Iddo Tzameret

Introduction to Concrete Complexity

Lecture Notes

Imperial College London
Dept. of Computing

February 16, 2025

Contents

0.1 Summary of Module

We are interested in approaches to the fundamental hardness questions in computational complexity.

Computational complexity: the study of which problems can be efficiently computed and which cannot.

Efficiency: we understand efficiency as Polynomial Time computability. A Boolean function $f : \{0,1\}^n \to \{0,1\}$ is said to be efficiently computable if there is a polynomial-time Turing Machine M that, on input x in $\{0,1\}^n$ outputs $f(x)$ and runs in time n^c for some constant c. Note that n^c is a polynomial in the input length n (the exponent c does not depend on the input length n).

The arguably main question of the theory of computation, the one that subsumes in some sense many problems in other parts of computing, once is the P vs NP question: Can we separate P from NP, namely, is there a language in NP that is not in P? In other words, can we prove that SAT (Boolean satisfiability problem) cannot be solved in polynomial time? And yet again, roughly, can problems whose solutions once given can be verified efficiently, can be solved efficiently?

We are interested in *concrete* approaches, namely, considering a simple, usually combinatorial-looking, model of computation, such as a Boolean circuit, and establishing lower bounds against the size of circuits required to prove certain specific functions that are given to us concretely (usually, these functions also possess some straightforward combinatorial properties; e.g., they represented specific graph problems). In this sense, the question is concrete because the result is unconditional (namely, it does not depend on unproved assumptions, such as $\mathsf{P} \neq \mathsf{NP}$), and the model itself is concrete: it is a (primarily combinatorial) object of which its size we lower bound in precise terms (e.g., circuit C computing function $f(x)$ must have size $2^{|x|}$, where $|x|$ is the bit-size of the input x).

Three main concrete approaches to the fundamental hardness questions are the following:

1. Circuit Complexity
2. Proof Complexity
3. Algebraic Complexity

We shall see a bit from each, mainly circuit complexity and some basic proof complexity while commenting briefly on algebraic complexity.

Other approaches to the fundamental hardness questions are usually more intrinsic to complexity theory. In that respect, the whole of computational complexity theory could be viewed as “approaching" the fundamental hardness questions through complexity class, reductions, concrete lower bounds and the relation between these notions and results. One intriguing approach that makes this attempt in particular is the “Meta Complexity" approach. We are not going to touch on this in this course.

Chapter 1
Introduction to Circuit Complexity

1.1 Basic Circuit Complexity

Definition 1.1 (Boolean circuit) Let $n \in \mathbb{N}$ and $x_1, \ldots, x_n$ be n variables. A Boolean Circuit C with n inputs is a directed acyclic graph. It contains n nodes with no incoming edges, called the *input nodes* and a single node with no outgoing edges, called the *output node*. All other nodes are *internal nodes* or *gates*, and are labelled by the logical gates $\vee$, $\wedge$, $\neg$ (i.e., logical OR, AND, NOT, resp.). The $\vee$, $\wedge$ nodes have fan-in (i.e., number of incoming nodes) 2, and $\neg$ has fan-in 1. The *size of* C, denoted $|C|$, is the number of nodes in the underlying graph. C is called a *formula* if each node has at most one outgoing edge (i.e., the underlying graph is a tree),

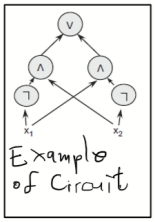
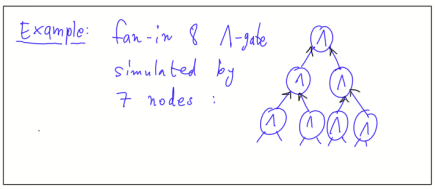


Fig. 1.1 Example of a simple Boolean circuit.

Comment

Fan-in $d > 2$ can be simulated by a tree of $d - 1$ nodes:



Question 1:

What is the function computed by the circuit below?

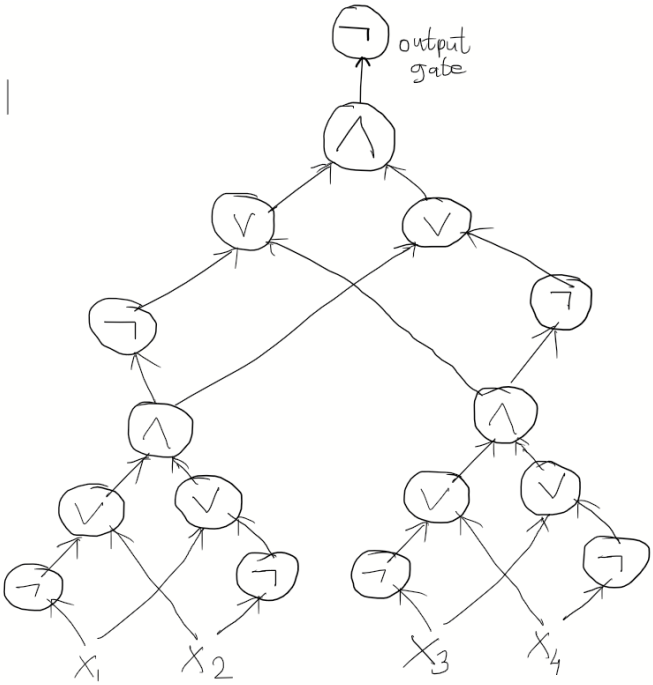


Fig. 1.2 Question

See the answer at the end of the chapter (page 10).

Definition 1.2 Let $T : \mathbb{N} \to \mathbb{N}$ be a function. A $T(n)$-***sized circuit family*** is a sequence $\{C_n\}_{n\in\mathbb{N}}$ of Boolean circuits, where C_n has n input-gates (i.e., n inputs-bits) and a single output-bit such that $|C_n| \leq T(n), \forall n \in \mathbb{N}$.

Slice of function[a]. Let $f : \{0,1\}^* \to \{0,1\}$ be a Boolean function. We can consider the *slice of size n* of f to be the function $f_n : \{0,1\}^n \to \{0,1\}$, which is f restricted to *inputs of length precisely n*.

[a] Not to be confused with *The Slice Function*.

1.2 Basic concepts: circuit families, language recognition, and function computation

Let $\mathbb{N}$ denote the set of natural numbers starting from 1.

A *language* is a set of (finite) strings. Note that the language can contain infinite many strings, only that each string is finite. A *string* is an ordered sequence of symbols from a fixed constant size alphabet. We shall use mainly strings over the alphabet consisting of two symbols $\{0,1\}$. Hence, a language is simply a set $L \subseteq \{0,1\}^*$ (recall, that $\{0,1\}^*$ is the set of all finite 0-1 strings, including the empty string).

Remark 1.1 We can also define a circuit-family where some input lengths $n \in \mathbb{N}$ are *skipped* in the sequence, e.g., the length of every input should be even. We shall not use this subtlety here.

A language $L \subseteq \{0,1\}^*$ is said to be *in* $\mathsf{SIZE}(T(n))$, namely,

$$L \in \mathsf{SIZE}(T(n)),$$

if there is a $T(n)$-size circuit family $\{C_n\}_{n\in\mathbb{N}}$ such that

$$\forall n \in \mathbb{N}\ \forall x \in \{0,1\}^n : x \in L \Leftrightarrow C_n(x) = 1.$$

In this case, we say that the family $\{C_n\}_{n\in\mathbb{N}}$ **decides** the language L.

Similarly, for a Boolean (single-output) *function* $f : \{0,1\}^* \to \{0,1\}$, $f \in \mathsf{SIZE}(T(n))$ if there exists $T(n)$-size circuit family $\{C_n\}_{n\in\mathbb{N}}$ such that $\forall n \in \mathbb{N} \quad \forall x \in \{0,1\}^n$, $f(x) = 1 \Leftrightarrow C_n(x) = 1$. In this case, we say that the family $\{C_n\}_{n\in\mathbb{N}}$ **computes** the function f.

Similarly, we can consider $\{f_n\}_{n\in\mathbb{N}}$ to be the *family* of Boolean functions $f_n : \{0,1\}^n \to \{0,1\}$, for all n at least 1 (so, this is a family of all slices of f). Hence, the family of functions f_n, for each input length n, denoted $\{f_n\}_{n\in\mathbb{N}}$, is simply another way of writing the single function $f : \{0,1\}^* \to \{0,1\}$.

1.3 Quick recap: growth functions, O-notation and run-times

O-Notation

We usually use n for the length of inputs. Namely, if the input is $x \in \{0,1\}^*$, then we use n to denote $|x|$, i.e., $n = |x|$. In complexity we have to distinguish between a *constant* and a *growing number*. We wish to consider how a given function behaves asymptotically, namely, when its input length n grows to infinity: while n grows, we shall say that c is *a constant* if c is independent of n (namely, while n grows, c is fixed).

Recall the following notation for "asymptotic less than":

1. $f(n) = O(g(n))$ if there exist a positive constant c (independent of n) and $n_0 \in \mathbb{N}$, such that $\forall n \geq n_0 \quad f(n) \leq cg(n)$ (for two functions $f, g : \mathbb{N} \to \mathbb{N}$).
2. $f(n) = o(g(n))$ if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$. That is, there is *no* positive constant c and no $n_0 \in \mathbb{N}$ such that $f(n) \geq cg(n)$, for all $n \geq n_0$.

And here are the dual notation for "asymptotic greater than" symbols:

3. $g(n) = \Omega(f(n))$, if there exist a positive constant c and $n_0 \in \mathbb{N}$, such that $\forall n \geq n_0$, $g(n) \geq cf(n)$. Hence, $g(n) = \Omega(f(n))$ iff $f(n) = O(g(n))$.
4. $g(n) = \omega(f(n))$ if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$. That is, there is *no* positive constant c and no $n_0 \in \mathbb{N}$ such that $f(n) \geq cg(n)$, for all $n \geq n_0$. Hence, $g(n) = \omega(f(n))$ iff $f(n) = o(g(n))$.

Basic Growth Rates

1. When $f = O(\log n)$ the base b of $\log_b n$ is immaterial (why?).
2. *Polynomial growth rate* means $n^{O(1)} = 2^{O(\log n)} = n^c$, for some constant c, namely c is independent of n.
3. $2^{O(n)}$ means $\leq 2^{cn}$ for some constant c independent of n.
4. *Exponential growth rate* (similarly, *exponential bound*) means for us 2^{n^δ}, for a real constant $\delta > 0$. (Some texts insist that "exponential growth" should only refer to 2^{cn}, for a constant $c > 0$. Note the difference between this and our definition!)

Time Complexity

We mainly consider the worst-case time for a problem to be solved by a given TM.

Definition 1.3 (Running time of Turing Machines) Let M be a TM that halts on every input. Let $f : \mathbb{N} \to \mathbb{N}$ be a function, s.t., $f(n)$ is the max number of steps it takes M to halt on inputs of length n. We say f is the running time of M. And that M runs in time $f(n)$.

Be sure to recall the following basic concepts (though we are not going to use them concretely in this course; see e.g., [2]).

Time bounds for Turing Machines (TMs): counts the maximal number of steps a specific Turing Machine M is doing when input with a string of size n.

The class P: Polynomial time computation (also denoted PTIME, or polytime, or p-time).

The class NP*, verifiers, short certificates*: Nondeterministic Polynomial-time computation. Namely, the class of languages that can be decided by a nondeterministic Turing Machine in time bounded by a polynomial in the input length. That is, if the input is of size n the running time of the nondeterministic Turing machine should not exceed n^c, for some constant n independent of n (e.g., $c = 40$).

SAT: the Boolean Satisfiability problem: given a propositional formula determine if there exists a boolean (i.e., 0-1) assignment that satisfies the formula.

SAT *is* NP*-complete (Cook-Levin theorem)*: See e.g., [2, 3].

Definition 1.4 (polynomial-size circuits; P/poly) The class P/poly is the class of languages that are decidable by polynomial-size circuit families. That is, $\mathsf{P/poly} = \bigcup_{c \in \mathbb{N}} \mathsf{SIZE}(n^c)$.

Theorem 1.1 (Small circuits simulate polytime Turing Machines) *Every language determined by a (deterministic) polynomial-time Turing Machine can be computed by a polynomial-size circuit family, and in symbols:* $\mathsf{P} \subseteq \mathsf{P/poly}$.

Proof Assignment/tutorial. [See for a sketch in Arora-Barak'10, Sec. 6.1.1. page 110; [2]]. □

1.4 Our main focus: lower bounds

Lower bounds, hard functions. We say that we have a *super-polynomial lower bound against* P/poly, namely a super-polynomial lower bound against (polynomial-size) Boolean circuits, if the following occurs: there exists a Boolean function $f : \{0,1\}^* \to \{0,1\}$ such that no polynomial-size circuit family $\{C_n\}_n$ computes f. In other words, f is not in P/poly. If we show that $f \notin \mathsf{SIZE}(T(n))$, we say that we have *proven a $T(n)$-lower bound for f* (against Boolean circuits). We say that this f is *hard* for $\mathsf{SIZE}(T(n))$.

Discussion: Motivation for Circuit Complexity

1. Non-uniformity: Notice that for every fixed input length n_0 in a circuit family $\{C_n\}_n$, the circuit C_{n_0} can behave very differently from the other circuits in the family. Namely, each circuit C_n may compute correctly the nth slice f_n of a Boolean function $f : \{0,1\}^* \to \{0,1\}$, while for each different n the circuit looks very different from the other circuits. This means that the family is *non-uniform*, which formally means that there is no Turing Machine that given an input size n as its input, outputs a description of the circuit C_n (for that input size).
   Therefore, we may want to discover whether there is a non-uniform circuit family that solves SAT. This still would not mean that there is a polytime Turing Machine solving SAT. In other words, perhaps for each input length n, C_n is a circuit of quadratic size $O(n^2)$ that solves SAT?
2. The notion of a Boolean circuit is mathematically "*cleaner*" than Turing Machines; so we might have better hope to prove lower bounds against this model, instead of directly against Turing Machines run-time.
3. Note the following curious *open* problem: $\mathsf{NEXP} \stackrel{?}{\subseteq} \mathsf{P/poly}$.
   Notice that by the *time hierarchy theorem* $\mathsf{EXP} \not\subseteq \mathsf{P}$ (that is, exponential time EXP is properly a bigger class than P). Hence, clearly, $\mathsf{NEXP} \not\subseteq \mathsf{P}$. But when we consider the non-uniform version of P, namely, $\mathsf{P/poly}$, we already do not know if every language in NEXP is also in $\mathsf{P/poly}$. (We do know that there are languages in $\mathsf{P/poly}$ that are not in NEXP, because $\mathsf{P/poly}$ is not a uniform class, hence even undecidable languages are in $\mathsf{P/poly}$, but not in the uniform class NEXP.)

1.4.1 Most functions are hard: Shannon lower bound

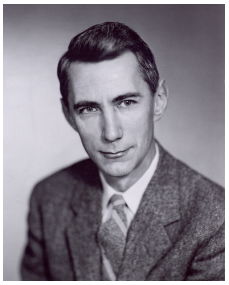


Fig. 1.3 C.E. Shannon. Source: By Unknown author - Tekniska Museet, CC BY 2.0, https://commons.wikimedia.org/w/index.php?curid=144716444

Theorem 1.2 (Shannon lower bound) *For every $n > 1$ there exists a function $f : \{0,1\}^n \to \{0,1\}$ that cannot be computed by a circuit of size $\leq 2^n/10n$.*

To prove this theorem, we are going to use a *counting argument*.

Counting argument. The counting argument at its core is a use of the *pigeonhole principle*, which states that we cannot place n pigeons in $n-1$ pigeonholes, given that each hole can fit at most a single pigeon. Or dually, that we cannot cover n holes with $n-1$ pigeons.

Proof We are going to count the number of distinct possible circuits of size at most $2^n/10n$, and conclude that this number is smaller than the number of possible distinct Boolean functions $f : \{0,1\}^n \to \{0,1\}$. Hence, we cannot cover all possible Boolean functions with n variables with the set of circuits of size at most $2^n/10n$. Namely, there is a Boolean function that cannot be computed by any circuit of size at most $2^n/10n$.

Claim: We can encode a boolean circuit of size S with at most $5 \cdot S \log S$ bits.

Proof: Using adjacency list to encode a circuit of size at most S. Note that since S is the upper bound of the size of the circuit, we can label the gates of the circuit by numbers from $\{0, 1, 2, \ldots, S-1\}$. Thus, every gate i can be encoded by $\lceil \log S \rceil$ bits. Each entry i in the list contains at most two numbers $j, k < i \leq S-1$ designating that the gates j, k have both an incoming edge into the gate i (if the gate i is "$\neg$" it has fan-in one, and we need only one number). This amounts to $\leq 2 \cdot \lceil \log S \rceil$ bits. We also need to encode the gate type $(\vee, \wedge, \neg)$ and for this we need only two extra bits. So all in all, each cell in the list uses only $2 + 2 \cdot \lceil \log S \rceil$ many bits.

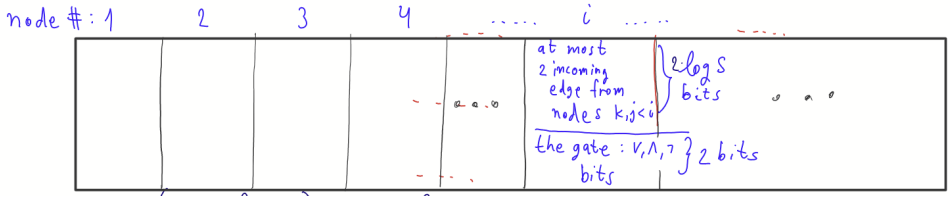


Fig. 1.4 The encoding scheme for a Boolean circuit of size S.

We have:

$$S \cdot (2 + 2\lceil \log S \rceil) = 2S + 2S\lceil \log S \rceil \leq 4S\lceil \log S \rceil \leq 5S \log S.$$

$\square_{\text{claim}}$

We now show that the number of circuits of size $\leq 2^n/10n$ is smaller than the number of possible Boolean functions with n inputs. Thus, there exists a Boolean function that cannot be computed by a circuit of size $\leq 2^n/10n$.

The number of circuits of size S is bounded from above by the number of distinct codes of circuits of size S, which is

$$\begin{aligned}
&\leq 2^{5 \cdot S \log S} \\
&= 2^{\left(5 \cdot \frac{2^n}{10n} \cdot \log\left(\frac{2^n}{10n}\right)\right)} = 2^{\left(5 \cdot \frac{2^n}{10n} \cdot (n - \log 10n)\right)} \\
&\leq 2^{\frac{5n}{10n} \cdot 2^n} = 2^{2^{n-1}}.
\end{aligned}$$

But this is less than 2^{2^n} the number of distinct Boolean functions $f : \{0,1\}^n \rightarrow \{0,1\}$. □

Note: The reason our size lower bound is $2^n/10n$, instead for example a stronger lower bound of $2^n/c$, for some constant c, is that encoding a circuit is *super linear* (i.e., $\Omega(n \cdot \log n)$). Note indeed that using smaller codes for circuits amounts to stronger (i.e., bigger) lower bounds: if we denote by $code(S)$ the maximal length needed to encode circuits of size S, then $2^{code(S)}$ is the number of different codes, which bounds the number of different functions computed by size S circuits. In the proof we need to show that $2^{code(S)} < 2^{2^n}$. Thus, as $code(S)$ gets smaller as a function of S, namely the encoding of circuits is more efficient, we get a better lower bound: we get that the inequality $2^{code(S)} < 2^{2^n}$ holds for *bigger* S.

1.4.2 Answers

Answer to Question 1

PARITY on 4 input bits. It outputs 1 iff the number of 1's in the input is odd. I.e., it computes the XOR of $x_1, \ldots, x_4$. In other words, it computes the function $x_1 + x_2 + x_3 + x_4 \mod 2$, where $x_i \in \{0,1\}$, for $i \in [4]$. (Understand why that is. Hint: recall that $\neg A \vee B$ is logically equivalent to $A \rightarrow B$. So $\neg(A \rightarrow B) \wedge \neg(B \rightarrow A)$ computes the XOR of A, B. Now, notice that this structure repeats throughout the circuit.)

