Iddo Tzameret

Introduction to Concrete Complexity

Lecture Notes

Imperial College London
Dept. of Computing

January 13, 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),

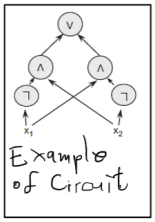
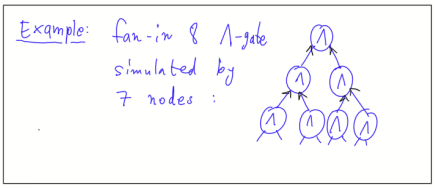


Fig. 1.1 Example of a simple Boolean circuit.

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?

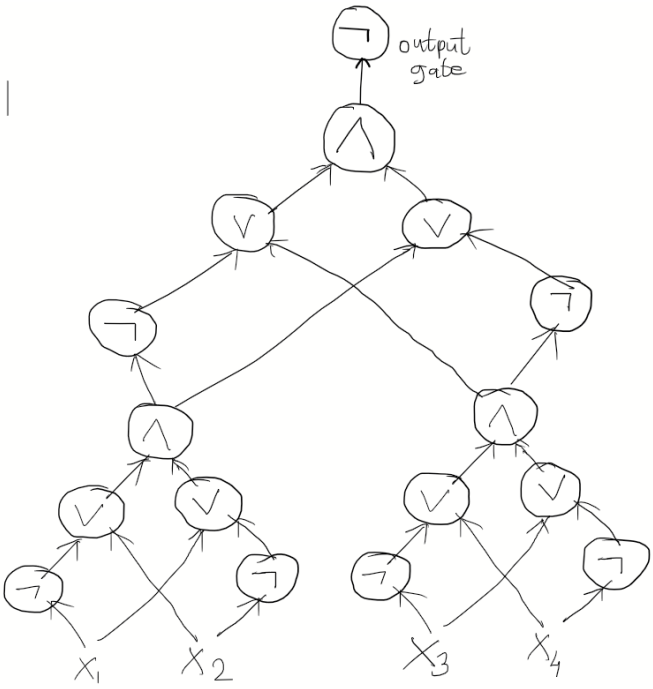


Fig. 1.2 Question

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

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).

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 $|C_n| \leq T(n), \forall n \in \mathbb{N}$.

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.

Slice of function. 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*.

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.

1. $f(n) = O(g(n))$ if there exist a 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* constant c and no $n_0 \in \mathbb{N}$ such that $f(n) \geq cg(n)$, for all $n \geq n_0$.

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., [1]).

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., [1, 2].

Definition 1.4 (polynomial-size circuits; $\mathsf{P/poly}$**)** The class $\mathsf{P/poly}$ 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; [1]]. □

1.4 Our main focus: lower bounds

Lower bounds, hard functions. We say that we have a *super-polynomial lower bound against* $\mathsf{P/poly}$, 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 $\mathsf{P/poly}$. 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

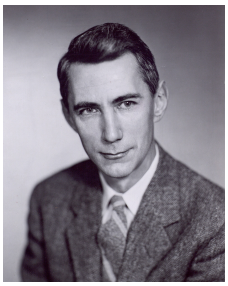


Fig. 1.3 C.E. Shannon. Source: By Unknown author - Tekniska Museet, CC BY 2.0, https://commons.wikimedia.org/w/index.php?curid=144716444

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.

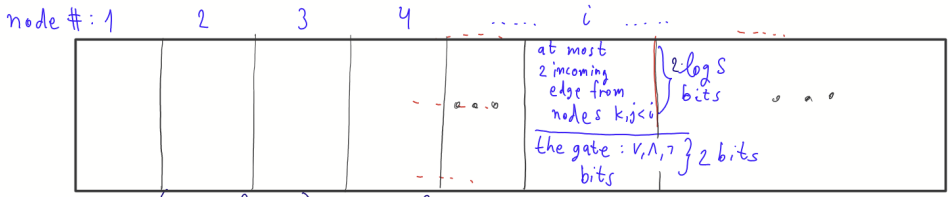


Fig. 1.4 The encoding scheme for a Boolean circuit of size S.

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 $f : \{0,1\}^n \to \{0,1\}$. □

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 $A \to B$. So $\neg(A \to B) \wedge \neg(B \to A)$ 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*.

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 $\forall \bar{x} \geq \bar{y}$, $f(\bar{x}) \geq f(y)$.

