Incorporating Opponent Models into Adversary Search

David Carmel and Shaul Markovitch

Computer Science Department
Technion, Haifa 32000
Israel
carmel@cs.technion.ac.il shaulm@cs.technion.ac.il

Abstract

This work presents a generalized theoretical framework that allows incorporation of opponent models into adversary search. We present the $M^*$ algorithm, a generalization of minimax that uses an arbitrary opponent model to simulate the opponent’s search. The opponent model is a recursive structure consisting of the opponent’s evaluation function and its model of the player. We demonstrate experimentally the potential benefit of using an opponent model. Pruning in $M^*$ is impossible in the general case. We prove a sufficient condition for pruning and present the $\alpha\beta^*$ algorithm which returns the $M^*$ value of a tree while searching only necessary branches.

Content areas: Decision theory, search, game playing

1 Introduction

The minimax algorithm [10] has served as the basic decision procedure for zero-sum games since the early days of computer science. The basic assumption behind minimax is that the player has no knowledge about the opponent’s decision procedure. In the absence of such knowledge, minimax assumes that the opponent selects an alternative which is the worst from the player’s point of view.

However, it is quite possible that the player does have some knowledge about its opponent. Such knowledge may be a result of accumulated experience in playing against the opponent, or may be supplied by some external source. How can such knowledge be exploited? In game theory, the question is known as the best response problem - looking for an optimal response to a given opponent model [3]. However, there is almost no theoretical work on using opponent models in adversary search. Korf [7] outlined a method of utilizing multiple-level models of evaluation functions. Carmel and Markovitch [2] developed an algorithm for utilizing an opponent model defined by an evaluation function and a depth of search.

Jansen [5] describes two situations where it is important to consider the opponent’s strategy. One is a swindle position, where the player has reason to believe that the opponent will underestimate a good move, and will therefore play a poorer move instead. Another situation is a trap position, where the player expects the opponent to overestimate and therefore play a bad move. Another situation, where an opponent model can be beneficial, is a losing position [1]. When all possible moves lead to a loss, an opponent model can be used to select a swindle move.

The minimax approach may fail to predict the opponent’s moves even when the player and the opponent evaluate positions identically. For example, assume that both the player and the opponent
prefer to exchange pieces whenever possible. According to the worst-case assumption of minimax, the player wrongly assumes that the opponent tries to avoid piece exchange\(^1\), while knowledge of the opponent’s strategy allows the player to predict correctly the opponent’s preferences, and therefore plan its moves better.

The goal of this research is to develop a theoretical framework that allows exploitation of an opponent model. We start by defining an opponent model to be a recursive structure consisting of a utility function of the opponent and the player model (held by the opponent). We then describe two algorithms, \(M^*\) and \(M^*_\text{pass}\), generalizations of minimax that can use an opponent model. Pruning in \(M^*\) is not possible in the general case. However, we have developed the \(\alpha\beta\) algorithm that allows pruning given certain constraints over the relationship between the player’s and the model’s strategies.

In Section 2 we describe the \(M^*\) algorithm. Section 3 describes a more efficient version of \(M^*\) that avoids node re-expansion. In section 4 we analyze the properties of \(M^*\). Section 5 describes a method for incorporating pruning into the algorithms. In section 6 we demonstrate experimentally the merit of modeling. Finally, Section 7 concludes.

### 2 The \(M^*\) algorithm

Let \(S\) be the set of possible game states. Let \(\text{Succ} : S \rightarrow 2^S\) be the successor function. Let \(\varphi : S \rightarrow S\) be an opponent model that specifies the opponent’s selected move for each state. The \(M\) algorithm takes a position \(\text{pos} \in S\), a depth limit \(d\), a static evaluation function \(f : S \rightarrow \mathbb{R}\), an arbitrary opponent model \(\varphi\), and returns a value.

\[
M(\text{pos}, d, f, \varphi) = \begin{cases} 
   f(\text{pos}) & \text{if } d \leq 0 \\
   \max_{s \in \text{Succ}(\text{pos})} (f(s)) & \text{if } d = 1 \\
   \max_{s \in \text{Succ}(\text{pos})} (M(\varphi(s), d-2, f, \varphi)) & \text{otherwise}
\end{cases}
\]

The algorithm selects a move in the following way. It generates the successors of the current position. It then applies \(\varphi\) on each of the successors to obtain the opponent’s response and evaluates each of the resulting states by applying the algorithm recursively (with reduced depth limit). It then selects the successor with the highest value. The algorithm returns the static evaluation of the input position when the depth limit is zero.

Note that \(M\) returns a value. \(M\) can also be defined to return the state with the maximal value instead of the value itself. To simplify the discussion, from now on we will assume that \(M\) returns either the state or the value according to the context.

