Topics on the Border of Economics and Computation November 13, 2005

Lecture 3

Lecturer: Noam Nisan *Scribe: Maor Ben-Dayan*

1 Introduction

In today lecture we will continue to explore the use of randomized algorithms in zero sum games, last week we started the analysis of binary game trees. We've seen that evaluating a game tree with 4 leaves becomes easier using a randomized algorithm compared to deterministic algorithms. The evaluation is easier when considering the hardest game tree for the given algorithm. When given a deterministic algorithm used by a player, the tree designer could plan a tree that would force the player to check all 4 leaves, but given a randomized algorithm, the tree designer cannot devise a tree that would require the algorithm more than 3 checks of the leaves (in the expected sense) in order to evaluate the game.

Theorem 1 *the expected number of leaves needed by the randomized algorithm (shown last week) to fully evaluate an alternating binary game tree with n leaves is $n^{log_4 3}$.*

Proof: Let h denote the number of moves each of the 2 players does during the game then $n = 2^{2h}$, since the height of the tree is $2h$.

We will prove the theorem by induction over h. We will denote the expected number of leaves the algorithm check for a tree of height $2h$ by $T(h)$.

For $h = 0$ the tree is composed of a single leaf so the algorithm only have to look at this single leaf and we have $T(0) = 1$.

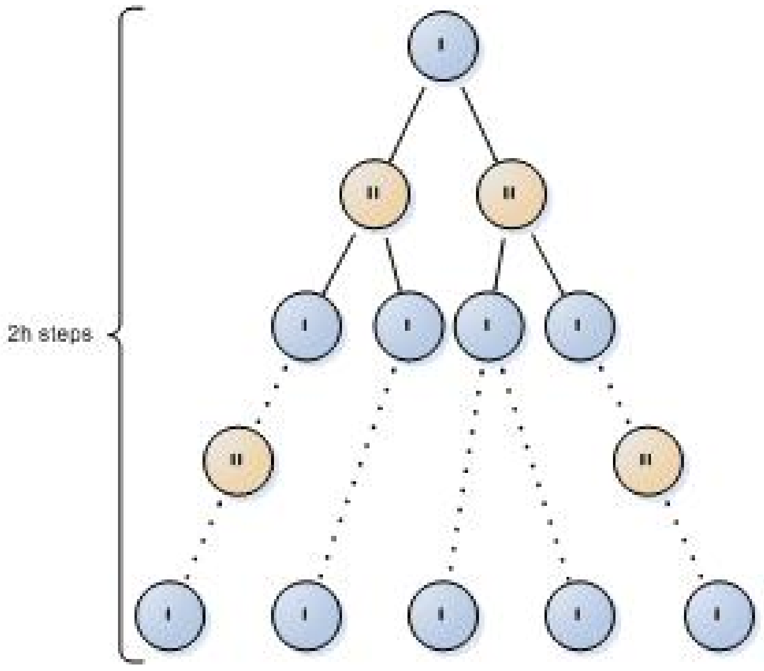

Figure 1: Alternating game tree of height $2h$

in Figure 1 we can see a tree of height $2h$. By the induction assumption we need to look at an expected $T(h-1)$ leaves for each of the 4 subtrees of height $2(h-1)$ we get (according to the theorem we proved last week) $T(h) \leq 3T(h-1)$, hence $T(h) \leq 3^h$ and since $n = 4^h$ we get $T(h) = n^{log_4 3}$

■

Remark In the proof above we didn't use information that can be gained about the tree while scanning the 4 subtrees. by using that kind of information we can gain a tight lower bound on the number of leaves we need to look at. This bound is approximately $n^{0.74}$ as opposed to $n^{log_4 3} \approx n^{0.79}$ we've shown.

