Lecture 2: Lower Bounds for Formula Size

Topics in Algorithms and Complexity Theory (Spring 2020)
Rutgers University
Swastik Kopparty
Scribe: Adeeb, Chaitanya

1 Boolean circuits

We recall that a boolean formula can be represented as a tree-shaped circuit, and we associate the size of that formula with the number of leaves in the circuit. The size of the circuit depends on basis chosen (i.e. functions $m : \{0,1\}^n \to \{0,1\}$ such as $\vee, \wedge, \oplus$, etc. allowed as gates in our formulas). When we allow gates computing any such function with 2 bits as input, we call this the *full binary basis*.

Example: Consider formulas in n variables labelled $x_1, \ldots, x_n$. In the full binary basis, it is clear to us $x_1 \oplus \cdots \oplus x_n$ has size n. However, if we limit to $\{\vee, \wedge, \neg\}$ as our basis, then this formula takes size n^2.

2 Nechiporuk's Lower Bound

The technique discussed in this section gives a lower bound of $\Omega\left(\frac{n^2}{\log n}\right)$ on formula size by relating to the contribution of a single variable to the total size of the formula. We demonstrate this by examining a specific application of the method. A small note for notation convention - if $a \in \mathbb{N}$, then $[a] = \{1, \ldots, a\}$.

Definition 1. *Suppose we have input $z_1, z_2, \ldots, z_b$, where each $z_i \in \{0,1\}^a$ and $n = ab$. Then we define ELEMENT-DISTINCTNESS (ED) as*

$$ED(z_1, \ldots, z_b) = \begin{cases} 1 & \text{if all } z_i \text{ distinct} \\ 0 & \text{otherwise} \end{cases}$$

We start by considering any formula computing ED among b inputs of length a. Let size s be the number of leaves in the corresponding tree-shaped circuit. We refer to each leaf as $z_{i,j}$, which reads as "the j^{th} bit of input z_i," where $i \in [b]$. From this, we define $s_1, \ldots, s_b$ as

$$s_i = \{\#\text{ of leaves labelled } z_{i,j} \mid j \in [a]\}$$

By basic counting, it is apparent that $\sum_{i=1}^{b} s_i = s$.

The idea behind Nechiporuk's argument comes down to examining a restriction of our boolean formula by considering only *one* of the input variables while holding all others constant. Let

$z_2, \ldots, z_n$ be constant a bit values. We can "push" these parts to be smaller and smaller since they are constant on all input, and all we really need to do is compare these smaller, constant parts to s_i bits. So, $ED(z_1, y_2, \ldots, y_b)$ where each y_i is a constant input and can be simplified. Hence, size of formula is s_1. This argument works for all s_i.

How many different formulas do we get when we vary $y_2, \ldots, y_b \in \{0,1\}^a$ where each formula computes $f(x) = ED(x, y_2, \ldots, y_b)$?

Since the function behaves same on every set of distinct $b-1$ numbers chosen from all possible 2^a numbers, and in all other sets of size $b-1$ at least one number is repeated, so the value of ED will always be 0 regardless of input. Hence, the number of unique formulas is $\binom{2^a}{b-1} + 1$.

Observe that the number of rooted binary trees given a full binary basis is $2^{O(s_1)}$. This yields the following:

$$\#\text{ of formulas of size } s_1 \text{ on } a \text{ inputs} \leqslant 2^{O(s_1)} a^{s_1}$$

Note: We have a useful inequality for binomial coefficients.

$$\left(\frac{a}{b}\right)^b \leqslant \binom{a}{b} \leqslant \left(\frac{e \cdot a}{b}\right)^b$$

Expanding from our earlier bound on the number of formulas, we obtain

$$2^{O(s_1)} a^{s_1} \geqslant \binom{2^a}{b-1} \geqslant \left(\frac{2^a}{b-1}\right)^{b-1}$$

Then

$$s_1 \left(\log a + O(1)\right) \geqslant (b-1) \log\left(\frac{2^a}{b-1}\right)$$

$$\Rightarrow s_1 \geqslant \frac{(b-1)\left(a - \log(b-1)\right)}{\log a + \mathcal{O}(1)}$$

The above holds for all s_i, of which there are b. From this, we obtain

$$s \geqslant b\left(\frac{(b-1)\left(a - \log(b-1)\right)}{\log a + \mathcal{O}(1)}\right) \approx \frac{b^2 a}{\log a}$$