Figure 1 shows a tree of depth 3 expanded by the \(M\) algorithm. At each level of the tree associated with the opponent, the player uses the model function \(\varphi\) to find the opponent’s response. At node \(b\), \(\varphi\) is called and returns node \(d\). The player then calls \(M\) to evaluate node \(d\). \(M\) evaluates the two leaves using the static evaluation function and returns 6, the maximal value. Therefore node \(b\) is worth 6 for the player. Similar reasoning shows that node \(e\), and therefore node \(c\) is worth 8 for the player. Therefore the player selects the move leading to node \(c\).

Let \(M^*_0(\text{pos}, d)\) be the regular minimax algorithm that searches to depth \(d\) using an evaluation function \(f_0\). Minimax uses itself with \(d - 1\) and \(f_0\) as an opponent model, therefore it can be

\(^1\)The player’s evaluation function has the total number of pieces as a component with a negative sign. The player assumes that the opponent tries to minimize this value, therefore maximizing the total number of pieces.
Figure 1: A search tree spanned by the $M$ algorithm with depth limit 3. Squares represent nodes where it is the player’s turn to play. Circles represent nodes where it is the opponent’s turn to play. The opponent model $\varphi$ is used to obtain the move selected by the opponent while the static evaluation function is used to evaluate the resulting state.

defined as a special case of $M$:

$$M_{(f_0, d)}(pos) = M(pos, d, f_0, M_{(f_0, d-1)}(pos))$$

Assume that the player uses an evaluation function $f_1$. A natural candidate to serve as the model $\varphi$ in the $M$ algorithm is $M^1$ with another evaluation function $f_0$. We call this special case of $M$ the $M^1$ algorithm:

$$M_{(f_1, f_0), d}(pos) = M(pos, d, f_1, M_{(f_0, d-1)}(pos)).$$

The $M^1$ algorithm works by simulating the opponent’s minimax search (to depth $d - 1$) to find its selected move and evaluates the selected move by calling itself recursively (to depth $d - 2$).

Figure 2: The search trees spanned by calling $M^1$ with depth limit of 3. Part (a) shows the two calls to minimax for determining the opponent’s choice. Note that the opponent is a maximizer. Squares represent nodes where it is the player’s turn to play. Circles represent nodes where it is the opponent’s turn to play. The values in part (a) correspond to $f_0$. After the player determines that the opponent chooses to move from $b$ to $e$ and from $c$ to $f$, it evaluates the value of $e$ and $f$ by calling $M^1$ recursively. The values in part (b) correspond to $f_1$.

Figure 2 shows an example of a search to depth 3 performed by the $M^1$ algorithm. Part (a) shows the two calls to minimax to simulate the opponent’s search. Note that the opponent is a
maximizer. Part \((b)\) of the figure shows the two recursive calls to \(M^1\) applied to the boards selected by the opponent. Note that while the opponent’s minimax procedure believes that in node \(c\) the player selects the move leading to node \(k\), the player actually prefers node \(j\). Note that the player is only interested in the move selected by the simulation of the minimax search of the opponent. The value returned by the minimax search is completely ignored.

We can define the \(M^n\) algorithm, for any \(n\), to be the \(M\) algorithm with \(\varphi = M^{n-1}\). \(M^n\) can be formally defined as follows:

\[
M^n((f_n, f_{n-1}, \ldots, f_0), d)(\text{pos}) = M(\text{pos}, f_n, d, M^{n-1}((f_{n-1}, \ldots, f_0), d-1))
\]

Thus, a player using the \(M^1\) algorithm assumes that its opponent uses \(M^0\) (minimax), a player using the \(M^2\) algorithm assumes that its opponent uses \(M^1\), etc. We will define the \(M^\star\) algorithm that includes every \(M^n\) algorithm as a special case. \(M^\star\) receives as a parameter a player which includes information about both the player’s evaluation function and its model of the opponent.

**Definition 1** A player is a pair defined as follows:

1. Given an evaluation function \(f\), \(P = (f, \text{NIL})\) is a player (with a modeling-level 0).
2. Given an evaluation function \(f\) and a player \(O\) (with modeling level \(n - 1\)), \(P = (f, O)\) is a player (with a modeling level \(n\)).

The first element of a player is called the player’s strategy and the second element is called the opponent model.

Thus, a zero-level modeling player, \((f_0, \text{NIL})\), is one that does not model its opponent. A one-level modeling player, \((f_1, (f_0, \text{NIL}))\), is one that has a model of its opponent, but assumes that its opponent is a zero-level modeling player. A two-level modeling player, \((f_2, (f_1, (f_0, \text{NIL})))\), is one that uses a strategy \(f_2\), and has a model of its opponent, \((f_1, (f_0, \text{NIL}))\). The opponent’s model uses a strategy \(f_1\) and has a model, \((f_0, \text{NIL})\), of the player. The recursive definition of a player is in the spirit of the Recursive Modeling Method (RMM) by Gmytrasiewicz, Durfee and Wehe [4].