Chapter 2
Monotone Circuit Lower Bounds

We have seen that proving that SAT is not in P/poly, i.e., cannot be solved by polynomial-size circuits, implies that $\mathsf{P} \neq \mathsf{NP}$. Due to the notorious difficulty of this and related questions, we are also interested in proving *weaker* lower bounds, namely, lower bounds against *restricted* classes of circuits. Although this does not settle the main lower bound questions, it is still considered an important step towards the bigger questions, at least from the methodological perspective. Here, we study such a restricted circuit class and prove a lower bound against it: Boolean circuits without negation gates, which are also called *monotone circuits*.

Monotone Circuits

Definition 2.1 (Monotone circuit) A *monotone circuit* is a Boolean circuit that contains fan-in two gates AND and OR, but has *no* NOT gates.

Note in particular that monotone circuits can compute only monotone functions: a Boolean function is said to be monotone if *increasing the number of ones* in the input cannot flip the value of the function from 1 to 0. More precisely, for $\bar{x}, \bar{y} \in \{0,1\}^n$, write $\bar{x} \geq \bar{y}$ iff $\forall i \in [n], x_i \geq y_i$, where $[n]$ denotes $\{1, \ldots, n\}$. (Here, $x_i \geq y_i$ for Boolean x_i, y_i means simply that $1 \geq 0$ and $0 \geq 0$, $1 \geq 1$, while $0 \not\geq 1$.)

Definition 2.2 (Monotone function) A Boolean function $f : \{0,1\}^n \to \{0,1\}$ is said to be *monotone* if $\forall \bar{x} \geq \bar{y}$, $f(\bar{x}) \geq f(y)$.

2.1 The CLIQUE Problem

Many NP-complete problems are monotone. One example of an NP-complete decision problem is CLIQUE we describe now. Given an undirected graph $G = (V, E)$ with n nodes, a *k-clique* in G is a set $U \subseteq V$ of size k, such that every pair of nodes $u_1, u_2 \in U$ is connected by an edge (in E), and in symbols:

$$\forall u_1 \in U \, \forall u_2 \in U \, (u_1 \neq u_2 \Rightarrow (u_1, u_2) \in E).$$

Recall that a computational decision problem is a language over a finite alphabet (usually $\{0, 1\}$). Here, our language consists of all the strings that encode (in some natural way) an accepted graph, i.e., a graph with n nodes that contains a k-clique. The natural way to encode a graph in our case is this: a graph $G = (V, E)$ with n nodes, is encoded by $\binom{n}{2}$ input variables x_{ij}, where the semantic of the encoding is: $x_{ij} = 1$ iff $(i, j) \in E$. In other words, if the input variable $x_{ij} = 1$, our input graph contains the edge (i, j), and otherwise it does not.

We are interested in CLIQUE(k, n) for a *fixed* k, considered as the following Boolean function:

The computational problem **CLIQUE**(k, n):
Input: Undirected graph $G = (V, E)$ with n nodes encoded as a length $\binom{n}{2}$ binary string (each bit represents an edge x_{ij}, $i < j \in [n]$), and a number k (given in unary, i.e., 1^k).
Accept: if the graph G contains a k-clique.
Reject: otherwise.

It is known that CLIQUE(k, n) is NP-complete (see standard complexity textbooks; e.g., Papadimitriou 1994). Note that CLIQUE(k, n) is a monotone function: if we add 1's to the input, we only *increase* the chance it has a k-clique. Since CLIQUE(n, k) is a monotone (Boolean) function we can compute it by a monotone Boolean circuit. But the question remains whether we can compute CLIQUE(n, k) with small monotone circuit.

Example of a monotone circuit computing CLIQUE(n, k)

Consider all $\binom{n}{k}$ k-sub-graphs in G, and check if at least one of those is a clique:

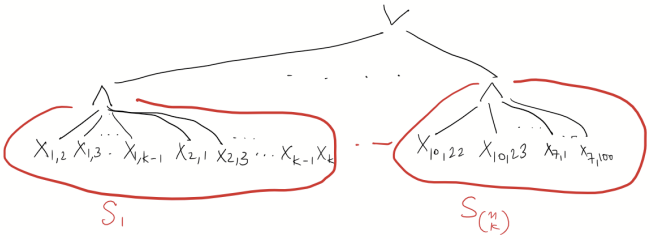


Fig. 2.1 Naive way to compute CLIQUE

$S_1, S_2, \ldots, S_{\binom{n}{k}}$ are the $\binom{n}{k}$ subgraphs in G each of size k. Size of this circuit: $O\left(k^2 \cdot \binom{n}{k}\right)$.

2.1.1 Approximators

A circuit for computing CLIQUE(n, k), consisting of an OR gate $\bigvee$ of many S_i's, each of size k and where each S_i is an AND of the edge variables in S_i, as in the example above, is called an *approximator* for CLIQUE(n, k). More formally, we have the following.

Approximators $\mathrm{APX}(X_1, \ldots, X_m)$ **for CLIQUE**. Let V be a set of nodes and let $X_1, \ldots, X_n \in V$ be a collection of subsets of nodes. The *approximator* $\mathrm{APX}(X_1, \ldots, X_m)$ is defined to be $\bigvee_{r=1}^{m} \bigwedge_{i<j \in X_r} x_{ij}$, with x_{ij} the Boolean variables representing that there is an edge between nodes i and j. Note that the X_i's may have different sizes (and specifically their sizes may be different from k).

Note that $\mathrm{APX}(X_1, \ldots, X_m)(\alpha) = 1$, for $\alpha \in \{0,1\}^{\binom{n}{2}}$, precisely when in the graph G over the nodes V described by the assignment α, at least one of the X_i subgraphs is a clique. The idea behind the approximator is that it provides a good approximation for the CLIQUE function, in the sense that, depending on the number of sets m and size of each set X_i, for many input graphs it provides the correct answer.

We shall also use the abbreviated notation $\mathrm{APX}(\mathcal{X})$ for $\mathrm{APX}(X_1, \ldots, X_m)$ when $\mathcal{X} = \{X_1, \ldots, X_m\}$.

Example of simple useful asymptotic computations

Note that when $k \in (\Omega(\log n), O(n - \log n))$, $\mathrm{APX}\left(S_1, \ldots, S_{\binom{n}{k}}\right)$, where $S_1, \ldots, S_{\binom{n}{k}}$ are all possible subsets of size k from the set of n nodes V, is of *super-polynomial size*, because $\binom{n}{k}$ is super polynomial for k in this range. For the sake of getting used to asymptotic estimates such as these, it is helpful to go over the computation in more detail, as follows.

We shall estimate the asymptotic behaviour of $\binom{n}{\log n}$, showing it is superpolynomial, namely, $\binom{n}{\log n} = n^{\Omega(\log n)}$. First, we observe that

$$\binom{n}{k} \geq \left(\frac{n}{k}\right)^k.$$

To prove this lower bound we do the following. Write $\binom{n}{k} = \frac{n!}{k!(n-k)!} = \frac{\prod_{j=0}^{k-1}(n-j)}{k!} = \prod_{j=0}^{k-1} \frac{n-j}{k-j}$. Notice that each factor $\frac{n-j}{k-j}$ in this product is at least $\frac{n}{k}$, and there are k factors, giving the lower bound $\left(\frac{n}{k}\right)^k$.

Now, to lower bound $\binom{n}{\log n}$ we have:

$$\begin{aligned}
\binom{n}{\log n} \geq \left(\frac{n}{\log n}\right)^{\log n} &= \left(\frac{2^{\log n}}{2^{\log(\log n)}}\right)^{\log n} \\
&= 2^{(\log n - \log\log n)\cdot \log n} \\
&= 2^{\log^2 n - \log n \cdot \log\log n} \\
&= 2^{\log^2 n - o(\log^2 n)} \\
&= 2^{c\log^2 n}, \quad \text{for some constant } c, \text{ say, } c = 1/2 \\
&= \left(2^{\log n}\right)^{c\cdot \log n} = n^{c\log n},
\end{aligned}$$

which means that $\binom{n}{\log n} = n^{\Omega(\log n)}$.

2.2 Clique is hard for monotone circuits

The following theorem shows the naive (brute-force) way of computing the CLIQUE function with a monotone circuit shown in Figure 2.1 cannot be improved much.

Fig. 2.2 Alexander Razborov. Creator: Kozloff, Robert | Credit: Photo by Robert Kozloff. Copyright: The University of Chicago

Theorem 2.1 (Razborov [4]; cf. [1]) *Let $k = \sqrt[4]{n}$. Then, every monotone circuit computing* CLIQUE(n, k) *has size* $2^{\Omega(\sqrt[8]{n})}$.

That is, there exists a constant c such that for large enough $n \in \mathbb{N}$, if C_n computes CLIQUE(n, k) then $|C_n| \geq 2^{c\cdot\sqrt[8]{n}}$.

The rest of this chapter aims to prove this theorem. *Our exposition is taken mostly from Papadimitriou's textbook* [3].

2.3 Proof of monotone circuit lower bounds Theorem 2.1 using the Approximation Method

Approximation Method Overview

We first provide an overview of the approach we take to prove Theorem 2.1 which is called *the approximation method*. We shall describe a way of approximating any **monotone** circuit for CLIQUE(n, k) by a approximator, namely a big OR of cliques, as follows.

Approximation Method: The upshot of the method is the following: *by way of contradiction,* consider a purported small monotone circuit computing CLIQUE(n, k), and show by induction, gradually progressing from the input nodes towards the output node, that the function computed at every gate can be well approximated by a small approximator. Then, if the number of nodes in the circuit is small the output gate is also well approximated by a small approximator. Now, use an auxiliary argument to show that there is no small approximator approximating well CLIQUE(n, k).

More precisely, we have:

1. Given a monotone circuit C, we shall construct a approximator APX($X_1, \ldots, X_m$) for some m and $|X_i| \leq l$ (for some l, and for $i = 1, \ldots, m$), that approximates CLIQUE(n, k) with **precision** that is dependent on the number of gates in C.
2. That is, if the number of gates in C is small the precision is good, namely the approximator APX($X_1, \ldots, X_m$) for CLIQUE(n, k) we end up with makes *few* mistakes on the CLIQUE(n, k) function (a mistake happens when the circuit answers "NO" on an input that has a k-clique, or "YES" on an input that has no k-clique).
   The **approximation** (i.e., construction of an approximator for CLIQUE(n, k) given the circuit C) will proceed in steps, one step for each gate of the monotone circuit:
   a. If C is a monotone circuit computing CLIQUE(n, k) we can *approximate* any gate OR or AND in C with an approximator.
   b. Each such approximation step introduces rather few errors (false positives and false negatives).
3. We show that every approximator APX($X_1, \ldots, X_m$) ($|X_i| \leq l$ for some l, for all $i = 1, \ldots, m$), ought to make *exponentially many errors* on the function CLIQUE(n, k). From 2 above we conclude that there exists no circuit C with a small number of gates.

Parameters & notation

Recall we want to compute CLIQUE(n, k) with n the number of nodes in the graph and k the size of a clique within the graph. From now on we set:

$$\begin{aligned} k &= \sqrt[4]{n} \\ l &= \sqrt[8]{n} \\ p &\approx \sqrt[8]{n} \\ M = (p-1)^l \cdot l! \quad &\approx (\sqrt[8]{n} - 1)^{\sqrt[8]{n}} \cdot (\sqrt[8]{n})! \\ &\leq (\sqrt[4]{n})^{\sqrt[8]{n}}. \end{aligned}$$

Each (hypothetical) approximator we use in the approximation is:

$$\mathrm{APX}\,(X_1, \ldots, X_m)$$

for $m \leq M$ and $|X_i| \leq l, \forall i \in [m]$.

2.3.1 The Input Set: Extreme Graphs, False Positive and Negatives

Here we consider the input graphs we are going to analyse. The input set to the CLIQUE(n, k) function are *all possible* binary strings of length $\binom{n}{2}$, namely, all possible (encodings) of undirected graphs with n nodes. This set is easily divided into two groups:

Accept-instances: $G = (V, E)$ is a graph on n modes that contains a k-clique.
Reject-instances: $G = (V, E)$ is a graph on n nodes that does not contain a k-clique.

A crucial point is that *we are going to restrict attention only to a special* ***subset*** *of all possible input graphs*. We call these special input graphs *extreme graphs*.

Extreme Graphs

We restrict attention to only a subsets of accept and reject instances, which we call *extreme graphs*. This is sufficient for the proof (and simplifies it). Because extreme graphs are part of the input set, a correct circuit clearly must answer correctly on this subset. So, if we show that there must be mistakes made by small circuits on this subset of input graphs, it is immediate that there is no small monotone circuit that correctly computes CLIQUE(n, k).

Thus, our focus is on "extreme" cases of inputs:

Positive-inputs: $G = (V, E)$ has n nodes and a k-clique (i.e., a set of k nodes with all edges between them); while *no* other edge exist in G except for the k-clique. There are $\binom{x}{k}$ positive-inputs, each one is clearly an accept input to CLIQUE(n, k). For simplicity, we shall call these positive inputs k-***cliques*** (although these are

special extreme cases of k-cliques, since we assume that all edges outside the clique are absent.)

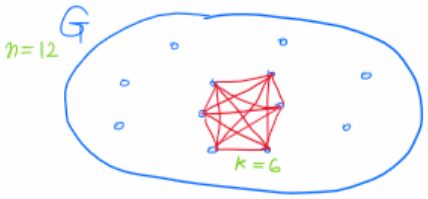


Fig. 2.3 Example of a *positive input*: a graph whose only edges are part of a (single) k-clique.

Negative-inputs: $G = (V, E)$ has n nodes and a $(k-1)$-colouring. I.e., $k-1$ independent sets (each with its own distinct colour), namely sets of nodes, such that in each set there are no edges between nodes. Moreover, *all* edges between nodes in different independent sets are present! For simplicity, we shall call these negative inputs $(k-1)$-***colouring*** (although these are special extreme cases of $k-1$-colourable graphs, since we assume that all edges between independent sets are present).

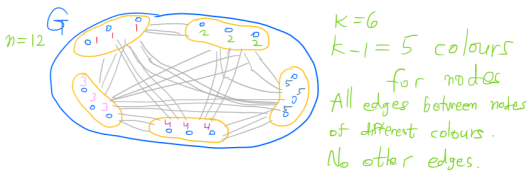


Fig. 2.4 Example of a *negative input* (i.e., $k-1$-colouring): a graph whose all and only edges are between distinct $(k-1)$-independent sets.

Note 2.1

1. There are $(k-1)^n$ negative -inputs. (We count twice two identical graphs with colours interchanged.
2. A $(k-1)$-colouring is a negative-input for CLIQUE(n, k) because a k-clique cannot be coloured by $k-1$ colours.
3. Assuming each independent set has at least one node (namely, there exist $k-1$ nodes with different colours) then even a *single edge* added to a $(k-1)$-colouring will make the graph contain a k-clique!

False Positive and False Negative Inputs

Let V be a set of n nodes, and APX($\mathcal{X}$) be an approximator, with $\mathcal{X}$ a collection $X_1 \ldots, X_m \subseteq V$ of subsets of nodes. We say that a graph $G(V, E)$ (with n nodes) is a **false positive** if APX($\mathcal{X}$)(G) = 1 while G is a $(k-1)$-colouring (and hence, not a k-clique). Similarly, a graph G with n nodes V is said to be a **false negative** if APX($\mathcal{X}$)(G) = 0 while G *is* a k-clique.

2.3.2 Concluding the Lower Bound using Structural Induction

What does it mean that an approximator APX($\mathcal{X}$) *approximates* the monotone circuit C? It means that "on most" extreme input graphs G, $C(G)$ = APX($\mathcal{X}$)(G), namely the approximator agrees with the actual circuit on most G's.

The approximation of the monotone circuit C that computes CLIQUE(n, k) is constructed by *induction on the size of C,* i.e., number of $\vee$, $\wedge$ gates in C.

Note 2.2 Such an induction is usually called ***induction on the structure of the circuit*** C (or *structural induction*), since we are going to induce on subcircuits of C, assuming the induction hypothesis holds for them. Using the structure of the circuit, we shall conclude that the induction hypothesis holds for the gate that connects those two subcircuits. The base case is showing that the induction statement holds for circuits of size 1 (i.e., singled variable circuits). The induction step is showing that the induction statement holds for $A \vee B$ assuming the induction statement holds separately for both A and B. And similarly for the case of $A \wedge B$.

Now, assume that C is a monotone circuit we wish to build an approximation for. We are going to build the approximator for the output gate C gradually, from the input gates of C through the internal gates, up to the output gate, so that the approximator of the output gate will be the approximator of the circuit C. The skeleton of this construction is done as follows, following a standard structural induction scheme:

Approximation of the output gate of C via structural induction

Recall that APX($\mathcal{X}$) stands for an approximator with the parameters described above $l = \sqrt[8]{n}, p \approx \sqrt[8]{n}, M = (p-1)^l \cdot l!$. For every gate g in C we are going to define approx(g) to be APX($\mathcal{X}$) for some sequence $\mathcal{X}$ of sets of nodes, i.e., an OR of cliques $\mathcal{X}$. The main challenge would be to define precisely which sets $\mathcal{X}$ are used for each gate approximator, which will be done in the sequel.

Base Approximation Step: Input nodes x_{ij}. Define the approximator of x_{ij}, denoted to be precisely x_{ij}, that is approx(x_{ij}) := x_{ij}. Note that x_{ij} is indeed an OR of (a single) AND, namely x_{ij} = APX($\{x_{ij}\}$), and so this definition is legit.

Induction Approximator Step:

Case 1: OR gate $g_1 \vee g_2$. By induction hypothesis we already built the approximators of g_1, g_2, approx(g_1), approx(g_2), respectively. Using these, we construct an approximation of $g_1 \vee g_2$ denoted approx$(g_1 \vee g_2)$ (we shall show how to define this approximator as APX$(\mathcal{X})$, for some $\mathcal{X}$, in the sequel).

Case 2: AND gate $g_1 \wedge g_2$. By induction hypothesis we already built the approximators of g_1, g_2, approx(g_1), approx(g_2), respectively. We construct an approximation of $g_1 \wedge g_2$ denoted approx$(g_1 \wedge g_2)$ from these (again, we shall do this in the sequel).

Definition 2.3 (Introducing new false positives and false negatives) Let $g_1 \circ g_2$ be an $\vee$ or $\wedge$ gate with two incoming subcircuits g_1 and g_2, respectively (i.e., $\circ \in \{\vee, \wedge\}$). And let x_{ij} be in input node in the circuit (with no incoming edges).

- For *every* graph G, whether a positive or a negative instance, the approximator and the original function are the same: approx$(x_{ij})(G) = (x_{ij})(G)$. Hence we say that *neither a false positive nor a false negative was introduced*.
- Let the graph G be a *negative* input, i.e., a $(k-1)$-colouring. If

$$(\text{approx}(g_1) \circ \text{approx}(g_2))(G) = 0,$$

  namely, approx$(g_1) \circ$ approx(g_2) correctly identified G as a negative input, while

$$\text{approx}(g_1 \circ g_2)(G) = 1,$$

  we say the approximator step ***introduced*** G ***as a new false positive***.
- Let the graph G be a *positive* input, i.e., a k-clique. If

$$(\text{approx}(g_1) \circ \text{approx}(g_2))(G) = 1,$$

  namely, approx$(g_1) \circ$ approx(g_2) correctly identified G as a positive input, while

$$\text{approx}(g_1 \circ g_2)(G) = 0,$$

  we say the approximator step ***introduced*** G ***as a new false negative***.

Notice the subtlety in this definition: introducing new errors here means that we do not care for errors introduced before, namely, those errors incurred when we constructed each of the (separate) approximators for g_1 and g_2. We only count *new errors introduced over the function computed by* approx$(g_1) \circ$ approx(g_2).

