Lecture 1: Course overview, circuits, and formulas

Topics in Complexity Theory and Pseudorandomness (Spring 2013)
Rutgers University
Swastik Kopparty
Scribes: John Kim, Ben Lund

1 Course Information

Swastik Kopparty
Email: swastik.kopparty@rutgers.edu
Website: http://www.math.rutgers.edu/ sk1233/courses/Topics-S13/

Good reference books worth flipping through:

- Jukna: Extremal Combinatorics
- Jukna: Boolean Function Complexity
- Wegener: Complexity of Boolean Functions
- Vadhan: Pseudorandomness
- Luby and Wigderson: Pairwise Independent Hashing and Applications

2 Course Overview

2.1 Circuits and Formulas

A typical setting is that there is a function

$$f : \{0,1\}^n \to \{0,1\}$$

we want to compute. We will be studying whether this function can be computed using little resources.

Definition 1. *A boolean* circuit *is a directed acyclic graph with* **wires** *carrying some value in* $\{0,1\}$*, and these values are processed by* **gates** *computing certain prespecified boolean operations.*

Example 2. *For example, using the gates AND* ($\wedge$)*, OR* ($\vee$) *and NOT* ($\neg$)*, the function:* $f(x_1, x_2, x_3) = (x_1 \wedge \neg x_2) \vee x_3$ *is representable by a circuit with* 3 *gates.*

Definition 3. *A* formula *is a circuit where each gate has fan-out at most 1.*

Formulas represent computation where the results of subcomputations cannot used more than once.

2.2 Branching Programs

Branching programs are used to model low-space computation. We will see some interesting results about the power of branching programs. We will also come across them when we study pseudorandomness.

2.3 Data Structures

Data Structures are schemes for storing data in a way that we can answer questions about the data quickly. A sample problem is to design a data structure for the SET MEMBERSHIP problem: We are given a set $S \subseteq [m]$ with $|S| = n$. We want to store something compactly in memory so that we can answer questions of the form "Is $i \in S$?" (where $i \in [m]$) by probing very few locations in memory.

There is a trivial data structure that uses m bits of memory and allows us to answer questions with just 1 probe to memory: Let the i'th bit of memory equal 1 iff $i \in S$. Then to answer whether $i \in S$, we check the value of the i'th bit. This algorithm achieves minimum runtime, but requires a lot of space (m bits).

What if we want to save as much space as possible? If we use s bits of space, then what is stored in memory can take on at most 2^s possible values. Now since what is stored in memory should suffice to answer all the questions "is $i \in S$?", what is stored in memory should determine S. Thus $2^s \geq \binom{m}{n}$ and so the space required is at least $\log_2 \binom{m}{n}$ bits. This is because the query determines whether $i \in S$ for all $i \in [m]$, so the query actually determines the subset S. The lower bound on the space requirement then follows from the fact that there are $\binom{m}{n}$ n-element subsets of $[m]$. A simple algorithm that achieves the lower bound on space maps each choice of S to a unique sequence of $\left\lceil \log \binom{m}{n} \right\rceil$ bits. The algorithm simply reads off the bits and uses the map to find the corresponding set S and answer the query. This algorithm uses the minimal amount of space ($\left\lceil \log \binom{m}{n} \right\rceil$ bits), but has a large query time of $\log \binom{m}{n}$.

These algorithms seem to imply that there is a tradoff between space and query time. However, we will see an algorithm that uses $\log \binom{m}{n}(1 + o(1))$ bits with a query time of $O(1)$.

2.4 Pseudorandomness

The other major theme of the class is pseudorandomness, or the generation of randomness.

What is randomness? How do we get it? How do we know when we've gotten it?

The big insight that starts the theory is that when we "want" randomness, it is for some purpose. We then define pseudorandomness to be whatever suffices for that purpose.

Definition 4. *A sequence of bits is* pseudorandom *if no algorithm of interest can distinguish between it and truly random bits.*

We will be much more concrete when the time comes, but this is the gist of it.

