Decision Tree Complexity

Sanjeev Arora

Princeton University

1 Decision Trees

A *decision tree* is a model of computation used to study the number of bits of an input that need to be examined in order to compute some function on this input. Consider a function $f : \{0,1\}^n \rightarrow \{0,1\}$. A decision tree for f is a tree for which each node is labelled with some x_i, and has two outgoing edges, labelled 0 and 1. Each tree leaf is labelled with an output value 0 or 1. The computation on input $x = x_1 x_2 \ldots x_n$ proceeds at each node by inspecting the input bit x_i indicated by the node's label. If $x_i = 1$ the computation continues in the subtree reached by taking the 1-edge. The 0-edge is taken if the bit is 0. Thus input x follows a path through the tree. The output value at the leaf is $f(x)$. An example of a simple decision tree for the majority function is given in Figure 1

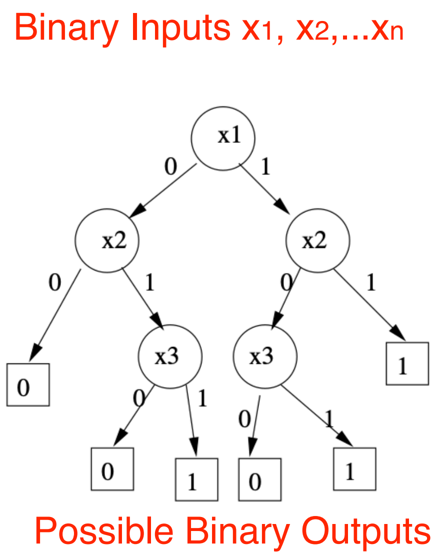


Figure 1: A decision tree for computing the majority function $Maj(x_1, x_2, x_3)$ on three bits. Outputs 1 if at least two input bits are 1, else outputs 0.

Recall the use of decision trees in the proof of the lower bound for comparison-based sorting algorithms. That study can be recast in the above framework by thinking of the input —which consisted of n numbers — as consisting of $\binom{n}{2}$ bits, each giving the outcome of a pairwise comparison between two numbers.

We can now define two useful decision tree metrics.

Definition 1 *The cost of tree t on input x, $cost(t, x)$, is the number of bits of x examined by t.*

Definition 2 *The* **decision tree complexity** *of function f, $D(f)$, is defined as follows, where T below refers to the set of decision trees that decide f.*

$$D(f) = \min_{t \in T} \max_{x \in \{0,1\}^n} cost(t, x) \tag{1}$$

The decision tree complexity of a function is the number of bits examined by the most efficient decision tree on the worst case input to that tree. We are now ready to consider several examples.

Example 1 (*Graph connectivity*) Given a graph G as input, in adjacency matrix form, we would like to know how many bits of the adjacency matrix a decision tree algorithm might have to inspect in order to determine whether G is connected. We have the following result.

Theorem 1
Let f be a function that computes the connectivity of input graphs with m vertices. Then $D(f) = \binom{m}{2}$.

The idea of the proof of this theorem is to imagine an adversary that constructs a graph, edge by edge, in response to the queries of a decision tree. For every decision tree that decides connectivity, the strategy implicitly produces an input graph which requires the decision tree to inspect each of the $\binom{m}{2}$ possible edges in a graph of m vertices.

Adversary Strategy:
Whenever the decision tree algorithm asks about edge e_i,
answer “no” unless this would force the graph to be disconnected.

After i queries, let N_i be the set of edges for which the adversary has replied “no”, Y_i the set of edges for which the adversary has replied “yes”. and E_i the set of edges not yet queried. The adversary's strategy maintains the invariant that Y_i is a disconnected forest for $i < \binom{m}{2}$ and $Y_i \cup E_i$ is connected. This ensures that the decision tree will not know whether the graph is connected until it queries every edge.

Example 2 (*OR Function*) Let $f(x_1, x_2, \ldots x_n) = \bigvee_{i=1}^{n} x_i$. Here we can use an adversary argument to show that $D(f) = n$. For any decision tree query of an input bit x_i, the adversary responds that x_i equals 0 for the first $n-1$ queries. Since f is the OR function, the decision tree will be in suspense until the value of the nth bit is revealed. Thus $D(f)$ is n.