To prove the lower bound theorem, all we need to know are *some properties* of the approximator steps. In other words, given these properties it would not be important how precisely we defined the approximators for gates (that is, how precisely we picked the $\mathcal{X}$ in each approximator APX$(\mathcal{X})$). In particular, the following two lemmas, proved in the next section, are sufficient to conclude the proof.

Lemma 2.1 *Each approximation step introduces at most* $M^2 \cdot \frac{(k-1)^n}{2^p}$ *false positives.*

Lemma 2.2 *Each approximation step introduces at most* $M^2 \cdot \binom{n-l-1}{k-l-1}$ *false negatives.*

Given these two lemmas, we can now relatively easily conclude the lower bound. The idea is that, if we assume by way of contradiction that we are given a small monotone circuit computing CLIQUE(n, k), then we can approximate the output gate of this circuit with an approximator that does not make too many errors: the circuit is small and each original gate when approximated contributes only a few new errors. But it is not hard to show that a good approximator for the CLIQUE(n, k) function does not exist. We formalise this argument in what follows.

Proof (of Theorem 2.1, given Lemma 2.1 and Lemma 2.2)
We use the following claim.

Claim Every approximator APX$(X_1, \ldots, X_m)$ with $|X_i| \leq l$ and $m \leq M$ is either identically 0, that is, an OR of *zero* ANDs, which is the formula 0 (and thus is wrong on all positive instances, namely all positive instances are false negatives in this case), or outputs 1 on at least half of the (extreme) negative instances (namely, at least half of the $(k-1)$-colouring are false positives).

Proof If APX$(X_1 \ldots X_m)$ is not identically 0 (namely, not the OR of *zero* ANDs, i.e., not the formula 0), then it accepts at least those graphs G that have cliques on at least one of the sets X_i, for some $i \in [m]$. The size of X_i is at most $l < k$, thus many graphs G that are not k-cliques still have l-cliques, which may cause the approximator to evaluate to 1 when there is a $|X_i|$-clique on the nodes in X_i.

In Figure 2.5 we see an example. The subset X_i is circled in green and contain four nodes. To fool the approximator to accept a $k-1$-colouring, the $k-1$-colouring should contain a clique on the four nodes in X_i, which means that each node in X_i is assigned a different colour and hence sent to a distinct independent set. The figure shows an example where this happens: the five independent sets are circled in red, and no two nodes in X_i live in the same independent set. Hence, there is a clique on all nodes in X_i.

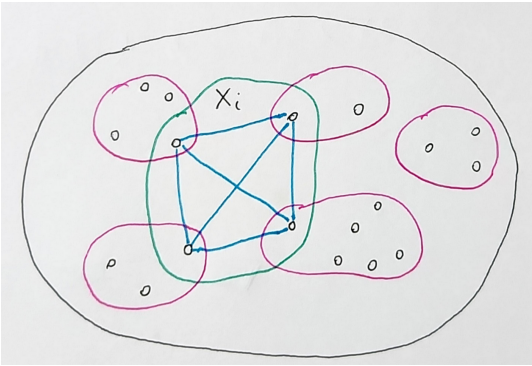


Fig. 2.5 Example of a $k-1$-colouring that fools an approximator that checks for a clique on the set X_i. The red sets are the independent sets, the green set is X_i and the blue edges are the clique formed on X_i that fools the approximator $\mathrm{APX}(X_1, \ldots, X_m)$.

Let us estimate how many such extreme graphs exist. Namely, we wish to show that there are many negative instances, i.e., $k-1$-colourings, that will nevertheless satisfy $\mathrm{APX}(X_1 \ldots X_m)$. It is simple to estimate this using *probability*. Consider a graph $G = (V, E)$ and assign randomly (and independently) from among $k-1$ colours to the nodes of G. Let $v_1 \neq v_2 \in V$ be two nodes, then we have

$$\Pr[\text{ colour of } v_1 = \text{ colour of } v_2] = \frac{1}{k-1},$$

because regardless of the colour v_1 got, there is only one colour out of $k-1$ options for v_2 that will match the colour of v_1.

Denote by $R(X_i)$ the event that in the random colouring of G, there exists a pair of nodes with the same colour in X_i. Then,

$$\Pr[R(X_i)] \leq \frac{\binom{|X_i|}{2}}{k-1} \leq \frac{\binom{l}{2}}{k-1} \leq \frac{1}{2}.$$

The first inequality from the left holds using the *union bound* and the fact that there are $\binom{|X_i|}{2}$ pairs of nodes in X_i. See below for a recap of the union bound and basic notions from discrete probability we shall use throughout this book. The rightmost inequality holds because $\frac{\binom{\sqrt[8]{n}}{2}}{\sqrt[4]{n}-1} = \frac{\frac{\sqrt[8]{n}\cdot(\sqrt[8]{n}-1)}{2}}{\sqrt[4]{n}-1} \leq \frac{\sqrt[4]{n}}{2\sqrt[4]{n}} = \frac{1}{2}$.

This means that out of all negative instances to CLIQUE (n, k) (namely, all $(k-1)$-colourings with n nodes), at least half of them are going to colour X_i with *different* colours. Hence, for these negative instances, there will be edges between all nodes in X_i, and thus for each of these negative instances G we have $\mathrm{APX}(X_1, \ldots, X_m)(G) = 1$. □

Union Bound: if in a random experiment there are m possible "bad" events, each with probability at most α, then the probability that at least one of the "bad" events occurred is at most $m\alpha$.

We recall some basic notions from **discrete (finite) probability**, and explain more formally the union bound in what follows.
In discrete probability we consider a *Sample Space* Ω as a finite set of all possible outcomes of a random experiment. The experiment is random in the sense that its outcomes are not deterministic rather hold probabilities. Each element e in Ω is a possible outcome of the random experiment, and is sometime called an *atomic outcome*, and holds a precise probability denoted $\Pr[e]$, such that:

1. $0 \leq \Pr[e] \leq 1$, for all $a \in \Omega$.
2. $\Sigma_{e\in\Omega} \Pr[a] = 1$.

An *event* is a set $E \subseteq \Omega$ of atomic outcomes. The probability of an event $E \subseteq \Omega$ is naturally defined as $\Sigma_{e\in E} \Pr[e]$.

The *union bound* provides an upper bound on the probability of the union of multiple events. Formally, if $E_1, E_2, \ldots, E_n$ are events in a probability space, the union bound states:

$$P\left(\bigcup_{i=1}^{n} E_i\right) \leq \sum_{i=1}^{n} P(E_i)$$

Intuitively, the union bound tells us that the probability of at least one of the events $E_1, E_2, \ldots, E_n$ occurring is at most the sum of the probabilities of each individual event. This bound is useful because it is often *much easier* to calculate or estimate the probabilities of *individual* events than to calculate the exact probability of their union.
If some events are not disjoint, their combined probability is generally less than the sum of their probabilities, so the union bound is not tight in this case. The union bound is tight (exact) if and only if the events are disjoint. Otherwise, it overestimates the probability due to double-counting overlaps among the events.

We are now ready to conclude the proof of Theorem 2.1 (given the two lemmas 2.1 and 2.2). Recall that $l = \sqrt[8]{n}$ and let $p = \sqrt[8]{n} \cdot \log n$. Thus,

$$\begin{aligned} M = (p-1)^l \cdot l! &\leq (\sqrt[8]{n} \cdot \log n)^{\sqrt[8]{n}} \cdot \sqrt[8]{n}! \\ &\leq (\sqrt[8]{n} \cdot \log n)^{\sqrt[8]{n}} \cdot \sqrt[8]{n}^{\sqrt[8]{n}} \\ &= (\sqrt[4]{n} \cdot \log n)^{\sqrt[8]{n}} < \left(n^{\frac{1}{3}}\right)^{\sqrt[8]{n}}, \text{ for large enough } n. \end{aligned} \tag{2.1}$$

Let C be a monotone circuit computing CLIQUE$(n, \sqrt[4]{n})$. The approximator of the output gate of C is written as $\mathrm{APX}(X_1, \ldots, X_m)$ for some $m \leq M$ and $|X_i| \leq l$. Based on the above claim, we have two cases:

Case 1: $\mathrm{APX}(X_1, \ldots, X_m)$ is identically 0. Thus, the number of *false negatives* $\mathrm{APX}(X_1, \ldots, X_m)$ has in total is the number of all possible positive instance, namely all possible k-cliques, which is $\binom{n}{k}$. By Lemma 2.2, each approximation step intro-

duces at most $M^2 \cdot \binom{n-l-1}{k-l-1}$ new false negatives. Therefore, *the number of gates in C is lower bounded by the total number of new errors introduced across all the gates in C (which is* $\binom{n}{k}$*), divided by the maximal number of errors introduced by each gate (which is* $\leq M^2 \cdot \binom{n-l-1}{k-l-1}$*).* In other words,

$$|C| \geq \frac{\binom{n}{k}}{M^2 \cdot \binom{n-l-1}{k-l-1}} = \frac{1}{M^2} \cdot \frac{n!}{k!(n-k)!} \cdot \frac{(k-l-1)!(n-l-k)!}{(n-l-1)!}$$
$$= \frac{1}{M^2} \cdot \frac{n}{k} \cdot \frac{n-1}{k-1} \cdots \frac{n-l+1}{k-l+1},$$

and since $\frac{n-i}{k-i} \geq \frac{n-l}{k}$ for $i = 0, \ldots, l-1$ and by Equation (2.1), we have

$$\geq \frac{1}{M^2} \cdot \left(\frac{n-l}{k}\right)^l \geq \frac{1}{n^{2/3 \cdot \sqrt[8]{n}}} \cdot \left(\frac{n - \sqrt[8]{n}}{\sqrt[4]{n}}\right)^{\sqrt[8]{n}}$$
$$\geq \left(\frac{\frac{1}{2}n}{n^{2/3} \cdot \sqrt[4]{n}}\right)^{\sqrt[8]{n}} = n^{\Omega(\sqrt[8]{n})} = 2^{\Omega(\sqrt[8]{n})}.$$

Case 2: $\mathrm{APX}(X_1, \ldots, X_n)$ has at least $\frac{1}{2}(k-1)^n$ *false positives* (half of the total number of $(k-1)$-colourings). By Lemma 2.1, each approximation step introduces at most $M^2 \cdot \frac{(k-1)^n}{2^p}$ new false positives. Therefore, similar to the argument in the previous case, we have:

$$|C| \geq \frac{\frac{1}{2}(k-1)^n}{M^2 \cdot \frac{(k-1)^n}{2^p}} = \frac{2^p}{2 \cdot M^2} > \frac{2^{\log n \cdot \sqrt[8]{n}}}{2 \cdot \left(n^{\frac{2}{3}}\right)^{\sqrt[8]{n}}} = n^{\Omega(\sqrt[8]{n})} = 2^{\Omega(\sqrt[8]{n})}.$$

This concludes the proof of Theorem 2.1. □

It is left to prove the two lemmas 2.1 and 2.2.

2.3.3 Proofs of Lemma 2.1 and Lemma 2.2: the Approximator Steps

Recall (see Definition 2.3) that we aim to show that for each input gate x_{ij} and each two internal gates g_1, g_2, the approximators $\mathrm{approx}(x_{ij})$, $\mathrm{approx}(g_1 \vee g_2)$ and $\mathrm{approx}(g_1 \wedge g_2)$, all defined as $\mathrm{APX}(\mathcal{X})$ for some families $\mathcal{X}$'s, well approximate x_{ij}, $\mathrm{approx}(g_1) \vee \mathrm{approx}(g_2)$ and $\mathrm{approx}(g_1) \vee \mathrm{approx}(g_2)$, respectively (where, by induction, $\mathrm{approx}(g_1)$ and $\mathrm{approx}(g_2)$ were already defined by $\mathrm{APX}(\mathcal{X})$ and $\mathrm{APX}(\mathcal{Y})$, for some two families $\mathcal{X}, \mathcal{Y}$).

We prove the base case of both lemmas in one shot, as the argument is the same.

Input node (base) Approximator Case:

We need to show that if x_{ij} is an input gate, then the approximator $\text{approx}(x_{ij}) = x_{ij}$ does not introduce many false positives or false negatives. This is immediate because the approximator for the input gate x_{ij} is x_{ij}, so no errors whatsoever are introduced.

$\vee$ and $\wedge$-gates (induction) Approximator Step

Goal to prove: Let $\text{APX}(\mathcal{X})$ and $\text{APX}(\mathcal{Y})$ be two approximators, with $\mathcal{X} = \{X_1, \ldots, X_m\}$, $\mathcal{Y} = \{Y_1, \ldots, Y_{m'}\}$, $m \leq M$, and $|X_i| \leq \ell$, for all $i \in [m]$, $m' \leq M$ and $|Y_i| \leq \ell$, for all $i \in [m']$. We wish to construct two approximators that do not introduce too many errors over $\text{APX}(\mathcal{X}) \vee \text{APX}(\mathcal{Y})$, and $\text{APX}(\mathcal{X}) \wedge \text{APX}(\mathcal{Y})$, respectively, in the sense that they introduce at most $M^2 \cdot \frac{(k-1)^n}{2^p}$ new false positives and at most $M^2 \cdot \binom{n-l-1}{k-l-1}$ new false negatives.

Case 1: $\vee$-gate.

Naive (wrong) attempt: $\text{APX}(\mathcal{X}) \vee \text{APX}(\mathcal{Y})$ is approximated by $\text{APX}(\mathcal{X} \cup \mathcal{Y})$. That is, $\text{APX}(X_1, \ldots, X_m, Y_1, \ldots, Y_m)$. At first glance this is a good solution because it does not introduce *any* errors (why?). But there is a *problem*: what if $m + m' > M$?

The solution to this problem is nontrivial. We need to cleverly *reduce* the number of sets $X_1, \ldots, X_m, Y_1, \ldots, Y_{m'}$. To do this we use a combinatorial lemma called The Sunflower Lemma, explained in what follows.

2.3.3.1 The Sunflower Lemma

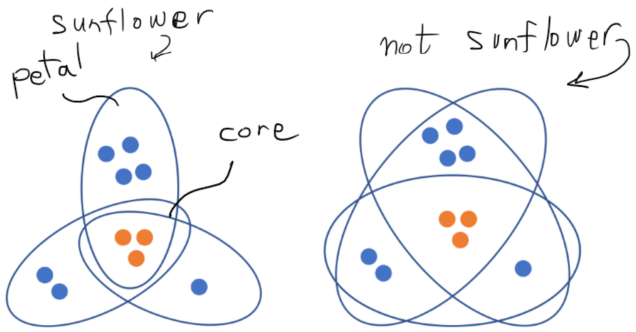


Fig. 2.6 From https://theorydish.blog/2021/05/19/entropy-estimation-via-two-chains-streamlining-the-proof-of-the-sunflower-lemma/ by Lunjia Hu

Definition 2.4 (Sunflower) Let U be some universe, namely a set of elements (e.g., nodes). Let $P = \{P_1, \ldots, P_p\}$ be a family of distinct sets from the universe, i.e., $P_i \subseteq U$, for each i, with p some natural number. We call the family P a *sunflower* if all pairs $P_i \neq P_j$ in the family P share the *same* intersection, called the *core* of the sunflower. In other words, there is a (possibly empty) set core $\subseteq U$, such that for all $i \neq j$, $P_i \cap P_j = \text{core}$. If P is a sunflower we call the P_i's the *petals* of the sunflower P.

In other words, in a sunflower P the intersection of every pair $P_i \neq P_j$ is fixed: $P_i \cap P_j = P_{i'} \cap P_{j'}$, for all $i \neq j$ and $i' \neq j'$. Note that it is okay if the core is the empty-set! This means all petals are (pairwise) disjoint.

Sunflower Lemma (Erdös-Rado): For every ℓ, p, let Z be a family of more than $M = (p-1)^\ell \cdot \ell!$ non-empty sets each of size $\leq \ell$ (over some universe U). Then, Z contains a sunflower of size p. In other words, Z contains p sets $\{P_1, \ldots, P_p\}$, each P_i has size $\leq \ell$, that share the same core.

Proof (Proof of the Sunflower Lemma) By induction on ℓ.

Base case: $\ell = 1$. Thus the statement we need to show is that p different singletons form a sunflower. Which is true (the core is $\varnothing$).

Induction step: $\ell > 1$. Consider a family $D \subseteq Z$ of pairwise disjoint sets that is maximal in the sense that if we add any new set in Z to the family D, the sets in D are not pairwise disjoint anymore. That is, every set in $Z \backslash D$ intersects some set in D.

Case 1: If D contains $\geq p$ sets then D is a sunflower with an empty core, and we are done.

Case 2: Otherwise, let E be the *union* of all sets in D. Since $|D| < p$, i.e., D contains less than p sets, and each set in D has size $\leq \ell$, we know that $|E| \leq (p-1) \cdot \ell$.

Moreover, E intersects every set in Z by assumption. Since Z has more than M sets by assumption, and each set intersects some element of E, there exists an element $d \in E$ that intersects $> \frac{M}{(p-1)\cdot l} = (p-1)^{l-1} \cdot (l-1)!$ sets in Z. [1]

Consider

[1] If a set E intersects all sets in a family of sets $X_1, .., X_M$ then there is an element in E that appears in $\geq \frac{M}{|E|}$ sets X_i. Otherwise, each element in E appears in $< \frac{M}{|E|}$ sets X_i. Thus, E intersects $< |E| \cdot \frac{M}{|E|} = M$ sets X_i, which is a contradiction to the assumption. Another way to look at this is through the pigeonhole principle: if E intersect each set X_i, then there is a functional mapping from each of the M sets X_i to a (single, arbitrarily chosen) element of E that occurs also in X_i. Thus, there must be an element of E with at least $\frac{M}{|E|}$ sets X_i that maps to it.

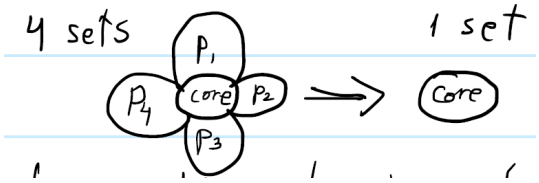
$$Z' := \{z \backslash \{d\} \mid z \in Z \text{ and } d \in z\}.$$

We know that Z' contains more than $M' = (p-1)^{l-1} \cdot (l-1)!$ sets. By *induction hypothesis* (since M' is "M with ℓ decreased by one"), Z' contains a sunflower denoted $\{P_1, \ldots, P_p\}$ with $|p_i| \leq \ell - 1$, for all i. Hence, $\{P_1 \cup \{d\}, \ldots, P_p \cup \{d\}\}$ is a sunflower in Z. □

Plucking

By the Sunflower Lemma, for every family of $> M$ nonempty sets, each of cardinality $\leq l$, we can find a sunflower with at least p sets, assuming $l = \sqrt[8]{n}$, $p = \sqrt[8]{n} \cdot \log n$ and $M = (p-1)^l \cdot \ell!$.

By *plucking a sunflower* we mean the process of replacing all petals by their core,



as in the following figure.

Corollary 2.1 (Repeated plucking) *With the parameters above, if we have $> M$ sets in a family, by repeated plucking we can reduce the number of sets to $\leq M$ (if we cannot apply plucking anymore we know by the Sunflower Lemma that the number of sets is $\leq M$).*

We use the notation pluck(Z) to denote the result of repeated plucking of a family Z of sets, until $|Z| \leq M$.

2.3.3.2 Approximator Step Continued: Approximating OR and AND using Plucking

We are finally ready to define precisely how to approximate OR and AND gates.

Definition 2.5 (Approximators for $\vee$ and $\wedge$ gates)

The approximate for $g_1 \vee g_2$: Assume that approx(g_1) = APX($\mathcal{X}$) and approx(g_2) = APX($\mathcal{Y}$). Define

$$\text{approx}(g_1 \vee g_2) := \text{APX}(\text{pluck}(\mathcal{X} \cup \mathcal{Y})).$$

The approximate for $g_1 \wedge g_2$: Assume that approx(g_1) = APX($\mathcal{X}$) and approx(g_2) = APX($\mathcal{Y}$). Define

$$\text{approx}(g_1 \wedge g_2) := \text{APX}\left(\text{pluck}\left\{X_i \cup Y_j : \left|X_i \cup Y_j\right| \leq l \text{ and } X_i \in \mathcal{X}, Y_j \in \mathcal{Y}\right\}\right).$$

Proof of Lemma 2.1

We need to show that each approximation step introduces at most $M^2 \frac{(k-1)^n}{2^p}$ false positives.