A good example of a randomized algorithms comes from graph connectedness. Given an undirected graph G on n vertices, is G connected? Suppose only $O(\log n)$ space is allowed. It is not obvious

how to do this at all. The classical breadth-first-search (BFS) takes $\Omega(n)$ space.

There is a beautiful classical low-space algorithm for this problem:

For every pair of vertices (s, t), do:

1. Start a random walk on G starting at s.
2. Run it for n^{10} steps.
3. If you never see t, say DISCONNECTED.

If you never said DISCONNECTED, say CONNECTED.

On every graph, this randomized algorithm outputs the correct answer with probability greater than 0.999. An amazing result of Reingold shows how to make this algorithm deterministic. It involves taking a "pseudorandom walk". We will prove this result in the course.

2.5 Hardness vs Randomness

If a problem is hard, then you can't guess the answer, and so the answer is "random" to you. Perhaps you can then use the answer to that problem as the random bits to feed into a randomized algorithm. The theory of hardness versus randomness makes this rigorous. Roughly: if you can give me f that is *hard* for some computational model C, then you can get *pseudorandom generators* that generate bits that look random to C. We will prove versions of this and use it to get actual pseudorandom generators for some interesting computational models.

2.6 Concrete pseudorandomness

I will just mention some words:

- Expander graphs
- ϵ-biased sets
- k-wise independent sets
- randomness extractors

3 Circuits and Formulas

Let $f : \{0,1\}^n \to \{0,1\}$. Recall that a circuit is built from wires carrying 0 or 1 and gates. There are two common bases for the gates: the *full binary basis* and the *de Morgan basis*. The de Morgan basis consists of $\wedge$ (AND), $\vee$ (OR), and $\neg$ (NOT). The full binary basis consists of gates computing all 16 functions $b : \{0,1\}^2 \to \{0,1\}$, and includes the de Morgan basis. Note that all basis elements have *fan-in*≤ 2 (number of inputs into gate). The *fan-out*, the number of outputs out of a gate, is unrestricted in circuits. In formulas, the fan-out for each gate is 1.

Observation 5. *Every function has a formula.*

To see this, note that for each $y \in \{0,1\}^n$, there is a $\delta_y : \{0,1\}^n \to \{0,1\}$, which is expressible as a $\wedge$ of n variables and complements of variables, such that $\delta_y(x) = 1$ iff $x = y$. Writing

$$f(x) = \bigvee_{\substack{y \in \{0,1\}^n \\ f(y)=1}} \delta_y(x)$$

completes the proof.

We can use this proof to get an upper bound on the number of gates of f:

$$gates(f) \leq (n-1)(2^n - 1) \leq O(n \cdot 2^n).$$

3.1 Circuit size basics

Maybe every function has a circuit of size $\texttt{poly}(n)$? This is ruled out by the following fundamental result.

Theorem 6 (Shannon)**.** $\forall \epsilon > 0$*, most functions* $f : \{0,1\}^n \to \{0,1\}$ *require circuits of size at least* $\frac{2^n}{n}(1-\epsilon)$.

Proof. In 1 line: There are lots of functions, but not too many small circuits.

The total number of functions is 2^{2^n}. Suppose we consider circuits of size (number of gates) at most s. Each gate takes exactly two inputs that are either other gates or input bits. So there are at most $\binom{s+n}{2}$ ways to choose inputs for each gate. Since there are 16 binary functions in the full binary basis, this gives at most $16\binom{s+n}{2}$ ways to choose a gate function and its inputs. Since there are at most s gates, the number of circuits of size $\leq s$ is at most $16^s\binom{s+n}{2}^s$.

