CS 388R: Randomized Algorithms, Fall 2023 8-30-23

Lecture 4: Game tree evaluation and randomized complexity theory

Prof. Eric Price *Scribe: Daniel Kuddes, Kristin Sheridan*

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

1 Overview

In this lecture, we showed how to quickly determine if a player in a two player game has the ability to force a win. In particular, consider a game in which player 1 can make one of two choices, then Player 2 makes one of two choices, and so on until a particular number of rounds has passed. We will show how to decide with high probability whether there is a strategy for Player 1 such that no matter the moves made by Player 2, Player 1 will win.

Additionally, we briefly discussed how problems with randomized algorithms are characterized in complexity theory, including discussing the classes ZPP, RP, coRP, PP, and BPP, and their relative relationships.

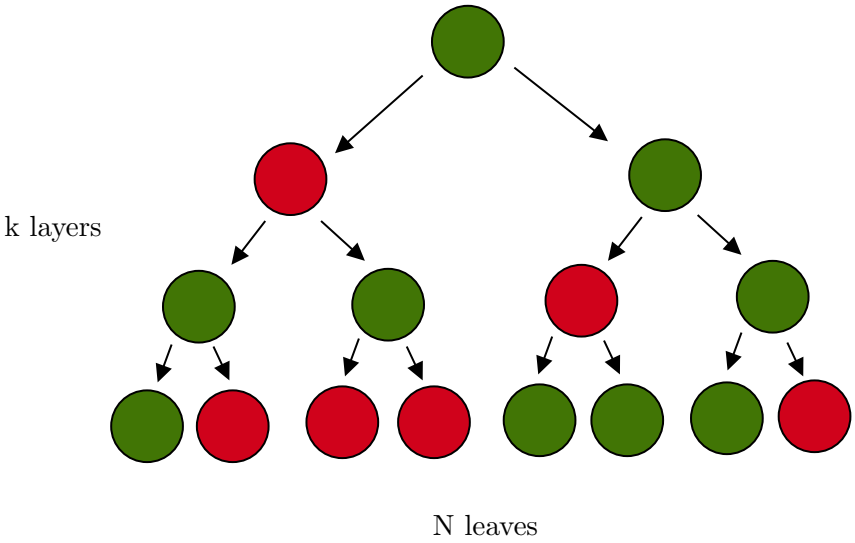
2 Game tree evaluation

Consider a two-player game such that in each round, a player makes one of two choices (alternating which player makes the choice). After k rounds, the game terminates and one of the two players is declared a winner (no ties). We say that player 1 can "force a win" if there is a strategy for player 1 such that no matter the decisions made by player 2, player 1 wins the game. To analyze this possibility, consider the following tree construction:

- The root of the tree is labeled with the starting state of the game
- If node is not labeled with an end state of the game, it gets two children, each labeled with a state that is reachable by the decision of the player whose turn it is. (The player whose turn it is may form part of the game state.)
- If a node is labeled with an end state of the game, it is a leaf and is colored green (1) if Player 1 wins in that end state and red (0) if Player 2 wins in that end state
- If both children of a node have been assigned a color, the color of this node is assigned to be the NAND of the children's colors (where green is 1 and red is 0)

Note that the color of a node labeled with a particular state is green if the player whose turn it is in that state can force a win from that point. This clearly holds at the bottom (leaf) level. If the player in the next turn can force a win, no matter which choice the player at this level makes at this moment (i.e. both children are green), then obviously the player at this level cannot force a win (i.e. it should be red). Otherwise, if there is a child that is colored red, that child must have

two children that are colored green (or else be an end state that is colored red). In the latter case, the player at this level is able to force a win by making the choice to enter the game state where the other player loses. In the earlier case, no matter the choice the player makes at the next round, this player can force a win on the previous round (by induction as we go up the tree).



Consider methods of evaluating the marking (red/green or 0/1) at the root when there are N leaves and $k+1$ layers, depending on the model of computation used.

