Lecture 1: Introduction to Boolean functions

Rajat Mittal

IIT Kanpur

The use and importance of binary numbers is clear to any student of computer science. We represent data (numbers, images, files) in computers as a string from $\{0,1\}^n$. Here, the numbers $0, 1$ can have multiple interpretations:

- as part of a number base 2,
- as encoding of TRUE (1) and False (0), and
- as elements of the field $\mathbb{F}_2$.

Exercise 1. What is the field $\mathbb{F}_2$?

An operation on these strings can be viewed as a function which takes a string from $\{0,1\}^n$ to $\{0,1\}$ or a sequence of such functions. A *Boolean function* takes as an input a string from $\{0,1\}^n$ and outputs either a 0 or a 1. We will represent a Boolean function f by $f : \{0,1\}^n \to \{0,1\}$, where n is called the *arity* of the Boolean function.

Given that we use binary representation to store information in the computer, these functions and their study has been a central part of theoretical computer science. They are the basic objects of study in circuit design, complexity theory, logic, cryptography and many more branches of computer science.

The main objective of the course is to introduce you to the analysis of Boolean functions. Through the course, we will try to answer following questions.

- What tools do we have to study Boolean functions?
- What are the properties of Boolean functions? How are they *special*?
- How to *measure* the complexity of Boolean functions? We will study one of the first and simplest way, called decision tree complexity.
- What are other mathematical notions of complexity for a Boolean function? How are these measures related to each other and to decision tree complexity?

We will see techniques to analyze these functions, this is known as Fourier analysis on Boolean hypercube. The analysis is very natural and interesting on its own; though, we will see applications in different domains. Later in the course, our focus will be on many complexity measures (combinatorial, computational, algebraic) which allow us to capture the complexity of these Boolean functions. In this course, we restrict ourselves to decision tree complexity and measures related to decision tree complexity. The relationship between these measures will also be studied in this course. Depending on time, we will cover some of the ways in which the lower bounds on these complexity measures impact computer science.

Through out the course, we will see that Boolean functions are *special*. They have nice structure and properties. This makes the study of Boolean functions not just interesting mathematically, but also gives rise to many applications in computer science. Next section will illustrate a few simple applications in the field of theoretical computer science.

1 Application(s)

We will see application of Boolean functions in two areas: error correcting codes and cryptography. Before we start, it will be helpful to set up some notation. A boolean function $f : \{0,1\}^n \to \{0,1\}$ takes as input a binary string x of length n and outputs a bit. The i-th position of x will be denoted by x_i.

One way to measure *distance* between two strings is called *Hamming distance*:

$$h(x,y) = |\{i : x_i \neq y_i\}|.$$

1.1 Error correcting codes

For this section we will view $\{0,1\}$ as field $\mathbb{F}_2$, i.e., the Boolean function is going to be $f : \mathbb{F}_2^n \to \mathbb{F}$. Even if you are not familiar with fields, it is enough to understand that addition and multiplication will be done mod 2.

Using this perspective, we can define *Linear functions*.

$$L_a(x) = \sum_i a_i x_i \mod 2.$$

One of the simplest linear function is addition modulo 2, also known as the PARITY function. For two strings of length k, we can define their addition by adding them at respective coordinates. Let us see an application of linear functions in the field of error correction.

Communicating data between two parties is a very routine task. Unfortunately, most of the channels of communication are *noisy* and lead to error. We will consider the model where at most t bits can be flipped when sent through channel in a message of length n, and the errors are random. The task is to still being able to communicate inspite of the errors. This is the problem solved by *error correcting codes*.

One of the most natural and simple example is a *repetition code*. Suppose at most 1 bit can be flipped out of 3. We can send 000 instead of 0 and 111 instead of 1. By taking majority at the receiving end, we will be able to get the correct message. This mapping from 0 to 000 and 1 to 111 is called the encoding.

In general, an error correcting code on alphabet $\{0,1\}^k$ is defined through the encoding function Enc : $\{0,1\}^k \to \{0,1\}^n$. Here n is larger than k (introducing redundancy), and the hope is that *the small errors* can be corrected by this encoding. The strings from $\{0,1\}^k$ are called messages and their enconding in $\{0,1\}^n$ are called the codewords.

