Game-Theoretic Techniques

Jonathan Crowell

Department of Computer Science and Electrical Engineering,
West Virginia University,
Morgantown, WV
crowell@csee.wvu.edu

1 Introduction and Goals

In previous lectures we presented a randomized quicksort algorithm that ran in time $O(n \log n)$. However, there are deterministic sorting algorithms which also run in time $O(n \log n)$, leading one to question the utility of randomization. In this section we present a randomized algorithm for a problem such that:

1. Its expected run time is better than the worst-case complexity of any deterministic algorithm, and
2. No randomized algorithm can do better (i.e. it will establish a lower bound on the running time of any randomized algorithm.)

2 Game Tree Evaluation

Game trees are extensively used in the artificial intelligence field, especially in game-playing problems. Consider a computer that plays a game of tic-tac-toe against a user. At the root of the game tree is an tic-tac-toe grid with the computer's first move marked in (we assume the computer always goes first with the same first move). This root has eight children, corresponding to the eight different answering moves the user can make. Each of these children has more children in turn, corresponding to the subsequent moves that can be made.

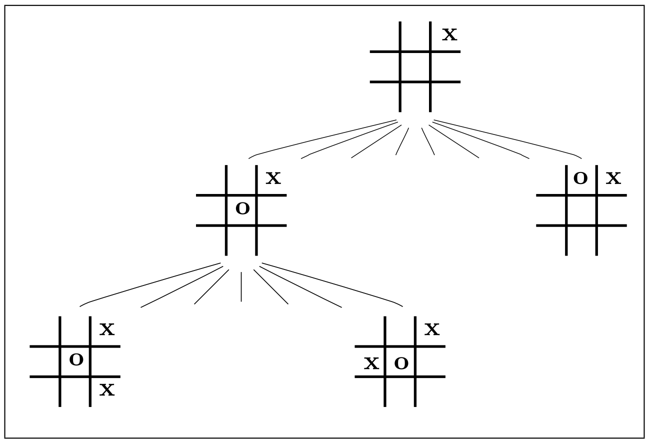

Figure 1: Part of a possible game-tree for tic-tac-toe

The leaves of the tree represent the value of the game. One player tries to maximize this value, while the other tries to minimize it. In the case of tic-tac-toe the tree will be finite, but this need not always be the case. The game of chess, for instance, could have an infinite game tree. Deciding who will win a game given a starting configuration is known as the *game tree problem.*

2.1 MIN-MAX and AND-OR Game Trees

For our purposes, a limited definition of a game tree will suffice:

Definition 2.1 *A* **game tree** *is a full, balanced, binary tree in which internal nodes at an even distance from the root are labeled MIN and internal nodes at an odd distance (including the root) are labeled MAX. Each node of the tree is a record containing four fields: "type," "l-child," "r-child," and "value." "Type" is MIN, MAX, or LEAF, "l-child" and "r-child" are pointers to the node's left and right children respectively (which will be null in the case of a leaf), and "value" is the value returned by the node – either* 0 *or* 1*. The value returned by a MIN node is the lesser of the values of its two children, and the value returned by a MAX node is the greater of the values of its two children. Initially only the leaves have values.*

We make the following observations about game trees as defined above:

Observation 2.1 *Every root-to-leaf path goes through the same number of MIN and MAX nodes (including the root).*

Observation 2.2 *If the number in observation (2.1) is k, then the depth of the tree is $2k$ and the number of nodes in the tree is $2^{2k} - 1 = 4^k - 1$ (excluding the leaves). The number of leaves is $2^{2k} = 4^k$*

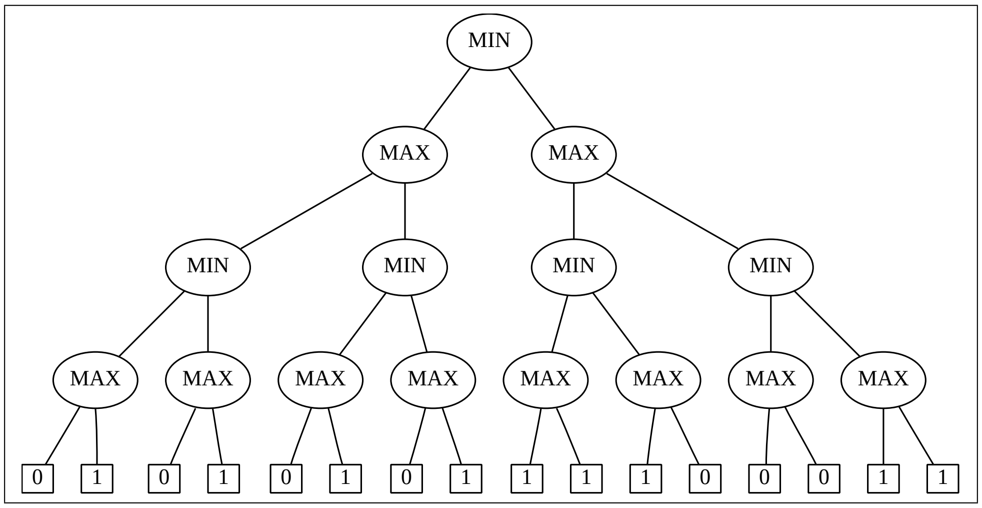

Figure 2: A min-max tree of depth $2k = 2 \times 2 = 4$, with $2^{2k} - 1 = 4^2 - 1 = 15$ internal nodes and $2^{2k} = 4^2 = 16$ leaves.

