Lecture 4: Decision tree complexity

Rajat Mittal

IIT Kanpur

This lecture note will introduce the first complexity measure coming from the computational world. The complexity measure is called *decision tree complexity* and is arguably the simplest measure for a Boolean function. In particular, there are concrete techniques and results about lower bounds on this complexity measure (as opposed to circuit complexity).

We will introduce decision trees complexity measure, see some examples and construct lower bounds in the first section. The same measure can be studied in the randomized computational model. These measures are also known as *deterministic query complexity* and *randomized query complexity* respectively.

1 Deterministic decision trees

The decision tree model (or equivalently query model) assumes that we have access to the input x through an *oracle*. At any point, the algorithm can ask (query) the oracle the value of bit x_i. The task of the algorithm is to find the value of $f(x)$ and minimize the number of queries to the oracle. Such an algorithm can be represented by a *decision tree*.

Let $f : \{0,1\}^n \to \{0,1\}$ be a Boolean function. A *decision tree* (or a decision tree algorithm) for f, on input x, *queries* some subset of the bits of the input $(x_1, x_2, \cdots, x_n)$ and then outputs the value of $f(x)$. The k-th query of the algorithm can depend on the result of the first $k-1$ queries, this is called the *adaptive* query model.

We can depict the algorithm as a binary tree, where each node contains the index queried and two children correspond to the query output being 0 or 1. The leaves denote the output of the function. For example, a decision tree for OR will look like Figure 1.

The complexity of a decision tree is the depth of this tree. In the previous example, the complexity is n.

Exercise 1. Can there be more than one decision tree for a function? What about OR?

The deterministic decision tree complexity of a function f, denoted by $\mathrm{D}(f)$, is the minimum complexity of a decision tree computing f. We will look at different models in the next section and the term *deterministic* will be clear then. In all these models, we would like to give lower bound on the complexity of different Boolean functions.

Exercise 2. What is the maximum possible $\mathrm{D}(f)$ for any function f in terms of n?

We have already seen a decision tree for OR, its depth was n. Can there be a better decision tree, with complexity less than n?

We can easily construct an *adversary* argument to show that $\mathrm{D}(\mathrm{OR})$ is n. Suppose there was a decision tree, computing OR, with depth $n-1$; let us create an input where the decision tree will not output the correct value.

Look at the root node and assign it 0, follow the 0 branch and assign the next variable to be 0, keep following the path (assigning 0 to every variable) till we reach the leaf. Since the depth is less than n, there is a free variable (its value is not assigned). By fixing the value of that variable to be 0 or 1, the OR of the constructed input can be made 0 or 1. Though, the decision tree will give the same value for both of these fixings. In other words, the decision tree does not compute OR.

The adversary argument can be formalized. If any d chosen queries (even adaptively) can be answered in such a way that the function value is not determined, then d is a lower bound on the deterministic decision tree complexity. Let $l\mathrm{D}(f)$ be the optimal lower bound using this argument. This means, $l\mathrm{D}(f)$ is the maximum d such that any adaptively chosen d queries can be answered keeping the function value undetermined.

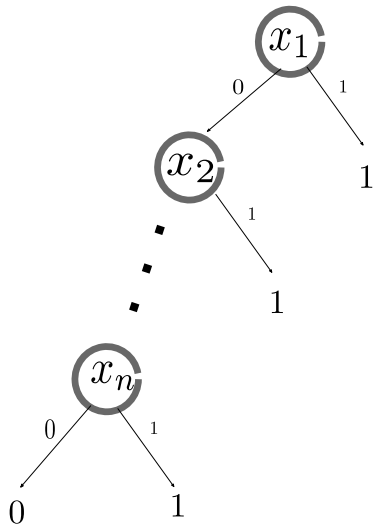

Fig. 1. A possible decision tree for OR.

Exercise 3. Show that $l\mathrm{D}(f) \leq \mathrm{D}(f)$.

