CS 388R: Randomized Algorithms Fall 2015

Lecture 3 — September 2, 2015

Prof. Eric Price *Scribes: Nick Walther, Travis Brannen*

1 Overview

In this lecture we will compare deterministic, non-deterministic, and randomized algorithms for game tree evaluation with respect to time complexity and introduce Yao's minimax principle as a method for finding a lower bound running time for randomized algorithms.

2 Starter Problem

Question: Suppose I take a "simple random walk" starting at 0. At each step, I either go from x to $x+1$ with probability 1/2 or to $x-1$ with probability 1/2. The walk stops when I reach -10 or 100. What is the probability I reach -10?

Answer: $\frac{10}{11}$

Recursive solution:

Let p_n be the probability that if we start at n that we reach -10 before 100.

There are two special values of p where no steps are taken, $p_{-10} = 1$ and $p_{100} = 0$.

For every other point, $p_n = \frac{1}{2}p_{n-1} + \frac{1}{2}p_{n+1}$.

Solving this recursive equation simplifies to $p_n = \frac{100-n}{110}$. For $n = 0$ we get $\frac{10}{11}$.

Another solution:

Let x_i be an iid (independent identically distributed) random variable that represents the choice at step i. Let $Y_t = \sum_{i \le t} x_i$. If $Y_{i-1} \in (-10, 100)$, then $x_i \in \{-1, 1\}$, else $x_i = 0$.

$$\begin{aligned} E[Y_t] &= E\left[\sum x_i\right] \\ &= \sum E[x_i] \\ &= 0 \end{aligned}$$

As t approaches ∞, there are only two possible outcomes, -10 and 100. Thus the expected value at ∞ is equal to sum of these outcomes multiplied by probability of each outcome. Let p again be the probability of reaching -10.

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

2.1 Martingale

Definition: A martingale is a model of a fair game in which knowledge of past outcomes is useless in predicting future events. In particular, a martingale is a sequence of random variables $V_1, V_2, \ldots$ where at any point in the sequence, knowing the values of the previous variables doesn't change the expectation of the current varible.

$$E\,[\,V_3\,] = E\,[\,V_3 \,|\, V_2 \wedge V_1\,]$$

Example: Consider a game at a casino that doesn't take a cut off each bet.
Start with $x = 1$. At each stage of the game you bet x. If you win the game you stop playing, and if you lose you double your bet ($x \to 2x$) and continue playing.

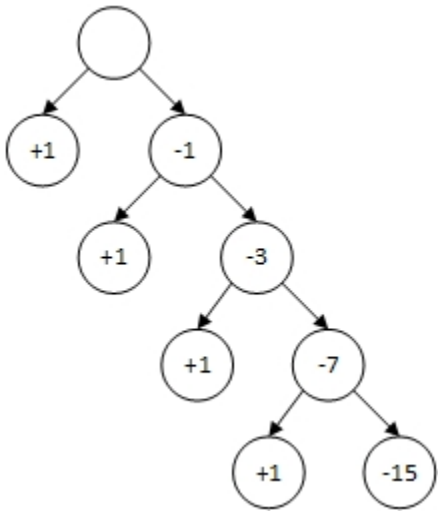

Figure 1: Martingale Casino Game

In this scenario you almost always gain money.
With probability $1 - \frac{1}{2^n}$ you gain 1.
With probability $\frac{1}{2^n}$ you lose $2^n - 1$.

3 Game Tree Evaluation

Consider the following game. Two players take turns deciding which path of a binary tree to travel down. The leaves at the bottom have been assigned a winner. When the bottom of the tree is reached the winner is determined by the leaf.

