Iddo Tzameret

Introduction to Concrete Complexity

Lecture Notes

Imperial College London
Dept. of Computing

January 28, 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),

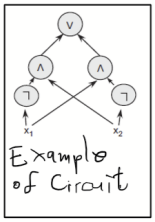
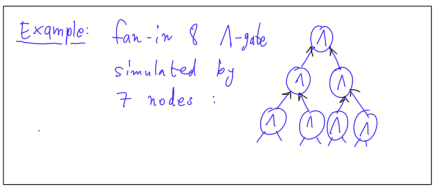


Fig. 1.1 Example of a simple Boolean circuit.

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?

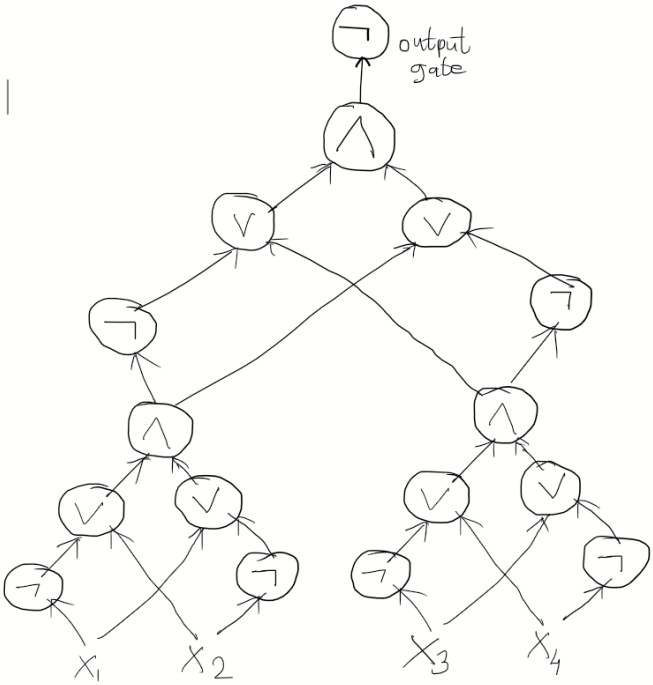


Fig. 1.2 Question

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

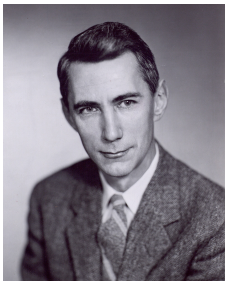


Fig. 1.3 C.E. Shannon. Source: By Unknown author - Tekniska Museet, CC BY 2.0, https://commons.wikimedia.org/w/index.php?curid=144716444

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.

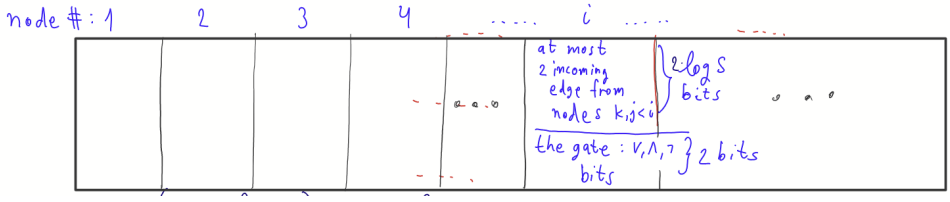


Fig. 1.4 The encoding scheme for a Boolean circuit of size S.

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:

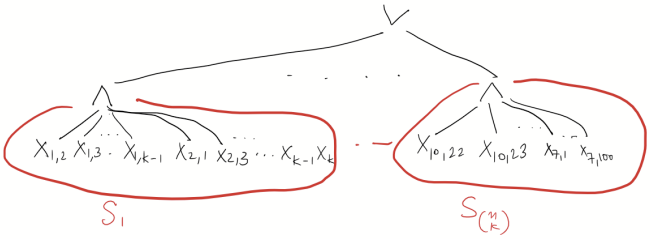


Fig. 2.1 Naive way to compute CLIQUE

$S_1, S_2, \ldots, S_{\binom{n}{k}}$ 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 computations 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. Throughout the proof, from now on we set:

$$k = \sqrt[4]{n}.$$

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

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. 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.)

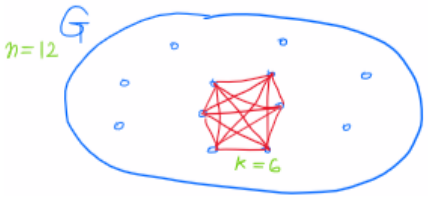


Fig. 2.3 Example of a *positive input*: a graph whose only edges are part of a (single) k-clique.

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).