Example 3 Consider the AND-OR function, with $n = 2^k$. We define f_k as follows.

$$f_k(x_1, \ldots, x_n) = \begin{cases} f_{k-1}(x_1, \ldots x_{2^{k-1}-1}) \wedge f_{k-1}(x_{2^{k-1}}, \ldots x_{2^k}) & \text{if } k \text{ is even} \\ f_{k-1}(x_1, \ldots x_{2^{k-1}-1}) \vee f_{k-1}(x_{2^{k-1}}, \ldots x_{2^k}) & \text{if } k > 1 \text{ and is odd} \\ x_i & \text{if } k = 1 \end{cases} \tag{2}$$

A diagram of a circuit that computes the AND-OR function is shown in Figure 2. It is left as an exercise to prove, using induction, that $D(f_k) = 2^k$.

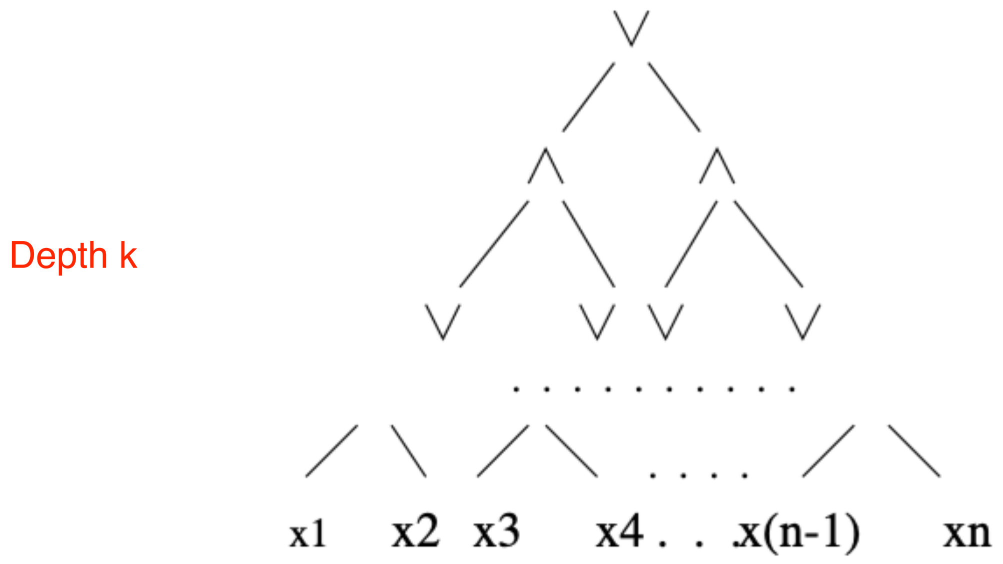


Figure 2: A circuit showing the computation of the AND-OR function. The circuit has k layers of alternating gates, where $n = 2^k$.

2 Certificate Complexity

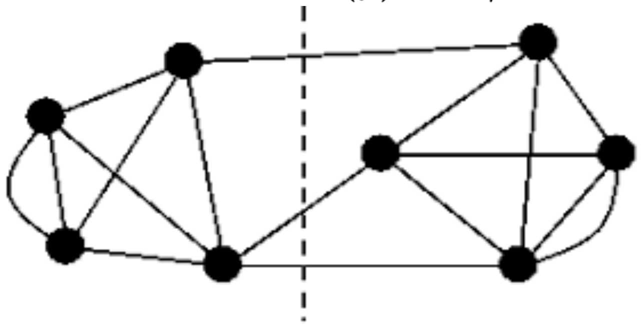
We now introduce the notion of *certificate complexity*, which, in a manner analogous to decision tree complexity above, tells us the minimum amount of information needed to be convinced of the value of a function f on input x.

