A (very) brief introduction to Game Theory

David Kempe

November 22, 2018

Abstract

This is an absolutely minimal introduction to game theory, with a focus on (mostly zero-sum) matrix games and equilibria thereof. It then goes on to relate equilibria to linear programming, and to lower bounds for randomized algorithms. Any reader who wants a more comprehensive or better introduction is referred to several classic and new books on the subject, listed below.

1 Basics of Game Theory

Game theory is a framework (or set of tools) for reasoning about the outcomes of interactions between *agents* (also called *players*) who pursue goals that may not be aligned with each other, and whose actions affect each other. A lot of interactions can be fruitfully viewed through the lens and analysis techniques of game theory, including economic interactions between individuals or companies, social interactions, and many scenarios in which individuals compete for limited resources (such as traffic, access to desirable locations, and many others).

Typically, there is a set of agents, and one specifies what the possible actions and interactions are, as well as how they jointly affect the *utility* of each agent, which is a suitable measure of how "happy" the agent is with the outcome. Defining the "right" utility for a real-world scenario is of course highly non-trivial.

Another important aspect of specifying a game-theoretic interaction is the knowledge that players have about the utility (which may depend on unobservable factors), and about each other. Beyond that, typically, classical game theory assumes that players are rational and capable of identifying which action is best for them, given the actions chosen by other players. Naturally, this assumption is questionable (though often necessary to derive mathematical insights), and a large body of work studies various deviations from this assumption, often under the name "behavioral game theory."

Yet another important consideration is what we would consider the "outcome" of a game. If our goal is to analyze or evaluate the outcomes, we need to specify what we mean by this. There are a number of different notions, appropriate for different types of games, and we will discuss some of them below.

When evaluating the outcome of a game, we are often interested in how well some aggregate "goal" is met, such as the total utility of all agents, or other measures of social benefit. The presence of selfish agents will typically lead to a worse societal outcome than if some benevolent powerful authority chose the actions for everyone. The resulting loss in overall utility has been analyzed a lot, and is somewhat akin to the notion of approximation algorithms. We will discuss it a bit more below.

Another level beyond merely *analyzing* games is to *design* the game in the first place. Often, there is a principal entity (e.g., government, owner of a site, etc.) who has the power to define some of the rules of the game, which may include the available actions or utilities of the agents. For instance, think of a traffic designer who may choose whether to put carpool lanes or paid car lanes on a freeway, or how high to set fines for violating the rules governing those lanes, or how often to have police patrol them. In so doing, the traffic designer will change the payoff for using carpool lanes, and thus the outcome of the game when the players subsequently act selfishly. Generally, this area is called *mechanism design*. It can be considered as akin to algorithm design, except the algorithm is going to be jointly executed by multiple selfish agents.

2 Two-Player Matrix Games and Beyond

Perhaps the simplest kind of game is two-player matrix games; if the number of actions for each player is small enough, one can explicitly write down the utility to each player as a matrix. Such games are even simpler when they are strictly competitive, in the sense that one player's gain is exactly the other's loss. Such games are called *zero-sum games*, because the sum of the payoffs of the two players is always 0. A very familiar example is *Rock-Paper Scissors*, with payoffs given in Table 1:

	R	P	S
R	0	-1	1
P	1	0	-1
S	-1	1	-1

Table 1: The payoff matrix A for Rock-Paper-Scissor

We have two players, often called the row player and the column player. Let's call them Rohit and Colette. The matrix A gives Rohit's payoff when he plays the strategy given by the name of the row, and Colette plays the strategy given by the name of the column. Because the game is zero-sum (there's a loser and a winner, or a tie), Colette's payoff is the negative of the given values.

Not all games are zero-sum, and not all games have the same number (or set) of strategies for both players. Table 2 is another example of a game, which we will call "Beach or Mountain."

	B	M
B	3,1	-3,-3
M	-3,-3	1,3
H	-3,-3	-3,-3

Table 2: The payoff matrices A_R, A_C for "Beach or Mountain"

Rohit and Colette want to spend the day together. Each of them can choose to go to either the beach of the mountains. In addition, Rohit might also choose to stay home. If they go to different places, both are unhappy and get utility -3. If they both go to the beach, then Rohit is very happy, while Colette is somewhat happy. If they both go to the mountains, then Colette is very happy, while Rohit is somewhat happy.

Here, the two players have different matrices (which we wrote by comma separation in the table); hence, such general games are called *bimatrix games*.

Naturally, we can extend these ideas to games with more than 2 players. In general, we might have n players, each having an n-dimensional tensor (generalization of a matrix) A_i giving the payoff of the player for any combination of strategies played by the n players. Typically, when the number of players gets large, games are rarely specified in this way — if nothing else, even if each player only has two strategies, each tensor would have 2^n entries. Games with a large number of players typically exhibit a lot of structure, so the payoffs are defined in different ways. Particularly common classes are *graphical games*, in which a player's utility only depends on the actions of her neighbors in a graph, and *anonymous games*, in which a player's payoff only depends on the *number*, but not the *identity* of other players playing each possible strategy. As an example, when you drive somewhere, you mostly care about how many other cars are on the road, but not which specific people are driving on the road with you.