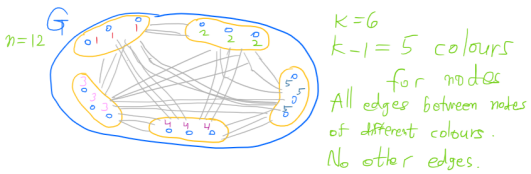


Fig. 2.4 Example of a *negative input* (i.e., $k-1$-colouring): a graph whose all and only edges are between distinct $(k-1)$-independent sets.

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 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, 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 (approx$(g_1) \circ$ approx$(g_2))(G) = 0$, namely, approx$(g_1) \circ$ approx(g_2) correctly identified G as a negative input, while 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 (approx$(g_1) \circ$approx$(g_2))(G) = 1$, namely, approx$(g_1) \circ$approx(g_2) correctly identified G as a positive input, while 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 negative instances 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.

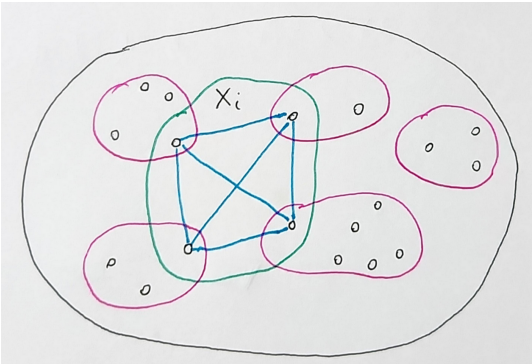


Fig. 2.5 Example of a $k-1$-colouring that fools an approximator that checks for a clique on the set X_i. The red sets are the independent sets, the green set is X_i and the blue edges are the clique formed on X_i that fools the approximator APX$(X_1, \ldots, X_m)$.

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 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 $\text{APX}(X_1, \ldots, X_m)(G) = 1$. □

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 a in Ω is a possible outcome of the random experiment, and is sometime called an *atomic outcome*, and holds a precise probability denoted $\Pr[a]$, such that:

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

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

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 introduces 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 $\mathrm{approx}(x_{ij}) = x_{ij}$ 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.

∨ and ∧-gates (induction) Approximator Step

Goal to prove: Let APX($\mathcal{X}$) and 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 APX($\mathcal{X}$)∨APX($\mathcal{Y}$), and APX($\mathcal{X}$)∧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: ∨-gate.

Naive (wrong) attempt: APX($\mathcal{X}$) ∨ APX($\mathcal{Y}$) is approximated by APX($\mathcal{X} \cup \mathcal{Y}$). That is, 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

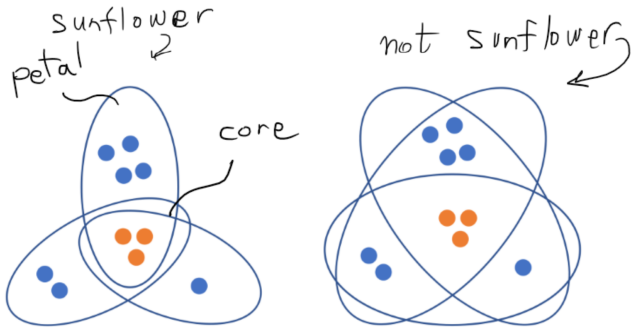


Fig. 2.6 From https://theorydish.blog/2021/05/19/entropy-estimation-via-two-chains-streamlining-the-proof-of-the-sunflower-lemma/ by Lunjia Hu

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 $\emptyset$).

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

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

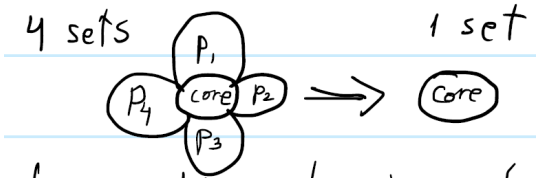
We know that Z' contains more than $M' = (p-1)^{l-1} \cdot (l-1)!$ sets. By *induction hypothesis* (since M' is "M with ℓ decreased by one"), Z' contains a sunflower denoted $\{P_1, \ldots, P_p\}$ with $|p_i| \leq \ell - 1$, for all i. Hence, $\{P_1 \cup \{d\}, \ldots, P_p \cup \{d\}\}$ is a sunflower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

[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.

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 $\text{pluck}(Z)$ 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 $\text{approx}(g_1) = \text{APX}(\mathcal{X})$ and $\text{approx}(g_2) = \text{APX}(\mathcal{Y})$. 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 $\text{approx}(g_1) = \text{APX}(\mathcal{X})$ and $\text{approx}(g_2) = \text{APX}(\mathcal{Y})$. 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 $\text{APX}(X_1, \ldots, X_m)$ and $\text{APX}\left(Y_1, \ldots, Y_{m_m}\right)$ and consider a false positive introduced by

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

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

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

while

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

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 $\left\{Z_1,\ldots,Z_p\right\}$ 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)).