Many NP problems are monotone, like CLIQUE:

Given an undirected graph $G = (V, E)$ with n nodes, a k-clique in G is a set $U \subseteq V$ of size k, st. every pair of nodes $u_1, u_2 \in U$ is connected by an edge (in E):

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

Recall that a computational (decision) problem is a *language*, namely an infinite set of finite strings over a finite alphabet (usually the alphabet $\{0,1\}$). Here, our language consists of all the strings that encode (in some natural way) an accepted graph, i.e., a k-clique with n nodes. 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, as the following Boolean function:

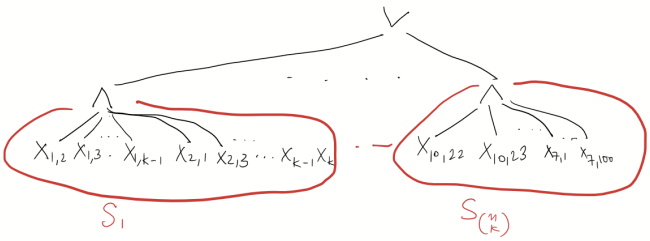
The computational problem **CLIQUE**(k, n):
Input: Undirected graph $G = (V, E)$ with n nodes, and a number k (given in unary, i.e., 1^k).
Accept: if the graph G contains a k-clique.
Reject: otherwise.

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.

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

"Run" over all $\binom{n}{k}$ k-sub-graphs in G, and check if at least one of those is a 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: $O\left(k^2 \cdot \binom{n}{k}\right)$.

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 a *crude-circuit* for CLIQUE(n, k). More formally, we have the following.

Crude-circuits $CC(X_1, \ldots, X_m)$ **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 *crude-circuit*

$CC(X_1, \ldots, X_m)$ is defined to be $\bigvee_{r=1}^{m} \bigwedge_{i<j \in X_r} x_{ij}$. Note that the X_i's may have different sizes (and specifically their sizes may be different from k).

Note that $CC(X_1, \ldots, X_m)(\alpha) = 1$, 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.

Note 2.1 When $k = \omega(\log n)$, $CC\left(S_1, \ldots, S_{\binom{n}{k}}\right)$, where $S_1, \ldots, S_{\binom{n}{k}}$ are all possible subsets of sizer k from the set of n nodes V, is of *exponential size*.

The following theorem shows this naive monotone circuit cannot be improved much.

Fig. 2.1 Alexander Razborov. Creator: Kozloff, Robert | Credit: Photo by Robert Kozloff. Copyright: The University of Chicago

Theorem 2.1 (Razborov [3])
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 from Papadimitriou's textbook** [2].

Approximation Method

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 crude circuit, namely a big OR of cliques, as follows.

1. Given a monotone circuit C , we shall construct a crude circuit $CC(X1, \ldots, Xm)$ for some m and $|X_i| \leq l$ (for some l, all $i = 1, \ldots, m$), that approximates CLIQUE(n, k) with **precision** that is dependent on the number of gates in C.
2. I.e., if the precision is not good, namely the crude circuit $CC(X1, \ldots, Xm)$ for CLIQUE (n,k) we end up with makes *many errors* on the CLIQUE (n, k) function (i.e., says "NO" on an input that has a k-clique, and "YES" on an input that has no k-clique), it means that the circuit C has *many gates*, and vice versa.
3. We show that every crude circuit $CC(X_1, \ldots, X_m)$ ($|X_i| \leq l$ 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 the number of gates in C was exponential.

The **approximation** (i.e., construction of a crude circuit for CLIQUE(n, k) given the circuit C) will proceed in steps, one step for each gate of the monotone circuit:

1. If C is a monotone circuit computing CLIQUE(n, k) we can *approximate* any gate OR or AND in C with a crude circuit.
2. Each such approximation step introduces rather few errors (false positives and false negatives).

2.1 Proof of monotone circuit lower bounds

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. We set:

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

Goal: Show that every monotone circuit computing CLIQUE(n, k) has size at least $2^{c\sqrt[8]{n}}$ for some constant c (for sufficiently large n).

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

Each crude-circuit we use in the approximation is:

$$CC(x_1, \ldots, x_m)$$

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

- The approximation of the monotone circuit C that computes CLIQUE(n, k) is done by induction on the size of C, ie., number of $\vee, \wedge$ gates in C.
- Comment: Such induction is also called "Induction on the structure of C".
- Such induction proceeds as follows:

Base Case

$|C| = 1$, ie., C consists of only a single input gate g_{ij}. Recall g_{ij} is an input gate denoting whether $(i, j) \in E$, for $i, j \in V$. That is, if there is an edge between i and j in the input graph G.

