

Iddo Tzameret

# Introduction to Concrete Complexity

Lecture Notes

Imperial College London  
Dept. of Computing

January 17, 2025

(C) 2025 Iddo Tzameret.

Copyright: Creative Commons (e.g., CC BY-NC-SA): Allows sharing and adaptation with non-commercial restrictions.



# Contents

0.1	Summary of Module .....	1
<b>1</b>	<b>Introduction to Circuit Complexity .....</b>	<b>3</b>
1.1	Basic Circuit Complexity .....	3
1.2	Basic concepts: circuit families, language recognition, and function computation .....	5
1.3	Quick recap: growth functions, $O$ -notation and run-times .....	6
1.4	Our main focus: lower bounds .....	7
1.4.1	Most functions are hard: Shannon lower bound .....	8
1.4.2	Answers .....	10
<b>2</b>	<b>Monotone Circuit Lower Bounds .....</b>	<b>11</b>
2.1	The CLIQUE Problem .....	11
2.1.1	Approximators .....	13
2.2	Clique is hard for monotone circuits .....	14
2.3	Proof of monotone circuit lower bounds Theorem 2.1 using the Approximation Method .....	15
2.3.1	The Input Set: Extreme Graphs, False Positive and Negatives .....	16
2.3.2	Concluding the Lower Bound using Structural Induction .....	18
2.3.3	Proofs of lemma 2.1 and lemma 2.2: the approximator steps .....	21
<b>3</b>	<b>Constant Depth Circuit Lower Bounds .....</b>	<b>27</b>
3.1	Defining constant depth circuits .....	27
	References .....	29



## 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  $f : \{0, 1\}^n \rightarrow \{0, 1\}$  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  $P \neq NP$ ), 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.



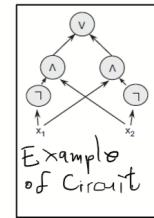
# Chapter 1

## Introduction to Circuit Complexity

### 1.1 Basic Circuit Complexity

**Definition 1.1 (Boolean Circuit)** Let  $n \in \mathbb{N}$  and  $x_1, \dots, x_n$  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.

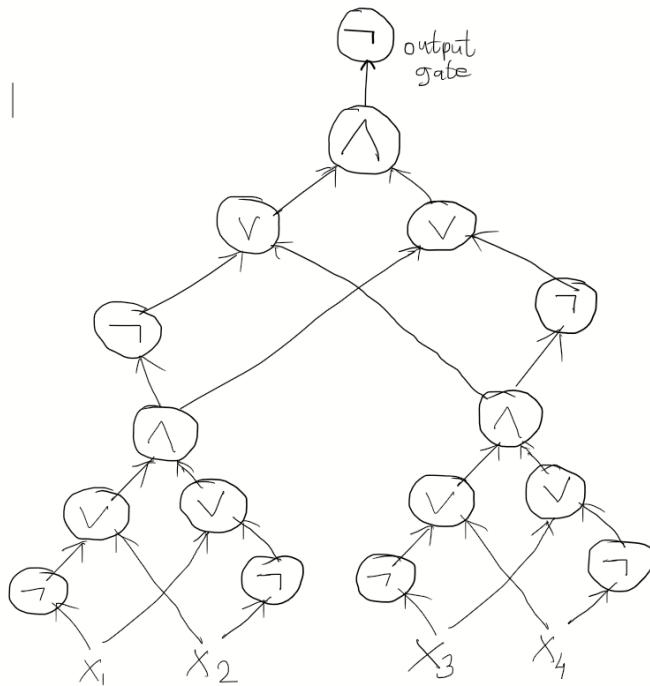


*Comment*

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  $T : \mathbb{N} \rightarrow \mathbb{N}$  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<sup>a</sup>.** Let  $f : \{0, 1\}^* \rightarrow \{0, 1\}$  be a Boolean function. We can consider the slice of size  $n$  of  $f$  to be the function  $f_n : \{0, 1\}^n \rightarrow \{0, 1\}$ , which is  $f$  restricted to inputs of length precisely  $n$ .