2.1 Deterministic

The straightforward deterministic algorithm for evaluating the marking at the root of the tree evaluates the root at every node in the tree.

Algorithm 1 Deterministic algorithm for deciding the marking of a given node x

Input: Node x
Output: The marking of x

```
if x is a leaf then
    Return 1 if the player whose turn it is wins the end state and 0 otherwise
if Algorithm 1(x.left)=0 then
    Return 1
else if Algorithm 1(x.right)=0 then
    Return 1
else
    Return 0
```

This algorithm is clearly correct since it is exactly computing the NAND of the two children of x, so we need only analyze the runtime. This algorithm additionally runs in $O(N)$ time by standard implementations, and it can be shown that there exist implementations on which it takes $\Omega(N)$ time. In fact, it can be shown that any deterministic algorithm for game tree evaluation takes $\Omega(N)$ time.

2.2 Non-deterministic

Now consider non-deterministic methods for evaluating the marking at the root of the tree. Recall that non-deterministic algorithms take in an input and a polynomial sized *advice string*. If the correct answer is 1, there exists a choice of advice string that causes acceptance. If the correct answer is 0, there does not exist an advice string that causes acceptance.

Algorithm 2 Non-deterministic (or randomized) algorithm for deciding the marking of a given node x

Input: A node x, advice string (or random string) y of the same length as x
Output: The marking of x with advice string y

```
if x is a leaf then
    Return 1 if the player whose turn it is wins the end state and 0 otherwise
if y[0] = 0 then
    first ← x.left, second ← x.right
else if y[0] = 1 then
    first ← x.right, second ← x.left
if Algorithm 2(first, y[1 :])=0 then
    Return 1
else if Algorithm 2(second, y[1 :])=0 then
    Return 1
else
    Return 0
```

Note that this algorithm is again correct because it exactly computes the NAND of the two children. The advice string does not change our answer, only the order we examine the children in.

To consider the runtime of the algorithm, we define the following values:

$W(i) :=$ time to evaluate the marking of a node i levels above the root, when given a correct advice string and the correct value at this node is green

$L(i) :=$ time to evaluate the marking of a node i levels above the root, when given a correct advice string and the correct value at this node is red

We would like to solve this recursion. Note that if a node is green then at least one child is red. The advice string can tell us which of those nodes to check, and we get $W(i) \leq L(i-1)$. However, if a node is red we must check that *both* children are green, which requires $2W(i-1)$ work. Thus, we get $W(i) \leq L(i-1) \leq 2W(i-2)$. Rolling out this recursion, we get $W(i) \leq 2^{i/2}$. (Note that $2^{i/2} \leq 2 \cdot 2^{(i-2)/2} = 2^{i/2}$.)

Thus, overall work required to evaluate the root is $2^{k/2}$. Since it's a complete binary tree of k levels, $N = 2^k$ and this means overall time is $O(\sqrt{N})$, a factor of $\sqrt{N}$ faster than the deterministic time algorithm took on such inputs.

2.3 Randomized

Non-deterministic algorithms aren't something we can generally evaluate easily, since it requires extra knowledge of the advice string. Thus, we will consider a modification of Algorithm 2 where y is chosen randomly, rather than being an advice string specific to the input x. Note that previous correctness analyses apply here, so we need only analyze the expected runtime.

Now let $W(i)$ be the *expected* time to evaluate that a winning node at level i above the root is winning, where the randomness is over the choice of string y. Likewise, let $L(i)$ be the expected time to evaluate that a winning node at level i above the root is winning. Note that we still have $L(i) = 2W(i-1)$, as a losing evaluation can only be obtained by checking that both child branches are winning.

This leaves us with bounding $W(i)$. If both children are red, then the random string doesn't matter and $W(i)$ costs only $L(i-1)$. If one child is red and one child is green, with probability $1/2$, we check the red branch first and obtain cost $L(i-1)$. With probability $1/2$, we check the green branch first and obtain cost $W(i-1)+L(i-1)$. Thus, total expected cost is $\frac{1}{2}L(i-1)+\frac{1}{2}(L(i-1)+W(i-1)) = L(i-1)+\frac{1}{2}W(i-1)$. Since costs are non-negative, this is no less than $L(i-1)$ and the worst case instance has one green and one red child, so we need only analyze that case.

