Learning and Using Opponent Models in 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

While human players adjust their playing strategy according to their opponent, computer programs, which are based on the minimax algorithm, use the same playing strategy against a novice as against an expert. This is due to the assumption of minimax that the opponent uses the same strategy as the player. This work studies the problem of opponent modeling in game playing. M^*, an algorithm for searching game-trees using an opponent model is described and analyzed. We demonstrate experimentally the benefit of using an opponent model and show the potential harm caused by the use of an inaccurate model. We then describe an algorithm, M^*_ϵ, for using uncertain models when a bound on the model error is known. Pruning in M^* is impossible in the general case. We prove a sufficient condition for pruning and present a pruning algorithm, $\alpha\beta^*$, that returns the M^* value of a tree, searching only necessary subtrees. Finally, we present a method for acquiring a model for an unknown player. First, we describe a learning algorithm that acquires a model of the opponent's depth of search by using its past moves as examples. Then, an algorithm for acquiring a model of the player's strategy, both depth and function, is described and evaluated. Experiments with this algorithm show that when a superset of the set of features used by a fixed opponent is available to the learner, few examples are sufficient for learning a model that agrees almost perfectly with the opponent.

1 Introduction

> "...At the press conference, it quickly became clear that Kasparov had done his homework. He admitted that he had reviewed about fifty of DEEP THOUGHT's games and felt confident he understood the machine..." [11]

One of the most notable challenges that the Artificial Intelligence research community has been trying to face during the last five decades is the creation of a computer program that can beat the world chess champion. Most activity in the area of game-playing programs has been concerned with efficient ways of searching large game trees. However, good playing performance involves additional types of intelligent processes. The quote above highlights one type of such a process that is performed by expert human players: acquiring a model of their opponent's strategy.

While human players adjust their playing strategy according to their opponent, computer programs play the same against a novice as against an expert. Most playing programs use the minimax

*This report supersedes CIS9402

algorithm for search [19]. The main assumption of this algorithm is that the opponent uses the same strategy as the player.

There are several situations where the modeling approach has advantage over the non-modeling approach of the standard minimax procedure. Jansen [6], describes two situations in which it is important to consider the opponent's playing ability. 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. Choosing trap or swindle positions is good strategy when the player has reason to believe that its opponent searches to shallower depth than itself. Another situation, where an opponent model can be beneficial, is a losing position [2]. If all possible moves lead to a loss, minimax chooses one of them arbitrarily, in contrast to human players that can utilize their opponent model in order to select a swindle move.

All the above cases assume that the opponent makes error in evaluating moves. However, the potential benefit of modeling is not limited to such cases. When the opponent and the player evaluate positions differently, it may be possible to exploit the knowledge about the opponent preferences. For example, if the player prefers crowded boards and knows that the opponent exchanges figures when possible, she can avoid positions that enable the opponent to enforce figure exchanges.

The minimax approach may fail to predict the opponent's moves even when the player and the opponent evaluate positions identically. Assume for example that both players prefer to exchange figures when possible. Thus the total number of figures with a negative sign is a component of the player's and the opponent's evaluation function. However, the player assumes that the opponent tries to minimize the player's function, therefore to maximize its negation. It turns out that the player assumes that the opponent's function contains the total number of figures with *positive* sign. That means that minimax implicitly forces the player to wrongly assume that the opponent avoids figure exchanges.

Several researchers have pointed out the importance of opponent modeling [17, 2, 7, 9, 1, 18], but the acquisition and utilization of an opponent model have not received much attention in the computational game research community. Korf [9] outlined a method of generalizing the minimax algorithm for utilizing multiple-level models of evaluation functions. The work described in this paper builds on Korf's research and expands it in several ways.

The goal of this research is to study the utilization and acquisition of opponent models in game-playing. We make an attempt to find answers to the following questions:

1. Assuming that we possess an opponent model, how can we utilize it?
2. What are the potential benefits of using opponent models?
3. How does the accuracy of the model effect its benefit?
4. How can we use an uncertain model?
5. How can a program acquire a model of its opponent?

Section 2 of this paper deals with the first question: Developing algorithms for using an opponent model. Section 3 deals with the second and the third questions: Measuring the potential benefits of modeling and testing the effects of modeling accuracy on its benefits. In section 4, we describe an algorithm for using uncertain models. Section 5 describes a method for incorporating pruning into the algorithms. Section 6 describes algorithms for acquiring opponent models. Finally, section 7 concludes.

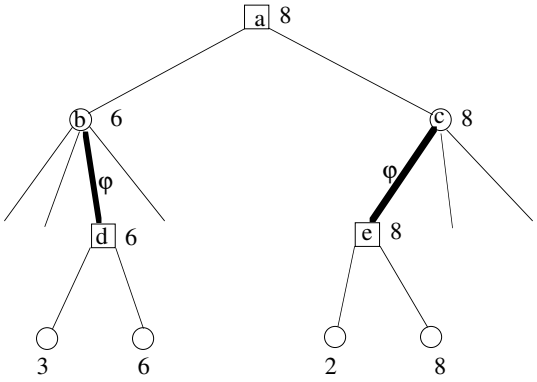

Figure 1: A search tree spanned by the M algorithm with depth limit 3. Squares represent nodes where it is the player's turn to play. Circles represent nodes where it is the opponent's turn to play. The opponent model φ is used to get the move selected by the opponent while the static evaluation function is used to evaluate the resulting board.

2 Using opponent models

Before studying the problem of *acquiring* an opponent model, we develop a methodology for *using* it. In this section we present a set of algorithms for using opponent models in adversary search.

2.1 The M algorithm: Using an arbitrary opponent model

Let S be the set of possible game states. 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, a depth limit, a static evaluation function, an arbitrary opponent model φ, and returns a selected move and a value.

The algorithm selects a move in the following way. It generates the successors of the current position. It then applies φ on each of the successors to get the opponent's response and evaluates each of the resulting boards 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.

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 φ to find the opponent's response. At node b, φ 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 worth 6 for the player. Similar reasoning shows that node e, therefor node c, worth 8 for the player. Therefor the player selects the move leading to node c.

Figure 2 shows the formal listing of the M algorithm.

Note that the φ is used as a black box. In the following subsections we make more explicit assumptions about the nature of φ.

2.2 The M^1 algorithm: Using MINIMAX as a model

Let M^0 be the regular minimax algorithm. A natural candidate to serve as the model φ in the M algorithm is M^0 with a given static evaluation function f_{opp}. We call this special case of M, where

```
Procedure M (pos, f_pl, depth, φ)
  if depth = 0
    return ⟨NIL, f_pl(pos)⟩
  else
    max_value ← −∞
    SUCC ← MoveGen(pos)
    for each succ ∈ SUCC
      if depth = 1
        player_value ← f_pl(succ)
      else
        opp_board ← φ(succ)
        ⟨player_board, player_value⟩ ← M (opp_board, f_pl, depth − 2, φ)
      if player_value > max_value
        max_value ← player_value
        max_board ← succ
    return ⟨max_board, max_value⟩
```

Figure 2: The M algorithm

$\varphi = M_0$, the M^1 algorithm. This algorithm works by simulating the opponent's minimax search to find its selected move, as described in the previous subsection.

Figure 3 shows an example of 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 from the boards selected by the opponent. Note that while the opponent's minimax procedure believes that in node e 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.

2.3 The M^* algorithm: An arbitrary modeling level

We can define the M^2 algorithm to be the special case of M where $\varphi = M^1$. 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:

$$M^0{}_{f_0,d}(pos) = \text{minimax}(pos, f_0, d) \quad (1)$$

$$M^n{}_{f_n,d}(pos) = M(pos, f_n, d, M^{n-1}{}_{f_{n-1},d-1}) \quad (2)$$

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. In this subsection we define the M^* algorithm that includes every M^n algorithm as a special case. M^* gets as a parameter a *player* which includes information about the player's evaluation function and about its model of the opponent.

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

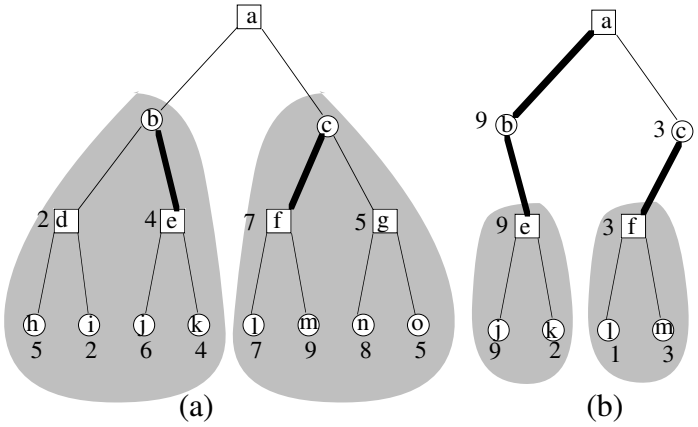

Figure 3: The search trees spanned by calling M^1 with depth limit of 3. Part (a) shows the two calls to minimax for determining the opponent's choice. Note that the opponent is a maximizer. Squares represent nodes where it is the player's turn to play. Circles represent nodes where it is the opponent's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

1. *Given an evaluation function f, $P = (f, NIL)$ is a* player.
2. *Given an evaluation function f and a* player *O, $P = (f, O)$ is a* player.

