(67686) Mathematical Foundations of AI May 14, 2008

Lecture 1

Lecturer: Aviv Zohar *Scribe: Aviv Zohar*

1 Introduction

The ability of computers to play games has been a focus of interest from the very beginning of the age of computers. Games such as Chess provide a problem that is very simply and elegantly defined, yet seems to require a high level of intelligence, and so programming computers to play games was one of the first challenges taken on by computer scientists.

Shannon [2] and Turing [3] both wrote papers on Chess playing programs in the early 50's. They suggested ideas such as using min-max search and evaluation functions, as well as pruning the game tree. Their work is the basis of all modern computer programs, but the first actual program that played Chess only appeared in 1956 and actually played a variant on a 6x6 board (a standard board took too much memory and computation for the computers of the time).

Chess play was considered a tough challenge to crack. Initially, there was doubt whether a computer would ever be able to beat a top human player. The feeling was that intuition, originality and expertise that humans possessed would never be matched by computers. In 1968 a Chess master named David Levy made a famous bet that no computer program will be able to beat him in the next 10 years. He won the bet in 1978 when he defeated "Chess 4.7" – one of the top programs of that time, but commented that soon computers would defeat even the best human players.

In 1997, Deep Blue defeats the human world champion Garry Kasparov in a public match that to many symbolizes the end of the era of human supremacy in Chess. While Deep Blue used special hardware to evaluate moves and was in fact a super-computer, Modern computer players regularly beat grand-master level players using in-expensive hardware (personal computers that are now present at every home). The increase in performance is due to the exponential increase in speed of modern hardware, as well as the use of more sophisticated game playing algorithms.

All current top programs that play Chess employ extensive search in the game tree, and so demonstrate that human intuition and originality can actually be countered by brute force search using a fast enough machine.

2 Game Trees and the Value of a Game

We will now examince a class of games that can be represented using trees. These will include games such as Chess, Checkers, Tic-Tac-Toe and other board games.

Definition 1 (informal) *A Game is an interaction between 2 or more players at the end of which each player is awarded a payment (negative payments are also possible and are treated as a cost).*

We will only consider two player games, that are played in turns, starting from some initial position. We will also restrict ourselves to games that are fully competitive (zero-sum), and that can be represented by trees.

Definition 2 *A zero-sum game is a game in which the sum of payments to the players is always 0. I.e, the gains of one player always come at the expense of the other player.*

We will represent the games using a tree. This is known as an extensive form representation:

- Each position p in the game is a vertex in the tree.
- Every vertex is also associated with one of the two players (The one whose turn it is to play).
- The root of the tree is the initial position in the game.
- The children of a vertex p in the tree are game positions that can be reached from that vertex (by the appropriate player) in one move. We shall denote their set by: $Children(p)$.
- Each leaf in the tree is associated with a set of payments that is awarded to the players at that position.

  Since we will be dealing exclusively with zero sum games, we will only specify the payment to the player associated with the leaf. We shall denote the payment at position p by $v(p)$[1].

The game then proceeds in the following manner: Beginning from the root of the tree, the player associated with the current vertex chooses the child of the current vertex it wishes to move to. Eventually a leaf must be reached. At this point the game ends, and players get the payments that are associated with this leaf.

For example, in Chess, the initial set up of the board is the root of the tree, and white always plays first. White can then choose which piece to move (out of several possible legal moves). Each move that can be made by the white player results in a different game position. These will be the children of the root. The player associated with each of these children is the black player, that can decide on the next move that will follow. The outcomes in Chess (victory, defeat, or a tie) can be matched to the payments {-1,0,1}. The tree in Chess is finite since after a position is repeated 3 times a tie is declared, and so every game must end at some point.

Another example of a game tree with payments is shown in figure 1.

Definition 3 *We define the strategy s of a player to be function from the vertices of the graph he is associated with to the children of these vertices. If $s(p) = p_1 \in Children(p)$ then the player will select position p_1 as the move he makes in position p.*

Now that we have fully defined the game, we turn to characterizing a set of strategies of the players that is in some way "optimal":

Theorem 4 (Zermelo's Theorem) *there exist two strategies s_1, s_2 of the first and second player accordingly, and a value v^* such that:*

[1]Note that this representation is somewhat different than the standard one used in Game-Theory. The standard representation usually mentions all payments from the point of view of the first player. The first player then seeks to maximize this number while the second player seeks to minimize it, and thus that representation is known as the *Min-Max* representation. The standard we use here is called the *Neg-Max* representation (for reasons that will soon be obvious) and will yield a much simpler analysis and formulation of the Alpha-Beta algorithm.