Set $s = \frac{2^n}{n}(1-\epsilon)$. We want to show that the number of circuits of size at most s is much less than the total number of functions.

$$\begin{aligned}
16^s \binom{s+n}{2}^s &\leq 16^s (s+n)^s \\
&\leq 16^s (2s)^s \\
&= 32^s s^s \\
&= 32^{\frac{2^n}{n}(1-\epsilon)} \left(\frac{2^n}{n}(1-\epsilon)\right)^{\frac{2^n}{n}(1-\epsilon)} \\
&= \left(\frac{32(1-\epsilon)}{n}\right)^{\frac{2^n}{n}(1-\epsilon)} 2^{2^n(1-\epsilon)} \\
&< 2^{2^n(1-\epsilon)} \\
&\ll 2^{2^n}.
\end{aligned}$$

□

Theorem 7 (Shannon and Lupanov)**.** $\forall \epsilon > 0$, $\forall$ *functions* $f : \{0,1\}^n \to \{0,1\}$, $\exists$ *circuits of size at most* $\frac{2^n}{n}(1+\epsilon)$.

We will only prove an upper bound of $O(\frac{2^n}{n})$.

Proof. First, we find a circuit of size $O(2^n)$ using the recurrence:

$$f(x_1, ..., x_n) = (x_1 \wedge f(1, x_2, \ldots, x_n)) \vee (\overline{x_1} \wedge f(0, x_2, \ldots, x_n)).$$

$$\begin{aligned} \Rightarrow size(n) &\leq 3 + 2 \cdot size(n-1) \\ \Rightarrow size(n) &= O(2^n). \end{aligned}$$

More explicitly, this gives us a sequence of representations of the function, in terms of residual functions on fewer bits which we obtain by fixing the first few input bits to certain values:

$$f(x_1, ..., x_n) = (x_1 \wedge f(1, x_2, \ldots, x_n)) \vee (\overline{x_1} \wedge f(0, x_2, \ldots, x_n)).$$

as

$$\begin{aligned} f(x_1, ..., x_n) =& (x_1 \wedge x_2 \wedge f(1, 1, x_3, \ldots, x_n)) \vee (\overline{x_1} \wedge x_2 \wedge f(0, 1, x_3, \ldots, x_n)) \\ & \vee (x_1 \wedge \overline{x_2} \wedge f(1, 0, x_3, \ldots, x_n)) \vee (\overline{x_1} \wedge \overline{x_2} \wedge f(0, 0, x_3, \ldots, x_n)). \end{aligned}$$

$$\ldots$$

To get a circuit of size $O(\frac{2^n}{n})$, we stop the recursion after t steps (t to be determined later). At this stage of the recurrence, for each $(b_1, \ldots, b_t) \in \{0,1\}^t$ we need to compute $f(b_1, \ldots, b_t, x_{t+1}, \ldots, x_n)$, thought of as a function on the last $n-t$ bits. Now the total number of functions on the variables $x_{t+1}, \ldots, x_n$ is $2^{2^{n-t}}$. The idea is this: if $2^{2^{n-t}} \ll 2^t$, then it is more economical to compute all the $2^{2^{n-t}}$ functions once beforehand, rather than to compute the residual function for each of the 2^t branches of the recursion. So we compute all $2^{2^{n-t}}$ functions on $x_{t+1}, \ldots, x_n$ and feed them to the appropriate gates of the representation of f using t steps of the recursion. Call this circuit C.

Then $size(C) \leq 3 \cdot 2^t +$ size required to compute all functions on $n-t$ bits.

Claim 8. *Size required to compute all functions on* k *bits* $\leq 2^{2^k}(1 + o(1))$.

We prove the claim by induction on k. We first compute all functions on $k-1$ bits, which takes size $2^{2^{k-1}}$. For each f_1 and f_2 in this class, compute $(x_k \wedge f_1(1, x_2, \ldots, x_{k-1})) \vee (\overline{x_k} \wedge f_2(0, x_2, \ldots, x_{k-1}))$. This gives all functions on k bits. Hence,

$$size(k) \leq \left(2^{2^{k-1}}\right)^2 + 2 \cdot 2^{2^{k-1}} + 2^{2^{k-1}} = 2^{2^k}(1 + o(1)).$$

This completes the proof of the claim.