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

Definition 2 *Given a player $P = (S, O)$. The* modeling level *of a player is defined by the recurrence:*

$$ML(P) = \begin{cases} 0 & if\ O = NIL \\ ML(O) + 1 & otherwise \end{cases} \tag{3}$$

Thus, a zero-level modeling player, (S_0, NIL), is one that does not model its opponent. A one-level modeling player, $(S_1, (S_0, NIL))$, 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, $(S_2, (S_1, (S_0, NIL)))$, is one that uses a strategy S_2, and has a model of its opponent, $(S_1, (S_0, NIL))$. The opponent's model uses a strategy S_1 and has a model, (S_0, 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 [15].

M^* gets 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 done by applying the algorithm recursively with the opponent model as the player. The player then evaluates each of its optional moves by evaluating the outcome of its opponent's reaction by applying the algorithm recursively using its own strategy.

Figure 4 shows an example for a search tree spanned by $M^*(a, 3, f_2(f_1, f_0))$. 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 what 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

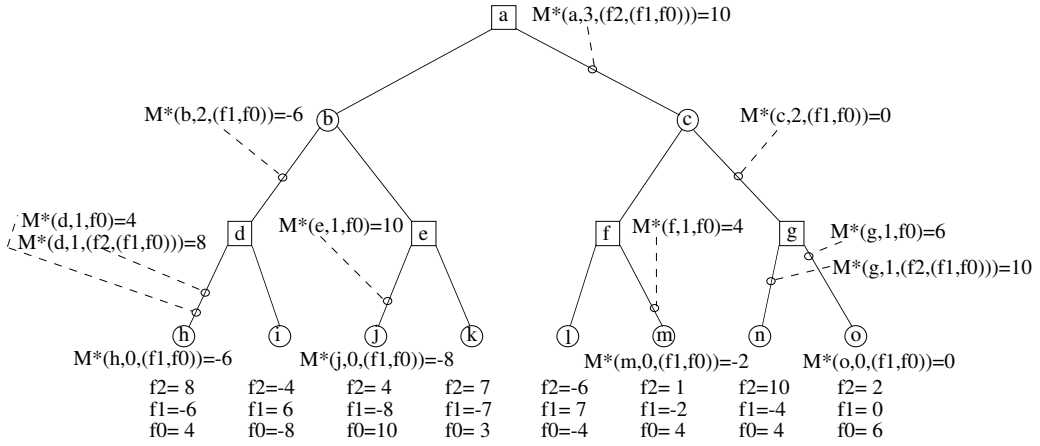

Figure 4: The set of recursive calls generated by calling $M^*(a, 3, (f_2(f_1, f_0)))$. Each call is written next to the node it is called from. The dashed lines show what move is selected by each call.

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 tests how much node d is worth according to its 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 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 o. 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 leads to node b with a value of 7.

The formal listing of the M^* algorithm is shown in figure 5.

We claimed that M^* is a generalization of the M^n algorithm. The M^n algorithm calls minimax when n reaches 0. However, note that M^* does not contain such a call. The reason is that $minimax(f, d, board) \equiv M^*(\overbrace{(f, (-f, (f, \ldots)))}^{d})$ (For the same reasons that the NEG-MAX notation is equivalent to minimax.) Therefore, if the modeling level of the original player n is smaller than the depth of the search tree d, we replace the 0-level modeling player f_0 by $\overbrace{(f_0, (-f_0, (f_0, \ldots)))}^{d-n}$.

2.4 A one-pass version of M^*

It is obvious that the M^* algorithm does 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 to V differently for values associated with the player and values associated with the opponent. In a player's node (a node where it is the

```
Procedure M*(pos, depth, (f_pl, OPP_MODEL))
  if depth = 0
    return ⟨NIL, f_pl(pos)⟩
  else
    max_value ← −∞
    SUCC ← MoveGen(pos)
    for each succ ∈ SUCC
      if depth = 1
        player_value ← f_pl(succ)
      else
        ⟨opp_board, opp_value⟩ ← M*(succ, depth − 1, OPP_MODEL)
        ⟨player_board, player_value⟩ ← M*(opp_board, depth − 2, (f_pl, OPP_MODEL))
      if player_value > max_value
        max_value ← player_value
        max_board ← succ
    return ⟨max_board, max_value⟩
```

Figure 5: The M^* algorithm

player's turn to play)[1], for values associated with the player ($V_n, V_{n-2}, \ldots$), V_i gets the maximal V_i value among its children. For values associated with the opponent ($V_{n-1}, V_{n-3}, \ldots$) , V_i gets the V_i value of the child that gave the maximal value to V_{i-1}. For example, the opponent believes (according to the model) that the player evaluates nodes by V_{n-2}. At a player's node, the opponent assumes that the player will select the child with maximal V_{n-2} value. Therefore, the value of the current node for the opponent is the V_{n-1} value of the selected child with the maximal V_{n-2} value. At an opponent's node, we do the same but the roles of the opponent and the player are switched.

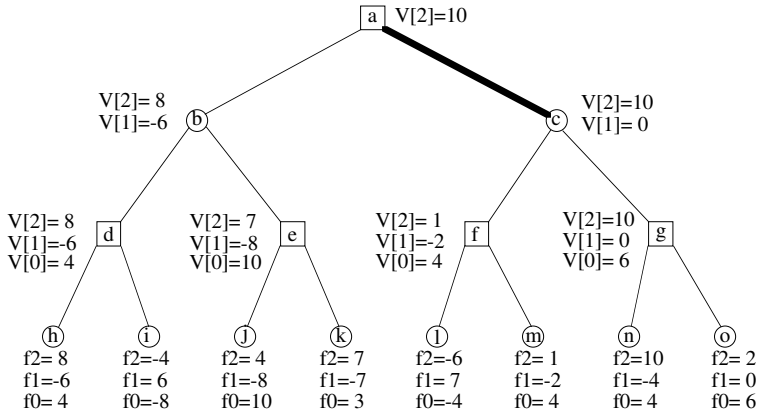

Figure 6: The value vectors propagated by M^*_{1-pass}. This is the same tree as the one shown in the previous figure.

Figure 6 shows an example for a tree spanned by M^*_{1-pass}. Let us look at node e to understand the way that the algorithm works. Three players evaluate node e: the player, the opponent model,

[1]Traditionally such a node is called a MAX node. However, we assume that both players are maximizers

and the opponent's model of the player. The opponent's model of the player evaluates all the successors using evaluation function $f0$ 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 $f1$, $(f1(j) = -8)$. The player actually prefers node k with the higher $f2$ value $(f2(k) = 7)$. Thus, the vector propagated from node e is $\langle 7, -8, 10\rangle$. Note that the values in the vectors correspond to the results of the recursive calls in figure 4.

Figure 7 lists the M^*_{1-pass} algorithm.

```
Procedure M*_{1-pass}(pos, depth, (f_n, (f_{n-1}, (..., f_0)...)))
  if depth = 0
    return ⟨f_n(pos), ..., f_0(pos)⟩
  else SUCC ← MoveGen(pos)
       V ← ⟨-∞, ..., -∞⟩
       for each succ ∈ SUCC
            succ_V ← M*_{1-pass}(succ, depth - 1, (f_n, (f_{n-1}, ..., f_0)...))
            for each i associated with current player
                 if succ_V[i] > V[i]
                    V[i] ← succ_V[i]
                    if i < n
                       V[i+1] ← succ_V[i+1]
       return ⟨V[n], ..., V[depth]⟩
```

Figure 7: M^*_{1-pass}: A version of the M^* algorithm that performs only one pass over the search tree

The following theorem shows that M^* and M^*_{1-pass} are equivalent.

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

Proof:

For $d = 0$ and $d = 1$ the proof is immediate.

Assume that M^* and M^*_{1-pass} return the same value for a tree of depth $d \leq depth - 1$. We will prove that they return the same value for $d = depth$.

1. For each successor at level 1, M^* first determines the board at level 2 selected by its opponent. By the induction hypothesis,

   $$M^*(succ, d-1, \text{OPP\_MODEL}) = M^*_{1-pass}(succ, d-1, \text{OPP\_MODEL}) = V[n-1]$$

   and therefore these will be the boards with the maximal $V[n-1]$.

2. M^* determines the value of each successor by calling itself on the board selected by the opponent. By the induction hypothesis

   $$M^*(b, d-2, PLAYER) = M^*_{1-pass}(b, d-2, PLAYER) = V[n]$$

   and therefore these M^* values will be equal to the $V[n]$ values of these nodes.

3. M^* associates with each successor the player's value of the board selected by the opponent. M^*_{1-pass} assigns to each successor the $V[n]$ value associated with the board with maximal $V[n-1]$ value. Thus, the value assigned to each successor by M^* is equal to its $V[n]$ value.