<sup>a</sup> 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  $\text{SIZE}(T(n))$ , namely,

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

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  $f : \{0, 1\}^* \rightarrow \{0, 1\}, f \in \text{SIZE}(T(n))$  if there exists  $T(n)$ -size circuit family  $\{C_n\}_{n \in \mathbb{N}}$  such that  $\forall n \in \mathbb{N} \ \forall x \in \{0, 1\}^n, f(x) = 1 \Leftrightarrow C_n(x) = 1$ . 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  $f_n : \{0, 1\}^n \rightarrow \{0, 1\}$ , 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  $f : \{0, 1\}^* \rightarrow \{0, 1\}$ .

### 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  $f, g : \mathbb{N} \rightarrow \mathbb{N}$ ).
2.  $f(n) = o(g(n))$  if  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$ . 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  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$ . 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  $2^{n^\delta}$ , 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  $f : \mathbb{N} \rightarrow \mathbb{N}$  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; P/poly)** The class P/poly is the class of languages that are decidable by polynomial-size circuit families. That is,  $P/\text{poly} = \bigcup_{c \in \mathbb{N}} \text{SIZE}(n^c)$ .

**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:  $P \subseteq P/\text{poly}$ .

**Proof** Assignment/tutorial. [See for a sketch in Arora-Barak'10, Sec. 6.1.1. page 110; [1]].  $\square$

## 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  $f : \{0, 1\}^* \rightarrow \{0, 1\}$  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  $f \notin \text{SIZE}(T(n))$ , we say that we have *proven a T(n)-lower bound for f* (against Boolean circuits). We say that this  $f$  is *hard* for  $\text{SIZE}(T(n))$ .

### 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  $n$ th slice  $f_n$  of a Boolean function  $f : \{0, 1\}^* \rightarrow \{0, 1\}$ , 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:  $\text{NEXP} \stackrel{?}{\subseteq} \text{P/poly}$ .

Notice that by the *time hierarchy theorem*  $\text{EXP} \not\subseteq \text{P}$  (that is, exponential time  $\text{EXP}$  is properly a bigger class than  $\text{P}$ ). Hence, clearly,  $\text{NEXP} \not\subseteq \text{P}$ . But when we consider the non-uniform version of  $\text{P}$ , namely,  $\text{P/poly}$ , we already do not know if every language in  $\text{NEXP}$  is also in  $\text{P/poly}$ . (We do know that there are languages in  $\text{P/poly}$  that are not in  $\text{NEXP}$ , because  $\text{P/poly}$  is not a uniform class, hence even undecidable languages are in  $\text{P/poly}$ , but not in the uniform class  $\text{NEXP}$ .)

#### 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  $f : \{0, 1\}^n \rightarrow \{0, 1\}$  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 *pigeon-hole 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  $f : \{0, 1\}^n \rightarrow \{0, 1\}$ . 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  $\{0, 1, 2, \dots, S-1\}$ . 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:

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

□<sub>claim</sub>

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  $f : \{0, 1\}^n \rightarrow \{0, 1\}$ .  $\square$

**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  $x_1, \dots, x_4$ . In other words, it computes the function  $x_1 + x_2 + x_3 + x_4 \bmod 2$ , 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 P  $\neq$  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 or 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  $\{1, \dots, n\}$ . (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  $f : \{0, 1\}^n \rightarrow \{0, 1\}$  is said to be *monotone* if  $\forall \bar{x} \geq \bar{y}, f(\bar{x}) \geq f(\bar{y})$ .

### 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  $\text{CLIQUE}(k, n)$  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.,  $1^k$ ).

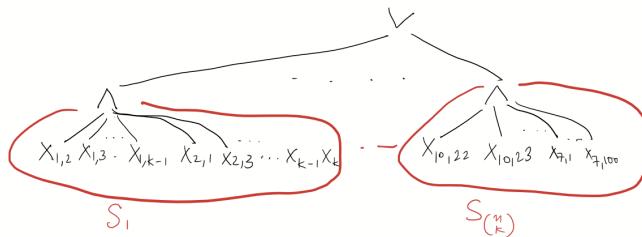
*Accept:* if the graph  $G$  contains a  $k$ -clique.