So $size(C) \leq 3 \cdot 2^t + 2^{2^{n-t}}$. Set $t = n - \log(n - \log n)$. Then

$$\begin{aligned} size(C) &\leq 3 + 2 \cdot 2^{n-\log(n-\log n)} + 2^{2^{\log(n-\log n)}} \\ &= \frac{3 \cdot 2^n}{n - \log n} + 2^{n-\log n} \\ &= 3 \cdot \frac{2^n}{n - \log n} + \frac{2^n}{n} \\ &= O\left(\frac{2^n}{n}\right). \end{aligned}$$

□

The best known lower bound on circuit size for an explicit function is only $5n$. Yet we know that there are functions out there (in fact most functions out there) whose smallest circuit has exponential size. One of the biggest problems of complexity theory is to find an explicit such function.

4 Formula size basics

We saw earlier that every function of size n can be computed by a circuit of size $(1+\epsilon)2^n/n$, and that relatively few functions can be computed by a circuit of size less than $(1-\epsilon)2^n/n$. The corresponding tight lower bound on the size of formulas for most functions is $2^n/\log n$.

Theorem 9 (Shannon)**.** *Most boolean functions on n variables cannot be computed by a formula of size less than $(1-\epsilon)2^n/\log n$, for any $\epsilon > 0$.*

Theorem 10 (Lyupanov)**.** *Every boolean function on n variables can be computed by a formula of size $O(2^n/\log n)$.*

These theorems hold in both the full binary basis and the de Morgan basis.

Theorem 9 can be proved by the following counting argument; we won't see a proof of 10.

Proof. We will compare a lower bound on the number of formulas of size s to the number of functions of size n.

A formula of size s can be written as a binary tree with s internal nodes. Each internal node is labeled by a gate, and each leaf is labeled by a variable, the complement of a variable, 0, or 1.

The number of binary trees with s internal nodes is given by the Catalan number $C_s \leq 4^s$. Each internal node is labeled with one of the gates in our basis; if there are b gates in the basis, the number of ways to do this labeling is b^s; for any basis, this is at most 16^s, for the full binary basis. There are $(2n+2)$ ways to label each leaf in the tree, and there are $s+1$ leaves.

Thus, the total number of formulas of size s is less than $(2n+2)^{s+1}4^s16^s \leq n^s c^s$ for some constant $c > 1$.

The number of functions from $(0,1)^n$ to $(0,1)$ is 2^{2^n}.

If $s \leq (1-\epsilon)2^n/\log n$ for any $\epsilon > 0$, then $n^s c^s << 2^{2^n}$. Thus, only a small fraction of all the boolean functions on n variables can be computed by any formula of size $(1-\epsilon)2^n/\log n$.

□

5 The complexity class NC^1

The class of functions computable by a formula of size $\texttt{poly}(n)$ is the same as the class of functions computable by a formula of size $\texttt{poly}(n)$ and depth $O(\log n)$.

Theorem 11. *Every formula of size s can be converted into an equivalent formula of depth $O(\log s)$.*

Proof. To prove this theorem, we need the following lemma:

Lemma 12. *In any rooted binary tree with s vertices, there is a vertex g such that the tree rooted at g has between $s/3$ and $2s/3$ vertices.*

Proof. A vertex that is the root of a subtree with more than $2s/3$ vertices has a child that is the root of a subtree with at least $s/3$ vertices. We can use this fact to design a simple algorithm to find a vertex that is the root of a subtree with between $s/3$ and $2s/3$ vertices.

Start with the root as the current vertex. Select the child of the current vertex that has at least $s/3$ vertices in its subtree. If this vertex has at most $2s/3$ vertices, we're done. Otherwise, set this vertex as the current vertex, and repeat the process. □

Now we'll use this lemma to prove theorem 11 The proof will be by induction; the base case is trivial. The inductive hypothesis is that for any $s' < s$ every formula of size s' can be expressed as a formula of depth $\leq c \log s' + d$ (for constants c, d).