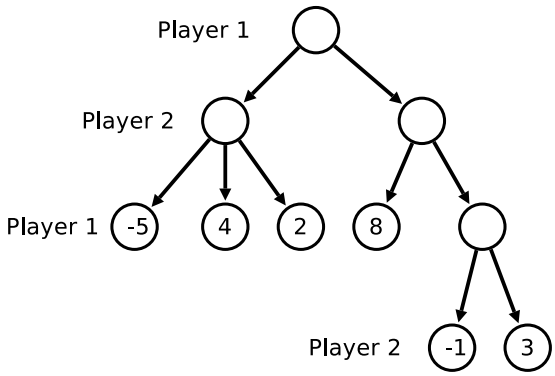

Figure 1: An example of a game tree for a two player, zero-sum game.

- *If player 1 plays s_1 he gets at least v^* no matter what player 2 does.*
- *If player 2 plays s_2 he gets at least $-v^*$ no matter what player 1 does.*

v^* is called the value of the game. Notice also that if the players play s_1, s_2 the theorem implies that the reward that is handed out is exactly v^*.

A key element in the proof of Zermelo's Theorem, is the extension of the value function of a game to all vertices of the game tree as follows:

$$V(p) = \begin{cases} v(p) & \text{if p is a leaf} \\ \max(-V(p_1), \ldots, -V(p_k)) & \text{if } Children(p) = \{p_1, \ldots, p_k\} \end{cases} \quad (1)$$

This extension is "sensible" since each player tries at each position to maximize its own value. I.e, it tries to maximize the negation of the value of his opponent (Which is the value assigned to the children of the current position in the tree). Figure 2 shows a game tree with this extended value function.

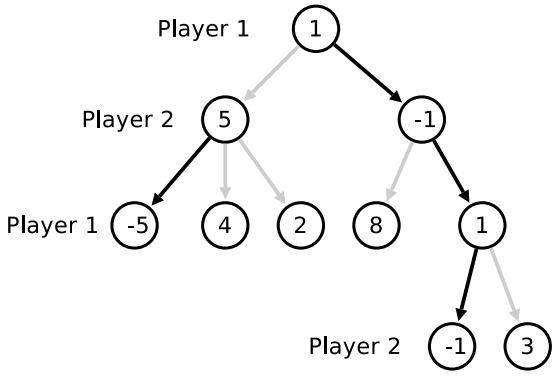

Figure 2: An example of a game tree with the optimal strategies of players (marked by the black arrows) and the value function extended to internal nodes of the tree.

Proof We will prove (by induction on the tree) that the value function V(p) gives the value of the game when applied to the root, and that the strategy of each player is to select the subtree from which this value came. Note that this induction relies on the fact that every subtree rooted at some vertex p is itself a smaller game.

The base case is when we consider a sub-tree that is composed of a single leaf. In this case, obviously one player gets $v(p)$, and the other player gets $-v(p)$ no matter what the other does.

We will now proceed with the induction step. For a subtree rooted at vertex p, and a child p_i we know from the induction assumption that the current player (at p) can gain $-V(p_i)$ if it chooses this child as its move using some strategy in the sub-tree. Since this is true for all children, then by choosing the child that maximizes this payment, the current player has a strategy that guarantees a payment of $\max(-V(p_1), \ldots, -V(p_k)) = V(p)$. The other player is helpless to affect the choice at this point, and can therefore only guarantee a payment of -V(p) by sticking to the optimal strategy that it has for the subtree the current player chooses for him. ■

Observation 5 *The strategies provided by Zermelo's Theorem only provide a safety level. They may not be the optimal strategy in response to an arbitrary strategy that is chosen by the adversary. For example, in the game depicted in Figure 2, Player 1's safety strategy is to go from the root to the right child. However, if player 2 played badly in the left sub-tree (and would choose the leaf with the payment of 4 for example) then it would be best for player 1 to chose the left sub-tree from the root. Obviously this is a very risky strategy that relies on a serious error by player 2.*

3 Evaluation of the Game Tree and the Alpha-Beta Algorithm

Evaluation of the value of the game is in fact a procedure that also finds the safe strategy for each player as well as the expected reward (in case the other player also plays optimally). A good approach to game playing is to follow this strategy.