Exercise 2. What are n, k for the 3-bit repetition code discussed above?

How much errors can we tolerate? Suppose in the range of Enc, any two codewords have Hamming distance at least d. Then, it is clear that if the channel performs less than d errors, we will be able to recognize that there is an error. If there are less than $d/2$ errors, there is a unique nearest codeword to the received message. In principle, we can correct these errors. Though, as a computer scientist, we look for efficient ways to encode and decode (correct) these messages.

Exercise 3. Prove the assertions above.

This minimum Hamming distance between two codewords is known as the *distance of the code* and is one of the most important parameters of the code. A code is specified by three parameters, n, k, d. For the code $Rep : \{0,1\} \to \{0,1\}^n$, where we just repeat 0 and 1 both n times, the distance is n, it is a $[n, 1, n]$ code.

Exercise 4. What is the problem with this code? Its distance is maximum possible.

Even though the distance is best, we are able to encode only 2 messages with strings of length n. The *rate* of the code is k/n, and we would like to keep that high too. Unfortunately, keeping the distance and rate high, go against each other. Let us ask a simpler question. Suppose we want the distance to be $n/2$, what rate can we achieve?

Exercise 5. What do you think rate could be?

We will now construct an optimal code using linear Boolean functions, called *Hadamard codes*. The idea is very simple. Let $n = 2^k$, $Enc_H(x) : \{0,1\}^k \to \{0,1\}^n$ will be the truth table of the linear function whose coefficients are from x, $f_x(z) = \sum_i x_i z_i \mod 2$. In other words, we take all possible 2^k dot products with x and that is its encoding.

Note 1. Since $n = 2^k$, there is a natural mapping between numbers from 0 to $n-1$ and binary strings of length k, what is that? We will use this mapping below and in the course a lot of times.

Lemma 1. *The distance of Hadamard code is $n/2$.*

Proof. Suppose there are two messages x, y, their corresponding codeword is given by the truth table of the functions f_x, f_y respectively. Remember that here the functions take inner (dot) product modulo 2 with the input, e.g., $f_y(z) = \sum_i y_i z_i \mod 2$.

We would like to show that the truth tables of f_x, f_y have distance $n/2$ if $x \neq y$. Since $x \neq y$, without loss of generality, assume that they differ at the first position. Divide the entire set of inputs into $n/2$ pairs such that inside any pair the inputs differ on just the first bit.

Exercise 6. Show that for exactly one of the inputs out of the pair z, z', the value of f_x, f_y is different.

This shows that the Hamming distance between the codewords for x, y is exactly $n/2$, proving the lemma. □

So Hadamard code is a $[n, \log n, n/2]$ code. What is its rate?

Another nice property of Hadamard code is that it can be corrected (randomized algorithm, succeeds with high probability) by a very simple algorithm. We can correct up to $n/4$ errors, half of the distance.

Lemma 2. *For a position $i \in [n]$ of the codeword y, it can be recovered from its erroneous copy z ($h(y, z) \leq n/4$) with high probability by using constant number of queries from z.*

Proof. The main idea behind the proof is: given the *correct codeword* y, and two strings of length k, say z, z',

$$y_{z+z'} = y_z + y_{z'} \mod 2$$

Exercise 7. Can you prove this easy idea? What should be the randomized algorithm?

We know that z can not be different from y at many places (at most $n/4$). Remember that any index $j \in [n]$ can be thought of as a bit string of length k. So if we pick a random place $j \in [n]$, then both $j, j + i$ will be same for y, z with high probability (at least $1/2$) . This implies that $z_j + z_{j+i} = y_j + y_{j+i} = y_i$ (notice that $j + (j + i) = i \mod 2$).

Exercise 8. Why is the probability $1/2$ and not better?

The idea of the algorithm is pretty simple. We pick a random j and output $z_j + z_{j+i}$, and this succeeds with probability $1/2$. To increase the probability, we can repeat this multiple times.