Given a formula F, we can use the lemma to find a sub-formula F_g with size in the range $[s/3, 2s/3]$. We can also find two formulas F_0 and F_1, which are F with the sub-formula F_g replaced by 0 and 1, respectively. These formulas also have size in the range $[s/3, 2s/3]$.

It is easy to check that $F = (F_g \wedge F_1) \vee (\neg F_g \wedge F_0)$. Thus, by applying the inductive hypothesis, we can replace F with an equivalent formula of depth $2 + c\log(2s/3) + d \leq c\log(s) + d$ (for suitable c, d). □

The largest possible size of a formula of depth d is 2^d. Thus, a formula with logarithmic depth must have polynomial size.

Theorem 11 shows that all formulas with size $O(n^c)$ have equivalent formulas with size $O(n^{c'})$ and depth $O(\log n)$. Thus, there is a single complexity class that captures all functions that have formulas with polynomial size, and all functions that have formulas with polynomial size and logarithmic depth. This is the class NC^1.

If we begin with a circuit with the fan-in bounded by 2 and depth d, an upper bound on the size of the formula computing the function would be 2^d. To do this, simply replace any sub-circuit having more than one output with several copies of that sub-circuit, each having a single output.

Thus, NC^1 also equals the class of all functions that have circuits of polynomial size and logarithmic depth.

6 Upper bounds on formula size for specific functions

Several specific problems are known to be in NC^1.

PARITY is the family of boolean functions that evalute to 0 if the sum of the inputs is even and to 1 if the sum of inputs is odd.

Theorem 13. $\mathsf{PARITY} \in \mathsf{NC}^1$.

Proof. In the full binary basis, we can calculate the parity function on $\{0,1\}^n$ as $\mathsf{PARITY}\{0,1\}^a \oplus \mathsf{PARITY}\{0,1\}^{n-a}$. Thus, we can use binary divide and conquer to calculate the parity function with a formula of size $O(n)$.

In the de Morgan basis, $x_1 \oplus x_2$ can be expressed as $(x_1 \vee x_2) \wedge \neg(x_1 \wedge x_2)$. Using this substitution, we can calculate the parity function with a circuit of size $O(n)$. In order to convert this circuit to a formula, we need to duplicate each input to the `xor` gate so that no gate has more than one output.

Let $F(n)$ be the minimal size of a formula, counting only `or` and `and` gates, for PARITY of n bits in the de Morgan basis. Using the construction described in the previous paragraph, $F(n) \leq 3 + 2F(\lfloor n/2 \rfloor) + 2F(\lceil n/2 \rceil)$. Thus, $F(n) \leq O(n^2)$.

□

We can also do operations on integers with small formulas. All integers are assumed to be nonnegative and written in base 2. We first show that we can add two n bit integers using a formula of size $\texttt{poly}(n)$.

Theorem 14. *Integer addition is in* NC^1.

Proof. We'll assume binary inputs and use the full binary basis for the formula we build.

If we can compute the i^{th} carry bit c_i, then we can compute the i^{th} bit of the solution as

$$s_i = x_i \oplus y_i \oplus c_i,$$

where x_i and y_i are the i^{th} bits of the input. The last bit of the solution will be

$$s_{n+1} = c_{n+1}.$$

The i^{th} bit of the carry will be 1 iff there exists a $j < i$ such that $x_j = y_j = 1$, and for all $j < k < i$ it is not the case that $x_k = y_k = 0$. This condition is directly expressible as a formula:

$$c_i = \bigvee_{j<i} \left(x_j \wedge y_j \wedge \bigwedge_{j<k<i} (x_j \vee y_j) \right).$$

The number of gates needed to compute the carry is thus $\sum_{i=1}^{n+1}\sum_{j=1}^{i-1}(3+\sum_{k=j+1}^{i-1}2) \leq O(n^3)$. The number of additional gates required to calculate the solution from the carry and the inputs is $O(n)$, so addition of two n bit numbers can be performed by a formula in the full binary basis with $O(n^3)$ gates. □

It is also possible to add n numbers, each n bits long, using a formula of size $\mathsf{poly}(n)$.

