of $F_b = F|_{x_n=b}$. If S_0 is empty then $S = \{x'1 \mid x' \in S_1\}$. Then by definition $s_F(x'1) = s_{F_1}(x') + 1$ and

$$\sum_{x\in S} 2^{S_F(x)} = \sum_{x'\in S_1} 2^{s_F(x'1)} = \sum_{x\in S_1} 2^{s_{F_1}(x')+1} \geq 2 \cdot 2^{n-1}$$

by the induction hypothesis for F_1. The same bound holds if S_1 is empty. Finally, if both S_0 and S_1 are non-empty, then

$$\sum_{x\in S} 2^{S_F(x)} = \sum_{x'\in S_0} 2^{s_F(x'0)} + \sum_{x'\in S_1} 2^{s_F(x'1)} \geq \sum_{x'\in S_0} 2^{s_{F_0}(x')} + \sum_{x'\in S_1} 2^{s_F(x')} \geq 2^{n-1} + 2^{n-1} = 2^n$$

by the inductive hypothesis applied to F_0 and F_1. □

Proof of Theorem 1.1. Let S be the set of satisfying assignments for F. Fix some $x^* \in S$ and let $j = s_F(x^*)$. There are j critical clauses on input x^*. Each execution through the **repeat** loop of the PPZ algorithm induces a uniformly random permutation π on the variables.

Define E_1 be the event that for at least j/k of the j critical clauses $C_{x^*,i}$ on input x^*, the critical variable x_i occurs last among the variables in the clause under the permutation π. Let ℓ be the random variable for this number. For each critical clause, the critical variable occurs last with probability at least $1/k$ since each clause has size at most k. Therefore $\mathbb{E}[\ell] \geq j/k$. We claim that with probability that E_1 holds is at least $1/(kj - j + 1) \geq 1/(kn)$: Since j, k, and ℓ are integers, a value of ℓ below j/k must be most $(j-1)/k$. Since ℓ is an integer bounded by j, the probability that $\ell < j/k$ is at most the probability that applying Markov's inequality to random variable $j - \ell$, whose expected value is $\leq j - j/k = (k-1)j/k$, is at least $j-(j-1)/k = ((k-1)j+1)/k$ which is $((k-1)j+1)/k$. This means that if ℓ is below its expected value, then $j - \ell$ is at least $((k-1)j+1)/((k-1)j)$ times its expected value which, by Markov's inequality, occurs with probability at most $1 - 1/((k-1)j+1)$.

Now assume that E_1 holds and consider the probability that the assignment chosen in the **while** loop agrees with x^*. At each iteration, if the value of x_i is forced then it certainly agrees with x^*; otherwise it agrees with with x^* with probability $1/2$. Since E_1 holds, there are at most $n - j/k$ assignments to agree with x^* that are not forced since the $\geq j/k$ variables in the critical clauses witnessing event E_1 will all be forced. Therefore this occurs with probability at least $2^{j/k-n} = 2^{s_F(x^*)/k-n}$ and hence the probability that a single iteration of the **repeat** loop finds satisfying assignment x^* is at least $\frac{1}{kn} 2^{s_F(x^*)/k-n}$. Putting

it all together we have

$$\begin{aligned}
&\Pr[\text{a } \textbf{repeat} \text{ iteration outputs a satisfying assignment}] \\
&= \sum_{x \in S} \Pr[\text{a } \textbf{repeat} \text{ iteration outputs satisfying assignment } x] \\
&\geq \frac{1}{kn} \sum_{x \in S} 2^{s_F(x)/k - n} \\
&= \frac{1}{kn} 2^{n/k-n} \sum_{x \in S} 2^{(s_F(x)-n)/k} \\
&\geq \frac{1}{kn} 2^{n/k-n} \sum_{x \in S} 2^{s_F(x)-n} \\
&\geq \frac{1}{kn} 2^{n/k-n} \qquad \text{by Proposition 1.4.}
\end{aligned}$$

Since PPZ runs this $n^2 2^{n(1-1/k)}$ times it succeeds with probability $1 - o(1)$. □

In particular when $k = 3$, PPZ has a running time of $2^{2n/3+o(n)}$ which is at most 1.5875^n for large n.

PPZ also show how their randomized algorithm can be made deterministic with essentially the same complexity but the details make the algorithm substantially messier.

The PPSZ algorithm

This algorithm was improved by Paturi, Pudlak, Saks, and Zane to yield the PPSZ algorithm, which replaces the unit clause test for forced variables with a slightly different criterion tests than unit clauses. It checks whether x_i or $\overline{x_i}$ can be derived by bounded *resolution* on constant-size clauses. The resolution rule is given by

$$\frac{A \vee x,\ B \vee \overline{x}}{A \vee B}$$

and lets one add the new clause $A \vee B$ if both clauses $A \vee x$ and $B \vee \overline{x}$ are known. This is a sound rule since variable x connect make both of the original clauses true. In general, this can increase the sizes of the derived clauses and have exponential size, but if there is a constant size limit on the size of the derived clauses (e.g. a limit of 5 variables when $k = 3$) then all derived clauses can be found in polynomial time.

The success probability of each iteration for PPSZ is $2^{cn/k-n-o(n)}$ for $c = \pi^2/6$ and hence the savings in the exponent is roughly cn/k. The original PPSZ analysis only applied when the sensitivity of every satisfying assignment was n (the satisfying assignments are all isolated). This was extended by Hertli to the general case and subsequent papers have improved the analysis only very slightly. For the special case that $k = 3$, the current best general bound due to Scheder is 1.306973^n. Again this algorithm can be determinized without significant loss.

Schöning's Random Walk Algorithm

The basic idea of this algorithm is repeated local search starting from a random initial assignment.

Theorem 1.5. *Suppose that F is a satisfiable k-CNF formula with each clause of size exactly k. Then Schöning's algorithm finds a satisfying assignment for F with probability $1 - o(1)$.*

Algorithm 2 Schöning's algorithm.
Input k-CNF formula F in n Boolean variables $x_1, \ldots, x_n$.

```
1:  function SCHOENING(F)
2:      repeat 30 · 2^n (1 − 1/k)^n times:
3:          Choose an assignment α ∈ {0,1}^n uniformly at random
4:          repeat 3n times
5:              if F is satisfied by α then
6:                  Halt and return satisfying assignment α.
7:              else
8:                  Let C be a clause of F such that C(α) = 0
9:                  Choose a uniformly random variable x_i in C.
10:                 α ← α^{⊕i}                ▷ Flip assignment α_i to make C true.
11:         until
12:     until
```

Proof. Suppose that F has some satisfying assignment α^*. For each clause C of F, we identify one special variable x_C whose associated literal in C is set to true by assignment α^*. Each time Step 9 is executed, since clause C has exactly k variables, the special variable x_C in C is chosen with probability exactly $1/k$. Fix an iteration of the outer **repeat** loop. For $t = 0, \ldots, 3n$ let $X_t \in \{0, 1, \ldots, n\}$ be the random variable counting the Hamming distance between the current assignment α and α^*. If X_t is not satisfying then $X_{t+1} = X_t - 1$ with probability at least $1/k$ corresponding to the case that the special variable is chosen. If some $X_t = 0$ then the algorithm halts and succeeds. (It may also succeed if it finds some other satisfying assignment.) Since the initial α is chosen uniformly at random:

$$\Pr[X_0 = j] = \binom{n}{j} 2^{-n}.$$

Define a Markov chain Y_t such that $Y_0 = X_0$ and for $0 < j$ satisfies

$$\Pr[Y_{t+1} = j - 1 \mid Y_t = j] = \frac{1}{k} \qquad \text{and} \qquad \Pr[Y_{t+1} = j + 1 \mid Y_t = j] = 1 - \frac{1}{k}, \tag{1}$$

and if $Y_t = 0$ then $Y_{t+1} = 0$. Clearly $X_t \le Y_t$. so

$$\Pr[\exists t \in [0, 3n],\ X_t = 0] \ge \Pr[\exists t \in [0, 3n],\ Y_t = 0].$$

If $k = 2$ then the walk Y_t is an unbiased random walk and will reach 0 with high probability in $O(n^2)$ steps. For $k \ge 3$, the walk is biased away from 0 and will be far away from 0 if we let it run too long, but there is some probability of reaching 0 in the early stages. If $Y_0 = j$ then every $i \ge 0$, if $t = 2i + j$ the probability that $Y_t = 0$ and $Y_{t'} > 0$ for $t' < t$ is equal to the probability that the Markov chain has i increasing steps and $i + j$ decreasing steps. For each fixed pattern of increasing and decreasing steps that does not have prefix that reaches 0, this occurs with probability

$$\left(\frac{1}{k}\right)^{i+j} \cdot \left(1 - \frac{1}{k}\right)^{i}.$$

If we didn't have the condition that all proper prefixes have value > 0, there would be $\binom{2i+j}{i}$ choices for the pattern of increasing/decreasing steps.

A standard theorem called the Ballot Theorem gives the following:

Claim 1.6. *The number of* good *sequences that have i increasing steps, $i+j$ decreasing steps and no proper prefix with an excess of j decreasing steps is*

$$\binom{2i+j}{i}\frac{j}{2i+j}.$$

Proof of Claim. The general idea to prove the claim is to show that for any particular vector $v \in \binom{2i+j}{i}$ exactly j of the cyclic shifts of v are good. We prove this by induction on i. For $i = 0$, all j of the shifts of v are the same (and are good). We write $+1$ for each of the i increasing steps and -1 for each of the $i+j$ decreasing steps. For $i > 0$, there must be some consecutive steps in cyclic order on v consisting of an increasing step followed by a decreasing step. A good cyclic shift of v cannot end on either of these two steps since for the first it would be less than $-j$ one step before the end and for the second would be equal to $-j$ two steps before the end. If we let v' be the string with these two steps removed, then the end points of the good cyclic shifts of v' are precisely those that are good for v. By induction exactly j of the $2(i-1)+j$ shifts of v' are good and hence j shifts of v are good. □

Therefore

$$\begin{aligned}
&\Pr[\exists t \in [0,3n],\ X_t = 0] \\
&\geq \Pr[\exists t \in [0,3n],\ Y_t = 0] \\
&= \sum_{j=0}^{n} \Pr[X_0 = j] \cdot \sum_{t=2i+j\leq 3n} \binom{2i+j}{i}\frac{j}{2i+j}\left(\frac{1}{k}\right)^{i+j}\cdot\left(1-\frac{1}{k}\right)^{i} \\
&\geq \sum_{j=0}^{n} \Pr[X_0 = j] \cdot \sum_{i=0}^{j} \binom{2i+j}{i}\frac{j}{2i+j}\left(\frac{1}{k}\right)^{i+j}\cdot\left(1-\frac{1}{k}\right)^{i} \\
&= \sum_{j=0}^{n} \Pr[X_0 = j] \cdot \sum_{i=0}^{j} \binom{2i+j}{i}\frac{j}{2i+j}\left(\frac{1}{k}\right)^{i+j}\cdot\left(1-\frac{1}{k}\right)^{i} \\
&\geq \frac{1}{3}\sum_{j=0}^{n} \Pr[X_0 = j] \cdot \sum_{i=0}^{j} \binom{2i+j}{i}\left(\frac{1}{k}\right)^{i+j}\cdot\left(1-\frac{1}{k}\right)^{i}.
\end{aligned}$$

For $k = 3$ it turns out that the dominant term in

$$\sum_{i=0}^{j} \binom{2i+j}{i}\left(\frac{1}{k}\right)^{i+j}\cdot\left(1-\frac{1}{k}\right)^{i}$$

occurs when $i = j$. That term is $\binom{3j}{j}(1/3)^{2j}(2/3)^{j}$. Using Stirling's formula $\binom{3j}{j}$ is asymptotically $\frac{2}{\sqrt{3\pi j}}3^{3j}/2^{2j}$ so the term is asymptotically equal to $\frac{2}{\sqrt{3\pi j}}2^{-j} \geq \frac{1}{\sqrt{5n}}2^{-j}$. Plugging this in we get that the probability of success is asymptotically at least

$$\frac{1}{3\sqrt{5n}}\sum_{j=0}^{n}\Pr[X_0 = j]\cdot 2^{-j} = \frac{2^{-n}}{3\sqrt{5n}}\sum_{j=0}^{n}\binom{n}{j}2^{-j} = \frac{2^{-n}}{3\sqrt{5n}}(3/2)^{n}.$$

which is roughly $(3/4)^n$ so the running time is roughly $(4/3)^n$. The savings over brute force search is roughly a $(2/3)^n$ factor.

More generally, the dominant term occurs when i is roughly $j/(k-2)$ so that $i+j$ is roughly $j(k-1)/(k-2)$ and i is roughly $j/(k-2)$ so the ratio of the two roughly matches the ratio of the corresponding probabilities. For this dominant term, the Stirling approximation gives that $\binom{2i+j}{i}$ is asymptotically at least

$$\frac{1}{\Theta(\sqrt{n})} \cdot k^{2i+j}/(k-1)^{i+j}$$

and the corresponding term is $\frac{1}{\Theta(\sqrt{n})} \cdot (k-1)^{-j}$. Plugging this in we get that the probability of success is asymptotically at least

$$\frac{2^{-n}}{\Theta(\sqrt{n})} \sum_{j=0}^{n} \binom{n}{j} (k-1)^{-j} = \frac{2^{-n}}{\Theta(\sqrt{n})} (k/(k-1))^n.$$

With a bit more care, one can observe that the $\Theta(1/\sqrt{j})$ factor (which became $1/\sqrt{n}$) can be removed because there are roughly $\Theta(\sqrt{j})$ terms of roughly equal size. The savings over brute force is roughly $(1-1/k)^n$. □

Observe that as k grows $1-1/k$ is roughly $e^{-1/k} = 2^{-1/(k \ln 2)}$. Therefore the savings over brute force for Shöning's algorithm is a $2^{-n/(k \ln 2)} \geq 2^{-1.4427n/k}$ factor. This is better than PPZ but not as good as PPSZ since $\pi^2/6$ is roughly 1.6645.

Better k-SAT algorithms?

In the three algorithms discussed so far, there is a running time savings of $2^{\Theta(n/k)}$ versus brute force search. There is a later algorithm of Chan and Williams based on a completely different approach, that de-randomizes probabilistic polynomials; it also gets savings of this character, though with a substantially worse constant in the exponent. This algorithm not only determines satisfiability, it also computes the exact number of satisfying assignments.

There is no general algorithm known that does better, though Viya and Williams have shown that for random k-CNF formulas, the basic PPZ algorithm almost surely has savings at least $2^{\Omega(\log k)n/k}$. It is not clear what the hardest instances would be for the PPZ algorithm.

2 The Exponential Time Hypothesis

The *Exponential Time Hypothesis (ETH)* of Impagliazzo and Paturi is simply the hypothesis that the worst-case complexity of 3-SAT on formulas in n variables is at least $2^{\delta n}$ for some constant $\delta > 0$.

The ETH was motivated by the fact that the best algorithms for 3-SAT known at the time all had running times at least c^n for some constant $c > 1$. In the roughly 25 years since it was formulated, that state of affairs has not changed. Impagliazzo and Paturi considered it a hypothesis rather a conjecture, in that they simply found it plausible; they were most concerned about what its consequences would be if it were true.

What does the ETH imply for NP-complete problems? Consider the implication for the problem INDEPENDENT-SET. Consider the standard reduction 3-SAT$\leq_P$INDEPENDENT-SET which maps a 3-CNF formula to a graph with $3m$ vertices, one per literal occurrence, with edges joining every pair of occurrences of the same variable with opposite sign.

Suppose that we tried to claim that ETH implies that INDEPENDENT-SET requires $2^{\delta' n}$ time for some $\delta' > 0$. We would need to show that it is impossible for INDEPENDENT-SET to be solved in time $2^{\delta N}$ for N-node graphs for some $\delta < \delta'$ given the ETH. However, the graph size in the reduction has $N = 3m$ where m could be as large as $\Theta(n^3)$. A solution in time $2^{\delta N}$ would then be in time $2^{3\delta m}$ which would be much larger than the trivial SAT algorithm which runs in time $2^n poly(m)$ and hence would not imply anything new.

However ETH *would* imply that INDEPENDENT-SET on N-node graphs requires time $2^{\Omega(N^{1/3})}$ using the same reduction.

Sparsification

To get around this problem, Impagliazzo, Paturi, and Zane devised a way to *sparsify* formulas to show that ETH is equivalent to ETH for *sparse* 3-SAT formulas with n variables and only $O(n)$ clauses.

In general, one cannot reduce a 3-CNF formula F with n variables and m clauses to one with only $O(n)$ clauses, so one needs to reduce the formula F to multiple other formulas. The total work on these formulas must also not be not too large. Impagliazzo, Paturi, and Zane showed that this approach does work and Calabro, Impagliazzo, Paturi significantly improved the analysis (full details in Calabro's dissertation) yielding the following theorem:

Theorem 2.1 (Sparsification Lemma)**.** *Let $\varepsilon > 0$ and $k \geq 3$ be a constant. There is a $2^{\varepsilon n} poly(n)$ time algorithm that takes a k-CNF formula F on n variables and produces $2^{\varepsilon n}$ k-CNF formulas, $F_1, \ldots, F_{2^{\varepsilon n}}$, such that F is satisfied if and only if $\bigvee_i F_i$ is satisfied and each F_i has n variables and $\left(k/\varepsilon\right)^{O(k)} n$ clauses. In fact, each variable is in at most $poly(1/\varepsilon)$ clauses, and the F_i are over the same variables as F.*

Corollary 2.2. *The ETH implies that there exist some constants $c, \delta > 0$ such that satisfiability of 3-CNF formulas in n variables and at most cn clauses (sparse formulas) requires time at least $2^{\delta n}$.*

Proof. Each of the formulas F_i in the Sparsification Lemma has $O(n)$ clauses in n variables and hence is sparse. Suppose that the conclusion is false. If 3-SAT is easy for sparse formulas, say with running time at most $2^{\varepsilon' n}$ for every $\varepsilon' > 0$, then one could solve 3-SAT in time $2^{\varepsilon'' n'}$ time for any $\varepsilon'' > 0$ and arbitrary clause density as follows:

Applying the Sparsification Lemma with $\varepsilon = \varepsilon' = \varepsilon''/3$, computing all the formulas F_i. Check the satisfiability of each one in turn in time $2^{\varepsilon' n}$ and output YES iff all of the F_i are satisfied. The total

runtime would be $poly(n) \cdot 2^{2\varepsilon'} = poly(n)2^{2\varepsilon''/3} < 2^{\varepsilon''}$. This would violate the ETH, so it must be the case that ETH is true even for sparse formulas. □

Here is a sampling of the consequences/equivalent formulations of the ETH given the Sparsification Lemma.

