17 Lecture 17, December 1, 2014

17.1 Branching Processes

Now let's see the analysis of Moser-Tardos from a more general viewpoint. Let μ be a probability distribution on the nonnegative integers. The *branching process* or *Galton-Watson process* with distribution μ is the following tree-valued random variable T:

T has a root. Each vertex v of T gets some N_v children, for N_v independently distributed according to μ. Let $\bar{\mu} = E(N_v)$ (possibly infinite).

Theorem 75 *TFAE:*

1. *T is a.s. finite.*
2. *$\bar{\mu} \leq 1$ and $\mu_1 < 1$.*

(See, for example, Grimmett and Stirzaker [33] §5.4 Thm (5).)

The intuition is this. In the subcritical regime, i.e., $\bar{\mu} < 1$, each parent has less than 1 child on average—so no wonder the generations die out with probability 1. In the supercritical regime, $\bar{\mu} > 1$, things are not so definite—it could happen that the root has no children at all, or even at any depth, that all vertices have no children—but, the number of vertices at a level is generally drifting upwards, and as it grows, the likelihood of population collapse decreases drastically, so overall, the probability of the tree being finite is less than 1. The critical case $\bar{\mu} = 1$ is (as always in these kinds of problems) hardest to determine and here we have two cases. One is that $\mu_1 = 1$ in which case the process is deterministic, T is infinite and there is nothing more to say. The other is that $\mu_1 < 1$ which implies that $\mu_0 > 0$. Now the number of children at each level of the tree is just drifting without bias up or down. However, there is an absorbing boundary at 0: extinction is forever. This process is not a random walk with bounded step size, such as we have studied, but intuitively it behaves similarly, and it goes extinct with probability 1 for basically the same reason.

It is not too hard to make the above arguments formal but here we prove formally only part of the theorem:

If $\bar{\mu} < 1$ then a.s. T is finite.

Proof: We will use the shorthand $\mu_{\geq i} = \sum_{j \geq i} \mu_j$.

When a vertex has N children, we list them in an arbitrary "birth order" as children $1, \ldots, N$. The "address" of a vertex of the tree is a finite string of positive integers $(X_1 \ldots X_\ell)$: the root is represented by the empty string and the address of a vertex is its parent's address followed by its birth order.

For a string $(X_1 \ldots X_\ell)$, let $[\![X_1 \ldots X_\ell]\!]$ denote the (indicator rv of) the event that this address exists in the tree. An equivalent characterization of this event is that

1. The root has at least X_1 children, *and*
2. The vertex (X_1) has at least X_2 children, *and*
3. The vertex $(X_1 X_2)$ has at least X_3 children, *and* etc.

Note that $\bar{\mu} = \sum_{i \geq 0} i\mu_i = \sum_{i \geq 1} \mu_{\geq i}$. Recall that by assumption $\bar{\mu} < 1$.

Now, essentially by definition, we have the following:

$$\Pr([\![X_1 \ldots X_\ell]\!]) = \prod_{j=1}^{\ell} \mu_{\geq X_j}$$

The event that T is infinite is equivalent to the event that infinitely many events $[\![\vec{X}]\!]$ occur, which is to say, $\sum_{\vec{X}} [\![\vec{X}]\!] = \infty$. Lemma (7), the first Borel-Cantelli lemma, tells us that T is almost surely finite, because, by distributivity (applied separately for each value of ℓ; no infinite products are involved):

$$\begin{aligned}
\sum_{\vec{X}} \Pr([\![\vec{X}]\!]) &= \sum_{\ell \geq 0} \sum_{\vec{X}:|\vec{X}|=\ell} \prod_{j=1}^{\ell} \mu_{\geq X_j} \\
&= \sum_{\ell \geq 0} \prod_{j=1}^{\ell} \sum_{X \geq 1} \mu_{\geq X} \\
&= \sum_{\ell \geq 0} \prod_{j=1}^{\ell} \bar{\mu} \\
&= \sum_{\ell \geq 0} \bar{\mu}^{\ell} \\
&= \frac{1}{1-\bar{\mu}} < \infty
\end{aligned}$$

17.2 Game tree evaluation: deterministic algorithms

A *game tree* is a tree—in general it can be infinite, but here we assume it finite—in which levels of the tree alternately belong to the "0 player" and the "1 player". The leaves of the tree are labeled with either 0's or 1's. The game starts with a token at the root, and the players take turns, each playing when the token is at a level they own, moving the token down to one of its current children. The goal of each player is to reach a leaf with its own label.

The game always has a winner: this we observe by induction on the depth of the tree (maximum depth of any leaf). Label a vertex with the identity of the player who wins the game if we root the tree at that vertex. Depth 0 is trivial, already correctly labeled. Depth $t \geq 1$: if the owner is b, label the vertex b if any of the children has label b, otherwise label the vertex $1 - b$.

Let N be the number of leaves. We are interested in the worst-case complexity of evaluating the tree. (We assume we know the structure of the tree but not the leaf labeling.)

Clearly we can evaluate the tree deterministically with N queries; what is interesting is that this is best possible. We prove this by induction on the depth of the tree; we show that the adversary can reply to queries in such a way that so long as we have not queried all leaves, he retains control over the labeling of the root. For depth 0, i.e., $N = 1$, the root label is precisely the adversary's label.

Before carrying out the formal induction it's worth seeing how it works just for depth 1: if the root is owned by player b, the adversary keeps replying with "$1 - b$" to every query, thereby leaving the value of the root in question, until the final leaf is queried.