Thus, overall we have $W(i) \leq L(i-1)+\frac{1}{2}W(i-1) \leq 2W(i-2)+\frac{1}{2}W(i-1)$. We would now like to solve this recursion, and we can do so using matrices. Consider the following equation:

$$\begin{bmatrix} \frac{1}{2} & 2 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} W(i-1) \\ W(i-2) \end{bmatrix} = \begin{bmatrix} W(i) \\ W(i-1) \end{bmatrix}$$

Note that this equation requires $W(i) = 2W(i-2)+\frac{1}{2}W(i-1)$ and $W(i-1) = W(i-1)$, both of which must be true according to our analysis. Unrolling this equation to the base case, we get the following equation.

$$\begin{bmatrix} \frac{1}{2} & 2 \\ 1 & 0 \end{bmatrix}^{k-1} \begin{bmatrix} W(1) \\ W(0) \end{bmatrix} = \begin{bmatrix} W(k) \\ W(k-1) \end{bmatrix}$$

The base cases are $W(0) = 1$ and $L(0) = 1$. $W(1)$ is $\frac{1}{2} \cdot L(0) + \frac{1}{2}(L(0)+W(0)) = 1.5$. Thus, our overall equation is

$$Ay = \begin{bmatrix} \frac{1}{2} & 2 \\ 1 & 0 \end{bmatrix}^{k-1} \begin{bmatrix} \frac{3}{2} \\ 1 \end{bmatrix} = \begin{bmatrix} W(k) \\ W(k-1) \end{bmatrix}.$$

Recall that we can write the first matrix, which we denote A, as $(\lambda_1 v_1 v_1^\intercal + \lambda_2 v_2 v_2^\intercal)$, where λ_i are the eigenvalues of A and v_i are the corresponding orthonormal eigenvectors of A. Since v_1, v_2 span $\mathbb{R}^2$, we can write y as $y_1 v_1 + y_2 v_2$ for some y_1, y_2. Then we have

$$\begin{aligned}(\lambda_1 v_1 v_1^\intercal + \lambda_2 v_2 v_2^\intercal)^k (y_1 v_1 + y_2 v_2) &= (\lambda_1^k v_1 v_1^\intercal + \lambda_2 v_2 v_2^\intercal)(y_1 v_1 + y_2 v_2) \\ &= y_1 \lambda_1^k v_1 + y_2 \lambda_2^k v_2,\end{aligned}$$

where we have used the fact that orthogonality of v_1, v_2 implies $v_1 v_2^\intercal = v_2 v_1^\intercal = 0$. This implies that the overall cost $W(i)$ is a constant times λ_1^k plus a constant times λ_2^k. Thus, we need only find the eigenvalues of A and take the larger one to get an asymptotic bound. Recall that the eigenvalues of A are the values for which $det(A - \lambda I) = 0$. For this choice of A, these are the values for which $-\frac{1}{2}\lambda + \lambda^2 - 2 = 0$, which is when $\lambda = \frac{1 \pm \sqrt{32}}{4}$. The larger is the positive value, so we have overall time $O(\left(\frac{1+\sqrt{32}}{4}\right)^k) = O(\left(\frac{1+\sqrt{32}}{4}\right)^{\log N}) \approx O(2^{0.753k}) = O(N^{0.753})$, which is somewhere between the runtime obtained by the non-deterministic algorithm and the runtime obtained by the deterministic algorithm.

3 Complexity of randomized algorithms

Recall that in complexity theory, we consider *languages*, or subsets of the binary strings. First, we review standard non-randomized complexity theory classes.