\(M^\star\) receives a position, a depth limit, and a player, and outputs a move selected by the player and its value. The algorithm generates the successor boards and simulates the opponent’s search from each of them in order to anticipate its choice. This simulation is achieved by applying the algorithm recursively with the opponent model as the player. The player then evaluates each of its optimal moves by evaluating the outcome of its opponent’s reaction by applying the algorithm recursively using its own strategy.

Figure 3 shows an example of a search tree spanned by \(M^\star(a, 3, f_2(f_1, f_0))\). The numbers at the bottom are the static values of the leaves. The recursive calls applied to each node are listed next to the node. The dashed lines indicate which move is selected by each recursive call.

The player simulates its opponent’s search from nodes \(b\) and \(c\). The opponent simulates the player by using its own model of the player from nodes \(d\) and \(e\). At node \(d\) the model of the player used by the opponent \((f_0)\) selects node \(h\). The opponent then applies its \(f_1\) function to node \(h\) and concludes that node \(h\), and therefore node \(d\), are worth \(-6\). The opponent then applies the player’s model \((f_0)\) to node \(e\), concludes that the player select node \(j\), applies its own function \((f_1)\) to node \(j\), and decides that node \(j\), and therefore node \(e\), are worth \(-8\). Therefore, the opponent model, when applied to node \(b\), selects the move that leads to node \(d\). The player then evaluates node
The set of recursive calls generated by calling $M^*(a,3,(f_3(f_1,f_0)))$. Each call is written next to the node it is called from. The dashed lines show which move is selected by each call.

d using its own criterion ($f_2$). It applies $M^*$ to node $d$ and concludes that node $d$, and therefore node $b$, are worth $8$. Simulation of the opponent from node $c$ yields the selection of node $g$. The player then evaluates $g$ according to its own strategy and finds that it is worth $10$ (the value of node $n$). Note that when the opponent simulates the player from node $g$, it wrongly assumes that the player selects node $a$. Therefore, the player selects the move that leads to $c$ with a value of 10. Note that using a regular minimax search with $f_2$ would have resulted in selecting the move that leads to node $b$ with a value of 7.

The formal listing of the $M^*$ algorithm is shown in figure 4.

```
Procedure $M^*(pos, depth, (f_{pl}, OPP\_MODEL))$
    if depth = 0
        return $\langle NIL, f_{pl}(pos)\rangle$
    else
        $max\_value \leftarrow -\infty$
        $SUCCESS \leftarrow MoveGen(pos)$
        for each $succe \in SUCCESS$
            if depth = 1
                $player\_value \leftarrow f_{pl}(succe)$
            else
                $\langle opp\_board, opp\_value \rangle \leftarrow M^*(succe, depth - 1, OPP\_MODEL)$
                $\langle player\_board, player\_value \rangle \leftarrow M^*(opp\_board, depth - 2, (f_{pl}, OPP\_MODEL))$
                if $player\_value > max\_value$
                    $max\_value \leftarrow player\_value$
                    $max\_board \leftarrow succe$
        return $\langle max\_board, max\_value \rangle$
```

Figure 4: The $M^*$ algorithm

The $M^n$ algorithm calls $M^0$ when $n$ reaches 0. However, note that $M^*$ does not contain such a call. This will work correctly when the modeling level is larger than the search depth. If the
modeling level of the original player \( d_n \) is smaller than the depth of the search tree \( d \), we replace the 0-level modeling player \( f_0 \) by \( (f_0, (-f_0, (f_0, \ldots))) \).

3 A one-pass version of \( M^* \)

It is obvious that the \( M^* \) algorithm performs multiple expansions of parts of the search tree. We have developed another version of the \( M^* \) algorithm, called \( M^*_{1-pass} \), that expands the tree one time only, just as minimax does. The algorithm expands the search tree in the same manner as minimax. However, node values are propagated differently. Whereas minimax propagates only one value, \( M^* \) propagates \( n + 1 \) values, \( (V_n, \ldots, V_0) \). The value \( V_i \) represents the merit of the current node according to the \( i \)-level model, \( f_i \). \( M^*_{1-pass} \) passes values associated with the player and values associated with the opponent in a different manner. In a player’s node (a node where it is the player’s turn to play)\(^2\), for values associated with the player \( (V_n, V_{n-2}, \ldots) \), \( V_i \) receives the maximal \( V_i \) value among its children. For values associated with the opponent \( (V_{n-1}, V_{n-3}, \ldots) \), \( V_i \) receives the \( V_i \) value of the child that gave the maximal value to \( V_{i-1} \).

Figure 5 shows an example for a tree spanned by \( M^*_{1-pass} \). This is the same tree as the one shown in Figure 3. Let us look at node \( e \) to understand the way in which the algorithm works. Three players evaluate node \( e \): the player, the opponent model, and the opponent’s model of the player. The opponent’s model of the player evaluates all the successors using evaluation function \( f_0 \) and selects board \( j \) with a value of 10. The opponent model knows that it has no effect on the decision taken at node \( e \), since it is the player’s turn to move. It therefore assigns node \( e \) the value of \( j \), selected by the player model, using its own evaluation function \( f_1 \), \( (f_1(j) = -8) \). The player actually prefers node \( k \) with the higher \( f_2 \) value \( (f_2(k) = 7) \). Thus, the vector propagated from node \( e \) is \((7, -8, 10)\). Note that the values in the vectors correspond to the results of the recursive calls in figure 3.