4. M^* returns the maximal value for its successors, while M^*_{1-pass} returns the maximal $V[n]$ of its successors. These are the same values. □

2.5 The complexity of M^* and M^*_{1-pass}

It looks as M^*_{1-pass} is always prefered over M^* because it expands less nodes. M^*_{1-pass} expands each node in the tree once, while M^* re-expands many nodes. However, while M^*_{1-pass} performs less *expansions* than M^*, it may perform more *evaluations*. Look for example at Figure 4. 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-pass} must apply every evaluation function for every leaf.

For analyzing the complexity of M^* and M^*_{1-pass}, let us assume that the search tree has a uniform branching factor b and a depth limit of d. The number of nodes expanded by M^*_{1-pass} is the number of internal nodes in the tree, i.e., $\frac{b^d-1}{b-1}$. M^*_{1-pass} spans b^d leaves. At each leaf it fills a vector of d elements. Therefore the number of calls to evaluation function is bounded by db^d. However, when the modeling level, ML, is smaller than d, the number of calls to evaluation functions is $(ML+1)b^d$ (the last $d-(ML+1)$ entries are just the values of $f0$ with alternating sign.)

An upper limit on the number of node expansions and calls to evaluation function in M^* is given in the following lemma.

Lemma 1 *1. The number of calls to the expansion function MoveGen by M^* is bounded by $(b+1)^{d-1}$*

2. The number of calls to evaluation functions by M^* is bounded by $(b+1)^d$

Proof:

(1) The number of calls to $MoveGen$ in M^* is one plus the number of calls in the b iterations over the successors. For each successor we call M^* twice, once with $depth = d-1$ (for the opponent simulation) and once with $depth = d-2$ (for evaluating the opponent selection) . Therefore, the number of expansions can be expressed by the following recurrence:

$$T_{ex}(b,d) = \begin{cases} 0 & \text{if } d = 0 \\ 1 & \text{if } d = 1 \\ 1 + b\,[T_{ex}(b,d-1) + T_{ex}(b,d-2)] & \text{otherwise} \end{cases} \tag{4}$$

We prove by induction that $T_{ex}(b,d) \leq (b+1)^{d-1}$. The inequality holds for $d = 0$ and $d = 1$. Assume that it holds for every $n < d$.

$$\begin{aligned} T_{ex}(b,d) &= 1 + b\,[(T_{ex}(b,d-1) + T_{ex}(d-2)] \\ &\leq 1 + b\,[(b+1)^{d-2} + (b+1)^{d-3}] \\ &= 1 + b\,[(b+2)(b+1)^{d-3}] \\ &\leq (b+1)^2(b+1)^{d-3} \\ &= (b+1)^{d-1} \end{aligned}$$

(2) For each of the b successors generated for the current position the algorithm is recursively called twice: Once to simulate the opponent with depth $d-1$ and once to evaluate the resulting board with depth $d-2$. Therefore, the number of calls to evaluation functions by M^* for such a tree with depth d can be measured by the following recurrence :

$$T_{ev}(b,d) = \begin{cases} 1 & \text{if } d = 0 \\ b & \text{if } d = 1 \\ b\left[T_{ev}(b,d-1) + T_{ev}(b,d-2)\right] & \text{otherwise} \end{cases} \tag{5}$$

We prove by induction that $T_{ev}(b,d) \leq (b+1)^d$. The inequality holds for $d = 0$ and $d = 1$. Assume that it holds for every $n < d$.

$$\begin{aligned} T_{ev}(b,d) &= b\left[T_{ev}(b,d-1) + T_{ev}(b,d-2)\right] \\ &\leq b\left[(b+1)^{d-1} + (b+1)^{d-2}\right] \\ &= b(b+2)(b+1)^{d-2} \\ &\leq (b+1)^2(b+1)^{d-2} \\ &= (b+1)^d \end{aligned}$$

□

The above analysis 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 player P with $ML(P) = 1$, M^* expands 1.3 times more nodes than M^*_{1-pass} but M^*_{1-pass} performs 1.5 times more calls to evaluation functions.

2.6 The relationship between M^* and *MAXN*

The *MAXN* algorithm [13] 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 possible to extend M^* and M^*_{1-pass} to n-player games. In such a case, each player will have a vector of models, one for each of the opponents.

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*, in each level of the search tree, only one player has the 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 6. Table 1 shows the steps taken by each of the algorithms at node e.

We can use M^* to simulate *MAXN* for the case of two players by using the player $P1 = (f1,(f2,(f1,(f2,\ldots))))$.

MAXN	M^*_{1-pass}
At node e, it is player3's turn to move	At node e, it is the player's turn to play
Player3 uses function f0 to evaluate boards	The opponent's model of the player uses function f0 to evaluate boards
Player3 selects board j (with a value of 10)	The opponent's model of the player selects board j (with a value of 10)
Player2 evaluates node j using $f1$, $(f1(j) = -8)$) and propagates this value to e	The opponent model evaluates node j using $f1$, $(f1(j) = -8)$) and propagates the value to e
Player1 knows that it has no effect on the decision in e	*The player knows that it is his turn to move in e, therefore it can decide what move will be **actually** taken*
Player1 evaluates node j using $f2$, $(f2(j) = 4)$) and moves this value to e	The player evaluates both j and k using $f2$, decides that k is a better node and that it will actually select k, and moves $f2(k) = 7$ to node e
The vector propagated up is $\langle 10, -8, 4 \rangle$	The vector propagated up is $\langle 10, -8, 7 \rangle$

Table 1: The difference in the behavior between *MAXN* and M^*_{1-pass} in node e in Figure 6.

2.7 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.

Lemma 2 *Assume that M^* and Minimax use the same f_{player}. Then*

$$Minimax(pos, depth, f_{player}) \leq M^*(pos, depth, (f_{player}, OPP\_MODEL)) \quad (6)$$

for any OPP_MODEL.

Proof:

We will prove that this property exists 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 the version of M^*_{1-pass}.

For $depth = 0$, for any OPP_MODEL

$$Minimax(pos, depth) = f_{player}(pos) = M^*_{1-pass}(pos, depth, (f_{player}, \text{OPP\_MODEL}) \quad (7)$$

Let us assume correctness for $d \leq depth$ and prove it for $d = depth + 1$
If *pos* is a player's node,

$$Minimax(pos, depth + 1) = \max\{Minimax(succ, depth) \mid succ \in \text{SUCC}(pos)\} \quad (8)$$

According to the inductive assumption,

$$\leq \max\left\{M^*_{1-pass}(succ, depth, (f_{player}, \text{OPP\_MODEL})) \mid succ \in \text{SUCC}(pos)\right\}$$
$$= M^*_{1-pass}(pos, depth + 1, (f_{player}, \text{OPP\_MODEL}))$$

If pos is an opponent's node,
let s be the successor with the maximal value according to the opponent's model.

$$(9) \qquad Minimax(pos, depth+1) = \min\{Minimax(succ, depth) \mid succ \in \mathrm{SUCC}(pos)\}$$

$$\leq Minimax(s, depth)$$

According to the inductive assumption,

$$\leq M^*_{1-pass}(s, depth, (f_{player}, \mathrm{OPP\_MODEL}))$$
$$= M^*_{1-pass}(pos, depth+1, (f_{player}, \mathrm{OPP\_MODEL})) \quad \square$$

The intuition behind the above lemma is that minimax assumes an adversary model of the opponent. It assumes that the opponent knows the player's strategy, and its only interest is to select moves that are worst according to the player's strategy. The lemma says, that if you have a *good* reason to believe that your opponent's model is different than that, you could only benefit by using M^* instead of minimax. The reader should note that the above lemma 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.

2.8 Incorporating depth of search into the model

The M^* algorithm as well as minimax (which is a special case of M^*), incorporates an implicit assumption about the depth to which the opponent searches. M^* that searches a tree of depth d, assumes that its opponent searches to level $d-1$, and then assumes that its opponent assumes that the player searches to level $d-2$, etc. This is a potentially wrong assumption, but it is a good defensive mechanism. The player assumes that its opponent searches as deep as the player can simulate the opponent's search. It is meaningless for the player to assume that its opponent searches to a deeper level since it can not simulate a search to such depth. However, as was discussed in the introduction, if the player has reason to believe that its opponent searches to a lesser depth, then it can utilize this belief against the opponent (to set traps, for example).

But why does the player have to search to level d_1 if it knows that its opponent searches to level $d_0 \ll d_1$? In order to predict the opponent's selection, it is indeed enough to simulate its search to level d_0. However, in order to *evaluate* the merit of the opponent's selection for the player, it searches as deep as it can.

We have extended M^* to handle models of depth of search. We define a strategy to be a pair (f_{player}, d_{player}). A *player* is then a pair $((f_{player}, d_{player}), \mathrm{MODEL})$ where MODEL is also a player. The extended algorithm is different from its simpler version on one point. Whereas the simple algorithm allocates as many resources as possible to the call that simulates the opponent's search (which is the current depth allocation minus one), the extended algorithm allocates the simulation as many resources as it believes that the opponent would have used. Of course, regardless of the model of its opponent's depth of search, the player should never exceed the total depth limit allocated for the procedure . Figure 8 shows an example of applying M^* with a player that searches to depth 3 and an opponent model that searches to depth 1.

Figure 9 lists the extended M^* algorithm.