Exercise 9. How many times do you need to repeat to make the success probability better than $9/10$?

□

It can be shown that for the distance achieved, Hadamard codes have the best rate (though still pretty small for practical purposes). The idea to improve the rate further (Reed-Solomon codes) comes by taking polynomials over $\mathbb{F}_q$ (q bigger than 2) instead of linear functions; these codes are used in practice.

1.2 Extra reading: cryptography

In cryptography, one of the main problem studied is to transmit a message under the presence of an adversary, who can potentially intercept the message. This problem is tackled by *encrypting* the message at the sender's end and then *decrypting* the message at receiver's end. The encryption-decryption protocols for this problem can be broadly divided into two classes: symmetric key encryption and asymmetric key encryption.

- *Symmetric key encryption:* In this case, the encryption and decryption key are essentially the same (they are same or related by a simple transformation). One standard example is *one time pad* where a secret key is shared between the two parties. One time pad is a simple protocol and relies on the properties of the Boolean representation. We will look at it in detail below.

– *Asymmetric key encryption:* Not surprisingly, the encryption and decryption key are different in this case. They are known as public and private key. The sender encrypts the message using the public key of the receiver and only the receiver can decrypt the message using her own private key. This is also known as *public-key encryption*. The security of such protocols are dependent upon some computational problem being hard to solve. One example is the famous RSA algorithm. What is the computational assumption behind RSA algorithm?

The protocol for one time pad will use the Boolean function called XOR. The XOR of two bits x and y, denoted by $x \oplus y$, is 1 if and only if x, y are different. We can generalize this to bitwise XOR, where x, y are n bit strings and $x \oplus y$ denotes the n bit string which results from taking XOR at every position separately.

Exercise 10. What is the bitwise XOR of 1100100 and 0111010?

If your answer is not same as 10111110, then please look at the definition of bitwise XOR again.

Let us take a look at the protocol for one time pad. Let us call the two parties Alice and Bob, as is standard in cryptographic literature. Suppose, Alice wants to transmit a secret message $M \in \{0,1\}^n$ to Bob. We assume that Alice and Bob share a secret key $R \in \{0,1\}^n$ (symmetric key encryption) between them. Notice that the key R is obtained/shared by both parties before Alice receives M. That means, R does not have any relation to M.

Note 2. In general, this need for secret key is a major weakness of symmetric key encryption. We will talk about it later. At this point, let us assume its existence.

The protocol is very simple, Alice encodes the message into a code C by the operation $C = M \oplus R$ (here $\oplus$ denotes bitwise XOR). The decoding is straightforward too, it turns out that $M = C \oplus R$.

Exercise 11. Show that $M = C \oplus R$.

We need to show that this protocol is secure. What about an adversary who can intercept the communication between Alice and Bob? Can we say that adversary does not get any information about M from C (given that it has no information about R)?

We will show that for any two codewords C_1 and C_2, $\Pr[C_1|M] = \Pr[C_2|M]$. This shows that the codeword does not leak any information about M. Adversary has access to all the bits of the codeword C, say the i-th bit is c_i. We know that $c_i = m_i \oplus r_i$ (let m_i, r_i be i-th bits of M, R respectively).

Exercise 12. Suppose $m_i = 1$, what is the probability that $c_i = 1$? What about the other combinations?

From the above exercise, bit c_i has equal probability of being 0 or 1 irrespective of m_i being 0 or 1. In other words, the bits $c_i = m_i \oplus r_i$ are completely random, and provide no information about m_i. This shows that the one time pad protocol is unconditionally secure. It does not depend on any computational assumption. It only relies on the fact that the bitwise XOR completely shuffles the codeword, irrespective of the initial message.

Exercise 13. Suppose the message $M \in \{0,1\}^n$ is $000\cdots0$, what is the probability that the codeword sent is $111\cdots1$?