Figure 6 lists the \( M^*_{1-pass} \) algorithm.

---

\(^2\)Traditionally such a node is called a MAX node. However, we assume that both players are maximizers.
Procedure $M^*_1 \text{-pass}(pos, depth, (f_0, (f_{n-1}, \ldots, f_0) \ldots))$

if $depth = 0$
    return $(f_0(pos), \ldots, f_0(pos))$
else $SUCC \leftarrow MoveGen(pos)$
    $V \leftarrow (-\infty, \ldots, -\infty)$
    for each $succe \in SUCC$
        $succe, V \leftarrow M^*_1 \text{-pass}(succe, depth - 1, (f_0, (f_{n-1}, \ldots, f_0) \ldots))$
        for each $i$ associated with current player
            if $succe, V[i] > V[i]$
                $V[i] \leftarrow succe, V[i]$
            if $i < n$
                $V[i + 1] \leftarrow succe, V[i + 1]$
    return $(V[0], \ldots, V[depth])$

Figure 6: $M^*_1 \text{-pass}$: A version of the $M^*$ algorithm that performs only one pass over the search tree

4 Properties of $M^*$

The following theorem shows that $M^*$ and $M^*_1 \text{-pass}$ return the same value.

**Theorem 1** Assume that $PLAYER$ is a $n$-level modeling player.
Let $(v, b) = M^*(pos, depth, PLAYER)$,
and let $(V[n]) = M^*_1 \text{-pass}(pos, depth, PLAYER)$. Then $v = V[n]$.

**Proof:**

The proof for this theorem and the next one can be found in a technical report (due to the blind review restriction we omit the cite of the tech-report). □

It seems as though $M^*_1 \text{-pass}$ is always preferred over $M^*$ because it expands less nodes. $M^*_1 \text{-pass}$ expands each node in the tree only once, while $M^*$ re-expands many nodes. However, while $M^*_1 \text{-pass}$ performs less expansions than $M^*$, it may perform more evaluations. See Figure 3 for example. $M^*$ never calls $f2$ for leaves $j, k, l$ and $m$ since they belong to subtrees under boards that were not selected by the opponent model. $M^*_1 \text{-pass}$ must apply all the evaluation functions for every leaf.

An upper limit on the number of node expansions and evaluations in $M^*$ and $M^*_1 \text{-pass}$ is given in the following lemma.

**Theorem 2** Assume that $M^*$ and $M^*_1 \text{-pass}$ search a tree with a uniform branching factor $b$ and depth $d$.

1. The number of calls to the expansion function by $M^*$ is bounded by $(b + 1)^{d-1}$. The number of calls by $M^*_1 \text{-pass}$ is $\frac{b^{d-1}}{1 - 1/b}$.
2. The number of calls to evaluation functions by $M^*$ is bounded by $(b + 1)^d$, the number of calls by $M^*_1 \text{-pass}$ is $db^d$. 

7
When the modeling level, $ML$, is smaller than $d$, the number of calls to evaluation functions by $M_{1-pass}^*$ is $(ML+1)b^d$ (the last $d-(ML+1)$ entries are just the values of $f0$ with alternating sign.)

The lemma implies that $M^*$ and $M_{1-pass}^*$ each has an advantage. If it is more important to reduce the number of node expansions than the number of evaluation function calls then one should use $M_{1-pass}^*$. Otherwise, $M^*$ should be used. For example, for $d=10$ and $b=30$ and a one-level modeling player, $M^*$ expands 1.3 times more nodes than $M_{1-pass}^*$ but $M_{1-pass}^*$ performs 1.5 times more calls to evaluation functions. Note that when the set of evaluation functions consist of the same features (perhaps with different weights), the overhead of multiple evaluation is reduced significantly.

4.1 The relationship between $M^*$ and minimax

We have already shown that $M^*$ is a generalization of minimax. An interesting property of $M^*$ is that it always selects a move with a value greater or equal to the value of the move selected by minimax that searches the same tree with the same strategy.

**Theorem 3** Assume that $M^*$ and Minimax use the same evaluation function $f$. Then

\[ \text{Minimax}(\text{pos}, \text{depth}, f) \leq M^*(\text{pos}, \text{depth}, (f, \text{OPP\_MODEL})) \]

for any OPP\_MODEL.

**Proof:**

We will prove that this property holds for every node in the tree spanned by the two algorithms, by induction on the depth of search. For convenience we will prove it for $M_{1-pass}^*$.