This is an easy case: We need to show a crude cat $\mathrm{CC}(X_1, \ldots, X_m)$ with $m \leqslant M$ and $|X_i| \leqslant l \quad \forall i \in [m]$ that approximates g_{ii} (without introducing too many errors; We shall count precisely the number of potential errors later). But the circuit $CC(\{i, j\}) = g_{ij}$ by definition. (Hence, no errors here!)

Induction Step

Given two crude circuits $\mathrm{CC}(\mathcal{X})$ and $\mathrm{CC}(\mathcal{Y})$, 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, $m' \leq M$ and $|Y_i| \leq \ell$, for all i.

We wish to construct another crude circuit for computing $CC(\mathcal{X}) \vee CC(\mathcal{Y})$, and $\mathrm{CC}(\mathcal{X}) \wedge \mathrm{CC}(\mathcal{Y})$.

Case 1: $\vee$-gate.

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

Solution: 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.

2.2 The Sunflower Lemma

Definition 2.3 (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 each 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.

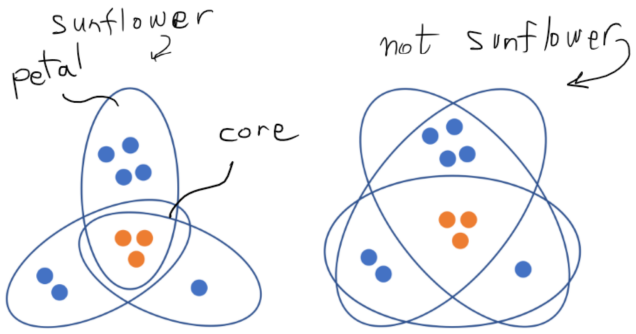


Fig. 2.2 From https://theorydish.blog/2021/05/19/entropy-estimation-via-two-chains-streamlining-the-proof-of-the-sunflower-lemma/ by Lunjia Hu

Note: It's 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 $\leqslant \ell$ (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 $\leqslant \ell$, and the intersection of every pair $P_i \neq P_j$ is fixed: $P_i \cap P_j = P_{i'} \cap P_{j'}$, for all $i \neq j \neq i' \neq j'$.

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 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 $|E| \leqslant (p-1) \cdot \ell$.

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 $> \frac{M}{(p-1)l} = (p-1)^{l-1} \cdot (l-1)!$ sets in Z.

Remark 2.1 If a set E intersect 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. □

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. This concludes the Sunflower Lemma's proof. □

Approximating OR and AND using Plucking - By the Sunflower Lemma, every family of $\geq M$ nonempty sets, each of cardinality $\leqslant l$, then we can find a sunflower in it with $l = \sqrt[3]{n}(p \approx \sqrt[3]{n})M = (p-1)^l \cdot \ell!$. - Plucking a sunflower: replacing all petals by their core. 4 sets

Corollary: If we have $> M$ sets in a family, by repeated plucking we can reduce the number of sets to $\leq M$ (if we cant apply plucking. anymore we know by the Sunflower Lemma that the number of sets is $\leqslant M$). pluck (z): the result of repeated plucking of a family of sets Z, until $|Z| \leqslant M$. Definition (Approximate OR and AND): foxily of sets The approximate $-V$ of two crude -circuits $CC(x)$ and cc(y) is: cc(pluck $(x \cup y)$).

The approximate $-\Lambda$ of $CC(x)$ and $CC(Y)$ is: cc(pluck $\left\{X_i \cup Y_j : \left|X_i \cup Y_j\right| \leqslant l \right.$ and $\left. X_i \in \chi, Y_j \in Y\right\}$)

False Positives & False Negatives

We shall now show that V and Λ-approximators are good, in the sense that they introduce few errors!

Out of all possible inputs to a cat computing Clique (n, k) (for $k = \sqrt[4]{n}$), the ne are: 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. We shall restrict attention t_0 only subsets of Accept and Rejed instances. This will be sufficient for the proof (it's more convenient that way). Thus, our focus is on "extreme" cases of inputs:

- Positive-lnputs: $G = (v, E)$ has x nodes and a k-clique (k nodes $w/$ all edges between them); while no other edge exist in G.