We note that the above is a useful bound only if $a < b$, so we "set" a to be a lot larger than $\log b$. Recall that $ab = n$. We can set $b = n/10 \log n$ and $a = n/b = 10 \log n$. Thus, if $b \gg 1$, then $a \gg \log b$. Going back to our expression, we get

$$s \geqslant \frac{1}{200} \cdot \frac{n^2}{\log^2 n} \cdot \frac{a \log n}{\log \log n} \geqslant \Omega\left(\frac{n^2}{\log n \cdot \log \log n}\right)$$

The method above is useful for analyzing other problems in formula size. Let $f : \{0,1\}^{k+2^k} \to \{0,1\}$ be the *Addressing Function*, where $f(i,x) = x_i$, where $i \in \{0,1\}^k$ and $x \in \{0,1\}^{2^k}$. We view i as an integer in the range $\left[2^k\right]$ which denotes a bit in x. Thus x_i is the i^{th} bit in x. We can extend this into the *Generalized Addressing Function* $GAF : \{0,1\}^n \to \{0,1\}$, where $n = \log b + ba + 2^a$. Then

$$GAF(u, i_1, i_2, \ldots, i_b, x) = x_{i_u}$$

where $u \in \{0,1\}^{\log b}$ is viewed as an integer in $[b]$ and $i_1, \ldots, i_b \in \{0,1\}^a$ are viewed as integers in $[2^a]$. The integer u is information as to which among $i_1, \ldots, i_b$ to pick, which we call i_u. Thus x_{i_u} is the i_u^{th} bit of x.

Homework (to be turned in): Prove the following claim: Let s be the size of a formula computing GAF. Show that:

$$s \geqslant \Omega\left(\frac{n^2}{\log n}\right)$$

(Hint: Choose an appropriate setting for a, b to get the bound)

3 AC^0 Circuits

AC^0 is the class of constant depth circuits with a basis of $\wedge, \vee, \neg$ gates with unbounded fan-in. Does this class of circuits have capacity to capture complicated functions? The answer is no.

Theorem 2. *n-bit parity needs exponential size in AC^0 setting.*

Similarly, we can define $AC^0(\oplus)$ as AC^0 circuits that are also equipped with unlimited fan-in $\oplus$ gates. Does adding n-bit parity as basis to above basis ($AC^0(\oplus)$), increase its capacity drastically? The answer is also no.

Theorem 3. *n-bit* Majority *needs exponential size in $AC^0(\oplus)$*

Lemma 4. *n-bit* Majority *has $\mathcal{O}(\log n), \text{poly}(n)$ size formulas / circuits with basis $\{\wedge, \vee, \neg\}$*

Proof. Let's look at a much easier problem which helps in solving n-bit Majority.

Define Base2 : $\{0,1\}^n \to \{0,1\}^{\log n}$, where Base2($x$) = # of 1's present in x.

Majority(x) is computed in $\mathcal{O}(\log n)$ time provided Base2(x) is computed in $\mathcal{O}(\log n)$ time, as we just need to compare result of Base2 with $\frac{n}{2}$. This takes $\log n$ depth since we compare $\log n$ bits. Hence, showing that Base2 has poly(n) size formula with depth $\mathcal{O}(\log n)$ proves the lemma. □

Let's see if divide and conquer approach helps in calculating Base2.

$$\text{Base2}(x_1, x_2, x_3 \cdots x_n) = \text{Base2}(x_1, x_2, \cdots x_{\frac{n}{2}}) + \text{Base2}(x_{\frac{n}{2}+1}, x_{\frac{n}{2}+2}, \cdots x_n)$$

We analyze the depth of this circuit as follows: it takes $\mathcal{O}(\log n)$ depth to add two numbers with $\log n$ bits since we sequentially add each corresponding bit to get carry and then proceed to next bit.

$$\begin{aligned}
\text{Depth} &= \mathcal{O}(\log n) + \mathcal{O}(\log \frac{n}{2}) + \mathcal{O}(\log \frac{n}{4}) + \cdots + \mathcal{O}(\log \frac{n}{2^i}) \\
&= \mathcal{O}(\log n) + \mathcal{O}(\log n - 1) + \mathcal{O}(\log n - 2) + \cdots + \mathcal{O}(1) \\
&= \mathcal{O}(\log^2 n)
\end{aligned}$$

We obtain a $\mathcal{O}(\log^2 n)$ circuit for Base2 but need a $\mathcal{O}(\log n)$ circuit, since only at $\mathcal{O}(\log n)$ depth conversion between circuits and formulas occur within polynomial size increase.