- A language L is in the class P (polynomial time) if there exists a deterministic polynomial time algorithm $\mathcal{A}$ such that for any string x, $\mathcal{A}(x) = 1$ if and only if $x \in L$.
- A language L is in the class NP (non-deterministic polynomial time) if there exists a non-deterministic polynomial time algorithm $\mathcal{A}(\cdot, \cdot)$ such that
  - for any string $x \in L$ there exists polynomial-sized string y such that $\mathcal{A}(x, y)$ accepts
  - for any $x \notin L$, $\mathcal{A}(x, y) = 0$ for all y.

  We call y the *advice string* and $\mathcal{A}$ should run in time polynomial in $|x|, |y|$.

We will expand our set of classes to include problems with polynomial time randomized algorithms. Here U is the uniform distribution over all binary strings of the relevant length.

- ZPP: (Zero error polynomial expected time) $L \in$ ZPP if there exists an expected polynomial time algorithm $\mathcal{A}(\cdot, \cdot)$ such that for all binary strings y that are of size $p(|x|)$ for some polynomial p, $\mathcal{A}(x, y)$ is 1 if $x \in L$ and 0 otherwise.
  - If $x \in L$, $\mathbb{P}_{y \sim U}[\mathcal{A}(x, y) = 1] = 1$
  - If $x \notin L$, $\mathbb{P}_{y \sim U}[\mathcal{A}(x, y) = 1] = 0$

  Additionally, $\exp_{y \sim U}[time(\mathcal{A}(x, y))]$ is polynomial in $|x|$, where $time(\mathcal{A}(x, y))$ is the amount of time that $\mathcal{A}(x, y)$ takes to run.
- RP (randomized polynomial time/one-sided error): $L \in$ RP if there exists a polynomial time algorithm $\mathcal{A}(\cdot, \cdot)$ such that
  - If $x \in L$, $\mathbb{P}_{y \sim U}[\mathcal{A}(x, y) = 1] \geq 1/2$
  - If $x \notin L$, $\mathbb{P}_{y \sim U}[\mathcal{A}(x, y) = 1] = 0$

  Note that any polynomial error in the case that $x \in L$ can be increased to error at most $1/2$ using only a polynomial number of repetitions.

class	fraction of advice strings that accept for $x \in L$	fraction of advice strings that accept for $x \notin L$	runtime
ZPP	1	1	expected polynomial
RP	$\geq 1/2$	0	polynomial
coRP	1	$\leq 1/2$	polynomial
BPP	$\geq 2/3$	$\leq 1/3$	polynomial
PP	$> 1/2$	$\leq 1/2$	polynomial
NP	at least 1 string	0	polynomial
coNP	0	at least one string	polynomial

Table 1: Overview of complexity classes discussed in lecture

- coRP (complement of RP): $L \in$ coRP if there exists a polynomial time algorithm $\mathcal{A}(\cdot, \cdot)$ such that
  - If $x \in L$, $\mathbb{P}_{y \sim U}[\mathcal{A}(x, y) = 1] = 0$
  - If $x \notin L$, $\mathbb{P}_{y \sim U}[\mathcal{A}(x, y) = 1] \leq 1/2$

  Note that any polynomial error in the case that $x \in L$ can be increased to error at most $1/2$ using only a polynomial number of repetitions. Additionally, coRP contains exactly the complements of all languages in RP.
- BPP (bounded probabilistic time): $L \in$ BPP if there exists a polynomial time algorithm $\mathcal{A}(\cdot, \cdot)$ such that
  - If $x \in L$, $\mathbb{P}_{y \sim U}[\mathcal{A}(x, y) = 1] \geq 2/3$
  - If $x \notin L$, $\mathbb{P}_{y \sim U}[\mathcal{A}(x, y) = 1] \leq 1/3$

  Note that any polynomial error can be decreased to error at most $1/3$ using a polynomial number of repetitions and taking the majority answer. This class is the most commonly used in complexity theory.
- PP (probabilistic polynomial time): $L \in$ PP if there exists a polynomial time algorithm $\mathcal{A}(\cdot, \cdot)$ such that
  - If $x \in L$, $\mathbb{P}_{y \sim U}[\mathcal{A}(x, y) = 1] > 1/2$
  - If $x \notin L$, $\mathbb{P}_{y \sim U}[\mathcal{A}(x, y) = 1] \leq 1/2$