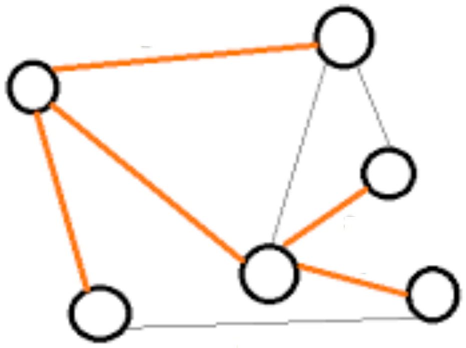
DEFINITION 3 *Consider a function $f : \{0,1\}^n \to \{0,1\}$. If $f(x) = 0$, then a* **0-certificate** *for x is a sequence of bits in x that proves $f(x) = 0$. If $f(x) = 1$, then a* **1-certificate** *is a sequence of bits in x that proves $f(x) = 1$.*

DEFINITION 4 *The* **certificate complexity** *$C(f)$ of f is defined as follows.*

$$C(f) = \max_{x:input} \{\textit{number of bits in the smallest 0- or 1- certificate for } x\} \tag{3}$$

EXAMPLE 4 If f is a function that decides connectivity of a graph, a 0-certificate for an input must prove that some cut in the graph has no edges, hence it has to contain all the possible edges of a cut of the graph. When these edges do not exist, the graph is disconnected. Similarly, a 1-certificate is the edges of a spanning tree. Thus for those inputs that represent a connected graph, the minimum size of a 1-certificate is the number of edges in a spanning tree, $n-1$. For those that represent a disconnected graph, a 0 certificate is the set of edges in a cut. The size of a 0-certificate is at most $(n/2)^2 = n^2/4$, and there are graphs (such as the graph consisting of two disjoint cliques of size $n/2$) in which no smaller 0-certificate exists. Thus $C(f) = n^2/4$.


Edge Cut of a Graph. It is a 0-certificate if it has no edges


Spanning Tree of a Graph. It is a 1-certificate if it contains all the vertices

EXAMPLE 5 We show that the certificate complexity of the AND-OR function f_k of Example 3 is $2^{\lceil k/2 \rceil}$. Recall that f_k is defined using a circuit of k layers. Each layer contains only OR-gates or only AND-gates, and the layers have alternative gate types. The bottom layer receives the bits of input x as input and the single top layer gate outputs the answer $f_k(x)$. If $f(x) = 1$, we can construct a 1-certificate as follows. For every AND-gate in the tree of gates we have to prove that both its children evaluate to 1, whereas for every OR-gate we only need to prove that *some* child evaluates to 1. Thus the 1-certificate is a subtree in which the AND-gates have two children but the OR gates only have one each. Thus the subtree only needs to involve $2^{\lceil k/2 \rceil}$ input bits. If $f(x) = 0$, a similar argument applies, but the role of OR-gates and AND-gates, and values 1 and 0 are reversed. The result is that the certificate complexity of f_k is $2^{\lceil k/2 \rceil}$, or about $\sqrt{n}$.