*Reject:* otherwise.

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

#### Example of a monotone circuit computing $\text{CLIQUE}(n, k)$

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

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

### 2.1.1 Approximators

A circuit for computing  $\text{CLIQUE}(n, k)$ , consisting of an OR gate  $\vee$  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  $\text{CLIQUE}(n, k)$ . More formally, we have the following.

**Approximators  $CC(X_1, \dots, X_m)$  for CLIQUE.** Let  $V$  be a set of nodes and let  $X_1, \dots, X_n \in V$  be a collection of subsets of nodes. The *approximator*  $CC(X_1, \dots, X_m)$  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  $CC(X_1, \dots, X_m)(\alpha) = 1$ , for  $\alpha \in \{0, 1\}^{\binom{n}{2}}$ , precisely when in the graph  $G$  over the nodes  $V$  described by the assignment  $\alpha$ , at least one of the  $X_i$  subgraphs is a clique.

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

#### Example of simple useful asymptotic computations

Note that when  $k \in (\Omega(\log n), O(n - \log n))$ ,  $CC(S_1, \dots, S_{\binom{n}{k}})$ , where  $S_1, \dots, S_{\binom{n}{k}}$  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:

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

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 [3])

Let  $k = \sqrt[4]{n}$ . Then, every monotone circuit computing  $\text{CLIQUE}(n, k)$  has size  $2^{\Omega(\sqrt[4]{n})}$ .

That is, there exists a constant  $c$  such that for large enough  $n \in \mathbb{N}$ , if  $C_n$  computes  $\text{CLIQUE}(n, k)$  then  $|C_n| \geq 2^{c \cdot \sqrt[4]{n}}$ .

The rest of this chapter aims to prove this theorem. *Our exposition is taken from Papadimitriou's textbook [2].*

### 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  $\text{CLIQUE}(n, k)$  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  $\text{CLIQUE}(n, k)$ , 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  $\text{CLIQUE}(n, k)$ .

More precisely, we have:

1. Given a monotone circuit  $C$ , we shall construct a approximator  $CC(X_1, \dots, X_m)$  for some  $m$  and  $|X_i| \leq l$  (for some  $l$ , and for  $i = 1, \dots, m$  ), that approximates  $\text{CLIQUE}(n, k)$  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  $CC(X_1, \dots, X_m)$  for  $\text{CLIQUE}(n, k)$  we end up with makes *few* mistakes on the  $\text{CLIQUE}(n, k)$  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  $\text{CLIQUE}(n, k)$  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  $\text{CLIQUE}(n, k)$  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  $CC(X_1, \dots, X_m)$  ( $|X_i| \leq l$  for some  $l$ , for all  $i = 1, \dots, m$  ), ought to make *exponentially many errors* on the function  $\text{CLIQUE}(n, k)$ . From 2 above we conclude that there exists no circuit  $C$  with a small number of gates.

#### Parameters & notation

Recall we want to compute  $\text{CLIQUE}(n, k)$  with  $n$  the number of nodes in the graph and  $k$  the size of a clique within the graph. We set:

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

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

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

$$CC(X_1, \dots, X_m)$$

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  $\text{CLIQUE}(n, k)$  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  $n$  nodes 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*.

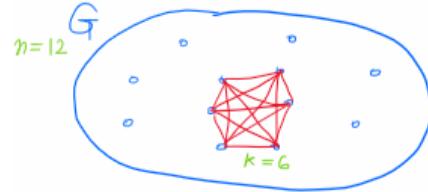
#### Extreme Graphs

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  $\text{CLIQUE}(n, k)$ .