However, though we can compute Base2 of first half and second half above in parallel, we are constrained by the sequentiality of addition due to the carry bits. If we can somehow parallelize the step of adding bits, we may obtain better depth than $\mathcal{O}(\log^2 n)$.

Iterated Addition

Let's look at computing a general function which adds m numbers that are k bits each.

$SUM : \big(\{0,1\}^k\big)^m \mapsto \{0,1\}^{k+\lceil \log m \rceil}$ *s.t.* $SUM(y_1, y_2, \cdots y_m) = \sum_{i=1}^{m} y_i$ where each $y_i \in \{0,1\}^k$ is an integer of k bits, and the result of SUM is a base 2 representation of the integer sum.

Claim 5. *SUM can be computed by a formula of $\mathcal{O}(\log m)$ depth and poly size.*

Proof. Let $x, y, z \in \{0,1\}^k$ and x_i is i^{th} bit of x.

$$\begin{aligned}
x + y + z &= \left(\sum_{i=0}^{k-1} 2^i x_i\right) + \left(\sum_{i=0}^{k-1} 2^i y_i\right) + \left(\sum_{i=0}^{k-1} 2^i z_i\right) \\
&= \sum_{i=0}^{k-1} 2^i (x_i + y_i + z_i) \\
&= \sum_{i=0}^{k-1} 2^i \big(\text{Parity}(x_i, y_i, z_i) + 2 \cdot \text{Majority}(x_i, y_i, z_i)\big) \\
&= \sum_{i=0}^{k-1} 2^i \cdot \text{Parity}(x_i, y_i, z_i) + \sum_{i=0}^{k-1} 2^{i+1} \cdot \text{Parity}(x_i, y_i, z_i) \\
&= p + c
\end{aligned}$$

where p and c are two numbers of size $k+1$.

Parity and Majority of 3 bit inputs are constant depth circuits with depth 3. Therefore, by using a constant depth we can convert m numbers of size k bits to $\frac{2}{3}m$ numbers of size $k+1$ bits with same sum. Hence, with depth $\mathcal{O}(\log m)$, we get 2 numbers whose sum is the desired result and each number is of size $k + \log m$.

Now that we have two numbers, we use our addition circuit of depth $k + \log m$ (number of input bits) to get the final sum. Recall that formula of size s can be converted to formula of size $\leq s^k$ and depth $\mathcal{O}(\log s)$. Thus the final addition of two numbers is reduced to depth $\mathcal{O}(\log(k + \log m))$.

$\therefore$ Total depth of formula for SUM would be $\mathcal{O}(\log m) + \mathcal{O}(\log(k + \log m)) \approx \mathcal{O}(\log m)$.

□

Corollary 6. Base2 *can be computed by a formula of* $\mathcal{O}(\log n)$ *depth and poly size.*

Proof. Observe that Base2 is a special case of SUM where $m = n$ and $k = 1$. Hence, the depth of the corresponding circuit is $\mathcal{O}(\log n) + \mathcal{O}(\log(1 + \log n)) \approx \mathcal{O}(\log n)$. □

Note: Corollary 6 proves the claim made in Lemma 4.

Corollary 7. *$MULT$, or integer multiplication of two n-bit numbers, has a formula of depth* $\mathcal{O}(\log n)$ *and poly size.*

4 CNF's and DNF's

Definition 8. *A **CNF** is a formula of form $C_1 \wedge C_2 \wedge \cdots \wedge C_m$ where each C_i is a clause of the form $X_{i1} \vee X_{i2} \vee \cdots \vee X_{ij}$ where each X_{ij} is a literal of the form either a variable or negation of the variable.*

Example: CNF on variables $x_1, x_2, x_3, \cdots x_6$ would be

$$F = (x_1 \vee x_2 \vee \bar{x_4}) \wedge (x_2 \vee \bar{x_4} \vee \bar{x_5} \vee x_6) \wedge (\bar{x_2} \vee \bar{x_3}) \wedge (x_3 \vee \bar{x_4} \vee \bar{x_5})$$

where $\bar{x_i}$ is negation of variable x_i.