For $\text{depth} = 0$, for any OPP\_MODEL

\[ \text{Minimax}(\text{pos}, \text{depth}) = f(\text{pos}) = M_{1-pass}^*(\text{pos}, \text{depth}, (f, \text{OPP\_MODEL})) \]

Let us assume correctness for $d \leq \text{depth}$ and prove for $d = \text{depth} + 1$

If $\text{pos}$ is a player’s node,

\[ \text{Minimax}(\text{pos}, \text{depth} + 1) = \max \{ \text{Minimax}(\text{succ}, \text{depth}) \mid \text{succ} \in \text{SUCC}(\text{pos}) \} \]

According to the inductive assumption,

\[ \leq \max \{ M_{1-pass}^*(\text{succ}, \text{depth}, (f, \text{OPP\_MODEL})) \mid \text{succ} \in \text{SUCC}(\text{pos}) \} \]

\[ = M_{1-pass}^*(\text{pos}, \text{depth} + 1, (f, \text{OPP\_MODEL})) \]

If $\text{pos}$ is an opponent’s node,

let $s$ be the successor with the maximal value according to the opponent’s model.

\[ \text{Minimax}(\text{pos}, \text{depth} + 1) = \min \{ \text{Minimax}(\text{succ}, \text{depth}) \mid \text{succ} \in \text{SUCC}(\text{pos}) \} \]

\[ \leq \text{Minimax}(s, \text{depth}) \]

According to the inductive assumption,
The theorem states that if you have a good reason to believe that your opponent’s model is different than yours, you could only benefit by using $M^*$ instead of minimax. The reader should note that the above theorem does not mean that $M^*$ selects a move that is better according to some objective criterion, but rather a subjectively better move (from the player’s point of view, according to its strategy). If the player does not have a reliable model of its opponent, then playing minimax is a good cautious strategy.

4.2 The relationship between $M^*$ and MAXN

The MAXN algorithm [9] also propagates vectors of values up the game tree. However, $M_{1-pass}$ and MAXN are targeted at different goals. MAXN is an extension of minimax that can handle $n$ players while $M_{1-pass}$ is an extension of two-player minimax that can handle $n$-level modeling.

It is tempting to claim that $M_{1-pass}$ is a special case of MAXN where each model stands for one player. Such a claim is wrong. In MAXN, at each level of the search tree, only one player has an active role of selecting a move. The other players passively pass their values that are associated with the player’s selection. In $M_{1-pass}$, every second model is associated with the player whose turn is to play, therefore half of the models selects a move, and the player above each of them passes its value that is associated with the selection of the model below it.

For this reason, the two algorithms propagate values up the tree differently. While all the values of a vector in MAXN come from the same leaf, the values of a vector in $M_{1-pass}$ may come from different leaves.

To illustrate the difference, let us look at Figure 5. Table 1 shows the steps taken by each of the algorithms at node $e$.

It is easy to show that $M^*$ can simulate MAXN for the case of two players by using the player $P1 = (f1, (f2, (f1, (f2, ...))))$.

5 Adding pruning to $M^*$

One of the most significant extensions of the minimax algorithm is the $\alpha/\beta$ pruning technique which can reduce the average branching factor, $b$, of the tree searched by the algorithm to approximately $\sqrt{b}$ [6]. This algorithm avoids searching subtrees that cannot effect the minimax value of the parent node.

Is it possible to add such an extension to $M^*$ as well? Unfortunately, if we assume a total independence between $f_1$ and $f_0$, it is easy to show that such a procedure cannot exist. Figure 7 illustrates a situation where using minimax with evaluation function $f_1$ (or $M^*$ with the player $(f1, (-f1, NIL))$) which can avoid searching node $g$, whereas $M^*$ that uses $f_1$ instead of $-f_1$, cannot perform this pruning. Knowing that the opponent will have at least $-5$ for node $e$ does not have any implications on the value of node $e$ for the player.

A similar situation arises in multi-player game trees. Luckhardt and Irani [9] conclude that pruning is impossible in Maxn without further restrictions about the players’ evaluation functions. Korf [8] showed that a shallow pruning for Maxn is possible if we assume an upper bound on the sum of the players’ functions, and a lower bound on every player’s function.
At node $e$, it is player 3’s turn to move. Player 3 uses function $f_0$ to evaluate boards. Player 3 selects board $j$ (with a value of 10). Player 2 evaluates node $j$ using $f_1$, $(f_1(j) = -8)$, and propagates this value to $e$. Player 1 knows that it has no effect on the decision in $e$. The vector propagated up is $<4, -8, 10>$. The basic assumption used for the original $\alpha\beta$ algorithm is that $f_1 + f_0 = 0$ (the zero-sum assumption). This assumption is used to infer a bound on a value of a node for a player based directly on the opponent’s value. A natural relaxation to this assumption is $|f_1 + f_0| \leq B$. This assumption means that while $f_1$ and $-f_0$ may evaluate a board differently, this difference is bounded. For example, the player may prefer a rook over a knight while the opponent prefers the opposite. In such a case, although the player’s value is not a direct opposite of the opponent’s value, we can infer a bound on the player’s value based on the opponent’s value and $B$.