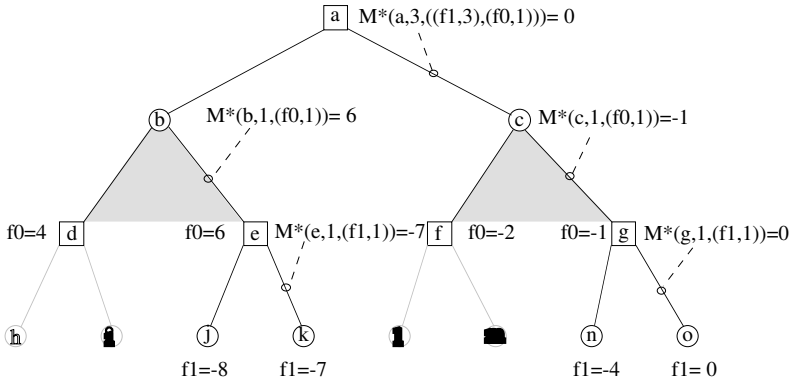

Figure 8: The set of recursive calls generated by calling $M^*(a, 3, ((f_1, 3), (f_0, 1)))$. Each call is written next to the node it is called from. The dashed lines show what move is selected by each call.

As in the former case, this algorithm can also be written in a one-pass form so that the game tree will be expanded one time only. The algorithm propagates vectors of values where each entry is associated with one of the models as before. But while the simplified non-recursive algorithm carries one value per model, the extended algorithm carries a list of values for each model. The $j'th$ element of the $i'th$ list is the value associated with a search to depth j by the $i'th$ model. In the original M^*, there was only one search frontier. Therefore, the non-recursive version propagated only one value for every model. However, the extended M^* can call itself recursively from any node to any depth. Hence, every level in the search tree is potentially a search frontier of one of those recursive calls. That is the reason why the extended non-recursive algorithm carries a list of values for each model. Figure 10 shows the value matrices propagated up the tree when calling M^*_{1-pass} on the same tree as in figure 8.

The extended one-pass version is shown in figure 11.

The original M^* algorithm, called with $(f_n, (f_{n-1}, (\ldots, f_0)\ldots))$, is equivalent to calling the extended M^* with $((f_n, d), ((f_{n-1}, d-1)\ldots, (f_0, 0)\ldots))$ (recall that if n, the level of modeling is less than the depth of search, we replace the zero-level player by a minimax player to make the top-level player a d-level modeler). In particular, we can get the minimax algorithm by calling the extended algorithm with $((f, d), ((-f, d-1), ((f, d-2), \ldots)\ldots))$. There is one difference in the way that we extend the zero-level player for the case where $n < d$. In the case of the extended algorithm, we replace the zero-level model by $((f_0, d_0), ((-f_0, d_0 - 1)\ldots, (f_0, 0)\ldots))$.

3 Experimental study: The potential benefit of using opponent models

Now that we have developed an algorithm for using opponent models, we would like to evaluate the potential benefit of using this algorithm. We conducted a set of experiments with the M^* algorithm to test the effect of various parameters on the benefit of using an opponent model.

3.1 Experimentation methodology

In order to make the experimentation more feasible, we limited the experiments to one-level modeling players, such that the player possesses a model of its opponent's strategy (an evaluation

```
Procedure M*(pos, depth, ((f_pl, d_pl), OPP_MODEL))
  if depth = 0
    return ⟨NIL, f_pl(pos)⟩
  else
    max_value ← −∞
    SUCC ← MoveGen(pos)
    for each succ ∈ SUCC
      if depth = 1
        player_value ← f_pl(succ)
      else
        ⟨opp_board, opp_value⟩ ← M*(succ, min(depth − 1, d_opp), OPP_MODEL)
        ⟨player_board, player_value⟩ ← M*(opp_board, depth − 2, ((f_pl, d_pl), OPP_MODEL))
      if player_value > max_value
        max_value ← player_value
        max_board ← succ
    return ⟨max_board, max_value⟩
```

Figure 9: An extended version of the M^* algorithm that can handle a model of depth

function and a depth of search), and assumes that its opponent is a standard minimax player. Furthermore, the actual opponent used for our experimentation was indeed a minimax player. In order to evaluate the algorithms, we also allowed a regular minimax player to play against the same opponent.

Therefore, the experiments described in this section involve three players[2]:

$$\begin{aligned} MSTAR &= ((f_{player}, d_{player}), (f_{model}, d_{model})) \\ MINIMAX &= ((f_{player}, d_{player}), NIL) \\ OPPONENT &= ((f_{opponent}, d_{opponent}), NIL) \end{aligned}$$

[2]Remember that M^* expands these players to d-level modeling players using the opposite function and a decreasing series of depth.

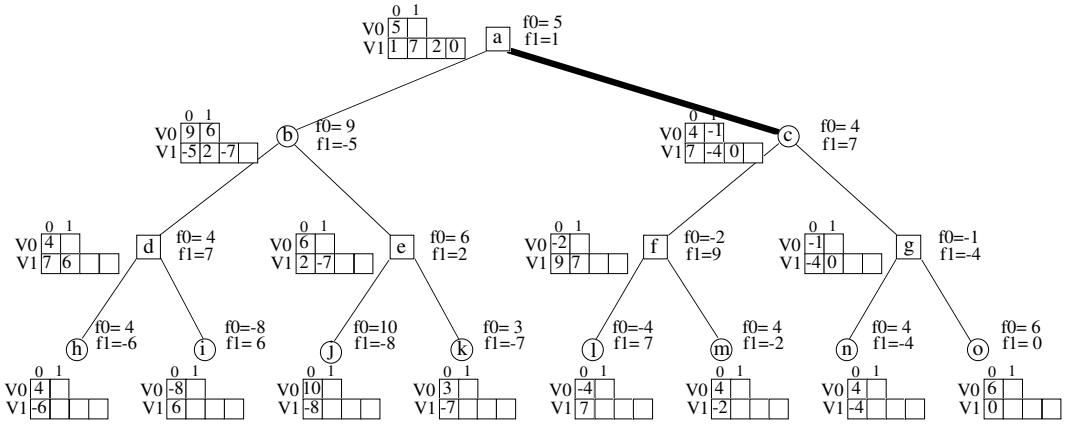

Figure 10: The value matrices propagated by $M^*_{1-pass}(a, 3, ((f_1, 3), (f_0, 1)))$.

```
Procedure M*_{1-pass}(pos, depth, ((f_n, d_n), ((f_{n-1}, d_{n-1}), (..., (f_0, d_0))...)))
    loop for i = n downto 0
        V[i] ← ⟨-∞, ..., -∞, f_i(pos)⟩
    if depth = 0
      return ⟨V[n], ..., V[0]⟩
    else SUCC ← MoveGen(pos)
        for each succ ∈ SUCC
            succ_V ← M*_{1-pass}(succ, depth - 1, ((f_n, d_n), ((f_{n-1}, d_{n-1}), (..., (f_0, d_0))...)))
            for each i associated with current player
                d = min(d_i, depth)
                for j = 1 to d
                    if succ_V[i][j-1] > V[i][j]
                      V[i][j] ← succ_V[i][j-1]
                      if j = d and i < n
                        for k = 1 to min(d_{i+1}, depth)
                            V[i+1][k] ← succ_V[i+1][k-1]
        return ⟨V[n], ..., V[depth]⟩
```

Figure 11: An extended non-recursive version of the M^* algorithm that can handle a model of depth

None of the players used pruning, which could not effect the results due to the fixed depth search that was performed. We measure the performance of a player by conducting a tournament with *OPPONENT*. We report here the results with respect to one dependent variable

Mean-points-per-game The player gets 2 points for a win, 1 point for a draw and zero points for a loss. The variable is the sum of points earned for the whole tournament divided by the number of games.

We measured the performance using other criteria, like the piece advantage at the end of the game, or numbr of wins, but we did not find significant difference between these measurements.

The experiments were conducted using the domain of checkers. This game was chosen because it is complicated enough to conduct realistic experiments, yet simple enough to perform significant number of games. We used evaluation functions based on that used for Samuel's checkers player[16].

A limit of 100 moves per game was set during the tournament. Therefore it is possible that although one algorithm is stronger than another, it can not win in 100 moves, and therefore finish most games in a draw. When the advantage of the stronger algorithm increases, it may win more games in 100 moves, thus increasing its mean-points-per-game.

In the experiments described in this section, we studied the effect of the following independent variables:

depth-difference The difference between the depth of search used by the player and that used by *OPPONENT*.

function-difference The difference in quality between the evaluation function used by the player and that used by *OPPONENT*.

depth-error The difference between the depth of search used by *MSTAR* for the model of its opponent, and that actually used by *OPPONENT*.

function-error The difference between the function used by *MSTAR* for the model of its opponent, and that actually used by *OPPONENT*.

In order to be able to measure the function quality and the function error, we defined the following functions:

$$\mathcal{F}_{f,a}(x) = \begin{cases} f(x) & \text{if } |(f(x)| < a \\ 0 & \text{otherwise} \end{cases} \tag{10}$$

For a given evaluation function f, we can create a sequence of functions by varying a. As a increases, the quality of the function increases. Therefore, we measure the distance in quality between $\mathcal{F}_{f,a_1}$ and $\mathcal{F}_{f,a_2}$ by $a_1 - a_2$.