Corollary 2.3. *The ETH is equivalent to the following:*

- *For every $k \geq 3$ there is a constant $s_k > 0$ such that k-SAT on n variable formulas requires time at least $2^{s_k n - o(n)}$.*
- *For each of the following NP-complete problems there is a constant $c > 0$ such that the problem requires time at least 2^{cN}:*
  - *INDEPENDENT-SET, VERTEX-COVER and DOMINATING-SET on graphs with N edges,*
  - *CLIQUE on N-node graphs,*
  - *3-COLOR on graphs with N edges,*
  - *k-COLOR on graphs with N edges,*
  - *SUBSET-SUM with N integers of at most N bits each,*
  - *HAMILTONIAN-PATH, HAMILTONIAN-CYCLE on graphs with N edges.*

We now focus on the proof of the Sparsification Lemma. The basic idea is a kind of tree search where we branch on the value of variables (or subclauses), simplifying the formula on each branch as we go. This will be effective in reducing the number and lengths of clauses while the simplified formulas are not sparse, yielding a frontier where we stop where the formulas are all sparse. Each final F_i is the conjunction of the path to the simplified formula at this frontier. It remains to argue that this frontier is small and that all the formulas at the frontier are sparsified.

At each step the algorithm will introduce new short clauses that may subsume (imply) other longer clauses that contain them. Whenever we do so we will reduce the formula by deleting all subsumed clauses. We call this operation $reduce(\cdot)$. We will view each CNF formula as a set of clauses using an implicit conjunction.

The key idea of the branching algorithm is that formulas with many clauses must have large sets of clauses that share a non-trivial subclause on which they overlap and that this overlap can be pulled out using the distributive law: either the common subclause is satisfied, which makes all of the clauses true, or the remaining part of each clause must be satisfied.

This is related to but different from the kinds of intersections in the sunflower lemma, where the core (common intersection) may be empty and the remaining sets (petals) must be disjoint.

Here we identify each clause with a set of literals and have the notion simply of an (h,p)-*flower*, which is a collection $\geq \theta_p^{\varepsilon,k}$ clauses of size exactly $(h+p)$ that all share a common subclause H, the *heart* of the flower, of size $h \geq 1$. We write P, the set of *petals* of the flower, for the set of p-clauses that remain from the flower when H is removed. The values $\theta_p^{\varepsilon,k}$ are parameters that we set later. We identify an (h,p)-flower by its heart H.

At each stage, there may be many different flowers that can be branched on. The algorithm always chooses to branch using flowers with the smallest current clause size that have that smallest petals (largest heart) so that the potential subsumptions are maximized.

Algorithm 3 Sparsification algorithm.
Input a k-CNF formula F in n Boolean variables $x_1, \ldots, x_n$.
Assumes a set of parameters $\theta_p^{\varepsilon,k}$ for $p = 1, \ldots, k$ that give the threshold for the number of petals of size p at which subformulas F^* are designated as (h,p)-flowers.

```
1: function SPARSIFY(F')
2:     F' ← reduce(F')                          ▷ Remove subsumed clauses from F'
3:     if there is some (h,p)-flower F* in F' then
4:         Choose an (h,p)-flower F* such that h+p is minimized and then h is maximized.
5:         Let H be the heart of F* and P be the set of petals of F*.
6:         Sparsify(F' ∪ {H})                   ▷ First branch is the case that H is set to true.
7:         Sparsify(F' ∪ P)                     ▷ Second branch is the case that all petals in P are set to true.
8:     else
9:         Append formula F' to the list of output formulas.
```

The values of the parameters $\theta_p^{\varepsilon,k}$ algorithms depend on k and ε solely via a parameter $\alpha = \alpha_{\epsilon,k} \geq 2$ that we will define later. For simplicity of notation, we will drop the k and ε and just write θ_p instead of $\theta_p^{\varepsilon,k}$. These are defined in terms of auxiliary parameters $\beta_{k'}$ and are defined by

$$\begin{aligned}
&\beta_1 = 2 \\
&\beta_{k'} = \sum_{h=1}^{k'-1} 4\alpha\beta_{k'-h}\beta_h \qquad \text{for } 2 \leq k' < k \\
&\theta_0 = 2 \\
&\theta_{k'} = \alpha\beta_{k'} \qquad \text{for } 1 \leq k' < k.
\end{aligned}$$

Note that since $\theta_0 = 2 > 1$, an (h,p)-flower must have $p > 0$.

For $k' \leq k$, we say that a formula F' is *k'-sparsified* iff for every $j \leq k'$ and every h with $0 < h < j$, F' does not contain an $(h, j-h)$-flower.

The execution of the sparsification algorithm produces a binary tree with each node u labelled by a formula $F_u = reduce(F')$ where F' is the formula at which the node is called. By definition, u is a leaf of this tree if and only if F_u is k-sparsified. We write H_u for the heart found during the call at node u and P_u for the set of petals at node u. If u is not a leaf and v, w are the left and right children of u, respectively, then $F_v = reduce(F_u \cup \{H_u\})$ and $F_w = reduce(F_u \cup P_u)$.

The following proposition is immediate from the fact that the $F' \equiv reduce(F')$ and that any truth assignment that satisfies a flower satisfies its heart or satisfies all of its petals.

Proposition 2.4. *$F_u \equiv F_v \vee F_w$.*

Lemma 2.5. *Suppose that F_u is k'-sparsified.*

(a) If C' is an arbitrary clause, with $|C'| = h < j \leq k'$, there are fewer than θ_{j-h} j-clauses $C \in F_u$ containing C'.

(b) For $j \leq k'$, F_u contains fewer than $2\theta_{j-1}n/j$ clauses of length j.

Proof. For (a), assume that there are at least θ_{j-h} j-clauses in F_u that contain C'. Let H be their common intersection. Then $C' \subseteq H$ so $|H| \geq h$. Since $\theta_{j-|H|} \leq \theta_{j-h}$, this would contradict the assumption that F_u is k'-sparsified.

For (b), suppose that F_u contains at least $\frac{2n}{j}\theta_{j-1}$ j-clauses. Therefore there are at least $2n\theta_{j-1}$ total literals occurring in these j-clauses. It follows that at least one literal occurs in at least θ_{j-1} clauses which would be a $(1, j-1)$ flower contradicting the assumption that F_u is k'-sparsified. □

Since the formulas output by the algorithm are k-sparsified, we can immediately apply part (b) of the above lemma to obtain the following.

Corollary 2.6. *Every formula output by the sparsification algorithm contains at most $c_{k,\varepsilon}n$ clauses for $c_{k,\varepsilon} = \sum_{j=1}^{k} \frac{2\theta_{j-1}}{j}$.*

It remains to show that the total number of leaves is small and to bound $c_{k,\varepsilon}$... In any of the formulas F_u, we say that a clause $C \in F_u$ is *new* iff it is not in the original formula at the root; that is, it is either the heart or one of the petals introduced along the path from the root to u. In moving from F_u to F_v or F_w, the new clauses added may eliminate clauses from F_u when the $reduce$ operation is applied.

Lemma 2.7. *If a new clause C' with $|C'| = i < j$ eliminates any new j-clause from F_u then it eliminates at most $2\theta_{j-i} - 2$ total j-clauses, both original and new.*

Proof. Suppose that C' eliminates some new j-clauses and at least $2\theta_{j-i} - 1$ total j-clauses. Consider the first node u' along the path from the root to u that contains all of the eliminated j-clauses. The node u' cannot be the root since C' eliminates at least one new j-clause so it has some parent u'' along the path.

At that parent u'', the formula $F_{u''}$ must be j-sparsified since some new j-clause is added at u' and the sparsification procedure chooses the minimal clause size for the flowers it selects, which must have been larger than j.

By Lemma 2.5(a), there are at most $\theta_{j-i} - 1$ j-clauses of $F_{u''}$ that contain C'. Since there are at least $2\theta_{j-i} - 1$ total j-clauses in $F_{u'}$ that contain C', at least θ_{j-1} of them must have been added in moving from $F_{u''}$ to $F_{u'}$.

Now u' cannot be a left child of u'' since only a single clause $H_{u''}$ is added to $F_{u''}$ and $\theta_{j-i} \geq \theta_0 \geq 2$. On the other hand, if u' is a right child of u'', we must have a set $P' \subseteq P_{u''}$ of at least θ_{j-i} petals that contain C' and these petals in P' must have size j. Let $h = |H_{u''}|$ so the $\geq \theta_{j-i}$ clauses $F'_{u''} \subseteq F_{u''}$ corresponding to P' must have size exactly $h + j$. All of these clauses contain $H' = H \cup C'$ which, since H and C' must be disjoint, has size $h + i$. Therefore the subset $F'_{u''}$ is an $(h + i, j - i)$ flower in $F_{u''}$ contradicting the maximality of h in choosing a flower at node u''. □

Lemma 2.8. *For $j < k$, the number of new clauses of size $\leq j$ that ever get created on a root-leaf path is at most $\beta_j n$.*

Proof. The proof is by induction on j. Let N_j be number of such clauses. For $j = 1$, there are at most $2n$ 1-clauses that could possibly be created which is at most $\beta_1 n$ since $\beta_1 = 2$. Now suppose that it is true for $j - 1$. The number of new clauses created on the path is at most the number of clauses at the end plus the number that were created and then deleted, therefore, if we let $E_{j,k'}$ be the number of k' clauses on the path eliminated by j-clauses, we have

$$\begin{aligned} N_j &\leq N_{j-1} + \sum_{i=1}^{j-1} E_{i,j} + \frac{2n}{j}\theta_{j-1} \qquad \text{by Lemma 2.5(b)} \\ &\leq N_{j-1} + \sum_{i=1}^{j-1} (2\theta_{j-i} - 2)N_i + 2\theta_{j-1}n/j \end{aligned}$$

since there are at most N_i new clauses of size i along the path and each eliminates at most $2\theta_{j-1} - 2$

j-clauses by Lemma 2.7,

$$\begin{aligned}
&\leq \beta_{j-1}n + \sum_{i=1}^{j-1}(2\theta_{j-i}-2)\beta_i n + 2\theta_{j-1}n/j \qquad \text{by induction hypothesis}\\
&< \beta_{j-1}n + 2\sum_{i=1}^{j-1}\alpha\beta_{j-i}\beta_i n + 2\alpha\beta_{j-1}n \qquad \text{since } \theta_i = \alpha\beta_i \text{ for } i \geq 1\\
&\leq 4\sum_{i=1}^{j-1}\alpha\beta_i\beta_{j-i}n \qquad \text{since } \alpha \geq 2,\\
&= \beta_j n
\end{aligned}$$

as required. □

Corollary 2.9. *Every root-leaf path is of length at most $\beta_{k-1}n$.*

Proof. Each step creates at least one new clause. □

Lemma 2.10. *There are at most $(k-1)n/\alpha$ petal steps on every root-leaf path.*

Proof. By Lemma 2.8, for $j \leq k-1$, there are at most $\beta_j n$ total new j-clauses created along the path. Each petal steps that creates j-clauses, creates θ_j such j-clauses, so the number of petal steps that create j-clauses is at most $\beta_j n/\theta_j = n/\alpha$. There are only $k-1$ possibilities for j, which gives the bound. □

Proof of Sparsification Lemma. We suppose that $0 < \varepsilon \leq 1$ and $k \geq 3$. Define $\alpha = \frac{2(k-1)^2}{\varepsilon}\lg\frac{32(k-1)^2}{\varepsilon} \geq 2$. Though it is a pain to actually argue, one can show that $\beta_j \leq 4(32\alpha)^{j-1}$. (This is not simply by induction with this bound.) We can describe each path to a leaf by a sequence of at most $\beta_{k-1}n$ steps with at most $p \leq (k-1)n/\alpha$ petal moves. We use the fact that $\sum_{j=0}^{\ell}\binom{n}{j} \leq 2^{H_2(\ell/n)n}$ where $H_2(\gamma)$ is the binary entropy function equal to $\gamma\lg(1/\gamma) + (1-\gamma)\lg(1/(1-\gamma))$. It follows that the total number of leaves is at most

$$\sum_{p=0}^{(k-1)n/\alpha}\binom{\beta_{k-1}n}{p} \leq 2^{H_2(\frac{k-1}{\alpha\beta_{k-1}})\beta_{k-1}\ n}.$$

It therefore suffices to show that $H_2(\frac{k-1}{\alpha\beta_{k-1}})\beta_{k-1} \leq \varepsilon$. For $\gamma < 1/2$, it isn't hard to show that $H_2(\gamma) \leq \gamma\lg(4/\gamma)$, so

$$\begin{aligned}
H_2(\frac{k-1}{\alpha\beta_{k-1}})\beta_{k-1} &\leq \frac{k-1}{\alpha}\lg(\frac{4\alpha\beta_{k-1}}{k-1})\\
&\leq \frac{\varepsilon}{2(k-1)\lg\frac{32(k-1)^2}{\varepsilon}}\cdot\lg(\frac{16\cdot 32^{k-2}\alpha^{k-1}}{k-1}) \qquad \text{by definition of } \alpha \text{ and the bound on } \beta_{k-1}\\
&\leq \frac{\varepsilon}{2\lg\frac{32(k-1)^2}{\varepsilon}}\cdot\lg(\frac{32\alpha}{k-1}) \leq \varepsilon \qquad \text{since } k-1 \geq 2
\end{aligned}$$

and hence the bound follows. □

k-SUM

ETH has some surprising consequences inside P.

The k-SUM problem is a parameterized variant of SUBSET-SUM that given n integers of $O(\log n)$ bits each and a target t asks whether or not there are k integers that sum to t.

Proposition 2.11. *k-SUM can be solved in $O(n^{\lceil k/2 \rceil} \log n)$ time.*

Proof. Here is an algorithm: Compute the sums of all the $\binom{n}{\lceil k/2 \rceil}$ subsets of the input list of size $\lceil k/2 \rceil$. Sort these sums. For each of the $\binom{n}{\lfloor k/2 \rfloor}$ subsets S of $\lfloor k/2 \rfloor$ input integers run binary search on the sorted list for the value $t - \sum_{i \in S} x_i$. The cost is dominated by the time to compute the sorted list. □

Theorem 2.12. *ETH implies that there is a constant $\varepsilon > 0$ such that k-SUM requires time $\Omega(n^{\varepsilon k})$.*

Proof. The general idea of the argument is that we will see how to use algorithms for k-SUM to decide 3-SAT on formulas with n variables and $O(n)$ clauses. To do this we first reduce 3-SAT to 1-in-3-SAT which will be more convenient; A 3-CNF formula for 1-in-3-SAT will be a yes instance iff there is an assignment that makes exactly one literal true in each clause. To reduce 3-SAT to 1-in-3-SAT we add 4 new variables for each original 3-clause and replace $(x \vee y \vee z)$ by $(\neg x \vee a \vee b)(y \vee b \vee c)(\neg z \vee c \vee d)$. The resulting formula has $O(n)$ variables and $O(n)$ clauses.

(We have several cases: Suppose that the assignment satisfying $x \vee y \vee z$ satisfies y; then we can extend it by setting b and c to false and setting $b = x$ and $s = z$. Suppose that the assignment does not satisfy y; if it also does not satisfy x then we set $a = b = 0$, $c = 1$ and $d = 0$ since it must set z to true. The subcase that z is not satisfied is symmetric. If both x and z are satisfied, then we set $a = 0, b = 1, c = 0, d = 1$. Conversely consider any 1-in-3 satisfying assignment of the new formula. If it sets $b = c = 0$ then y must be true which satisfies $x \vee y \vee z$. It it sets $b = 1$, then we must gave $\neg x = 0$ and hence $x = 1$ which satisfies $x \vee y \vee z$. If it sets $c = 1$ then we must have $\neg z = 0$ which again satisfies $x \vee y \vee z$.)

The main idea of the reduction is that we break up the n variables of the 1-in-3-SAT formula into k chunks of n/k variables each, $V_1, \ldots, V_k$ and have one integer corresponding to each partial assignment to those n/k variables for a total of $N = k2^{n/k}$ integers. We think of each integer as being written in base $k+1$ so there won't be any carries when we add up k of these numbers. There will be one digit for each of the $m = O(n)$ clauses and k digits to correspond to which chunk a number corresponds to. For each truth assignment to V_i the number will have the digit in chunk position i set to 1, with the rest of the chunk bits 0, and will have a 0 or 1 for each clause digit depending on whether it sets 0 or 1 literals in the clause to true. (If it sets more than one literal in a clause to true, we don't include a number for that partial truth assignment since it cannot possibly be extended to 1-in-3-satisfying.) The target t will simply be the all 1's string of length $k + m$ in base $k + 1$. These numbers are easy to compute in $O(km2^{n/k}) = O(Nm) = O(kN \log N)$ time.

By construction, each number takes $O(m \log k)$ bits to represent which is $O(k \log k \log N)$ bits and hence $O(\log N)$ for k fixed.

It is easy to see by construction that there are k numbers that sum to t iff the original formula has a 1-in-3-satisfying assignment iff the original formulas is satisfiable. A running time of $N^{o(k)}$ for k-SUM would give a running time of $(k2^{n/k})^{o(k)}$ for n variable 3-SAT which would be $2^{o(n)}$ contradicting the ETH. □

3 The Strong Exponential-Time Hypothesis (SETH)

The *Strong Exponential-Time Hypothesis (SETH)* is that the sequence of constants s_k for k-SAT given by Corollary 2.3 satisfies $\lim_{k\to\infty} s_k = 1$ or, equivalently, that for every $\varepsilon > 0$ there is a k such that k-SAT requires time at least $2^{(1-\varepsilon)n}$.

SETH was stated as a possibility at the end of the original Impagliazzo-Paturi paper that defined the ETH. Since our best upper bounds on s_k are at least $1 - O(1/k)$, SETH seems consistent with the best algorithms we know.

Unlike the ETH, which we have shown is very robust w.r.t. the choice of NP-complete problems, SETH is specialized to satisfiability problems. Nonetheless, we will see that SETH has many strong and surprising consequences.

Orthogonal Vectors

The *(Boolean) Orthogonal Vectors (OV)* problem takes as input a set of n vectors in $\{0,1\}^d$ and asks whether there is a pair of vectors a, b in the set such that the inner product $a \cdot b = 0$ over the integers.