3 Game Outcomes and Equilibria

What would we consider possible "outcomes" of Rock-Paper-Scissors? Let's say that just one round of the game is played. If Rohit has to play first, then whatever action he chooses, Colette will choose an action that makes him lose. Similarly, if Colette goes first, she always loses. Typically, we consider game play as simultaneous, unless otherwise specified. But what would be a suitable definition of the outcome of simultaneous play here? We said earlier that we assume that players are rational, i.e., capable of choosing an action that is best for them. The problem is that what is best for Rohit depends on what Colette is doing, and vice versa. That is, neither really wants to commit to a strategy without knowing what the other is doing. If we were considering a dynamic, they would continuously update their strategy after hearing what the other is doing.

If we were looking for something that can be considered a "final" outcome, it seems reasonable that it should be "stable," in the sense that neither player would want to change their strategy, knowing what the other player is doing. Such outcomes are called *equilibria* of the game.[1] What we just saw is that there is no equilibrium for Rock-Paper-Scissors, because at least one player will always want to change their strategy.

In order to be able to still define equilibria, one workaround is to allow the players to randomize their strategies. That is, Rohit (or Colette) does not have to commit to playing one of Rock, Paper, Scissors, but instead can commit to something like "I will play each with probability $\frac{1}{3}$", or "I will play each of Rock, Paper with probability $\frac{1}{4}$, and Scissors with probability $\frac{1}{2}$." That is, each player's strategy space is now the set of all probability distributions over actions.

We can now define equilibria similarly, except we use the much larger set of distributions as strategy sets. We call these equilibria *mixed equilibria*, and to distinguish equilibria in which players cannot randomize, we call those *pure equilibria*. We saw earlier that Rock-Paper-Scissors does not have any pure equilibria, but it does have a mixed equilibrium: each of Rohit and Colette chooses each strategy with probability $\frac{1}{3}$. Now, we can see that neither of them can do strictly better by changing strategies.

3.1 Existence and Computation of Nash equilibria

The obvious question is now whether *every* game has a mixed equilibrium once we allow randomization. This question was famously answered affirmatively by Nash:

Theorem 1 (Nash [11]) *Every game with a finite number of players and a finite number of strategies per player has a mixed equilibrium.*

In Nash's honor, equilibria of such games are now typically called *Nash equilibria*. Subsequent to Nash's theorem, Theorem 1 has been generalized beyond finite numbers of players and strategies. Generalizations typically involve compact strategy spaces and continuous utility functions; see [6] for a very general version.

Nash's proof is inherently non-constructive, relying on Brouwer's Fixed Point Theorem. Essentially, the argument shows that a sequence of suitable updates must eventually converge to an equilibrium. In particular to computer scientists, this naturally raises the question of whether one can actually *find* a Nash equilibrium efficiently. This question cannot be NP-complete, as the decision problem "Does this game have a Nash equilibrium?" is trivial: the answer is always "Yes." While one can instead ask about the existence of Nash equilibria with certain additional properties, this would deviate from the original question: find *any* Nash equilibrium.

To address this question more formally, Papadimitriou [14] defined complexity classes for problems in which one tries to find an object whose existence is guaranteed by certain non-constructive proofs which can be reduced to an abstract search problem in a graph. Most relevant here is the class PPAD, which contains variants of Brouwer's Fixed Point Theorem, the Nash Equilibrium problem, and several others. Using a suitable notion of reduction, Daskalakis, Goldberg and Papadimitriou and Chen and Deng proved the following:

[1]one equilibrium, two equilibria

Theorem 2 ([3, 2]) *It is PPAD-complete to find a Nash equilibrium*

1. *for n-player games with $n \geq 3$ [3].*
2. *for 2-player games [2].*

These hardness results very much rely on the games not being zero-sum. For zero-sum games, we will see how to compute (mixed) equilibria in Section 5, along the way establishing the existence of equilibria constructively.

3.2 Other game outcomes

There are several other natural notions of what an outcome in a game could be, which we discuss here briefly.

If there is a natural order to the play — for instance, Rohit always has to choose before Colette — then Rohit will optimize his strategy against the known fact that Colette will choose whatever is best for her. This type of outcome is often called *Stackelberg equilibrium.* Usually, when one uses this name, one emphasizes that the first player derives an *advantage* from going first. Notice that this is not the case for Rock-Paper-Scissors, but it is true for Beach or Mountains. If Rohit goes first in that game, he can commit to going to the beach, leaving Colette a choice between going to the mountains (and getting utility -3) or going to the beach (and getting utility 1). Thus, Rohit gets a better outcome by going first. For zero-sum games, the first player's strategy is often referred to as a Mini-Max (or Maxi-Min) strategy, because he is choosing a row that maximizes the payoff he will get, which in the case of a zero-sum game will be the minimum of the entries in that row.