Thus, our focus is on “extreme” cases of inputs:

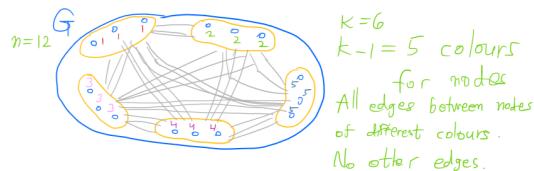
**Positive-inputs:**  $G = (V, E)$  has  $n$  nodes and a  $k$ -clique (i.e., a set of  $k$  nodes with all edges between them); while *no* other edge exist in  $G$  except for the  $k$ -clique. There are  $\binom{n}{k}$  positive-inputs, each one is clearly an accept input to  $\text{CLIQUE}(n, k)$ . 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*: 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. In fact, 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  $CC(X)$  be an approximator, with  $X$  a collection  $X_1 \dots, X_m \subseteq V$  of subsets of nodes. We say that a graph  $G(V, E)$  (with  $n$  nodes) is

a **false positive** if  $CC(\mathcal{X})(G) = 1$  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  $CC(\mathcal{X})(G) = 0$  while  $G$  is a  $k$ -clique.

### 2.3.2 Concluding the Lower Bound using Structural Induction

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

The approximation of the monotone circuit  $C$  that computes  $\text{CLIQUE}(n, k)$  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*** (*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

**Base step:** Input nodes  $x_{ij}$ . Build an approximator  $CC(x_{ij})$  (we shall see this in the next section).

**Induction step (the Approximator Step):**

**Case 1:** OR gate  $g_1 \vee g_2$ . By induction hypothesis we already built the approximators of  $g_1, g_2$ ,  $CC(g_1), CC(g_2)$ , respectively. We thus need to construct  $CC(g_1 \vee g_2)$  from  $CC(g_1), CC(g_2)$  (we shall do this in the sequel).

**Case 2:** AND gate  $g_1 \wedge g_2$ . By induction hypothesis we already built the approximators of  $g_1, g_2$ ,  $CC(g_1), CC(g_2)$ , respectively. We thus need to construct  $CC(g_1 \wedge g_2)$  from  $CC(g_1), CC(g_2)$  (again, we shall do this in the sequel).

We call the two cases of the induction step above *the approximator steps*. Namely, we have an approximator step for an OR gate and an AND gate, respectively.

**Introducing new false positives and false negative.** Let the graph  $G$  be a false positive of  $CC(g_1 \vee g_2)$  (i.e., a  $(k - 1)$ -colouring of which  $CC(g_1 \vee g_2)(G) = 1$ ). We say the approximator step **introduced  $G$  as a new false positive** if  $(CC(g_1) \vee CC(g_2))(G) = 0$ , namely,  $G$  is *not* a false positive of  $(CC(g_1) \vee CC(g_2))$ . Similarly, let the graph  $G$  be a false positive of  $CC(g_1 \wedge g_2)$ . We say the approximator step **introduced  $G$  as a new false positive** if  $(CC(g_1) \wedge CC(g_2))(G) = 0$ .

To prove the lower bound theorem, all we need to know are some properties of the approximator steps. 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.*

#### **Proof (Proof of Theorem 2.1 monotone circuit lower bound from Lemma 2.1 and Lemma 2.2)**

*Claim* Every approximator  $CC(X_1, \dots, X_m)$  with  $|X_i| \leq l$  and  $m \leq M$  is either identically 0 (and thus is wrong on all positive-instances), or outputs 1 on at least half of the (extreme) negative instances.  $\square$

**Proof** If  $CC(X_1 \dots X_m)$  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 (because the approximator evaluates to 1 when there is a  $|X_i|$ -clique on the nodes in  $X_i$ ).

Let us count how many such graphs exist, using *probability*, for simplicity. Consider a graph  $G = (V, E)$ ; 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 clearly have

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

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 because there are  $\binom{|X_i|}{2}$  pairs of nodes in  $X_i$ , and we use the union bound. The rightmost inequality holds because  $\frac{\binom{\frac{8n}{2}}{2}}{\sqrt[4]{n-1}} = \frac{\frac{8n}{2} \cdot \frac{8n-1}{2}}{\sqrt[4]{n-1}} \leq \frac{\frac{8n}{2} \cdot \frac{8n}{2}}{2\sqrt[4]{n}} = \frac{1}{2}$ .

