CS 388R: Randomized Algorithms Fall 2017

Lecture 3 — Sep 7, 2017

Prof. Eric Price *Scribe: Kevin Song, Matt Denend*

NOTE: THESE NOTES HAVE NOT BEEN EDITED OR CHECKED FOR CORRECTNESS

1 Overview

In this lecture, we study various algorithms for game-tree evaluation and examine their complexities with respect to random-access queries of the leaf nodes.

But first, an appetizer.

2 Random Walk with Sticky Barriers

Question. Suppose we start at zero on a number line and walk randomly, taking steps of ± 1 with equal probability. We will stop taking steps when we reach either -10 or 100. What is the probability that we will end at 100? Figure 1 illustrates this problem.

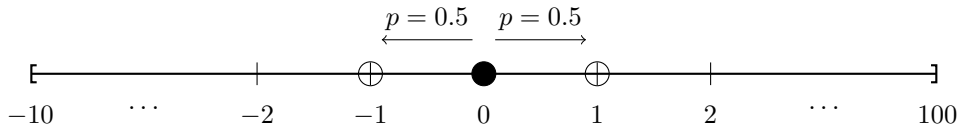


Figure 1: An illustration of the random walk problem. We start at 0 and have a 50% probability of going in either direction, and will stop when we reach -10 or 100.

Soln. Let X_t be the random variable whose value is the position of the random walker at step t. Since we started at 0, $X_0 = 0$.

Furthermore, note that since we walk left and right with equal probability,

$$\mathrm{E}[X_t | X_{t-1}] = X_{t-1}$$

that is to say, the expected value of the position on the number line does not change when we take a step.[1] By conditioning all the way back to $t = 0$, we find that

$$\mathrm{E}[X_t] = \mathrm{E}[X_t | X_{t-1}, X_{t-2} \ldots, X_0] = X_0 = 0 \quad \forall t \in \mathbb{N}$$

But we know that, almost certainly, we must eventually wind up at one of the endpoints (since once we hit an endpoint, we cannot leave it). Therefore, we can treat this as a linear problem. Letting p be the probability of reaching 100, we find that

[1] A process like this in which the expectation does not change is known as a *Martingale Process*.

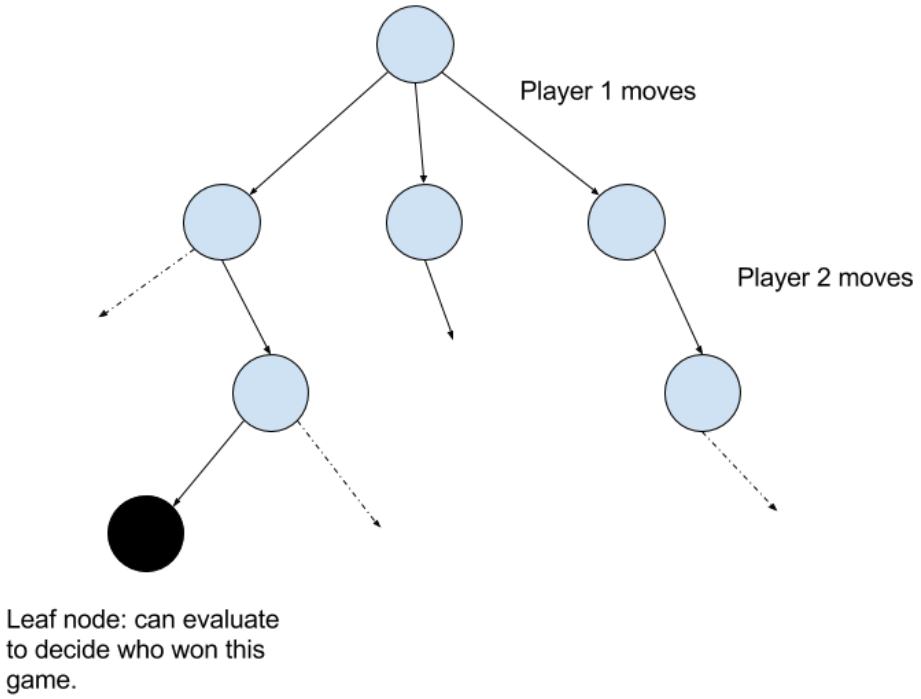


Figure 2: An example of a game tree. The dotted lines represent transitions to states not shown in the diagram. In this example, it is possible to end the game in just 3 moves.

$$\begin{aligned} E[X_\infty] =& 100 \cdot p + -10 \cdot (1-p) \\ =& 100p + 10p - 10 = 0 \\ \implies & 110p = 10 \\ \implies & p = \frac{1}{11} \end{aligned}$$

So we reach 100 with probability 1/11 and -10 with probability 10/11.

3 Game Trees

A *game tree* is a tree of possible game states and decisions that players can make in a game. For example, the root node is the starting state of the game. The edges going from the root node to the first level represent the moves that the first player can make, the edges from the first level to the second are the moves the second player can make, etc.

Leaf nodes in game trees represent an end state of the game. These can be queried to decide who won the game. Of course, if you are playing this game, your goal is to find an internal node where all the leaves end in you winning. This is how some chess AIs work: they explore some portion of a game tree and use heuristics to bound the remainder, then estimate the best possible move.