Another important notion is that of *dominant strategies.* A dominant strategy is a very stable outcome: it is a strategy that is best for a player, never mind what the other player does. The existence of a dominant strategy means that a player does not need to figure out what the other player is or might be doing, but can simply pick the "clearly best" strategy and play it. Naturally, dominant strategies are even rarer than pure equilibria, and in fact, neither of the games we defined earlier have dominant strategies for either player.

Another important equilibrium notion is motivated by the *Traffic Intersection* game given in Table 3. Imagine two cars at an intersection. If a driver stops, they will lose a unit of wasted time. If a driver goes through the intersection, they get one unit. But if both drivers drive at the same time, they get into an accident, which is very bad for both.

	Stop	Go
Stop	-1,-1	-1,1
Go	1,-1	-10,-10

Table 3: The payoff matrices A_R, A_C for "Traffic Intersection"

There are two natural pure equilibria: Rohit stops and Colette drives, or Rohit drives and Colette stops. There is a third (mixed) equilibrium, where each of them independently drives with a tiny probability p, and stops with the remaining probability $1 - p \approx 1$. Then, with probability p^2, an accident occurs.

The role of a traffic light is to enable equilibria in which no accidents ever occur, yet we are not stuck in an equilibrium where only Rohit ever gets to drive. A traffic light can be modeled as randomness shared between both players: it allows us to have a light that is either (Red, Green) or (Green, Red), with equal probability. Rohit and Colette can now base their decision on the light as well: one equilibrium would be that whoever has the green light will drive, and whoever has the red light will not drive (or vice versa). This is an equilibrium: if Rohit knows that Colette will always drive when she sees a green light, he will prefer to not drive when he sees a red light. Such coordinated equilibria are called *correlated equilibria.*

Another important observation is that in the standard notion of equilibrium, we consider an outcome stable if no player can unilaterally improve their utility by changing strategies. This does not say that there isn't a group of multiple players who can together change their strategy and do better. Naturally, one can define notions of equilibria with respect to changes by multiple players.

There are additional equilibrium notions, including *coarse correlated equilibria*, outcomes of learning algorithms in games, and others.

3.3 (In)efficiency of equilibria

As we mentioned previously, the choices made by selfish players at equilibrium may not be optimal in their effect on overall utility. For example, one equilibrium of the Traffic Lights game involves randomization, and thus the possibility of accidents. Another example would be if we change the "Beach or Mountains" game slightly, so that both Rohit and Colette prefer the Beach. Now, there is still an equilibrium in which they go to the mountains, because given that the other is going to the mountains, neither would prefer unilaterally going to the beach.

In a sense, we can think of equilibrium solutions as similar to the outcome of a local search algorithm, when no simple improvements are possible any more. Then, by analogy to approximation algorithms, we can ask how *much* worse the equilibrium outcome is as compared to the optimum solution, which gets to just tell everyone what to do.[2]

There are two natural notions that have been studied extensively in the literature, called *Price of Anarchy* [5] and *Price of Stability* [1]. Both study the ratio of the cost/utility of the optimal outcome with that of an equilibrium outcome. The price of anarchy uses the *worst* equilibrium; the motivation for this definition is systems in which one has no control over what players will do, except for knowing that they will reach some equilibrium. The price of stability instead considers the *best* equilibrium; the motivation here is systems in which one can make a "suggestion" to players, but the suggestion is such that selfish players will in fact follow it. Then, it is possible to get them into the best equilibrium.

4 Basic Properties and Equilibria of (Zero-Sum) Games

After this very condensed overview, let us now be a little more formal, and derive some useful properties. We begin with a zero-sum game, in which Rohit's payoff is given by the matrix $A \in \mathbb{R}^{n \times m}$ (while Colette's payoff is given by $-A$). For Rock-Paper-Scissors, we saw that playing first is a disadvantage. We first show that this is true for all zero-sum games. Notice that when Rohit plays first, his payoff is $\max_i \min_j a_{i,j}$, because for any row i that he chooses, Colette will choose the column j giving him the lowest (and thus her the highest) payoff. When Colette plays first, Rohit's payoff is $\min_j \max_i a_{i,j}$.

Proposition 3 *For all matrices $A = (a_{i,j})_{i,j}$, we have* $\max_i \min_j a_{i,j} \leq \min_j \max_i a_{i,j}$.

Proof. Let j^* be the strategy that minimizes $\max_i a_{i,j}$, i.e., Colette's best strategy when she plays first. Then, $\max_i \min_j a_{i,j} \leq \max_i a_{i,j^*}$: Rohit can only do better if Colette always plays j^* instead of reacting optimally to his choice. So $\max_i \min_j a_{i,j} \leq \max_i a_{i,j^*} = \min_j \max_i a_{i,j}$. ■

Next, let's look a little more closely at the need for randomization. We saw earlier that in general, even zero-sum games do not have pure equilibria, but they do have mixed equilibria. Suppose that there is an order to the game: Rohit moves first and commits to a randomized strategy. Does Colette still need to randomize to respond optimally? For Rock-Paper-Scissors, that's not true: once Rohit commits to playing each strategy with probability $\frac{1}{3}$, all actions are equally good for Colette, so she can just play a pure strategy. (The reason she needs to randomize at equilibrium is so that Rohit cannot improve his strategy by doing something different — but if Rohit has to commit first, this is not a concern.) The next lemma shows that this is a general property of bimatrix games (not just zero-sum games). Here and in the following, we will identify mixed strategies with probability vectors whose entries are non-negative and add up to 1.