Case 1: OR-approximator. We start with $\mathrm{APX}(X_1, \ldots, X_m)$ and $\mathrm{APX}(Y_1, \ldots, Y_{m_m})$ and consider a false positive introduced by

$$\mathrm{APX}(\mathrm{pluck}(X_1, \ldots, X_m, Y_1, \ldots, Y_{m'})).$$

That is, a $(k-1)$-colouring $G = (V, E)$ such that,

$$\mathrm{APX}(X_1, \ldots, X_m)(G) = 0 \quad \text{and} \quad \mathrm{APX}(Y_1, \ldots, Y_m)(G) = 0 \tag{2.2}$$

while

$$\mathrm{APX}(\mathrm{pluck}(X_1, \ldots, X_m, Y_1, \ldots, Y_{m'}))(G) = 1. \tag{2.3}$$

We consider each plucking involved in Equation (2.3), and bound from above the number of false positive introduced by this plucking. (Note that plucking is the only reason a false positive could be introduced.)

Consider a single plucking[2]: replace sunflower $\{Z_1, \ldots, Z_p\}$ by its core Z (see Figure 2.7).

By Equation (2.2): Y_i's and X_i's are all non-cliques in G. Thus, by Equation (2.3), Z *is* a new clique identified by the approximation and for every $i \in [p]$, the petal of Z_i contains two nodes with the *same colour* in G (otherwise, there would have been a clique on some Z_i in contrast to Equation (2.2)).

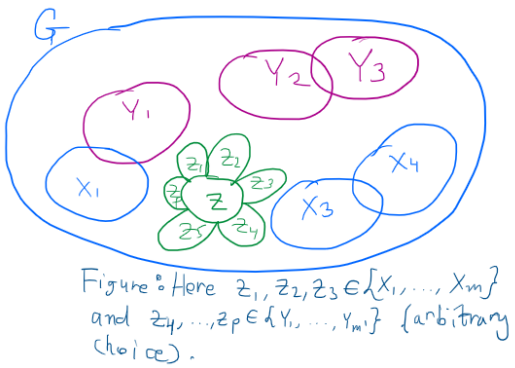


Fig. 2.7 Example

[2] It is fine to consider each single plucking, and count how many errors introduced in each such plucking. The reason is that once a plucking was done, we take the core Z of some sunflower; and this new set Z replaces all other set in the sunflower. Then we count the number of new errors introduced. Now, this Z could be replaced again in future pluckings, and we shall count the new errors introduced again.

We estimate the number of such $(k-1)$-colourings, using simple probability. For this purpose we consider the following question: choosing a $(k-1)$-colouring of the nodes randomly and independently, *what is the probability that every (petal of) Z_i has repeated colours but the core Z does not*?

As before, let $R(X)$ be the probability that X has repeated colours.

$$\Pr\left[R(Z_1) \wedge \ldots \wedge R\left(Z_p\right) \wedge \neg R(Z)\right] \leq \Pr\left[R(Z_1) \wedge \ldots \wedge R\left(Z_p\right) \mid \neg R(Z)\right]$$

(by the definition of conditional probability $\Pr[A \wedge B] = \Pr[A \mid B] \cdot \Pr[B]$)

$$= \prod_{i=1}^{p} \Pr\left[R(Z_i) \mid \neg R(Z)\right]$$

(Z_i's do not have common nodes, except those in Z, so given $\neg R(Z)$ the events $R(Z_i)$ are independent)

$$\leq \prod_{i=1}^{p} \Pr\left[R(Z_i)\right]$$

(the probability of repetition of colours is increased if we do not restrict ourselves to colourings with no repetitions in the core $Z \subseteq Z_i$)

$$\leq \frac{1}{2^p}$$

(because as we have seen before in Section 2.3.2, $\Pr[R(X)] \leq \frac{1}{2}$, for $|X| \leq \ell$).

Finally, since the approximation step entails up to $\frac{2M}{p-1}$ pluckings (each plucking decreases the number of sets by p − 1, and there are no more than $2M$ sets when we start), the lemma holds for the $\vee$ approximation step, because (recall the total number of $(k-1)$-colouring is $(k-1)^n$)

$$\frac{2M}{p-1} \cdot \frac{(k-1)^n}{2^p} \leq M^2 \cdot \frac{(k-1)^n}{2^p}.$$

Case 2: $\wedge$-gate. Consider now an AND approximation step of approximators $\mathrm{APX}(\mathcal{X})$ and $\mathrm{APX}(\mathcal{Y})$. It can be broken down in three phases, as follows.

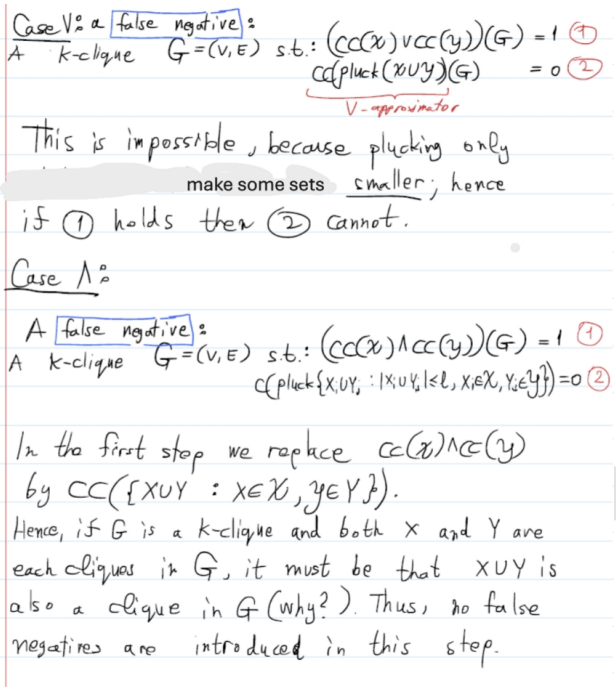
First, we form $\mathrm{APX}(\{X \cup Y : X \in \mathcal{X}, Y \in \mathcal{Y}\})$; this introduces no false positives, because any graph in which $X \cup Y$ is a clique must have a clique in both X and Y, and thus it was accepted by both constituent approximators.

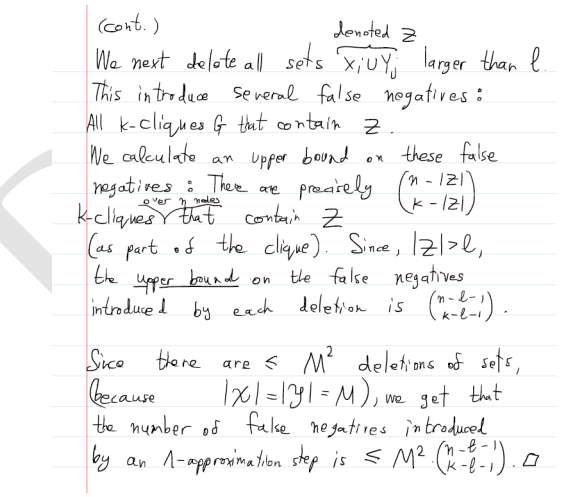
The second phase omits from the approximator circuit several sets (those of cardinality larger than ℓ), and can therefore introduce no false positives (because even less graphs are identified as having cliques).

The third phase entails a sequence of fewer than M^2 plucking, during each of which, by the analysis of the OR case above, at most $2^{-p}(k-1)^n$ false positives are introduced.

The proof of the lemma is complete: because at total we introduced $\leq M^2 \cdot \frac{(k-1)^n}{2^p}$ false positives.

Proof of Lemma Lemma 2.2

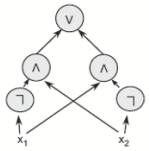




Chapter 3
Constant Depth Circuit Lower Bounds

Recall Definition 1.1 of Boolean circuits. We define the ***depth*** of a Boolean circuit C as the maximal length (i.e., number of edges) in a path from a leaf to the output node.

Here is an example of a circuit of depth 3:



Recall that we are interested in the study of the asymptotic size of circuit families, not a single circuit. That is, we want to consider how circuit size grows when the number of inputs n grows. In this chapter we shall focus on circuit families $\{C_n\}_{n=1}^{\infty}$ whose number of inputs bits n grows, their size $|C_n|$ grows polynomially while their depth is *fixed* throughout the family, namly is a constant c independent of n.

If the depth of the circuit is constant and the fan-in of gates is at most two, then the number of functions we can compute with such constant-depth circuits is constant. For example, the number of variables (appearing as leaves, i.e., input nodes) is constant that way, so for a large enough number of inputs n, we will not be able to compute functions that read all the n inputs. This means that the model is *not complete*: for a constant d, a depth-d circuit cannot compute all Boolean functions over n inputs for every $n \in \mathbb{N}$.

To solve this problem and make the model of constant-depth circuits meaningful we allow ***unbounded fan-in*** gates: unbounded fan-in AND gates: $(\varphi_1 \wedge \varphi_2 \wedge \ldots \wedge \varphi_\ell)$, and unbounded fan-in OR gates $(\varphi_1 \vee \cdots \vee \varphi_\ell)$, where the φ_i's are circuits (of constant depth) by themselves, with possibly *joint* nodes.

Definition 3.1 (Depth-d circuit family, constant-depth family) A circuit family $\{\mathrm{C}_n\}_{n=1}^{\infty}$ with unbounded fan-in $\wedge, \vee$ gates and fan-in one negation gate is said to be

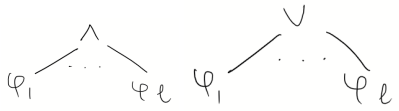


Fig. 3.1 Illustration of unbounded fan-in AND and OR gates.

of *depth-d*, if the depth of C_n is d, for all $n \in \mathbb{N}$. We denote by depth(C_n) the depth of the circuit C_n. If d is a constant (independent of n) we call $\{C_n\}_{n=1}^{\infty}$ a *constant-depth family of circuits*.

Notation: AC^0 denotes the complexity class consisting of all the functions $f : \{0,1\}^* \to \{0,1\}$ computable by constant-depth circuit families (equivalently, all decision problem decidable by AC^0 circuit families).

Alternating Formulas

We are going to use the following "normal-form" for constant-depth circuits. This is done for the sake of simplicity, and does not change the size of the circuits significantly, as we remark below.

A Boolean circuit is said to be an *alternating formula* if it is a formula (i.e., the underlying circuit graph is a tree), and every OR gate is followed by an AND gate followed by an OR gate, etc. Moreover, all negations in an alternating formula are pushed to the bottom of the circuit: that is, a negation node can only be connected to an input node (and that input node is not connected to any other node except a negation node). Here is an illustration:

For simplicity, we can assume that a constant-depth circuit has two types of inputs: every variables x_i has a negative copy $\overline{x}_i$ (thus, getting rid of the need to specify a negation gate explicitly).

Note that in an alternating formula every *layer* (equivalently, *level*) of the circuit, namely, nodes of a given depth l, are all of the same type: either all $\wedge$, all $\vee$ or all input variables (positive or negative).

We shall assume that all AC^0 circuits are alternating formulas:

Important: From now on, when speaking about constant-depth circuits (i.e., AC^0 circuits), we assume by default the fan-in of $\vee, \wedge$ gates is *unbounded* ($\neg$ has fan-in one), and the circuits are *alternating formulas*, and the **depth** is defined as maximal number of alternations between $\wedge$ and $\vee$ gates (or vice versa), namely the number of layers in the formula, excluding the bottom layer which contains only the input variables (positive or negative).

Exercise 3.1 Show that every depth-d circuit can be transformed into an alternating formula of depth-$O(d)$ circuit and with all negation gates on the leaves, with a polynomial size increase only.

—

Hint: we can "unwind" the circuit into a formula (a tree; that is, if we used a subcircuit more than once [namely, it has out-degree more than 1], we copy this subcircuit for each such usage). Because the depth is constant this will increase the size of the circuit by at most a polynomial (exponential in a constant—i.e., polynomial). To make the formula alternating, first push all the negation to the bottom, using *de Morgan rules*. And then, add dummy edges and gates to make every path from leaf to the output gate alternating between OR and AND. These all incur a polynomial increase in size.

3.1 The PARITY Function

The function PARITY determines the parity of the total number of ones in the input bits:

$$\text{PARITY}(x_1, \ldots, x_n) := \begin{cases} 1 & \text{if the number of 1's in } (x_1, x_2, \ldots, x_n) \text{ is odd,} \\ 0 & \text{if the number of 1's in } (x_1, x_2, \ldots, x_n) \text{ is even.} \end{cases}$$

Equivalently, PARITY is defined as the XOR of the input bits, denoted $x_1 \oplus \cdots \oplus x_n$. Another equivalent definition is using modulus: $x_1 + \cdots + x_n = 1 \mod 2$.

We denote by PARITY_n the PARITY function restricted to inputs of length precisely n.

The main result in this chapter is the following:

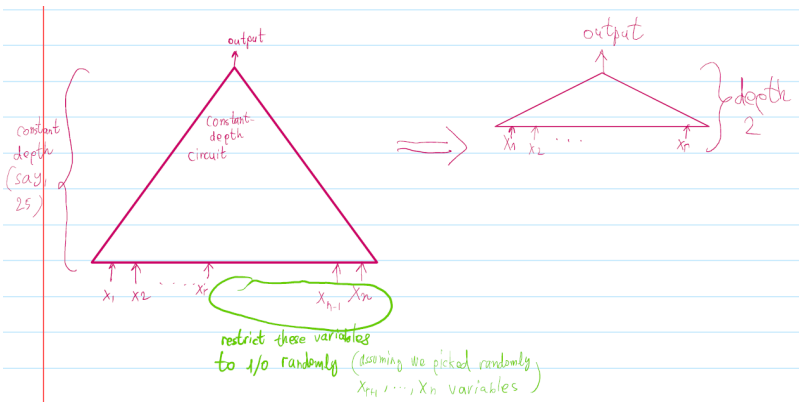
Theorem 3.1 (AC^0 lower bound for PARITY) *Every circuit family of depth-d computing the* PARITY_n *function requires size* $2^{\Omega\left(n^{1/(d-1)}\right)}$.

Note: this means that if d is constant, then the size lower bound is exponential: $2^{\Omega\left(n^{1/(d-1)}\right)} = 2^{\Omega(n^k)}$, for some constant $0 < k < 1$.

The theorem is a consequence of the fact that every function in AC^0 can be reduced to a constant function by setting relatively few variables to constants. The proof uses the Random Restriction Method described in the sequel.

3.2 The Skeleton of the Lower Bound Argument

Upshot of argument: The idea is to randomly restrict (assign) the variables of the circuit (of which we want to show it cannot compute PARITY). If the circuit is small, such a restriction collapses (gradually) the depth of the circuit to a depth-2 circuit. If the depth of the initial circuit was constant, this gradual step-by-step depth reduction needs to be done only for a constant many times. If we made random assignments (restrictions) only for a constant many times, we can rest assure that we have not assigned (i.e., fixed) too many variables. Hence, the function PARITY under these restrictions still has a lot of free variables. But such PARITY with sufficiently many free variables cannot have small depth 2 circuits (this we show by an auxiliary elementary argument).



Note 3.1 Notice the similarity with the monotone circuit lower bound proof: we use structural induction on the circuit purported to correctly compute PARITY.

Here are the main steps of the argument with slightly more detail.

Random Restriction Method: Structure of Argument

1. By way of contradiction, start with a purported small depth-d circuit C for PARITY_n. We are going to assign to some of the input variables of the circuit C (and hence, also to the function PARITY_n it is supposed to compute), random 0,1 values. We show the result of these assignments leads to conflicting effects: with high probability, while the circuit restricted to these randomly chosen partial assignments becomes a circuit that computes a very simple function, the function PARITY is still a rather complicated function to compute even after all these restriction.
2. Let $\overline{x}$ be n variables $x_1, \ldots, x_n$. Pick a random assignment $\rho_0 : \overline{x} \to \{0, 1, *\}^n$ with "$x_i \to *$" meaning that x_i is *un-assigned*, namely, stays a *free* variable. We make sure that enough x_i's stay free, as this is crucial to get our contradiction.

3. Depth Reduction Step:
   a. If C is a small circuit then the *function* computed by $C \restriction \rho_0$, namely the restriction of C to the assignment ρ_0, has a depth $d-1$ circuit of small size.
   b. Sequentially we assign new further random restrictions $\rho_1, \ldots, \rho_{d-3}$ (extending ρ_0) so that we reduce the depth of C by 1 each time, until the depth is 2.
4. At this point, we end up with a circuit C' of depth 2, namely a k-DNF or a k-CNF, for some small k that computes the function $\text{PARITY}_n \restriction \rho_0\rho_2 \ldots \rho_{d-3}$.
5. We now use an auxiliary statement to reach a contradiction: since we made sure that each assignment ρ_i leaves enough variables free, $\text{PARITY}_n \restriction_{\rho_0\rho_2 \ldots \rho_{d-3}}$ is the PARITY function on k' variables, i.e., $\text{PARITY}_{k'}$. But this is still a relatively "complex" function: in particular, it is a very simple argument to show that there are no k-CNF formulas for $\text{PARITY}_{k'}$, whenever $k' > k$, which will conclude the argument.

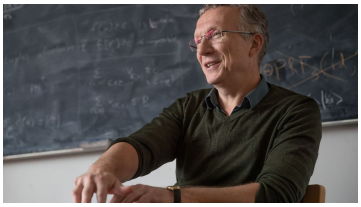


Fig. 3.2 Johan Håstad

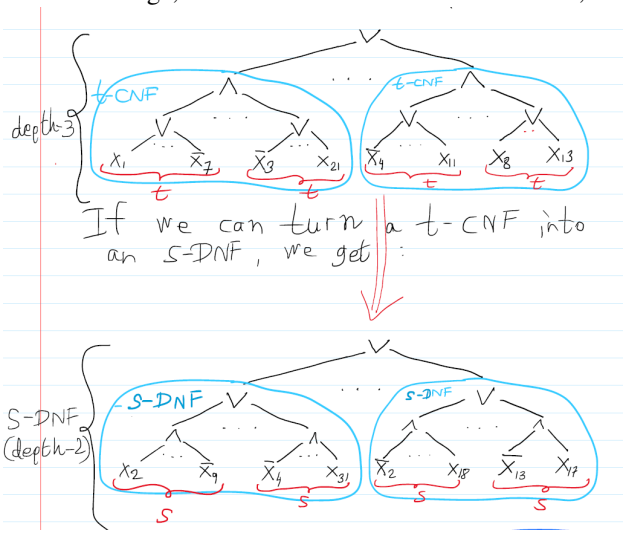
3.2.1 Depth Reduction Step: the Switching Lemma

We shall use the following notation.

- *Boolean (propositional) variables:* $x_1, x_2, \ldots$.
- *Literal:* x_i or $\neg x_i$ (i.e., a variables or its negation).
  A negation of a variable x_i is written as eiter $\neg x_i$, or $\overline{x_i}$.
- *Clause:* A disjunction of literals.
  Example: $(x_1 \vee \neg x_2 \vee x_3)$
- *CNF:* Conjunctive Normal Form formula, namely a conjunction of clauses.
  Example: $(x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee x_4 \vee x_5) \wedge (x_2 \vee \neg x_7 \vee \neg x_{20} \vee \neg x_{102})$
- *DNF:* Disjunction Normal Form formulas, namely a disjuncion of ANDs of literals (an AND of literals is sometimes called *a term*.
  Example: $(x_1 \wedge x_4 \wedge x_7) \vee (x_8 \wedge \neg x_9 \wedge x_{10}) \vee (x_3 \vee \neg x_2)$
- *t-clause:* A clause with at most t literals.
  Example: 3-clause: $(x_1 \vee \neg x_2 \vee x_7)$

- *t-CNF:* A CNF with all clauses being t-clauses.
- *s-DNF:* A DNF with all terms having at most s literals.