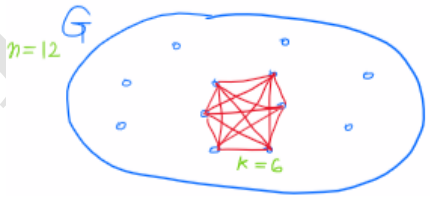


Fig. 2.3 Enter Caption

There are $\binom{x}{k}$ positive-inputs, each one is dearly an accept input to CUQ $\cup E(n, k)$. - Negative Inputs: $G = (v_1 E)$ has n nodes and $(k-1)$-colouring (independent sets). In other words, nodes V are coloured by some $k-1$ colours (one colour for each

independent set [ie., set of nodes w/ no edge between them]). And all edges between notes w/ different colours.

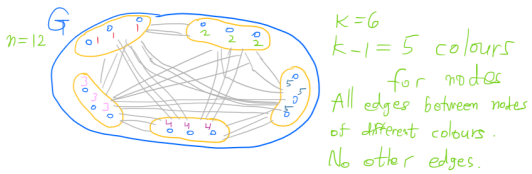


Fig. 2.4 Enter Caption

Note: 1) There are $(k-1)^n$ negative -inputs. (We count twice two identical graphs w/ colours interchanged.) 2) A $(k-1)$-colouring is a negative-input for CUQUE(n, k) because a k-clique cannot be coloured by $k-1$ colours. 3) In fact, even a single edge added to a $(k-1)$-colouring will make the graph contain a k-clique!

V-approximator of $\mathrm{CC}(x) \vee \mathrm{CC}(y)$ introduces 1) a false negative:

A k-clique $G = (v, E)$ s.t.: $(\mathrm{cc}(x)\,\mathrm{vcc}(y))(G) = 1$ 2) a false positive: cedpluck$(x \cup y)(G) = 0$ V-aterosimator $A(k-1)$-coburing $G = (v, E)$ s.t.

$$(\mathrm{cc}(x) \vee \mathrm{cc}(y))(G) = 0$$

$$\frac{\mathrm{cc}(pluck(x \cup y))(G)}{V\text{-approvinator}} = 1$$

1 -apporimator of $\mathrm{CC}(x) \wedge \mathrm{CC}(y)$ introduces 1) a false negative:

A k-clique $G = (v, E)$ s.t. : $(\mathrm{cc}(x) \wedge \mathrm{cc}(y))(G) = 1$ 2) a false positive: $cc\,(P \mid \mathrm{luck}\ \{x_i\,|y_i : |x_i \cup y_i| \leq l, x_i \in X, y_i \in\}) = 0$

$$cc\left(pluck\left\{x_i \cup Y_j : |x_i \cup y_0| \leq l, x_i \in \chi, y_i \in\right\}\right) = 1$$

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

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

Proof of monotone Pkt lower bound from these lemmas. CLAIM: Every crude-ctt CC $(X_1, \ldots, x_m)\, w/\ |x_i| \leq l$ and $m \leqslant M$ is either identically 0 (and thus is wrong on all positive-instances), or outputs 1 on at least half of the negative -instances. Proof: If CC $(x_1 \ldots x_m) \neq 0$, then it accept's at least those graphs G that have cliques on at least one of the sets X_i, for some i. The size of X_i is $\leqslant l < k$, thus many graphs G that are not k-cliques still have e-cliqnes

cont. Let's count how many such graphs exist, using probability. Consider a graph $G = (V, E)$; assign randomly (and independently) from among $(k-1)$ colours to the modes of G. Let $v_1 \neq v_2 \in V$ be two nodes.

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

Denote by $R(X_i)$ the evert that in the random colouring to G, there exists a pair of nodes with the same colour in X_i Then, $\Pr[R(x_i)] \leqslant \frac{\binom{x_i|}{2}}{k-1} \leqslant \frac{\binom{l}{2}}{k-1} \leqslant \frac{1}{2}$. there are $\binom{|x_i i}{2}$ pairs of nodes in X_i. We use the union bound. This means that out of all negative-instances to CLIQUE (n,k) (namely, all $(\ k-1)$-colouring of a graph w/ n nodes), at least half of them are going to colour X_i w/ different colours. Hence, for these negative -instance, there will be edges between all nodes in X_i, and thus $(C(x_1,\ldots,x_m)(G) = 1$ for each of these negative instances G.