An experiment was conducted by varying the value of one of the independent variables and fixing the values of the rest. For each value of the tested variable, we measured the performance of the player by letting it play a set of 100 games against *OPPONENT*.

3.2 Comparing M^* to *MINIMAX*

The experiments described in this subsection demonstrate the potential advantage of using M^* with a correct opponent model over Minimax. In the first experiment the player searches to depth 4 while the opponent searches to depth 2. *MINIMAX* assumes that *OPPONENT* searches to depth 3 while M^* knows that the opponent searches to depth 2. The results of this experiment are shown in the following table.

	Wins	Draws	Losses	Points
MINIMAX vs. *OPPONENT*	340	455	5	1135
MSTAR vs. *OPPONENT*	416	379	5	1211

In the second experiment all players searched to depth 4. *MINIMAX* and M^* used one function while *OPPONENT* used another. M^* knows its opponent's function while *MINIMAX* implicitly assumes that its opponent uses the opposite of its own function. The following table shows the results obtained.

	Wins	Draws	Losses	Points
MINIMAX vs. *OPPONENT*	94	616	90	804
MSTAR vs. *OPPONENT*	126	618	56	870

In the third experiment all players searched to depth 4. The three players use the same evaluation function

$$f_1(board, player) = Material(board, player) - 0.004 \cdot TotalFigures(board)$$

However, while M^* knows its opponent's function, *MINIMAX* implicitly assumes that its opponent uses the function

$$\begin{aligned} f_1(board, opponent) &= -f_0(board, player) \\ &= -Material(board, player) + 0.004 \cdot TotalFigures(board) \\ &= Material(board, opponent) + 0.004 \cdot TotalFigures(board) \end{aligned}$$

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

	Wins	Draws	Losses	Points
MINIMAX vs. *OPPONENT*	171	461	168	803
MSTAR vs. *OPPONENT*	249	357	194	855

All three experiments in this section show the advantage of the modeling approach over the minimax approach when using the correct model. In the next sections we test the effect of model error and difference in playing ability on this advantage.

One may argue that since M^* does not prune, it may lose its advantage over *MINIMAX* if we allow *MINIMAX* to use $\alpha\beta$ prunning and allocate both players equivalent time. In Section 5 we show how prunning can be performed in M^*.

3.2.1 The effect of the difference in playing ability on the benefit of modeling

The purpose of this experiment is to test the effect of the difference between the players' ability on the benefit of modeling. The basic hypothesis tested is that as the ability difference increases (in favor of the player), the potential merit of using the minimax assumption decreases. A stronger player, who knows that its opponent is weak, can benefit from this knowledge.

We conducted two experiments. In the first one we kept the evaluation function fixed and varied the depth difference. In the second we kept the depth difference fixed and varied the difference between the quality of the evaluation functions. *MSTAR* knew the strategy of *OPPONENT* and used M^* while *MINIMAX* used its own strategy as a model for *OPPONENT*.

For the experiment with the depth difference, we varied d_{player} from 2 to 8, and $d_{opponent}$ from 2 to d_{player}. For each combination of d_{player} and $d_{opponent}$, we conducted one tournament between M^* and *OPPONENT* and another tournament between *MINIMAX*, with d_{player} and f_{player}, and *OPPONENT*.

The left graph in Figure 12 shows the results obtained for depth difference. For each such combination of d_{player} and $d_{opponent}$ we measure the mean-points-per-game obtained by M^* and by *MINIMAX*. We subtract the second from the first and obtain the benefit of M^* over *MINIMAX*. This benefit is averaged over all the tournaments with the same $d_{player} - d_{opponent}$. Each point i on the X axis represents all the tournaments where $d_{player} - d_{opponent} = i$. The Y value is the average benefit for that i.

The right graph in Figure 12 shows the results obtained for function difference. Both M^* and *MINIMAX* used the same evaluation function f_{player}. *OPPONENT* used the function $f_{opponent} = \mathcal{F}_{f,a}$. M^* used $f_{model} = f_{opponent}$. The X axis stands for the a factor that was used to deteriorate the function quality. When it increases, so does the difference between f_{player} and $f_{opponent}$. The Y axis measures, as before, the benefit of using *MSTAR* over *MINIMAX*.

Both experiments exhibited similar behavior. The advantage of *MSTAR* over *MINIMAX* increased with the difference in playing ability between their strategy and *OPPONENT*'s strategy up to a certain point where the advantage started to decline. The increase in the benefit can be explained by the observation that *MINIMAX* was too careful in predicting its opponent's moves, while *MSTAR* utilized its model and exploited the weaknesses of its opponent to its advantage. The decline in higher differences can be explained by the observation that when the difference in playing ability becomes larger, *MINIMAX*, as well as *MSTAR*, win in almost all games. In such a case there is a little place for improvement by modeling.

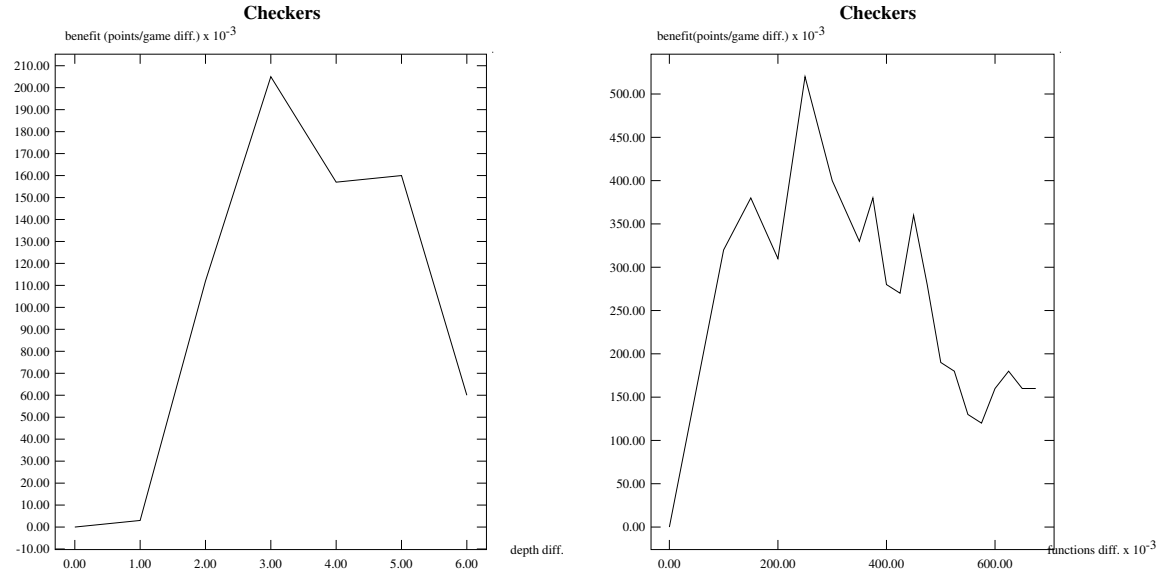

Figure 12: The benefit of knowing the opponent strategy as a function of the playing ability difference. Measured by difference in mean points per game. The left graph shows the benefit of using M^* over minimax as a function of the difference in depth search between the player and the opponent. The right graph shows the benefit as a function of the difference in the quality of the evaluation function.

The experiments with the function difference did not test the situation where the two opponents have similar playing ability but use different evaluation functions. In Section 6.3 we show that M^* has advantage also in such a situation.

3.2.2 The effect of the modeling error on the player's performance

The previous experiments demonstrated the benefit of using opponent model. In the following experiments we tested the risk of using a wrong model. We conducted two experiments in the domain of checkers. For the first experiment, we fixed $f_{player} = f_{model} = f_{opponent}$ and varied the values of d_{player} and $d_{opponent}$ in the range $[2, 9]$ and d_{model} in the range $[1, d_{player} - 1]$. For the second experiment, we fixed the depth parameters of the strategies $d_{player} = d_{opponent} = 6, d_{model} = 5$ and varied the functions f_{player}, f_{model} and $f_{opponent}$ by varying the a parameter of the $\mathcal{F}_{f,a}$ (see eq. 10) functions in the range $[0, 1]$.

A tournament of 100 games between the player and the opponent was conducted for each combination of values. The performance of the player is measured by mean-points-per-game as before. Figure 13 shows the results obtained. The left part of the figure shows the results obtained for the depth error. A graph labeled j stands for all cases where $d_{player} - d_{opponent} = j$. For example, the graph marked by 0 represents the accumulated results for the cases where $d_{player} = d_{opponent}$ and the graph marked by 2 is for the cases where $d_{player} - d_{opponent} = 2$ (4-2,5-3,6-4,7-5,8-6,9-7). Every point x on the X axis stands for all cases where $d_{opponet} - d_{model} = x$. For example, the point $x = 3$ for the graph labeled 2 represents the tournaments with the combinations $(d_{player}, d_{opponent}, d_{model}) \in \{(6, 4, 1), (7, 5, 2), (8, 6, 3), (9, 7, 4)\}$. The results for the function error, shown in the right part of the figure, were accumulated similarly. A graph labeled j stands for all cases where $a_{player} - a_{opponent} = j$ and every point x on the X axis stands for all cases where $a_{opponet} - a_{model} = x$.