Proposition 4 (Loomis's Theorem) *Let $\boldsymbol{r}, \boldsymbol{c}$ be equilibrium randomized strategies for Rohit and Colette in the bimatrix game (A_R, A_C).*

[2]For instance, it may sacrifice the utility of some players for the common good, or keep each player from being too selfish and ruining the outcome for everyone else.

1. *If $r_i > 0$ and $r_{i'} > 0$ (i.e., Rohit plays both i and i' with non-zero probability), then Rohit's utility from playing i and i' is the same, i.e., $(A_R\boldsymbol{c})_i = (A_R\boldsymbol{c})_{i'}$. (Similarly for Colette.)*
2. *The second player to play has an optimal response that is a pure strategy.*

Proof. To prove the first part, notice that if one of $(A_R\boldsymbol{c})_i, (A_R\boldsymbol{c})_{i'}$ were larger, then Rohit could unilaterally improve his utility by putting more probability weight on that one, and less on the other.

For the second part, because all strategies that Rohit randomizes between at equilibrium give him the same payoff, he can obtain the same payoff by just playing any one of them. ■

5 Computing Equilibria

We can leverage Proposition 4 to phrase the problem of computing an optimal first-mover strategy for Rohit as a linear program. There is a variable r_i for each pure strategy i, namely, the probability that Rohit plays i. In addition, in order to express the objective, we have one more variable u, which is Rohit's expected utility. Beyond the constraints ensuring that the r_i are a probability distribution, we need to express that Colette will choose the response that makes Rohit as badly off as possible (because the game is zero-sum). By Proposition 4, we only need to consider each of Colette's pure strategies. Rohit will thus get the minimum of his utilities, over all strategies j that Colette could play. We express this by saying that u is upper-bounded by the utility for each of Colette's responses j. Hence, our LP is the following:

$$\begin{array}{lll} \text{Maximize} & u & \\ \text{subject to} & u \leq \sum_i a_{i,j} r_i & \text{for all } j \\ & \sum_i r_i = 1 & \\ & r_i \geq 0 & \text{for all } i. \end{array}$$

Taking the dual of this LP, we see that that it is:

$$\begin{array}{lll} \text{Minimize} & v & \\ \text{subject to} & v \geq \sum_j a_{i,j} c_j & \text{for all } i \\ & \sum_j c_j = 1 & \\ & c_j \geq 0 & \text{for all } j. \end{array}$$

That is, the dual of Rohit's first-mover optimization problem is Colette's first-mover optimization problem, of choosing a mixed strategy to minimize Rohit's payoff against his best response. These LPs imply a few things: first, we can efficiently compute optimal first-mover strategies for both players. Second, the LP solutions $\boldsymbol{r}$ and $\boldsymbol{c}$ actually form an equilibrium. This follows from complementary slackness of linear programming, which says that whenever a variable (r_i or c_j) is strictly positive, then the corresponding constraint in the other LP must be tight. Thus, whenever Colette includes a strategy j in her randomization, it must be a best response to Rohit's mixed strategy, and vice versa. Finally, the strong duality theorem implies the following:

Theorem 5 (Von Neumann's Minimax Theorem) *$\max_{\boldsymbol{r}} \min_{\boldsymbol{c}} \boldsymbol{r}^\mathsf{T} A \boldsymbol{c} = \min_{\boldsymbol{c}} \max_{\boldsymbol{r}} \boldsymbol{r}^\mathsf{T} A \boldsymbol{c}$.*

Thus, when both players play randomized strategies, the value of the game is the same regardless of which player commits first. Of course, combining Theorem 5 with Proposition 4, it also follows that $\max_{\boldsymbol{r}} \min_j (\boldsymbol{r}^\mathsf{T} A)_j = \min_{\boldsymbol{c}} \max_i (A\boldsymbol{c})_i$. And by weak duality alone, we get that for every mixed strategies $\boldsymbol{r}, \boldsymbol{c}$:

$$\min_j (\boldsymbol{r}^\mathsf{T} A)_j \leq \max_i (A\boldsymbol{c})_i. \tag{1}$$

6 Yao's Minimax Theorem

For the purpose of algorithm analysis, Theorem 5 and the weaker version (Inequality (1)) allow us to prove a non-trivial result that helps prove lower bounds on the performance of randomized algorithms. We can think about the choice of algorithms and inputs as a game between two players, one choosing a deterministic algorithm (we'll call him Alan from now on) and one choosing an input for the algorithm (we'll call her Indira). There is some measure of cost of the algorithm: most often, this will be the algorithm's running time, but it may also be memory usage, or an approximation guarantee, or some other measure of how well or poorly the algorithm is performing. When a deterministic algorithm A runs on an input I, it incurs some cost $C(A, I)$.