The same idea works higher up. For root depth $t \geq 1$, suppose the root v is owned by player b and has children $v_1, \ldots, v_c$. Inside each of the subtrees rooted at v_i ($i = 1, \ldots, c$), the adversary uses his recursive strategy up until the very last leaf in the subtree is queried. That is, until that point, his responses don't depend on what queries and answers have happened outside the subtree, and they simply ensure that $\ell(v_i)$ is undetermined. Finally when the last leaf under v_i is queried, if that is not the last leaf of v to be queried, the leaf label is chosen to set $\ell(v_i) = 1 - b$.

In this way, $\ell(v)$ is not determined until the last leaf under v is queried. At that point, if the last leaf is in say the v_i subtree, we can (by induction) use the choice at the leaf to set $\ell(v_i)$ at will. The setting at the other children of v to $1 - b$ ensures that $\ell(v) = \ell(v_i)$.

We have shown:

Theorem 76 *The deterministic query complexity of game tree evaluation equals the number of leaves of the tree.*

This problem has an equivalent interpretation in terms of *evaluating a boolean formula.* Interpret a "0-owned" internal vertex as an AND gate, and a "1-owned" internal vertex as an OR gate.

17.3 Game tree evaluation: randomized algorithms

Let us now focus on game trees with a uniform structure. The most canonical example is alternating levels of binary AND and OR gates.

This can be simplified even further with the identity $\mathrm{OR}(x, y) = \neg\mathrm{AND}(\neg x, \neg y)$. So we can reduce to the problem of evaluating a complete binary tree of NAND gates. (In the conversion, it may be necessary to flip the input gates or the output gate or both; but this does not affect the query complexity of the problem.)

NAND	0	1
0	1	1
1	1	0

The key to savings in the randomized case is that when either input to a NAND gate is found to be 0, it is determined that the output is 1 and therefore we don't need to look at the other input. This suggests a DFS evaluation should be efficient. Of course, this makes sense for deterministic evaluation too, but if we go in a prescribed order, the adversary will always make us see a 1 first. (This is exactly what we exploited in the lower bound in the previous section.) A randomized algorithm has the advantage that it can run the DFS in a random order, so if either child evaluates to 0, we have half probability of looking at that child first and saving the other evaluation.

17.3.1 Interlude: MAJ3 trees of depth n

There's a small technical challenge in analyzing the NAND tree so let's look at an easier question: The complete ternary tree of depth n in which every non-leaf vertex is a Majority gate. (A singleton node is depth 0.) Let S_n be the expected number of leaves evaluated by the randomized DFS strategy.

Whatever the values of the three children of a node, we only have to evaluate the last child if the first two disagreed. At least one of the three pairs is always in agreement, so the probability we first find a disagreeing pair is at most $2/3$. So

$$S_n \leq (2/3) \cdot 3 \cdot S_{n-1} + (1/3) \cdot 2 \cdot S_{n-1} = (8/3)S_{n-1}$$

Hence (keeping in mind $S_0 = 1$), $S_n \leq (8/3)^n = N^{\log_3 8/3} \cong N^{0.893}$ where $N = 3^n$ is the number of leaves.

Notice that what we have here is very much like a branching process. Every node has either two or three children (until level n at which the branching process is cut off), and the savings in the randomized algorithm comes from showing that, to use our earlier notation, $\bar{\mu} \leq 8/3$.

17.3.2 Back to the complete NAND tree

Let's see how well random DFS performs here. Let $T_n =$ worst-case running time for evaluating a complete binary tree of NANDs of depth n.

What is a little trickier here than for MAJ3 is that for the 0 output of a NAND gate, there is no "short certificate": you really need to look at both inputs to certify this output value. This is unlike for MAJ3 where no matter the output, there is always some pair of inputs which are enough to certify the output value. On the other hand for the 1 output of a NAND gate, there is a short certificate of length just 1, and from this we will get some savings.

What we need to show is that the "no short certificate" problem diminishes with depth because the adversary cannot set up a distribution on inputs which force us always toward this situation.

Formally we do this by keeping track not of one but of two expectations. For $b = 0, 1$ let T_n^b =worst-case running time on inputs which evaluate to b. Then

$$T_n^0 \leq 2T_{n-1}^1$$

because the only way for a NAND gate to evaluate to 0 is if both inputs evaluate to 1. The upper bound factor of 2 is only the trivial bound, but what is helpful is that the expression is in terms of the T^1 side, where we *do* have a short certificate:

$$T_n^1 \leq \frac{1}{2} \cdot T_{n-1}^0 + \frac{1}{2} \cdot (T_{n-1}^1 + T_{n-1}^0) = T_{n-1}^0 + \frac{1}{2} \cdot T_{n-1}^1$$

because if the gate outputs 1 then at least one of its inputs is a 0, and there's probability at least half that we find a 0 input first.

So we can write the vector recursion

$$\begin{pmatrix} T_n^0 \\ T_n^1 \end{pmatrix} \leq \begin{pmatrix} 0 & 2 \\ 1 & \frac{1}{2} \end{pmatrix} \begin{pmatrix} T_{n-1}^0 \\ T_{n-1}^1 \end{pmatrix}$$

The eigenvalues of this matrix are $\frac{1 \pm \sqrt{33}}{4}$, and the larger in absolute value is $\frac{1+\sqrt{33}}{4}$, so we find that $T_n^0, T_n^1 \in O((\frac{1+\sqrt{33}}{4})^n) \cong 1.686^n$, which is to say $N^{\lg \frac{1+\sqrt{33}}{4}} \cong N^{0.7537}$, where $N = 2^n$ is the number of leaves.

Of course, our next step should be to see whether there is a lower bound to match this performance, for either the MAJ3 or NAND trees. It turns out that our algorithms are optimal but this takes some work. The first step is a fundamental fact about randomized complexity.