  This class is actually quite overpowered and thus not frequently used in complexity theory. The Hamiltonian cycle is in fact contained in this class. (Imagine picking a random permutation of vertices and checking if you have a Hamiltonian cycle, and then flipping a coin and outputting the result if it is not. There is a slightly higher than $1/2$ probability of outputting 1 if there is a Hamiltonian cycle (at least $1/2 + 1/n!$) and exactly $1/2$ probability if there is not one.)

To summarize these classes, see the following chart.

3.1 Containment of classes

Note that the we know the following holds:

$$P \subseteq ZPP = RP \cap coRP \subseteq RP \subseteq BPP.$$

Note that we cannot show that the containment is strict for any of the above, and in fact it is often hypothesized (but not known) that $P = BPP$. However, we do not even know if NP contains BPP or vice versa.

Here are a few arguments for the above containments:

- $P \subseteq ZPP$, as we can always ignore our random string and run a polynomial time algorithm if we have one
- $ZPP = RP \cap coRP$.
  - $ZPP \subseteq RP \cap coRP$: Take a ZPP algorithm and terminate after 2 times the expected time. Output the given answer if it is done. If it has not terminated, output 1 to create a coRP algorithm and 0 to create an RP algorithm.
  - $RP \cap coRP \subseteq ZPP$: Run the coRP algorithm and the RP algorithm for the given problem. In a constant number of expected runs, either the RP algorithm will output 1, which implies $x \in L$ or the coRP algorithm will output 0, which implies $x \notin L$.
  - $RP \subseteq BPP$, as any RP algorithm is automatically a BPP algorithm as well if you just boost the probability of correct answers to 2/3, which can be done by running the algorithm a constant number of times.

Finally, recall that P/poly is the class of polynomial time "algorithms with advice" (or non-uniform polynomial computation). In particular, it is the class of languages that have a polynomial time algorithm $\mathcal{A}(\cdot, \cdot)$ such that for any $n \in \mathbb{N}$, there exists y_n such that for all $x \in \{0,1\}^n$, $\mathcal{A}(x, y_n)$ is 1 if and only if $x \in L$. (Additionally, y_n must have size polynomial in n.) Note that the relationship between P/poly and NP is also unknown.

Theorem 1 (Adelman's Theorem)**.** *$BPP \subseteq P/poly$*

Proof. Consider a language $L \in$ BPP. We have an algorithm $\mathcal{A}(\cdot, \cdot)$ such that for any x, $\mathcal{A}(x, y)$ is correct for at least $1 - \delta$ of the strings $y \in \{0,1\}^{p(n)}$ for some polynomial $p(n)$ where $n = |x|$. The probability that $\mathcal{A}$ gives the wrong answer on x for a randomly chosen y is at most δ, so by the union bound the probability that a given choice of y gives the wrong answer on *some* x is at most $\delta \cdot 2^n$, as there are 2^n inputs of length n. If we set $\delta < 1/2^n$, this probability is smaller than 1, so there exists some string that does not give the wrong answer on any input. (Note that the probability of the BPP algorithm succeeding can be boosted to $1 - \delta$ by running it repeatedly and taking the majority answer/using Chernoff bounds.) □

References

[A78] Leonard Adleman. Two Theorems on Random Polynomial Time. *Proceedings of the Nineteenth Annual IEEE Symposium on Foundations of Computer Science*, pp. 75-83, 1978.

[BG81] Charles H. Bennet, John Gill. Relative to a random oracle A, $P^A \neq NP^A \neq coNP^A$ with probability 1. *J. Comput. Syst. Sci.*, 10(1):96–113, 1981.

[M95] Rajeev Motwani, Prabhakar Raghavan. "Game-theoretic Techniques". *Randomized Alogorithms. Cambrige University Press*, pp. 28-30, 1995.