For $d = l\mathrm{D}(f) + 1$, there is a set of adaptive queries (of size at most d) which will determine the function value. These queries give a decision tree of depth d. So, the adversary argument is tight. For any $d < \mathrm{D}(f)$, any d adaptively chosen queries can be answered in such a way that the function value is not fixed.

Let us consider the decision tree complexity of Graph connectivity problem. For this problem, the input is a string of length $\binom{n}{2}$, indexed by edges, each bit specifying whether the corresponding edge is present or not. The Graph connectivity problem tell us whether the graph is connected or not. We will show that the deterministic tree complexity of this problem is $\binom{n}{2}$.

Exercise 4. What will the strategy for adversary?

You might want to keep the graph *barely connected*. The intuition would be to answer 0 except if the tree becomes disconnected. Let us make it more formal. Say, at any point, not queried edges are blue, and if the edge is queried and kept then it is red. The goal is to keep the graph (with blue and red edges) barely connected. That means when a blue edge is queried, we will make it red if and only if the removal of it makes the graph disconnected.

In other words, the answer will be 0 if there is a path between the vertices of the queried edge using red and blue edges (obviously ignoring the queried edge). From the strategy, if we include the last edge, the graph will be connected.

Exercise 5. Prove, using induction, if all the edges are not queried then the red edges can't make a cycle.

It only remains to be proven that we don't get a spanning tree before the last edge. Suppose we do get a spanning tree and some blue edges are left. Let e be one of the blue edges, we know that e will create a cycle with the spanning tree.

Exercise 6. Show that the last edge added to the cycle before e should not have been added.

So, this problem also turns out to have full $\mathrm{D}(f)$ (equal to the number of variables). You might wonder if all problems have full $\mathrm{D}(f)$. You will show in the assignment that for addressing function on $n + 2^n$ bits, the deterministic decision tree complexity is $n + 1$.

Relation with degree: We will try to relate the two complexity measures seen, degree and deterministic decision tree. The easy direction is to show that for all Boolean functions:

$$\mathrm{D}(f) \geq deg(f).$$

Assume that f is from $\{0,1\}^n$ to $\{0,1\}$, this does not change the degree or $\mathrm{D}(f)$.

Exercise 7. Given a decision tree for a Boolean function $f : \{0,1\}^n \to \{0,1\}$, can you construct a polynomial representation for it?

The technique is very similar to interpolation (as done for truth table). We will create a indicator polynomial for every leaf. Summing up indicator polynomials for leaves labelled 1 will give us the polynomial. You will complete this argument in the assignment.

Exercise 8. Do you know a function where $\mathrm{D}(f) = deg(f)$?

In the other direction, it is known that $\mathrm{D}(f) = O(deg(f)^3)$. The proof of this will be covered later in the course. Though, this relation *need not* be tight. We only know a function f such that $\mathrm{D}(f) = \Omega(deg(f)^2)$, and nothing better.

In general, we can say a lot about the Fourier spectra of decision tree. Suppose the decision tree has depth k and size s (number of leaves).

Exercise 9. What is the upper bound on s in terms of k?

We already know that the degree is at most k. From the previous notes, we also know that every Fourier coefficient is a multiple of $1/2^k$ (granularity is $1/2^k$). From the way you constructed polynomial from the decision tree, the number of non-zero Fourier coefficients (called *sparsity*) is bounded by $s2^k$. From this representation you can also prove the following.

Exercise 10. Prove that $\sum_S \left|\widehat{f}(S)\right| \leq s \max_x |f(x)|$.

A more difficult exercise is to prove that the Fourier spectrum of f is ϵ concentrated on degree up to $\log(\frac{s}{\epsilon})$ [2].

Composition of D*:* The degree lower bound allows us to lower bound the deterministic tree complexity of composed functions. Clearly, $\mathrm{D}(f \circ g) \geq deg(f \circ g)$.