This means that out of all *negative-instances* to CLIQUE  $(n, k)$  (namely, all  $(k-1)$ -colouring of a graph 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  $(C(x_1, \dots, x_m))(G) = 1$  for each of these negative instances  $G$ .  $\square$

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

We recall some basic notions from discrete probability, and explain more formally the union bound in what follows.

In discrete probability we consider a *Sample Space*  $\Omega$  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  $\Omega$  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.  $\sum_{a \in \Omega} \Pr[a] = 1$ .

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

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

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

Intuitively, the union bound tells us that the probability of at least one of the events  $E_1, E_2, \dots, E_n$  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.

Recall:  $l = \sqrt[8]{n}$  and let  $p = \sqrt[8]{n} \cdot \log n$ . Thus,  $M = (p - 1)^l \cdot l! < \left(n^{\frac{1}{3}}\right)^{\sqrt[8]{n}}$ , for large enough  $n$ .<sup>1</sup>

- Let  $C$  be a monotone circuit computing CLIQUE  $(n, \sqrt[4]{n})$ .
- We apply the approximators at each gate of  $C$  iteratively (using structural induction as above).

- The output gate of  $C$  thus is written as a CC  $(X_1, \dots, X_m)$  for some  $m \leq M$  and  $|x_i| \leq l$ .

- Based on the above, we have two cases:

**Case 1:** CC  $(X_1, \dots, X_m) \equiv 0$ . Thus, the number of false negatives introduced is the total of all possible positive-instance, namely all possible  $k$ -cliques:  $\binom{n}{k}$ .

Thus,

$$|C| \geq \frac{\binom{n}{k}}{M^2 \cdot \binom{n-l-1}{k-l-1}} \geq \frac{1}{M^2} \left(\frac{n-l}{k}\right)^l \geq n^{c\sqrt[8]{n}}, \text{ for } C = 1/12.$$

**Case 2:** CC  $(x_1, \dots, x_n)$  has at least  $\frac{1}{2}(k-1)^n$  false positives (half of  $(k-1)$ -colourings).

By Lemma XXX:  $|c| \geq \frac{\frac{1}{2}(k-1)^n}{M^2 \frac{(k-1)^n}{2^p}} = \frac{2^p}{2 \cdot M^2} > n^{c \cdot 8} n$ , for  $c = \frac{1}{3}$ .  $\square$

---


$$\begin{aligned} M &\leq (\sqrt[8]{n} \cdot \log n)^{\sqrt[8]{n}} \cdot \sqrt[8]{n}! \\ &\leq \dots \cdot \sqrt[8]{n}^{\sqrt[8]{n}} \\ &\leq (\sqrt[8]{n} \cdot \log n)^{\sqrt[8]{n}} < (n^{\frac{1}{3}})^{\sqrt[8]{n}} \end{aligned}$$

### 2.3.3 Proofs of lemma 2.1 and lemma 2.2: the approximator steps

#### Base Case

$|C| = 1$ , i.e.,  $C$  consists of only a single input gate  $g_{ij}$ . Recall that  $g_{ij}$  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 an approximator  $CC(X_1, \dots, X_m)$  with  $m \leq M$  and  $|X_i| \leq l$ , for all  $i \in [m]$  that approximates  $g_{ij}$  (without introducing too many errors; we shall count precisely the number of potential errors later). But the circuit  $CC(\{i, j\})$  is  $g_{ij}$  by definition. (Hence, no errors here.)