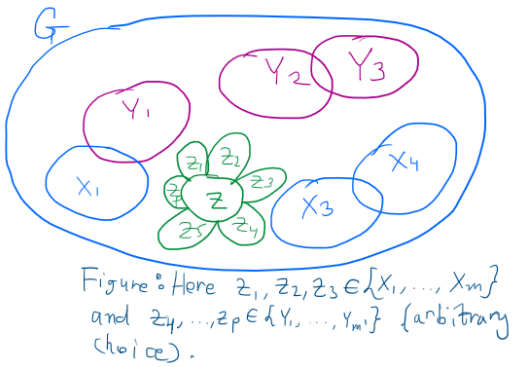


Fig. 2.7 Example

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.

[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.

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

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

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

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

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

(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 $\Pr[R(X)] \leq \frac{1}{2}$, 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.

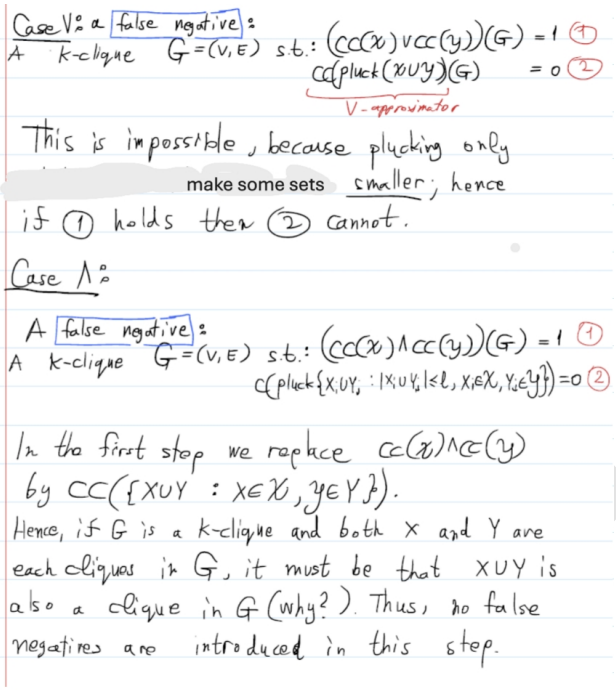
First, we form $\mathrm{APX}(\{X \cup Y : X \in \mathcal{X}, Y \in \mathcal{Y}\})$; this introduces no false positives, because any graph in which $X \cup Y$ is a clique must have a clique in both X and Y, and thus it was accepted by both constituent approximators.

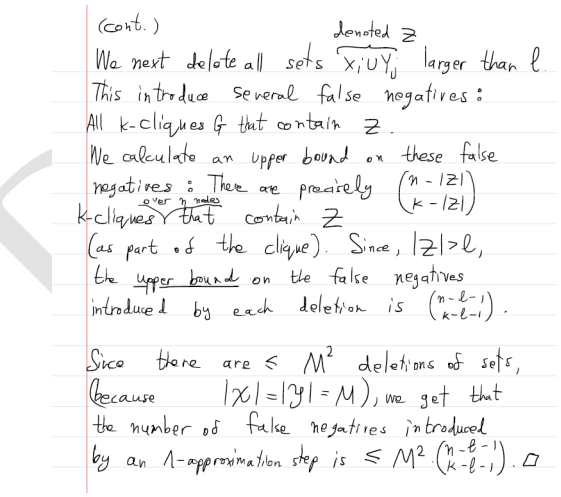
The second phase omits from the approximator circuit several sets (those of cardinality larger than ℓ), and can therefore introduce no false positives (because even less graphs are identified as having cliques).

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

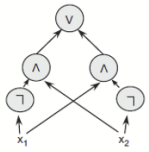




Chapter 3
Constant Depth Circuit Lower Bounds

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.

Note 3.1 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 $\{C_n\}_{n=1}^{\infty}$ with unbounded fan-in $\wedge, \vee$ gates and fan-in one negation gate is said to be

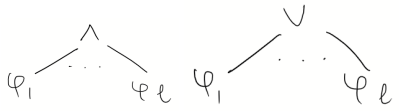


Fig. 3.1 Illustration of unbounded fan-in AND and OR gates.

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:

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*.