The obvious algorithm for OV takes time $\Theta(n^2 d)$ by simply computing all of the $\binom{n}{2}$ inner products. The following theorem of Ryan Williams shows that SETH implies that this is nearly optimal for vectors of $O(\log n)$ bits each.

Theorem 3.1. *SETH implies that for every $\varepsilon > 0$, there is a constant $c > 0$ such that OV with $d \le c \log_2 n$ requires time at least $n^{2-\varepsilon}$.*

Proof. By SETH, we can choose n and k sufficiently large that k-SAT on sparse n variable formulas requires time at least $2^{(1-\varepsilon/3)n}$. We can assume that these hard formulas have $m = O_k(n)$ clauses and that n is even. The basic idea will be to split the variables of such a formula F into two parts V_1 and V_2 of size $n/2$ and associate one input vector with each partial truth assignment to V_1 or V_2 respectively. Each input vector will have length $m + 2$.

A vector will begin with 01 if it corresponds to an assignment to V_1 and 10 if it corresponds to an assignment to V_2. For each of the remaining m positions, the vector will have a 0 in position j if the partial assignment satisfies the j-th clause of F and will have a 1 in position j if it does not. Given a formula F, the set of all such vectors can be computed in time $O(Nm)$.

There are a total of $N = 2^{n/2+1}$ vectors and $m + 2$ is $O_k(\log N)$ and hence $O_\varepsilon(\log N)$.

Correctness is easy to see: If there is a satisfying assignment to F then we choose the pair of vectors corresponding to this assignment. Each clause will be satisfied by one or both halves of the assignment so at least one of the two vectors must have a 0 in the clause position and the first two bits of the two vectors each have one 0.

Conversely, if there are two orthogonal vectors, since there are no cancellations, they must correspond to assignments to opposite halves of input bits and, because they are orthogonal, one or the other half of the assignment (or both halves) must satisfy every clause.

An OV algorithm running in time $N^{2-\varepsilon}$ on these inputs would therefore give a k-SAT algorithm running in time time $(2^{n/2+1})^{2-\varepsilon} O(m) = 2^{n(1-\varepsilon/2)+2-\varepsilon} O(m) < 2^{n(1-\varepsilon/3)}$ since n is sufficiently large, contradicting our choice of k. □

Though the Orthogonal Vectors problem appears to have nothing at all to do with k-SAT, this reduction shows that one can view any algorithm for OV as an algorithm for k-SAT.

Rather than having just one set, for OV, we often find it convenient to separate the two sets of vectors into U and V of size N and m coordinates each and the problem asks whether there is a pair of orthogonal vectors, one from U and one from V. Clearly the same reduction shows the same lower bound based on SETH.

Approximating Diameter

We can use the two set variant of OV to get a lower bound for graph problems: Consider the problem of computing the *diameter* of an undirected graph with n vertices and m edges. By computing BFS from each vertex gives an $O(mn)$ algorithm for diameter. Can we do better if we only want to approximate it?

Theorem 3.2. *SETH implies that one cannot decide diameter 2 versus diameter 3 on n vertex m-edge (for $m = O(n \log n)$) in less than $mn^{1-\varepsilon}$ time for any $\varepsilon > 0$.*

Proof. We use the split version of OV with $d = O(\log n)$. We have one node for each element of U, one for each element of V, one node for each of the set C of d coordinates and two extra nodes X and Y that are neighbors. X is joined to every element of U and C. Y is joined to to every element of V and C. So far the graph has diameter exactly 3 since U to V has exactly distance 3 and C has distance 2 from U and V. Now we join every vector $u \in U$ with $u_j = 1$ to node j in C and every vector $v \in V$ with $v_j = 1$ to node j in C.

The resulting graph has $O(n)$ vertices and $m = O(nd) = O(n \log n)$ edges. If every pair $u \in U$ and $v \in V$ have a shared 1, the diameter will be 2; if there is an orthogonal pair $u \cdot v = 0$, the diameter will be 3. An algorithm deciding diameter 2 versus diameter 3 running in time $mn^{1-\varepsilon}$ would be time $O(n^{2-\varepsilon} \log n)$ algorithm for split OV, violating SETH. □

Fine-grained reductions

The reduction from SETH to OV is of a radically different form from our usual format that involves only small changes in input sizes based on complexity bounds. The Sparsification Lemma also involved a very different form of reduction in that we took one problem instance, a n-variable m-clause k-CNF formula and produced a larger number of n variable sparse k-CNF formulas in order to reduce ETH to sparse ETH.

The notion of fine-grained reduction incorporates both of these ideas along with the idea that we only care about the high-order part of the complexity for each problem:

Definition 3.3. Given computational problems A and B and complexity bounds $a, b : \mathbb{N} \to \mathbb{R}^+$, we say that A (a, b)-reduces to B iff for every $\varepsilon > 0$ there is a $\delta > 0$ such that there is an $O(a(n)^{1-\delta})$ time algorithm that takes an input x of size n, and produces $y_1, \ldots, y_k$ of sizes $n_1, \ldots, n_k$ such that $x \in A$ iff every $y_j \in B$ and $\sum_{j=1}^{k} b(n_j)^{1-\varepsilon} \leq a(n)^{1-\delta}$.

The following is essentially immediate from the definition.

Proposition 3.4. *If B can be solved in time $b(n)^{1-\varepsilon}$ for some $\varepsilon > 0$, and A (a, b)-reduces to B then A can be solved in time $O(a(n)^{1-\delta})$ for some $\delta > 0$.*

In particular, we have shown that sparse CNF-SAT $(2^n, n^2)$-reduces to OV on $O(\log n)$-bit words.

4 Algorithms for Orthogonal Vectors?

Write $OV_{n,d}$ for the split Orthogonal Vectors problem where we have two set U and V of n vectors in $\{0,1\}^d$. We've shown that for every $\varepsilon > 0$, SETH implies that $OV_{n,d}$ requires time $n^{2-\varepsilon}$ time for $d \geq c_\varepsilon \log n$ for some constant $c_\varepsilon > 0$. In particular SETH also implies that there is no $n^{2-\varepsilon} poly(d)$ algorithm or indeed $n^{2-\varepsilon} 2^{o(d)}$ algorithm for $OV_{n,d}$.

Other related problems: Subset-Query: Given a collection $S_1, \ldots, S_n \subseteq [d]$ and a database $\mathscr{D}$ of n subsets of $[d]$, is there a set $T \in \mathscr{D}$ such that $S_i \subseteq T$?

Partial-Match: Given n queries $x_1, \ldots x_n \in \{0,1,\star\}^d$ and a size n database $\mathscr{D} \subseteq \{0,1\}^d$, is there a $y \in \mathscr{D}$ that matches some x_i in all of x_i's non-$\star$ positions.

Lemma 4.1. *OV, Subset-Query, and Partial-Match problems are equivalent up to a factor 2 in d.*

Proof. For the reductions between OV and Subset-Query, observe that we can identify the sets $S_1, \ldots, S_n$ with their characteristic vectors in the set U and identify the elements of $\mathscr{D}$ with their complement vectors in V. The condition that u and v are orthogonal implies that whenever $u_i = 1$, we must have $v_i = 0$ and therefore the complement vector $\overline{v}$ has $\overline{v}_i = 1$, which is precisely the condition for containment.

We first reduce OV to Partial-Match, by replacing every 0 in a vector $u \in U$ by $\star$ to get the corresponding x_j vector which is in $\{\star, 1\}^d$ and replace each $v \in V$ by $\overline{v}$ to get $\mathscr{D}$. The OV property ensures that some element of $\mathscr{D}$ has 1's wherever the corresponding x_j does which is a partial match.

Reducing Partial-Match to OV, we use two coordinates for each coordinate of each x_j and each element of $\mathscr{D}$: A 0 in the x_j in Partial-Match becomes 01, a 1 becomes 10 and a $\star$ becomes 00. On the other hand, a 0 in $\mathscr{D}$ becomes 10 and a 1 becomes 01. This ensures that the $\star$ coordinates in Partial-Match can never cause non-trivial inner product but that other values will match iff they do not cause non-trivial inner product. □

Lemma 4.2. *Let $\varepsilon > 0$. For $d \leq (1-\varepsilon)\log n$, $OV_{n,d}$ can be solved in time $\tilde{O}(n^{2-2\varepsilon})$.*

Proof. With these parameters there are at most $n^{1-\varepsilon}$ distinct numbers in each of the lists. We simply mark all elements of $\{0,1\}^d$ that appear in each list and compare the pairs. This takes time $O(dn)$ to compute the two lists and $d \cdot 2^{2d}$ to compare all the pairs which is $O(2^{2-2\varepsilon} \log n)$. □

What if the dimension d is $c(n)\log n$ for some function $c(n) \geq 0$? The following is the best algorithm known for $OV_{n,d}$ due to Abboud, Williams, and Yu. It uses some clever tricks to speed things up based on probabilistic polynomials.

Theorem 4.3. *$OV_{n,d}$ for $d = c(n)\log n$ can be solved in time $n^{2-\Theta(1/\log c(n))}$ by a randomized algorithm with small error.*

The algorithm will show that for sufficiently small sets of vectors, we can repeat the computation of the solution of $OV_{s,d}$ so that the total cost is much less than doing it $O(n^2/s^2)$ times. We split the $OV_{n,d}$ problem into $\theta(n^2/s^2)$ subproblems of the form $OV_{s,d}$ as follows: Split U and V into $q = \lceil n/s \rceil$ sets $U_1, \ldots, U_q$ and $V_1, \ldots, V_q$ where each U_i and V_i has size s and solve all q^2 subproblems.

We can express the subproblem involving U_i and V_j as a depth 3 circuit involving the bits of the vectors in the two sets: In particular there is a big OR of width s^2 at the top, then an AND of width d for the coordinates and a $\neg u_k \vee \neg v_k$ for each $k \in [d]$ for each $u \in U_i$ and $v \in V_j$.

The algorithm will first replace this depth 3 formula using low-degree polynomials and the following result of Razborov that was used by Razborov and Smolensky to prove circuit lower bounds for $AC^0[2]$ circuits.

Lemma 4.4. *For $\ell > 0$, and randomly chosen $r_{ij} \in \{0,1\}$ for $i \in [t]$ and $j \in [\ell]$, define $\mathbb{F}_2$ polynomials $O_t(y_1,\ldots,y_\ell) = 1 + \prod_{i=1}^{t}(1+\sum_{j=1}^{\ell} r_{ij}y_j)$ and $A_t(y_1,\ldots,y_\ell) = \prod_{i=1}^{t}(1+\sum_{j=1}^{\ell} r_{ij}(1+y_j))$. Then for all $y_1,\ldots,y_\ell$, $\Pr_r[O_t(y_1,\ldots,y_\ell) \neq \bigvee_{i=1}^{\ell} y_j] \leq 2^{-t}$ and $\Pr_r[A_t(y_1,\ldots,y_\ell) \neq \bigwedge_{i=1}^{\ell} y_j] \leq 2^{-t}$.*

Proof. If $\bigvee_{j=1}^{\ell} y_j = 0$ then $O_t(y_1,\ldots y_\ell) = 0$ since every $\sum_{j=1}^{\ell} r_{ij}y_j = 0$. Now suppose that $\bigvee_{j=1}^{\ell} y_j = 1$. Then there is some j such that $y_j = 1$. For each fixed r_{ik} for $k \neq j$, exactly one of the two choices of r_{ij} will make $\sum_{k=1}^{\ell} r_{ik}y_k = 1$ since $y_j = 1$, so a single term in the product will be 0 with probably $1/2$. The t terms are independent so the probability that the product is not 0 will be 2^{-t} and hence the polynomial will be 1 with probability exactly $1-2^{-t}$ as required. The properties of polynomial A_t follow since $AND(y_1,\ldots,y_\ell) = \overline{OR}(\overline{y}_1,\ldots,\overline{y}_\ell)$ and $\overline{y}_i = 1 + y_i$ as an $\mathbb{F}_2$ polynomial. □

We will choose a probabilistic polynomial approximating $OV_{s,d}$ as follows: For the bottom gates $\neg u_j \vee \neg v_j$, we write the polynomial $1+u_jv_j$ and feed this into the polynomials for the higher level gates. For each of the fan-in d AND gates at level 2 of the circuit for $OV_{s,d}$ we choose $t = \lceil 3\log s \rceil$ and for the fan-in s^2 top gate we choose $t = 2$. There are s^2 second level AND gates so by a union bound, for each fixed set of inputs, the probability that there is an error in any second level gates is at most $s^2/s^3 = 1/s$. The probability that the output gate computes an incorrect value given correct inputs is at most $1/4$, so the total error probability is at most $1/4 + 1/s \leq 1/3$ for $s \geq 12$. (We can reduce this error probability to polynomially small by repeating independent trials $O(\log n)$ times and take the majority answer. Note that the errors go in both directions.)

When we expand the resulting polynomial for $OV_{s,d}$, it will be important to understand the number of monomials we get. When expanded, the polynomials A_t and O_t are multilinear and have degree t in ℓ variables and therefore has at most $\sum_{i=0}^{t}\binom{\ell}{i} \leq (\ell e/t)^t$ distinct monomials. At the middle level we observe that $t = 3\log s$ and $\ell = d$. The level 1 monomials are of the form $1 + u_jv_j$ which means that there are still only $(ed/3\log s)^{3\log s}$ monomials in the resulting formula (since the u_j and v_j variables are always paired with each other). Finally, these monomials are substituted into the top level polynomial which has $t = 2$ and $\ell = s^2$ and only $O(s^4)$ monomials of degree 2. The final number of monomials $M(s,d)$ is therefore $O(s^4(de/(3\log s)^{6\log s})$ which is $O(s^4(d/\log s)^{6\log s})$ which is at most $s^5(d/\log s)^{6\log s}$.

This number of monomials can be much bigger than the original $O(s^2d)$ circuit size depending on the relationship between s and d. How do we get a savings? The idea will be that computing this polynomial on q^2 pairs of inputs will be less expensive than simply repeating it q^2 times.

The underlying idea of this algorithm will be to use a particular form of fast matrix multiplication due to CopperSmith that we will use to multiply polynomials with few monomials quickly:

Theorem 4.5 (Coppersmith 1982)**.** *One can multiply an $N \times N^{0.172}$ matrix by an $N^{0.172} \times N$ matrix in $O(N^2\log^2 N)$ arithmetic operations.*

Corollary 4.6. *Given a polynomial $P(x_1,\ldots,x_\ell,y_1,\ldots,y_\ell)$ over $\mathbb{F}_2$ with at most $N^{0.1}$ monomials and two list of vectors $U = \{u_1,\ldots,u_N\} \subseteq \{0,1\}^\ell$ and $V = \{v_1,\ldots,v_N\} \subseteq \{0,1\}^\ell$ we can compute all the values $P(u_i,v_j)$ for $u_i \in V$ and $v_j \in V$ in time $\tilde{O}(N^2)$.*

Proof. Let m be the number of monomials of P. Create an $N \times m$ matrix with each row r corresponding to an element of U with the value of the j-th entry equal to the value of the part of the j-th monomial involving U_r. Similarly define an $m \times N$ matrix using the same monomial order for the rows and with c-th column having entry j that is the value of the part of the j-th monomial involving V_c. The matrices can be constructed in time $Nm\ell$. Using Coppersmith's algorithm, their matrix-product can be produced in time $\tilde{O}(N^2)$ and the entry (i,j) of their product taken modulo 2 clearly contains $P(u_i,v_j)$ by construction. □

In our calculations we will have $N = q = \lceil n/s \rceil$ and the number of monomials $m = M(s,d)$ which is at most $s^5(d/3\log s)^{6\log s}$.

We set $s = n^{\delta/\log c(n)}$ for some small $\delta > 0$. Then $\log s$ is $\delta(\log n)/\log c(n) = \delta d/(c(n)\log c(n))$. Hence $d/\log s$ is at most some constant times $c(n)\log c(n)$. When we raise this to the power $6\log s$ we get some $2^{\delta'\log n}$ for a small $\delta' > 0$ which is $n^{\delta'}$. Putting this all together, even when multiplied by s^5 we get that the result is less than $(n/s)^{0.1}$. By the lemma, the total cost is $\tilde{O}(q^2) = \tilde{O}(n^2/s^2)$ which is $n^{2-\Theta(1/\log c(n))}$.

Chan and Williams (SODA 2016, ACM ToA 2021) showed how to compute deterministic polynomials that give correct modular counts of the sum of many OR functions and these polynomials can be used instead of the probabilistic polynomials used here.

5 More on ETH and Parameterized Complexity

Many NP-complete problems have natural parameters in addition to the input size. For example, we can consider the k-CLIQUE problem that asks whether there is a clique of size n in an input graph. We can do the same thing with k-INDEPENDENT-SET, k-DOMINATING-SET, k-VERTEX-COVER. Each of these has an $O(n^k)$ algorithm that is polynomial in n for each fixed value of k. We identify these parameterized problems as *fixed-parameter tractable* iff there is an algorithm with running time $f(k)n^{O(1)}$ for some function f. These algorithms will run efficiently even for large n when k is small.

Lemma 5.1. *There is an algorithm for k-VERTEX-COVER with running time $O(2^k n)$.*

Proof. We define a simple binary search tree of height k as follows:

If G has no edges halt and accept.
Else if $k = 0$ return (failed branch)
Else Choose some edge (u,v) in G.
Try vertex u in the cover: Recursively search for a vertex cover of size $k-1$ in $G \setminus N(u)$.
Try vertex v in the cover: Recursively search for a vertex cover of size $k-1$ in $G \setminus N(v)$.

Clearly this algorithm takes linear time to modify G at each step and has only $O(2^k)$ calls. □

Exercise: Show that ETH implies that k-VERTEX-COVER cannot be solved in time $2^{o(k)}n^{O(1)}$.

On the other hand we can prove that ETH implies that k-CLIQUE is not fixed-parameter tractable in a strong sense. This was shown by Chen, Chor, Fellows, Huang, Juedes, Kanj, and Xia:

Theorem 5.2 (Chen et al. 2004)**.** *ETH implies that there is no $f(k)n^{o(k)}$ algorithm for k-CLIQUE.*

Proof. We have seen that ETH implies that there is $2^{o(n)}$ algorithm for 3-COLOR. Suppose that there is an $f(k)n^{k/\alpha(k)}$ algorithm for k-CLIQUE for some function $\alpha(k)$ that goes to infinity with k. Define $k(n) =$ the largest value of k such that $f(k) \leq n$ and $k^{k/\alpha(k)} \leq n$. Clearly, $k(n)$ is monotone increasing goes to infinity with n. We will set $k = k(n)$.