We are now ready to conclude the main result. Recall: $l = \sqrt[8]{n}$ and Let $p = \sqrt[8]{n} \cdot \log n$. Thus, $M_i = (p-1)^l \cdot l! < \left(n^{\frac{1}{3}}\right)^{\sqrt[8]{n}}$, for large - Let C be a monotone cat computing CLQUE $(n, \sqrt[4]{n})$. - We apply the approximators at each gate of C iteratively. - The output gate of C thus is written as a $\mathrm{CC}(x_1,\ldots,x_m)$ for some $m \leqslant M$ and $|x_i| \leqslant l$. - Based on the above, we have two cases:

Case 1: $\mathrm{cc}(x_1,\ldots,x_m) \equiv 0$. Thus, the number of false negatives introduced is the total of all possible positive-instance, namely all possible $k-c$ cliques: $\binom{n}{k}$. Thus,

$$|C| \geq \frac{\binom{n}{k}}{\mathrm{lemmax}^2\binom{n-l-1}{k-l-1}} \geq \frac{1}{M^2}\left(\frac{n-l}{k}\right)^l \geq n^{c\sqrt[8]{n}},$$

Case 2 : $\mathrm{CC}(x_1,\ldots,x_n)$ has at least $\frac{1}{2}(k-1)^n$ false positives (half of $(k-1)$-colourings). By Lemma 2: $|c| \geq \frac{\frac{1}{2}(k-1)^n}{M^2\frac{(k-1)^n}{2^p}} = \frac{2^p}{2\cdot M^2} > n^{c\cdot 8}n$, for $c = \frac{1}{3}$.

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

Pe of F: Case 1: OR-approximator We start with $\overline{CC}(x_1,\ldots,x_m)$ and $\mathrm{CC}\left(y_1,\ldots,y_{m_m}\right)$ and consider a false positive introduced by

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

That is, a $G=(v,E)$ s.t.,

$$\mathrm{cc}(x_1,\ldots,x_m)(G) = 0, \quad \mathrm{cc}(y_1,\ldots,y_m)(G) = 0$$

and

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

We consider each plucking involved in (2), and bound from above the number of false positive introduced by this plucking. (Note this is the only reason a false positive can be introduced.)

- Consider a single plucking : replace sunflower $\left\{z_1,\ldots,z_p\right\}$ by its core Z.

By (1): $Y_i's\&X_i's(\&,'s)$ are all hon-cliques Thus, by (2) Z is a clique and every petal z_i has two nodes w/ the $\frac{\text{same colour!}}{L\,\mathrm{bg}(1)}$

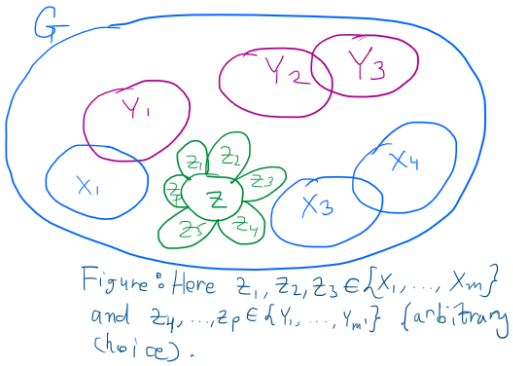


Fig. 2.5 Enter Caption

- We count the number of such colourings. - We do this probabilistically:

Choosing a $(k-1)$-colouring of the nodes randomly and independently, what's the probability every z_i has repeated colours but z does not? - As before, let $R(X)$ be the probability that X has repeated colours.

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

$$\text{pr}\left[R(z_1) \wedge \ldots \wedge R(z_p) \wedge \neg R(z)\right] \leqslant \text{pr}\left[R(z_1) \wedge \ldots \wedge R(z_p) \mid \neg R(z)\right]$$

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

$$\leqslant \prod_{i=1}^{p} \Pr\left[R(z_i)\right] \leqslant \frac{1}{2^p}$$

$$\Pr[A \| B] = \Pr[A \mid B] \cdot \Pr[B]$$

Li's don't have common nodes, except those in Z.

We've seen before that

$$\Pr[R(x)] \leqslant \frac{1}{2}$$

Probability of repetition of colours is increased if we doit restrict ourselves to colourings w/ no repetitions in $z \subseteq Z_i$.