The above assumption can be used in the context of the $M^{*}_{\text{pass}}$ algorithm to determine a bound on $V_i + V_{i-1}$ at the leaves level. But in order to be able to prune using this assumption, we first need to determine how these bounds are propagated up the search tree.

**Lemma 1** Assume that $A$ is a node in the search tree spanned by $M^{*}_{\text{pass}}$. Assume that $S_1, \ldots, S_k$ are its successors. If there exist non-negative bounds $B_1, \ldots, B_n$, such that for each successor $S_i$, $f_1 = 8$, $f_0 = -6$.

![Figure 7: An example for a search tree where the standard $\alpha\beta$ would have pruned the branch leading to node $g$. However, such pruning would change the $M^{*}$ value of the tree.](image-url)
and for each model $i$, $|V_{S_i}[i] + V_{S_j}[i-1]| \leq B_i$. Then, for each model $1 \leq i \leq n$:

$$|V_{A}[i] + V_{A}[i-1]| \leq B_i + 2 \cdot B_{i-1}.$$  

(9)

Proof:

Assume $V_{A}[i] = V_{S_i}[i]$ and $V_{A}[i-1] = V_{S_i}[i]$. If $j = k$, $V_{A}[i]$ and $V_{A}[i-1]$ were propagated from the same successor, therefore,

$$|V_{A}[i] + V_{A}[i-1]| \leq B_i.$$  

(10)

If $j \neq k$, $A$ is an $i$’th player node and therefore $V_{A}[i-1]$ and $V_{A}[i-2]$ were propagated from the same successor $S_j$. It follows that

$$B_{i-1} \geq V_{S_i}[i-1] + V_{S_i}[i-2] \geq V_{S_i}[i-1] + V_{S_j}[i-2].$$  

(11)

It is easy to show that for any successor $S$,

$$|V_{S}[i] - V_{S}[i-2]| \leq B_i + B_{i-1}.$$  

(12)

Summing up the two inequalities for the $S_i$ successor, we get

$$B_i + 2 \cdot B_{i-1} \geq V_{S_i}[i] + V_{S_i}[i-1] = V_{A}[i] + V_{A}[i-1].$$  

For the second side of the inequality,

$$V_{A}[i] + V_{A}[i-1] = V_{S_i}[i] + V_{S_i}[i-1] \geq V_{S_i}[i] + V_{S_i}[i-1] \geq -B_i \geq -B_i - 2 \cdot B_{i-1}. \quad \square$$  

5.1 The $\alpha\beta*$ algorithm

Based on lemma 1, we have developed an algorithm, $\alpha\beta^*$, that can perform a shallow and deep pruning assuming bounds on the absolute sum of functions of the player and its opponent model. The algorithm takes as input a position, a depth limit, and for each model $i$, a strategy $f_i$, an upper bound $B_i$ on $|f_i + f_{i-1}|$, and a cutoff value $\alpha_i$. It returns the $M^*$ value of the root by searching only those nodes that might affect this value.

The algorithm works similarly to the original $\alpha\beta$ algorithm, but is much more restricted as to which subtrees can be pruned. The $\alpha\beta^*$ algorithm only prunes branches that all models agree to prune. In regular $\alpha\beta$, the player can use the opponent’s value of a node to determine whether it has a chance to attain a value that is better than the current cutoff value. This is based on the opponent’s value being exactly the same as the player’s value (except for the sign). In $\alpha\beta^*$, the player’s function and the opponent’s function are not identical, but their difference is bounded. The bound on $V_i + V_{i-1}$ depends on the distance from the leaves level. At the leaves level, it can be directly computed using the input $B_i$. At distance $d$, the bound can be computed from the bounds for level $d - 1$ as stated by lemma 1.

A cutoff value $\alpha_i$ for a node $v$ is the highest current value of all the ancestors of $v$ from the point of view of player $i$. $\alpha_i$ is modified at nodes where it is player $i$’s turn to play, and is used for pruning where it is player $i - 1$’s turn to play. At each node, for each $i$ associated with the player whose turn it is to play, $\alpha_i$ is maximized any time $V_i$ is modified. For each $i$ such that $i - 1$ is associated with the current player, the algorithm checks whether the $i$ player wants its model (the $i - 1$ player) to continue its search.

\footnote{For sake of clarity, the algorithm computes the bound $B$ for each node. However, for efficient implementation, a table of the $B$ values can be computed once at the beginning of the search, since they depend only on the $B_i$ and the depth.}
Figure 8: An example of pruning performed by $\alpha\beta^*$.

Figure 8 shows a search tree similar to the one in figure 7 with one difference: every leaf $l$ satisfies the bound constraint $|f_1(l) + f_0(l)| \leq 2$. This bound allows the player to perform a cutoff of branch $g$, knowing that the value of node $c$ for the opponent will be at least $-5$. Therefore, its value will be at most 7 for the player.