A game tree in which each node has d children and there are k MAX and k MIN nodes is signified by $T_{d,k}$. We will work exclusively with binary game trees, denoted by $T_{2,k}$. In binary computer science terms, a MIN node can be thought of as a logical AND and a MAX node as a logical OR.

Input	Output of a MIN or AND node	Output of a MAX or OR node
0, 0	0	0
0, 1	0	1
1, 0	0	1
1, 1	1	1

Figure 3: Input and output within a game tree

We are interested in the evaluation problem, which is:

Given $T_{2,k}$ and leaf values, what is the value returned by the root?

2.2 Deterministic Evaluation

Algorithm (2.1) is a recursive deterministic algorithm for game tree evaluation which can be called from the main program with the command `Evaluate`(root). Note that the algorithm does not necessarily evaluate every node. In the case of an OR node, if one child returns a 1 then evaluation of the other node is unnecessary since regardless of whether it is a 1 or a 0, the OR node will return a 1. Likewise, if an AND node is found to have one 0 child, we know immediately that the node will return the value 0.

```
Evaluate(node)
if (node.type = "LEAF") then
   return(node.value)
end if
if (node.type = "OR") then
   p1 = Evaluate(node.right)
   if (p1 = 1) then
      return p1
   else
      return (Evaluate(node.left))
   end if
end if
if (node.type = "AND") then
   p1 = Evaluate(node.left)
   if (p1 = 0) then
      return p1
   else
      return (Evaluate(node.right)
   end if
end if
```

Algorithm 2.1: Deterministic evaluation of an and-or game tree

Algorithm (2.1) is representative of all deterministic algorithms in that it can be forced to read all the leaves of the tree. An opponent who is aware of the deterministic evaluation strategy can always assign initial leaf values that will force the algorithm to evaluate every node (see [1], prob. 2.1, pg. 41). So, for example, if the evaluation of an OR node that returns a 1 begins with the evaluation of the right child, the opponent will make sure to hide the 1 on the left and arrange for a 0 on the right. Since any deterministic algorithm can be made to read all 4^k leaves of on some instance of $T_{2,k}$, the worst case cost for any deterministic algorithm is $4^k = n$.

2.3 Randomized Evaluation

Algorithm (2.2) is a randomized evaluation algorithm that uses coin tosses. Whether the algorithm will evaluate the right child first or the left child first is dependent on the coin toss and is impossible for an adversary to know in advance.

Note that algorithm (2.2) is a Las Vegas algorithm in that it always returns the correct result, and in terms of deterministic efficiency is no worse than a deterministic algorithm.

2.4 Cost Analysis of Randomized AND-OR Game Tree Evaluation

Given the randomized evaluation strategy, we would like to know what the expected cost of evaluating a node is. To begin, consider an OR node with two leaves. If the node is to return a 1, then at least one of the leaves must return a 1. With probability 1/2 this leaf will be selected first and only one evaluation will need to be done. The other possibility (also occurring with probability 1/2) is that both leaves will need to be evaluated. The expected number of evaluations, $\mathbf{E}_1^{\text{OR}}$, is given by

$$\begin{aligned} \mathbf{E}_1^{\text{OR}} &= (\frac{1}{2} \times 1) + (\frac{1}{2} \times 2) \\ &= \frac{3}{2} \end{aligned} \tag{1}$$

```
RandEvaluate(node)
if (node.type = "LEAF") then
    return(node.value)
end if
if (node.type = "'OR") then
    Toss a coin
    if (Heads) then
        p1 = RandEvaluate(node.left)
        if (p1 = 1) then
            return p1
        else
            return(RandEvaluate(node.right))
        end if
    end if
    if (Tails) then
        p2 = RandEvaluate(node.right)
        if (p2 = 1) then
            return p2
        else
            return(RandEvaluate(node.left))
        end if
    end if
end if
if (node.type = "AND") then
    Toss a coin
    if (Heads) then
        p1 = RandEvaluate(node.left)
        if (p1 = 0) then
            return p1
        else
            return(RandEvaluate(node.right))
        end if
    end if
    if (Tails) then
        p2 = RandEvaluate(node.right)
        if (p2 = 0) then
            return p2
        else
            return(RandEvaluate(node.left))
        end if
    end if
end if
```