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 2 M sets when we start), the lemma holds for the OR approximation step: because e

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

Cont. of proof of Lemma 1

Consider now an AND approximation step of crude circuits $CC(x)$ and $C(y)$. It can be broken down in three phases: First, we form $CC(\{X \cup Y : X \in \chi, Y \in Y\})$; this introduces no false positives, because any graph in which XUY is a clique must have a clique in both and Y, and thus it was accepted by both constituent crude circuits. The second phase omits from the approximator circuit several sets (those of cardinality larger than ℓ), and can therefore introduce no false positives. 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 $2^{-\mathrm{P}}(\mathrm{k}-1)^{\mathrm{n}}$ 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

Lemma 2: Each approximation step introduces $\leqslant M^2\binom{n-l-1}{k-l-1}$ false negatives. PRoOF: Case V.a false negative: A k-clique $G = (v, E)$ st.: $(\mathrm{cc}(x) \vee \mathrm{ccc}(y))(G) = 1$ (

$$\underbrace{\mathrm{colpluck}(x \cup y)(G)}_{V\text{-aproximator}} = 0$$

This is impossible, because plucking only deletes sets and make them smaller; hence if (1) holds then (2) cannot. Case 1: A false negative: A k-clique $G = (v, E)$ st: : $(\mathrm{cc}(x) \wedge \mathrm{cc}(y))(G) = 1$

$$c\left(P/|\ uck\left\{x_i \cup Y_j : |x_i \cup Y_i| \leq l, x_i \in \chi, Y_i \in\right\}\right) = 0 \ (2)$$

In the first stop we replace $ca(x) \wedge c(y)$ by $CC(\{X \cup Y : x \in X, y \in Y\})$. Hence, if G is a K-clique and both X and Y are each cliques in G, it must be that $X \cup Y$ is also a clique in $G(why\ ?\ )$. Thus, no false negatives are introduced in this step.

$\overbrace{\text{(cont.)}}^{\text{denoted}}$ larger than l. This introduce several false negatives:

We next delete all sets $\overbrace{X_i \cup Y_j}\, z$

All k-cliques G that contain Z. We calculate an upper bound on these false negatives: There are precisely $\binom{n-|z|}{k-|z|}$ k-cliques er that contain z (as part of the clique). Since, $|z| > l$, the upper bound on the false negatives introduced by each deletion is $\binom{n-l-1}{k-l-1}$.

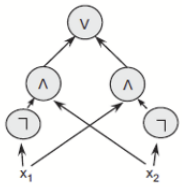
Since there are $\leqslant M^2$ deletions of sets, (because $|\chi| = |y| = M$), we get that the number of false negatives introduced by a_n1-approximation step is $\leqslant M^2 \cdot \binom{n-t-1}{k-l-1}$.

Chapter 3
Constant Depth Circuit Lower Bounds

3.1 Defining constant depth circuits

Recall that the depth of a circuit is the maximal length of a path from a leaf to the output node.

Here is an example of a circuit of depth 3:



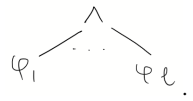
Recall that we are interested in the asymptotic study of circuit families, not a single circuit. That is, we want to consider how circuit size grows when the number of inputs n to a family of Boolean functions grows.

We can consider circuits of constant depth. This means that the depth of the circuit is *independent of the number of inputs* n. In other words, while n the number of input gates can grow to infinity in the Boolean circuit family $\{C_n\}_{n=1}^{\infty}$, the depth of each circuit C_n stays the same!

Note: If the depth of the circuit is constant & the fan-in of gates is at most 2, then the number of functions we can compute w/ 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. Thus, this model is *not* complete: for a constant d, a depth-d circuit cannot compute all Boolean functions over n inputs for every $n \in \mathbb{N}$.

Corollary: To make the model of const. depth meaningful we allow *unbounded* fan-in gates:

Unbounded fan-in AND ($\varphi_1 \wedge \varphi_2 \wedge \ldots \wedge \varphi_\ell$):



Unbounded fan-in OR ($\varphi \vee \cdots \vee \varphi_\ell$): Where φ_i's are circuits (of constant depth)

by themselves, with possibly *joint* nodes.

Important: When speaking about constant-depth circuits, we assume by default the fan-in of $\vee, \wedge$ is *unbounded*. ($\neg$ has fan-in one as always.)

To Be Completed!

References

1. Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, New York, NY, USA, 2009.
2. Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, Reading, Massachusetts, 1994.