Both experiments show similar behavior. The benefit of using an opponent model is maximal

for the case of no error. When the error increases, the benefit of using the model decreases. Overestimation (negative error) is less harmful than underestimation.

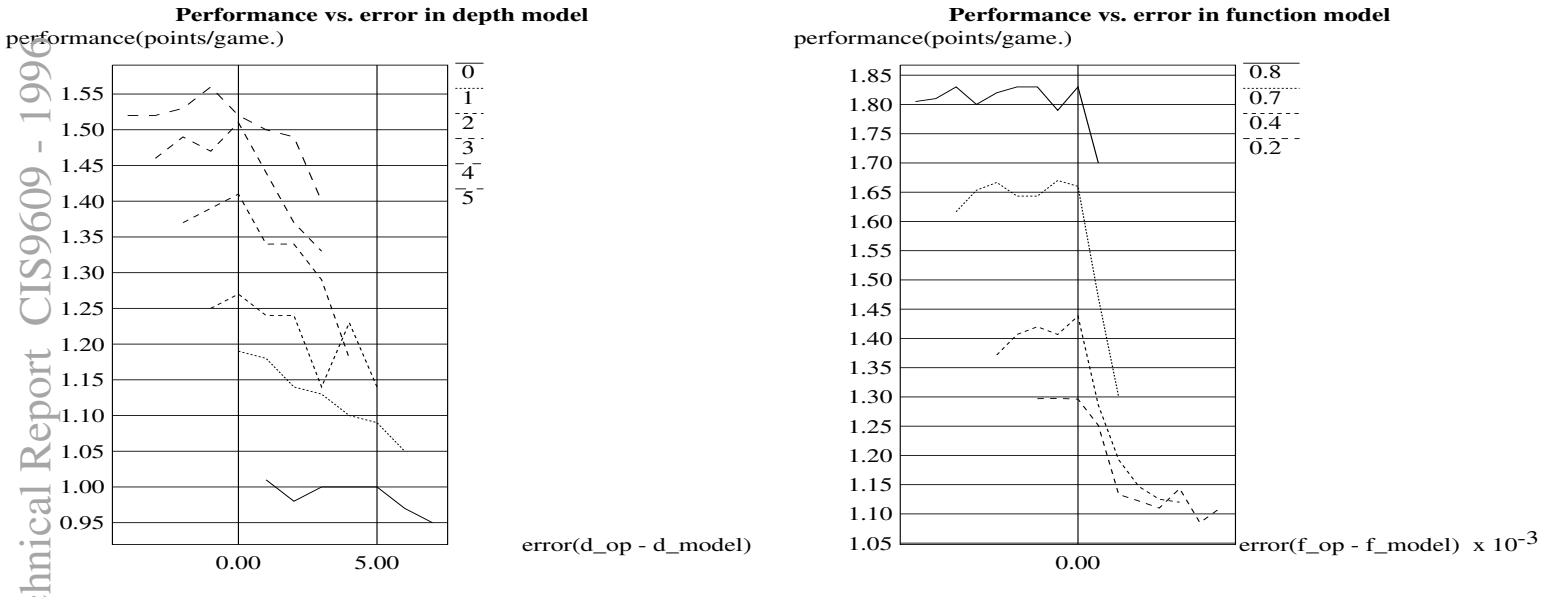

Figure 13: The performance of M^* as a function of the modeling error. The left graph shows the performance of M^* as a function of the error in depth model for various values of $d_{player} - d_{opponent}$. The right graph shows the performance of M^* as a function of the error in function model for various values of $f_{player} - f_{opponent}$.

3.3 Summary

The experiments described in this section confirmed the hypothesis that it is beneficial to use an opponent model, and that the benefit is greater against weaker opponents. It also demonstrates the harmfulness of using wrong model. In the next section we describe an algorithm for using a wrong model with a bounded error.

4 Using uncertain models

In the previous sections, we assumed that the player is certain about its opponent model. However, it is quite possible that a player is uncertain about its opponent model, especially when the model is learned by the player. In this section, we generalize M^* to a new algorithm, M^*_ϵ that can handle uncertain models.

There are several possibilities for representing uncertainty about the model. We assume that the player possesses an upper bound on the distance between the model's evaluation function and the actual opponent's function. This model of uncertainty is quite common and is realistic when the player tries to learn the opponent's function based on its past moves (We show such learning algorithm in Section 6). It is feasible to enhance this algorithm with a procedure for estiamting a bound on its error.

The M^*_ϵ algorithm assumes that the player possesses, in addition to a model of its opponent's function, an upper bound, ϵ, on the distance between the model function and the actual opponent's function. Therefore, a player is now defined as $((f, \epsilon), MODEL)$, where MODEL is also a player.

This means that each evaluation function that appears in a player (the player's function, its model's function, its model's model function etc.), has an associated bound on its error. Such a player will be called an *uncertain* player, while a player with no error bound, such as the one used by M^*, will be called a *certain* player. Since the highest level player is certain about its own function, the error bound associated with the top level function will be zero.

A player $((f_1, 0), (f_0, \epsilon_0))$ assumes that the opponent uses a function $\hat{f}_0$ that satisfies

$$\forall x \left[f_0(x) - \epsilon_0 \leq \hat{f}_0(x) \leq f_0(x) + \epsilon_0 \right]$$

The error bounds associated with the model's functions represent the player's uncertainty about its opponent. It is possible that the opponent is also an uncertain player. In such a case, its functions will have associated error bounds. We define the error bounds associated with the model's functions to apply to the range determined by the opponent's error bounds. For example, a player $((f_2, 0), ((f_1, \epsilon_1), (f_0, \epsilon_0)))$ assumes that its opponent is $((\hat{f}_1, 0), (\hat{f}_0, \hat{\epsilon_0}))$ where

$$\forall x \left[f_1(x) - \epsilon_1 \leq \hat{f}_1(x) \leq f_1(x) + \epsilon_1 \right] \tag{11}$$

$$\forall x \left[\hat{f}_0(x) - \hat{\epsilon_0}, \hat{f}_0(x) + \hat{\epsilon_0} \right] \subseteq [f_0(x) - \epsilon_0, f_0(x) + \epsilon_0] \tag{12}$$

The input for the M^*_ϵ algorithm is the same as for the M^* algorithm, but with a player that fits our extended definition. The output is different. Instead of returning a board and a value, the new algorithm returns a set of boards and a range of values. The meaning of the range is, that if we would have run M^* with any set of functions that satisfy the error constraints, we would have received a value that falls within the returned range. For every board in the set of returned boards, there is a set of functions satisfying the error constraints, for which M^* would have returned that board as the selected move.

The algorithm generates all the successors and calls itself recursively with the opponent model to determine which *set* of moves (boards) the opponent can choose from for each of the successors. For each board in such a set of boards, the player calls itself recursively to determine the range of values that the board is worth for the player. Since the player does not know which board of the set will actually be selected by the opponent, it takes the union of these ranges as the range of values that the successor is worth for the player. In this stage, the player has an associated range of values for each of its alternative moves. The lower bound of the range returned by the algorithm is the maximal minimums of all these ranges. The reason is that even with the worst possible set of functions satisfying the error constraints, the player is guaranteed to have at least the maximal minimums. The upper bound is the maximal maximums of all ranges since none of the boards can have a value which exceeds this maximum. The set of boards returned is the set of all boards that can have a maximal value. If the highest value of a range associated with a board is less than one of the minimums, there is no possibility that this board will be selected (the other board is guaranteed to have a higher value).

The algorithm returns a set of boards with a range of values. In order to select a move, a player can employ various selection strategies. A natural candidate is *maximin* [12, page 275-326], a strategy that selects the board with maximal minimum.

Look at the left tree in Figure 14. The opponent selects a move at node b. The player knows that the value that the opponent assigns to node d is in the range $[-1, 1]$, the one for node e is in $[7, 9]$ and the one for f is in the range $[9, 11]$. There are no circumstances under which the opponent will select node c, since its maximum is smaller than the minimum of e and the minimum of f. It is possible that node e would be selected (if $f_0(e) = 9$ and $f_0(f) = 9$. There is also a possibility

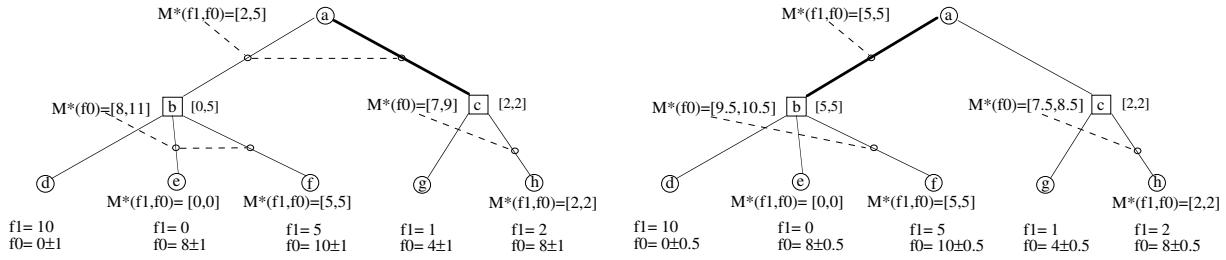

Figure 14: An example for the sequence of calls produced by M^*_ϵ. The top-left figure shows the calls for the case of $\epsilon = 1$ while the right one is for the case of $\epsilon = 0.5$.

that node f will be selected (if $f_0(f) > 9$). Therefore the player concludes that the opponent may select either node e or node f. The minimal possible value that can be assigns to node b by the opponent is 9. The maximal possible value is 11. Therefore node b worth for the opponent a value in $[9, 11]$. Node e worth 0 for the player and node f worth 5, therefore node b worth to the player a value in $[0, 5]$. Similar reasoning shows that node c worth $[7, 9]$. Any rational decision algorithm, including maximin, would select node c.

