5 Boolean Decision Trees (February 11)

5.1 Graph Connectivity

Suppose we are given an undirected graph G, represented as a boolean adjacency matrix $A = (a_{ij})$, where $a_{ij} = 1$ if and only if vertices i and j are connected by an edge. How hard is it to decide whether G is connected? Specifically, how many entries in the adjacency matrix do we have to examine? As usual, we want the worst-case running time of the best possible algorithm, as a function of n, the number of vertices:

$$D(n) = \min_A \max_G T(A, G)$$

Here I'll use $D(n)$ instead of $T(n)$ to emphasize that we are looking at *deterministic* algorithms. Clearly $D(n) \le \binom{n}{2}$, since the adjacency matrix is symmetric and has zeros on the diagonal.

One way to derive a lower bound for any decision problem is to consider the size of the smallest *proof* or *certificate* that verifies the result. To prove that a graph is connected, it is clearly both necessary and sufficient to reveal the edges in an arbitrary spanning tree of G. This tree constitutes a *certificate* that G is connected, or a 1-*certificate* for short. Conversely, if we want to prove that a graph is *disconnected*, we need to demonstrate a cut with no crossing edges, that is, a partition of the vertices into two disjoint subsets, such that no edge has one endpoint in each subset. Such a cut is called a 0-*certificate*. Clearly, any algorithm that tests connectedness must check all the edges in some 1-certificate before returning TRUE and all the edges in a 0-certificate before returning FALSE.

Let $C_0(n)$ and $C_1(n)$ respectively denote the maximum size of a 0- or 1-certificate in an n-vertex graph, and let $C(n) = \max\{C_0(n), C_1(n)\}$. We immediately have the following general result.

Theorem 1. $D(n) \ge C(n)$.

In the case of graph connectivity, we have

$$C_0(n) = \lceil n/2 \rceil \cdot \lfloor n/2 \rfloor = (n^2 - (n \bmod 2))/4 \qquad \text{and} \qquad C_1(n) = n - 1,$$

so $D(n) \ge C(n) = (n^2 - (n \bmod 2))/4 = \Omega(n^2)$. In other words, the trivial algorithm "check every edge" is optimal up to a small constant factor. Surprisingly, this algorithm is actually *exactly* optimal!

Theorem 2. $D(n) = \binom{n}{2}$

Proof: I'll describe an adversary strategy that requires any algorithm to examine every edge. The adversary maintains two graphs, Y and M, each with n vertices; initially, Y is empty and M is a clique. Y ('yes') contains the edges that the algorithm has examined and found to be present in the fictional input graph. M ('maybe') contains any edge that *might* be in the fictional input graph; in other words, an edge (i, j) is *absent* from M if the algorithm knows that $a_{ij} = 0$. Note that Y is a subgraph of M.

The adversary uses the following simple strategy when the algorithm examines an potential edge: return FALSE unless that answer would force the fictional input graph to be disconnected.

```
EXAMINE(i, j):
  if (i, j) ∈ Y
        derisively return TRUE
  else if (i, j) ∉ M
        mockingly return FALSE
  else if M \ (i, j) is disconnected
        add (i, j) to Y
        grudgingly return TRUE
  else
        remove (i, j) from M
        sigh and return FALSE
```

We easily observe that with this strategy M is always connected and Y is always acyclic. Moreover, whenever the adversary adds an edge between two components of Y, every other edge joining those two components of Y has already been queried and removed from M. It follows that Y becomes connected only after $\binom{n}{2}$ queries; at that moment, both Y and M consist of the same spanning tree, and the algorithm can safely return TRUE. Before the $\binom{n}{2}$th query, M is connected and Y is disconnected. Since both graphs are consistent with the adversary's answers, either could serve as the fictional input graph, which means the algorithm cannot possibly determine the correct output. □

A graph property like connectivity that requires looking at every possible edge to detect is called *evasive*. We will return to evasive graph properties in a future lecture.

5.2 String Properties (Boolean functions, languages, whatever...)

Let's look at these ideas in a little more generality. A *string property*[1] is any function of the form $F : \{0,1\}^n \to \{0,1\}$. Let $T(A, s)$ denote the number of bits in an n-bit input string s that an algorithm A examines before correctly returning $F(s)$. The *deterministic decision tree complexity* of a string property F is, as usual, the worst-case running time of any algorithm to compute it, as a function of the input size n:

$$\boxed{D(n) = \min_{A} \max_{|s|=n} T(A, s)}$$

The model of computation is the same boolean decision tree that we saw in the very first lecture. For each input size n, we can model any algorithm by a rooted binary tree, where each internal node stores the index of the next bit to examine, and each leaf stores an output value. In this model, $T(A, s)$ is the depth of the path traversed by the input string s in tree A, and the deterministic complexity of any string property is the minimum depth of any tree that correctly computes it. A string property is *evasive* if $D(n) = n$.