#### Induction step

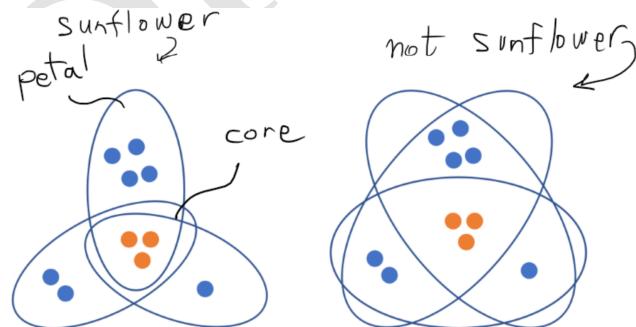
Given two approximators  $CC(\mathcal{X})$  and  $CC(\mathcal{Y})$ , with  $\mathcal{X} = \{X_1, \dots, X_m\}$ ,  $\mathcal{Y} = \{Y_1, \dots, Y_{m'}\}$ ,  $m \leq M$ , and  $|X_i| \leq \ell$ , for all  $i \in [m]$ ,  $m' \leq M$  and  $|Y_i| \leq \ell$ , for all  $i \in [m']$ , we wish to construct the approximators for computing  $CC(\mathcal{X}) \vee CC(\mathcal{Y})$ , and  $CC(\mathcal{X}) \wedge CC(\mathcal{Y})$ , respectively.

##### Case 1: $\vee$ -gate.

*Naive (wrong) attempt:*  $CC(\mathcal{X}) \vee CC(\mathcal{Y})$  is approximated by  $CC(\mathcal{X} \cup \mathcal{Y})$ . That is,  $CC(X_1, \dots, X_m, Y_1, \dots, Y_{m'})$ . 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  $X_1, \dots, X_m, Y_1, \dots, Y_{m'}$ . To do this we use a combinatorial lemma called The Sunflower Lemma.

#### 2.3.3.1 The Sunflower Lemma



**Fig. 2.5** From <https://theorydish.blog/2021/05/19/entropy-estimation-via-two-chains-streamlining-the-proof-of-the-sunflower-lemma/> by Lunjia Hu

**Definition 2.3 (Sunflower)** Let  $U$  be some universe, namely a set of elements (e.g., nodes). Let  $P = \{P_1, \dots, P_p\}$  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  $\text{core} \subseteq U$ , 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  $P$ .

Note: It's okay if the core is the empty-set! This means all petals are (pairwise) disjoint.

**Sunflower Lemma** (Erdős-Rado): For every  $\ell, p$ , 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  $\{P_1, \dots, P_p\}$ , each  $P_i$  has size  $\leq \ell$ , 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  $\ell$ .

*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  $Z \setminus D$  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  $|E| \leq (p-1) \cdot \ell$ .

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

*Remark 2.1* If a set  $E$  intersect all sets in a family of sets  $X_1, \dots, X_M$  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.  $\square$

Consider

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

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

### 2.3.3.2 Induction step of approximator construction continued: approximating OR and AND using Plucking

- By the Sunflower Lemma, every family of  $\geq M$  nonempty sets, each of cardinality  $\leq l$ , 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  $\leq M$ ). pluck ( $z$ ): the result of repeated plucking of a family of sets  $Z$ , until  $|Z| \leq M$ . Definition (Approximate OR and AND): foxily of sets The approximate  $-V$  of two crude -circuits  $CC(x)$  and  $cc(y)$  is:  $cc(\text{pluck}(x \cup y))$ .

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

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

A k-clique  $G = (v, E)$  s.t.:  $(cc(x) \vee cc(y))(G) = 1$  2) a false positive:  $cc(\text{pluck}(x \cup y))(G) = 0$   $V$ -aterosimulator  $A(k - 1)$ -coburing  $G = (v, E)$  s.t.

$$\begin{aligned} (cc(x) \vee cc(y))(G) &= 0 \\ \frac{cc(\text{pluck}(x \cup y))(G)}{V\text{-approximator}} &= 1 \end{aligned}$$

1 -apprimator of  $CC(x) \wedge CC(y)$  introduces 1) a false negative:

A k-clique  $G = (v, E)$  s.t.:  $(cc(x) \wedge cc(y))(G) = 1$  2) a false positive:  $cc(\text{pluck}(x \cup y))(G) = 0$

$$cc(\text{pluck} \{x_i \cup Y_j : |x_i \cup y_j| \leq l, x_i \in X, y_j \in Y\}) = 1$$