The below figure illustrates a single Depth Reduction Step. Such a step is also called "Switching", as it switches a t-CNF into an s-CNF, or vice versa.



Note 3.2 We can always turn a t-CNF into an s-DNF for some *big* s (by distributing out $\vee, \wedge$ using de Morgan's rules). But then we've not maintained the invariant that the bottom level has fan-in $\leq s$ (namely, s is small enough). In other words, we will end up with a depth-2 circuit, but of huge size computing $f \restriction_\rho$. This will not give us a contradiction since every function is computable by an exponential-size circuit.

Examples of Assignments and their Consequences

How can we turn a t-CNF into an s-DNF while keeping s small? We can turn the t-CNF to a (general) DNF using de Morgan rules (just distribute $\wedge$ over $\vee$), for example

$$(x_1 \vee \bar{x}_2) \wedge (x_3 \vee \bar{x}_4) \equiv (x_1 \wedge x_3) \vee (x_1 \wedge \bar{x}_4) \vee (\bar{x}_2 \wedge x_3) \vee (\bar{x}_2 \wedge \bar{x}_4),$$

and then try to *eliminate literals* in big ANDs by using assignments to the literals in those terms. This may reduce indeed the size of terms, and if we do this enough we may be able to reduce all the terms that are wider than s.

Consider the following DNF:

$$(x_1 \wedge x_2 \wedge \bar{x}_3 \wedge \bar{x}_5 \wedge \bar{x}_8) \vee (\cdots \wedge \bar{x}_9 \wedge \ldots) \vee \ldots.$$

We can assign 1 to x_2 and 1 to x_9. In the first conjunct (i.e., the first AND or term) it results in the *deletion of* x_2, while in the second conjunct, it results in *deletion of the whole conjunct*:

Assigning 1 to x_2:

$$(x_1 \wedge x_3 \wedge \bar{x}_5 \wedge \bar{x}_8) \vee (\cdots \wedge \bar{x}_9 \wedge \ldots)$$

Assigning 1 to x_9:

$$(x_1 \wedge x_3 \wedge \bar{x}_5 \wedge \bar{x}_8) \vee \underbrace{(\cdots \wedge 0 \wedge \ldots)}_{\text{false}}$$

Note 3.3

1. Literal $l_j \leftarrow 0$ (i.e., assigning 0 to literal l_j)

$$(l_j \wedge D) \equiv \text{false}.$$

   So the whole AND disappears.
2. Literal $l_j \leftarrow 1$:

$$(l_j \wedge D) \equiv D.$$

   So we reduced the size of the AND by 1.

But how can we choose the assignment of 0 and 1's to variables that guarantees eliminating all the large conjuncts, without fixing all the variables in the DNF formula (recall that by Item 2 above we need to keep sufficiently many variables x_i free)? The idea is to use the *probabilistic method*: choosing a *random* $g : \overline{x} \to \{0, 1, *\}^n$ will lead with **positive** probability to an assignment that eliminates *all* large conjuncts, while leaving enough variables free. When an event occurs with a positive probability, in this case the event of choosing an assignments with the desired properties, it means that there **exists** at least one such assignment. This will be enough for our purpose: the existence of such an assignment is sufficient to conclude our proof, because our argument hinges on a proof by contradiction: if there exists a small constant-depth circuit for PARITY, then there *exists* also a desired assignment (formally, a sequence of assignments) that will yield our contradiction as in Item 5.

Switching via Random Restrictions

Notation:

1. A *restriction* ρ for l variables in $\bar{x}$ is a function $\rho : \bar{x} \to \{0, 1, *\}$, assigning either 0, 1 values to the variables or leaving them free, namely $\rho(x_i) = *$ means that x_i is *unassigned* (or "free") according to ρ.
2. For a function $f : \{0,1\}^n \to \{0,1\}$ and a restriction $\rho : \bar{x} \to \{0, 1, *\}$, we write $f \restriction \rho$ to denote the function f where the variables are set according to ρ.
   If we then apply another restriction τ to the variables in $f \restriction \rho$, we write $f \restriction_{\rho\tau}$ to denote $(f \restriction \rho) \restriction \tau$. In other words,

$$f \restriction_{\rho\tau} = f \restriction_{\rho \cup \tau}$$

where $\rho \cup \tau$ is the union of two functions.

Definition 3.2 (Random Restriction) Let $0 < p < 1$. Let $\mathcal{R}$ be a distribution on partial restrictions ρ to the variables $\overline{x}$ with $|\overline{x}| = n$, as follows.

$$\Pr[\rho(x_i) = *] = p$$
$$\Pr[\rho(x_i) = 0] = \Pr[\rho(x_i) = 1] = \frac{1-p}{2}.$$

In other words, we pick randomly and independently for each variable x_i with $i \in [n]$, either to leave it free with probability p, or to fix it to 1 or 0 with equal probability $(1-p)/2$.

Theorem 3.2 (Switching Lemma) *For every positive integers r and k and n, if a Boolean function f with n variables is computable by an r-DNF and ρ is a random restriction as above with $0 < p \leq \frac{1}{9}$, then with probability at most $(9pr)^s$, $f \restriction_\rho$* cannot *be computable by as an s-CNF.*

The Switching Lemma thus shows that the "bad" event of a function computed by an r-DNF formula not being able to switch, under a random restriction, to a function that can be computed by an s-CNF, is small. This would mean that with high enough probability the "good" event happens: a random assignment switches an r-DNF to an s-CNF. In practice, we usually need to have the probability of the bad event to be *really* small, in this case exponentially small, so that the probability that *none* of a collection of bad events happens in the random experiment is small (using the union bound), and specifically smaller than 1, to be able to claim there exists an object with the required properties (in our case, an assignment that would switch exponentially many r-DNFs).

However, we note that this probabilistic argument via the union bound is *not* sufficient to conclude the argument. We shall also need to make sure that such a "good" random assignment *leaves many variables free*. For this purpose, we will use a common tool in probabilistic analysis and concrete complexity: *concentration bounds* (in our case it will be the Chernoff Bound). We explain this point in the the next section when proving the lower bound.

Exercise 3.2 Show that by applying the Switching Lemma to the function $\neg f$ we can get the same result with the terms "DNF" and "CNF" interchanged. Namely, the probability that an r-CNF does not switches to an s-DNF is at most $(9pr)^s$.

3.3 Proof that PARITY does not have small circuits, using the Switching Lemma

Recall that we want to prove Theorem 3.1 stating that every circuit family of depth-d computing the PARITY function requires size $2^{\Omega\left(n^{1/d-1}\right)}$.

So, suppose that C is a depth-d alternating formula circuit of size s computing PARITY_n. We are going to show that $s = 2^{\Omega\left(n^{1/d-1}\right)}$, since assuming otherwise would lead to a contradiction.

Recall that we assume that C is an alternating formula, with d layers, where in each layer all gates are either $\wedge$ or $\vee$. The proof proceeds by first a "preprocessing step" in which we use the Switching Lemma to decrease the fan-in of the *bottom* layer gates, followed by $d-2$ depth reduction steps using the Switching Lemma in which we gradually reduce the depth of the circuit, from bottom to top, by one depth, until we reach a depth-2 circuit. After this, we can conclude the theorem: if the circuit was too small, we end up with a k-CNF (or k-DNF) computing PARITY over more than k variables which is easyly shown to be impossible.

Decreasing Bottom Layer Fan-in

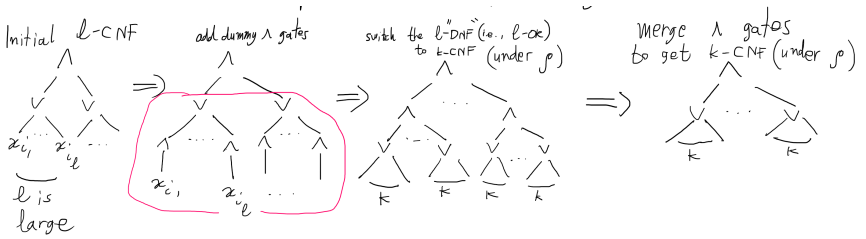
This step is needed in order to optimise the lower bound, namely to reach the tight lower bound of $2^{\Omega(n^{1/d-1})}$. % we may explain this point later.

Consider the bottom two layers of the given constant-depth circuit. We assumed that the layers alternate between $\wedge$ and $\vee$, thus the depth-2 sub-formulas at these bottom two layers are all either DNFs or CNFs. Let us assume without loss of generality that they are all CNFs. The bottom fan-in of these CNFs may be large, namely, it may be ℓ-CNFs for a large ℓ. Since ℓ may be very large, using the Switching Lemma to turn this ℓ-CNF to an s-DNF will be too expensive, in the sense that applying the Switching Lemma with this ℓ would give us probability of $\leq (9p\ell)^s$, but with *large* ℓ, which later would mean we get a weaker lower bound (e.g., we sould get a lower bound of $2^{\Omega(n^{\frac{1}{d}})}$, which is weaker, namely smaller, than our lower bound of $2^{\Omega(n^{\frac{1}{d-1}})}$). % This is caused by a weaker union bound computation (we explain this after we finish the proof).

For this reason, we shall reduce the fan-in of the bottom layer (i.e., layer 1) directly using the Switching Lemma (while maintaining a lower probability of failure).

Claim An ℓ-CNF does not turn into a k-CNF under a random restriction ρ with probability $\leq (9p_0)^k$, for p_0 the probability a variable is free in the random restriction ρ. Specifically, when $k = 2 \cdot \log S$, if we choose $p_0 = \frac{1}{18}$, the probability of failure is at most $\frac{1}{S^2}$.

Proof (Claim) The below figure illustrates the simple argument:



In particular, we switch the lower 1-DNFs (*in red*) into k-CNF with a random restriction, so by the Switching Lemma, the probability the switching fails is

$$\leq (9 \cdot p_0 \cdot 1)^k = (9 \cdot \frac{1}{18})^{2 \cdot \log S} = \frac{1}{2^{2 \cdot \log S}} = \frac{1}{S^2}.$$

□

By this claim, and using the union bound, the probability that at least one ℓ-CNF in layer 2 does not switch is at most $S \cdot 1/S^2 = 1/S$.

Repeated Depth Reduction Steps

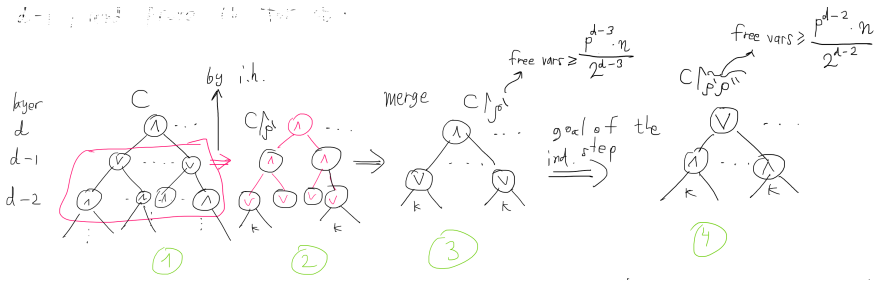
We now start from a circuit $C\restriction_{\rho_1}$ with k-CNF at the bottom, for $k = 2 \log S$. We go by induction on depth, to prove the following.

Lemma 3.1 (Induction Statement) *Let C be a depth at least d circuit of size S with layer-1 (bottom) gates all of fan-in $\leq k = 2 \log S$ (either $\wedge$ or $\vee$), with n input variables $\overline{x}$ (with positive and negative copies). Assume that the gates in layer d are all $\wedge$ gates (resp., $\vee$ gates). Then, there exists a restriction $\rho : \overline{x} \to \{0, 1, *\}$ with at least $\frac{p^{d-2} \cdot n}{2^{d-2}}$ free variables, where $p = \frac{1}{36 \cdot \log S}$, such that the function computed by $g\restriction_{\rho}$ for every $\wedge$ (resp. $\vee$) gate in layer d is computable by a k-DNF (resp., a k-CNF).*

Proof (Lemma 3.1) We prove it for the case $\wedge$ in layer d. The $\vee$ case is similar.

Base case: $d = 2$. Since the bottom layer is assumed to be of fan-in $\leq k$, this case holds immediately.

Induction step: We assume the induction statement holds for layer $d - 1$, and prove it for d. The following figure illustrates the argument.



□

The setup of the induction step: In (**1**) in the figure, we have (by induction hypothesis) a depiction of part of the circuit C, showing a single $\wedge$ gate in layer d (there may be other $\wedge$ gates in this layer, and there may be other gates above it in layers higher than d). In (**2**) we see the $\wedge$ gate in layer d under the restriction ρ', i.e., $C \restriction_{\rho'}$, with ρ' having $\geq \frac{p_0^{d-3} \cdot n}{2^{d-3}}$ free variables. Hence, its children, which are $\vee$ gates, have turned under ρ' to k-CNFs. (**3**) simply merged the $\wedge$ gate in layer d to its children in layer $d-1$, all of which turned to $\wedge$ gates themselves. Our goal is to get to circuit (**4**), by applying the Switching Lemma with a random restriction ρ'' and then compute an upper bound on the probability that such a switch to k-DNF occurs for *all* the gates in layer d. We explain this now.

Using the Switching Lemma with the parameters $r = s = k = 2 \log S$, with probability $\leq (9pk)^k$, a *single* k-CNF in layer d does not switch to a k-DNF. By the union bound, with probability $\leq S \cdot (9pk)^k$ the bad event that *at least one* k-CNF in layer d does *not* switch to a k-DNF occurs. We now use another bound to estimate the probability of the event that a random restriction has too few free variables. The tool we use is the Chernoff bound.

Chernoff Concentration Bound. Random variables and expectations: A random variable X is a mapping from the sample probability space of elementary (or "atomic") outcomes to the real line, $X : \Omega \to \mathbb{R}$. For example, consider a random variable with $X_i = 1$ if E_i occurs, and 0 otherwise. (For example, a random event is flipped as "heads"). The indicator variable X taking values in $[0, 1]$, with $X_i = 1$ if E_i happened, 0 otherwise.
If a random variable X takes its values from a set A, we define the expectation of X as follows:

$$\mathbb{E}[X] = \sum_{a \in A} Pr[X = a] \cdot a \,.$$

Linearity of expectation: For a collection of random variables $X_1, X_2, ..., X_n$ (not necessarily independent), we have:

$$\mathbb{E}\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} \mathbb{E}[X_i].$$

Concentration around the expectation: For a random variable $X = \sum_{i=1}^{n} X_i$, that is a sum of *independent* random variables (say each X_i takes values in $\{0, 1\}$), a strong bound is given by the Chernoff inequality. In particular, it says that the probability that this X is at least twice the expectation is inversely-exponentially small in the expectation.

Theorem 3.3 (Chernoff bound) *Let $X_1, X_2, ..., X_n$ be independent identically distributed random variables such that $Pr[X_i = 1] = p$ and $Pr[X_i = 0] = 1 - p$, for some $0 \leq p \leq 1$. Consider the new random variable $X = \sum_{i=1}^{n} X_i$, with expectation $\mu = pn$. Then, for any $0 < \delta < 1$,*

$$Pr[|X - p| > \delta p] \leq 2e^{-\frac{\delta^2 \mu}{3}}.$$

We apply Chernoff bound to our case as follows. Every single variable x_i has a probability p to stay free under our new restriction ρ''. Hence, the expected number μ of free variables is pN, where N is the number of (free) variables in $C \restriction_{\rho'}$ which is $\geq \frac{p^{d-3} \cdot n}{2^{d-3}}$ by induction hypothesis. Hence, the expected number of free variables is $\mu \geq pN = \frac{p^{d-2} \cdot n}{2^{d-3}}$. (Since the event that a variable stays free under ρ'' is independent from the events that other variables stay free, the expected value is achieved with high probability by the Chernoff bound.) In other words, the bad event that the number of free variables are less than half of the expected value μ is at most $2e^{-\delta^2 \cdot \frac{\mu}{3}}$, with $\delta = \frac{1}{2}$ and $\mu = \frac{p^{d-3} \cdot n}{2^{d-2}}$.

The following is thus the upper bound on the probability that at least one bad event happened (using the union bound):

$$S(9pk)^k + 2e^{-\delta^2 \cdot \frac{\mu}{3}}, \quad with \quad \delta = \frac{1}{2} \quad \text{and} \quad \mu = \frac{p^{d-3} \cdot n}{2^{d-3}}, \tag{3.1}$$

where the left most term is by a union bound on all gates, and the rightmost term is the bad event estimated by the Chernoff bound.[1]

Equation (3.1) is equivalent to

$$S\left(9 \cdot \frac{1}{36 \cdot \log s} \cdot 2 \log s\right)^{2 \log s} + 2e^{-\frac{\left(\frac{1}{36 \cdot \log S}\right)^{d-3} \cdot n}{2^{d-1} \cdot 3}}.$$

Let us denote the right summand above by B. We get:

$$= S \cdot \left(\frac{18}{36}\right)^{2 \cdot \log S} + B = S \cdot \frac{1}{S^2} + B = \frac{1}{S} + \underbrace{e^{-\frac{\frac{2}{n}}{(36 \cdot \log S)^{d-3} \cdot 2^{d-1} \cdot 3}}}_{B}$$

We need to compute: $e^{-\frac{2}{(36 \log S)^{d-3} \cdot 2^{d-1} \cdot 3}} \geq$ → we need (B) $\leq 1 - \frac{1}{S}$

$\frac{n}{(36 \log S)^{d-3} \cdot 2^{d-1} \cdot 3} \geq 2 \Rightarrow n \geq (36 \log S)^{d-3} \cdot 2^{d-1} \cdot 6 \Rightarrow$

$\underbrace{\sqrt[d-3]{\frac{n}{6 \cdot 2^{d-1}}}}_{36} \geq \log S \Rightarrow S \leq 2^{\frac{\sqrt[d-3]{\frac{n}{6 \cdot 2^{d-1}}}}{36}}$ → but this is correct. if $S > 2^{\sqrt[d-3]{n}} > 2^{\sqrt[d-2]{n}}$ we finish the proof because

$2^{\frac{d-1}{d-3}} < 2^2 = 4$ □

(Note: $(\log S)^{d-3} \cdot 2^{d-1} = n^{\frac{d-3}{d-1}} \cdot 2^{d-1}$ $\Big|_{S = 2^{n^{\frac{1}{d-1}} \cdot 2^{d-1}}}$)

□

Concluding the proof

Using the Induction Statement together with the Bottom Switching Step, we can now conclude the proof of the lower bound. Given a circuit of depth d with n variables, we first apply the Bottom Layer Switching (with sufficiently high probability using Chernoff's bound, the number of free variables after this step are at least $(1/36) \cdot n$; we skip this computation).

[1] Note that we have used the union bound on the following bad events, given a random pick of a single assignment ρ: S many potential bad events that a gate in layer d does not switch, together with a single event that the number of free variables in ρ falls below half the expected number of free variables.

Then, we have a circuit C of depth-d with layer 2 consisting of k-CNF gates. We then apply the Induction Statement. We reach a circuit $C \restriction_\rho$ of depth-2, namely a k-CNF (or k-DNF) with $\rho : \bar{x} \to \{0, 1, *\}$ being a restriction with $\geq \frac{p^{d-2} \cdot n}{2^{d-2}}$ free variables and $p = \frac{1}{36 \cdot \log S}$, such that the function computed by $g \restriction_\rho$ for every $\wedge$ gate g in layer d (resp., $\vee$ gate) can be computed by a k-DNF, with $k = 2 \log S$ (resp. a k-CNF). And specifically, when d is the depth of C, we know that $C \restriction_\rho$ as a whole is computed by a k-CNF (in this case d is a layer with a single gate).