Algorithm 2.2: Randomized evaluation of an AND-OR game tree

Similar analysis with the 0s and 1s switched can be applied to the AND nodes to find that the expected cost of evaluating an AND node that is to return a 0 is:

$$\begin{aligned}\mathbf{E}_0^{\text{AND}} &= (\frac{1}{2} \times 1) + (\frac{1}{2} \times 2) \\ &= \frac{3}{2}\end{aligned} \tag{2}$$

In the cases in which the AND node returns a 1 or the OR node returns a 0, it may seem at first that the randomized algorithm has no advantage, since two evaluations will necessarily be required. For an AND node to return a 1, however, both of its OR-node children must also return 1s — which is the "good" case for them. Similarly, for an OR node to return a 0, both of its AND-node children must also return a 0 — which is the "good" case for them. The algorithm benefits because what is bad for an AND node is good for an OR node and vice versa.

Lemma: 2.1 *The expected cost of the randomized algorithm for evaluating $T_{2,k}$ is at most 3^k.*

Proof: *By induction on k.*

Base Case: $k = 1$

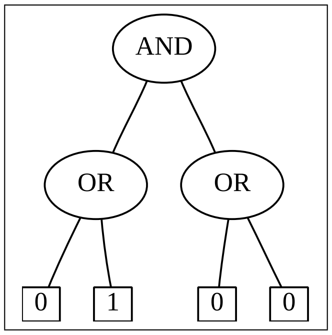

Figure 4: A possible base case game tree

$T_{2,1}$ will have one AND node at the root, two OR nodes as children, and four leaves, as in figure (4). There are two cases to consider: those in which the root returns a 1, and those in which the root returns a 0.

Case 1: *If the root of T evaluates to 0, then at least one of the OR nodes must evaluate to 0. With probability 1/2 this OR node is chosen first, incurring a cost of 2 (if the OR node returns a 0, it will have had to evaluate both leaves), and with probability 1/2 both sub-trees will need to be evaluated, incurring a cost of $1/2 \times 3/2 + 1/2 \times (3/2 + 2)$. The expected cost of evaluating T, then, is at most*

$$\frac{1}{2} \times 2 + \frac{1}{2} \times \left(\frac{3}{2} + 2\right) = 2\tfrac{3}{4}$$

Case 2: *If the root of T evaluates to 1, then both OR nodes will have to be evaluated, and the cost will be $2 \times 3/2 = 3$.*

The cost for evaluating the tree, then, is $\leqslant 3$, and since $k = 1$ in this case and $3^k = 3$, the cost is $\leqslant 3^k$, and the base case is established.

Inductive Case:

Assume that for all $T_{2,k-1}$, the expected evaluation cost $\leqslant 3^{k-1}$. Consider a tree T whose root is an OR node with two children, each of which is the root of a $T_{2,k-1}$ game tree. Figure (5a) depicts such a tree. The evaluation of such a tree involves two cases: those in which the root evaluates to 1, and those in which it evaluates to 0.

The OR-node cases

Case 1: *If the root of T evaluates to 1, then at least one of the two $T_{2,k-1}$ sub-trees must evaluate to 1. With probability 1/2 this sub-tree is chosen first, incurring a cost $\leqslant 3^{k-1}$, and with probability at most 1/2 both sub-trees will need to be evaluated, incurring a cost of $2 \times 3^{k-1}$. The expected cost of evaluating T, then, is at most*

$$\frac{1}{2} \times 3^{k-1} + \frac{1}{2} \times 2 \times 3^{k-1} = \frac{3}{2} \times 3^{k-1}$$

Case 2: *If the root of T evaluates to 0, then both sub-trees will have to be evaluated, and the cost will be $2 \times 3^{k-1}$.*

The AND-node cases

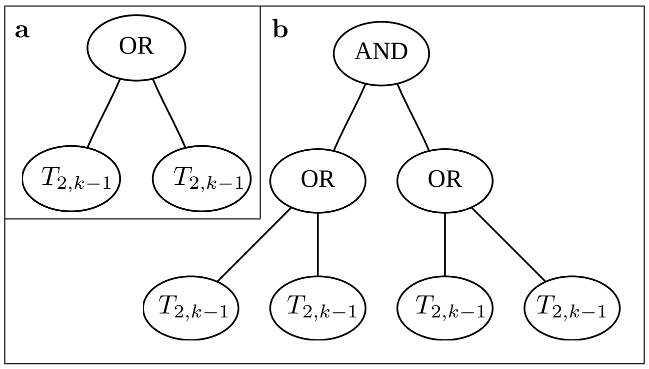

Figure 5: **a** is an OR node with two $T_{2,k-1}$ subtrees, and **b** is the full $T_{2,k}$ rooted at an AND node which connects two copies of the tree from **a**.

The root $T_{2,k}$ is, by definition (2.1), an AND node. The children of this root will be sub-trees rooted at OR nodes - like the one just discussed (see figure (5)). The evaluation of $T_{2,k}$ can again be broken down in to two cases: those in which the root returns a 1, and those in which it returns a 0.