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  $\overline{CC}(x_1, \dots, x_m)$  and  $CC(y_1, \dots, y_{m_m})$  and consider a false positive introduced by

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

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

$$cc(x_1, \dots, x_m)(G) = 0, \quad cc(y_1, \dots, y_m)(G) = 0$$

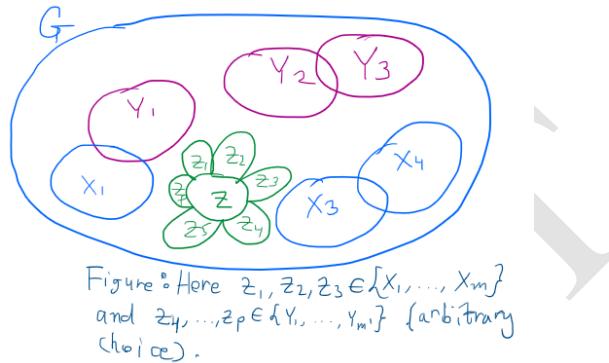
and

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

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  $\{z_1, \dots, z_p\}$  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  $\frac{\text{same colour!}}{L \text{ bg}(1)}$



**Fig. 2.6** 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.

$$\Pr[R(z_1) \wedge \dots \wedge R(z_p) \wedge \neg R(z)] \leq \Pr[R(z_1) \wedge \dots \wedge R(z_p) \mid \neg R(z)]$$

$$\begin{aligned} &= \prod_{i=1}^p \Pr[R(z_i) \mid \neg R(z)] \\ &\leq \prod_{i=1}^p \Pr[R(z_i)] \leq \frac{1}{2^p} \end{aligned}$$

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

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

We've seen before that

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

Probability of repetition of colours is increased if we do it 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  $2M$  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 approximators  $CC(x)$  and  $C(y)$ . It can be broken down in three phases: First, we form  $CC(\{X \cup Y : X \in \chi, Y \in \gamma\})$ ; 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  $\ell$ ), 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  $2^{-p}(k-1)^n$  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 XXX

We need to show that each approximation step introduces  $\leq M^2 \binom{n-l-1}{k-l-1}$  false negatives.

**Proof (Proof of Lemma XXX)** Case V:a false negative: A  $k$ -clique  $G = (v, E)$  st.:  $(cc(x) \vee ccc(y))(G) = 1$

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

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.:  $(cc(x) \wedge cc(y))(G) = 1$

$$c(P/\lvert \text{uck } \{x_i \cup Y_j : |x_i \cup Y_j| \leq l, x_i \in \chi, Y_j \in \gamma\} \rvert) = 0 \quad (2)$$

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.

(cont.) 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  $\sigma$  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  $\leq M^2$  deletions of sets, (because  $|\chi| = |y| = M$ ), we get that the number of false negatives introduced by  $a_n$  1-approximation step is  $\leq M^2 \cdot \binom{n-l-1}{k-l-1}$ .

DRAFT

DRAFT

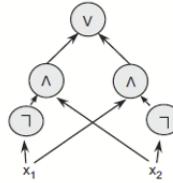
# Chapter 3

## 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:



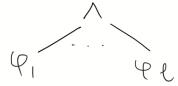
Recall that we are interested in the asymptotic study of circuit families, not a single circuit. That is, we want to consider how circuit size grows when the number of inputs  $n$  to a family of Boolean functions grows.

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!

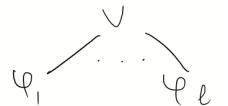
Note: If the depth of the circuit is constant & the fan-in of gates is at most 2, then the number of functions we can compute w/ such constant-depth circuits is constant. For example, the number of variables (appearing as leaves, i.e., input nodes) is constant that way, so for a large enough number of 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 ( $\varphi_1 \wedge \varphi_2 \wedge \dots \wedge \varphi_\ell$ ):



Unbounded fan-in OR ( $\varphi \vee \dots \vee \varphi_\ell$ ): Where  $\varphi_i$ '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.
3. 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.