Fact 9. *Every boolean function $f : \{0,1\}^n \mapsto \{0,1\}$ has a CNF representation. Let y be an n bit number where the function gives 0. Hence, f can be written as*

$$\begin{aligned} f(x) &= \bigwedge_{ys.t f(y)=0} \big[Is(x \neq y)\big] \\ &= \bigwedge_{ys.t f(y)=0} \bigvee_{i=1}^{n} \big[Is(x_i \neq y_i)\big] \\ Is(x_i \neq y_i) &= x_i \ \text{ if } \ y_i = 0 \\ Is(x_i \neq y_i) &= \bar{x_i} \ \text{ if } \ y_i = 1 \end{aligned}$$

where each condition $Is(x \neq y)$ is an $\vee$ clause of n-bits as x is an n-bit number. It checks whether any of the individual bits of x, y differ i.e. Is($x_i \neq y_i$) where x_i, y_i are i^{th} bits of x,y respectively.

The definition of DNF follows similarly.

Definition 10. *A **DNF** is a formula of form $C_1 \vee C_2 \vee \cdots \vee C_m$ where each C_i is a clause of the form $X_{i1} \wedge X_{i2} \wedge \cdots \wedge X_{ij}$ where each X_{ij} is a literal of the form either a variable or negation of the variable.*

Example: DNF on variables $x_1, x_2, x_3, \cdots x_6$ would be

$$F = (x_1 \wedge x_2 \wedge \bar{x_4}) \vee (x_2 \wedge \bar{x_4} \wedge \bar{x_5} \wedge x_6) \vee (\bar{x_2} \wedge \bar{x_3}) \vee (x_3 \wedge \bar{x_4} \wedge \bar{x_5})$$

where $\bar{x_i}$ is negation of variable x_i.

Fact 11. *Every boolean function $f : \{0,1\}^n \mapsto \{0,1\}$ has a DNF representation. Let y be an n bit number where the function gives 1. Hence, f can be written as*

$$\begin{aligned} f(x) &= \bigvee_{ys.t f(y)=1} \big[Is(x = y)\big] \\ &= \bigvee_{ys.t f(y)=1} \bigwedge_{i=1}^{n} \big[Is(x_i = y_i)\big] \\ Is(x_i = y_i) &= \bar{x_i} \text{ if } y_i = 0 \\ Is(x_i = y_i) &= x_i \text{ if } y_i = 1 \end{aligned}$$

where each condition $Is(x = y)$ is an $\wedge$ clause of n-bits as x is an n-bit number. It checks whether all of the bits $x_i = y_i$ where x_i, y_i are i^{th} bits of x, y respectively.

Definition 12. *A k-**CNF** is a CNF formula where the number of literals in each clause is at most k.*

Lemma 13. *(Part of Cook-Levin Theorem) Any CNF can be written as a 3-CNF formula if extra literals are allowed. Formally, for every $f : \{0,1\}^n \to \{0,1\}$, there exists $g : \{0,1\}^{n+m} \to \{0,1\}$ such that*

- *g has 3-CNF formula where every clause is of size 3.*
- *$\forall x$ where $f(x) = 1 \iff \exists y$ s.t. $g(x, y) = 1$*

Example: A CNF with 4 literals can be written as a 3-CNF.

$$(x_1 \vee x_2 \vee x_3 \vee x_4) = (x_1 \vee x_2 \vee y_1) \wedge (\bar{y_1} \vee x_3 \vee x_4)$$

However, if extra literals are not forbidden, we are led to the following question: can every boolean function be written in 3-CNF form? The answer is no.

Lemma 14. *No 4-CNF can be written as a 3-CNF formula.*

Proof. An arbitrary 4-CNF formula consists of $\vee$ clauses of size 4. Suppose towards contradiction that a single clause with 4 literals can be expressed as 3-CNF. Then

$$(x_1 \vee x_2 \vee x_3 \vee x_4) = \bigwedge_i (y_{i,1} \vee y_{i,2} \vee y_{i,3})$$

where $y_{i,j}$ are literals using the variables x_1, x_2, x_3. Let us consider an input where a clause consisting of variables $y_{i,1}, y_{i,2}, y_{i,3}$ are zero, and thus the value of the R.H.S. of equation is zero. However, since value of only 3 literals are fixed, we can always choose the value of 4^{th} literal such that the L.H.S of equation evaluates to 1. Contradiction, so no boolean function in 4-CNF form can be written as a 3-CNF formula. □

On the contrary, let us formally define when a boolean function can be written as k-CNF.