Given a graph G on n vertices, we split the vertices of G into k groups of size n/k. We define a new graph H where each vertex of H corresponds to a 3-coloring of one of the k groups of vertices of G. H has $k3^{n/k}$ vertices. We connect each pair of vertices of H iff the colorings don't conflict with respect to

G. (Vertices for partial colorings that themselves are not consistent with G will be isolated.) It is easy to see that H has a k clique iff G has a proper 3-coloring.

The running time of the presumed k-CLIQUE algorithm will be $f(k)(k3^{n/k})^{k/\alpha(k)} \leq nk^{k/\alpha(k)}3^{n/\alpha k} \leq n^2 3^{n/\alpha k(n)}$. Since $k(n)$ goes to infinity with n, $\alpha(k(n))$ goes to infinity with n so this is a $2^{o(n)}$ algorithm contradicting the ETH. □

Using standard fixed-parameter reductions that are polynomial in the input size and change the parameter by at most a constant factor, one can get lower bounds for other problems also.

Corollary 5.3. *ETH implies that there is no $f(k)n^{o(k)}$ algorithm for k-SET-COVER, k-HITTING-SET, k-BIPARTITE-DOMINATING-SET, k-CONNECTED-DOMINATING-SET.*

6 Longest-Common Subsequence

At STOC 2015, Backurs and Indyk showed that another problem that OV reduces to is is Edit Distance. Later at FOCS 2015, Abboud, Backurs, and Vassilevska-Williams, and Bringman and Kunneman extended this to finding the length of the *Longest Common Subsequence (LCS)*, with the latter showing that this hardness extends to the case of the binary alphabet. LCS is equivalent to a special case of Edit Distance in which the cost of insertion or deletion is 1 and the cost of substitution is 2. More formally, given strings $A, B \in \Sigma^*$ define $LCS(A, B)$ to be the maximum k such that there are sequences $i_1 < \ldots < i_k$ and $j_1 < \ldots < j_k$ for which $A_{i_1} = B_{i_1}, \ldots, A_{i_k} = B_{j_k}$.

There are simple natural dynamic programming algorithm for Edit Distance (and hence LCS) that run in time $O(n^2)$ where $|A| = |B|$ and the best algorithms known only shave off a $\log n$ factor.

Theorem 6.1. *SETH implies that LCS over binary strings does not have an $O(n^{2-\delta})$ algorithmn for any $\delta > 0$.*

The proof of this begins by looking at a problem closer to OV. Define $LCS\text{-}PAIR_{N,m}$ to be the problem: Given sequences $a_1, \ldots, a_N, b_1, \ldots, b_N \in \Sigma^m$, find the $\max_{i,j} LCS(a_i, b_j)$. We describe the reduction from $OV_{N,m}$ to LCS-Pair along the lines given by Bringman and Kunneman.

Lemma 6.2. *$OV_{N,m}$ reduces in linear time to $LCS\text{-}PAIR_{N,cm}$ for some constant c.*

Proof. The general idea is to produce a local substitution of each character of u_i and v_j according to different substitutions.

Define the strings $0_u = 10011$, $1_u = 11100$, $0_v = 11001$, and $1_v = 00111$. Observe that $LCS(0_u, 0_v) = LCS(0_u, 1_v) = LCS(1_u, 0_v) = 4$ but $LCS(1_u, 1_v) = 3$.

We want to ensure that any LCS for a_i and b_j involves a character-by-character match of u_i and v_j in $\{0,1\}^m$. To do this we define $code_u(u_i)$ to be the string where we replace each 0 or 1 of u_i by the corresponding 0_u or 1_u and we separate each pair by, say, a string of three 2's; do the same for $code_v(v_j)$ except we use 0_v and 1_v instead. Define $a_i = code_u(u_i)$ and $b_j = code_v(v_j)$. In particular, if $u_i = 001$ and $v_j = 011$ then

$$a_i = code_u(001) = 100111222100112221100$$
$$b_j = code_v(011) = 110012222001112220111$$

Therefore the total string length m' is $8m - 3$. It is clear that every LCS of a_i and b_j and must match all the 2's, which means that corresponding coordinates much be matched. If u_i and v_j are orthogonal

then $LCS(a_i, b_j) = 4m + 3(m-1) = 7m-3$, which we denote by S. On the other hand, if u_i and v_j are not orthogonal then the contribution is only 3 instead of 4 in all the coordinates with common 1's and hence $LCS(a_i, b_j) \leq S-1$. The reduction simply compares the length of the LCS to S. □

To obtain a lower bound for LCS, we will need to concatenate these strings to a single pair of strings A and B so that the length of the LCS of the whole string will be larger iff there is *some* pair of orthogonal u_i and v_j. To do this we need to control things so that we know the contribution of each the failed matches also. With the above construction, the more overlapping 1's, the worse the value. To fix this we use a slight modification of the above construction that always guarantees a match of precisely 1 less than the maximum possible.

To do this we add an extra dummy coordinate to the encoding. Define $code'_u(u_i) = code(u_i 0)$ and $code'_v(v_j) = code(v_j 1)$ as well as an extra "easy string" $e = code(0^m 1)$. Observe that $(u_i, 0)$ and $(v_j, 1)$ are orthogonal iff u_i and v_j are. Also every vector $(v_j, 1)$ has precisely one coordinate with overlapping 1's with $(0^m, 1)$. Let $m'' = 8m+5$ be the length of $code'_u(u_i)$. Now define

$$a'_i = code'_u(u_i)3^{m''}e$$
$$b'_j = 3^{m''}code'_v(v_j)3^{m''}$$

The total length of each of a'_i and b'_j is $\ell = 3m'' = 24m+15$. Observe that any LCS for the two strings must match one of the two groups of 3's in b_j in its entirely to the middle group of 3's in a_i and then include an LCS between $code'_v(v_j)$ and either $code'_u(u_i)$ of e. If u_i and v_j are orthogonal then we get a total contribution of $S' = m'' + S + 7$ where S is the value from the above lemma since the extra coordinate gives a total contribution of 7. In particular $S' = 8m+5+7m-3+7 = 15m+9$. In the latter case there is precisely 1 segment where the contribution is 3 instead of 4 so we get $m'' + S + 6 = S' - 1$ (there is a contribution of 3 for the matching 2's and another 3 from the LCS between 1_u and 1_v).

We now are in a position to describe the construction of the strings A and B for the proof of the theorem. The idea will be that the contribution will be to allow an arbitrary rotated alignment of the combined string of encodings $b_1, \ldots, b_N$, suitably separated, with the the string of encodings $a_1, \ldots, a_N$. This will necessitate two copies of one of the strings to allow for the orthogonal pair (u_i, v_j) to have $i > j$ as well as $i \leq j$.

We define the strings

$$A = a'_1 4^\ell a'_2 4^\ell \ldots 4^\ell a'_N 4^\ell a'_1 4^\ell a'_2 4^\ell \ldots 4^\ell a'_N$$
$$B = 4^{N\ell} b'_1 4^\ell b'_2 4^\ell \ldots 4^\ell b'_N 4^{N\ell}.$$

In particular, both $|A| = |B| = (4N-1)\ell$ which is $O(Nm)$. A has $(2N-1)\ell$ 4's and $2N\ell$ characters from the a'_i which is $O(Nm)$, while B has has $N\ell$ characters from the b'_j and $(3N-1)\ell$ 4's. Clearly one can match all of the $(2N-1)\ell$ 4's in A in B.

If some pair u_i and v_j are orthogonal we can get a common subsequence as follows: Write $\Delta = i-j+1$. We get a common subsequence by aligning the subsequence $b_1 4^\ell \ldots 4^\ell b_N$ with some $a_\Delta 4^\ell \ldots 4^\ell a_{\Delta-1}$ where we write $a_k = a_{N+k}$ for $k \leq 0$. and matching up all the 4's outside of the subsequence in A with the 4's at the beginning and end of B. Observe that this aligns a'_i and b'_j. In total, this matches all $4^{(2N-1)\ell}$ 4's in A plus S' of the characters in a'_i and b'_j and at least $S'-1$ characters for each of the other $N-1$ positions for a total LCS length of $\geq S'' = (2N-1)\ell + NS' - (N-1)$.

On the other hand, suppose that there is no pair of orthogonal elements u_i, v_j. Consider any LCS of A and B and consider how it matches up characters of B inside A. If there is some b'_j that has characters

matching with more than one of the a'_i strings then it must NOT match any of the ℓ 4's in between them. This costs ℓ in terms of the 4's but could potentially increase the number of matches inside the string b'_j, but since matching b'_j to just one of the a'_i would already give $S'-1$ so in the best case the amount of increase would be at most $\ell - S' + 1$ which is strictly less than the loss of 4's. Therefore, any LCS of A and B must match and $(2N-1)\ell$ 4's in A and each b'_j to at most one of the a'_i strings. No matter which i it is matched to, the longest part of the LCS inside b'_j can be at most $S'-1$, so the the total length of the LCS is at most $(2N-1)\ell + N(S'-1) = S''-1$. (This length is actually achievable.)

Therefore, $OV_{N,\log_2^2 n}$ is reducible in time $O(n) = O(N^2 \log^2 n)$ to LCS on length n strings over $\{0,1,2,3,4\}$. This proves the theorem except for the reduction of the alphabet size to binary. That reduction requires a more subtle way to put the various strings in order to replace the symbols $2,3,4$ by binary strings.

Encoding LCS using binary strings instead

The idea, due to Bringman and Künneman, is quite general. It assumes that we have two sets of binary codes, one for $\mathscr{C}_x$ and one for $\mathscr{C}_y$ with the following properties:

- $\mathscr{C}_x \subset \{0,1\}^{\ell_x}$ and has s_x 1's.
- every string in $\mathscr{C}_y \subset \{0,1\}^{\ell_y}$ and has s_y 1's.

For example, $\mathscr{C}_x = \{0_u, 1_u\}$ we have $\ell_x = 5$ and $s_x = 3$ and $\mathscr{C}_y = 0_v, 1_v$ has $\ell_y = 5$ and $s_y = 3$. We saw that for single bits u and v, $x = code_u(u) \in \mathscr{C}_x$ and $y = code_v(v) \in \mathscr{C}_y$, $LCS(x,y) = 4$ if $u \cdot v = 0$ but $LCS(x,y) = 3$ if $u \cdot v = 1$.

The general idea is to build an *alignment gadget* to encode longer strings of elements from $\mathscr{C}_x$ and $\mathscr{C}_y$ such that the best way to build an LCS must involve matching individual codes. This will be done by separating these codes using very long blocks of 0's and very long blocks of 1's.

Define $\gamma_1 = \ell_x + \ell_y$, $\gamma_2 = 6(\ell_x + \ell_y)$, $\gamma_3 = 10(\ell_x + \ell_y) + 2s_x - \ell_x$ and $\gamma_4 = 13(\ell_x + \ell_y)$.

For a string $z \in \mathscr{C}_x \cup \mathscr{C}_y$, we write $G(z) = 1^{\gamma_2} 0^{\gamma_1} z 0^{\gamma_1} 1^{\gamma_2}$. Observe that for $x \in \mathscr{C}_x$ and $y \in \mathscr{C}_y$, $LCS(G(x), G(y)) = LCS(x,y) + 2\gamma_1 + 2\gamma_2 = LCS(x,y) + 14(\ell_x + \ell_y)$. This follows because there is no advantage to NOT matching the corresponding strings of 0's and 1's at the beginning and end of the $G(x)$ and $G(y)$.

For $n \geq m$, $x_1, \ldots, x_n \in \mathscr{C}_x$ and $y_1, \ldots, y_m \in \mathscr{C}_y$ define

$$x = G(x_1) 0^{\gamma_3} G(x_2) 0^{\gamma_3} \cdots 0^{\gamma_3} G(x_n) \tag{2}$$

$$y = 0^{n\gamma_4} G(y_1) 0^{\gamma_3} G(y_2) 0^{\gamma_3} \cdots 0^{\gamma_3} G(y_m) 0^{n\gamma_4}. \tag{3}$$

Observe that since x_i has s_x 1's so $G(x_i)0^{\gamma_3}$ so it has $2\gamma_2 + s_x = 12(\ell_x + \ell_y) + s_x$ 1's. It also has $2\gamma_1 + \ell_x - s_x + \gamma_3 = 2(\ell_x + \ell_y) + \ell - s_x + 10(\ell_x + \ell_y) + 2s_x - \ell_x = 12(\ell_x + \ell_y) + s_x$ 0's, so it is balanced. (Further observe that this is at most γ_4.) Also, every prefix of $G(x_i)0^{\gamma_3}$ has at least as many 1's as 0's since $G(x_i)$ begins with γ_2 1's and $G(x_i)$ has at most $2\gamma_1 + \ell_x - s_x < \gamma_2$ 0's.

By our observation, x has fewer than $n[12(\ell_x + \ell_y) + s_x]$ 0's which is at most $n\gamma_4$.

We want to claim that the best LCS of x and y is given by the max over all choices of $0 \leq \Delta \leq n-m$ of the matching that aligns the $G(x_{\Delta+1})0^{\gamma_3} \cdots 0^{\gamma_3} G(x_{\Delta+m})$ with $G(y_1)0^{\gamma_3} \cdots 0^{\gamma_3} y_m$ and matches all the remaining 0's in x with the 0's at the beginning and end of y, since there are enough of them in total. That number of extra 0's matched at the ends of such an alignment is independent of the choice of Δ, since the number of 0's in each x_i is exactly the same, and the total length of the LCS is simply some

fixed value plus $\sum_{k=1}^{m} LCS(x_{i+\Delta}, y_i)$, which is the sum of the shifted LCS alignments of the original encodings.

It remains to show that no other alignment can give a better LCS than one of these alignments Δ. Consider some other alignment of x with y. Now for each of the blocks $G(y_1), \ldots, G(y_m)$ of y we can define $x(j)$ to be a substring of the x string that contains the portion of the LCS that matches $G(y_j)$ and $z(j)$ for $j = 1, \ldots, m-1$ to be the substring of x that aligns with the 0^{γ_3} after $G(y_j)$. Given this alignment, if $x(j)$ contains more than half of some x_i string (a piece inside $G(x_i)$ then, since there might be more than one such x_i, define $i^*(j)$ to be the leftmost such i. This function i^* might not be defined for some values of j but, if it is, then $i^*(j) > i^*(j')$ for $j > j'$. Thus i^* is a partial increasing 1-1 function from $[m]$ to $[n]$.

Claim: If $i^*(j)$ is defined then there are at least as many characters of not matched between $x(j)$ and $G(y_j)$ as there are not matched between $G(x_{i^*(j)})$ and $G(y_j)$.

If $x(j)$ touches some $G(x_i)$ other than $G(x_{i^*(j)})$ then there was more than one candidate for $i^*(j)$ then there is huge number of unmatched characters in $x(j)$ since it contains the intervening sequence $0^{\gamma_1}1^{\gamma_2}0^{\gamma_3}1$ if it is to the right or $10^{\gamma_3}1^{\gamma_2}0^{\gamma_1}$ if it is to the left which yields a much greater distance number than $\ell_x + \ell_y$ which is an upper bound on the number of unmatched characters between $G(x_{i^*(j)})$ and $G(y_j)$.

If not, then the string $x(j)$ is contained in $0^{\gamma_3}G(x_{i^*(j)})0^{\gamma_3}$. Since $G(y_j)$ begins and ends with $\gamma_2 > \ell_x + \ell_y$ 1's, $x(j)$ must begin and end with 1's or the LCS with $G(y_j)$ will have more than $\ell_x + \ell_y$ unmatched characters. Further $x(j)$ must have more than $|G(y_j)| - \gamma_2$ characters or there will again be more than γ_2 unmatched characters. Note that the right substring $x_{i^*(j)}0^{\gamma_1}1^{\gamma_2}$ or the left substring $1^{\gamma_2}0^{\gamma_1}x_{i^*(j)}$ have length $\ell_x + \gamma_1 + \gamma_2$, they are too short, so the string $x(j)$ must look like $1^a0^{\gamma_1}x_{i^*(j)}0^{\gamma_1}1^b$ for some a and b. It is easy to see that the fewest mismatches would occur when $a = b = \gamma_2$ which is exactly the case of the claim.

Claim: If $i^*(j)$ is not defined then $x(j)$ and $G(y_j)$ have at least $\ell_x + \ell_y$ mismatches which is at least than the number of mismatches between $G(y_j)$ and any $G(x_i)$.

Since $i^*(j)$ is not defined then there is some i such that $x(j)$ is contained in $x_i0^{\gamma_1}1^{\gamma_2}0^{\gamma_3}1^{\gamma_2}0^{|gamma_1}x_{i+1}$ and contains less than half of x_i and x_{i+1} (or an end case where it begins or ends with 1^{γ_2}). If $x(j)$ contains 1's on both sides of the central 0^{γ_3} then it contains γ_3 0's which exceeds the number in $G(y_j)$ by more than $\ell_x + \ell_y$. If $x(j)$ only contains 1's on one side of the central $)^{\gamma_3}$ then it contains at most $s_x + \gamma_2 \leq \ell_x + \gamma_2$ 1's. However $G(y_j)$ contains at least $2\gamma_2$ 1's and $\gamma_2 - \ell_x > \ell_x + \ell_y$ so there are more than $\ell_x + \ell_y$ unmatched characters.

Since the $z(j)$ are perfectly aligned in the alignments based on Δ, the middle sequence of y ignoring the 0's at the ends is always aligned at least as well as before. The bottom line from this is that the total cost is at least as large as one of the consecutive alignments.

7 LCS is hard even given a very weak SETH

Abboud, Hansen, Vassilevska-Williams, and Williams at STOC 2016 showed that LCS is hard even if very high complexity *CIRCUIT-SAT* is hard. We sketch a simpler proof of their theorem.

Theorem 7.1. *If there is no $2^{n-o(n)}$ algorithm to compute satisfiability for*

- *depth $o(n)$ circuits,*
- *size $2^{o(n)}$ Boolean formulas, or*

- *verifiers given by space $o(\sqrt{n})$ nondeterministic Turing machines,*

then LCS over an alphabet of size $n^{o(1)}$ does not have an $O(n^{2-\varepsilon})$ algorithm for any $\varepsilon > 0$.

One key observation is that the previous construction did not make use of the full range of parameters possible in constructing A and B. All we needed for the conclusion is that each a_i for the *LCS-PAIR* problem has size $N^{o(1)}$ rather than restricting it to $O(\log^2 N)$.