Figure 15 shows the M^*_ϵ algorithm.

Procedure $M^*_\epsilon\,(pos, depth, ((f_{pl}, \epsilon_{pl}), \text{OPP\_MODEL}))$
  if $depth = 0$
    return $\langle NIL, [f_{pl}(pos) - \epsilon_{pl}, f_{pl}(pos) + \epsilon_{pl}]\rangle$
  else
    $SUCC \leftarrow MoveGen(pos)$
    for each $succ \in SUCC$
      if $depth = 1$
        $succ\_range \leftarrow [f_{pl}(succ) - \epsilon_{pl}, f_{pl}(succ) + \epsilon_{pl}]$
      else
        $\langle opp\_boards, opp\_range\rangle \leftarrow M^*_\epsilon\,(succ, depth - 1, \text{OPP\_MODEL})$
        for each $board \in op\_boards$
          $\langle pl\_boards, pl\_range\rangle \leftarrow$
            $M^*_\epsilon\,(board, depth - 2, ((f_{pl}, \epsilon_{pl}), \text{OPP\_MODEL}))$
          $succ\_ranges \leftarrow succ\_ranges \cup \{pl\_range\}$
        $succ\_range \leftarrow [\min(i), \max(j)]_{[i,j]\in succ\_ranges}$
      $root\_ranges \leftarrow root\_ranges \cup \{succ\_range\}$
    $[root_{min}, root_{max}] \leftarrow [\max(i), \max(j)]_{[i,j]\in root\_ranges}$
    $root\_boards \leftarrow \{b \in SUCC \mid b_{max} \geq root_{min}\}$
    return $\langle root\_boards, [root_{min}, root_{max}]\rangle$

Figure 15: The M^*_ϵ algorithm

There are other adversary search algorithms [3, 14] that return a range of values, as does M^*_ϵ. However, these algorithms were designed for different purposes. The B* algorithm [3] returns a range of values due to uncertainty associated with the player's evaluation function. Nevertheless, unlike the M^*_ϵ algorithm, B* adapts the basic zero-sum assumption and propagates values in the

same manner as does minimax. The conspiracy numbers algorithm [14] also manipulates ranges of values. However, these ranges provide a heuristic measure of the accuracy of the minimax root values of incomplete subtrees.

It is easy to see that M^*_ϵ is a generalization of M^*. To get M^* we only need to call M^*_ϵ with all error bounds equal to zero[3]. Furthermore, we can prove a stronger relationship between the two algorithms. The following theorem states that if we call M^* with a certain player whose functions fall within the error bounds of an uncertain player, then it will return a value that is within the bounds returned by M^*_ϵ for the uncertain player and it will return a move that is a member in the set of moves returned by M^*_ϵ for the uncertain player.

Theorem 2 *Let P be an* uncertain *player. Let $\langle B, [i,j]\rangle = M^*_\epsilon(pos, depth, P)$. Let P_c be a* certain *player consisting of arbitrary functions that satisfy the error constraints of P. Let $\langle b, value\rangle = M^*(pos, depth, P_c)$. Then $value \in [i,j]$ and $b \in B$*

Proof:

By induction on d, the depth of the search tree. For $d = 0$ and $d = 1$ the proof follows immediately.

Assume that the theorem is true for $d < k$. We will show that it is true for $d = k$. According to the first inductive assumption, for any $succ \in SUCC(pos)$, $M^*(succ, k-1, OP\_MODEL(P_c))$ returns a board b that belongs to B, the group of boards returned by $M^*_\epsilon(succ, k-1, OP\_MODEL(P))$. According to the second inductive assumption, for any $b \in B$, $M^*(b, k-2, P_c)$ returns v, a value in $[i,j]$, the range returned by $M^*_\epsilon(b, k-2, P)$. For any $succ \in SUCC(pos)$, M^*_ϵ returns a range $[a_1, a_2]$ such that a_1 is the minimal value of all the ranges of boards in B, and a_2 is the maximal value of all these ranges. Therefore, $value$, the M^* value of b that is associated with $succ$, belongs to the range that M^*_ϵ associates with $succ$. Finally, M^* returns the maximal value among the $succ$ values, and in the same manner as for $depth = 1$, this value belongs to the range returned by M^*_ϵ, and all boards with this M^* value, will belong to the group of boards returned by M^*_ϵ. □

It is interesting to note that if we call M^*_ϵ with a model that has an arbitrary opponent function and infinite error bounds, and with the *maximin* selection strategy, we actually get a minimax player. Thus, while in section 2 we interpreted minimax as a player who assumes that its opponent uses its own function (with opposite sign), here we interpret minimax as a player who has no model of its opponent and is therefore totally uncertain about its reactions. Whenever M^*_ϵ simulates the opponent, all successor's boards will be returned since none of them can be excluded. The player will then compute its values for these boards and will select the one with maximal minimum value, which is exactly what minimax does.

Figure 14 shows an example of two calls for M^*_ϵ on the same search tree, once with $\epsilon = 1$ and once with $\epsilon = 0.5$. In the case of $\epsilon = 1$, M^*_ϵ selects the same move as minimax does. In the case of $\epsilon = 0.5$, M^*_ϵ selects the same move as M^* does.

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 about $\sqrt{b}$ [7].

[3] In fact, M^* obtained by a specialization of M^*_ϵ has advantage over the original M^*. In the original M^*, we did not handle the case where a node has successors with equal values. M^*_ϵ called with zero error bound will correctly assume the worst case for opponent nodes with more than one possible outcome, while M^* would have un-justifiably returned the first.

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_{player} and f_{opp_model}, it is easy to show that such a procedure cannot exist. Figure 16 illustrates a situation where using minimax with evaluation function f_1 (or $M^*$with the player $(f_1, (-f_1, NIL))$) can avoid searching node g, whereas M^* that uses f_0 instead of $-f_1$, cannot perform this pruning. Knowing that the opponent will have at least -5 for node c does not have any implications on the value of node c for the player.

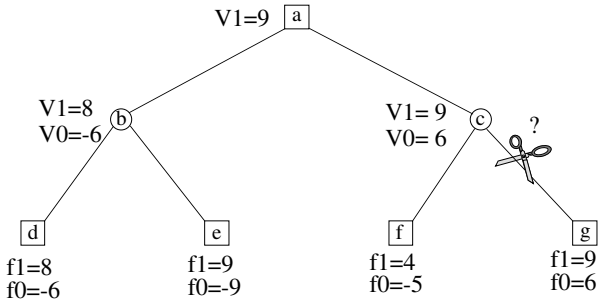

Figure 16: An example for a search tree where the standard $\alpha\beta$ would have pruned the branch leading to node g. However, such pruning would change the M^* value of the tree.

A similar situation arises in multi-player game trees. Luckhardt and Irani [13] describe a search algorithm, $Maxn$, for multi-player games and conclude that pruning is impossible without further restrictions about the players' evaluation functions. Korf [10] 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.

The basic assumption used for the original $\alpha\beta$ algorithm is that $f_{player} + f_{opponent} = 0$ (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 $|f_{player} + f_{opponent}| \leq b$. This assumption means that while f_{player} and $-f_{opponent}$ 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^*_{1-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 3 *Assume that A is a node in the search tree spanned by M^*_{1-pass}. Assume that $S_1, \ldots, S_k$ are its successors. If there exist non-negative bounds $B_0, \ldots, B_n$, such that for each successor S_j, and for each model i, $|V_{S_j}[i] + V_{S_j}[i-1]| \leq B_i$. Then, for each model $1 \leq i \leq n$:*

$$|V_A[i] + V_A[i-1]| \leq B_i + 2 \cdot B_{i-1}. \tag{13}$$

Proof:

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

$$|V_A[i] + V_A[i-1]| \leq B_i \leq B_i + 2 \cdot B_{i-1}. \tag{14}$$

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_k. It follows that

$$B_{i-1} \geq V_{S_k}[i-1] + V_{S_k}[i-2] \geq V_{S_k}[i-1] + V_{S_j}[i-2]. \tag{15}$$

It is easy to show that for any successor S,

$$|V_S[i] - V_S[i-2]| \leq B_i + B_{i-1}. \tag{16}$$

Summing up the two inequalities for the S_j successor, we get

$$B_i + 2 \cdot B_{i-1} \geq V_{S_j}[i] + V_{S_k}[i-1] = V_A[i] + V_A[i-1]. \tag{17}$$

For the second side of the inequality,

$$V_A[i] + V_A[i-1] = V_{S_j}[i] + V_{S_k}[i-1] \geq V_{S_k}[i] + V_{S_k}[i-1] \geq -B_i \geq -B_i - 2 \cdot B_{i-1}. \ \square \tag{18}$$

5.1 The $\alpha\beta^*$ pruning algorithm

Based on lemma 3, 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 α_i. It returns the M^* value of the root by only searching nodes that might affect this value.