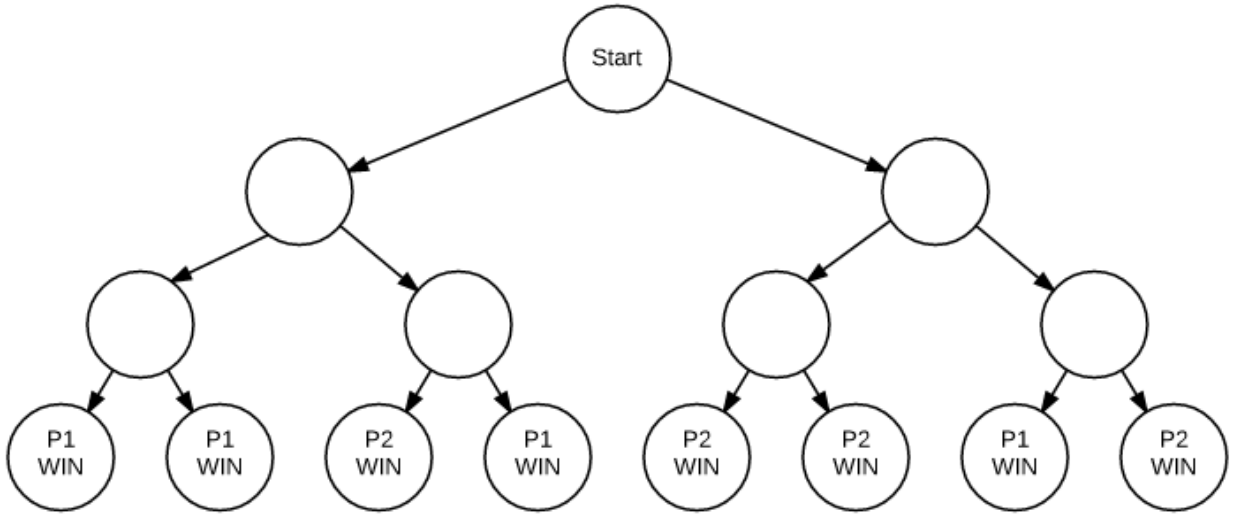

Figure 2: Generic Game Tree

3.1 Deterministic Algorithms

There are some simple deterministic algorithms for determining who will win the game with a particular tree and choice for who goes first.

3.1.1 Algorithm 1

Assume the leaves of the tree that will make P1 win have value 1, and the others value 0. To fill in the second to bottom row of the tree, the operation varies depending on whose turn is next at that row. If it is P1's turn, for each node take the maximum value of each child node. If it is P2's turn, for each node take the minimum value of each child node. So at each level traversing the tree upward, the operation flips from max to min. When the root node is reached, a 0 indicates P2 won and a 1 indicates P1 won. See Figure 3 for an example.

3.1.2 Algorithm 2

This time the meaning of the node value will vary at each level depending on which player takes their turn at that level. For instance, take a look at Figure 4 and assume that P1 goes first. Then at levels 0 and 2, a 1 indicates P1 will win, and a 0 indicates P1 will lose, while at levels 1 and 3, a 1 indicates P2 will win, and a 0 indicates P2 will lose, while at

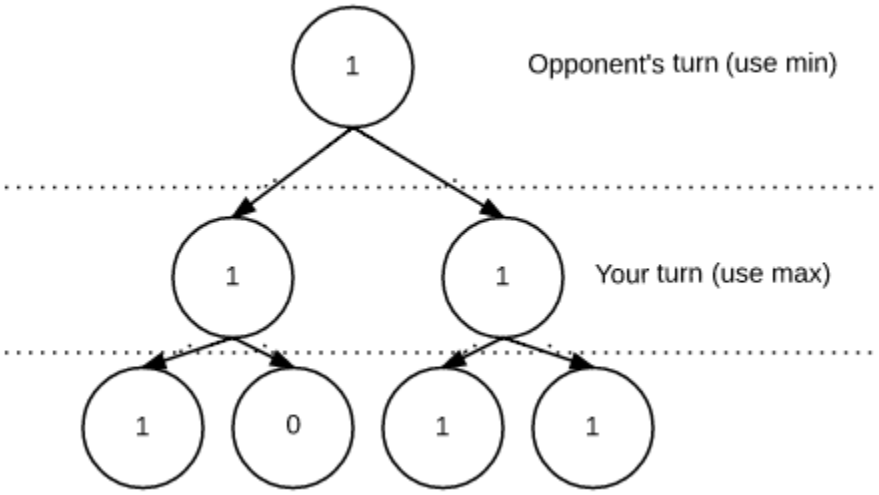

Figure 3: Deterministic Algorithm 1

If given a tree where only the leaf nodes have value, the algorithm to find the winner (root node's value) is simply to take the NAND operation at each level.