We note that by the usual formula balancing construction, the first two classes are identical. The third follows from the first by the fact that $\mathsf{NSPACE}(S(n)) \subseteq \mathsf{DEPTH}(S^2(n))$.

Proof Sketch. We use the same framework to convert from *LCS-PAIR* to *LCS* so we just give the description for the a_i and b_j strings for *LCS-PAIR*. We use the same separation of the input variables into U and V and $N = 2^{n/2}$ assignments to each player as before. (We will index assignments by α and β so as not to confuse notation.) We will define a_α and b_β recursively. We will define a a^v_α and b^v_β for each gate v in the circuit. We will first produce them as weight formulas and then argue that the weights can be removed. We will assume wlog that the depth $o(n)$ circuit has been converted to a balanced formula in which all negations have been pushed to the leaves and each gate at depth k has two predecessors at depth $k-1$.

We create a^v_α and b^v_β by induction on $k = depth(v)$. We will construct these so that $LCS(a^v_\alpha, b^v_\beta)$ is maximal iff v evaluates to 1 on input (α, β).

Suppose that $k = 0$. Then u is labelled by a literal ℓ_i that is either x_i or $\neg x_i$. Set

$$a^u_\alpha = \begin{cases} * & \text{if } i > n/2 \text{ or } \ell_i(\alpha) = 1 \\ \$ & \text{otherwise,} \end{cases}$$

and

$$b^u_\beta = \begin{cases} * & \text{if } i \leq n/2 \text{ or } \ell_i(\beta) = 1 \\ \# & \text{otherwise,} \end{cases}$$

We will ensure that none of the a strings contain $\#$ and none of the b strings contain \$. Clearly both will have a $*$ iff literal ℓ_i is set to true on assignment (α, β).

Now suppose that u has children v and w at depth $k-1$.
If $u = v \wedge w$ then define

$$a^u_\alpha = P_k a^v_\alpha Q_k a^w_\alpha P_k$$
$$b^u_\beta = P_k b^v_\beta Q_k b^w_\beta P_k$$

where P_k and Q_k are new symbols of weight $W_k = 3^k$. Clearly any optimal LCS for this pair must align all of the P_k and Q_k and hence will have a maximal LCS iff both $LCS(a^v_\alpha, b^v_\beta)$ and $LCS(a^w_\alpha, b^w_\beta)$ are maximal.
If $u = v \vee w$ then define

$$a^u_\alpha = P_k a^v_\alpha Q_k a^w_\alpha P_k$$
$$b^u_\beta = Q_k b^w_\beta P_k b^v_\beta Q_k$$

In this second case, the optimal LCS must match precisely one P_k and one Q_k and include the LCS of precisely one of the two pairs (a^v_α, b^v_β) or (a^w_α, b^w_β). Hence it will be maximal iff at least one of the

two matches is maximal. (Note that the target weight of the maximal match is different depending on whether the gate is an $\vee$ or an $\wedge$ gates, but this doesn't matter since we know the target size from the property of the circuit itself. We could alternatively simply use higher weight in the OR gadget to make them equal. We end up with $o(n)$ different symbols.

The weighted a_α and b_β strings are the ones for the output gate of the circuit. We can remove all the weights easily by replacing a symbol σ with weight w by w consecutive copies of σ without changing the LCS size. Clearly we can just use entire blocks corresponding the weighted LCS. The claim is that this is the best we can do. Suppose that we have an LCS that does not match things blockwise for σ and consider the leftmost *partial match* of σ between A and B. That, say matches the i-th element in the block in A to the j-th element in the block in B. This (i, j) match splits the LCS. Suppose wlog that $i \leq j$. Clearly that are at most $w - i + 1$ elements of the LCS that touch either of these two blocks since there is no match in the block to the left of the (i, j) match and matches in the LCS cannot cross each other. We can do at least well (obtaining w matched pairs in the blocks) by locally replacing all of those edges by a full matching of the two blocks.

Note that the total weight for depth k is c^k for some constant c, each resulting string has length $2^{o(n)} = N^{o(1)}$. □

8 Circuit-SAT Algorithms Imply Lower Bounds

We consider Boolean circuits over the De Morgan basis consisting of binary $\wedge$, $\vee$ and $\neg$.

Theorem 8.1 (Shannon)**.** *At least a* $1 - o(1)$ *fraction of Boolean functions require circuit size at least* $2^n/n - o(2^n/n)$.

Proof. We will not optimize the constants and assume for convenience that the negations have been pushed to the inputs via de Morgan's Law so we have access to literal gates $x_1, \ldots, x_n, \overline{x_1}, \ldots, \overline{x_n}$ and all internal nodes are labelled via $\wedge$ and $\vee$. We count the number of circuits with S $\wedge$ and $\vee$ gates. We can label each internal gate based on its two inputs which are each either gates or literals and its type. There fewer than $(S+2n)^2$ choices of inputs per gate and 2 choices of gate label so there are $[2(S+2n)^2]^S$ possible gates and S choices for the name of the output gate. In total, this is at most $S^{O(S)} = 2^{O(S \log S)}$ different circuits of size at most $S \geq 2n$. On the other hand, there are 2^{2^n} possible Boolean functions. Therefore for some constant $\varepsilon > 0$ if $S \log S$ is at most $\varepsilon 2^n$, the number of circuits of size at most S is a vanishingly small fraction of all functions. This implies that S is $\Omega(2^n/n)$. The sharper bounds follows from a more careful count since there are $S!$ different ways of naming the internal gates; this shows that the number of distinct functions computed by circuits of size at most S is $S^{S+o(S)}$. □

This is matched by a result of Lupanov.

Theorem 8.2 (Lupanov)**.** *Every Boolean function* f *on* n *bits can be computed by a Boolean circuit of size* $2^n/n + o(2^n/n)$.

Sketch. Observe that over all assignment α to the last $n-k$ variables there are only 2^{2^k} possible functions among $f(x_1, \ldots, x_k, \alpha)$. We can compute all such functions using dynamic programming in total size at most 2^{2^k}.

We now build up the Boolean functions on longer prefixes that we need using $F(x_1, \ldots, x_{m+1}, \beta) = \neg x_{m+1} \wedge F(x_1, \ldots, x_m, 0, \beta) \vee x_{m+1} \wedge F(x_1, \ldots, x_m, 1, \beta)$.) This add $O(2^{n-k}) = O(2^n/2^k)$ gates in total. We choose k such that 2^{2^k} is roughly $2^n/n$. In that case 2^k is $n - \log_2 n$ and the total number of gates is $O(2^n/n)$. Again, with a slightly sharper construction and analysis one can obtain the claimed bound. □

Despite the fact that almost all functions are exponentially hard for circuits, getting our hands on any hard functions is quite elusive.

The largest classes of circuits where we have very simple explicit functions with strong lower bounds was the class $AC^0[p]$ for p prime which consists of constant-depth circuits with unbounded fan-in AND gates, OR gates, and mod-p gates where a mod-p gate outputs 0 if the sum of its Boolean inputs is 0 modulo p and outputs 1, otherwise. (Alternatively, in the special case that p is a prime a mod-p gate with inputs $y_1, \ldots, y_k$ computes the quantity $(y_1 + \ldots y_k)^{p-1}$ mod p. This does not nicely extend to cases of mod-m gates.) Razborov and Smolensky used the probabistic polynomials that we described before (and its generalization to other moduli other than 2) to show that any such circuit that computes the mod-q function for $q \neq p$ prime requires exponential size.

Though Shannon's bounds are far from explicit, if we go to a sufficiently high complexity class, we can find functions that require large circuits.

Proposition 8.3. *There is a function family in EXPEXP (double exponential-time) that requires circuit size* $\Omega(2^n/n)$.

Proof. The idea for an input x of size n is to search through all possible tables of Boolean functions on n bits until we find one that has circuit complexity at least $S(n) = \varepsilon 2^n/n$. We search through all possible 2^{2^n} truth tables of such functions starting with the all 0's table in lexicographic order. For each such function we try all circuits of size $< S(n)$ of which there are fewer than 2^{2^n}. For each such circuit, we evaluate the function on all 2^n inputs which takes only $S(n)2^n$ time. If we find a circuit that is correct on all inputs we move on to the next truth table in order. The value of the function on input x is defined to be the entry for x in the first truth table where all of the circuits fail. □

It is known that for every integer $k \geq 0$, the complexity class $\Sigma_2^P = NP^{NP}$ contains functions that require size at least $\Omega(n^k)$. Using an exponential translation of this result, we can obtain that the complexity class $EXPEXP$ in the above proposition can be reduced to $E^{\Sigma_2^P}$ where $E = DIIME(2^{O(n)})$.

What about $NEXP$, nondeterministic exponential time? It is consistent with our knowledge that every problem in $NEXP$ has polynomial-size circuit. (despite the fact that we expect that there are problems in NP that don't have polynomial-size circuits - which would be stronger than $P \neq NP$ but incomparable with ETH and SETH). Ryan Williams proved the following stronger result.

Theorem 8.4 (Williams)**.** *For any class of circuits $\mathscr{C}$ that is closed under $\vee$ and $\wedge$ if $\mathscr{C}$-SAT for circuits of size n^k for all k can be solved deterministically in time $2^n/n^{10}$ then $NEXP$ does not have polynomial-size circuits in $\mathscr{C}$.*

Note that this is only a polynomials savings over brute force which would take time $O(2^n n^k)$ using standard CIRCUIT-SAT algorithms.

Theorem 8.5 (Murray-Williams)**.** *There is a $\varepsilon > 0$ such that for any class of circuits $\mathscr{C}$ that is closed under $\vee$ and $\wedge$ if $\mathscr{C}$-SAT for circuits of size n^k for all k can be solved deterministically in time 2^{n-n^ε} then $NTIME(n^{polylogn})$ does not have polynomial-size circuits in $\mathscr{C}$.*

Theorem 8.6 (Murray-Williams)**.** *If $\mathscr{C}$-SETH is false then for every k, NP contains functions that require $\mathscr{C}$-circuits of size larger than n^k.*

In the following, we sketch the most basic of the ideas for the arguments which are quite involved. Since the assumptions are algorithmic, we will need to leverage *some* lower bound. That lower bound is the nondeterministic time hierarchy theorem.

Theorem 8.7 (Nondeterministic Time Hierarchy Theorem)**.** *For all $T(n)$ that is the running time of some TM and functions t such that $t(n+1)$ is $o(T(n))$, $NTIME(T(n)) \not\subseteq NTIME(t(n))$.*

The proof of this theorem which is covered in CSE 531 is much trickier than the one for the deterministic time hierarchy theorem and it doesn't work for very fast growing functions.

Corollary 8.8. $NTIME(2^n) \not\subseteq NTIME(o(2^n))$.

The general idea of the structure of the proof is to show that if we have both

1. $NEXP$ has polynomial-size $\mathscr{C}$ circuits, and
2. There is a deterministic $\mathscr{C}$-SAT algorithm with running time $2^n/n^{10}$ for all polysize $\mathscr{C}$ circuits

then we can produce an $NTIME(o(2^n))$ algorithm for any $L \in NTIME(2^n)$, which violates the nondeterministic time hierarchy theorem.

To use assumption 1, we apply the following result of Impagliazzo, Kabanets, and Wigderson.

Lemma 8.9 (Easy Witness Lemma)**.** *If $NEXP$ has polynomial-size circuits then for every $NEXP$ verifier V for a language $L \in NEXP$ (which runs in deterministic time 2^{n^c} and given as input a string x of length n and accepts iff there is a witness string y of length 2^{n^c} such that $V(x, y)$ accepts), then for all large enough $x \in L$ there is a circuit $C_{V,x}$ with n^c inputs bits and size n^d for some constant d computing a Boolean function such that the truth table T of the function $C_{V,x}$ is a witness for $x \in L$, namely $V(x, T)$ accepts.*

This lemma can be specialized to specific circuit classes also. The general idea of its proof is very complicated. The idea is that if it isn't true then you can nondeterministically find truth tables of very hard functions, then one can use this truth table as the basis of a pseudorandom generator (PRG). This PRG can let you derandomize a PCP for NEXP and get that MA (Merlin-Arthur which is like NP but with randomized verifiers) is in NSUBEXP ($\bigcap_{\varepsilon>0} NTIME(2^{n^\varepsilon})$). The assumption that NEXP has polysize circuits directly implies that everything in EXP has polysize circuits, which implies that EXP is the same as MA via the Karp-Lipton theorem, which implies that EXP is in NSUBEXP. But the NEXP has polysize circuits means that NSUBEXP only needs circuits of some fixed polynomial size n^k for some fixed k. That contradicts what we know; in particular, we know that EXP requires circuits of arbitrarily large polynomial size since it contains Σ_2^P. Whew!

The point here is that we can reduce the amount of nondeterminism for $NTIME(2^n)$ from 2^{n^c} down to n^d, namely guessing the circuit that must exist. (Note that we only care that it is down to $o(2^n$ when we started with $c = 1$.) This circuit gives us a nice efficient way to access any bits of the witness string also. The construction gives a $\mathscr{C}$-circuit of this size if NEXP has polynomial-size $\mathscr{C}$-circuits.

We haven't used the second of our assumptions yet, namely that there is a $2^n/n^{10}$ deterministic algorithm for $\mathscr{C}$-CIRCUIT-SAT. We need to be able to replace the $O(2^n)$ time for the verifier V for L with something shorter. If we don't, we won't save anything. To do this one needs a very special structured PCP for NEXP. The general idea of a PCP is that rather than having a verifier V that looks at the whole witness y, one encodes y in a larger string $E(y)$ so that one can examine some small randomly chosen portion of $E(y)$ and be able to check that this should be accepted efficiently based on that small portion. For NEXP, the standard PCPs examine only a polynomial size portion of the exponentially long string $E(y)$. These PCPs have the property that if $x \in L$ then no matter what the portion examined in $E(y)$ will cause the verifier to accept and any $x \notin L$, the string E will be rejected with constant probability. In particular, we need to argue that the entire process of taking the random bits r that let one choose which part of $E(y)$ to look at, compute those bits based on the circuit that produces y, and make the decision about whether to reject can be done in some n^k size $\mathscr{C}$-circuit for some k. It happens that such very structured PCPs do exist once we know that we have an n^d size circuit for the bits of y.

We just need to know whether there is an r that causes us to reject, which is where we use the C-CIRCUIT-SAT algorithm. This runs in $o(2^n)$ deterministic time. Therefore, the whole algorithm, both the guess and verifier run in $o(2^n)$ time, which contradicts the nondeterministic time hierarchy theorem.

In the application, we even know that, say, either all the r choices are bad (cause the PCP to accept) or at least half of them cause the PCP to reject. In fact, we get the extra promise that if the circuit is satisfiable then at least half the assignments are satisfying.

Therefore we can replace all the $\mathscr{C}$-CIRCUIT-SAT problems with their "Gap" versions that have this promise.

Williams gave a clever algorithm using the polynomial method for computing satisfiability of circuits in the class $ACC^0 \supset \bigcup_m AC^0[m]$ which consists of constant-depth unbounded fan-in circuits of AND, OR, and mod-m gates for arbitrary many different m. (The ACC in ACC^0 stands for Alternating Circuits with Counters.)

Theorem 8.10. *For $\varepsilon > 0$, there is a deterministic algorithm for satsfiability of n variable ACC^0 circuits of size at most 2^{n^ε} in time 2^{n-n^ε}.*

Corollary 8.11 (Williams, Murray-Williams)**.** *$NEXP$ does not have polynomial-size ACC^0-circuits. In fact, $NTIME(n^{polylogn})$ does not have such circuits.*

9 SAT solvers

DPLL Algorithm

This basic algorithm is a simplified version of one by Davis, Logeman, and Loveland (CACM 1962), modifying an approach of Davis and Putnam (CACM 1960) which was quite different. The original goal of both was as a component of a decision procedure for first-order logic. Confusion over the terminology and attribution for the algorithm in the 1990s was settled by agreement to reference all of the authors.

Algorithm 4 The DPLL algorithm for satisfiability search. This is invoked as DPLL(F,nil). This is a *complete* algorithm in that failure of the search implies that F is unsatisfiable.

```
1:  function DPLL(F, A)
2:      while F contains a clause x of size 1 do
3:          F ← F_{x←1}; A ← (A, x)                    ▷ Unit propagation
4:      if F is empty then
5:          Halt and output satisfying assignment A
6:      if F contains the empty clause ⊥ then
7:          return
8:      else choose unset literal x                     ▷ Decision literal
9:          DPLL( F_{x←1} , (A, x))
10:         DPLL( F_{x←0} , (A, ¬x))
```

The DPLL algorithm is closely related to a method for inferring new clauses from existing ones, called *Resolution*. This has precisely one inference rule, the *resolution rule*:

$$\frac{A \vee x;\ B \vee \overline{x}}{A \vee B}$$

This rule is sound since any truth value of x cannot make both of the given clauses true, so one of A or B must be made true. A *Resolution refutation* is a sequence of clauses ending in the empty clause $\perp$, each of which is either a given clause or follows from two prior clauses via the resolution rule.

Proposition 9.1. *A given set F of clauses is unsatisfiable iff there is a Resolution refutation of F.*

A Resolution refutation is *tree-like* iff each derived clause is used at most once in any resolution rule.

Proposition 9.2. *The trace of every DPLL algorithm that fails to find a satisfying assignment for a CNF formula F corresponds to a tree-like refutation of F.*

Proof. At each of the leaves of the tree of partial assignments A explored, the empty clause $\perp$ corresponds to an original clause falsified by the corresponding assignment A. The unit propagation nodes each only have one child but we can make each such node a binary node by adding a leaf node for the alternative polarity of the literal involved. That corresponds to falsifying the original clause that was a unit clause under assignment A and therefore that leaf can also be labelled by an original clause that the partial assignment that reaches it falsifies.

We now walk up the tree building the tree-resolution proof. Now consider the last decision literal x at node u above two leaf nodes v where x is set to true and w where x is set to false. Let C_v and C_w be the original clauses that are violated at nodes v and w. The partial assignment A that reaches node

u together with setting x to true. Then A together with x falsifies C_v and A together with $\overline{x}$ falsifies C_w. Since u is not a leaf, A does not falsify either C_v or C_w. Therefore $C_v = D \vee \overline{x}$ and $C_w = D \vee x$, for some clauses D and E where A falsifies D and A falsifies E. We can apply the resolution rule to C_v and C_w to derive the clause $C_u = D \vee E$ which we use to label node u. The assignment A falsifies C_u. Applying this operation inductively up the tree, we can label every node by a clause such that the partial assignment reaching the node falsifies the associated clause and each node is the resolvent of its two child, resolving on the literal being branched on.