Open Problem 1 *The above algorithm is the most efficient algorithm there is for evaluating game trees.*

We'll now look at a non alternating game tree of height $2h$. The game which it describes is the following:
Player 1 has h consecutive moves, in each move he can make one of 2 possible choices. Once player 1 is done, player 2 also gets a series of h consecutive moves. Figure 2 is a diagram of such a game tree.

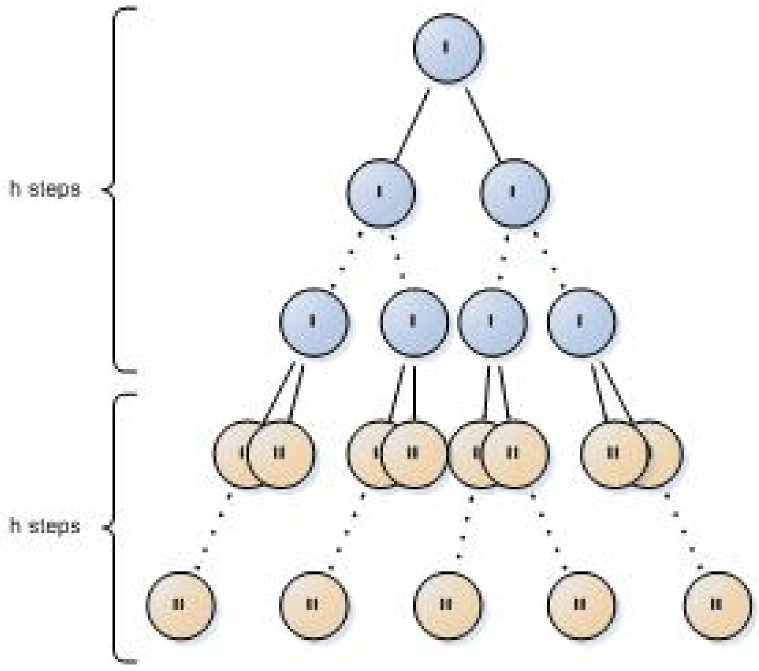

Figure 2: Non alternating game tree of height $2h$

It's possible to look at the non-alternating game as the following matrix:

$$\begin{array}{cc}
 & \overbrace{\hphantom{a_{1,1} \quad \cdots \quad a_{1,2^h}}}^{\text{Player B moves}} \\
\text{Player A moves} & \left[\begin{array}{ccc}
a_{1,1} & \cdots & a_{1,2^h} \\
\cdots & \cdots & \cdots \\
\cdots & \cdots & \cdots \\
\cdots & \cdots & \cdots \\
a_{2^h,1} & \cdots & a_{2^h,2^h}
\end{array}\right]
\end{array}$$

Each of the 2^h rows in the matrix describes one possible move made by player A and each of the 2^h columns describes one of the possible series of moves made by player B. each entry in the matrix contain the reward of player A from the game.

What is the purpose of A in the described game ? Since we're dealing with a binary game player A must find a row filled with ones. How hard can it be ?

Remember, our analysis tries to find the hardest matrix for the algorithm given the best randomized algorithm. In other words, We are playing a game in which we will pick a randomized algorithm and an evil adversary will provide the hardest possible matrix for our algorithm. We would like to pick an algorithm for which the provided matrix will be the easiest (will require the exploration of the least expected number of cells to provide player A's move).

Let's look at a matrix randomly filled with $\{0, 1\}$ (each cell has been drawn i.i.d with $p(1) = 0.5$). this matrix has a very small probability $(= 2^h(\frac{1}{2})^{2^h})$ of containing a row full of ones. A randomized algorithm trying to locate a row of ones in the matrix will have to (with probability very close to 1) check all n rows (since, with high probability, he won't find an all ones row). In order to establish that a certain row is not all 1s it will have to check an expected 2 cells (each cell in the row has probability of $\frac{1}{2}$ to be 1 or 0). To summarize, given a random binary matrix, the expected number of cells any randomized algorithm will have to check is $\approx 2^h \cdot 2$ which is much less than the number of cells in the matrix $(2^h \cdot 2^h)$.