Case 1: *If $T_{2,k}$ evaluates to 1, then both sub-trees rooted at OR nodes return 1. By equation (1) and linearity of expectation, the expected cost of evaluating $T_{2,k}$ is*

$$2 \times \frac{3}{2} \times 3^{k-1} = 3^k. \tag{3}$$

Case 2: *If $T_{2,k}$ evaluates to 0, then at least one of its children returns 0. With probability 1/2 this child is chosen first, so the expected cost of evaluating $T_{2,k}$ is at most*

$$\frac{1}{2}\left(2 \times 3^{k-1}\right) + \frac{1}{2}\left(\frac{3}{2} \times 3^{k-1} + 2 \times 3^{k-1}\right) = 2\tfrac{3}{4} \times 3^{k-1} \leqslant 3^k$$

So we see that the expected cost of evaluating $T_{2,k}$ does not exceed 3^k in either case. □

We have established that algorithm (2.2) has a cost $\leqslant 3^k$ when evaluating an AND-OR tree. We know that such a tree has $n = 4^k$ leaves, and therefore $k = \log_4 n$. If we substitute $\log_4 n$ for k in the cost of algorithm (2.2), we see that the cost is $\leqslant 3^{\log_4 n}$. By the rule of logarithms which states that $x^{\log_a b} = b^{\log_a x}$ we have the cost of algorithm (2.2) $\leqslant n^{log_4 3} = n^{0.793}$.

At the beginning of this chapter, two goals were stated. The first was to present a randomized algorithm that has a smaller expected running time than any deterministic algorithm. As we saw in the analysis of algorithm (2.1), the cost for a deterministic solution to this problem is n. By proving that our randomized algorithm has an expected cost of only $n^{0.793}$ we have accomplished our first goal.

3 The Minimax Principle

Our next goal is to establish that no randomized algorithm can have a lower expected running time. The standard technique for proving a lower bound on a randomized algorithm is *the minimax principle.* We begin our discussion with an overview of game theory (not directly related to the game trees discussed above).

3.1 Two-Person Games

Consider the game Paper-Rock-Scissors, known and loved the world over. It is played between two people who simultaneously make a sign signifying paper, rock, or scissors. Paper beats rock by covering it, rock beats scissors by denting it, and scissors beat paper by cutting it. When the same sign is chosen, the outcome is a draw. To make the game more interesting for our purposes, we further state that the winner must pay the loser a dollar.

Paper-Rock-Scissors is known as a *two-person zero-sum game*, and we can represent it using a payoff matrix, **M**, such as the one in figure(6).

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

Figure 6: Payoff matrix, **M**, for Paper-Rock-Scissors. The numbers represent the amount paid by **C** (the column player) to **R** (the row player) for the different scenarios. If the number is negative, then **C** receives money.

The game is called a zero-sum game because the net amount won by the two participants (call them **R** and **C** — for Rebecca and Courtney (or Row and Column)) is zero. One of them may or may not walk away with more money at the end of the contest, but the total amount of money between them will not have decreased or increased. In general, we may represent any two-person zero-sum game with an $m \times n$ payoff matrix. m and n do not have to be equal — it may be that one competitor has many strategies to choose from, while the other has only a few.

The guiding force in picking a strategy is the minimization of loss. Assume this is a *zero-information* game, meaning that neither player has any information about her opponent's strategy. Suppose that **R** chooses strategy i. Then she is guaranteed a payoff of $\min_j M_{ij}$ regardless of which strategy **C** chooses. The *optimal strategy*, V, is one which maximizes her minimum payoff. $V_{\mathrm{R}} = \max_i \min_j M_{ij}$ is the lower bound on the payoff to **R** when she uses her optimal strategy. An optimal strategy for **C** is one that minimizes the maximum she will have to pay to **R**. So $V_{\mathrm{C}} = \min_j \max_i M_{ij}$. The following inequality holds for all payoff matrices:

$$\max_i \min_j M_{ij} \leqslant \min_j \max_i M_{ij}$$

<u>Proof</u>: *We assume that* **M** *is* $n \times m$. *Let* $S = \{min_j M_{1j}, min_j M_{2j}, \ldots, min_j M_{nj}\}$ *and* $T = \{max_i M_{i1}, max_i M_{i2}, \ldots, max_i M_{im}\}$ *Thus, we have to show that the maximal element of S is less than or equal to the minimal element of T. Instead, we will show that each element of S is $\leq$ each element of T; the theorem follows.*

Consider row 1; let the minimum element in this row be $M_{1j'}$ Clearly $M_{1j'}$ can at most equal the maximum in column j'. Now pick any column other than j', say k. If the maximum element in column k, is not in row 1, (say it is row i') then $M_{1j'}$ must be less than this element, since $M_{1j'} \leq M_{1k} \leq M_{i'k}$. On the other hand, if the maximum element is in row 1, then $M_{1j'}$ cannot exceed that element, since $M_{1j'}$ is the minimum element of row 1.