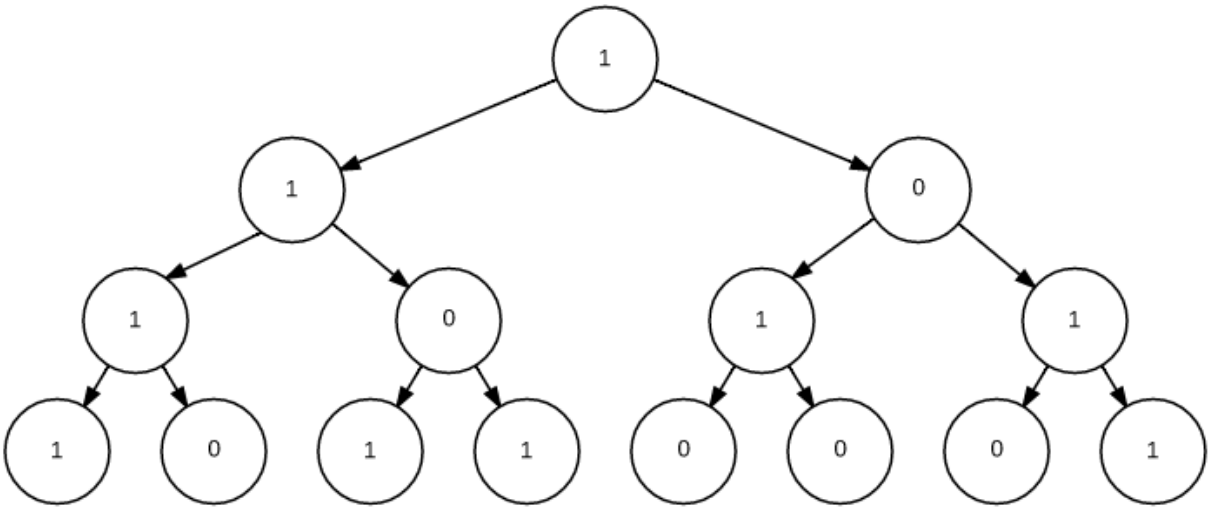

Figure 4: Deterministic Algorithm 2 (NAND)

Because both of these algorithms require that you look at every node in the tree, they run in O(n) time.

3.2 Non-deterministic Algorithms

Let the node values have the same meaning as they did in deterministic algorithm 2. Now assume we have an advice string which tells us only which nodes to look find the answer. For a NAND operation, seeing a single 0 allows you to determine the answer will be a 1. By looking at only the nodes required to reach the top of the tree, we are able to reduce run time. Figure 5 below shows us an example where only 6 of the nodes need to be inspected to learn P1 will win.

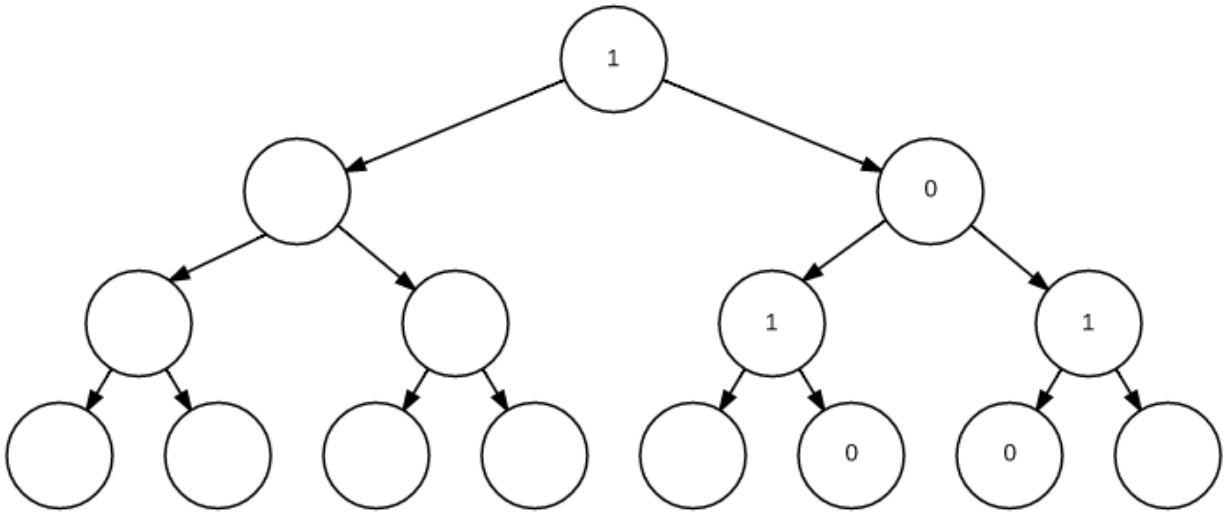

Figure 5: Non-deterministic NAND Algorithm

On an intuitive level, this will be faster than the deterministic way because some sub-trees will not be inspected at all. The question is how many leaves do we need to look at on average?