But now we get a contradiction if k is less than the number of free variables, which is

$$\frac{p^{d-2} \cdot n}{2^{d-2}}.$$

This is because every k-CNF can be made the constant function 0 with k assignments of 0/1 values to its variables: Pick one k-clause, e.g.,

$$x_1 \vee \overline{x_2} \vee x_3 \vee \overline{x_4},$$

assign the literals in the clause all zero:

$$x_1 = 0,\ x_2 = 1,\ x_3 = 0,\ x_4 = 1.$$

However, Parity for $r > k$ many inputs *cannot* be made a constant function without assigning all its r variables. Hence, when the number of free variables

$$\frac{p^{d-2} \cdot n}{2^{d-2}} > k = 2 \log S,$$

we get a contradiction. In other words, we have:

$$2 \log S \geq \frac{p^{d-2} \cdot n}{2^{d-2}}.$$

We now compute to find out what is the lower bound for S:

$$2 \log S \geq \frac{p^{d-2} \cdot n}{2^{d-2}} = \frac{\left(\frac{1}{36 \log S}\right)^{d-2} \cdot n}{2^{d-2}} = \left(\frac{1}{\log S}\right)^{d-2} \cdot n \cdot \left(\frac{1}{72}\right)^{d-2}.$$

Hence,

$$\begin{aligned}
(\log S)^{d-1} &\geq \frac{1}{2}\left(\frac{1}{72}\right)^{d-2} \cdot n \implies \\
\log S &\geq \frac{1}{2}\left(\frac{1}{72}\right)^{\frac{d-2}{d-1}} \cdot n^{\frac{1}{d-1}} \implies \\
\log S &= \Omega\left(n^{\frac{1}{d-1}}\right) \implies S = 2^{\Omega(\sqrt[d-1]{n})},
\end{aligned}$$

concluding the proof. □

Proof of the Switching Lemma

Random Restriction

Let $0 < p < \frac{1}{q}$.

Let $\mathcal{R}$ be a distribution of partial restrictions ρ to the variables $\bar{x}$ (where $|\bar{x}| = n$) as follows:

$$\Pr[\rho(x_i) = *] = p$$

$$\Pr[\rho(x_i) = 0] = \Pr[\rho(x_i) = 1] = \frac{1-p}{2}$$

Function Assumption

From now on, let f be an **r-DNF** with n variables.

Order Assumption

Order: We assume a fixed order on conjuncts in f, and a fixed order on literals in each conjunct.

Canonical Tree $T(f|_\rho)$

A decision tree for $f|_\rho$ is defined as follows:

1. Look through f for a conjunct C such that $C|_\rho \neq 0$.
   - If no such C exists, output "0" (i.e., the tree is the single node 0).
   - Otherwise, let C_1 be the first such conjunct.
2. Let β_1 be the variables that appear starred in $C_1|_\rho$ (i.e., they are unassigned). Query all variables in β_1 in order. Denote by Π the assignment/path given to these variables.
3. If $C_1|_{\rho\Pi} = 1$, output "1" (i.e., a node 1 in this leaf).
   - If β_1 was empty (meaning $C_1|_\rho = 1$), output "1" as well.

4. Otherwise, $C_1|_{\rho\Pi} = 0$ and go to step (1), starting with $\rho\Pi$ in place of ρ, looking at conjunct C_2, which is the first one such that $C_2|_{\rho\Pi} \neq 0$.
5. Continue in this manner until you run out of conjuncts.

Example: (1)

$$F: \quad r\text{-DNF} \quad (x_1 \wedge x_2 \wedge x_4 \wedge \bar{x}_3) \vee (x_5 \wedge \bar{x}_2 \wedge x_4) \vee \dots$$

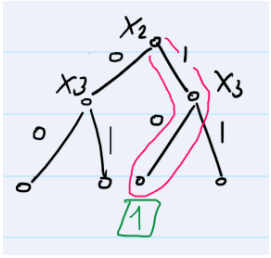
$$\rho: \quad \begin{array}{ccccc} x_1 & x_2 & x_3 & x_4 & x_5 \\ 1 & * & * & 1 & * \end{array}$$

$$F|_\rho = (x_2 \wedge \bar{x}_3) \vee (x_5 \wedge \bar{x}_2) \vee \dots$$

Start with the first (from left) conjuncts in $F|_\rho$ that is not 0. In our case it's

$$C_1|_\rho = (x_2 \wedge \bar{x}_3).$$

Build query tree for $C_1|_\rho$; i.e., query each variable in order they appear in $C_1|_\rho$:

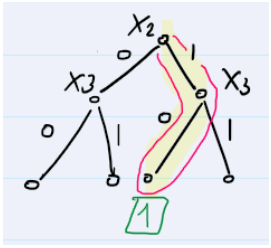


Note that the red path ($x_2 \leftarrow 1, x_3 \leftarrow 0$) determines an assignment Π_1 (for x_2, x_3) that satisfies 1, and hence satisfies $F|_\rho$. Thus, we label the leaf of this path by 1 (and finish the path there, with no more queries).

$$F: \quad r\text{-DNF} \quad (x_1 \wedge x_2 \wedge x_4 \wedge \bar{x}_3) \vee (x_5 \wedge \bar{x}_2 \wedge x_4) \vee \dots$$

$$\rho: \quad \begin{array}{ccccc} x_1 & x_2 & x_3 & x_4 & x_5 \\ 1 & * & * & 1 & * \end{array}$$

$$F|_\rho = (x_2 \wedge \bar{x}_3) \vee (x_5 \wedge \bar{x}_2) \vee \dots$$

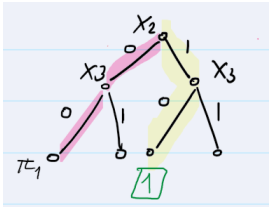


Consider each leaf in the tree above. It determines a path Π_1 from the root to the leaf.
Now, in each such leaf (except the ones labeled by 1), we need to continue querying the variables to determine an *extension* of the assignment Π_1, so that this extended assignment will either satisfy or falsify $F|_\rho$.

$$F: \quad r\text{-DNF} \quad (x_1 \wedge x_2 \wedge x_4 \wedge \bar{x}_3) \vee (x_5 \wedge \bar{x}_2 \wedge x_4) \vee \dots$$

$$\rho: \quad \begin{array}{ccccc} x_1 & x_2 & x_3 & x_4 & x_5 \\ 1 & * & * & 1 & * \end{array}$$

$$F|_\rho = (x_2 \wedge \bar{x}_3) \vee (x_5 \wedge \bar{x}_2) \vee \dots$$



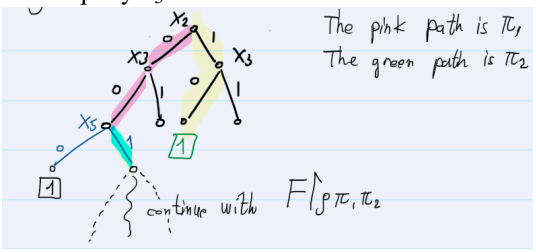
Consider now the leftmost leaf. Then

$$\Pi_1 = \{x_2 = 0, x_3 = 0\}$$

We continue to build the tree from this leaf.
The first (left-most) clause in $F|_{\rho,\Pi_1}$ that is non-zero is

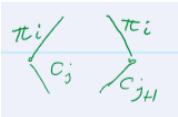
$$C_2|_{\rho,\Pi_1} = x_5$$

So we query x_5:



The pink path is Π_1, the green path is Π_2.
Continue with $F|_{\rho,\Pi_1,\Pi_2}$.

Note that each layer in a DT can contain queries related to different clauses. This can happen for instance if Π_i on one node makes 0 the next clause C_j, while Π_i on a different node in the same layer doesn't nullify C_j:



Claim Let g be a DNF and assume that the height of $T(g) \leq s$. Then g can be written as an s-CNF as well as an s-DNF.

Proof Each leaf of $T(g)$ determines an assignment π that fixes g to either 1 or 0 (i.e., $g|_\pi \in \{0, 1\}$).

Thus, for every assignment π to the variables, $g|_\pi = 0 \Leftrightarrow \pi$ leads to a 0-leaf in $T(g)$.

Let A_π be the AND determined by an assignment π: if $\pi(x_i) = 1$ add x_i, if $\pi(x_i) = 0$ add $\neg x_i$.

$$\bigvee_{\pi \in J} A_\pi$$

is an s-DNF computing the function $\neg g$.

Thus,

$$\neg \bigvee_{\pi \in J} A_\pi = \bigwedge_{\pi \in J} \neg A_\pi$$

is an s-CNF computing $\neg\neg g = g$.

In other words, a canonical decision tree for F decides $\neg F$, if we consider the assignments on 0-leafs. Thus, if we turn these ANDs corresponding to these assignments to OR of the negated literals, and AND all these ORs, we get a CNF computing F.

Theorem 3.4 (Switching Lemma) *If f is an r-DNF with n variables and ρ is a random restriction as above with $p < \frac{1}{q}$, then with probability $\leq (9pq)^s$, the height of $T(f|_\rho)$ is bigger than s.*

This means that $f|_\rho$ **cannot** be written as an s-CNF with probability $\leq (9pq)^s$, by the claim above.

Discussion of the main idea behind the proof

Let $\mathcal{R}_\rho$ be the distribution of random restrictions, taken with probability $P_\rho[X_i = *] = p$ as before.

Let S be the set of "bad" restrictions $\rho \in \mathcal{R}_\rho$. That is, restrictions for which $T(f|_\rho)$ has height $> s$. We wish to bound from above $P_\rho[\rho \in S]$.

A priori, we only have the trivial bound:

$$P_{\rho \in \mathcal{R}_\rho}[\rho \in S] \leq 1,$$

namely, 100% of restrictions in $\mathcal{R}_\rho$ are bad.

We show how to reduce this probability by a factor of about $\left(\frac{1}{4}\right)^s$. Namely, get:

$$P_{\rho \in \mathcal{R}_\rho}[\rho \in S] \leq \left(\frac{1}{4}\right)^s \cdot (\text{small number}).$$

But how can we show this?

(cont.) Discussion of the main idea of the proof

The idea is to show that the probability of the event of picking a bad restriction is bounded from above by the probability of picking a restriction from a *different* set of restrictions, denoted T.

If S contained a *single* restriction ρ, this would be easy to achieve:

$$\Pr_{\mathcal{R}}[S] \leq \frac{1}{4^s}, \quad \Pr_{\mathcal{R}}[\{\rho\}] \leq \frac{1}{4^s} \quad (\text{because trivially } \Pr_{\mathcal{R}}[\{\rho\}] \leq 1)$$

for every restriction ρ to s additional variables.

The first inequality follows by the fact that making a restriction *longer* only *increases* its probability, since fixing a *single* variable to $\{0, 1\}$ has probability

$$\frac{1-p}{2} > \frac{8p}{2} = \frac{4}{9},$$

while fixing a variable to $*$ has probability only $< \frac{1}{9}$.

Here is a simple example when extending the assignment by one bit:

$$\Pr_{\mathcal{R}}[S] = \frac{1}{1-p} \cdot p \cdot \Pr_{\mathcal{R}}[S_0] < 2 \cdot \frac{1}{9} \cdot \frac{1}{9} \Pr_{\mathcal{R}}[S_0] = \frac{1}{9^2} \Pr_{\mathcal{R}}[S_0].$$

Distribution $\mathcal{R}$

$$\rho \longrightarrow \rho_0$$

Fig. 3.3 Mapping a single restriction ρ to a restriction ρ_0 with bigger probability. The event (of picking) ρ is depicted as a smaller size circle than the event (of picking) ρ_0, to represent their respective probabilities (weight or "mass" in the probability distribution $\mathcal{R}$).

(cont.)

But we need to do something similar not to merely a single assignment ρ, but rather all S'. To show that the sum of the probabilities of *all* of the restrictions in S' is at most $\frac{1}{4^s} \cdot$ (small number), we can try showing that *each* $\rho \in S'$ can be mapped to a *distinct* restriction ρ_0 as above.

If the mapping, denoted Θ, is done to *distinct* restrictions, that is, the mapping is *one-to-one* (*"injective"*), then we know that the sum of all probabilities of picking bad restrictions, namely the event of picking a restriction from S', is mapped to the event of picking a restriction in the *image* of Θ.

But because every event has probability smaller than one we finish. We explain it as follows:

$$\Pr_{\mathcal{R}}[S'] = \sum_{\rho \in S'} \Pr_{\mathcal{R}}[\rho] = \sum_{\rho \in S'} \left(\frac{1}{4^s} \cdot \Pr_{\mathcal{R}}[\Theta(\rho)] \right) = \frac{1}{4^s} \cdot \sum_{\rho \in S'} \Pr_{\mathcal{R}}[\Theta(\rho)].$$

Now, we'd like to claim that

$$\sum_{\rho \in S'} \Pr_{\mathcal{R}}[\Theta(\rho)] \leq 1$$

and finish. But this is not necessarily true for every sum of probabilities: if the map Θ is not one-to-one we may end up adding an event with a very large probability more than once, for example, hence exceeding one. The way to ensure...

(cont.)

But we need to do something similar not to merely a single assignment ρ, but rather all S'. To show that the sum of the probabilities of *all* of the restrictions in S' is at most $\frac{1}{4^s} \cdot$ (small number), we can try showing that *each* $\rho \in S'$ can be mapped to a *distinct* restriction ρ_0 as above.

If the mapping, denoted Θ, is done to *distinct* restrictions, that is, the mapping is *one-to-one* (*"injective"*), then we know that the sum of all probabilities of picking bad restrictions, namely the event of picking a restriction from S', is mapped to the event of picking a restriction in the *image* of Θ.

But because every event has probability smaller than one we finish. We explain it as follows:

$$\Pr_{\mathcal{R}}[S'] = \sum_{\rho \in S'} \Pr_{\mathcal{R}}[\rho] = \sum_{\rho \in S'} \left(\frac{1}{4^s} \cdot \Pr_{\mathcal{R}}[\Theta(\rho)] \right) = \frac{1}{4^s} \cdot \sum_{\rho \in S'} \Pr_{\mathcal{R}}[\Theta(\rho)].$$

Now, we'd like to claim that

$$\sum_{\rho \in S'} \Pr_{\mathcal{R}}[\Theta(\rho)] \leq 1$$

and finish. But this is not necessarily true for every sum of probabilities: if the map Θ is not one-to-one we may end up adding an event with a very large probability more than once, for example, hence exceeding one. The way to ensure...

$$\sum_{\rho \in S'} \Pr_{\mathcal{R}}[\Theta(\rho)] \leq 1$$

is by making sure Θ is one-to-one, hence we don't count probabilities more than once.

Or in other words, picking restrictions from Image($\Theta(S')$) is an *event* of our probability distribution, and so an event has always probability at most one.

Constructing the one-to-one map

The upshot of argument

- Our goal is to show there is a *one-to-one* mapping from bad restrictions $\rho \in S'$ to some *longer* restrictions ρ_δ.
  Bad: restriction whose $T(f \restriction \rho)$ has depth $> s$.
- Thus, we get that the event $\Pr_{\mathcal{R}}[\rho \in S'] \leq \frac{1}{4^s} \cdot$ (small).
- To show the mapping is *one-to-one*, we build a map from $\rho \in S'$ to longer restrictions ρ_δ such that given ρ_δ, we can *(uniquely) recover* ρ. (Hence, the map is one-to-one).
- The map is defined as encoding ρ by ρ_δ + *(small helper code)*.
- **Note:** If I know ρ_δ, it does not *a priori* mean I can recover ρ, because a different ρ' can be mapped to $\rho'_\delta = \rho_\delta$ (*the domain of 0/1 assignment in ρ and ρ' may be different*).
- The idea to recover ρ, given ρ_δ, is to map ρ to ρ_δ + helper code. The helper code will be small but allow us to recover ρ given ρ_δ.
- The helper code is built based on the following idea: Consider $\rho \in S'$, and let Π be the first (leftmost) path in $T(f \restriction \rho)$ exceeding length s.
  (We shall prune this path to length s).
- Then, given ρ_δ and the helper code, we will be able to recover which variables were assigned in ρ_δ by that path Π.
- This means we are able to distinguish between ρ_δ and ρ, hence recover ρ *(uniquely)*.

Proof of the Switching Lemma

Proof

Let $\rho \in S'$.
Let Π be the first path in $T(f_\rho)$ with length $> s$.

We use the following notation:

- $C_1, \ldots, C_k$: conjuncts.
- $\beta_1, \ldots, \beta_k$: unset variables.
- $\pi_1, \ldots, \pi_k$: assignments to β_i according to Π.
- $\Theta_1, \ldots, \Theta_k$: unique assignments to β_i that is consistent with $C_i|_{\rho,\Pi_1,\ldots,\Pi_{i-1}}$.

These terms in f were queried along Π, in order.

Helper Code

- β_i is coded by string β_i', of $|\beta_i|$ numbers, each $\leq 2r$ (recording for each var in β_i: its location in C_i (number $\leq r$), and whether it is the last var in β_i (one bit)).
- $\beta := \beta_1' \ldots \beta_k'$
- $|\{\rho_{\beta'} : \rho \in S'\}| \leq (2r)^s$.
- Π_i is coded as a string π_i' of $|\beta_i|$ bits.
- $\Pi := \Pi_1'\Pi_2' \ldots \Pi_k'$, where the number of possible Π' strings is 2^s.

Example of Coding

Example:

$$f := (x_1 \wedge x_2 \wedge x_4 \wedge \bar{x}_3) \vee (x_5 \wedge \bar{x}_2 \wedge x_4) \vee (\bar{x}_3 \wedge x_4 x_7) \vee (x_6 \wedge \bar{x}_5) \vee \ldots$$

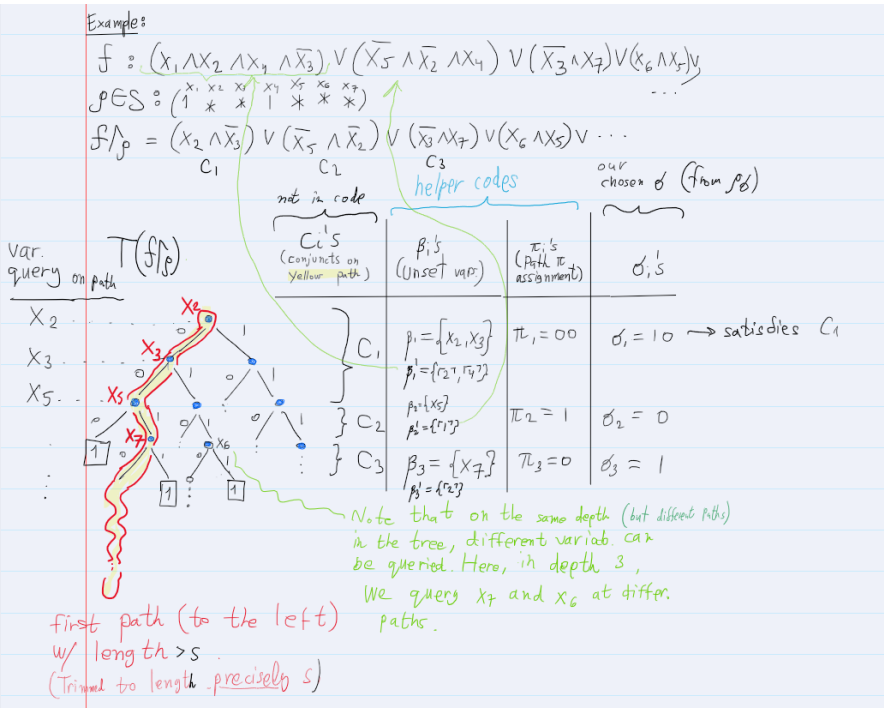
$$\rho \in S' : \quad x_1 \quad x_2 \quad x_3 \quad x_4 \quad x_5 \quad x_6 \quad x_7$$

$$\rho = (1 \quad * \quad * \quad 1 \quad * \quad * \quad *)$$

$$f_\rho = (x_2 \wedge \bar{x}_3) \vee (x_5 \wedge \bar{x}_2) \vee (x_6 \wedge x_5) \vee \ldots$$

Tree Representation $T(f_\rho)$

- First path (to the left) with length $> s$.
- Trimmed to length precisely s.