Since row 1 was picked arbitrarily, the same argument holds for each element in S.

□

When $V_{\mathrm{R}} = V_{\mathrm{C}}$, the game is said to have a solution, V, called the *saddle point*, which is the specific choice of optimal strategies that lead to that payoff. In our example of Paper-Rock-Scissors, $V_{\mathrm{R}} = -1$ and $V_{\mathrm{C}} = 1$, so the game does not have a solution. In the following modified payoff matrix for Paper-Rock-Scissors, however, note that $\max_i \min_j M_{ij} = 0 = \min_j \max_i M_{ij}$, so this altered form of the game does have a solution.

	Paper	Rock	Scissors
Paper	0	1	2
Rock	-1	0	1
Scissors	-2	-1	0

Figure 7: Modified payoff matrix, **M**, for Paper-Rock-Scissors that has a saddle point at Paper-Paper.

When a game such as the original version of Paper-Rock-Scissors has no solution, then there is no optimal strategy for either player. In this case, any information about the other player's strategy can be used to improve the payoff, and an opponent who gains knowledge of a deterministic strategy for a game will soon gain the upper hand. A randomized strategy, however, holds this devious technique at bay.

3.2 Randomized Gaming Strategies

In game theory parlance a deterministic strategy is known as a *pure* strategy and a randomized strategy is known as a *mixed* strategy. A mixed strategy is a probability distribution on the set of possible strategies. Consider a probability distribution vector, $\vec{\mathbf{p}}$, on the rows, and a probability distribution vector, $\vec{\mathbf{q}}$, on the columns. We can say the following things about $\vec{\mathbf{p}}$ and $\vec{\mathbf{q}}$:

1. Every element in $\vec{\mathbf{p}}$ and $\vec{\mathbf{q}}$ is between 0 and 1.
2. All the elements in $\vec{\mathbf{p}}$ sum to 1, as do all the elements in $\vec{\mathbf{q}}$
3. **R** will pick row i with probability p_i and **C** will pick column j with probability q_j.

The payoff is a random variable, and its expectation is calculated by multiplying each entry $M_{ij} \in \mathbf{M}$ by the probability, q_j that it will be selected by C and the probability p_i that it will be selected by R, and then adding up all these adjusted entries. This is equivalent to multiplying the matrix **M** by the transposed vector $\vec{\mathbf{p}}^{\mathrm{T}}$ (the transposition is necessary for correct multiplication), and then by the vector $\vec{\mathbf{q}}$. The following equation, then, gives the expected payoff:

$$\mathbf{E}[\text{payoff}] = \sum_{i=1}^{n}\sum_{j=1}^{m} p_i M_{ij} q_j = \vec{\mathbf{p}}^{\mathrm{T}}\mathbf{M}\vec{\mathbf{q}} \tag{4}$$

In the randomized technique, choosing a strategy amounts to choosing a probability distribution vector from the set of all probability distributions. The optimal strategies for **R** and **C** are

$$V_{\mathrm{R}} = \max_{\vec{\mathbf{p}}} \min_{\vec{\mathbf{q}}} \vec{\mathbf{p}}^{\mathrm{T}}\mathbf{M}\vec{\mathbf{q}}$$
$$V_{\mathrm{C}} = \min_{\vec{\mathbf{q}}} \max_{\vec{\mathbf{p}}} \vec{\mathbf{p}}^{\mathrm{T}}\mathbf{M}\vec{\mathbf{q}}$$

where the min and max range over all possible probability distributions.

Theorem: 3.1 **von Neumann's Minimax Theorem:** *For any two-person game, specified by* **M**, $V_R = V_C$, *i.e. all games have solutions when randomized strategies are used.*

The particular probability distributions chosen by **R** and **C** which lead to the solution are denoted $\hat{\vec{\mathbf{p}}}$ and $\hat{\vec{\mathbf{q}}}$ respectively. The saddle point is denoted $(\hat{\vec{\mathbf{p}}}, \hat{\vec{\mathbf{q}}})$.

Corollary: 3.1 *The largest expected value that* **R** *can guarantee using a mixed strategy is equal to the smallest expected value that* **C** *can guarantee using a mixed strategy.*

If $\vec{\mathbf{p}}$ is fixed then, since **M** is known, $\vec{\mathbf{p}}^{\mathrm{T}}\mathbf{M}\vec{\mathbf{q}}$ reduces to a linear expresion in $\vec{\mathbf{q}}$ of the form $(c_1q_1 + c_2q_2 + ... + c_mq_m)$. The opponent can now minimize his payoff by setting to 1 the q_j that has the smallest coefficient. So, interestingly, if **C** knows the probability distribution that **R** is using, her mixed strategy reduces to a pure strategy. This also works in the opposite direction, of course. This observation leads to a simplified version of von Neumann's minimax theorem. Let $\vec{\mathbf{e_k}}$ denote a vector with 1 in the kth position and 0 everywhere else.