The above matrix is not, however, the hardest input matrix, we will now prove that randomized algorithms cannot do much better than a deterministic one in the non-alternating game.

Theorem 2 *Any randomized algorithm operating on a $\vee_{2^h}\wedge_{2^h}$ tree (non-alternating game tree of $n = 4^h$ leaves) should check, at least, an expected $\frac{n}{2}$ leaves.*

Proof: Let A_{rand} be the set of all randomized algorithms for solving our problem, M the set of all $2^h \times 2^h$ binary input matrices (remember, 1 means player A won, and 0 means player B won) and $T(a)$ the distribution of all possible series of "coin tosses" of algorithm $a \in A_{rand}$. Using these, we would like to show that $\min\limits_{a\in A_{rand}} \max\limits_{m\in M} \underset{\sim T(a)}{Exp}(\text{ cells checked }) \geq \frac{n}{2}$.

By using Yao's theorem we have $\min\limits_{a\in A_{rand}} \max\limits_{m\in M} \underset{t\sim T(a)}{Exp}(\text{ cells in } M \text{ checked by } a\ ) \geq$ $\max\limits_{D} \min\limits_{a\in A_{det}} \underset{\sim D}{Exp}(\text{ cells checked })$ where D is some distribution over M. All we have to do is show that $\max\limits_{D} \min\limits_{a\in A_{det}} \underset{\sim D}{Exp}(\text{ cells checked }) \geq \frac{n}{2}$.

Note: we made our task easier by restricting ourselves to deterministic algorithms but made our lives "harder" by having to analyze the expected behavior on a hard distribution of input matrices rather then the worst case behavior on a single hard matrix.

Let us look at a few distributions of input matrices and find a lower bound for the number of cells a deterministic algo. will have to check.

Distribution No. 1

let $m \sim Uniform(M_1)$ where $M_1 \subset M$ is the set of all the input matrices with a single row of 1s and the rest of the matrix filled with 0s. Our algorithm have to find the 1s row and verify that it is, in fact, filled with 1s (the algorithm does not know anything about the distribution).

Every algorithm which is looking for a row of 1s in a matrix will have to look at an expected $\frac{2^h}{2}$ rows until reaching the all 1s row. For each 0 filled row, the algorithm has to check a single cell, for the last row the algorithm will have to look at all the 2^h cells in order to verify that the row is indeed filled with 1s. All in all the expected number of cells a deterministic algorithm will have to check is $\geq 2^h + \frac{2^h}{2}$.

Distribution No. 2

let $m \sim Uniform(M_2)$ where $M_2 \subset M$ is the set of all the input matrices with a single row of 1s and the rest of the cells filled, half with 0s and half with 1s.

As before, the algorithm will have to look at an expected $\frac{2^h}{2}$ rows until reaching the all 1s row and for the last row the algorithm will have to look at all the 2^h cells. For each of the other rows checked, the algorithm has to check an expected 2 cells until finding a 0. All in all the expected number of cells a deterministic algorithm will have to check is $\geq 2^h + 2\frac{2^h}{2} = 2(2^h) = 2\sqrt{n}$.

Distribution No. 3

let $m \sim Uniform(M_3)$ where $M_3 \subset M$ is the set of all the input matrices with one cells in each row containing 0 and the other cells containing 1. For every matrix drawn from M_3 the algorithm has to output 0 (player B can ensure a victory), however, any deterministic algorithm which always gives the correct answer (for matrices in and out of the distribution support) will have to check all the rows (and find all the 0s in the matrix) before answering.

In order to find a '0' in a specific row, the algorithm will have to check an expected $\frac{2^h}{2} = \frac{\sqrt{n}}{2}$. Since all the rows are i.i.d knowledge of the other rows does not help the algorithm in performing on any row in the matrix, therefore, any deterministic algorithm will need to check an expected $\sum_{2^h rows} \frac{2^h}{2} = 2^h \frac{2^h}{2} = \frac{n}{2}$ cells.