Definition 15. *For any boolean function $f : \{0,1\}^n \to \{0,1\}$, the **0-certificate complexity**, $C_0(f)$, is the smallest k where for every $x \in \{0,1\}^n$ where $f(x) = 0$, there exists $S \subseteq [n]$ such that*

- $|S| \leqslant k$
- *For every $y \in \{0,1\}^n$ such that $y|_S = x|_S$, we have $f(y) = 0$*

*where $x|_S$ are bits of x at the indices selected by subset S (read as the **restriction** of x to S).*

$$\text{For } x \in \{0,1\}^n, \text{ and } S \subseteq [n]$$
$$x|_S = (x_i)_{i \in S} \in \{0,1\}^{|S|}$$

The definition above states that for every possible n bit inputs, there always exists a subset of input bit positions S such that restricting them, fixes the value of the function to be 0. Thus, S is the certificate for which the function evaluates to zero. The minimum size of such a subset in bits needed to fix / certify the value of the function is the *certificate complexity*. In this case, S fixes the function to evaluate to 0, so we refer to it as the *0-certificate complexity*.

Lemma 16. *f can be written as a k-CNF iff $C_0(f) \leqslant k$.*

Proof. This proof trivially follows from the definition of 0-certificate complexity. By definition, for every input x where a given boolean function evaluates to zero, there exists a subset S of bits from these inputs, and when the bits in S are fixed appropriately, the function evaluates to zero. The minimum number of bits needed to be fixed, which is exactly the 0-certificate complexity, k.

From Fact 9, every function can be written as CNF

$$f(x) = \bigwedge_{\forall y \text{ s.t. } f(y)=0, \exists S \text{ s.t. } |S| \leqslant k} \left[Is(x|_S \neq y|_S)\right] \text{ as } C_0(f) \leqslant k, \text{ existence of such set S is guaranteed}$$
$$= \bigwedge_{(\forall y \text{ s.t. } f(y)=0 \exists S \text{ s.t. } |S| \leqslant k)} \bigvee_{j \in S} \left[Is(x_j \neq (y|_S)_j)\right] \text{ where each clause is of size k}$$

Hence, f can be written as k-CNF. □

Definition 17. *For any boolean function $f : \{0,1\}^n \to \{0,1\}$, the **1-certificate complexity**, $C_1(f)$, is the smallest k where for every $x \in \{0,1\}^n$ with $f(x) = 1$, there exists $S \subseteq [n]$ such that*

- $|S| \leqslant k$
- *For every $y \in \{0,1\}^n$ such that $y|_S = x|_S$, we have $f(y) = 1$*

Lemma 18. *f can be written as a k-DNF iff $C_1(f) \leqslant k$.*

Proof. Similar to the above proof. □

Example:

$$\begin{aligned} f(x) &= x_1 \wedge x_2 \wedge \cdots \wedge x_n \\ C_0(f) &= 1 \\ C_1(f) &= n \end{aligned}$$

Definition 19. ***Certificate complexity**, $C(f) = max(C_0(f), C_1(f))$.*

5 Decision Trees

Definition 20. ***Decision trees** are a model of computation where each input bit is read, and the decision to read next input bit is taken based on the inputs read so far. It models the execution of a boolean function with output bit at the leaf.*

Definition 21. ***Deterministic decision trees** choose the next input bit to read deterministically based on current input bit read. Similarly, **randomized decision trees** choose the next input bit to read based on the outcome of a random event alongside with the current input bit read.*

Definition 22. *The **depth** of the decision tree is the length of the longest path from root to leaf in the tree.*

Definition 23. *The **deterministic decision tree depth** of a boolean function f is the smallest depth of all possible decision trees for f and is denoted by $DT(f)$ (the 'D' stands for deterministic).*

Lemma 24. *If a boolean function f has a decision tree of depth $\leqslant d$, then $C(f) \leqslant d$.*

Examples:

- $DT(n \text{ bit } AND) = n, \ \because C_1(n \text{ bit } AND) = n$
- $DT(n \text{ bit } XOR) = n, \ \because C_1(n \text{ bit } XOR) = n \ \text{ and } \ C_0(n \text{ bit } XOR) = n$
- $DT(n \text{ bit Maj}) \geqslant n/2 + 1, \ \because C_1(n \text{ bit Maj}) = C_0(n \text{ bit Maj}) = n/2 + 1$

We see $DT\,(n \text{ bit Maj}) = n$, since for any corresponding decision tree, we can always design an adversary input where $0, 1$ are alternatively varied. The resulting output is only decided by last input bit before the leaves. So, the depth should be n.

Fact 25. *If f depends on n variables, then $DT(f) \geq \log n$.*