Coding Table

C_i's (Conjuncts on yellow path)	β_i (Unset variables)	π_i (Partial Π Assignment)	Θ_i
C_1	$\beta_1 = \{x_2, x_3\}$ $\beta_1' = \{r_2, r_7, r_7\}$	$\pi_1 = 00$	$\Theta_1 = 10$ $\Rightarrow$ satisfies C_1
C_2	$\beta_2 = \{x_5\}$ $\beta_2' = \{r_1, r_7\}$	$\pi_2 = 1$	$\Theta_2 = 0$
C_3	$\beta_3 = \{x_7\}$ $\beta_3' = \{r_7\}$	$\pi_3 = 1$	$\Theta_3 = 1$

Notes:

- On the same depth (but different paths) in the tree, different variables can be queried.
- Here, at depth 3, we query x_7 and x_6 at different paths.

Continuation

The Coding Map Θ is a function:

$$\Theta : S \to \mathcal{R} \times (2r)^s \times 2^s$$

defined by:

$$\Theta : \rho \mapsto (\rho\delta, \beta', \pi')$$

Notes

Only given $\rho\delta$, we don't know how to find ρ because we don't know what's the domain of ρ and what's the domain of $\rho\delta$. We only know the domain of $\rho\delta$.

Claim

Θ is one-to-one. Namely, given $(\rho\delta, \beta', \pi')$ we can recover (uniquely) ρ.
(Recall $C_1, C_2, \ldots$ are the clauses we query along the left-most path of length $> s$ in $T(f_\rho)$.)

Proof

We claim that this is an injection. To see this, suppose we are given $(\rho\delta, \beta', \pi')$. We may recover ρ as follows:

- First, we can easily recover all strings β'_i and π'_i.
- Now let C'_1 be the first term in F such that $C'_1|\rho\delta \neq 0$.
- C'_1 cannot come before C_1, and by construction C_1 is not falsified by $\rho\delta$.
- So we must have $C'_1 = C_1$.
- From β'_1 and C_1 we can recover β_1, and from this and π'_1 we can recover π_1.
- σ_1 and π_1 were assignments to the same variables, so we can construct a restriction $\rho\sigma_1[\pi_1/\sigma_1] = \pi_1\sigma_2 \ldots \sigma_k$.

- Let C_2' be the first term in F such that $C_2'|\rho\pi\sigma_2 \ldots \sigma_k \neq 0$.
- Then as above, C_2' must equal C_2, and we can recover β_2 and π_2 and carry on in the same way.

Once we have recovered all the β_i's, we know exactly what changed between ρ and $\rho\delta$ and can recover ρ.

claim

Note: $\rho\delta$ is not necessarily a bad restriction.

Note on defining sigma to be consistent with the conjuncts in F

The point is that we want to avoid a situation in which $C_i|\rho \not\equiv 0$ but $C_i|\rho\delta \equiv 0$. If this happens, then we can't know what was the clause queried by π_i; it might have been C_i, but because we ask for the first term in F such that $C_i'|\rho\delta \neq 0$, we don't get to C_i.

Decode

ρ is mapped to ρ_δ with helper codes $\beta_1', \beta_2', \beta_3', \ldots, \pi_1', \pi_2', \pi_3', \ldots$.

$$\rho_\delta : x_1, x_2, x_3, x_4, x_5, x_6, \ldots$$

$$1 \quad 1 \quad 0 \quad 1 \quad 0 \quad * \quad 1$$

We want to recover ρ given ρ_δ and helper codes.

- Consider the first AND clause C_1 in $f|\rho_\delta$ such that $C_1'|\rho_\delta \neq 0$.
- Since δ never falsifies an AND in f by **construction**, then $C_1' = C_1$ (i.e., the first AND such that $f|\rho \neq 0$).
- C_1' cannot come before C_1 (in f) because $C_1'|\rho_\delta \neq 0 \Rightarrow C_1'|\rho \neq 0$, since ρ_δ extends ρ.
- Thus, we know that C_1 was the first queried AND when we constructed $T(f|\rho_\delta)$.
- We know $\beta_1' = \{x_2, x_3\}$, the position in C_1 of the literals queried. So we can find out they were x_2, x_3. Hence, we also know π_1.
- So we can change ρ_δ: instead of δ, assigning x_2, x_3 assign π_1 to x_2, x_3. In symbols, we get:
$$\rho_\delta[\delta \to \pi_1]$$
- Now, we do the same decoding procedure with $T(f|\rho_\delta[\delta \to \pi_1])$, which is precisely the decision tree rooted at the third layer of the tree, left-most node (rooted at x_5).

- At the end, we know what is:

$$\rho_\delta[\delta \to \pi_1, \ldots, \delta_k \to \pi_k]$$

  Since we know all $\pi_1, \ldots, \pi_k$, we now know what was ρ: We take ρ_δ and assign stars to all variables queried on the path π_C.

We now can finish the proof of the Switching Lemma
Let Θ_1 be the projection of Θ to the first coordinate (ρ_δ).

When we fix helper codes, Θ_1 is one-to-one.

Explanation:

Fix some helper codes β_1', π_1' and let $S_{\beta_1', \pi_1'}$ be all $\rho \in S$ such that $\Theta(\rho) = (\rho_\delta, \beta', \pi')$ for some δ. That is, all bad assignments with helper codes β_1', π_1'.

Consider the projection of Θ to the first component:

$$\Theta : S \to \mathcal{R} \times (2r)^s \times 2^s \quad \text{(Recall definition)}$$

$$\Theta : \rho \mapsto (\rho_\delta, \beta', \pi')$$

$$\Theta_1 : \rho \mapsto \rho_\delta$$

By Claim, $\Theta_1 \circ S_{\beta_1', \pi_1'} \to \mathcal{R}$ is one-to-one.

Bounding Probability

We want to bound from above the probability that a *random* restriction in $S_{\beta_1', \pi_1'}$ is chosen.

Namely, bound

$$\Pr_{\mathcal{R}}[\rho \in S_{\beta_1', \pi_1'}]$$

By definition:

$$\Pr_{\mathcal{R}}[\rho \in S_{\beta_1', \pi_1'}] = \sum_{\rho \in S_{\beta_1', \pi_1'}} \Pr_{\mathcal{R}}[\rho] = \sum_{\rho \in S_{\beta_1', \pi_1'}} \left(\frac{2p}{1-p}\right)^s \Pr_{\mathcal{R}}[\Theta_1(\rho)]$$

i.e., the probability (in $\mathcal{R}$) of choosing the past $\Theta_1(\rho)$, which is ρ_δ for some δ of size s with helper codes β_1', π_1'.

Probability of ρ_δ

Since ρ_δ sets exactly s variables that were unset in ρ, we have:

$$\Pr_{\mathcal{R}}[\rho_\delta] = p^{-s} \cdot \left(\frac{1-p}{2}\right)^s \cdot \Pr_{\mathcal{R}}[\rho]$$

Unset S means setting the s variables that you just unset to p values.

By factoring out $\left(\frac{2p}{1-p}\right)^s$ we get from (1):

$$\Pr\left[\rho \in S_{\beta',\pi'}\right] = \sum_{\rho \in R} \left(\frac{2p}{1-p}\right)^s \Pr_R[\Theta(\rho)] = \left(\frac{2p}{1-p}\right)^s \sum_{\rho \in S_{\beta',\pi'}} \Pr_R[\Theta(\rho)]$$

But $\sum_{\rho \in S_{\beta',\pi'}} \Pr_R[\Theta(\rho)]$ is a probability of an event,

namely it is ≤ 1, because for $\rho \in S_{\beta',\pi'}$, we never repeat the same restriction, namely, if $\rho \neq \rho' \in S_{\beta',\pi'}$, then $\Theta(\rho) \neq \Theta(\rho')$.

(as was discussed above; otherwise we may have counted more than once the same restriction in $\sum_{\rho \in S_{\beta',\pi'}} \Pr_R[\Theta(\rho)]$).

So $\sum_{\rho \in S_{\beta',\pi'}} \Pr_R[\Theta(\rho)] \leq 1$ and we get:

$$\Pr\left[\rho \in S_{\beta',\pi'}\right] \leq \left(\frac{2p}{1-p}\right)^s.$$

Finally, S is the union of the sets $S_{\beta',\pi'}$ over all possible strings β', π':

$$\Pr[\rho \in S] \leq (2r)^s \cdot 2^s \cdot \left(\frac{2p}{1-p}\right)^s$$

$$= \left(\frac{8pr}{1-p}\right)^s \leq (9pr)^s, \quad \text{for } p < \frac{1}{q}.$$

□Switching Lemma

Chapter 4
PROOF COMPLEXITY

We've seen **computational** complexity. There we wished to understand the resources required to compute a function. And specifically, what is the minimal size of a Boolean circuit computing a given function (as a function of the number of input bits).

We now shall consider **proof complexity**. Instead of analysing the size of circuits computing Boolean functions, we address the question:

Given a propositional proof system, and a tautological statement (valid statement; namely a propositional formula that is 1 under any assignment), what is the shortest proof of this statement?

Like circuit complexity was motivated by P vs NP (if SAT does not have small circuits then $P \neq NP$), proof complexity is motivated by NP vs $coNP$ question.

NP: all languages with polynomial-size witnesses.

coNP: all languages whose *negative* instances have polynomial-size witnesses.

Formally:

Recall

$L \in NP$: There is a relation $R(\bar{x}, \bar{y})$ in P and a constant C, such that $x \in L$ iff

$$\exists \bar{y}(|\bar{y}| \leq |\bar{x}|^C \wedge R(\bar{x}, \bar{y}) = 1).$$

$L \in coNP$: Let

$$\bar{I} = \{x \in \{0,1\}^* \mid x \notin L\}$$

. Then $L \in coNP$ iff $\bar{I} \in NP$.

In other words, there is a relation $R(\bar{x}, \bar{y})$ in P and a constant C, such that $x \in L$ iff

$$\forall \bar{y}(|\bar{y}| \leq |\bar{x}|^C \rightarrow R(\bar{x}, \bar{y}) = 0).$$

For concreteness, we can think of $NP, coNP$ as:

$$SAT \sim NP$$
$$UNSAT \sim coNP$$

Given a CNF formula, accept if there exists a satisfying assignment
Indeed:

Given a CNF formula, accept if there is no satisfying assignment.

SAT is NP-complete, hence,

$UNSAT$ is coNP-complete.

- For language L in NP, we know we have short (polynomial-size) proofs for every x in L: if $L = SAT$, then the short proof is simply the satisfying assignment.
- For language L in coNP, we don't know if x in L have short proofs. If $L = UNSAT$, and x is in L, then what kind of short proofs are there for unsatisfiability? It's open if there are!
- To study this question we investigate proofs in different proof systems (like circuits in different circuit classes).

Example of a Concrete Proof System: Resolution Proof System

- Start from an unsatisfiable CNF $K = C_1 \wedge \cdots \wedge C_m$, where each C_i is a clause.
- A *resolution refutation* of K is a sequence of clauses $D_1, \ldots, D_t$ such that $D_t = \square$ (the empty clause, false), and every D_i is either C_i for some $i = 1, \ldots, m$, or was derived from D_j, D_k with $j, k < i$ via the resolution derivation rule:

Resolution Rule:

From $A \vee x$ and $B \vee \neg x$
Derive $A \vee B$

Where A, B are some clauses themselves.
Weakening Rule:

From A (a clause)
Derive $A \vee l$ (l a literal)

The **size** of a resolution refutation is the number of clauses it has.
Example:

- As a sequence of clauses, we can depict the same refutation as follows:

$x_1 \vee x_2 \vee x_3$, $x_4 \vee x_2 \vee x_3$, $x_1 \vee x_2 \vee \neg x_4$, $x_1 \vee x_3$, $x_3 \vee x_4$, x_3, x_1, False

Axioms and derivations:

- 1, 2 (axioms)

- Derive from 1, 2
- Axiom
- Derive from 4, 3
- Axiom
- Derive from 5, 6
- Axiom
- Derive from 7, 8

Comments: The example depicted a **tree-like** resolution refutation because the underlying structure is a **tree**. Formally, this means that every clause in the refutation is used at most once in a resolution rule (to derive a new clause). Hence, the underlying structure of the proof can be drawn as a **tree**.

A resolution refutation that is **not** a tree, namely, the same occurrence of a clause can be used in more than one resolution-rule applications, is called a **DAG-like refutation**. Because the refutation can be drawn as a **Directed Acyclic Graph**.

Comment: We use the word "proof" and "refutation" interchangeably in proof complexity.

A refutation of a CNF is a proof that it is unsatisfiable. Or in other words, a proof that the negation of the CNF is a tautology, i.e., evaluates to 1 under any assignment.

Theorem 4.1 *Resolution is a complete refutation system for unsatisfiable CNF formulas.*

Proof (Completeness of Resolution Refutation)

Step 1: Assume F is an Unsatisfiable CNF Formula

Let F be a CNF formula that is **unsatisfiable**. This means that every truth assignment to the variables of F leads to a contradiction. Our goal is to show that the empty clause (denoted as $\square$) can be derived using **resolution**.

Step 2: Construct a Resolution Proof by Induction on the Number of Variables

We use an **inductive argument** on the number of variables $n + 1$ in F:

- **Base Case (1 variable):** If F contains just one variable, say x_1, and is unsatisfiable, then it must contain both x_1 and $\neg x_1$ as unit clauses.
- Resolving these two immediately yields the empty clause $\square$.
- **Inductive Step:** Assume that resolution is complete for CNFs with n variables.
- Now, consider a CNF F with $n + 1$ variables.
- We perform the **splitting method** by considering:
  - $F[x_{n+1} = 1]$, the simplified formula when x is set to true.
  - $F[x_{n+1} = 0]$, the simplified formula when x is set to false. Since F is unsatisfiable, both $F[x_{n+1} = 1]$ and $F[x_{n+1} = 0]$ must also be unsatisfiable.
  - By the **induction hypothesis**, there exist resolution refutations for both. We can glue these refutations into one, as is shown in the Glueing Lemma below, by resolving on x_{n+1}, eventually leading to $\square$.

Step 5: Conclusion

By induction, we have shown that if F is unsatisfiable, then we can derive the empty clause $\square$ using resolution. Therefore, the resolution refutation system is **complete** for unsatisfiable CNFs.

Lemma: The resolution refutation of $F_{x_i:=0} \vdash \square$ and the resolution refutation of $F_{x_i:=1} \vdash \square$ can be glued into a refutation of F.

This suffices for our proof of completeness, when we put $i = n + 1$.

Proof of Lemma:

Assume $F_{x_i:=0} \vdash \square$, meaning that we have a resolution refutation of $F_{x_i:=0}$.

Then, $F \vdash x_i$, meaning there is a resolution derivation of x_i from F.

Consider $F_{x_i:=0} \vdash \square$.

- We are going to turn this refutation from $F_{x_i:=0}$ to a **derivation** of x_i from F.

This is done by, roughly, adding OR of x_i to each clause in the refutation $F_{x_i:=0} \vdash \square$.

- How do we do this precisely? Consider three types of clauses in F:

1) **Clauses containing literal x_i:** We denote this set of clauses by F_1.

2) **Clauses containing literal $\neg x_i$:** We denote this set of clauses by F_2. (F_1 is disjoint from F_2, because we assume we donâĂŹt have clauses of the form $C' \vee x_i \vee \neg x_i$, since we can get rid of such clauses without increasing the size).

3) **Clauses without x_i and without $\neg x_i$:** We denote this set of clauses by F_3. "'

Now consider what happens to $F = F_1 \cup F_2 \cup F_3$ when we assign $x_i = 0$ in $F_{x_i:=0}$:

1. A clause in F_1 looks like $x_i \vee D$.
   - So in $F_{x_i:=0}$, it turns into D.
2. A clause in F_2 contains $\neg x_i$.
   - So in $F_{x_i:=0}$, this clause becomes True and is erased.
3. A clause in F_3 does not contain the variable x_i, so in $F_{x_i:=0}$ it stays the same.

Now we are ready to convert $F_{x_i:=0} \vdash \square$ to $F \vdash x_i$ (i.e., a derivation of the clause x_i from F).

Idea:

Simply add $\vee x_i$ to each clause in $F_{x_i:=0} \vdash \square$.

It remains to show that doing so turns $F_{x_i:=0} \vdash \square$ into $F \vdash x_i$.

1. A clause D in $F_{x_i:=0}$ was the clause $D \vee x_i$ in F,
   - which when assigning $x_i = 0$ turned into D. Now, when we add $\vee x_i$, we get back to the original clause $D \vee x_i$ in F.
2. A clause $D \vee \neg x_i$ in F_2 has disappeared in $F_{x_i:=0}$, so it doesnâĂŹt appear in $F_{x_i:=0} \vdash \square$.
3. A clause D in F_3 does not contain x_i or $\neg x_i$. When we add $\vee x_i$ to D, we get a new clause not in F.
   - But we can derive, using a single **Weakening Rule**, the clause $D \vee x_i$.

- Hence, in every place $D \vee x_i$ appears, we simply add D to the resolution derivation, which turns it into a legal resolution derivation from F.

e) If in $F_{x_i:=0} \vdash \square$ we used the resolution rule:

$$\frac{A \vee u \quad B \vee w}{A \vee B}$$

for some variable $w \in \{x_1, \ldots, x_n\} \setminus \{x_i\}$ (*recall that variable x_i doesnâĂŹt occur in $F_{x_i:=0}$*), then when adding $\vee x_i$, it turns into a **legit** resolution rule:

$$\frac{x_i \vee A \vee u \quad x_i \vee B \vee w}{x_i \vee A \vee B}$$

$$x_i \vee A \vee B$$

$\square$ (Claim)

Illustration of the transformation from F to $F_{x_i:=0}$

$x_i \in F_1$	$x_i \in F_2$	$x_i, \neg x_i \notin F$
$(x_i \vee D_1), (x_i \vee D_2)$	$(\neg x_i \vee C_1), (\neg x_i \vee C_2)$	F_3
Original F	Original F	$F_{x_i:=0}$
$D_1, \ldots, D_k$		F_3
F_1		$F_3 \vee x_i$ (weakening)

By the claim, we get a resolution derivation π of x_i from F.

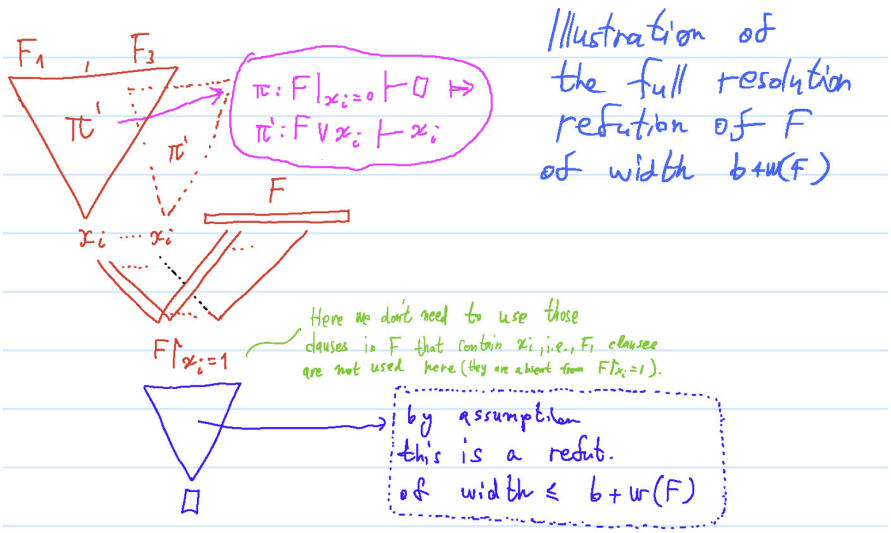
- Now, we use $F \vdash x_i$ on the clauses of F to get rid (resolve) all the literals $\neg x_i$ in F.

This is simple: derive x_i from F, and then resolve $D \vee \neg x_i$ in F by x_i to get D.