If Alan has to pick his deterministic algorithm first, then he is effectively the first mover in a zero-sum game. After he chooses A, Indira will choose an input I that makes the cost as large as possible. Conversely, if Indira had to commit to an input I first, Alan's job would be really easy: he would just compute the answer to I, and use an algorithm that has the correct answer hardwired, outputting it in constant time.

In algorithm analysis, we think of the algorithm A being specified first, and the input being a worst-case input for A. Since Alan needs to commit first, he can vastly improve his performance by choosing a randomized algorithm. A randomized algorithm can be considered as a distribution over deterministic algorithms.

To make this more precise, we fix an input size n. As a result, there will be only finitely many inputs of size n.[3] We also restrict attention to algorithms that *always* (not just in expectation, or with high probability) run in some bounded time $f(n)$ (exponential or worse is fine). Then, there are only finitely many possible algorithms (if we identify an algorithm with the way in which it accesses and processes memory). Furthermore, in $f(n)$ steps, an algorithm can at most flip $f(n)$ coins, i.e., the number of coin flips by the algorithm is also bounded.

Therefore, we can consider a randomized algorithm as a distribution over deterministic algorithms. If the randomized algorithm flips some maximum number $f(n)$ of coins, then we can perform all those coin flips before the algorithm even starts running and write them down somewhere; the algorithm will consult them whenever a coin flip is called for. But now, the algorithm itself is deterministic, and the coins simply choose which deterministic algorithm is being run. Thus, a randomized strategy for Alan is exactly a randomized algorithm.

Applied specifically to this game, Theorem 5 states that the cost of the best randomized algorithm on its worst-case input is equal to the cost for the following scenario: first, Indira chooses a worst possible input distribution, then Alan picks the best deterministic algorithm, knowing Indira's input distribution. This input allows us to prove lower bounds on the cost of the best randomized algorithm, without having to explicitly figure out what they are or constructing bad inputs for them. In fact, in order to just prove a lower bound, Indira does not even need to use a worst possible input distribution; any input distribution will give a lower bound, although if the distribution is really easy to handle, the lower bound will not be very strong. From Inequality (1), we thus obtain the following theorem:

Theorem 6 (Yao's MiniMax Theorem) *Fix a problem Π and an input size n. Let $\mathcal{I}$ be the (finite) set of all inputs of size n, and $\mathcal{A}$ the (finite) set of all algorithms Π on inputs of size n under consideration. Let $\boldsymbol{q}$ be any distribution over $\mathcal{I}$, and $A_{\boldsymbol{q}}$ the optimal deterministic algorithm for the input distribution $\boldsymbol{q}$. Let $\boldsymbol{p}$ be any distribution over deterministic algorithms, i.e., any randomized algorithm, and let $I_{\boldsymbol{p}}$ be the worst-case input for the randomized algorithm $\boldsymbol{p}$. Then,*

$$\mathbb{E}_{I \sim \boldsymbol{q}}\left[C(A_{\boldsymbol{q}}, I)\right] \leq \mathbb{E}_{A \sim \boldsymbol{p}}\left[C(A, I_{\boldsymbol{p}})\right].$$

In words, $\mathbb{E}_{I \sim \boldsymbol{q}}\left[C(A_{\boldsymbol{q}}, I)\right]$ is a lower bound on the worst-case expected cost of any randomized algorithm for Π.

[3]We actually count the number of bits in the input. For instance, this rules out a model in which inputs may contain real numbers of arbitrary precision.

Theorem 6 reduces the often difficult task of designing bad inputs for randomized algorithms to the often more manageable task of designing bad distributions over inputs for deterministic algorithms. Of course, the bounds one derives are only as strong as the input distribution allows: if the distribution is simple (e.g., Indira plays a pure strategy, or randomizes only over very few inputs), then Alan's task is very easy, and the deterministic algorithm will perform very well. This does not show that a randomized algorithm can solve Π well — it only implies that the distribution may have been too easy.

7 An Application: Game Tree Evaluation

As an application of Yao's MiniMax Theorem, we will consider an algorithmic problem that also lets us discuss another type of game model. In many types of games (in particular: games of strategy such as chess, Tic Tac Toe, etc.), two players alternate taking turns. Various states correspond to the end of the game, and at that point, we know the "value" or outcome of the game. Games that are specified in this way are called *extensive form games*, and they extend beyond "game" examples to many situations in which competitors take turns making decisions. Here, we will only consider games *with full information*, i.e., where both players have full information about all relevant parameters and each other — this rules out games such as poker, in which players have access to information (their own cards) not available to the other player.

When the game is zero-sum, as before, one player (Max) wants to maximize the outcome, while the other (Mina) wants to minimize it. Whenever it is Max's turn, he will choose an action that will maximize his outcome from all options, knowing full well that Mina will choose an action minimizing it among available options (and she knows that Max will choose an action maximizing his outcome, etc.) Thus, we can think of the game as a tree, in which the nodes at alternating levels are labeled "Max" and "Min". Leaves of the tree have specified values, and the values of internal nodes can be defined inductively: the value of a Max node is the maximum of the values of its children, while the value of a Min node is the minimum of the values of its children. An important algorithmic question is then to calculate the value of the root of the tree, which corresponds to finding out who will win the game (and with what score) if both players play perfectly.