We now present a straightforward approach to the evaluation of the value of the game via a recursive algorithm that exhaustively evaluates the entire tree (the program is quite simple but it is interesting to contrast it with the Alpha-Beta algorithm that we shall present later).

Algorithm 1 Exhaustive Tree Evaluation

```
procedure FindValue(Position p)
    if isLeaf(p) then
        return v(p);
    m = −∞;
    let {p_1, ..., p_d} = Children(p);
    for k = 1, ..., d do
        m = max(m, −FindValue(p_k));
    return m;
```

Note that during the naive evaluation, every reachable game position is visited, and all possible endings of the games are evaluated. This approach is very often too expensive even

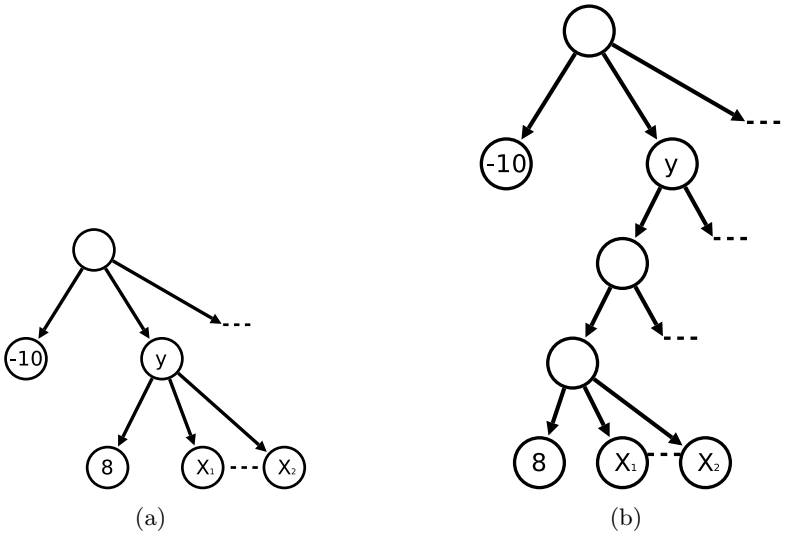

Figure 3: Examples of game trees in which some of the tree does not have to be evaluated in order to determine the value of the root. The subtrees rooted at vertices x_1, x_2 can be ignored since they will have no impact on the value of the root.

for very fast computers. Chess for example, is estimated to have a branching factor of around 35. I.e., every position in Chess, potentially has 35 different legal moves. Games of Chess tend to last on the order of 100 moves, and so the size of the game tree is quite large – too large in fact for modern computers.

Since the entire tree is too large to evaluate, modern programs only expand the nodes of the tree up to a certain depth. A non-terminal node that is reached at the limit, is then evaluated using a heuristic function that attempts to approximately determine if the position will lead to victory, defeat, or a tie. These evaluation functions are usually very simple, and the strength of the approach relies heavily on the large depth of search (look-ahead). It has been demonstrated that computer players that search deeper into the tree gain a higher rating (per extra ply) in competitions.

3.1 Evaluating Only a Part of the Tree

It is often possible to compute the value of the root (and to find the optimal strategies for both players) by evaluating only a part of the game tree. Figure 3 demonstrates two trees where some effort can be saved.

In both examples, we assume that we have already computed the value of some sub-games (the sub-trees rooted at the vertices that have a value of -10 and 8). With this information in hand, we can skip the evaluation of the subtrees rooted at node x_1 and x_2. The reason we can prune these nodes is the following: If $V(x) > 8$ then the player that needs to choose between those nodes will never choose x (As it gives his opponent a higher payoff). Therefore, this will not be the value of the node y. If on the other hand $V(x) \leq 8$, and this value eventually reaches y i.e. $V(y) = -V(x) \geq -8$, then the root node will not choose that subtree because $V(y) \geq -8 > -10$ and so selecting the node whose value is -10 is the optimal move at the root.

Naturally, the saving that we've seen in the example relies heavily on the order in which the tree was evaluated as well as the exact values that were found. We would later find it useful to evaluate the best possible moves as early as possible in order to maximize our savings in terms of evaluated nodes.