At the end we obtain a clause labelling the root of the tree that is falsified by the empty partial assignment. This means that the clause labelling the root must be $\perp$ and the whole thing is a tree-resolution refutation. Observe that the tree-resolution refutation exactly follows the structure of the DPLL execution. □

Note that the above simulation works no matter how the decision literals are chosen. In fact, one can show that for every tree-resolution refutation of F can be pruned and converted into a DPLL execution of the same structure with some choice of decision literals the same size

CDCL SAT solvers

CDCL stands for Conflict-Directed Clause Learning. These are the most important practical algorithms for SAT solving and formal reasoning. In DPLL, when the search fails because of a conflict in line 7, the recursive calls simply backtrack and the last decision is simply undone. However, while the conflict was found using the last branching decision, it may not depend on any other recent branching decision, so changing those decisions may not impact the conflict. The general idea of conflict-directed clause learning, is to record somewhat more about the decisions and unit propagations make during the proof search, using a data structure called a *conflict graph* and replace line 7 of DPLL with a *conflict analysis* step which adds a new *learned clause* to F, which summarizes the reason for the conflict and can be used to simplify future searches:

The algorithm maintains the partial assignment A, which is called the *trail*. The *conflict graph* is a directed graph with one vertex for each literal in the trail A. Decision literals are source nodes. For every literal x in A assigned through propagation of a unit clause that was originally a clause C of F, all of the other literals z in C must have previously appeared as $\neg z$ in A; in the conflict graph we put an edge from each of these $\neg z$ literals to x. It is possible that this unit propagation produces the empty clause $\perp$ rather than a literal, if a decision is made that causes an immediate contradiction.

Definition 9.3. Given a conflict graph $G = (V, E)$, a *source-sink cut* in G is a set of vertices $U \subset V$ such that

- U contains all sources (decision literals) in G,
- U does not contain the sink node labeled $\perp$.
- U there are no edges from $V - U$ to U in G.

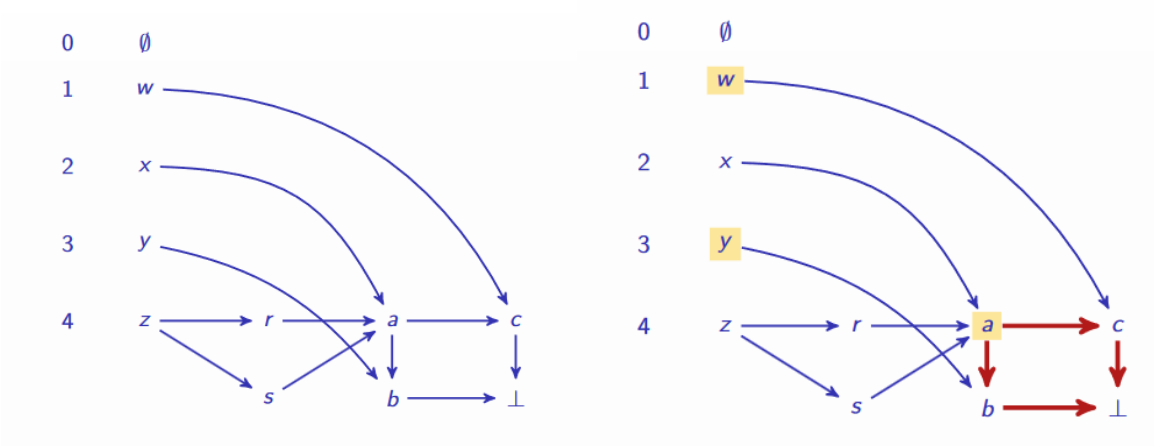
Given such a cut U, let E_U be the set of edges that lead from U to $V - U$. Observe that E_U is a set of edges whose removal eliminates all paths from source nodes to $\perp$. Given such a cut U, we define *conflict clause* C_U to be the clause whose literals are negations of literals at tails of edges in E_U.

CDCL solvers choose one of these clauses that is guaranteed to cause immediate unit propagation at a level higher than the current level. Such a clause is called an *asserting*; the level where this unit propagation takes place is call the *assertion level* of the clause. Note that the clause consisting of the negations of all the decision literals will be asserting at a level one above the current level. If we add that clause, this will be the equivalent to DPLL since unit propagation will switch to the other assignment of the last decision literal.

Algorithm 5 Simplified CDCL ideas for satisfiability search.

```
1:  function SIMPLE-CDCL( F )
2:      Set A to nil and conflict graph to empty.
3:      Set decision level to 0.
4:      while true do
5:          while F contains a clause consistent with A with only one literal x not set by A do
6:              A ← (A, x)
7:              Propagate(x,F)                                   ▷ Unit propagation
8:              Add x and edges to conflict graph
9:          if F has no active clauses then Halt and output satisfying assignment A
10:         if conflict graph has ⊥ then
11:             if decision level=0 then Halt and output "unsatisfiable".
12:             Use conflict graph to find asserting conflict clause C_U   ▷ Analyze Conflict
13:             Add C_U to F
14:             Set decision level to assertion level of C_U, pruning A and the conflict graph.
15:         else
16:             Choose unset literal x according to decision heuristic     ▷ Decision literal
17:             Increment decision level.
18:             A ← (A, x).
19:             Propagate(x,F).
20:             Add source x to conflict graph tagged with decision level.
```

The asserting learned clause is typically found by using the 1UIP (1st Unique Implication Point) cut given by a single node (the first starting from $\bot$) that separates $\bot$ from the decision literal at the last level. Example of a conflict graph and asserting learned clause $(\overline{w} \vee \overline{v} \vee \overline{a})$ given by the 1UIP is the following:



Proposition 9.4. *Learned clauses are derivable using a number of steps of the resolution rule at most the size of the last decision level.*

Proof. We show the idea using the example. The conflict graph implies that clauses $\overline{b} \vee \overline{c}$, $\overline{y} \vee \overline{a} \vee b$, $\overline{w} \vee c$ must be part of the formula. Resolving the 1st with the 2nd gives $\overline{y} \vee \overline{a} \vee \overline{c}$ and resolving this with the 3rd gives the desired learned clause. The general case follows similarly working backwards from $\bot$. □

There are a few optimizations that are critical for the good performance of CDCL. One is that the algorithm only maintains the trail rather than the residual (simplified) formula at each step. All that is required is that the algorithm be able to tell when a clause in the residual formula has at most one unset literal outside of A. Learned clauses can also get quite long, even if the original formula was in 3-CNF. To do this the algorithm maintains the names of just two *watched literals* for each original and learned clause that has not been satisfied by A, as well as a list of which active clauses have these literals. Every time that A sets a literal, each the watched literal pair for each in the list is updated. (If one of these is falsified by the newly assigned literal, then another literal in the clause is chosen to be watched if possible.) This has the advantage of being cache-efficient.

Another optimization is that the algorithm can periodically choose to restart from the root, but keeping around learned clauses so that the algorithm doesn't get stuck at large decision levels; a simple example might be if a clause of size 1 is learned which enforces the assignment to one of the variables, but it can also be useful if there simply have been a large number of decisions.

Good CDCL algorithms also use information about the conflict graph and learned clauses to help with the decision heuristic for new literals is based on which literals have shown up in recent learned clauses, using a tunable parameter. A popular one is called VSIDS, which uses multiplicative weight updates for variables. Depending on the application, decisions about which polarity to use for each literal may be to always try false first, or to use the same sign as was last used, or to randomize the choice.

Also, it turns out that some frontier clauses include more literals than necessary because the negations of some of those literals are implied by the negations of other sets of literals in the clause. Simplifying such clauses is call *clause minimization*.

Finally, the number of learned clauses can grow very quickly, which would overwhelm the storage. To avoid this, CDCL algorithms usually have a cache pruning phase that periodically removes learned clauses that have not been used recently. (Often this is in the form of a marking algorithm that periodically halves the number of learned clauses, removing old learned clauses that have not been used since the last cache pruning.)

Algorithm 6 CDCL for satisfiability search.

```
1:  function CDCL( F )
2:      Choose two watched literals for each clause of F, if possible.
3:      Set A to nil and conflict graph to empty.
4:      Set decision level to 0.
5:      while true do
6:          while F contains a clause with only 1 watched literal x do
7:              A ← (A, x)
8:              Propagate(x,F)                                  ▷ Unit propagation
9:              Add x and edges to conflict graph
10:         if F has no active clauses then Halt and output satisfying assignment A
11:         if conflict graph has bot then
12:             if decision level=0 then Halt and output "unsatisfiable".
13:             Use conflict graph to find asserting conflict clause C_U   ▷ Analyze Conflict
14:             Minimize C_U
15:             Add C_U to F and add two watched literals for C_U
16:             Update decision heuristic
17:             if restart chosen then
18:                 Set A to nil and conflict graph to empty.
19:                 Set decision level to 0.
20:             else if clause pruning chosen then
21:                 Remove oldest half of learned clauses unused since last pruning.
22:             else
23:                 Set decision level to assertion level of C_U, pruning A and the conflict graph.
24:             Update watched literals.
25:         else
26:             Choose unset literal x according to decision heuristic   ▷ Decision literal
27:             Increment decision level.
28:             A ← (A, x).
29:             Propagate(x,F).
30:             Add source x to conflict graph tagged with decision level.
```

Theorem 9.5. *The trace of every CDCL SAT solver run on an unsatisfiable CNF formula F yields a Resolution refutation of F of at most the same size.*

10 Resolution Proofs

Definition 10.1. For a CNF formula F, we write $\mathrm{Res}(F)$ for the minimum number of clauses in any Resolution refutation of F. If F is satisfiable then $\mathrm{Res}(F) = \infty$.

Definition 10.2. For a CNF formula F write $\mathrm{w}(F) = \max_{C \in F} |C|$ and for a resolution proof P, define the *width* of P, $\mathrm{w}(P) = \max_{C \in P} |C|$.

For a CNF formula F define $\mathrm{width}(F)$ to be the minimum width $\mathrm{w}(P)$ over all resolution refutations P of F (and ∞ if no such refutation exists).

Proposition 10.3. *(a) Let P be a resolution derivation of a clause C from CNF formula F. For any restriction ρ on the variables of F, $P|_\rho$ is a resolution derivation of $C|_\rho$ from $F|_\rho$.*

(b) For literal z, if $\mathrm{width}(F_{z \leftarrow 1}) \leq w$ then either $\mathrm{width}(F) \leq w$ or there is a resolution derivation of $\neg z$ from F of width at most $w+1$.

Proof. Part (a) is immediate. For part (b), let P' be a refutation of $F_{z \leftarrow 1}$ of width at most w. For the new derivation, replace each input clause of $F_{z \leftarrow 1}$ by the corresponding clause of F and retain the same sequence of resolution steps (which is possible since none of the resolution steps involve the literal z) to yield a proof P. The replacement of input clauses of $F_{z \leftarrow 1}$ by those of F may add $\neg z$ to some input clauses. It is immediate, inductively, that every clause of proof P' either stays the same or has $\neg z$ added to it in P. If the output clause is still the empty clause $\perp$, then all the clauses of $F_{z \leftarrow 1}$ leading to the output clause of P' were in the original formula F, so P' is the required refutation. Otherwise, the clause width of P is at most $w+1$ and the output clause is $\neg z$. □

Theorem 10.4. *Let F be a CNF formula in n variables. If $\mathrm{Res}(F) \leq S$ then*

$$\mathrm{width}(F) \leq \max(2\sqrt{2n \ln S}, \sqrt{2n \ln S} + \mathrm{w}(F)).$$

Proof. Set $W = \lceil \sqrt{2n \ln S} \rceil$. We say that a clause C is *wide* iff $\mathrm{w}(C) \geq W$. We prove the following claim by induction on n and k:

CLAIM: If $(1 - W/2n)^k \, S < 1$ then any CNF formula F in n variables having a resolution refutation with $\leq S$ wide clauses has $\mathrm{width}(F) \leq \max(W, w(F)) + k$.

Before proving the claim, we observe that the case $k = W$ is sufficient to prove the statement, since $(1 - W/2n)^W < e^{-W^2/2n} \leq 1/S$ by the choice of W.

The case $k = 0$ is trivial, since a refutation with no wide clauses has width at most W.

For the general case, let P be a resolution refutation of F with n variable and $\leq S$ clauses of width $\geq W$ and suppose that $(1 - W/2n)^k \, S \leq 1$. Choose the literal z appearing in the most wide clauses of P. Since there are $\leq 2n$ possible literals and $\geq W$ distinct literals per wide clause, z appears in $\geq WS/2n$ wide clauses.

Consider the restrictions $z \leftarrow 1$ and $z \leftarrow 0$: Then $P_{z \leftarrow 1}$ is a resolution refutation of $F_{z \leftarrow 1}$ and every clause of P containing z is satisfied and hence removed (and others are only shortened), so $P_{z \leftarrow 1}$ has $S' \leq S - WS/2n = (1 - W/2n) \, S$ wide clauses. Therefore $(1 - W/2n)^{k-1} \, S' \leq (1 - W/2n)^k \, S \leq 1$. It follows by our inductive hypothesis with $k' = k - 1$, that

$$\mathrm{width}(F_{z \leftarrow 1}) \leq \max(W, \mathrm{w}(F_{z \leftarrow 1})) + k - 1 \leq \max(W, \mathrm{w}(F)) + k - 1.$$

By Proposition 10.3(b), either $\text{width}(F) \leq \max(W, \text{w}(F)) + k$, and we are done, or there is a derivation P' of $\neg z$ from F of width $\leq \max(W, \text{w}(F)) + k$.

Now, $P_{z \leftarrow 0}$ is a refutation of $F_{z \leftarrow 0}$. By the inductive hypothesis applied to $F_{z \leftarrow 0}$, which has $n' = n-1$ variables, there is a refutation P'' of $F_{z \leftarrow 0}$ of width at most $\max(W, \text{w}(F)) + k$. We next resolve $\neg z$ with clauses of F containing z to produce every clause of $F_{z \leftarrow 0}$ used in P''. This part requires width at most $\text{w}(F)$.

Putting the parts together we obtain a single refutation of F of width at most $\max(W, \text{w}(F)) + k$ as required for the induction step. □

Corollary 10.5. *Let F be a CNF formula in n variables. If there is a resolution refutation of F or size at most S then there is an algorithm running in time $n^{O(\sqrt{n \log S} + w(F))}$ that will find a resolution refutation of F.*

Proof. Apply a width-increasing search on proofs up to the width bound W^* given in Theorem 10.4. That is, it applies all possible resolution inferences that yield clauses of width w beginning with $w = \text{w}(F)$ and increasing until a refutation is found. The number of distinct clauses up to this width bound W^* is less than $\sum_{w=0}^{W^*} \binom{2n}{w}$ and the running time is polynomial in this number of potential clauses. Plugging in the different values of W^* and using standard binomial bounds for the two cases yields the claimed running times. □

We restate Theorem 10.4 in the following convenient form:

Theorem 10.6. *For any CNF formula F in n variables,*

$$\text{Res}(F) \geq e^{\min((\text{width}(F) - w(F))^2/2n, \text{width}(F)^2/8n)}.$$

This implies that sufficiently strong resolution width lower bounds suffice for proving resolution size lower bounds. It is known that the width-size relationship in Theorems 10.4 and 10.6 cannot be improved beyond a logarithmic factor in width or a polynomial factor in size.

Boundary expansion and resolution clause width

Definition 10.7. For a bipartite graph $G = (L, R, E)$ and a set $S \subseteq L$, the *boundary* of S, denoted ∂S, is the set of all $v \in R$ that have exactly one neighbor $u \in S$.

Graph $G = (L, R, E)$ is an (r, c)-*boundary expander* iff for every $S \subset L$ with $|S| \leq r$, the boundary ∂S satisfies $|\partial S| \geq c|S|$.

Definition 10.8. Any CNF formula F corresponds to a bipartite graph $G_F = (L, R, E)$ where L is the set of clauses of F, R is the set of variables of F, and $(C, x) \in E$ iff variable x appears in clause C.

Given a set of clauses S, the boundary of S, ∂S, in G_F is a set of variables, but since each occurs with a unique sign in the clauses of S, we can also interpret ∂S as a set of literals when it is convenient.

Lemma 10.9. *If F is a CNF formula, C^* is a clause and there is a resolution derivation of C^* from F for which S is the set of clauses of F that have a path to C^*, or then $|C^*| \geq |\partial S|$.*

Proof. Observe that the literals in the boundary variables of S pass through to C^* without cancellation. □

Boundary expansion plays a role in a wide variety of lower bound arguments in proof complexity. In particular, it suffices to prove width lower bounds for resolution proofs and hence lower bounds on resolution proof size using Theorem 10.6.

Theorem 10.10. *Any CNF formula F for which G_F is an (r,c)-boundary expander requires resolution refutation width $> cr/2$.*

Proof. Without loss of generality we may assume that $c > 0$. Define the *complexity* of a clause C in the refutation of F to be the size of the subset of clauses S of F that have a path to C in the proof. In particular, since G_F is an (r,c)-boundary expander, any set S of clauses of size at most r has $|\partial S| > c|S| > 0$ by Lemma 10.9 and hence $C \neq \bot$. Therefore, the empty clause at the root of the proof must have complexity $> r$. The input clauses have complexity 1. By the soundness of the resolution rule, complexity is sub-additive; that is, if C is derived from A and B, then the complexity of C is at most the sum of the complexities of A and B.

We therefore follow the proof back from the root, always taking the branch of larger complexity, which will be at least $1/2$ of the previous complexity by sub-additivity. Eventually this must pass through a clause C^* with paths from a minimal subset S of clauses of F with $r/2 < |S| \leq r$ that yields C^*. Since G_F is an (r,c)-bipartite expander, $|C^*| \geq |\partial S| \geq c|S| > cr/2$ as required. □

ETH holds for Resolution proofs

Let $\mathscr{F}_n^{k,m}$ be a distribution of random k-CNF formulas with m clauses chosen uniformly randomly and independently from the set of $2^k\binom{n}{k}$ possible clauses on k distinct variables.