An important special case is obtained when all leaves have values of 0 or 1. Then, a Max node computes an OR of the values of its subtrees, while a Min node computes an AND. Such trees model decision rules: you might choose to buy stock in a particular company if the general economic outlook is good, and (the company has a quarterly announcement coming up or the sun is shining today). Then, in order to evaluate whether to buy stock, you need to calculate the value of the AND/OR tree, i.e., find out whether it is 0 or 1.

Of course, this is an easy algorithmic task: you can either solve it recursively using DFS or bottom-up with dynamic programming. Either way, the time it takes will be $\Theta(n)$, the number of nodes in the tree. That is also the number of leaves that need to be evaluated. Often, we think of the evaluation of leaves as costly: at least, it takes effort to do so, and possibly, it costs actual money to get access to relevant economic data. Hence, we would like an algorithm that queries fewer than n leaves.

When the algorithm has to be deterministic, it is not difficult to see that one cannot evaluate the tree with fewer than n leaf queries in the worst case. While the process of querying leaves is adaptive, for a deterministic algorithm, an adversary can predict which will be the next leaf queries, and thus design the whole input ahead of time to achieve exactly the same behavior. One can prove pretty easily by induction on the height of the tree:

Proposition 7 *For any AND/OR tree T of height h and any deterministic evaluation algorithm A, there are value assignments to the leaves of T that make T evaluate to true/false, and for which A must query all leaves before knowing the value.*

Hence, we turn to randomization to do better.

7.1 Snir's Algorithm

We will derive and analyze a randomized algorithm for evaluation of *binary* AND/OR trees. To motivate the randomized algorithm, consider an AND node. If both of its subtrees evaluate to 1, then the only way to obtain the value of the node is to evaluate both subtrees. But if both evaluate to 0, then an algorithm definitely will only need to evaluate one subtree to get the value. And if exactly one subtree evaluates to 0, then the algorithm can save itself the second subtree if it evaluates that one first. The problem is that for a deterministic algorithm, an adversary can always ensure that the subtree the algorithm evaluates first evaluates to 1. That's exactly where randomization helps.

Similarly, for an OR node, if both subtrees evaluate to 1, the algorithm will only need to evaluate one of them, and if exactly one evaluates to 1, the algorithm has a chance to only evaluate one subtree. Because AND and OR nodes alternate, an input that's bad for an AND node (both subtrees evaluate to 1) means that at the next level, the input cannot be of the worst type (both subtrees evaluate to 0) for the OR nodes. This suggestions the following algorithm, known as Snir's Algorithm (or Random DFS):

Algorithm 1 Snir's Algorithm for AND/OR tree evaluation

```
1: if the current node $v$ is a leaf then
2:     Query it and return the value.
3: else
4:     Pick one of the two children $v_1, v_2$ of $v$ uniformly at random.
5:     Recursively evaluate the picked child $v_i$.
6:     if it determines the value of $v$ then
7:         Return that value.
8:     else
9:         Recursively evaluate the other child $v_{3-i}$.
10:        Return the value of $v$.
```

To analyze the running time, let $T^{0/1}_{\text{AND/OR}}(h)$ be the time to evaluate a tree of height h whose root is an AND/OR node, when it evaluates to 0/1. For an AND node to evaluate to 1, or an OR node to evaluate to 0, we always need to evaluate both subtrees, so all we can guarantee is that

$$\begin{aligned}\mathbb{E}\left[T^1_{\text{AND}}(h)\right] &\le 2\mathbb{E}\left[T^1_{\text{OR}}(h-1)\right],\\ \mathbb{E}\left[T^0_{\text{OR}}(h)\right] &\le 2\mathbb{E}\left[T^0_{\text{AND}}(h-1)\right].\end{aligned}$$

Now consider an AND node that evaluates to 0. If both of its subtrees evaluate to 0, then the expected cost will be the expected cost to evaluate one OR subtree to 0. If one of its subtrees evaluates to 0 and the other to 1, then to determine the value, the algorithm will need to evaluate one subtree to 0; with probability $\frac{1}{2}$, it first evaluates the wrong subtree, which evaluates to 1. The case for an OR node that evaluates to 1 is analogous. Therefore,

$$\begin{aligned}\mathbb{E}\left[T^0_{\text{AND}}(h)\right] &\le \mathbb{E}\left[T^0_{\text{OR}}(h-1)\right] + \frac{1}{2}\mathbb{E}\left[T^1_{\text{OR}}(h-1)\right],\\ \mathbb{E}\left[T^1_{\text{OR}}(h)\right] &\le \mathbb{E}\left[T^1_{\text{AND}}(h-1)\right] + \frac{1}{2}\mathbb{E}\left[T^0_{\text{AND}}(h-1)\right].\end{aligned}$$