Figure 9: An example of deep pruning performed by $\alpha\beta^*$.

Figure 9 shows pruning performed at a deeper level of the tree. The opponent’s model of the player (the player with $f_1$) is ready to prune node $j$ as in Figure 8. However, to stop the search the player itself (with $f_3$) must also agree to prune. The player reasons that its opponent makes the selection in node $f$ and the opponent model (with $f_2$) has already a value of at least $-2$. Because of the constraint $|f_3 + f_2| \leq 3$, the player infers that it can get at most 5 from node $j$. However, it already has a value of 7 from previous search, therefore it agrees to prune.

The $\alpha\beta^*$ algorithm is listed in figure 10. The following theorem proves that $\alpha\beta^*$ always returns the $M^*$ value of the root.
Theorem 4  Let $V = \alpha \beta^*(pos, d, \text{PLAYER}, (-\infty, \ldots, -\infty))$.

Let $V' = M_{i-\text{pass}}^2(pos, d, \text{PLAYER'})$ where $\text{PLAYER'}$ is $\text{PLAYER}$ without the bounds. Assume that for any leaf $l$ of the game tree spanned from position pos to depth $d$, $|f_i(l) + f_{i-1}(l)| \leq b_i$. Then, $V = V'$. 

Proof:

For proving that $\alpha \beta^*$ returns the $M^*$ value, it is sufficient to show that any node pruned by $\alpha \beta^*$ can have no effect on the $M^*$ value. Assume that $u$ is an opponent’s node, and after searching one of its successors, all the models associated with the player agree to prune. This means that for every model $i$ associated with the player, $\alpha_i \geq B_i - V[i - 1]$. From lemma 1, it follows that $V[i] + V[i - 1] \leq B_i$ and therefore $\alpha_i \geq V[i]$. Since $\alpha_i$ is the current best value for $V[i]$ of the parent node, there is no way that the current $V[i]$ will affect its father’s value. The same argument holds if $u$ is a player’s node.

As the player function becomes more similar to its opponent model (but with an opposite sign),
the amount of pruning increases up to the point where they use the same function, in which case
\( \alpha \beta^* \) prunes as much as \( \alpha \beta \).

Finding an upper bound on the sum of the evaluation functions is an easy task for most practical
evaluation functions [7]. Unfortunately, the bound on the sum of values increases with the distance
from the leaves and therefore reduces the amount of pruning. Therefore, using loose bounds will
probably prohibit pruning.

In order to get some idea of how much \( \alpha \beta^* \) prunes, we have run a simulation applying \( \alpha \beta^* \)
on uniform trees of depth \( d \), and fixed branching factor \( b \), whose leaves were assigned two random
values with a sum bounded by \( B \) (we mark such a tree by \( \text{TREE}(b, d, B) \)). The experiment was run
with one-level modeling player, \((f_1, f_0)\) such that \( |f_1 + f_0| \leq B \). The experiments were conducted
for \( \text{TREE}(2, 10, B) \). For each bound \( B \) we created 1000 trees and searched them using \( \alpha \beta^* \). Graph
11 shows the portion of leaves pruned as a function of the bound \( B \). As expected, the amount of
pruning decreases as the difference between the functions increases. The maximum level of pruning
is achieved for \( B = 0 \), where \( \alpha \beta^* \) is reduced to standard \( \alpha \beta \).

6 Experimental study: The potential benefit of \( M^* \)

We have performed a set of experiments to test the potential merit of the \( M^* \) algorithm. The
experiments involve three players: \texttt{MSTAR}, \texttt{MINIMAX} and \texttt{OPPONENT}. \texttt{MSTAR} is a one-level
modeling player. \texttt{MINIMAX} and \texttt{OPPONENT} are regular minimax players (using \( \alpha \beta \) pruning).
The experiments were conducted using the domain of checkers. Each tournament consisted of 800
games. A limit of 100 moves per game was set during the tournament.

In the first experiment all players searched to depth 4. \texttt{MINIMAX} and \( M^* \) used one function
while \texttt{OPPONENT} used another with equivalent power. \texttt{MSTAR} knows its opponent’s function
while \texttt{MINIMAX} implicitly assumes that its opponent uses the opposite of its own function. Both
\texttt{MSTAR} and \texttt{MINIMAX}, limited by their own search depth, wrongly assume that \texttt{OPPONENT}
searches to depth 3. The following table shows the results obtained.
The first row of the table shows that indeed the two evaluation functions are of equivalent power. The second row shows that MSTAR indeed benefited using the extra knowledge about the opponent.