Theorem: 3.2 **Loomis' Theorem:** *For any two-person zero-sum game specified by a matrix* $\mathbf{M}$*,*

$$\max_{\vec{\mathbf{p}}} \min_{j} \vec{\mathbf{p}}^T \mathbf{M} \vec{\mathbf{e_j}} = \min_{\vec{\mathbf{q}}} \max_{i} \vec{\mathbf{e_i}}^T \mathbf{M} \vec{\mathbf{q}}$$

Note that in the theorem $\vec{\mathbf{e_j}}$ has the same dimension as $\vec{\mathbf{q}}$, and $\vec{\mathbf{e_i}}$ has the same dimension as $\vec{\mathbf{p}}$.

We now have all the ingredients we need in order to achieve the second goal that was stated at the begining of this chapter, i.e. establish the a lower bound on the running time of any randomized algorithm.

3.3 Applying Game Theory to Algorithm Design: Yao's Technique

Let us redefine the roles of **R** and **C** in the above discussion. Suppose **C** is an algorithm designer. **R** is her adversary and is responsible for designing inputs that will thwart **C**'s algorithms. In the payoff matrix,

$$\begin{aligned} \text{columns} &= \{\text{all possible correct, deterministic, terminating algorithms for input of fixed size}\}, \\ \text{rows} &= \{\text{all possible inputs of a fixed size}\}, \text{ and} \\ M_{ij} &= \text{time taken by algorithm } A_j \text{ on input } I_i. \end{aligned}$$

Both sets are taken to be finite.

A pure strategy for **C** is the selection of one of the algorithms, and an optimal pure strategy for **C** is the choice of the best worst-case deterministic algorithm for solving the problem. A pure strategy for **R** is the selection of one of the inputs.

Definition 3.1 *The* **Deterministic Complexity** *of a problem is the worst case running time of any deterministic algorithm for the problem.*

A mixed strategy for **C** is the selection of a probability distribution over all the algorithms. Likewise, **R** is a probability distribution over the inputs. An optimal mixed strategy for **C** is an optimal Las Vegas algorithm.

Definition 3.2 *The* **Distributional Complexity** *of a problem is the expected running time of the best deterministic algorithm for the worst distribution on the inputs. (This complexity is smaller than the deterministic complexity since the algorithm knows the input distribution).*

We can now restate the theorems from the previous section:

Theorem: 3.3 **von Neumann/Loomis restatement:** *The distributional complexity of a problem equals the least possible expected running time achievable by any randomized algorithm.*

Corollary: 3.2 *Let* Π *be a problem with a finite set* $\mathcal{I}$ *of input instances of a fixed size and a finite set of deterministic algorithms,* $\mathcal{A}$*. For input* $I \in \mathcal{I}$ *and algorithm* $A \in \mathcal{A}$*, let* $C(I, A)$ *denote the running time of algorithm* A *on input* I*. For probability distributions* $\vec{\mathbf{p}}$ *over* $\mathcal{I}$ *and* $\vec{\mathbf{q}}$ *over* $\mathcal{A}$*, let* $I_{\vec{\mathbf{p}}}$ *denote a random input chosen according to* $\vec{\mathbf{p}}$ *and* $A_{\vec{\mathbf{q}}}$ *denote a random algorithm chosen according to* $\vec{\mathbf{q}}$*. Then,*

$$\max_{\vec{\mathbf{p}}} \min_{\vec{\mathbf{q}}} \mathbf{E}[C(I_{\vec{\mathbf{p}}}, A_{\vec{\mathbf{q}}})] = \min_{\vec{\mathbf{q}}} \max_{\vec{\mathbf{p}}} \mathbf{E}[C(I_{\vec{\mathbf{p}}}, A_{\vec{\mathbf{q}}})] \tag{5}$$

and

$$\max_{\vec{\mathbf{p}}} \min_{A \in \mathcal{A}} \mathbf{E}[C(I_{\vec{\mathbf{p}}}, A)] = \min_{\vec{\mathbf{q}}} \max_{I \in \mathcal{I}} \mathbf{E}[C(I, A_{\vec{\mathbf{q}}})] \tag{6}$$

This leads us directly to

Proposition: 3.1 **Yao's Minimax Principle:** *For all distributions* $\vec{\mathbf{p}}$ *over* $\mathcal{I}$ *and* $\vec{\mathbf{q}}$ *over* $\mathcal{A}$

$$\min_{A \in \mathcal{A}} \mathbf{E}[C(I_{\vec{\mathbf{p}}}, A)] \leqslant \max_{I \in \mathcal{I}} \mathbf{E}[C(I, A_{\vec{\mathbf{q}}})]$$

Observation 3.1 *The expected running time of the optimal deterministic algorithm for an arbitrarily chosen input distribution* $\vec{\mathbf{p}}$ *is a lower bound on the expected running time of the optimal randomized algorithm for the problem.*

This gives us the strategy for achieving the second goal: We choose any distribution $\vec{\mathbf{p}}$ on the input and prove a lower bound on the expected running time of all deterministic algorithms for that distribution. The strength of this strategy is that we can pick our $\vec{\mathbf{p}}$ to be a very convenient distribution.