Exercise 11. Show that

$$deg(f \circ g) = deg(f)deg(g).$$

The adversary method can be used to show that deterministic query complexity composes nicely (for example see [4][Lemma 3.1]),

$$\mathrm{D}(f \circ g) = \mathrm{D}(f)\mathrm{D}(g).$$

2 Randomized query complexity

The decision tree model can be extended to the randomized computation paradigm too. Here, the task is to come up with a *bounded error algorithm*. If you are not familiar, it is described in the next section.

2.1 Randomized algorithms

Most of the introductory algorithms you might have seen are all deterministic algorithms, e.g., sorting and network flows. That means, we *always* get the *exact* answer when the algorithm ends. The task there is to put a bound on the running time of such an algorithm.

We can relax algorithm's task a bit, with the input x, algorithm A can also use a random string (say $r \in \{0,1\}^k$ for some k chosen by algorithm). Hence, algorithm A is a function of the combined input x and r. A randomized/probabilistic algorithm is called to be successful, if for every x it outputs the correct answer *for most of the random strings* r.

Definition 1 (Bounded error randomized algorithm). *A randomized algorithm A accepts the function f in the bounded error model if,*

- *If $f(x) = 1$, then $\Pr_r(A(x,r) = 1) \geq \frac{2}{3}$.*
- *If $f(x) = 0$, then $\Pr_r(A(x,r) = 0) \geq \frac{2}{3}$.*

There are many ways to come up with a random string in the classical case, coin-toss, clock or many others. Note that the probability is computed only over the random strings used in the algorithm. In other words, the probability condition is true for all inputs x (and not just on a high fraction of inputs).

For instance, if we just want our algorithm to work on large fraction of inputs, since number of primes are small, it is easy to come up with an algorithm for testing if a number is prime. We will not call such an algorithm a bounded error algorithm.

The constant $\frac{2}{3}$ in the definition is not important, any constant strictly greater than $\frac{1}{2}$ will work. In this case, we will repeat the algorithm k times and take the majority vote to decide the output. Suppose the original algorithm (the one we are repeating) gives the correct answer with probability more than $\frac{1}{2} + \epsilon$. We assume that ϵ is a constant. Then after repeating it k times, using Chernoff bound, the probability that we get the wrong answer is less than

$$e^{-2\epsilon^2 k}.$$

Exercise 12. Read about Chernoff bound.

Note 1. The number of times we need to repeat the algorithm, k, in bounded error algorithm is independent of size of input and only depends on probability we want to achieve. So, k will be a constant.

2.2 Query complexity for randomized algorithms

Exercise 13. What should be the extension of decision tree in randomized computation model.

In a deterministic decision tree, once we query a bit, there are two possible queries in the next step (depending on the query output). For the randomized case, we can decide on the bit to query depending upon the output of the previous queries and a random string. Finally the computation should answer correctly according to the bounded error model.

There is an easier way to think about a randomized decision tree.

Exercise 14. Using induction, show that a randomized decision tree is equivalent to having a probability distribution over the space of all decision trees.

So there are two ways to think about a randomized query algorithm. One way is to think of a random string as additional input, where there is no cost of querying the random string. Other way is to think of the algorithm as a *randomized decision tree*, a probability distribution over deterministic decision trees.

Definition 2 (Randomized decision tree for f). *A randomized decision tree R is a probability distribution μ over the space of decision trees. R accepts the function f in the bounded error model if,*

$$\Pr_{D\sim\mu}(D(x) = f(x)) \geq \frac{2}{3}.$$

The depth (complexity) of R is the depth of the largest decision tree with nonzero probability.

Let $\text{cost}(R, x)$ be the expected number of queries on x by the randomized decision tree R. The cost of a randomized decision tree is the cost of R on the worst case input x.

Like in the case of deterministic decision tree complexity, the randomized decision tree complexity of an f, denoted $\text{R}(f)$, is the minimum possible complexity of a randomized decision tree computing f.

$$\text{R}(f) = \min_{R \text{ computing } f} \max_{x} \text{cost}(R, x).$$

Here, R is a randomized decision tree.