Lemma 10.11. *For $m \geq 2^k \ln 2 \cdot n$, random k-CNF formulas are unsatisfiable with probability $1 - o(1)$.*

Proof. There are 2^n possible assignments. For each such assignment, each clause is independently set true with probability $1 - 2^{-k}$ so the probability that the assignment satisfies the entire formula is then $(1-2^{-k})^m \leq (1-2^{-k})^{2^k \ln 2 \cdot n}$ Since $(1-2^{-k}) < e^{-2^{-k}}$ this is $o(e^{-\ln 2 \cdot n})$ and hence $o(2^{-n})$. Therefore the expected number of assignments that satisfy the formula is $o(1)$ which upper bounds the probability that the formula is satisfiable. □

Lemma 10.12. *Let $\Delta > 0$ and $m = \Delta n$. For some $1 > c, c' > 0$, with probability $1 - o(n)$ a random 3-CNF formula with $m = cn$ is an (r,c)-boundary expander for $r = c'n$ with probability $1 - o(1)$.*

Proof. Fix a set of clauses $\mathscr{C}$ of size s. We want to argue that $|\partial \mathscr{C}| \geq c'n$ for some c'. Any variable appearing in $\mathscr{C}$ that is not in boundary must appear at least 2 times among the $3s$ literals in $\mathscr{C}$, so it suffices to show that every subset $\mathscr{C}$ of size $s \leq r = c'n$ contains more than $q = (3+c)s/2$ distinct literals for some constant $c > 0$.

For a single $C \in \mathscr{C}$, Let $p = \binom{q}{3}/\binom{n}{3} \leq (q/n)^3$ be the probability that all variables of C are some fixed

subset Q of size q and there are $\binom{n}{q}$ such subsets. Now

$$\begin{aligned}
\Pr[|vars(\mathscr{C}| \le q] &\le \binom{n}{q} p^s \\
&\le \left(\frac{ne}{q}\right)^q \left(\frac{q}{n}\right)^{3s} \\
&= e^q \cdot \left(\frac{q}{n}\right)^{3s-q} \\
&= e^{(3+c)s/2} \cdot \left(\frac{(3+c)s/2}{n}\right)^{(3-c)s/2} \\
&= a^s \cdot \left(\frac{s}{n}\right)^{(3-c)s/2}
\end{aligned}$$

for some constant a depending on c. Therefore, the probability that G_F is not an (r,c)-boundary expander is at most

$$\begin{aligned}
\sum_{s=1}^{r} \binom{\Delta n}{s} a^s \cdot \left(\frac{s}{n}\right)^{(3-c)s/2} &\le \sum_{s=1}^{r} \left(\frac{a \cdot e \cdot \Delta n}{s}\right)^s \cdot \left(\frac{s}{n}\right)^{(3-c)s/2} \\
&= \sum_{s=1}^{r} \left[a \cdot e \cdot \Delta \cdot \left(\frac{s}{n}\right)^{(1-c)/2}\right]^s \\
&= \sum_{s=1}^{r} \left[b \cdot \Delta \cdot \left(\frac{s}{n}\right)^{(1-c)/2}\right]^s
\end{aligned}$$

for $b = a \cdot e$ which depends only on c.

To bound this quantity we split the sum into two cases depending on whether s is small or large.

We can choose constant $c' > 0$, depending on Δ and c so that the term in the sum for $s \le r = c'n$ is at most 2^{-s}. We use this for any threshold $t = t(n)$ that grows with n to bound the sum of all of the terms for $s > t(n)$ by $2^{-t(n)}$, which goes to 0 with n. For definiteness we just choose $t(n) = \log n$.

On the other hand, there are only $\log n$ terms for $s \le t(n)$ and each is $O((\frac{\log n}{n})^{(1-c)/2})$ since $s \ge 1$, so that sum also is $o(1)$ in n. □

Theorem 10.13. *There is a constant $\delta > 0$ such that random 3-CNF formulas F with $O(n)$ clauses with probability $1 - o(1)$, require* $\mathrm{Res}(F) \ge 2^{\delta n}$.

Proof. By Lemma 10.12, there are constants $c, c' > 0$ such that with probability $1 - o(1)$, G_F is an (r,c)=boundary expander for $r = c'n$ and hence, by Theorem 10.10, $\mathrm{width}(F) > cr/2 = c'cn/2$. Plugging this width lower bound into Theorem 10.6 yields the claimed bound with $\delta = (c'c)^2/32$. □

11 Communication Complexity and Lifting

Definition 11.1. A *(deterministic) decision tree* T with inputs in $\{0,1\}^n$ is a rooted binary tree with each internal node labeled by a variable x_i for $i \in [n]$, with the two out-edges labelled 0 and 1 indicating the value of x_i. Each root-leaf path in T defines a partial assignment that is the concatenation of the assignments on all edges in the path. The output of the decision tree on a truth assignment x is the label of the leaf reached by the unique root-leaf path whose associated partial assignment is consistent with x.

For a function f defined on $\{0,1\}^n$, we write the decision tree complexity of f, $C^{dt}(f)$ to be the minimum height of any decision tree computing f and $\text{size}^{dt}(f)$ to be the minimum number of leaves in any such decision tree,

Definition 11.2 (False Clause Search)**.** Given an unsatisfiable CNF formula F in n variables with clauses $C_1, \ldots, C_m$, we define the search problem $Search_F$ which takes as input $x \in \{0,1\}^n$, which must falsify F, and outputs any index $i \in [m]$ such that x falsifies clause C_j.

$Search_F$ is an example of a relation R. In general we write $R(x)$ for the set of legal outputs of R on input x. For any relation R, write $C^{dt}(R)$ and $\text{size}^{dt}(R)$, for the minimum $C^{dt}(f)$ and $\text{size}^{dt}(f)$ for any function f that is consistent with R.

Proposition 11.3. *DPLL run on an unsatisfiable input F is a decision tree for the search problem $Search_F$. In particular $\text{size}^{dt}(Search_F)$ is the minimum number of nodes in any DPLL run on input F (equivalently the minimum size of any tree-resolution refutation of F) and $C^{dt}(Search_F)$ is the minimum height of the DPLL search tree on input F.*

(Two-Party) Communication Complexity

Here were have two *players* or *parties*, usually designated as Alice and Bob who cooperate in order to compute based on shared inputs. Alice receives an input $x \in X$ and Bob receives $y \in Y$.

A *(deterministic) 2-party communication protocol* on $X \times Y$ is a rooted binary tree, with each internal node v labelled either by a function $\alpha_v : X \to \{0,1\}$, or a function $\beta_v : Y \to \{0,1\}$. The two out-edges of v are labelled 0 and 1 respectively. The root-leaf paths in a protocol followed on input x and y is given by following the out-edges given by $\alpha_v(x)$ (Alice speaks) or $\beta_v(y)$ (Bob speaks) depending on which kind of function labels v. The output of the protocol on input $(x,y) \in X \times Y$ is the label of the leaf.

For a function f defined on $X \times Y$, we write the deterministic communication complexity of f, $C^{Cc}(f)$ to be the minimum height of any 2-party communication protocol computing f and $\text{size}^{cc}(f)$ to be the minimum number of nodes in any deterministic 2-party protocol computing f. We extend these definitions to relations defined on $X \times Y$ also.

Examples: Consider $X = Y = \{0,1\}^n$. Then for any f on $X \times Y$ has $C^{cc}(f) \leq n+1$ since Alice can simply send her entire input to Bob after which he computes the answer. $Parity_n(x,y)$ that gives the total parity check for the string xy, has $C^{cc}(Parity_n) = 2$. $Maj_n(x,y)$ which has value 1 if there are at least as many 1's as 0's in xy has $C^{cc}(Maj_n) = O(\log n)$.

What about $EQ_n(x,y)$ which is 1 if x and y are equal and 0 otherwise?

In general given a function f defined on $X \times Y$, we define the *communication matrix of f*, M_f as the $|X| \times |Y|$ matrix whose (x,y) entry is $f(x,y)$.

In particular M_{EQ_n} is the $2^n \times 2^n$ identity matrix.

Definition 11.4. A *(combinatorial) rectangle* in $X \times Y$ is a set of the form $A \times B$ for $A \subseteq X$ and $B \subseteq Y$. A rectangle R is 1-rectangle of f iff f evaluates to 1 on every element of R, and a 0-rectangle of f iff f is always 0 on R. R is *monochromatic* iff f is constant on R.

Lemma 11.5. *The set of inputs in $X \times Y$ that reach each node v of a communication protocol is a combinatorial rectangle.*

Proof. We show this by induction starting at the root r which is rectangle $R_r = X \times Y$. Suppose inductively that $R_u = A_u \times B_u$ is the rectangle of inputs reaching node v in the protocol tree and let the child following outedge 0 be v_0 and the one following out-edge 1 be v_1.

Case Alice: If v is labelled by some $\alpha_v : X \to \{0,1\}$, then the set of nodes reaching v_0 is precisely $A_{v0} \times B_{v0}$ where $A_{v0} = A_v \cap \alpha_v^{-1}(0)$ and $B_{v0} = B_v$ and reaching v_1 is $A_{v1} \times B_{v1}$ where $A_{v1} = A_v \cap \alpha_v^{-1}(1)$ and $B_{v1} = B_v$.

Case Bob: If v is labelled by some $\beta_v : X \to \{0,1\}$, then the set of nodes reaching v_0 is precisely $A_{v0} \times B_{v0}$ where $A_{v0} = A_v$ and $B_{v0} = B_v \cap \beta_v^{-1}(0)$ and reaching v_1 is $A_{v1} \times B_{v1}$ where $A_{v1} = A_v$ and $B_{v1} = B_v \cap \beta_v^{-1}(1)$. □

Corollary 11.6. *Let f be defined on $X \times Y$. Then $X \times Y$ can be partitioned into at most $2^{C^{cc}(f)}$ monochromatic rectangles of f, corresponding to the leaves of a protocol with this complexity.*

Corollary 11.7. $C^{cc}(EQ_n) = n + 1$

Proof. M_{EQ_n} is a $2^n \times 2^n$ diagonal matrix and no two entries on the diagonal can be in the same rectangle. This gives 2^n 1-rectangles, plus there is at least one 0-rectangle giving more than 2^n such rectangles. Taking logarithms and rounding up we obtain the lower bound; the upper bound is true for all n-bit Boolean functions. □

Other important problems of interest in communication complexity include

- $Disj_n(x,y) = \bigvee_{i=1}^n x_i \wedge y_i$, called the disjointness function. The terminology comes from viewing x and y as characteristic functions of two subsets $S_x, T_y \subseteq [n]$, since $Disj_n(x,y) = 0$ iff the sets S_x and S_y are disjoint.
- $IP_n(x,y) = \sum_{i=1}^n x_i y_i \bmod 2$, the inner product function mod 2.
- $Index_m : [m] \times \{0,1\}^m$ where $Index_m(x,y) = y_x$.

Proposition 11.8. $C^{cc}(Disj_n) = C^{cc}(IP_n) = n + 1$ *and* $C^{cc}(Index_m) = m$.

Karchmer-Wigderson Games

We now connect communication complexity of search problems to the complexity of Boolean circuits over the De Morgan basis of binary $\wedge, \vee$ and $\neg$ gates. By De Morgan's law we can push all of the negations in a circuit or formula to the leaves so that we can assume that the inputs for a circuit are literals and all interior gates are binary $\wedge$ or $\vee$. We write $C(f)$ for the minimum circuit size for Boolean function f in that basis, $D(f)$ for the minimum circuit depth for f and $L(f)$ for the minimum number of leaves in any Boolean formula for f.

We also will consider *monotone* circuits which do not use any negated literals. Monotone circuits can only compute monotone Boolean functions, which are functions for which flipping any input bit from 0 to 1 cannot decrease the function value. For monotone f, we write $C^m(f)$, $D^m(f)$, and $L^m(f)$ for the monotone circuit complexity, depth complexity, and formula size of f.

Definition 11.9. Given a Boolean function $f : \{0,1\}^n \to \{0,1\}$ we define the *Karchmer-Wigderson relation* $KW_f : X \times Y$ to $[n]$ where $X = f^{-1}(1)$ and $Y = f^{-1}(0)$ be the relation

$$KW_f(x,y) = \{i \mid x_i \neq y_i\}.$$

If f is monotone then we know that for any input x, y for which $f(x) = 1$ and $f(y) = 0$, there must be some bit $x_i = 1$ and $y_i = 0$. We can therefore define the *monotone Karchmer-Wigderson relation* $mKW_f : X \times Y$ to $[n]$ where

$$mKW_f(x,y) = \{i \mid x_i > y_i\}.$$

Theorem 11.10. *For every Boolean function f, $C^{cc}(KW_f) = D(f)$ and $size^{cc}(KW_f) = L(f)$. For every monotone Boolean function f, $C^{cc}(mKW_f) = D^m(f)$ and $size^{cc}(f) = L^m(f)$.*

Proof. Without loss of generality the optimal circuit depth is given by a Boolean formula (a tree), since we can simply replicate gates. There are two directions here:

Given a Boolean formula T for f, the protocol tree for KW_f will have exactly the same structure as T. For each gate of the circuit v, we write f_v for the Boolean function computed at the node.

We maintain the invariant that the rectangle of inputs R_u that reaches node u is contained in $f_u^{-1}(1) \times f_u^{=1}(0)$. To do this, every $\vee$ gate becomes a node for Alice and every $\wedge$ gate becomes a node for Bob. If gate u is an $\vee$ gate, where $f_u = f_v \vee f_w$, since any string y reaching this node has $f_u(y) = 0$, we know that $f_v(y) = f(w) = 0$. On the other hand any string x reaching it has $f_u(x) = 1$ so at least one of $f_v(x) = 1$ or $f_w(x) = 1$. We define the function $\alpha_u(x)$ is 0 if $f_v(x) - 1$ and 1 otherwise.

The situation if the gate if $\wedge$ is dual: Alice's input x evaluates to 1 on both children, while Bob has to indicate one of v or w such that $f_u(y) = 0$.

Applying this at a leaf we reach a literal or its negation. We label the leaf by the index i of that literal, which must be correct for KW_f since x and y evaluate differently on that literal. If the circuit is monotone we must have $x_i = 1$ and $y_i = 0$ as required for mKW_f.

Hence the complexities of the Karchmer-Wigderson relations are at most that of the corresponding depth and size measures.

We now do the reverse simulation to show the other direction. We convert the protocol trees for KW_f or mKW_f to Boolean formulas with exactly the same structure as the protocol trees, replacing every node where Alice speaks by $\vee$ and every node where Bob speaks by $\wedge$. We work bottom-up arguing that we can maintain the property that the function f_u satisfies $R_u \subseteq f_u^{-1}(1) \times f_u^{-1}(0)$ where R_u is the rectangle of inputs reaching node u.

To get started we need to specify the leaf labels. At the leaves of the protocol for KW_f with output i, every input (x, y) in the rectangle R_ℓ associated with that leaf ℓ must have x_i and y_i must have opposite values. The values of both must be unique since this is a rectangle. If $x_i = 1$ and $y_i = 0$ in the rectangle then the circuit has the positive literal i at ℓ and if $x_i = 0$ and $y_i = 1$ then we have the negation of literal i at ℓ. This ensures that the property holds for f_ℓ.

Suppose that we have the property for children v and w of node u at which Alice speaks; then $f_u = f_v \vee f_w$ by construction. Suppose that $R_u = A_u \times B_u$. Then $R_v = A_v \times B_u$ and $R_w = A_w \times B_u$ with $A_u = A_v \cup A_w$ where $f_v(A_v) = 1$ and $f_w(A_w) = 1$ by the inductive property so therefore $f_u(A_u) = 1$. Since both $f_v(B_u) = 0$ and $f_w(B_u) = 0$, we have $f_u(B_u) = 0$. The same applies dually to the nodes where Bob speaks. The fact that this correctly computes f follows from the inductive hypothesis applied to the root. □

12 Deterministic Lifting

Definition 12.1. Given a Boolean function f or search problem (relation) R defined on $\{0,1\}^n$, and a Boolean function $g : X \times Y \to \{0,1\}$ (called the *gadget* function). We define the composition $f \circ g^n$ or $R \circ g^n$ on the set $X^n \times Y^n$ by $f \circ g^n(x,y) = f(z_1,\ldots,z_n)$ where $z_i = g(x_i,y_i)$ for $(x_i,y_i) \in X \times Y$.

Note that in the composed function $f \circ g^n$, Alice receives all of $x_1,\ldots,x_n$ and Bob receives all of $y_1,\ldots,y_n$.

Proposition 12.2. *Let $f : \{0,1\}^n \to \{0,1\}$. Then*

$$C^{cc}(f \circ g^n) \leq C^{dt}(f) \cdot C^{cc}(g).$$

More generally if R is a search problem defined on $\{0,1\}$ then

$$C^{cc}(R \circ g^n) \leq C^{dt}(R) \cdot C^{cc}(g).$$

Proof. The communication protocols are simple: The two players simulate the best decision tree protocol for f (or R) on input $z_1,\ldots,z_n$; whenever that protocol queries z_i, the players compute it using the best communication protocol for g using $C^{cc}(f)$ bits of communication. □

Key lifting question: Find broad circumstances in which the above algorithm is (close to) optimal; that is, when $C^{cc}(f \circ g^n)$ is $\Omega(C^{dt}(f))$ or $\Omega(C^{dt}(f)\, C^{cc}(g))$.

There are simple examples of f and g where the protocol of Proposition 12.2 is very far from optimal: Suppose that both functions are parity functions; that is, $f = \oplus_n$ and $g(x,y) = Parity_m(x,y)$ on $\{0,1\}^m \times \{0,1\}^m$. In that case, $C^{dt}(f) = n$ and $C^{cc}(g) = 2$ so the upper bound from Proposition 12.2 would be $2n$. However, the function $f \circ g^n$ is simply $Parity_{nm}$ so $C^{cc}(f \circ g^n) = 2$.

Our focus will be on general theorems showing that for some function g, Proposition 12.2 holds for *all* functions f and *all* relations R. We clearly cannot do this if g is $Parity_m$.

The first such general lifting theorem was proved by Raz and McKenzie was proven in order to derive lower bounds on the depth complexity of monotone circuits by showing lower bounds on monotone Karchmer-Wigderson game mKW of the composed function $f \circ g^n$. It therefore was important that the gadget g be monotone.

Theorem 12.3 (Raz-McKenzie 1999)**.** *For every search problem R defined on $\{0,1\}^n$ and m that is a sufficiently large polynomial in n we have*

$$C^{cc}(R \circ Index_m^n) = \Omega(C^{dt}(R)\ \log m).$$

Recall that $Index_m : [m] \times \{0,1\}^m$ is given by $g(x,y) = y_x$ so it is a monotone function and hence yields a monotone function for every monotone function f. It has $C^{cc}(Index_m) = \log m + 1$ so it shows that Proposition 12.2 is asymptotically tight for $Index_m$.