3.2 The Alpha-Beta Pruning Algorithm

We now present the Alpha-Beta Algorithm that evaluates the game tree and attempts to save time by ignoring parts of the tree that do not stand a chance to affect the value of the root.

Algorithm 2 Alpha-Beta Tree Evaluation

```
procedure AlphaBeta(Position p, α, β)
    if isLeaf(p) then
        return v(p);
    m = α;
    let {p_1, ..., p_d} = Children(p);
    for k = 1, ..., d do
        m = max(m, −AlphaBeta(p_k, −β, −m));
        if m ≥ β then return m;
    return m;
```

In order to compute the value of some position p, the Alpha-Beta algorithm should be invoked with parameters: AlphaBeta$(p, -\infty, \infty)$.

The reader may notice that the Alpha-Beta algorithm is not too different from the naive approach that was presented earlier. The main difference, is an added condition that may cause an early termination of the for loop. I.e., the parameter β serves as a bound that will stop further evaluation of child states if it is exceeded.

We now turn to proving the correctness of the algorithm. We shall do so using the following lemma that in essence shows that Alpha-Beta computes the value of a position in an in-exact manner if it is ever below α or above β but is otherwise precise.

Lemma 6 *The following 3 statements apply to the Alpha-Beta algorithm for all values of α, β and for any position p:*

$$\begin{aligned} AlphaBeta(p,\alpha,\beta) &\le \alpha \quad if \quad V(p) \le \alpha \\ AlphaBeta(p,\alpha,\beta) &= V(p) \quad if \quad \alpha < V(p) < \beta \\ AlphaBeta(p,\alpha,\beta) &\ge \beta \quad if \quad V(p) \ge \beta \end{aligned} \tag{2}$$

Proof The proof is by induction on the game tree. The base case is when p is a leaf. In this case the algorithm clearly returns the exact value $v(p) = V(p)$, and so the lemma holds. Now, let us assume that the lemma is true for game trees of height h and prove if for game trees of height $h+1$. Let p be a node at height $h+1$ with children $p_1, \ldots p_d$. We therefore have according to the induction assumption, and according to the fact that the Alpha-Beta

execution for the children of p is AlphaBeta$(p_k, -\beta, -m)$:

$$\begin{array}{rll} AlphaBeta(p_k, -\beta, -m) \leq -\beta & if & V(p_k) \leq -\beta \\ AlphaBeta(p_k, -\beta, -m) = V(p_k) & if & -\beta < V(p_k) < -m \\ AlphaBeta(p_k, -\beta, -m) \geq -m & if & V(p_k) \geq -m \end{array} \tag{3}$$

which is equivalent to:

$$\begin{array}{rll} -AlphaBeta(p_k, -\beta, -m) \geq \beta & if & -V(p_k) \geq \beta \\ -AlphaBeta(p_k, -\beta, -m) = -V(p_k) & if & m < -V(p_k) < \beta \\ -AlphaBeta(p_k, -\beta, -m) \leq m & if & -V(p_k) \leq m \end{array} \tag{4}$$

Now, if we observe the loop of the algorithm during the call on position p then it is clear that as long as the loop has not terminated on its k'th iteration

$$m = \max(\alpha, -AlphaBeta(p_1, -\beta, -m), \ldots, -AlphaBeta(p_k, -\beta, -m)) \tag{5}$$

We can now check the 3 different cases that we need to prove:

1. If $V(p) \leq \alpha$ then it must be (according to the definition of the value of a position) that

   $$\forall k \quad -V(p_k) \leq \alpha \tag{6}$$

   However, according to Equation 5 we have that $\alpha \leq m$ at all times and so we have that

   $$\forall k \quad -V(p_k) \leq m \tag{7}$$

   This in turn tells us that according to the induction assumption

   $$\forall k \quad -AlphaBeta(p_k, -\beta, -m) \leq \alpha \tag{8}$$

   and so the algorithm will output at most α as required.

2. If $V(p) \geq \beta$, then let k be the smallest index such that $-V(p_k) \geq \beta$. Up to that point the algorithm did not terminate, and according to the induction assumption we have

   $$-AlphaBeta(p_k, -\beta, -m) \geq \beta \tag{9}$$

   At this point, the algorithm will terminate and return a value that is greater than β, as required.