We can define the certificate complexity of a string property F as follows. Intuitively, a 1-certificate is a subset of the bits in the input s that forces $F(s) = 1$, and a 0-certificate is a subset of bits that forces $F(s) = 0$. The *certificate complexity* $C(s)$ of a string s is the size of the smallest certificate consistent with s. Finally, the *certificate complexity* $C(n)$ of F is the maximum certificate complexity of any n-bit input string. Equivalently, we have

$$\boxed{C(n) = \max_{|s|=n} \min_{A} T(A, s)}$$

[1]Admittedly, this is a rather bizarre name, but it is analogous to *graph properties* such as connectedness (which we just saw), acyclicity, planarity, and the like. A string property is just a boolean function with n arguments, or equivalently, a set of n-bit strings, or equivalently, a language.

where (as above) the min is taken over all algorithms A that correctly compute F *for all inputs*.

Originally, certificate complexity was known as *non-deterministic decision tree complexity*, since it corresponds to a nondeterministic variant of the boolean decision tree model. The best way to think of a nondeterministic decision tree is as a family of deterministic decision trees, where for each input, we use the *best* decision tree in the set for that input.[2]

We've seen one example of these definitions already. For the function $F : \{0,1\}^{\binom{n}{2}} \to \{0,1\}$ that indicates the connectivity of an n-vertex graph, we have $D(\binom{n}{2}) = \binom{n}{2}$ and $C(\binom{n}{2}) = \frac{n^2 - n \bmod 2}{2}$. As another example, let $F : \{0,1\}^{2n} \to \{0,1\}$ denote the "exactly half" function: $F(s) = 1$ if and only if s contains exactly n 1s and exactly n 0s. For this function, we easily observe that $D(2n) = C(2n) = 2n$.

5.3 Blum's Theorem

One of the most general results about decision tree complexity was proved by Manuel Blum.

Theorem 3 (Blum). *For any string property,* $C(n) \le D(n) \le C(n)^2$.

Proof: The first inequality $C(n) \le D(n)$ is almost trivial, since any algorithm must examine all the bits in a certificate before returning its output. Alternately, the inequality

$$\max_x \min_y f(x,y) \le \min_y \max_x f(x,y)$$

is easy to prove for *any* function $f(x,y)$.

To prove the other inequality, we describe an algorithm that examines $C(n)^2$ bits. Let π_s denote the smallest certificate for any input string s. Let $S_0 = \{\pi_s \mid F(s) = 0\}$ and $S_1 = \{\pi_s \mid F(s) = 1\}$ be the sets of all minimal 0- and 1-certificates, respectively. To simplify notation, let $f = C(n)$.

Our algorithm works in k phases, examining at most k bits in each phase. At each phase, we keep only the certificates that are consistent with the bits we've examined so far. Let S_0^i denote the subset of S_0 consistent with the bits seen in the first i phases, and define S_1^i similarly. In particular, we have $S_0^0 = S_0$ and $S_1^0 = S_1$. However, since it is pointless to carry around any bit whose value we already know, the algorithm only maintains the bits in each certificate that have not yet been queried. Thus, after each input bit x_j is queried, any certificate π that is defined in that bit position is either removed (if $\pi_j \ne x_j$) or its length is decreased by one (if $\pi_j = x_j$).

In the ith phase, the algorithm simply chooses an arbitrary 0-certificate $\pi \in S_0^{i-1}$ and queries all its (previously unqueried) bits. If all the queries agree with π, the algorithm correctly returns FALSE; otherwise, it continues with the next phase.

Now I claim that each phase reduces the length of every surviving 1-certificate by at least 1. Let σ be an arbitrary 1-certificate in S_1^{i-1}. Since every input string contains either a 0-certificate or a 1-certificate, but not both, there must be at least one bit position that appears in both σ and the chosen 0-certificate π. Moreover, this bit position was not queried in any earlier phase, because otherwise, at most one of them would have survived. If the input string agrees with π at that common bit position, σ is discarded; otherwise, the length of σ decreases, as claimed.

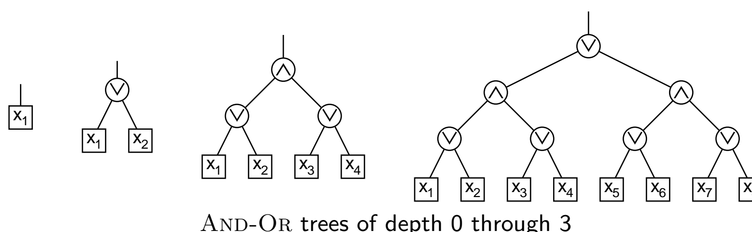
It follows that after at most k phases, we are left with either no 1-certificates, in which case the output must be FALSE, or a single empty 1-certificate (*i.e.*, a 1-certificate that matches the input in every bit position), in which case the output must be TRUE. Since each phase examines at most k bits, the algorithm examines at most k^2 bits altogether. □

[2] "Nondeterminism means never having to admit you're wrong."