Before proving this theorem we discuss one of its applications.

Raz and McKenzie used Theorem 12.3 together with an explicit sequence F_n of unsatisfiable 4-CNF formulas (based on pebbling pyramid graphs) that was known to require large depth Resolution refutations and defined a monotone Boolean function based on $f = F_n \circ Index_m^n$ such that the relation mKW_f is precisely $Search_{F_n} \circ Index_m^n$ and used the above theorem to derive new depth lower bounds on monotone circuits.

Given an unsatisfiable k-CNF formula F_n with n variables, for the 2-party communication problem $Search_{F_n} \circ Index_m^n$, Alice gets $x_1, \ldots, x_n \in [m]^n$, Bob gets $y_1, \ldots, y_n \in \{0,1\}^n$ and the goal is to output the index i of some clause of F_n that evaluates to false on the input $Index_m(x_1, y_1), \ldots, Index_m(x_n, y_n)) = (y_{1x_1}, \ldots, y_{nx_n})$. In other words, this just a protocol for $Search_{F_n}$ with each of whose input bits being determined by the vector of pointers $x_1, \ldots, x_n$.

A general form of the Raz-McKenzie construction of a monotone Boolean function was given by Göös and Pitassi as follows:

Lemma 12.4 (Göös-Pitassi 2012)**.** *Let F_n be an unsatisfiable k-CNF formula in n variables with t clauses and let $m \geq 2$ be an integer. There is an explicit monotone Boolean function $f_{F_n,m}$ with $N = tm^k$ input bits such that the communication complexity of $Search_F \circ Index_m^n$ is at most that of $mKW_{f_{F_n,m}}$.*

Proof. The function $f_{F_n,m}$ takes as an input a string α that is thought of as describing an k-CSP (constraint satisfaction problem with constraint size k) on the input space $[m]^{[n]}$ using the variable structure of the formula F_n. Each clause C_ℓ for $\ell \in [t]$ involves some subset S_ℓ of indices in $[n]$. Fix such an ℓ. The string α will have one bit for each way of choosing the k elements in $[m]^{S_\ell}$ indicating the truth table of the constraint indexed by ℓ in the k-CSP.

Sometimes the k-CSP defined by α will be satisfiable and sometimes it is not. We define $f_{F_n,m}(\alpha) = 1$ iff the k-CSP defined by α is satisfiable. Clearly changing 0's to 1's in α can only make the k-CSP defined by α more satisfiable so $f_{F_n,m}$ is a monotone function of α.

In order to simulate an algorithm for $Search_{F_n} \circ Index_m^n$, on input assignment $(x, y) \in [m]^n \times (\{0,1\}^m)^n$ with $x = (x_1, \ldots, x_n) \in [m]^n$ and $y = (y_1, \ldots, y_n) \in (\{0,1\}^m)^n$, Alice creates the string α_x corresponding to the k-CSP whose ℓ-th constraint is 1 iff for every $i \in S_\ell$, the input is consistent with x_i. The k-CSP given by α_x is satisfied by x so $f_{F_n,m}(\alpha_x) = 1$. On the other hand, Bob creates the string α_y corresponding to the unsatisfiable k-CSP whose ℓ-th constraint evaluates to 1 iff for the vector $x'_{S_\ell} \in [m]^{S_\ell}$, the partial assignment $(y_{ix'_i})_{i \in S_\ell}$ satisfies C_ℓ. Since F_n is unsatisfiable, for every $x' \in [m]^n$, the formula on F_n with inputs $y_{1x'_1}, \ldots, y_{nx'_n}$ is not satisfied by y and hence the k-CSP in given by α_y is unsatisfiable so $f_{F_n,m}(\alpha_y) = 0$.

The $mKW_{f_{F_n,m}}$ protocol for input pair (α_x, α_y) produces some index $(\ell, x'_{S_\ell}) \in [t] \times [m]^k$ such that α_x has value 1 and α_y has value 0. Since α_x has value 1, by definition $x'_{S_\ell} = x_{S_\ell}$. However, since α_y has value 0, by definition of α_y, the partial assignment $(y_{ix_i})_{i \in S_\ell}$ does not satisfy C_ℓ which means that ℓ is a correct output for $Search_{F_n} \circ Index_m^n$. □

Corollary 12.5. *If F_n is an unsatisfiable family of k-CNF formulas requiring Resolution refutation depth D then for $m = n^c$ for sufficiently large c, $f_{F_n,m}$ requires monotone circuit depth $\Omega(D \log n)$.*

Proof. The assumption implies that $C^{dt}(Search_{F_n}) \geq D$. By Theorem 12.3, $C^{cc}(Search_{F_n} \circ Index_m^n)$ is $\Omega(D \log n)$. By Lemma 12.4, this implies that $C^{cc}(mKW_{f_{F_n,m}})$ is $\Omega(D \log n)$ which implies that $f_{F_n,m}$, requires monotone circuit depth $\Omega(D \log n)$. □

Note that when k is constant and we can prove some Resolution depth lower bound of the form $D = n^\epsilon$ for $\epsilon > 0$ then by choosing some $m = n^c$ we obtain a lower bound of the form N^δ for some $\delta > 0$ on the explicit monotone Boolean function f defined by this construction.

Göös, Pitassi, and Watson improved the lower bound in Theorem 12.3 using a refined version of the argument to show that one can take m as small as $n^2 \log n$. One can find explicit 3-CNF Boolean formulas in n-variables and a linear number of clauses that require Resolution depth $\Omega(n/\log n)$ but this still yields a monotone function f with a fixed polynomial blow-up in n and so the lower bound is only of the form n^δ for some small $\delta < 1$.

A Deterministic Lifting Theorem We first observe that $Index_m$ is a universal gadget.

Proposition 12.6. *Any function $g : [m] \times [m] \to \{0,1\}$ is a subfunction of $Index_m$.*

Proof. On input $(x, y) \in [m] \times [m]$, define $y' \in \{0,1\}^m$ by $y'_i = g(i, y)$. Then by definition $Index_m(x, y') = y'_x = g(x, y)$. □

Therefore, any lifting theorem involving such a gadget g that has communication complexity $\Omega(m)$ yields a lifting theorem for $Index_m$.

Rather than proving the version of the lifting theorem given by Raz-McKenzie and Göös-Pitassi-Watson using $Index_m$, we prove the following lifting theorem using the Inner Product function IP_d for $d = O(\log n)$ which was proved independently by Chattopadhyay, Koucký, Loff, and Mukhopadhyay and by Wu, Yao, and Yuen. The description of the argument is based on a version due to Rao and Yehudayoff, but the key ideas are the same as all of the other proofs.

Theorem 12.7. *Let $f : \{0,1\}^n \to \{0,1\}$ for $n \geq 16$ and $d \geq 7 \log n$, then*

$$C^{cc}(f \circ IP_d^n) = \Omega(C^{dt}(f)\, d).$$

Similarly, if R is a relation on $\{0,1\}^n$ then

$$C^{cc}(R \circ IP_d^n) = \Omega(C^{dt}(R)\, d).$$

By the universality of $Index_m$, this implies Theorem 12.3 for $m \geq n^7$. The method of proof of Theorem 12.7 is constructive. In particular, we show how to take any communication protocol P computing $f \circ IP_d^n$ with C bits of communication and produces a decision tree T for f of height at most $20C/d$.

In particular, the protocol P operates on inputs $x, y \in (\{0,1\}^d)^n$, while the decision tree T built from P operates on inputs $z \in \{0,1\}^n$. T is built from P by simulation.

The general idea is that we begin the simulation at the root of both the protocol P and the as-yet-unbuilt tree T. At each node v in P and the corresponding nodes v' of T we maintain sets $A = X_{v,v'} \subseteq X$ and $B = Y_{v,v'} \subseteq Y$ such that the rectangle $A \times B$ of inputs consistent with the current simulation. We follow a path in P until the protocol P has "learned too much" about the portion of x and y in some coordinate $j \in [n]$. At that point we label the current node v' in T by a query to z_j and create two child nodes in T, v'_0 for the answer $z_= 0$ and v'_1 for the answer $z_i = 1$. For $b \in \{0,1\}$ and each node v'_b we find some fixed pair of strings $(x_j^b, y_j^b) \in IP_d^{-1}(b)$ and define $X_{v,v'_b} = X_{v,v'}|_{x_j = x_j^b}$ and $Y_{v,v'_b} = Y_{v,v'_b}|_{y_j = y_j^b}$ and continue the simulation. The goal is to show that $\Omega(d)$ steps of the protocol P are required per coordinate queried.

The simulation keeps track of a set $S \subseteq [n]$ of unqueried coordinates. We maintain the invariant that $|A| = |A_S|$ and $|B| = |B_S|$. The simulation depends on the sizes of the projections $A_{S'}$ and $B_{S'}$ of A and B on subsets $S' \subseteq S$ of unqueried coordinates. In particular, the simulation focuses on the sets of cases $S' = S \setminus \{j\}$ which we write as $S - j$ for simplicity.

We say that A is *j-abundant* iff $|A_S|/|A_{S-j}| \geq 2^{6d/7}$ and *j-sparse* otherwise. B is *j-abundant* iff $|B_S|/|B_{S-j}| \geq 2^{6d/7}$ and *j-sparse* otherwise. For $a \in A_{S-j}$ we say that a is *thin* iff a has $< 2^{4d/7}$ extensions in A_S. Similarly for $b \in B_{S-j}$ we say that b is *thin* iff b has $< 2^{4d/7}$ extensions in B_S. We say that A and B are *thick* iff there are no thin a or thin b for either A or B.

Note that A being j-abundant is about the average number of extensions for elements $a \in A_{S-j}$, whereas being thick bounds the worst-case number of number of extensions for all $a \in A_{S-j}$.

For a coordinate $j \in S$, and a value $b \in \{0,1\}$ we that a pair of values (x_j^b, y_j^b) is b-good iff $IP_d(x_j^b, y_j^b) = b$ and the set $A^b \times B^b$ consisting of the elements $(x,y) \in A \times B$ such that $x_j = x_j^b$ and $y_j = y_j^b$. satisfies $|A^b_{S-j} \times B^b_{S-j}| \geq |A_{S-j} \times B_{S-j}|/2$.

Algorithm 7 Deterministic simulation for IP lifting.

1: Initialize:
2: $S \leftarrow [n]$
3: $A, B \leftarrow (\{0,1\}^d)^n$
4: $v \leftarrow$ root of P
5: Create T be a tree with one root node v'
6: **procedure** BUILD-TREE(A, B, S, v, v')
7: **while** $S \neq \emptyset$ and v is not a leaf of P **do**
8: $R_v = (A_v, B_v) \leftarrow$ rectangle at node v of protocol P.
9: **if** for every $j \in S$, A and B are both j-abundant **then**
10: Let v_0 and v_1 be the children of v in P. ▷ Simulate another step of P.
11: Choose $b \in \{0,1\}$ for which $|(A \times B) \cap R_{v_b}| \geq |A \times B|/2$.
12: $v \leftarrow v_b$
13: $A \leftarrow A \cap A_{v_b}$ and $B \leftarrow B \cap B_{v_b}$
14: **else if** there is some $j \in S$ and thin $a \in A_{S-j}$ or thin $b \in B_{S-j}$ **then**
15: **while** there is a thin a for A or thin b for B **do** ▷ Prune the sets A and B
16: Remove all elements of A that extend a.
17: Remove all elements of B that extend b.
18: **else** ▷ Both A and B are thick but at least one is j-sparse
19: Create a query to z_j at node v', adding children v'_0 and v'_1 for query answers 0 and 1.
20: Let (x_j^0, y_j^0) be 0-good values in $(A \times B)_j$ such that $IP_d(x_j^0, y_j^0) = 0$. ▷ Requires proof
21: $A^0 \leftarrow A$ with j-th coordinate fixed to x_j^0.
22: $B^0 \leftarrow B$ with j-th coordinate fixed to y_j^0.
23: BUILD-TREE$(A^0, B^0, S - j, v, v'_0)$.
24: Let (x_j^1, y_j^1) be 1-good values in $(A \times B)_j$ such that $IP_d(x_j^1, y_j^1) = 1$. ▷ Requires proof
25: $A^1 \leftarrow A$ with j-th coordinate fixed to x_j^1.
26: $B^1 \leftarrow B$ with j-th coordinate fixed to y_j^1.
27: BUILD-TREE$(A^1, B^1, S - j, v, v'_1)$.
28: Make v' a leaf labelled by the label of leaf v (or the value of $f(z)$ or some element of $R(z)$ if $S = \emptyset$).

Analyzing the simulation Observe that when the procedure BUILD-TREE is called initially, both A and B are thick, every $a \in A_{S-j}$ and $b \in B_{S-j}$ has 2^d extensions in A_S and B_S respectively and $A_S/A_{S-j} = B_S/B_{S-j} = 2^d$. Therefore, A and B are both j-abundant and thick. This will cause the procedure to do simulation according to steps 10-13. At each step either A and B will decrease by a factor of at most 2 and this will change both the abundance and thickness properties by at most a factor of 2 per bit of communication. Therefore, it will take $\Omega(d)$ steps of communication in order to create a thin a or b or to make A or B j-sparse. Therefore it takes $\Omega(d)$ communication steps before the first query.

We need to keep versions of this property in order to allocate the cost of each query to $\Omega(d)$ bits of communication. Moreover, in the simulation in addition to ensuring this allocation that the values (x_j^0, y_j^0) and (x_j^1, y_j^1) in steps 20 and 24 both exist and have good properties assuming that both A and B are thick and either A or B is j-sparse. In particular, we will insist

Lemma 12.8. *Let A and B be the sets before a prune step and A' and B' be the sets after the prune step. Then $|A_S \times B_S| \geq (7/8)^2 |A'_S \times B'_S|$.*

Proof. Observe that the only way that a prune step occurs in the simulation is if the preceding step was a simulation step because prune steps don't follow each other and query steps only reduce $A \times B$ based on coordinates j that are removed from S and so don't contribute to the creation of thin a or b. Let A^- and B^- be the sets prior to that preceding simulation step. Then $|A_S^-|/|A_{S-j}^-| \geq 2^{6d/7}$ and $|B_S^-|/|B_{S-j}^-| \geq 2^{6d/7}$ for all $j \in S$. The simulation step implies that either $|A| \geq |A^-|/2$, $B = B^-$ or $A = A^-$ and $|B| \geq |B^-|/2$. In particular, $|A_S|/|A_{S-j}| \geq 2^{6d/7}/2$ and $|B_S|/|B_{S-j}| \geq 2^{6d/7}/2$. Rewriting, we obtain that $|A_{S-j}| \leq 2|A_S|/2^{6d/7}$ and $|B_{S-j}| \leq 2|B_S|/2^{6d/7}$. Each thin a (respectively b) defined on $S - j$ results in the removal of fewer than $2^{4d/7}$ elements of A (respectively B). Therefore the total number of elements removed from A is less than

$$\sum_{j \in S} |A_{S-j}| 2^{4d/7} \leq \sum_{j \in S} 2|A_S|/2^{2d/7} \leq 2n|A_S|/2^{2d/7} = 2n|A_S|/n^2 = 2|A_S|/n \leq |A_S|/8$$

since $d \geq 7 \log n$ and $n \geq 16$. Similarly we have the total number of elements removed from B being at most $|B_S|/8$. In particular, this implies that $|A'_S| \geq 7|A_S|/8$ and $|B'_S| \geq 7|B_S|/8$ which yields the claimed property. □

Corollary 12.9. *Assume that elements in steps 20 and 24 can always be found. For every step of the* BUILD-TREE *procedure, if A, B and S are the values at the beginning of the step and A', B' and S' are the values at the end of the step, then $|A'_{S'} \times B'_{S'}| \geq |A_{S'} \times B_{S'}|/2$.*

Proof. In the simulate step, the value of S does not change and the property holds by construction in Step 11. In the prune step the set S also does not change, and the property holds by Lemma 12.8, Finally, in each of the branches of the query step, the condition of the values found in Steps 20 and 24 being 0-good and 1-good respectively ensure that the property holds on both branches. □

Lemma 12.10. *Assuming that Steps 20 and 24 always are achievable the depth of the decision tree T is at most $20C(P)/d$.*

Proof. The measure of progress at each step will be based on

$$\Phi = \frac{|A_S \times B_S|}{2^{2|S|d}} \leq 1.$$

At the start, $\Phi = 1$. Observe that if A is j-sparse then $|A_{S-j}| > |A_S|/2^{6d/7}$ and we always have that $|A_{S-j}| \geq |A_S|/2^d$; analogous properties hold for B. Therefore after a query step to z_j, the new value of

Φ is

$$\begin{aligned}
\frac{|A^b_{S-j} \times B^b_{S-j}|}{2^{2(|S|-1)d}} &\geq \frac{|A_{S-j} \times B_{S-j}|}{2^{2(|S|-1)d+1}} && \text{since the restrictions on coordinate } j \text{ are } b\text{-good} \\
&= \frac{|A_{S-j}| \cdot |B_{S-j}|}{2^{2(|S|-1)d+1}} \\
&> \frac{|A_S| \cdot |B_S|}{2^{6d/7} \cdot 2^d \cdot 2^{2(|S|-1)d+1}} && \text{since one of } A \text{ or } B \text{ is } j\text{-sparse} \\
&= 2^{d/7-1} \frac{|A_S \times B_S|}{2^{2|S|d}}.
\end{aligned}$$

Therefore, each query step increases the progress measure Φ by a factor of at least $2^{d/7-1}$. Note that since $d = 7 \log n$ and $n \geq 16$, we have $d/7 - 1 \geq d/7 - d/28 = 3d/28$. There are at most $C(P)$ simulate steps and at most $C(P)$ prune steps in the course of the algorithm. Each simulate step or prune step leaves the set S unchanged and therefore reduces Φ by at most a factor of 2. Therefore there must be at most $2C(P)/(d/7-1)$ query steps in total which is at most $56C/(3d) < 20C/d$ query steps on any root-leaf path. □

Clearly, by construction every input in the set $A \times B$ always is consistent with the partial assignment to the variables z_i for $i \in [n] \setminus S$ and therefore the construction yields a correct decision tree for f.

It merely remains to prove that 0-good and 1-good inputs can be found in Steps 20 and 24. This relies on a certain *disperser property* of the inner product function IP_d.