Let time here imply the number of nodes inspected.

Let $w(h)$ be the time to inspect a tree of height h given root node value is 1.
Let $l(h)$ be the time to inspect a tree of height h given root node value is 0.

To inspect the value at the leaves takes constant time, so...

$$l(h = 0) = w(h = 0) = 1$$

For values higher up the tree, you must expose at least one of the nodes below it. For a win, you only need to know one of the child nodes below it is a loss. Our advice string allows us to only look at a single child node in this case, so...

$$w(h) = l(h - 1)$$

For a loss, you must confirm that both of the nodes below your nodes are wins, so...

$$l(h) = 2w(h - 1)$$

Solving this recurrence we get:

$$w(h) = 2w(h - 2) = 2^{h/2} = \sqrt{n}$$

Therefore the algorithm runs in $O(\sqrt{n})$ time.

3.3 Randomized Algorithms

Again, we will devise a strategy to determine the winner of the game tree using the NAND operation. In our non-deterministic solution, we relied on an advice string that we now don't possess. An intermediate strategy between inspecting every node (which is unnecessary and takes too long) and inspecting only the exact minimum number of nodes required to reach the top of the tree (which relies on an advice string) would be to inspect nodes at random, assigning values to higher nodes when possible and disregarding nodes whose parents have already been evaluated. At each pair of children, if you randomly inspect the 0 first, you don't have to inspect the other child and can simply assign the parent a 1.

Using this strategy, the case for assigning a 0 to a node is still the same. You have to determine that the value for both child nodes is a win:

$$l(h) = 2w(h-1)$$

The case of assigning a 1 to a node is a little more complicated. If you have both loser children, as soon as you look at the first one you can determine the node is a winner.

$$w(h) = l(h-1) \tag{1}$$

If you have one winner and one loser child, you have a 50% chance of picking the loser first and avoiding having to look at the winner. You also have a 50% chance of picking the winner first and needing to look at the loser to determine a value for the node.

$$w(h) = l(h-1) + 1/2w(h-1) \tag{2}$$

Since equation (2) gives us a higher run time than equation (1), we will use it to find an upper bound on run time of our algorithm. Therefore

$$w(h) \leq l(h-1) + 1/2w(h-1)$$

Solving for $w(h)$ using the equations for $w(h)$ and $l(h)$, we obtain

$$w(h) \leq l(h-1) + 1/2w(h-1)$$

Since $w(h) \leq l(h)$,

$$\begin{aligned}
&\leq l(h-1) + 1/2l(h-1) \\
&\leq 3/2l(h-1) \\
&\leq 3w(h-2) \\
&\leq 3^{h/2} \\
&\leq (2^h)^{\frac{\log 3}{2}} \\
&\leq n^{.793}
\end{aligned}$$

As you can see, this is faster than deterministic but not as good as non-deterministic. Now we want to know what is the lower bound run time for a randomized algorithm?

4 Lower Bound Run Time for Randomized Algorithm

4.1 Rock, Paper, Scissors

		Q		
		Rock	Paper	Scissors
P	Rock	0	-1	1
	Paper	1	0	-1
	Scissors	-1	1	0

Table 1: The amount of money Q pays P after each round depending on the result.

Consider the traditional game rock-paper-scissors where a bet of $1 is made in each round. In Table 1, the results of a single round are represented in matrix form. A 1 represents where the row player (P) has won and the column player (Q) has payed them $1, and a -1 represents where the P has lost and has payed Q $1.

As you can see, rock-paper-scissors is a zero-sum game as the gains of one player are balanced by the losses of the other player. The matrix, $\mathbf{M}$, is known as a payoff matrix. Let the vector $\mathbf{p}$ be the probability distribution on the rows of $\mathbf{M}$, or the chance P chooses a certain strategy, and let $\mathbf{q}$ be Q's probability distribution. Now the expected outcome of each round is given by $\mathbf{p}^T\mathbf{M}\mathbf{q}$.