Theorem 15. *Iterated integer addition is in* NC^1.

Proof. For any three integers, X,Y, and Z, we will show how to find two numbers S and C such that $X+Y+Z = S+C$ using a formula with constant depth. Using this procedure, we can reduce the problem of adding n integers to the problem of adding 2 integers with a formula with depth $O(\log_{3/2} n)$. Finally we can add two integers using the previous theorem.

For an integer A, let A_i denote its i'th bit (corresponding to the coefficient of 2^i). The trick is the following: define $S_i = X_i \oplus Y_i \oplus Z_i$, and define $C_{i+1} = \text{Majority}(X_i, Y_i, Z_i)$. These are the "sum" and "carry" respectively when we add the three bits X_i, Y_i, Z_i. It is easy to check that $S + C = X + Y + Z$. □

Corollary 16. *Integer multiplication is in* NC^1.

MAJ is the family of boolean functions on an odd number of input bits that outputs the majority value of its input bits.

Theorem 17. $\mathsf{MAJ} \in \mathsf{NC}^1$.

This follows immediately from theorem 16, since we can compute the sum of the bits of the input (treated as 1-bit integers) and compare it to $n/2$ using a formula of polynomial size.

7 Lower Bounds for Formulas

A few specific, non-trivial lower bounds are known for formulas. Krapchenko proved that parity requires a formula of size $\Omega(n^2)$ in the de Morgan basis. Subbotovskaya and Andreev found explicit functions having no formulas of size $o(n^{2.5})$ in the de Morgan basis. Nechiporuk proved an $\Omega(n^2)$ lower bound in the full binary basis. Håstad proved an $\Omega(n^3)$ lower bound for an explicit function; we will not cover this result in detail.

7.1 Krapchenko's Lower Bound

Krapchenko's method uses "formal complexity measures". A formal complexity measure FC assigns real numbers to boolean functions such that it has the following properties:

1. $FC(x_i) = 1$.
2. $FC(f) = FC(\neg f)$.

3. $FC(f \vee g) \leq FC(f) + FC(g)$.

   Properties 2 and 3 together imply

4. $FC(f \wedge g) \leq FC(f) + FC(g)$.

Formal complexity measures are a powerful tool to prove lower bounds in the de Morgan basis. Let $L(f)$ be the formula complexity of f; that is, the size of the smallest formula in the deMorgan basis for the function f.

Theorem 18. *Any formal complexity measure FC satisfies*

$$FC(f) \leq L(f).$$

Proof. We will proceed by induction on $L(f)$. The base case $L(f) = 1$ follows from property 1.

Consider the smallest formula for f. If $f = f_1 \vee f_2$, then

$$\begin{aligned} L(f) &= L(f_1) + L(f_2) \\ &\geq FC(f_1) + FC(f_2) \\ &\geq FC(f_1 \vee f_2) = FC(f). \end{aligned}$$

The case $f = f_1 \wedge f_2$ follows in the same manner, using property 4 rather than property 3. □

Theorem 19 (Krapchenko)**.** PARITY *requires a formula of size $\Omega(n^2)$ in the de Morgan basis.*

We will prove this theorem in the next class.

The main tool is the following formal complexity measure:

$$FC(f) = \max_{\substack{A \subseteq f^{-1}(0) \\ B \subseteq f^{-1}(1)}} \frac{e(A,B)^2}{|A||B|},$$

where $e(A, B)$ is the number of pairs $(x, y) \in A \times B$ such that x and y differ in exactly one coordinate.

In the next class we will show that it is indeed a formal complexity measure.

For PARITY, let us show that $FC(PARITY) \geq \Omega(n^2)$. Consider $A = f^{-1}(0)$ and $B = f^{-1}(1)$. Then $|A| = |B| = 2^{n-1}$, and $e(A, B) = n2^{n-1}$. So $FC(PARITY) \geq n^2$, and so $L(PARITY) \geq n^2$.