Proof conclusion

If we choose $D = M_3$ we get that every deterministic algorithm, and specifically the best deterministic algorithm will have to check an expected number of cells which is greater than $\frac{n}{2}$ which is what we wanted to show.

■

2 Nash equilibrium

Before formally defining Nash equilibrium we will first examine few games and analyze them. As always we will denote the players $1 \dots n$, player i will have a set of pure strategies S_i and a utility function $u_i : S_1 \times S_2 \times \dots \times S_n \to \mathbb{R}$.

2.1 Prisoner's Dilemma

In the prisoner's dilemma game, there are two players. Each has two choices, namely cooperate or defect. Each must make the choice without knowing what the other will do. The player's payoffs are shown in the following matrix:

column player (II)

		cooperate	defect
rows player (I)	cooperate	-1,-1	-10,0
	defect	0,-10	-9,-9

The first number in each cell is the reward of player 1 and the second is the reward of player 2 if the game ends in that cell. Using the above notations we can describe the prisoners dilemma as a 2 players game where: $S_1 = S_2 = \{cooperate, defect\}$ and, for example, $u_1(cooperate, defect) = -10$

What should player 1 do, cooperate or defect ? It is easy to see that no matter what the player 2 does it is better for player 1 to defect since his utility will be greater. The same holds for player 2. In this case we say that the defection strategy dominates the cooperation strategy for both players. To put it in a formal way, we have: **Notation** For u_i (the ith player's utility function) the following are different notations of the same: $u_i(\vec{x}) \leftrightarrow u_i(x_1, \dots, x_n) \leftrightarrow u_i(x_i, x_{-i}) \quad x_{-i} = (x_1, \dots, x_{i-1}, \quad , x_{i+1}, \dots, x_n)$ In other words x_{-i} denotes the strategies of all the player other than player i.

Definition 3 *Dominating strategy*

We say that strategy x_i strongly *dominates strategy y_i if: $\forall x_{-i} \;\; u_i(x_i, x_{-i}) > u_i(y_i, x_{-i})$.*

We say that strategy x_i weakly *dominates strategy y_i if: $\forall x_{-i} \;\; u_i(x_i, x_{-i}) \geq u_i(y_i, x_{-i})$ and $\exists x_{-i}$ such that $u_i(x_i, x_{-i}) > u_i(y_i, x_{-i})$*

2.2 Battle of the sexes

Another 2 players game in which a loving couple wants to go to the movies together, Surely they want to go together, but alas, as it sometimes happens, the boy wants to see the latest action film and the girl wants to watch a romantic one. The couple's utility function can be described by the following matrix:

Boy

		action	romance
Girl	action	1,2	0,0
	romance	0,0	2,1

In this game there is no dominating strategy for any of the players (check !). However we can clearly see that the 2 players would like to cooperate in order to maximize their utilities.

Definition 4 *Best reaction*

x_i will be called "best reaction to x_{-i}" if $\forall y_i \ \ u_i(x_i, x_{-i}) \geq u_i(y_i, x_{-i})$

Definition 5 *Nash equilibrium*

$x_1, \ldots, X_n$ will be said to be in a Nash equilibrium if for every $i \in \{1, \ldots, n\}$ x_i is a best reaction to x_i.

In other words, $x_1, \ldots, X_n$ are in a Nash equilibrium if, given that for all $1 \geq i \geq n$ player i plays x_i, no player wants to change their strategy.

We can look at a Nash equilibrium as a socially stable convention, that is, If everybody is acting in a certain way and the system is in a Nash equilibria, no one will want to change the way he/she acts.

In game theory it is assumed that games will always find their way to a Nash equilibrium (since this is a stable state). We will, however, be interested in ways of finding (or reaching) a Nash equilibrium in games and the complexity of algorithms which finds such equilibriums.