In many places the cost of R on x is defined to be the worst case length (over all decision trees with positive probability) of the path taken by the decision tree on x (instead of the expected path length defined by us). Since we are working with constant error (2/3), it is known that this cost do *not* give rise to a different complexity measure (both measures are related by constant). This is shown by cutting the algorithm after it has run for constant times the expected cost and using Markov's inequality.

We have seen two ways to lower bound the deterministic query complexity, adversary approach and degree lower bound. Both methods can be extended to the randomized world. We will see these extensions in future lectures.

2.3 Separation between randomized and deterministic query complexity

The randomized model would not be interesting if we can get only constant factor speedups (as compared to deterministic model).

Exercise 15. Show a function where randomized query complexity is a constant factor better than deterministic query complexity.

There are many examples known, we will give an example from [3] (attributed to Ravi Bopanna).

Example (Bopanna): Let $f_d := Maj_3^d$; in other words, it is the recursive composition of the majority function on 3 bits.

Exercise 16. Show that the deterministic query complexity of Majority function is full.

Since D composes perfectly, we get that $\text{D}(f_d) = 3^d$. We first construct an algorithm which always works, but its expected query complexity is *small* (such algorithms are *randomized Las Vegas algorithms*, we will not focus on them in this course). This Las Vegas algorithm will be converted into a randomized query complexity algorithm.

Note 2. The randomized algorithms formulated in this course are known as *randomized Monte Carlo algorithms*. You can always convert a Las Vegas algorithm into a Monte Carlo algorithm with a constant blowup in cost.

Exercise 17. How will you solve f_d using randomness, so that its expected cost is small for every input?

Notice that f_d can be viewed as a tree with d depth, each node having 3 children. The strategy seems pretty natural. At any point in the tree (node), we randomly pick two children and compute the output. If both of them give the same value, we don't need to compute the third children and save cost in expectation.

Let $c(f_d, x)$ be the expected cost of this strategy on input x (and $c(f_d)$ be the worst case expected cost). If input x gives the same value to all 3 children,

$$c(f_d, x) \leq 2c(f_{d-1}).$$

The expensive case is when two of the children have the same value, but the third one is different,

$$c(f_d, x) \leq (1/3)\,(2c(f_{d-1})) + (2/3)\,(3c(f_{d-1})) \leq (8/3)\,c(f_{d-1}).$$

In other words,

$$c(f_d) \leq (8/3)\,c(f_{d-1}).$$

So, the expected cost is $(\frac{8}{3})^d$ instead of 3^d, a significant saving. The following trick converts expected cost into worst cost. The cost increases by a constant factor, and error increases by a little bit (can be made small by repetition).

Suppose the expected cost is E. Run the old algorithm $10E$ times. If the algorithm stops, we answer according to the old algorithm. If it does not stop, we answer arbitrarily. By Markov's inequality, the algorithm will not stop with probability at most $1/10$. So the new algorithm's worst case cost is 10 times the original. The error has only increased by an additive factor of $1/10$ (assuming we always answer incorrectly when answering arbitrarily).

This is a general strategy for converting Las Vegas algorithm into a Monte Carlo algorithm. So we can make the usual randomized algorithm (Monte Carlo) with cost $O((\frac{8}{3})^d)$.

Exercise 18. Notice that $n = 3^d$, where n is the number of variables. We know $\mathrm{D}(f_d) = n$, express $\mathrm{R}(f_d)$ in terms of n.

Another example can be constructed by recursively combining $NAND$ function on 2 bits [3].

3 Quantum query complexity

The analogue of query complexity has been very useful in the quantum world. Most of the lower bounds on time complexity actually arise from lower bounds on query complexity (for example, Grover search). We try to give an introduction to the quantum query model, without delving much into quantum computing.

First, the randomness is inherent in quantum computing. There are many ways to come up with a random string (introduce randomness) in the classical case, coin-toss, clock or many others. In case of quantum computing, there need not be a random string, the probability will be taken over the inherent randomness of quantum states and measurements.