In the second experiment all players searched to depth 4. The three players used the same evaluation function

\[ f(\text{board}, \text{player}) = Material(\text{board}, \text{player}) - \frac{1}{25} \cdot Total\,\text{Pieces}(\text{board}) \]

However, while \( M^* \) knows its opponent’s function, MINIMAX implicitly assumes that its opponent uses the function

\[
\begin{align*}
   f(\text{board}, \text{opponent}) & = -f(\text{board}, \text{player}) \\
   & = -Material(\text{board}, \text{player}) + \frac{1}{25} \cdot Total\,\text{Pieces}(\text{board}) \\
   & = Material(\text{board}, \text{opponent}) + \frac{1}{25} \cdot Total\,\text{Pieces}(\text{board})
\end{align*}
\]

Therefore, while OPPONENT prefers exchange positions, MINIMAX assumes that it prefers to avoid them. The following table shows the results obtained.

<table>
<thead>
<tr>
<th>Wins</th>
<th>Draws</th>
<th>Losses</th>
<th>Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>MINIMAX vs. OPPONENT</td>
<td>171</td>
<td>461</td>
<td>168</td>
</tr>
<tr>
<td>MSTAR vs. OPPONENT</td>
<td>290</td>
<td>351</td>
<td>159</td>
</tr>
</tbody>
</table>

This result is rather surprising. Despite the fact that all players used the same evaluation function, and searched to the same depth, the modeling program achieved a significantly higher score.

We repeated the last experiment replacing the \( M^*_1 \)-pass algorithm with \( \alpha\beta^* \) and measured the amount of pruning. The bound used for pruning is:

\[
|f_1 + f_0| = |Material(\text{board}, \text{player}) - \frac{1}{25} \cdot Total\,\text{Pieces}(\text{board}) + Material(\text{board}, \text{opponent}) - \frac{1}{25} \cdot Total\,\text{Pieces}(\text{board})| \leq \frac{2}{25} \cdot Total\,\text{Pieces}(\text{board})
\]

While the average number of leaves-per-move for a search to depth 4 by \( M^*_1 \)-pass was 723, \( \alpha\beta^* \) managed to achieve an average of 190 leaves-per-move. \( \alpha\beta \) achieved an average of 66 leaves-per-move.

The last results raise an interesting question. Assume that we allocate a modeling player and an un-modeling opponent the same search resources. Is the benefit achieved by modeling enough to overcome the extra depth that the non-modeling player can search due to the better pruning? We have tested the question in the context of the above experiment. We wrote an iterative deepening versions of both \( \alpha\beta \) and \( \alpha\beta^* \) and let the two play against each other with the same limit on the number of leaves.

<table>
<thead>
<tr>
<th>Wins</th>
<th>Draws</th>
<th>Losses</th>
<th>Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \alpha\beta^* ) vs. OPPONENT</td>
<td>226</td>
<td>328</td>
<td>246</td>
</tr>
</tbody>
</table>
This result is rather disappointing. The benefit of modeling was outweighed by the cost of the reduced pruning. In general this is an example of the known tradeoff between knowledge and search. The question of when is it worthwhile to use the modeling approach remains open and depends mostly on the particular nature of the search tree and evaluation functions.

7 Conclusions

This paper describes a generalized version of the minimax algorithm that can utilize different opponent models. The $M^*$ algorithm simulates the opponent's search to determine its expected decision for the next move, and evaluates the resulted state by searching its associated subtree using its own strategy. We then present the $M_{\text{pass}}^*$ algorithm, a version of $M^*$ that expands the tree only once, but propagates all the necessary values.

One of the most important techniques in searching game trees is $\alpha\beta$ pruning. We explored the possibility of applying similar pruning techniques to $M^*$. Unfortunately, pruning is impossible in the general case since the zero-sum assumption does not hold. However, we developed a pruning algorithm, $\alpha\beta^*$, that utilizes a relaxed version of the zero-sum assumption to allow pruning. Pruning is allowed given a tight bound on the sum of the player's and model's functions.

Experiments performed in the domain of checkers demonstrated the advantage of $M^*$ over minimax. Can the benefit of modeling outweigh the cost of reduced pruning? We don't have a conclusive answer for that question. We expect the answer to be dependent on the particular domain and particular evaluation functions used.

There are still many remaining open questions. How does an error in the model effect performance? How can a player use an uncertain model? How can a player acquire a model of its opponent?

We answered some of these questions but could not include the results here due to lack of space (a full version of the paper that includes these parts is available as [reference to technical report omitted due to the requirement of blind review]). We have conducted a set of parametric experiments that tested various properties of $M^*$. These experiments showed that an error in the model can deteriorate performance significantly. We developed an algorithm, $M^*_{\text{error}}$, for the cases where the player knows a bound on the error. The algorithm converges to $M^*$ when the error bound approaches zero, and converges to minimax when the bound goes to infinity. We also developed an algorithm for learning an opponent model, both depth and evaluation function, using its moves as examples. The algorithm works by iteratively increasing the model depth and learning a function that best predicts the opponent's moves for that depth.

In the introduction we raised the question of utilizing knowledge about the opponent's decision procedure in adversary search. We believe that this work presents a significant progress in our understanding of opponent modeling, and can serve as a basis for further theoretical and experimental work in this area.

References