3. If $\alpha < V(p) < \beta$ then let k be the index for which the maximal value of $-V(p_k)$ is achieved. The algorithm will not terminate early (i.e., it will initiate a call for all children of p). We know that $\alpha < -V(p_k) < \beta$ and so by the induction assumption,

   $$-AlphaBeta(p_k, -\beta, -m) = -V(p_k) \tag{10}$$

   while for all $i \neq k$ the value of $-AlphaBeta(p_i, -\beta, -m)$ will be lower. And so at the k'th iteration m will assume the value of $-V(p_k) = V(p)$ and will not change after that point. The algorithm will output $V(p)$ as required.

■

According to the lemma we have just proven, invoking $AlphaBeta(p, -\infty, \infty)$ on the root will compute its value precisely, and thus the algorithm is correct.

4 An Analysis of Alpha-Beta Pruning

We have seen that Alpha-Beta pruning can save time by scanning less positions, but how significant is this saving? We shall begin by characterizing a good case for which a significant improvement occurs. Note that this is not exactly the best case for the algorithm, and that there may be trees for which more savings occur (although not in full balanced trees).

First we introduce a bit of notation that will be helpful later on. We shall denote each position p by a series that describes the path taken to reach it from the root. I.e., the series $(a_1, a_2, \ldots, a_l)$ will denote the position that is reached by starting at the root, then proceeding to child number a_1 of the root (according to the order of evaluation), then to child number a_2 of that node and so on.

The case we shall examine is one in which the first move evaluated by the algorithm is the optimal move for the player. That is:

$$V(a1, \ldots, a_l) = \begin{cases} v(a1, \ldots, a_l) & \text{if the position is a leaf} \\ -V(a1, \ldots, a_l, 1) & \text{otherwise} \end{cases} \tag{11}$$

Definition 7 *A position $p = (a_1, \ldots, a_l)$ is considered critical if at least one of the following holds:*

- *All odd positions in the series $(a_1, \ldots, a_l)$ are 1's.*
- *All even positions in the series $(a_1, \ldots, a_l)$ are 1's.*

For example, the following positions are critical: $(1, 2, 1, 3, 1, 4), (1, 1, 1), (2, 1, 3, 1)$. The root is also always considered critical.

We can now state the following theorem:

Theorem 8 *If the optimal move is always evaluated first during and execution of Alpha-Beta on a tree T, I.e., if*

$$V(a1, \ldots, a_l) = \begin{cases} v(a1, \ldots, a_l) & \textit{if the position is a leaf} \\ -V(a1, \ldots, a_l, 1) & \textit{otherwise} \end{cases} \tag{12}$$

Then the Alpha-Beta algorithm evaluates exactly the critical states in T

See the proof in [1].

Corollary 9 *If the optimal move is evaluated first on a full tree that has a branching factor of d and a height h, then the Alpha-Beta algorithm evaluates exactly $d^{\lfloor h/2 \rfloor} + d^{\lceil h/2 \rceil} - 1$ positions.*

The corollary is immediate from the theorem, simply by counting all critical states of each type (and subtracting 1 for the root that is counted twice). Notice that while the search is still exponentially large, the effective height of the tree that is searched is halved which allows a computer with a given amount of processing power to search deeper into a game tree. It is for this reason that modern programs try to evaluate potentially good child-positions before seemingly weaker ones.

Another interesting observation is that the critical states are exactly the ones needed for Zermelo's theorem. The value of the game is achieved (or exceeded) in two cases: when one

player plays the optimal strategy and the other plays any other one, or vice versa. The states visited during these games are exactly the critical ones (one of the players plays only 1 at every point which is the optimal move).

We now present one more theorem that shows that Alpha-Beta is optimal up to the order of evaluation of child-positions.

Theorem 10 *Let Alg be some algorithm that examines position and computes the value of the root of the game tree T exactly. Then, there exists an evaluation order on the tree T such that Alpha-Beta examines no more leaves than Alg did.*

The proof appears in [1].

References

[1] Donald E. Knuth and Ronald W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293–326, 1975.

[2] Claude E. Shannon. Programming a computer for playing chess. *Philosophical Magazine*, 41:256–275, 1950.

[3] Alan M. Turing. Digital computers applied to games. Appeared in 'Faster than thought', 1953. http://www.turingarchive.org/browse.php/B/7.