The question of which moves to take can be formulated as a minimax problem—if a leaf node labeled with 1 if white wins and 0 if black wins, then white's goal is to solve

$$\max_{s_1} \min_{s_2} \max_{s_3} \min_{s_4} \dots V(s_1, s_2, s_3, s_4, \dots)$$

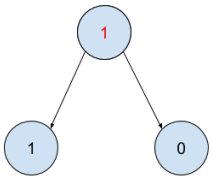


Figure 3: An example of a binary game tree. The black labels are provided. The current player (in the root node) can make a move after which their opponent cannot win (corresponding to taking the right branch). Therefore, the current player *can* win and the label of the root is 1.

where $V(\dots)$ is the game state that results from the given sequence of moves.

3.1 Alternate formulation of game trees

As it turns out, assigning 1 for white winning and 0 for black winning has some difficulties for the problems we try to tackle next. Instead, we can encode a game tree slightly differently: we will say that if the current player can win, the node will have a value of 1, otherwise it will have a value of zero.

An example of how a node can be labeled given the labeling of its children can be seen in Figure 3. Careful consideration of all the possible cases will show that the label of a node should be the NAND of all of its children. This leads naturally to the following question:

Question. Given a binary NAND tree with the values of the leaves, how can we compute the value of the root?

Clearly, this can be done in $O(n)$ time: simply compute the $n/2$ values at the level above the root, the $n/4$ value at the level above that, etc. until the root value is computed.

Can we do better?

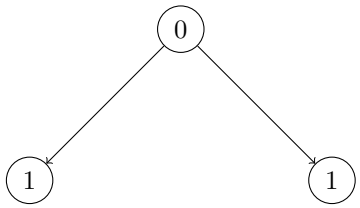
4 Game Trees Claims and Proofs

By observing Figure 4, we can see the difficulty of computing the root node deterministically. When we are computing the root of the tree from the bottom up, if a node a at level k is a 0, then we know that the parent of this node at level $k-1$ is 1. But if the node a at level k is 1, how can we be sure whether the node at level $k-1$ is 0 or 1? In the deterministic case, we may need to look at node a's sibling, call it b, and worst case, all of b's children to determine whether b is 0 or 1.

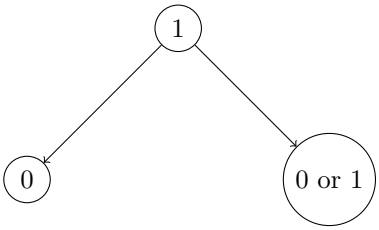
We can do better, but not deterministically. We now present the following claims:

1. Any deterministic algorithm requires $\Omega(n)$ queries.
2. There exists a non-deterministic algorithm using $\Theta(\sqrt{n})$ queries.
3. (a) There exists a randomized algorithm using $O(n^{0.748})$ queries.
   (b) Any randomized algorithm must use $\Omega(n^{0.693})$ queries.

Claim 1 won't be proven in these notes. The remainder of these notes will work towards proving Claims 2 and 3. Claim 3b will be started, but not finished until the next lecture.



(a) When root node is 0, both nodes must be 1.



(b) When root node is 1, at least one of the nodes must be 0.

Figure 4: These pictures explain the different NAND results you can have. The root being 0, for instance, means player A loses, and player B wins, while the root being 1 means that player A can win, but only if player B chooses the wrong move. The trouble of a deterministic algorithm computing from the bottom up arises when you have a node with a value of 1, and have no idea what the sibling is, and can't, for sure, determine the parent value.

4.1 Proving Claim 2

To prove claim 2, observe that we're allowed a non-deterministic algorithm, meaning that we can rely on an advice string. We use this to tell us what the other input of a sibling node is. Let $w(h)$ be the maximum number of leaves to check a height h tree over any tree that the root node of that tree wins (has a 1), while $l(h)$ is the same definition as $w(h)$, except that the root node for that tree loses (has a 0). We can define $l(h)$ and $w(h)$ as this system of inequalities:

$$\begin{aligned} l(h) &= 2 \cdot w(h-1) \\ w(h) &= l(h-1) \end{aligned}$$

Determining $l(h)$ is simple. We know that we need to check both nodes, because the only way for the root node to be a 0 is if both nodes are a 1. $w(h)$ is trickier for a deterministic algorithm: we know that at least one of the input nodes must be a 0, but we don't know which one. However, for a non-deterministic algorithm, we can rely on advice bits to tell us which input is a 0, and check only this.

Taking this system of inequalities, we can use substitution to compute:

$$\begin{aligned} l(h) &= 2 \cdot l(h-2) \\ &= 2^{h/2} \cdot l(0) \\ &= 2^{h/2} \\ &= \Theta(\sqrt{n}) \text{ Queries.} \end{aligned}$$

We can also use similar reasoning to draw the conclusion that $w(h) \leq 2^{h/2} = \sqrt{n}$.