Let $t_{\text{AND}}(h) = \max(\mathbb{E}\left[T^0_{\text{AND}}(h)\right], \mathbb{E}\left[T^1_{\text{AND}}(h)\right])$, and $t_{\text{OR}}(h) = \max(\mathbb{E}\left[T^0_{\text{OR}}(h)\right], \mathbb{E}\left[T^1_{\text{OR}}(h)\right])$. Then, by combining two consecutive levels, we obtain that

$$\begin{aligned}
\mathbb{E}\left[T^1_{\text{AND}}(h)\right] &\le 2\mathbb{E}\left[T^1_{\text{AND}}(h-2)\right] + \mathbb{E}\left[T^0_{\text{AND}}(h-2)\right] \ \le\ 3t_{\text{AND}}(h-2),\\
\mathbb{E}\left[T^0_{\text{OR}}(h)\right] &\le 2\mathbb{E}\left[T^0_{\text{OR}}(h-2)\right] + \mathbb{E}\left[T^1_{\text{OR}}(h-2)\right] \ \le\ 3t_{\text{OR}}(h-2),\\
\mathbb{E}\left[T^0_{\text{AND}}(h)\right] &\le 2\mathbb{E}\left[T^0_{\text{AND}}(h-2)\right] + \frac{1}{2}\mathbb{E}\left[T^1_{\text{AND}}(h-2)\right] + \frac{1}{4}\mathbb{E}\left[T^0_{\text{AND}}(h-2)\right] \ \le\ \frac{11}{4}t_{\text{AND}}(h-2),\\
\mathbb{E}\left[T^1_{\text{OR}}(h)\right] &\le 2\mathbb{E}\left[T^1_{\text{OR}}(h-2)\right] + \frac{1}{2}\mathbb{E}\left[T^0_{\text{OR}}(h-2)\right] + \frac{1}{4}\mathbb{E}\left[T^1_{\text{OR}}(h-2)\right] \ \le\ \frac{11}{4}t_{\text{OR}}(h-2).
\end{aligned}$$

Thus, by taking the maximum of the two cases, we see that

$$\begin{aligned}
t_{\text{AND}}(h) &\le 3t_{\text{AND}}(h-2),\\
t_{\text{OR}}(h) &\le 3t_{\text{OR}}(h-2).
\end{aligned}$$

Unrolling the recurrence, we get that the expected number of leaves queried to evaluate a tree of h levels is $3^{h/2}$. Because a complete binary tree with n leaves has $h = \log_2 n$ levels, the number of leaves queried by Snir's algorithm is $3^{\frac{1}{2}\log_2 n} = n^{\frac{1}{2}\log_2 3} \le n^{0.793}$. In particular, the speedup is from linear to significantly sublinear.

7.2 A Lower Bound on Randomized Algorithms

Next, we want to investigate whether $n^{0.793}$ leaf queries is the best that any randomized algorithm can do. So we'd like to prove a lower bound on the number of leaf queries of any randomized algorithm. To do so, we will employ Yao's Theorem (Theorem 6). That is, we need to define a distribution over inputs (true/false assignments) such that the best deterministic algorithm, knowing the distribution, will still need to query a lot of leaves.

Perhaps the most natural distribution to try is to have each leaf be true independently with probability $\frac{1}{2}$. There's a slight problem with this distribution, though. Let's say that the bottom layer consists of OR nodes. Each of them is now true with probability $\frac{3}{4}$. So each node in the next layer of AND nodes is true with probability $\frac{9}{16}$. It would be cumbersome for the analysis to deal with the fact that at each layer, the probabilities of being true are different — we'd like the output of the AND nodes to be true with the same probability as the leaves. This will necessitate a slightly different probability of being true for each leaf.

In fact, to simplify our analysis a little further, we can perform a transformation on the tree. A simple truth table calculation shows that if the tree has even height and the root node is an AND, then the root's value is the same as in a tree in which all nodes have been replaced with NOR gates. So we will switch our analysis to a tree in which all gates are NOR.

Now, we can look at just one level rather than two to reverse engineer the right value p for leaves to be true. The probability that the NOR gate is true when each input is true independently with probability p is $(1-p)^2$, because a NOR gate is true exactly when both inputs are false. We would like this to be equal to p, so our desired p solves $(1-p)^2 = p$, or $p^2 - 3p + 1 = 0$. The solution to this quadratic equation is $p = \frac{3-\sqrt{5}}{2}$, which is our choice of probability.

Against this distribution, the optimum deterministic algorithm is to query the leaves left-to-right, skipping any leaf whose value is not needed any more. This intuitively makes sense: if one has already begun evaluating a subtree, it is always better to continue evaluating it until its value is known, rather than switching to a different subtree. Having evaluated some leaf can never bring "bad news" that would make it seem much more expensive to fully evaluate the tree (and thus preferable to switch). While this is intuitive, a formal proof requires some care, and induction over the height of the tree. We'll skip it here, and instead focus on how many leaves this deterministic algorithm needs to query.