Definition 3 (Bounded error quantum algorithm). *An quantum algorithm A accepts the function $f : \{0,1\}^n \to \{0,1\}$ in the bounded error model if,*

- *If $f(x) = 1$, then $\Pr(A(x) = 1) \geq \frac{2}{3}$.*
- *If $f(x) = 0$, then $\Pr(A(x) = 0) \geq \frac{2}{3}$.*

For the cost in the quantum case, we allow quantum queries and accept the result in the bounded error model. Since quantum allows us to do queries in superposition, it does not give us a tree structure. So, we call such an algorithm a *quantum query algorithm*. The complexity of a quantum query algorithm is the number of queries done on the worst case input.

Define $\mathrm{cost}(Q, x)$ to be the maximum queries needed to compute x by a quantum query algorithm Q. The *quantum query complexity* of an f, denoted $\mathrm{Q}(f)$, is the minimum possible complexity of a quantum query algorithm computing f.

$$\mathrm{Q}(f) = \min_{algorithm\ Q} \max_{x} \mathrm{cost}(Q, x).$$

We move to the structure of a quantum query algorithm.

A generic algorithm in the query framework:

In the classical case, we can ask the oracle question i and it replies by x_i. Unfortunately, this oracle is not reversible, and hence not possible in the quantum world. Though, it is possible to modify this oracle and make it reversible. We will skip the details, and just remember that,

- On a quantum computer, state of the algorithm is a vector and gate is a matrix. A quantum oracle exists and is a unitary matrix (like all quantum gates).
- We can query indices in *superposition*.

Using this superposition intelligently (not just computing in parallel), allows us show separations between quantum query complexity and randomized query complexity. One famous example is Grover search [1], which shows that you can search in an unordered list of n elements with $\sqrt{n}$ quantum queries.

Let us see how a generic algorithm progresses in this framework of queries from an oracle. Any generic algorithm will start with a state ψ and apply unitaries (independent of input) and oracle (dependent on input) one after another. If there are l oracle queries, the final state can be written as,

$$\psi_l^x = U_l O_x U_{l-1} O_x \cdots U_1 O_x \psi.$$

Note 3. This representation is similar to decision tree in the sense that the nodes are like oracle queries, and the edges represent calculations (potentially computationally intensive) we don't care about (at least for lower bounds).

Most of the lower bound techniques quantify how O_x changes the state of the algorithm. Since the unitaries do not depend on x, they do not count in the complexity of the algorithm.

4 Assignment

Exercise 19. Show that the decision tree complexity of any non-constant symmetric function is n.

Exercise 20. Show that $\mathrm{D}(\mathrm{ADDR}_m) = m + 1$.

Exercise 21. Show that $\mathrm{D}(f \circ g) \leq \mathrm{D}(f)\mathrm{D}(g)$.

Exercise 22. Show that $\mathrm{D}(f) \geq deg(f)$. This will be done by making a polynomial from a decision tree with degree at most the depth of the decision tree.

Exercise 23. Is there an easier way to increase the success probability of a randomized algorithm if it errs on one side only?

Exercise 24. Show that the two ways of thinking about a ranodmized algorithm, having a random string as an input or a distribution over deterministic decision trees, are equivalent. Hint: you can use induction on number of random bits for a clear proof.

References

1. Lov Grover. Fast quantum mechanical algorithm for database search. *Proceedings of the 28th Annual ACM Symposium on the Theory of Computing*, 06 1996.
2. Ryan O'Donnell. *Analysis of Boolean Functions.* Cambridge University Press, 2014.
3. M. Saks and A. Wigderson. Probabilistic boolean decision trees and the complexity of evaluating game trees. *27th Annual Symposium on Foundations of Computer Science (sfcs 1986), Toronto, ON, Canada, 1986*, pages 29–38, 1986.
4. A. Tal. Properties and applications of boolean function composition. *ITCS '13: Proceedings of the 4th conference on Innovations in Theoretical Computer Science, Pages 441 - 454*, 2013.