- Note that this way we got a derivation of $F_{x_i:=1}$ from F.

This is because $F_{x_i:=1}$ contains precisely all the clauses in F that contain neither x_i nor $\neg x_i$ (F_3 above), and the clauses D such that $x_i \vee D$ is in F (i.e., F_2 above). (The clause $x_i \vee D$ in F gets deleted in $F_{x_i:=1}$.)

Theorem 4.2 *By assumption $F_{x_i:=1} \vdash \square$, namely, after we derived $F_{x_i:=1}$ from F with a resolution derivation, we derive $\square$, and finish the proof of the theorem.*



4.1 Propositional Proof Systems

Definition 4.1 (Propositional Proof System) A propositional proof system is a polynomial-time algorithm $A(\tau, \pi)$ with two inputs: - A tautology τ (a propositional formula that is satisfied by every assignment) or equivalently an unsatisfiable propositional formula, - Such that:

$$\exists \pi \in \Sigma^*(A(\tau, \pi) = 1) \text{ iff } \tau \text{ is a tautology.}$$

Important note: $A(\tau, \pi)$ is a poly-time algorithm for checking the **correctness** of the proof π of τ.

- The runtime of A is **not** polynomially bounded in the size of τ, rather in the size of **both** τ and its proof π! - It might happen that some tautologies τ do not have proofs π with size polynomial in τ in any propositional proof system (thatâĂŹs open! And would imply $P \neq NP$).

Intuition behind the definition of propositional proof systems: - A "proof" is something that can be **easily checked** once given. - "Easily" in computational complexity means polynomial-time. - Make sure that you understand the following distinction:

1. Checking the correctness of a given proof π.
2. Finding a proof π given τ.

We use the following terminology:

Completeness: A propositional proof system is complete because if τ is a tautology, then there exists a proof π. That is, **all** tautologies have proofs.

Soundness: A propositional proof system is sound because if τ has a proof, then τ is a tautology. That is, **only** tautologies have a proof.

Similar to circuit complexity, we consider the **asymptotic behavior** of proof systems:

A **family** of tautologies $\{\tau_n\}_{n=1}^{\infty}$ is a collection of propositional formulas τ_n such that τ_n is a tautology with n variables for all $n \in \mathbb{N}$.

Similarly, we consider a **family** of proofs for $\{\tau_n\}_{n=1}^{\infty}$.

Let P be a propositional proof system.

A **family** of P-proofs $\{\pi_n\}_{n=1}^{\infty}$ of the family of tautologies $\{\tau_n\}_{n=1}^{\infty}$ is a collection of P-proofs, such that π_n is a P-proof of τ_n for all $n \in \mathbb{N}$ (with n being the number of variables in π_n and τ_n).

Definition 4.2 A family of tautologies $\{\tau_i\}_{i=1}^{\infty}$ has **polynomial-size proofs** in a propositional proof system P, if there exists a family $\{\pi_i\}_{i=1}^{\infty}$ of proofs, and a constant c such that

$$|\pi_i| \leq |\tau_i|^c \quad \forall i.$$

A propositional proof system P is **poly-bounded** iff **any** family of tautologies $\{\tau_i\}_{i=1}^{\infty}$ has polynomial-size proofs.

Note: We sometimes restrict TAUT to the set of tautological DNFs; or (possibly) equivalently, the set of **UNSAT** CNFs.

Theorem 4.3 (Cook-Reckhow '79) *There is a poly-bounded propositional proof system* ***iff*** $\mathsf{NP} = \mathsf{coNP}$.

Proof:

If there is a Turing machine $A \in \mathsf{PTIME}$ and a constant c such that for all propositional formulas τ,

$$\exists \pi \text{ with } |\pi| \leq |\tau|^c, \quad A(\tau, \pi) = 1 \iff \tau \in \mathsf{TAUT}$$

then by the definition of the class NP, we get $\mathsf{TAUT} \in \mathsf{NP}$.

Since TAUT is coNP-complete, every problem in coNP can be reduced to TAUT, and thus, by $(*)$ it can be reduced to SAT. Namely, $\mathsf{coNP} \subseteq \mathsf{NP}$.

This also means that $\mathsf{NP} \subseteq \mathsf{coNP}$:

$$L \in \mathsf{NP} \Rightarrow L \in \mathsf{coNP} \subseteq \mathsf{NP} \Rightarrow L \in \mathsf{NP} \Rightarrow L \in \mathsf{coNP}.$$

Hence, $\mathsf{NP} = \mathsf{coNP}$.

Observe: Resolution is a propositional proof system under the abstract definition above, because the poly-time verifier $A(\tau, \pi)$ can be thought to simply check that π is a correct resolution refutation of τ.

Comment: Different propositional proof systems could be thought of as different verification algorithms A.

4.2 Size-Width Relations for Resolution

- We'll show that short resolution refutations $\Rightarrow$ small width resolution.

- This enables to lower bound the size of resolution refutations via lower bounding the width of resolutions.

Notations:

1. The size/length of a resolution refutation is the number of clauses in it.
2. Let $S_T(F \vdash \square)$ be the minimal size of a **tree-like** resolution refutation of F.
3. Let $S(F \vdash \square)$ be the minimal size of a **DAG-like** resolution refutation of F.
4. Let $w(F)$ be the **width** of a CNF F; namely, the maximal number of literals in a clause in F.
5. Let π be a refutation of F. Then the width of π, denoted $w(\pi)$, is the maximal width of a clause in π.
6. $W(F \vdash \square)$ is the minimal width of a refutation of F.

Theorem 4.4 (Main Theorem) *Let F be an unsatisfiable CNF with n variables. Then:*

1. $S_T(F \vdash \square) \geq 2^{(W(F \vdash \square) - w(F))}$
2. $S(F \vdash \square) \geq 2^{\Omega\left(\frac{(W(F \vdash \square) - w(F))^2}{n}\right)}$

4.3 Proof of Theorem 1

We prove that:

$$w(F \vdash \square) \leq \log_2(S_T(F \vdash \square)) + w(F).$$

Let $b \in \mathbb{Z}$ be the minimal value such that $S_T(F \vdash \square) \leq 2^b$. Proceed by induction on n (where n is the number of variables).

Base Case: If $b = 0$, then $S_T(F \vdash \square) \leq 2^0 = 1$. So the refutation is just $\square$. Thus,

$$0 \leq \log_2(1) + w(F).$$

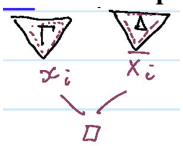
Case: If $n = 1$, then the only possible refutation is:

$$\frac{x_1 \quad \neg x_1}{\square}$$

And so $w(F) = 1$ and

$$1 - w(F \vdash \square) \leq \log_2(3) + 1.$$

Induction Step: The refutation must end with the following rule:



(W.l.o.g., assume that π is of size $\leq 2^b$.) Then by induction on b:

$$w(F_{x_i:=0} \vdash \square) \leq \log_2(S_T(F_{x_i:=0} \vdash \square)) + w(F) = b - 1 + w(F).$$

And by induction on n:

$$w(F_{x_i:=1} \vdash \square) \leq \log_2(S_T(F_{x_i:=1} \vdash \square)) + w(F) \leq b.$$

It suffices now to glue the two proofs together.
We need the following simple structural claims.
Notation: For a variable X we define $X^1 := X$ and $X^0 := \neg X$.

Definition 4.3 (Restriction of a proof/refutation) Let F be a CNF.
Define $F_{x_i:=\epsilon}$, $\epsilon \in \{0,1\}$ as follows:
For all clauses C in F do:

1. If C is a clause in F and $x_i^\epsilon \in C$, then delete C from F.
2. If $x_i^{1-\epsilon} \in C$, then delete the literal $x_i^{1-\epsilon}$ from C (keep C).
3. Otherwise, do nothing.

Examples:

1. $(x_1 \vee x_2 \vee x_3)$
$$F_{x_1:=0} \Rightarrow (x_2 \vee x_3)$$
2. $(x_1 \vee x_2 \vee \neg x_3)$
$$F_{x_1:=1} \Rightarrow (1 \vee x_2 \vee \neg x_3) \equiv 1$$
(Delete this clause because it is not usable in a refutation).

Conclusion: $F_{x:=\epsilon}$ for $\epsilon \in \{0,1\}$ is simply assigning $0,1$ in F for x, where $0,1$ then leads to deletion of literal or clause, respectively.

Lemma 4.1 *The refutation $F_{x_i:=0} \vdash \square$ of width $b - 1 + w(F)$ and the refutation $F_{x_i:=1} \vdash \square$ of width $b + w(F)$ can be glued into a refutation of F with width $b + w(F)$.*

Proof of Lemma:

Claim Assume $F_{x_i:=0} \vdash_{b-1+w(F)} \square$, meaning that from $F_{x_i:=0}$ we have a resolution refutation of width $\leq b-1+w(F)$. Then, $F \vdash_{b+w(F)} x_i$, meaning there is a resolution derivation of x_i from F of width $\leq b + w(F)$.

Proof of Claim:
Consider $F_{x_i:=0} \vdash \square$. By assumption, it has width $b - 1 + w(F)$.

- We are going to turn this refutation from $F_{x_i:=0}$ into a **derivation** of x_i from F of width $b + w(F)$ (i.e., we add one to the width). This is done by, roughly, adding $\vee x_i$ to each clause in the refutation of $F_{x_i:=0} \vdash \square$.

- **How do we do this precisely?** Consider three types of clauses in F:

1. **Clauses containing literal x_i:** We denote this set of clauses by F_1.
2. **Clauses containing literal $\neg x_i$:** We denote this set of clauses by F_2. (F_1 is disjoint from F_2, because we assume we do not have clauses of the form $C \vee x_i \vee \neg x_i$, since such clauses can be removed without increasing the size.)
3. **Clauses without x_i and without $\neg x_i$:** We denote this set of clauses by F_3.

Now consider what happens to $F = F_1 \cup F_2 \cup F_3$ when we assign $x_i := 0$ in $F_{x_i:=0}$:

1. A clause in F_1 looks like $x_i \vee D$. So in $F_{x_i:=0}$, it turns into D.
2. A clause in F_2 contains $\neg x_i$. So in $F_{x_i:=0}$, this clause becomes true and is erased.
3. A clause in F_3 does not contain the variable x_i, so in $F_{x_i:=0}$, it stays the same.

Now we are ready to convert

$$F_{x_i:=0} \vdash_{b-1+w(F)} \square$$

to

$$F \vdash_{b+w(F)} x_i$$

(i.e., a derivation of the clause x_i from F, of width $b + w(F)$).

Simply add $\vee x_i$ to each clause in $F_{x_i:=0} \vdash_{b-1+w(F)} \square$.

It remains to show that doing so turns

$$F_{x_i:=0} \vdash_{b-1+w(F)} \square \quad \text{into} \quad F \vdash_{b+w(F)} x_i.$$

1. Since we add one variable to each clause, the width is indeed $\leq b + w(F)$.
2. A clause D in $F_{x_i:=0}$ was the clause $D \vee x_i$ in F, which, when assigning $x_i = 0$, turned into D. Now, when we add $\vee x_i$, we get back to the original clause $D \vee x_i$ in F.
3. A clause $D \vee \neg x_i$ in F_2 has disappeared in $F_{x_i:=0}$, so it does not appear in $F_{x_i:=0} \vdash \square$.
4. A clause D in F_3 does not contain x_i or $\neg x_i$. When we add $\vee x_i$ to D, we get a new clause not in F. But we can derive, using a single **weakening rule**, the clause $D \vee x_i$. Hence, in every place $D \vee x_i$ appears, we simply add D to the resolution derivation, which turns it into a legal resolution derivation from F.

Cont. of Proof of "Gluing" Lemma

e) If in $F_{x_i:=0} \vdash \square$, we used the **resolution rule**:

$$\frac{A \vee v \quad B \vee w}{A \vee B}$$

for some variable $w \in \{x_1, \ldots, x_n\} \setminus \{x_i\}$ (*recall that variable x_i does not occur in $F_{x_i:=0}$*), then when adding $\vee x_i$, it turns into a **legit resolution rule**:

$$\frac{x_i \vee A \vee v \quad x_i \vee B \vee w}{x_i \vee A \vee B}$$

Illustration of the transformation from F to $F_{x_i:=0}$:

$x_i \in F_1$	$x_i \in F_2$	$x_i \notin F$
$(x_i \vee D_1), (x_i \vee D_2), \ldots$	$(\neg x_i \vee C_1), (\neg x_i \vee C_2), \ldots$	Original F
$D_1, \ldots, D_t$	Removed	F_3

Conclusion: By the claim, we obtain a resolution derivation π of x_i from F of width $b + w(F)$.

Cont. of Proof of "Gluing" Lemma

Now, we use $F \vdash_{b+w(F)} x_i$ on the clauses of F to get rid (resolve) all the literals $\neg x_i$ in F. This is simple: derive x_i from F, and then resolve $D \vee \neg x_i$ in F by x_i to get D.

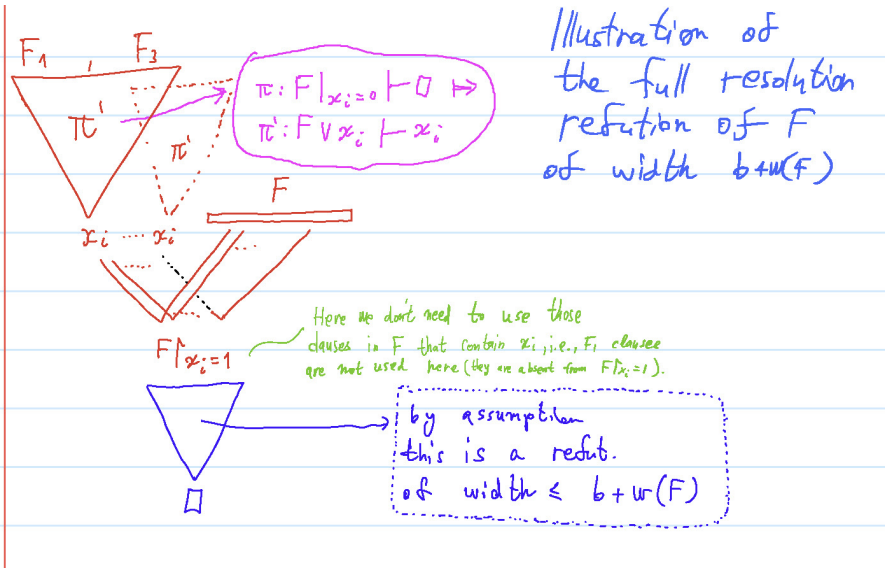
Note that this way we get a derivation of $F_{x_i:=1}$ from F, of width $b + w(F)$. This is because $F_{x_i:=1}$ contains precisely all the clauses in F that contain neither x_i nor $\neg x_i$ (F_3 above), and the clauses D such that $x_i \vee D$ is in F (i.e., F_2 above). (The clause $x_i \vee D$ in F gets deleted in $F_{x_i:=1}$.)

By assumption,

$$F_{x_i:=1} \vdash_{b+w(F)} \square,$$

namely, after we derived $F_{x_i:=1}$ from F with a resolution derivation of width $b+w(F)$, we derive $\square$, and finish the proof of the theorem.

$\square$ **Theorem**



We need the following simple structural claims.
Notation: For a variable x we define $x' := x$ and $x^0 := \neg x$.
Definition (Restriction of a proof/ref): Let F be a CNF.
Define $F_{x_i=\epsilon}$, $\epsilon \in \{0,1\}$ as follows:
For all clauses C in F do:

1. If C is a clause in F and $x_i^\epsilon \in C$, then delete C from F.
2. If $x_i^{1-\epsilon} \in C$, then delete the literal $x_i^{1-\epsilon}$ from C (keep C).
3. Otherwise, do nothing.

Examples

1.
$$(x_1 \vee x_2 \vee x_3)$$
$$\downarrow x_1 = 0$$
$$(0 \vee x_2 \vee x_3)$$
$$\downarrow \text{(simplified)}$$
$$(x_2 \vee x_3)$$

2.
$$(x_1 \vee x_2 \vee x_3)$$
$$\downarrow x_1 = 1$$
$$(1 \vee x_2 \vee x_3)$$
$$\downarrow \text{(delete it, since it's not usable)}$$

Conclusion: $F \vdash_{x=\epsilon}$ for $\epsilon \in \{0,1\}$ assigning 0 or 1 in F for x where $0,1$ then leads to deletion of literal, clause, respectively.

The Dag-like Refutation Case

Theorem 4.5 (Ben-Sasson, Wigderson 2001) *Let F be a CNF with $w(F) = k$.*
Then
$$w(F \vdash 0) \leq O\left(\sqrt{n \log(SIZE_{DAG}(F \vdash 0))}\right) + k.$$

Proof: Assume that π is the minimal resolution refutation of F. Let $|\pi| = s$.
If $s = 1$, then $\square \in F$ and we're done. Otherwise, $s > 1$.

- Let π^* be the set of *wide* clauses in π, meaning that they have more than d literals, for some d that we optimize later.
- Let
$$a = \left(1 - \frac{d}{2m}\right)^{-1}$$
(where m is the number of variables and b is from).
- We prove by induction on b, n that if $|\pi^*| \leq a^b$ then
$$w(F \vdash 0) \leq d + b + k.$$

Base Case: If $b = 0$, then $|\pi^*| < a^0 = 1$ and we're done. If $n = 1$, then $w(F \vdash 0) = 1$ and we're done.
Induction Step: By an *averaging* argument: because every wide clause has $> d$ literals, there exists a literal x_i (w.l.o.g.) that occurs in $\geq \frac{|\pi^*|d}{2m}$ wide clauses.

- *Kill* this literal by setting $x_i = 0$.
- We get a refutation $\pi_{x_i=0}$ of $F_{x_i=0}$ such that:

$$|\pi^*_{x_i=0}| < (a^b)\left(1 - \frac{d}{2m}\right) = \left(\frac{a^b}{a}\right) = a^{b-1}.$$

1. By the induction hypothesis, we thus get

$$w(F_{x_i=0} \vdash 0) \leq d + b + k - 1.$$

2. By the induction hypothesis, we also have

$$w(F_{x_i=1} \vdash 0) \leq d + b + k.$$

- Using the lemma from the previous page: (1) + (2) $\Rightarrow w(F \vdash 0) \leq d + b + k$.

It remains to prove the following:

Claim

$$d + b + k \leq O\left(\sqrt{n \log s}\right) + k, \quad \text{for } d = \sqrt{2n \log s}.$$

Proof (Proof of Claim)

$$d + b + k \leq d + \log_a \left(\frac{1}{1 - \frac{d}{2m}}\right) s + k.$$

Using the bound:

$$a^b \leq s \Rightarrow b \leq \log_a s = \frac{\log s}{\log a},$$

we obtain:

$$\leq d + \log\left(1 + \frac{d}{2n}\right) s + k.$$

Approximating:

$$\frac{1}{1 - \varepsilon} \approx 1 + \varepsilon, \quad \log_{1+\varepsilon} x \approx \frac{1}{\varepsilon} \log x,$$

we get:

$$\leq d + O\left(\frac{2n \log s}{d}\right) + k.$$

Substituting $d = \sqrt{2n \log s}$, we have:

$$= \sqrt{2n \log s} + O\left(\frac{2n \log s}{\sqrt{2n \log s}}\right) + k.$$

Simplifying:

$$= O\left(\sqrt{2n \log s}\right) + k.$$

$\square_{\text{claim}}$

$\square_{\text{thm}}$

References

1. Noga Alon and R. Boppana. The monotone circuit complexity of boolean functions. *Combinatorica*, 7(1):1–22, 1987.
2. Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, New York, NY, USA, 2009.
3. Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, Reading, Massachusetts, 1994.
4. Alexander A. Razborov. Lower bounds on the monotone complexity of some Boolean functions. *Dokl. Akad. Nauk SSSR*, 281(4):798–801, 1985. In Russian. English translation in *Soviet Math. Doklady*, 31:354-357, 1985.