We have already seen examples where the first inequality is tight. The second inequality can be tight as well. Consider the function

$$F_k(x_1, x_2, \ldots, x_{2^k}) = \begin{cases} x_1 & \text{if } k = 0 \\ F_{k-1}(x_1, x_2, \ldots, x_{2^{k-1}}) \wedge F_{k-1}(x_{2^{k-1}+1}, x_{2^{k-1}+2}, \ldots, x_{2^k}) & \text{if } k \text{ is even} \\ F_{k-1}(x_1, x_2, \ldots, x_{2^{k-1}}) \vee F_{k-1}(x_{2^{k-1}+1}, x_{2^{k-1}+2}, \ldots, x_{2^k}) & \text{if } k \text{ is odd} \end{cases}$$

This function F_k models an "AND-OR" tree of depth k: a boolean circuit in the form of a complete binary tree, where the leaves are inputs, gates alternate between AND and OR at each level, and the value at the root is the output. (Don't confuse the AND-OR tree with the decision tree that evaluates it!)



AND-OR trees of depth 0 through 3

It's easy to show that this function is evasive, but that it has much smaller certificate complexity:

$$\boxed{D(F_k) = 2^k} \qquad \boxed{C(F_k) = 2^{\lceil k/2 \rceil}}$$

In fact, it's so easy that I'll leave it as a homework exercise.

5.4 Randomized complexity

Recall that a *randomized* decision tree is just a probability distribution over a set of deterministic decision trees, and that the *randomized* complexity of a string property is the worst-case expected running time of the fastest randomized decision tree:

$$R(n) = \min_P \max_s \sum_A \Pr_P[A] \cdot T(A, s)$$

Again, I'm using $R(n)$ instead of the earlier notation $\overline{T}(n)$ to emphasize the randomization.

We've already seen examples where randomness helps a little bit, but a more extreme example might be more helpful. Let $M : \{0,1\}^3 \to \{0,1\}$ denote the boolean median (or majority) function

$$M(x, y, z) = \left\lfloor \frac{x + y + z}{2} \right\rfloor.$$

Then we can define the iterated median function $M_k : \{0,1\}^{3^k} \to \{0,1\}$ as $M_0(x_1) = x_1$ and

$$M_k(x_1, \ldots, x_{3^k}) = M(M_{k-1}(x_1, \ldots, x_{3^{k-1}}), M_{k-1}(x_{3^{k-1}+1}, \ldots, x_{2\cdot 3^{k-1}}), M_{k-1}(x_{2\cdot 3^{k-1}+1}, \ldots, x_{3^k}))$$

for all $k \geq 1$.[3]

[3] Douglas Hofstadter defined a three-player game called *Hruska* based on iterated medians. In a level-0 Hruska game, each player chooses an integer between 0 and 5; the player with the median choice wins the game, and adds his chosen number to his level-1 score. For any $i \geq 1$, a level-i game consists of six level-$(i-1)$ games, starting with level-i scores of zero; the winner is the player with the median level-i score at the end of the game, which is then added to that player's score at level $i+1$. In order to make the game fair (and well-defined), each round uses a different permutation of the players to break ties. I once won a level-3 Hruska game by having the median level-3 score, by having the lowest level-2 score, by having the median level-1 score, by choosing the number 5 on my 216th level-0 turn. I don't recommend playing a level-4 game, unless you *want* your brain to explode.

It is not hard to prove that $D(M_k) = 3^k$ and $C(M_k) = 2^k$; this is a good warm-up for the previous homework exercise. The randomized complexity fits between these two bounds. Consider the following randomized algorithm for computing $M(x, y, z)$:

> Query two of x, y, z at random. If they are equal, return their common value. Otherwise, query the remaining variable and return its value.

The probability of two variables that agree is at least 1/3, so the expected running time of this algorithm is at most 8/3. We can use this idea recursively to evaluate M_k quickly as well. Randomly choose two of the three instances of M_{k-1} and evaluate them recursively. If they return the same value, we're done; otherwise (with probability at most 2/3) we have to evaluate the third instance. We have the recurrence

$$R(M_k) \leq \frac{8}{3} R(M_{k-1}),$$

which has the obvious solution $R(M_k) \leq (8/3)^k$.

As it turns out, this algorithm is optimal—so $R(M_k) = (8/3)^k$—but we don't have the tools to prove this yet. Soon. Promise.

Notice that in this case, the randomized complexity falls strictly between the certificate complexity and the deterministic complexity. This should not be a surprise; even randomized algorithms have to run long enough to certify their answers. In general, we have the following trivial extension of Blum's theorem for any boolean function.

$$\boxed{C(n) \leq R(n) \leq T(n) \leq C(n)^2}$$

For the AND-OR tree function F_k, the randomized complexity is $(\frac{1+\sqrt{33}}{4})^k \approx 1.68614^k \approx n^{0.753}$. The upper bound follows from a careful randomized algorithm, which I will leave as the third part of the homework exercise. The matching lower bound is due to Saks and Widgerson[4]. In fact, Saks and Widgerson conjecture that this is a lower bound on the randomized complexity of *any* evasive boolean function. As far as I know, this conjecture is still open.

[4] *Proc. STOC* 1986