Unfortunately, the demand of having a secret key is big issue with this kind of encryption. This is the reason that many of the current cryptographic protocols use asymmetric key cryptography. Though, we also know that factoring can be solved in polynomial time on a quantum computer. This will result in the breakdown of many asymmetric key encryptions if a quantum computer (of decent size) comes into picture.

One option is to go back to symmetric key encryption. Fortunately, there exist a quantum protocol to generate a secret key between the two parties. This protocol was given by Bennett and Brassard, and is known as BB84 quantum key distribution protocol.

Exercise 14. What year was this protocol discovered in?

This was indeed a very simple example. We will see in this course that Boolean functions provide a very useful framework to capture and manipulate information. We have already seen an example in cryptography. We use Boolean functions to represent operations in circuits and Logic. A graph (in graph theory) can be represented as a binary string of length $\binom{n}{2}$, where n is the number of vertices. The properties of graphs (connectedness, existence of perfect matching, bipartite-ness etc.) can be viewed as a Boolean function. As mentioned before, the complexity of these functions play a central role in complexity theory. These functions can also be used to represent voting rules in social choice theory.

To understand these different fields, where information is represented by these functions, it is important to understand the structure and properties of Boolean functions. On the other hand, these properties/structure gives rise to many applications. The main objective of this course is to make you familiar with these special objects.

2 Boolean functions

We start by defining the Boolean function formally.

Definition 1. *A Boolean function is a function f from domain $\{0,1\}^n$ to range $\{0,1\}$, represented by $f : \{0,1\}^n \to \{0,1\}$. The number n (number of inputs) is also called the arity of the function. The most natural way to represent it is in terms of a* truth table*, where we explicitly write down the value of the function on all 2^n possible inputs.*

To take an example, let us take a function on $\{0,1\}^2$ which output 1 if and only if both inputs are 1. You might know it already, it is the AND function. The truth table for this function will be,

Input 1	*Input 2*	*Answer*
0	0	0
0	1	0
1	0	0
1	1	1

You might have seen other Boolean functions like OR, majority and parity. If not, we will define them later in this section. You might have seen them defined on a different domain, having a different representation.

Exercise 15. What is the number of distinct Boolean functions with arity n?

Notice that the representation in terms of a string of 0 and 1 is not unique; many a times 1 and -1 are used to represent and analyse Boolean functions. These representations are similar, most of the time we can move from one representation to another with some simple and natural transformations.

Exercise 16. When we generally switch from $\{0,1\}$ to $\{-1,1\}$ representation, 0 is mapped to 1 and 1 is mapped to -1. Can you think of a reason? (Hint: Group theory)

For n variables, let us keep the domain of our functions to be $\{0,1\}^n$ for now. A general element of this domain (an input for a Boolean function) will mostly be denoted by lower case alphabets (x, y or z). With such a representation, we will use x_i to denote the i-th "co-ordinate" (position) of the input/element x.

One of the important concept for this Boolean domain is known as *Hamming distance*. For two strings $x, y \in \{0,1\}^n$, the Hamming distance between them is defined to be the number of places where they differ. The *Hamming weight* of a string x is the number of 1's in the string. In $\{-1,1\}$ representation, it becomes the number of -1's.

The Boolean domain, $\{0,1\}^n$, with edges connecting elements at Hamming distance 1 is called the *Boolean hypercube*. (It is called *Hamming cube* too.)

Using the definition of Hamming weight, we can generalize the definition of AND function. Function $\text{AND} : \{0,1\}^n \to \{0,1\}$ is defined to be 1 if and only if the Hamming weight of the input is n.

In general, a Boolean function needs an n to be specified before it is defined. Though, most of the time, we will be interested in functions which can be generalized to any n, like the case of AND function. In such a situation, the function name (say AND) will refer to the set of these functions for all n. By the complexity of these kind of functions, we would mean the asymptotic complexity of these functions in terms of n.

2.1 Representations of Boolean functions

We have seen the truth table representation of a Boolean function. If the ordering of inputs is fixed, the Boolean function can be thought of as a list of length 2^n. Depending on the situation, this list can be taken as a vector with real elements or a vector with elements in $\mathbb{F}_2$.