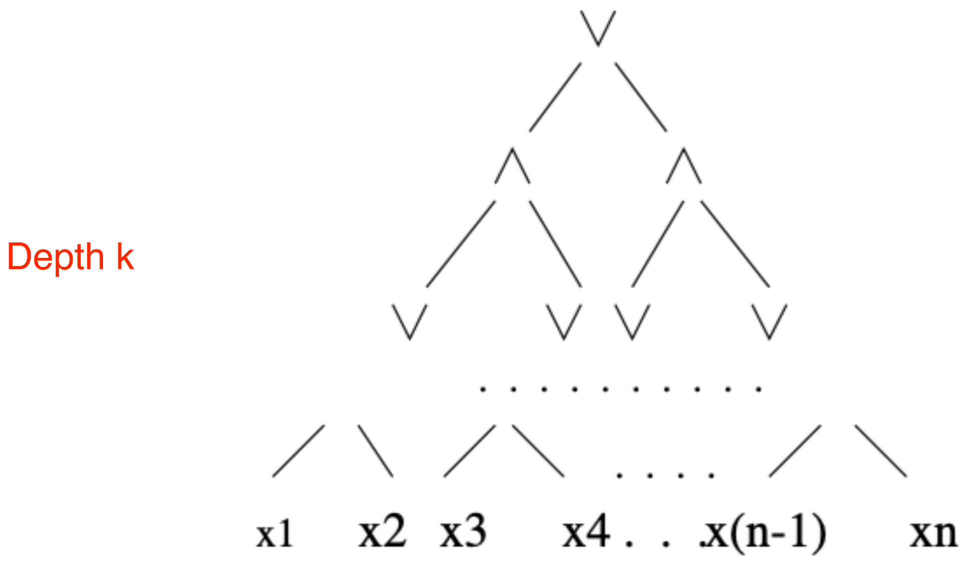


Figure 2: A circuit showing the computation of the AND-OR function. The circuit has k layers of alternating gates, where $n = 2^k$.

The following is a rough way to think about these concepts in analogy to Turing machine complexity as we have studied it.

$$\text{low decision tree complexity} \leftrightarrow \quad \mathbf{P} \tag{4}$$
$$\text{low 1-certificate complexity} \leftrightarrow \quad \mathbf{NP} \tag{5}$$
$$\text{low 0-certificate complexity} \leftrightarrow \quad \mathbf{coNP} \tag{6}$$

The following result shows, however, that the analogy may not be exact since in the decision tree world, $\mathbf{P} = \mathbf{NP} \cap \mathbf{coNP}$.

THEOREM 2
For function f, $D(f) \leq C(f)^2$.

It should be noted that there are some functions, such as the AND-OR function, for which this inequality is tight.

PROOF: Let S_0, S_1 be the set of minimal 0-certificates and 1-certificates, respectively, for f. Let $k = C(f)$, so each certificate has at most k bits.

REMARK 1 Note that every 0-certificate must share a bit position with every 1-certificate, and furthermore, assign this bit differently. If this were not the case, then it would be possible for both a 0-certificate and 1-certificate to be asserted at the same time, which is impossible.

The following decision tree algorithm then determines the value of f in at most k^2 queries.

Algorithm: Repeat until the value of f is determined: Choose a remaining 0-certificate from S_0 and query all the bits in it. If the bits are the values that prove the f to be 0, then stop. Otherwise, we can prune the set of remaining certificates as follows. Since all 1-certificates must intersect the chosen 0-certificate, for any $c_1 \in S_1$, one bit in c_1 must have been queried here. Eliminate c_1 from consideration if the certifying value of c_1 at the location is different from the actual value found. Otherwise, we only need to consider the remaining $k - 1$ bits of c_1.

This algorithm can repeat at most k times. For each iteration, the unfixed lengths of the uneliminated 1-certificates decreases by one. This is because once some values of the input have been fixed due to queries, for any 0-certificate, it remains true that all 1-certificates must intersect it in at least one location that has not been fixed, otherwise it would be possible for both a 0-certificate and a 1-certificate to be asserted. With at most k queries for at most k iterations, a total of k^2 queries is used. $\square$

3 Randomized Decision Trees

There are two equivalent ways to look at randomized decision trees. We can consider decision trees in which the branch taken at each node is determined by the query value and by a random coin flip. We can also consider probability distributions over deterministic decision trees. The analysis that follows uses the latter model.

We will call $\mathcal{P}$ a probability distribution over a set of decision trees $\mathcal{T}$ that compute a particular function. $\mathcal{P}(t)$ is then the probability that tree t is chosen from the distribution.

For a particular input x, then, we define $c(\mathcal{P}, x) = \sum_{tin\mathcal{T}} \mathcal{P}(t) cost(t, x)$. $c(\mathcal{P}, x)$ is thus the expected number of queries a tree chosen from $\mathcal{T}$ will make on input x. We can then characterize how well randomized decision trees can operate on a particular problem.

Definition 5 *The* **randomized decision tree complexity**, *$\mathcal{R}(f)$, of f, is defined as follows.*

$$\mathcal{R}(f) = \min_{\mathcal{P}} \max_{x} c(\mathcal{P}, x) \tag{7}$$

The randomized decision tree complexity thus expresses how well the best possible probability distribution of trees will do against the worst possible input for a particular probability distribution of trees. We can observe immediately that $\mathcal{R}(f) \geq C(f)$. This is because $C(f)$ is a minimum value of $cost(t, x)$. Since $\mathcal{R}(f)$ is just an expected value for a particular probability distribution of these *cost* values, the minimum such value can be no greater than the expected value.