We can analyze this number using a similar recurrence as for Snir's algorithm. When querying a NOR node at level h, we first evaluate one subtree fully. If it evaluates to 1, which happens with probability

$p = \frac{3-\sqrt{5}}{2}$, then the value of the NOR node is known. Otherwise, the other subtree must be evaluated as well. Thus, if $s(h)$ is the expected time to evaluate a tree of height h, then it satisfies the recurrence

$$s(h) = s(h-1) + (1-p)s(h-1) \;=\; \frac{1+\sqrt{5}}{2} s(h-1).$$

So $s(h) = \left(\frac{1+\sqrt{5}}{2}\right)^h$, and substituting that $h = \log_2 n$, we get that the number of leaves which the deterministic algorithm evaluates is $\left(\frac{1+\sqrt{5}}{2}\right)^{\log_2 n} = n^{\log_2(1+\sqrt{5})-1} \geq n^{0.694}$. Using Yao's Theorem, we have thus shown that every randomized algorithm must in the worst case evaluate at least $n^{0.694}$ leaves.

The attentive reader will have noticed that $0.694 \neq 0.793$; that is, there is a gap between the lower and upper bounds we proved. So either Snir's algorithm is not optimal, or our distribution is not the worst-case input distribution.

It turns out that the latter is the case. Among distributions with independent choices for leaves, our value of p is indeed the worst-case choice. But the choice to make the leaves independent is suboptimal. When the leaves (or larger subtrees) are true independently, there is a non-trivial chance that both are true. Then, an algorithm cannot go wrong: whichever subtree it evaluates first, it will know the value of the NOR gate without ever having to query the other subtree. To produce a distribution matching the upper bound from Snir's algorithm, we need to anti-correlate the values of leaves and subtrees. Starting from the root, we make a random choice whether it will be true or false. Then, for any NOR node (including the root), if we have decided that it will be true, then both subtrees must be false. But if we have decided that it will be false, we will make exactly one subtree true and one false, and choose randomly which one is which. Analyzing the optimal deterministic algorithm against this distribution is much harder, because now, learning about some leaf reveals information about other leaves as well. But with enough work, one can show that the best deterministic algorithm must query $n^{\frac{1}{2}\log_2 3}$ leaves.

8 Further Reading

Game Theory is a classic subject in economics, and there are several good introductory books on the topic from an economist's perspective, e.g., [10, 13]. For an even broader (and very comprehensive) introduction to micro-economic theory, see the textbook [7].

For a collection of surveys on different areas of algorithmic game theory, written from a CS perspective, see [12]. For a CS-oriented introduction to algorithmic game theory, mechanism design, and related topics, see [15, 4].

A more in-depth discussion of Yao's MiniMax Theorem and its applications is part of any standard textbook on randomized algorithms, such as [9, 8].

References

[1] Elliot Anshelevich, Anirban Dasgupta, Jon Kleinberg, Eva Tardos, Thomas Wexler, and Tim Roughgarden. The price of stability for network design with fair cost allocation. In *Proc. 45th IEEE Symp. on Foundations of Computer Science*, pages 295–304, 2004.

[2] Xi Chen and Xiaotie Deng. Settling the complexity of two-player nash-equilibrium. In *Proc. 47th IEEE Symp. on Foundations of Computer Science*, pages 261–270, 2006.

[3] Constantinos Daskalakis, Paul Goldberg, and Christos H. Papadimitriou. The complexity of computing a nash equilibrium. In *Proc. 38th ACM Symp. on Theory of Computing*, pages 71–78, 2006.

[4] Jason D. Hartline. *Mechanism Design and Approximation.* available at http://jasonhartline.com/MDnA/, 2011–2017.

[5] Elias Koutsoupias and Christos H. Papadimitriou. Worst-case equilibria. In *Proc. 16th Annual Symp. on Theoretical Aspects of Computer Science*, pages 404–413. Springer, 1999.

[6] Andreu Mas-Colell. On a theorem of Schmeidler. *J. of Mathematical Economics*, 13:201–206, 1984.

[7] Andreu Mas-Collel, Michael D. Whinston, and Jerry R. Green. *Microeconomic Theory.* Oxford University Press, 1995.

[8] Michael Mitzenmacher and Eli Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis.* Cambridge University Press, 2005.

[9] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms.* Cambridge University Press, 1990.

[10] Roger B. Myerson. *Game Theory: Analysis of Conflict.* Harvard University Press, 1997.

[11] John F. Nash. Equilibrium points in n-person games. *Proc. Natl. Acad. Sci. USA*, 36:48–49, 1950.

[12] Noam Nisan, Tim Roughgarden, Eva Tardos, and Vijay Vazirani, editors. *Algorithmic Game Theory.* Cambridge University Press, 2007.

[13] Martin J. Osborne and Ariel Rubinstein. *A Course in Game Theory.* MIT Press, 1994.

[14] Christos H. Papadimitriou. On the complexity of the parity argument and other inefficient proofs of existence. *Journal of Computer and System Sciences*, 48(3):498–532, 1994.

[15] Tim Roughgarden. *Twenty Lectures on Algorithmic Game Theory.* Cambridge University Press, 2016.