4 Back to the AND-OR trees, but now with NORs

Let us now return to our original problem of evaluating AND-OR game trees. Applying Yao's principle to the problem of game tree evaluation will give us a lower bound on any randomized algorithm for the problem. Our discussion will be simplified if we think in terms of a NOR tree rather than an AND-OR tree.

Lemma: 4.1 *The tree $T_{2,k}$ is equivalent to a balanced binary tree all of whose leaves are at distance $2k$ from the root, and all of whose internal nodes compute the NOR function (the NOR function returns the value 1 if both inputs are 0, and returns 0 otherwise).*

Proof: **By Induction**

Base Case: *A tree of two levels*

Take a tree of depth $2k$ and 4 leaves, such as the one in figure (8). We need to prove that the AND of the two OR nodes in (8a) returns the same result as the NOR of the two NOR nodes in (8b).

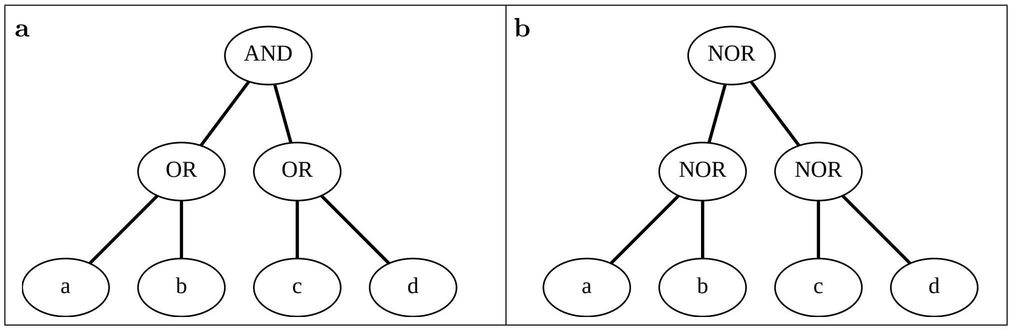

Figure 8: **a** is a base-case AND-OR tree and **b** is an equivalent NOR tree.

The result returned by the AND-OR tree will be $(a \wedge b) \vee (c \wedge d)$, and the result returned by the NOR tree will be $((a \wedge b)' \wedge (c \wedge d)')'$ (a NOR is simply the complement of an OR). Call the result Z:

$$\begin{aligned} Z &= ((a \vee b)' \vee (c \vee d)')' && [\textit{Initial NOR tree result}] \\ &= ((a \vee b)')' \wedge ((c \vee d)')' && [\textit{DeMorgan's Law}] \\ &= (a \vee b) \wedge (c \vee d) && [\textit{Removal of double complements}] \end{aligned}$$

Inductive Case:
Assume that any NOR tree of depth 2k-2 returns the same result as an AND-OR tree of the same depth. Now consider a NOR tree of depth 2k. It can be represented as figure (9b), where J1, J2, J3, and J4 are NOR tres of depth 2k-2. Consider the AND-OR tree represented in figure (9a). by inductive hypothesis, J1, J2, J3, and J4 will return the same values.

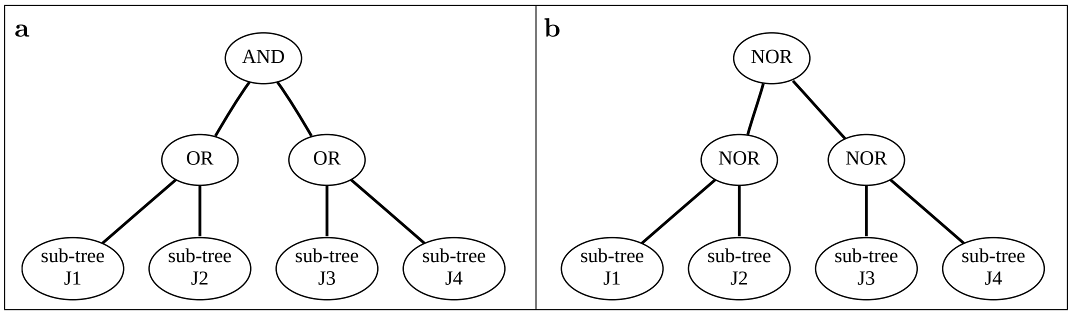

Figure 9: **a** is an inductive-case AND-OR tree and **b** is an equivalent NOR tree.