The strategy of each player is straight forward, P wants to maximize the expected payoff and Q wants to minimize the expected payoff. So P chooses $\mathbf{p}$ that maximizes the the payoff matrix and Q chooses $\mathbf{q}$ that minimizes the payoff matrix. An upper bound on the expected value for P, V_P is given when first Q chooses their best strategy and then P chooses their strategy. So

$$V_P = \max_p \min_q \mathbf{p}^T\mathbf{M}\mathbf{q}$$

Likewise, the lower bound on expected value for Q is when P chooses their strategy first. So

$$V_Q = \min_q \max_p \mathbf{p}^T\mathbf{M}\mathbf{q}$$

4.2 von Neumann's Minimax Principle

$$V_P = V_Q$$

von Neumann's Minimax Principle tells us that the largest payoff value P can guarantee is equal to the smallest payoff value Q can guarantee.

4.3 From Rock, Paper, Scissors to Algorithms and Inputs

Now let's try applying the same principle with different meanings for P, Q, $\mathbf{p}$, $\mathbf{q}$, and M. If we let the column player Q choose an algorithm from a set $\mathcal{A}$ of all correct deterministic algorithms, and let the row player P choose the input to the algorithm from a set $\mathcal{I}$ of all inputs, then the values in the cells of the matrix are the running time T of the selected algorithm on the selected input. This is the payoff matrix. As before, Q is trying to minimize the running time by choosing the most efficient algorithm, while the adversary P is trying to maximize the running time by choosing the most adversarial input.

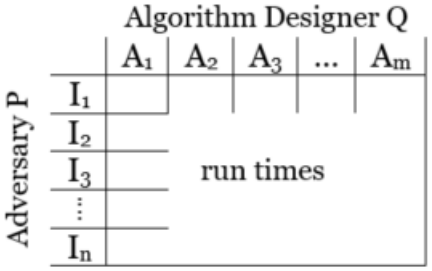

Figure 6: The run time of executing Q's algorithm on P's input.

Note that this payoff matrix may contain a value other than run time, such as computation space used or communication cost.

Let $\mathbf{q}$ be a mixed strategy for Q, or a probability distribution over the space of algorithms. Let $\mathbf{p}$ be a probability distribution over the space of inputs. Then from von Neumann's Principle, both the following statements are derived.

$$\max_{\mathbf{p}} \min_{\mathbf{q}} E[T(I_p, A_q)] = \min_{\mathbf{q}} \max_{p} E[T(I_p, A_q)]$$

$$\max_{\mathbf{p}} \min_{A \in \mathcal{A}} E[T(I_p, A)] = \min_{\mathbf{q}} \max_{I \in \mathcal{I}} E[T(I, A_q)]$$

4.4 Yao's Minimax Principle

For all distributions $\mathbf{p}$ over $\mathcal{I}$ and $\mathbf{q}$ over $\mathcal{A}$,

$$\min_{A \in \mathcal{A}} E[T(I_p, A)] \leq \max_{I \in \mathcal{I}} E[T(I, A_q)].$$

Yao's minimax principle states that the expected run time of the optimal deterministic algorithm for a random input is a lower bound on the expected run time of the optimal randomized algorithm on every input. This provides an important result for us. Now, in order to determine some lower bound on the expected run time for any randomized algorithm, we simply need to calculate the expected run time of the optimal deterministic algorithm on an arbitrary input.

Note: The minimax principle is the only known method for proving lower bounds on run times of Las Vegas style randomized algorithms. The minimax principle does not apply to Monty Carlo style algorithms.

4.5 Applying Minimax Principle to Game Tree Evaluation

Again consider the setup for deterministic algorithm 2. Let each node of the tree have value 1 with probability p and value 0 with probability $1-p$. To solve for p, we know that the probability a node has value 0 is the same as if each of its children have value 1.

$$\begin{aligned} 1-p &= p^2 \\ p &= \frac{\sqrt{5}-1}{2} \end{aligned}$$

Now to calculate expected run time on a tree of height h.

$$\begin{aligned} E[T(h)] &= (1-p)\cdot E[T(h-1)] + p\cdot 2E[T(h-1)] \\ &= (1+p)\cdot E[T(h-1)] \\ &= (1+p)^h \\ &= \left(\frac{\sqrt{5}+1}{2}\right)^h \\ &= (1.618)^h \\ &= n^{0.693} \end{aligned}$$

References

[1] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.