Lecture 1 : Course overview, Boolean Circuits and Formulas

CS596 : Advanced Topics in Algorithms and Complexity Theory (Spring 2020)
Rutgers University
Swastik Kopparty
Scribe: Harsha Tirumala

1 Course information

This course will cover assorted results in theoretical CS which are either recent or interesting.

- Information relevant to the course can be found on the Course-Webpage
- Topics expected to be covered:
  1. Boolean Circuits (Computing power, Lower bounds)
  2. Data Structures (Constructions, Lower bounds)
     - Set Membership
     - Polynomial Evaluations
  3. Learning theory
     - Learning a function from evaluations

2 Boolean functions, Circuits and Formulas

Boolean Functions

A *Boolean Function* $f(x) = f(x_1, x_2, ..., x_n)$ of n variables is a mapping $f : \{0,1\}^n \to \{0,1\}$. It is easy to verify that the total number of boolean functions $f : \{0,1\}^n \to \{0,1\}$ are 2^{2^n}. A few examples of boolean functions:

- Parity Function $\oplus_n(x) = 1$ iff $x_1 + x_2 + ... + x_n \equiv 1 \bmod 2$
- Majority function $Maj_n(x) = 1$ iff $x_1 + x_2 + ... + x_n \geq \lceil \frac{n}{2} \rceil$
- Threshold functions $Th^n_k(x) = 1$ iff $x_1 + x_2 + ... + x_n \geq k$

A few basic boolean operations are :

- NOT (negation) $\neg x = 1 - x$
- AND (conjunction) $x \wedge y = x.y$
- OR (disjunction) $x \vee y = 1 - (1 - x).(1 - y)$
- XOR (parity) $x \oplus y = x(1 - y) + y(1 - x) = (x + y) \text{mod } 2$

Monotone functions

For two vectors $x, y \in \{0,1\}^n$ we write $x \leq y$ if $x_i \leq y_i$ for all $1 \leq i \leq n$. A boolean function is *monotone* if $x \leq y$ implies $f(x) \leq f(y)$.
The threshold function $Th_k^n(x)$, the majority function, AND, OR are a few examples of monotone functions whereas the parity function is not monotone.

Fact 1. *(Monotone functions $\leftrightarrow$ (AND, OR) basis)*
A boolean function is monotone if and only if it can be computed using the (AND, OR) basis of boolean operations.

The backward direction is trivial and the forward direction is left as an exercise.

Boolean Circuits

Let Φ be a set of some boolean functions. A *boolean circuit* of n variables over the basis Φ is a sequence of boolean functions $g_1, g_2, .., g_s$ such that the first n functions are input variables $g_1 = x_1, ..., g_n = x_n$ and each subsequent gate $g_i = \psi(g_{i_1}, g_{i_2}, ..., g_{i_d})$ of some basis function $\psi \in \Phi$.

Informally, a Boolean circuit is a model of computation that computes functions $f : \{0,1\}^n \rightarrow \{0,1\}$ using gates and "wires". Each gate has at most two inputs and one output. The fan-out of a gate in Boolean circuits is *unrestricted*. The *size* of a circuit is the total number of its gates. The *depth* of a circuit is the length of the longest path from an input node to an output node in the computation graph of the Circuit.

Boolean circuits may be classified based on the set of gates (**basis**) that they are allowed to use : for example, the De-Morgan Basis (AND, OR, NOT), the full binary basis (all boolean functions with 2 inputs and 1 output) etc.

Each circuit can be viewed as a Directed Acyclic Graph (DAG) whose fan-in 0 nodes correspond to the input variables and fan-out 0 node(s) correspond to the output(s). Since every DAG can have its vertices ordered along a line (such that all edges go from left to right), there is a natural order of evaluation that leads to the output bit(s) generation for a given Boolean circuit.

It is interesting to study what sized circuits are needed to compute Boolean functions. Questions of particular interest that we will immediately deal with : (1) Do all Boolean functions have small circuits? (2) Are there Boolean functions that require large circuits? (3) If so, are we able to give an explicit description for such hard to compute functions?

Definition 2. *Circuit Complexity :* The circuit complexity of a function f wrt the basis G (denoted by Ckt-Size$_G(f)$) is defined as the size of the smallest circuit computing f that only uses the operations from the basis G.

The circuit complexity of a function f depends on the basis of functions that we are allowed to use.

For example, there is an $O(n)$ sized circuit that computes the parity function over the basis (AND, XOR) whereas we can conclude from Fact 1 above that parity cannot be computed by circuits over the basis (AND, OR).