Let the results returned by each of the four sub-trees be a, b, c, and d respectively. The result returned by the AND-OR tree will be $(a \wedge b) \vee (c \wedge d)$, and the result returned by the NOR tree will be $((a \wedge b)' \wedge (c \wedge d)')'$. The fact that these two *statements are equivalent was proved in the base case.* □

We are now ready to employ our strategy: we will choose a convenient distribution $\vec{\mathbf{p}}$ on the input and prove a lower bound on the expected running time of all deterministic algorithms for that distribution. The input in this case corresponds to the values of the leaves on the tree we are given to evaluate.

Suppose that each leaf is set to 1 with probability $p = (3 - \sqrt{5})/2$. Then the probability that the input to a NOR node is 0 is $1 - (3 - \sqrt{5})/2$, which is $(\sqrt{5} - 1)/2$. The probability that the output of a NOR node is 1 is equal to the probability that both its inputs are 0, which is

$$\left(\frac{\sqrt{5}-1}{2}\right)^2 = \frac{3-\sqrt{5}}{2} = p$$

So the probability that a NOR node with two leaves will return a 1 is p. Note that the probability, p, of a 1 being returned has, in effect, moved up a level in the tree. Likewise, two sibling NOR nodes returning 1 with probability p will cause their parent NOR node to return 1 with probability p. The probability distribution, p, of a node returning a 1 moves uniformly up the tree, so that ultimately every NOR node returns a 1 with probability p.

4.1 Depth-First Pruning

Consider a deterministic algorithm that is trying to evaluate a NOR tree. In order to evaluate the fewest leaves possible, the algorithm should determine the value of one entire sub-tree of a particular node before visiting any of the leaves of the other sub-tree since the value of the first sub-tree may render evaluation of the second sub-tree moot. If the value of the first sub-tree is 1, then the other sub-tree can be "pruned" immediately since the NOR node will return a 0 regardless of the value of the other sub-tree. When evaluating the first sub-tree, however, the algorithm should continue to make use of this short-cut and prune whenever possible. This is known as depth-first pruning.

Proposition: 4.1 *Let T be a NOR tree each of whose leaves is independently set to 1 with probability q for a fixed value $q \in [0, 1]$. Let W_T denote the minimum, over all deterministic algorithms, of the expected number of steps to evaluate T. Then, there is a depth-first pruning algorithm that evaluates T in the same number of expected steps, which is W_T. (See [3] for a proof.)*

Proposition (4.1) tells us that for the purposes of our lower bound, we may restrict our attention to depth-first pruning algorithms. For a proof of this proposition, see

Let W_h be the expected number of steps to evaluate a tree of height h that has n nodes, each of which is set to 1 with probability $p = (3 - \sqrt{5})/2$. We want to know what W_h is as a function of W_{h-1}. That is, what is the probability with which you will have to evaluate one or both of the sub-trees? Obviously

$$W_h = W_{h-1} + (1-p) \times W_{h-1} \tag{7}$$

The first term is the expected value of evaluating the first sub-tree, and the second term is the expected value of evaluating the second sub-tree, which will only have to be done when the first sub-tree returns a 0 — an even that will occur with probability $1 - p$.

Another way to formulate equation (7) is by noting that with probability p we will have to evaluate only one sub-tree, and with probability $(1-p)$ we will have to evaluate both sub-trees. Since the tree is a full binary tree of height $h = log_2 n$, we can substitute $\log_2 n$ for h. We can also substitute $(3 - \sqrt{5})/2$ for p:

$$\begin{aligned}
W_h &= p \times W_{h-1} + (1-p) \times [2 \times W_{h-1}] \\
&= (2-p)W_{h-1} \\
&= (2-p)[(2-p)W_{h-2}] \\
&= (2-p)^{h-1}W_1 \\
&= (2-p)^h \\
&= (2 - \tfrac{2-\sqrt{5}}{2})^{\log_2 n} \\
&= n^{\log_2(2-\frac{3-\sqrt{5}}{2})} \\
&= n^{\log_2(\frac{1+\sqrt{5}}{2})} \\
&= n^{0.694}
\end{aligned}$$

Our lower bound of $n^{0.694}$ is less than the expected run time of our randomized algorithm, which was at most $n^{0.793}$. This means we have not accomplished our second goal. The reason for this, however, is that the probability distribution that we chose for our analysis was extremely convenient. Since no reasonable algorithm would evaluate both children of a NOR node if both of them return 1, we need to construct a random distribution in which the values assigned to the leaves are not independent of each other — and thereby preclude the possibility of both inputs to a NOR node being 1. Only then would we be able to prove the best lower bound. This would require a much more difficult analysis, however, which we will not address but which is available in [2] for those who are interested.

References

[1] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms.* Cambridge University Press, 1995.

[2] M. Saks and A. Wigderson. Probabilistic boolean decision trees and the complexity of evaluating game trees. *Proceedings of the 27th Annual IEEE Symposium on Foundations of Computer Science*, pages 29–38, 1986.

[3] M. Tarsi. Optimal search on some game trees. *Journal of the ACM*, 30:389–396, 1983.