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:

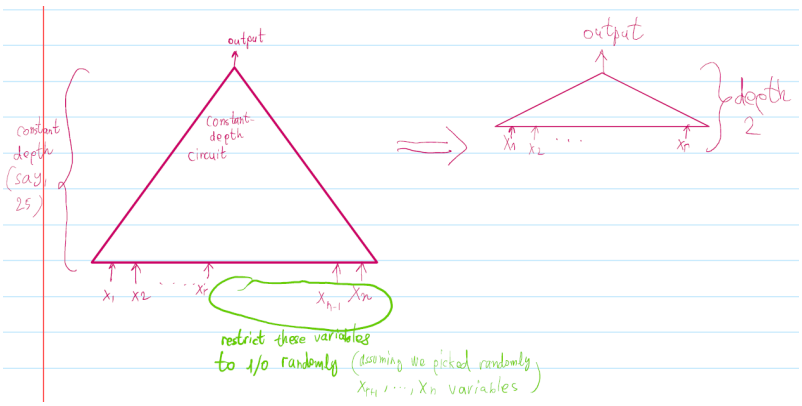
Theorem 3.1 (**AC^0 lower bound for** PARITY) *Every circuit family of depth-d computing the* PARITY_n *function requires size* $2^{\Omega\left(n^{1/(d-1)}\right)}$.

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 seuquel.

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 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 3.2 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.

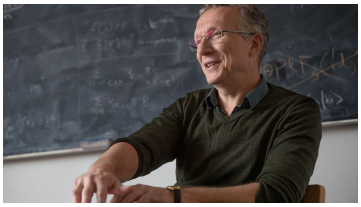


Fig. 3.2 Johan Håstad

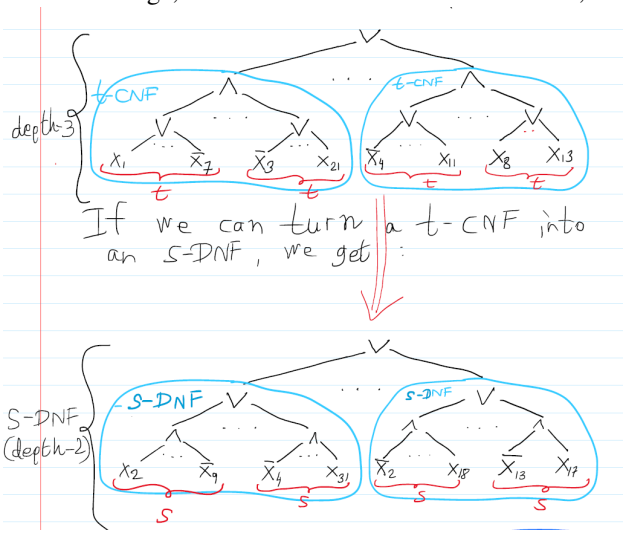
3.2.1 Depth Reduction Step: the Switching Lemma

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 3.3 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 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 3.4

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* $g : \bar{x} \to \{0, 1, *\}^n$ 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 6.

Switching via Random Restrictions

Notation:

1. An *assignment* ρ for l variables in $\bar{x}$ is a function $\rho : \bar{x} \to \{0, 1, *\}$, where $\rho(x_i) = *$ means that x_i is *unassigned* (or "free") by 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) *If f is an r-DNF with n variables 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 written 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" 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. We note that this would *not* be enough to conclude the argument. We shall also need to make sure that such a "good" random assignment exists moreover *leaves many variables free*. For this purpose, we shall use a common tool in probabilistic analysis and concrete complexity: *concentration bounds* (in our case it will be Chernoff Bound; see below).

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

Consider the bottom two levels of the given AC^0 circuit. Since we assume that the levels of AC^0 circuit always alternate between ANDs and ORs, the depth-2 sub-circuits at these bottom two levels are all either CNFs or DNFs. To apply the Switching Lemma to them, we need to make them into k-CNFs (or k-DNFs) first, for some *small* k, since the formulation of the lemma assumes k to be ... But why? what is the precise k t we need? We shall need it to be small for the probabilistic computation: namely, we need to bound the probability of failure, to be able to apply the union bound.

Bounding the bottom fan-in. Suppose these are all CNFs. We can view a given CNF ϕ on n variables as a depth-3 circuit AND of ORs of ANDs, with the bottom

AND gates of fan-in exactly 2, where each bottom AND gate takes as inputs two copies of the same variable. (For example, a CNF

$$(x \vee y) \wedge (z \vee w)$$

can be viewed as a depth-3 circuit

$$((x \wedge x) \vee (y \wedge y)) \wedge ((z \wedge z) \vee (w \wedge w)).)$$

Consider the bottom two levels of this depth-3 circuit. It is a 2-DNF. If we apply a random p_0-restriction with $p_0 = 1/20$, we get by the Switching Lemma that this 2-DNF will simplify (after the restriction) to a k-CNF, with all but at most 2^{-k} probability. Pick $k = 2 \cdot \log s$. Then the probability of not switching becomes at most $1/(s^2)$. We can merge these k-CNFs with the top AND gate of our original CNF ϕ, still getting a k-CNF after the merge.

Thus, after this p_0-restriction, we get each bottom CNF of our original depth-d AC^0 circuit is likely to become a k-CNF. By the Union Bound, the probability that some of at most s bottom CNFs does not become a k-CNF after a random p_0-restriction is at most $s \cdot 1/s^2 = 1/s$.

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.