Lemma 3. (Basis switching for Circuits) *If G_1, G_2 are two finite bases that each generate all functions, then for any function $f : \{0,1\}^n \to \{0,1\}$ we have Ckt-Size$_{G_1}(f) \leq k.$Ckt-Size$_{G_2}(f)$ (where Ckt-Size$_{G_i}$ is the circuit complexity of f wrt the basis G_i and k only depends on G_1, G_2)*

Proof. Since it is allowed to route inputs through to multiple gates in circuits, each gate of basis G_1 can be replaced by a constant number of gates of basis G_2 that compute the same function as the gate G_1 and hence there is a constant blow up in the size of the circuit in switching from basis G_1 to basis G_2. □

Boolean Formulas

A *formula* is a circuit all of whose gates have fan-out at most 1. The underlying computation graph of a formula is a tree. For example, if we consider the De-Morgan Basis, every formula ψ is of one of the following three forms : $\psi_1 \vee \psi_2$, $\psi_1 \wedge \psi_2$ or $\neg\psi_1$ - where ψ_1, ψ_2 are boolean formulae of smaller size.

Unlike boolean circuits that only incur a constant-size blow up across different bases, formula sizes can vary a lot across bases. For example, Parity can be computed using a (AND, XOR) formula of size $O(n)$ whereas any De-Morgan formula requires $\Omega(n^2)$ gates to compute the parity function (due to Krapchenko). We have the following result on size blow-up involved in shifting bases for formulas :

Theorem 4. (Basis switching for formulas) *If G_1, G_2 are two finite set of gates that each generates all functions, then for any function $f : \{0,1\}^n \to \{0,1\}$, we have Formula-Size$_{G_1}(f) \leq$ Formula-Size$_{G_2}(f)^k$ (where k depends only on G_1, G_2).*

Proof: The outline of the proof is as follows : we first show that any formula computing f of size s (in basis G) can be converted to a formula of size s^k, depth $O(\log s)$ in basis G. Then, since we now have a formula of low depth, we can replace the gates of basis G_1 by the equivalent combination of gates of basis G_2 to yield a polynomially blown up formula that computes f

Lemma 5. (Depth reduction) *If f has a formula of size s using the basis G_1, then f also has a formula of size $F(s) = s^k$, depth $D(s) = O(\log S)$ using the basis G_1.*

Proof. The high level idea here is that we can find a vertex of the computation tree that splits the tree into two big parts (say P and Q, where Q depends on P) - we can then separately run computations for P,Q to simulate the entire computation of the original tree as follows : First, run the computation tree of P. Next, create two copies for the computation of the dependant part (Q) : Q_o, Q_1 - one each corresponding to the two possible outcomes of P. A selector formula is then set up that selects the appropriate copy of Q based on the result of the computation of P as $f = (Q_0 \wedge \neg P) \vee (Q_1 \wedge P)$. This ensures reduction in depth at the cost of repeated copies of the computation tree of Q - but, since we are allowed a polynomial blow up, this is tolerable for us.

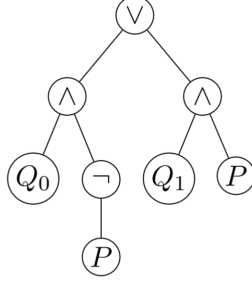

Figure 1: A selector formula for $f = (Q_0 \wedge \neg P) \vee (Q_1 \wedge P)$ in the DeMorgan basis

In order for the above idea to work, we need to show that there is a vertex in any tree that splits the tree into two trees of (roughly) similar sizes. The following fact guarantees the existence of such a vertex :

Fact 6. (Balanced splitting of a u-ary tree) *In any* u-ary tree *on n vertices, there exists a vertex such that the number of vertices on either side of the edge is at most* $(1 - \frac{1}{u})n$.

Proof. Pick the root in the tree. If removing the root splits the tree such that the number of vertices on each connected component is at most $(1 - \frac{1}{u})n$ as required, then we are done. Else, pick the adjacent vertex that belongs to the connected component that has more vertices (than desired) and check if this vertex works. Repeat this process until you find such a balancing vertex. □

Start from a leaf of the tree and keep moving in the direction of the larger connected component until a vertex achieves the balanced split guaranteed by fact 6. Apply the pre-computing trick described above to get a formula of size

$$F(s) \leq aF((1 - \frac{1}{u})s) + O(1)$$

$$\Rightarrow F(s) = O(s^k)$$

where the selector formula along with each of the components of the original tree incurs the $aF((1-\frac{1}{u})s)$ term. We basically prove the existence of such a construction by using induction on the size of the formula (details of inductive proof skipped to emphasise the idea rather than the proof). As for the depth of the new formula, we have the following recurrence based on the construction :

$$D(s) \leq D((1 - \frac{1}{u})s) + O(1)$$

$$\Rightarrow D(s) = O(\log S)$$

Note that it is important in the resolution of the recurrence that u is a constant.
This concludes proof of Theorem 4. □

Coming back to the Basis switching theorem, once we have a formula of size $O(s^k)$ and depth $O(\log s)$, then replacing each gate of the basis G_1 by the equivalent (combination of) G_2 gates and expanding, we have a new formula of size $O(s^k.2^{O(\log s)}) = O(s^{k'})$ (using the basis G_2). Hence, the polynomial blow-up in size in switching bases.