Similar to AND function, we can define OR, it is 0 if and only if the Hamming weight is 0. Considering the OR function for two bits, and taking the order of inputs to be $00, 01, 10, 11$, the vector corresponding to it is,

$$\begin{pmatrix} 0 \\ 1 \\ 1 \\ 1 \end{pmatrix}$$

Notice the difference between OR function and the usual English meaning of the term *or*. The function corresponding to the English term is the PARITY/XOR of two bits.

$$\begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \end{pmatrix}$$

We will denote the negation of a string by $\bar{x}$,

$$\forall i: \ x_i = 1 - \bar{x_i}.$$

Exercise 17. Convince yourself that $\mathrm{OR}(x) = 1 - \mathrm{AND}(\bar{x})$.

Another similar representation would be to just list the 1 inputs of a function, e.g., AND will be represented by $111 \cdots 1$.

You might already be familiar with the notation used in logic, $\mathrm{AND}(x_1, x_2, \cdots, x_n)$ is denoted by $x_1 \wedge x_2 \cdots \wedge x_n$ and $\mathrm{OR}(x_1, x_2, \cdots, x_n)$ is denoted by $x_1 \vee x_2 \cdots \vee x_n$. Every Boolean function can be written as an OR of AND, this is called the DNF (disjunctive normal form) representation. You will show in the exercise that every Boolean function can be written in the DNF representation.

You can also represent Boolean functions as polynomials (over reals and $\mathbb{F}_2$). We will study this representation in detail later.

Notice that the truth table representation and the vector representation always required around 2^n size. On the other hand, list, DNF and polynomial representation might be quite short for some functions.

Exercise 18. Show that any representation of a Boolean function will require 2^n size in the worst case.

The size of a particular representation for a function in itself is a measure of its complexity. We will study some of these measures later.

Even though we have listed many representations, the most convenient one is the description of a function in natural language. For example, MAJORITY function outputs 1 if the number of 0's are smaller than number of 1's in the input. You might be worried about the case when number of 0's is same as number of 1's. Since we are mostly interested in asymptotic complexity, this question can be ignored.

Exercise 19. Suppose n is odd, what is the number of inputs which output 1 for MAJORITY?

Another interesting function is called PARITY, it is 1 if the number of 1's in the input are odd. PARITY generalizes the It has a very simple representation as a polynomial in the $\{-1, 1\}$ world,

$$\text{PARITY}(x) = x_1 x_2 \cdots x_n.$$

Exercise 20. Which function has the same representation in the $\{0, 1\}$ world?

Note 3. In this case, function will be -1 if the number of -1's in the input are odd.

Have you already seen a function like parity, what about $n = 2$? Let us see a lower bound on the size of a representation for the parity function.

Exercise 21. What will be the minimum number of clauses needed to represent PARITY in DNF representation?

We will show that at least 2^{n-1} clauses are needed. The proof requires two claims:

- Any clause in the representation of PARITY will require all n variables. Assume that there are only $n-1$ variables in the clause (say x_n is not present). Then there is an assignment of $x_1, x_2, \cdots, x_{n-1}$ such that the function value is 1 irrespective of the value of x_n. This is a contradiction, because you can always set x_n in a different way to get different function value.
- A clause with n variable can only cover one true assignment.

Exercise 22. Prove this.

It is an easy exercise to finish the proof using the two claims (parity has 2^{n-1} true assignments).

This was a toy example, where we gave a lower bound on a certain complexity measure (size of DNF). We will see many such lower bounds throughout this course.

2.2 Symmetric Boolean functions

A very important subclass of Boolean functions is called *symmetric functions*. A function $f : \{0, 1\}^n \to \{0, 1\}$ is called symmetric if the function value only depends on the Hamming weight of the input.

Exercise 23. Show that the functions AND, OR, MAJORITY, PARITY are all symmetric.

The specification of a Boolean function becomes much simpler if it is known that the function is symmetric. We only need to specify the value of the Boolean function at each level of Boolean hypercube (one level corresponds to elements of same Hamming weight). Since there are $n + 1$ levels, a symmetric function can be viewed as a vector of dimension $n + 1$.