Example 6 Consider the majority function, $f = Maj(x_1, x_2, x_3)$. It is straightforward to see that $D(f) = 3$. We show that $\mathcal{R}(f) \leq 8/3$. Let $\mathcal{P}$ be a uniform distribution over the (six) ways of ordering the queries of the three input bits. Now if all three bits are the same, then regardless of the order chosen, the decision tree will produce the correct answer after two queries. For such x, $c(\mathcal{P}, x) = 2$. If two of the bits are the same and the third is different, then there is a 1/3 probability that the chosen decision tree will choose the two similar bits to query first, and thus a 1/3 probability that the cost will be 2. There thus remains a 2/3 probability that all three bits will need to be inspected. For such x, then, $c(\mathcal{P}, x) = 8/3$. Therefore, $\mathcal{R}(f)$ is at most 8/3.

4 Distributional Complexity

We now consider a related topic, distributional complexity. Where randomized complexity explores distributions over the space of decision trees for a problem, distributional complexity considers probability distributions on inputs. It is under such considerations that we can speak of "average case analysis." These two concepts turn out to be related in a useful way by Yao's Lemma. Let $\mathcal{D}$ be a probability distribution over the space of input strings of length n. Then, if A is a deterministic algorithm, such as a decision tree, for a function, then we define the distributional complexity of A on a function f with inputs distributed according to $\mathcal{D}$ as the expected cost for algorithm A to compute f, where the expectation is over the distribution of inputs.

Definition 6 *The* **distributional complexity** $d(A, \mathcal{D})$ *of algorithm A given inputs distributed according to* $\mathcal{D}$ *is defined as:*

$$d(A, \mathcal{D}) = \sum_{x:input} \mathcal{D}(x) cost(A, x) = \mathbf{E}_{x \in \mathcal{D}}[cost(A, x)] \tag{8}$$

Definition 7 *The* **distributional** *decision tree complexity,* $\Delta(f)$ *of function f is defined as:*

$$\Delta(f) = \max_{\mathcal{D}} \min_{A} d(A, \mathcal{D}) \tag{9}$$

Where A above runs over the set of decision trees that are deciders for f.

So the distributional decision tree complexity measures the expected efficiency of the most efficient decision tree algorithm works given the worst case distribution of inputs.

Yao's lemma relates distributional decision tree complexity to the earlier introduced notion of randomized decision tree complexity.

THEOREM 3
For all computational models in which both the set of inputs and the set of algorithms is finite, for any funcion f,

$$\mathcal{R}(f) = \Delta(f) \tag{10}$$

Yao's Lemma holds for decision trees, as they take inputs of fixed length, and as there is a finite number of labelled trees for inputs of any length. The Lemma is a version of von Neumann's minmax theorem from game theory. Here we consider one player to choose from a number of decision trees, and another player to choose from among a set of inputs, and we let the payoff equal the cost of running a tree on an input. The Lemma states that the cost is the same whether we let one player or the other choose first - whether we consider distributions over inputs or distributions over trees.

So in order to find a lower bound on some randomized algorithm, it suffices to find a lower bound on $\Delta(f)$. Such a lower bound can be found by postulating an input distribution $\mathcal{D}$ and seeing whether every algorithm has expected cost at least equal to the desired lower bound.

EXAMPLE 7 We return to considering the majority function, and we seek to find a lower bound on $\Delta(f)$. Consider a distribution over inputs such that inputs in which all three bits match, namely 000 and 111, occur with probability 0. All other inputs occur with probability 1/6. For any decision tree, that is, for any order in which the three bits are examined, there is exactly a 1/3 probability that the first two bits examined will be the same value, and thus there is a 1/3 probability that the cost is 2. There is then a 2/3 probability that the cost is 3. Thus the overall expected cost for this distribution is 8/3. This implies that $\Delta(f) \geq 8/3$ and in turn that $\mathcal{R}(f) \geq 8/3$. So $\Delta(f) = \mathcal{R}(f) = 8/3$.