4.2 Proving Claim 3a

For the randomized version, when we need to check the height for a 1 node, there's a 50% chance that we're taking the right path: we want the path that gives us the 0, since we're alternating turns. We use this system of formulas:

$$\begin{aligned} l(h) &= 2 \cdot w(h-1) \\ w(h) &\leq 1/2 \cdot l(h-1) + 1/2(w(h-1) + l(h-1)) \end{aligned}$$

$l(h)$ is the same as in the non-deterministic case. $w(h)$ takes into account two cases, each with 50% chance of happening: the case that we're right, which gives us the formula $l(h-1)$, and the case that we're wrong, in which we check both the winner AND track back to the loser, which gives us the formula $(w(h-1)+l(h-1)$. Solving for $w(h)$ using substitution:

$$\begin{aligned} w(h) &\leq 1/2 \cdot l(h-1) + 1/2(w(h-1) + l(h-1)) \\ &\leq l(h-2) + 1/2 \cdot w(h-1) \\ &\leq 2 \cdot w(h-2) + 1/2 \cdot w(h-1) \end{aligned}$$

We now set $w(h) = x^h$ then solve the characteristic equation for x to solve as follows:

$$\begin{aligned} x^2 &= 2 + 1/2x \\ 0 &= x^2 - 2x - 1/2 \end{aligned}$$

We apply the quadratic formula to get zeroes of $x = \frac{1 \pm \sqrt{33}}{4}$ and drop the negative zero (our base can't be negative) to get $x \approx 1.68$.

We now have $w(h) = 1.68^h$, but need to massage it to the form n^α for some α to determine number of needed queries. We want n to be 2^h then solve for alpha to determine the number of queries. We first convert the base 1.68 into 2:

$$1.68^h = 2^{\log_2 1.68h}$$

We want something of the form $2^{\alpha h}$ since we want to figure out what amount of queries we need to make, and given $2^{\log_2 1.68h}$, $\alpha = \log_2 1.68 \approx 0.748$. Hence, a randomized algorithm makes $O(n^{0.748})$ queries.

4.3 Proving Claim 3b

Proving the lower bound for a randomized algorithm is tricky. Hence, we will instead prove the lower bound of a deterministic algorithm with randomized inputs. The idea is to look at all of the seeds, and try to find something that performs better than the other seeds. Given some deterministic algorithm $A(I, S)$, where I is the input and S is a random seed:

$$\begin{aligned} \forall \text{ inputs } x \in I, \Pr_S[A(x,s) \text{ "good"}] \geq 3/4 \implies \\ \forall \text{ distributions } D_x \text{ over } I, \Pr_{S, x \in D_x}[A(x,s) \text{ "good" }] \geq 3/4 \implies \\ \exists s | \Pr_{S, x \in D_x}[A(x,s) \text{ "good" }] \geq 3/4 \end{aligned}$$

Hence, we can get a deterministic algorithm with one fixed seed to show that a lower bound exists. The non-obvious approach to solving this is the following statement: if $\exists D_x$ for which $\nexists$ any deterministic algorithm that works with 3/4 probability over $x \in D_x$, then $\nexists$ any randomized algorithm that works for all inputs x. A critical key is that we need to test this statement with a distribution of points, not a single point, because it is trivial to determine the output given a single point.

The overarching goal is to make all leaves with probability p, and have the root of two leaves with probability p to hence be $(1-p)^2$, we can accomplish this by making $p = \frac{\sqrt{5}-1}{2}$. With this distribution, you can't get better than $\Omega(n^{0.693})$ queries, which we'll soon prove. But first, an aside about Zero-Sum games, which are important to this proof.

4.4 Zero-sum games

Consider a zero-sum game such as rock-paper-scissors where players bet \$1 each round. This can be represented as a payoff table as follows:

		Rock	Paper	Scissors
Rock		0	-1	1
Paper		1	0	-1
Scissors		-1	1	0

Table 1: A payoff table for RPS. Amounts are player 1's winnings—player one picks the row and player 2 picks the column.

In this table, player 1 picks the row and player 2 picks the column, then the corresponding amount is transferred to player 1.

Now let us say that player 1 will play some probability distribution $X_1, \ldots, X_n$ (where each entry $j \in n$ is the probability that player 1 plays j), while player 2 will play another probability distribution $Y_1, \ldots, Y_n$. The expected payoff for this game for player 1 is then

$$\text{E[Player1]} = \sum_{i,j} M_{ij} X_i Y_j = X^T M Y$$

Player 1's goal is to maximize the payoff, while player 2's goal is to minimize it.

Let's assume that player 1 knows player 2's strategy—then the problem that player 1 would like to solve is

$$\max_X \min_Y X^T M Y$$

If player 2 knows player 1's strategy, then player 2 would like to solve

$$\min_Y \max_X X^T M Y$$

It would not be unreasonable to expect that knowledge of the other player's strategy would always affect your own—that is, if P2 knows P1's strategy, then P1 should change their strategy, which leads to P2 changing their strategy, and so on and so forth. However, in this case, von Neumann's minimax theorem tells us that

$$\max_X \min_Y X^T M Y = \min_Y \max_X X^T M Y$$

The same principle will be used to prove lower bounds on randomized algorithms.