The class of symmetric functions is not just important because many interesting functions belong to it, but also because it is arguably much simpler to study. We will see that many of the complexity measures are easier to figure out in case of symmetric functions.

Given the discussion up to this point, you might be tempted to believe that all interesting functions are symmetric. Let us introduce a very natural non-symmetric function, called *addressing* function.

For a parameter m, the function ADDR_m is defined on an input of size $n = m + 2^m$. The first m bits should be considered as an address, hence the name, out of 2^m positions. The output of the function is the value of the position (out of last 2^m bits) referred by the address.

Exercise 24. Say $m = 2$, what is the value of ADDR(101001)? (assume the lexicographic ordering of addresses)

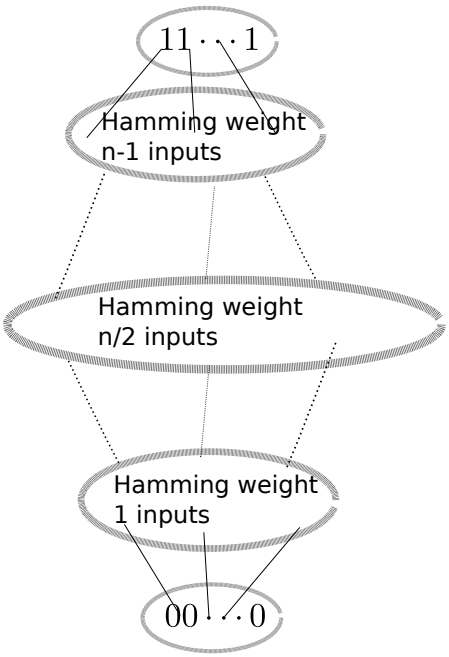

Fig. 1. A graphical representation of Boolean hypercube. Two vertices are joined iff they are at Hamming distance 1.

2.3 Composition of Boolean functions

Another interesting way to create Boolean functions is using *composition* of two Boolean functions. The idea is that the input of the *outer* Boolean function is generated by applying *inner* function on the input of the composed function. In other words, if f is the outer function and g is the inner function, we first apply g to different subsets of the input and then apply f to the resultant string. We have already seen an example, DNF representation. The outer function was OR and inner function was AND in that case.

The definition given above is very general. There can be several restrictions while composing two Boolean functions.

- Are the subsets (on which g is applied) disjoint?
- Are the arity of all such subsets same?
- What is the relation between arity of f and arity of g?

For our purposes, we will denote the composition of two functions $f : \{0,1\}^m \to \{0,1\}$ and $g : \{0,1\}^n \to \{0,1\}$ by $f \circ g$. The input of $f \circ g$ will have mn bits. It will be easier to represent them with m blocks of size n, where i-th block has inputs $x^i = \{x_{i1}, x_{i2}, \cdots, x_{in}\}$. Then,

$$f \circ g(x) = f(g(x^1), g(x^2), \cdots, g(x^m)).$$

Notice that all subsets (on which g is applied) are disjoint and have the same size. Sometimes we will denote the composition as $f_m \circ g_n$ to clarify that f is function with arity m and g with n.

A standard example of composition is called *tribes*. It is defined as OR $\circ$ AND. You can think of it as inputs divided into tribes (over which AND is taken). Then tribes can be thought of as a voting scheme where the object is chosen if and only if one tribe unanimously selects the object.

Exercise 25. What is OR $\circ$ OR? What about parity composed with parity? Is MAJORITY $\circ$ MAJORITY a majority function?

3 Assignment

Exercise 26. Read about the DNF and CNF representation. Show that every function can be written as a DNF with at most 2^n terms (using truth table).

Exercise 27. How many symmetric functions are there of size n?

Exercise 28. Construct two inputs to show that ADD is not symmetric.

Exercise 29. Construct two inputs to show that tribes is not symmetric.

Exercise 30. Give a decoding algorithm for Hadamard code using k queries.

Exercise 31. Read about group theory and characters of finite Abelian groups.