The algorithm works similarly to the original $\alpha\beta$ algorithm, but is much more restricted in what 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 get 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 3[4].

A cutoff value α_i for a node v is the highest current value of all the ancestors of v from the point of view of player i. α_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, α_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.

Figure 17 shows a search tree similar to the one in figure 16 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 18 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 just like in Figure 17. 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

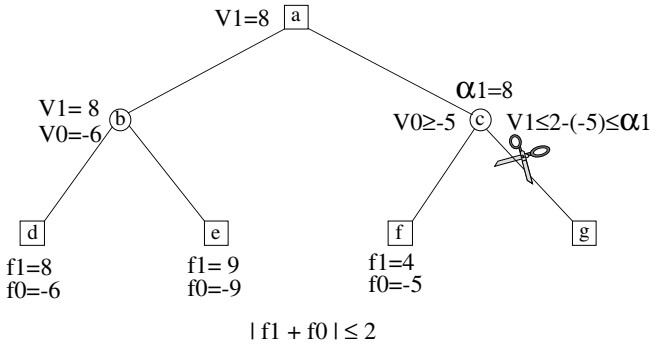

Figure 17: An example of pruning performed by $\alpha\beta^*$.

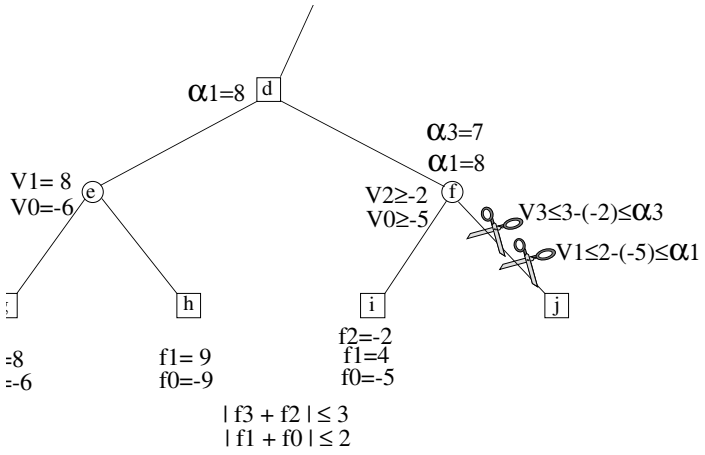

Figure 18: An example of deep pruning performed by $\alpha\beta^*$.

-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 19. The following theorem proves that $\alpha\beta^*$ always returns the M^* value of the root.

Theorem 3 *Let* $\langle V, B\rangle = \alpha\beta^*(pos, d, PLAYER, (-\infty, \ldots, -\infty))$. *Let* $V' = M^*_{1-pass}(pos, d, PLAYER')$ *where PLAYER' is 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 $M^*_{1-pass}(pos) = \alpha\beta^*(pos)$, it is sufficient to show that any node pruned by $\alpha\beta^*$ can have no effect on $M^*_{1-pass}(pos)$. 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 3, it follows that $V[i] + V[i-1] \leq B_i$ and therefore $\alpha_i \geq V[i]$. Since α_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. □

[4] 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.

```
Procedure αβ*(pos, depth,
            ((f_n, b_n)((f_{n-1}, b_{n-1}), (..., (f_0, b_0))...))(α_n, ..., α_0))
  if depth = 0
    return ⟨f_n(pos), ..., f_0(pos)⟩
  else
      B ← ComputeBounds(depth, ⟨b_n, ..., b_0⟩)
      V ← ⟨−∞, ..., −∞⟩
      SUCC ← MoveGen(pos)
      for each succ ∈ SUCC
          succ_V ← αβ*(succ, depth − 1,
                       ((f_n, b_n)((f_{n-1}, b_{n-1}), (..., (f_0, b_0))...))(α_n, ..., α_0))
          loop for each i associated with current player
              if succ_V[i] > V[i]
                V[i] ← succ_V[i]
                if i < n
                  V[i + 1] ← succ_V[i + 1]
              α_i ← max(α_i, V[i])
          if for every i not associated with current player [α_i ≥ B[i] − V[i − 1]]
            return ⟨V[n], ..., V[depth]⟩
  return ⟨V[n], ..., V[depth]⟩

Procedure ComputeBounds(d, ⟨b_n, ..., b_0⟩)
  if d = 0 return ⟨b_n, ..., b_0⟩
  else succ_B ← ComputeBounds(d − 1, ⟨b_n, ..., b_0⟩)
      loop for each i associated with current player
          B[i] ← succ_B[i] + 2 · succ_B[i − 1]
          if i < n  B[i + 1] ← succ_B[i + 1]
      return B
```

Figure 19: The $\alpha\beta^*$ algorithm

As the player function becomes more similar to its opponent model (but with an opposite sign), the level 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 [9]. 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 $TREE(b, d, B)$). 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 $TREE(2, 10, B)$. For each bound B we created 1000 trees and ran $\alpha\beta^*$ on them. Graph 20 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

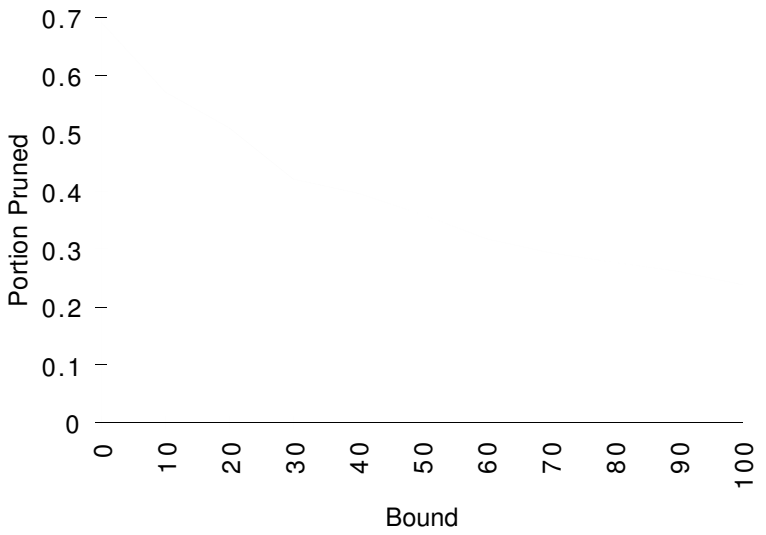

Figure 20: The portion of leaves pruned by $\alpha\beta^*$ as a function of the bound on $|f_1 + f_0|$.

is achieved for $B = 0$, where $\alpha\beta^*$ is reduced to standard $\alpha\beta$.

We repeated the third experiment of Subsection 3.2 replacing the M^*_{1-pass} algorithm with $\alpha\beta^*$. The bound used for pruning is:

$$\begin{aligned} |f_1 + f_0| &= |Material(board, player) - 0.004 \cdot TotalFigures(board) + \\ &\quad Material(board, opponent) - 0.004 \cdot TotalFigures(board)| \\ &\leq 0.008 \cdot TotalFigures(board) \end{aligned}$$

While the average number of leaves-per-move for a search to depth 4 by M^*_{1-pass} was 723, $\alpha\beta^*$ managed to acheive an average of 190 leaves-per-move. $\alpha\beta$ using f_1 achieved an average of 66 leaves-per-move.

The last results raise an interesting question. Assume that we allocate a modelling player and an unmodelling opponent the same search resources. Is the benefit achieved by modeling enough to overcome the extra depth that the non-modelling player can search due to the better prunning? This is an instance of the known tradeoff between knowledge and search. The answer to this question mostly depends on the particular nature of the game tree and evaluation functions. We have tested the question in the context the above experiment. We wrote an iterative deepening evrsions of both $\alpha\beta$ and $\alpha\beta^*$ and let the two play against each other using the same search resources. $\alpha\beta^*$ got 780 points out of 1600. We can see both players were almost equivalent, thus, the knowledge about the opponent strategy was indeed a worthwhile tradoff to the additional search performed by $\alpha\beta$.

6 Learning a model of the opponent's strategy

In the first part of this work we have presented methods for using opponent models. In this section we describe a methodology for acquiring such a model from examples. A set of boards with the opponent's decisions is given as input, and the learning procedure produces a model as output. This framework is similar to the scenario used by Kasparov as described in the opening quote.

To make the learning task more feasible, we assume that the opponent is a minimax player and its model therefore consists of two components: a depth of search and an evaluation function.

Since the space of possible depth values is much smaller than the space of possible function, we start by describing a method for learning the opponent's depth-of-search, and proceed with the more complicated task of learning its function.

6.1 Learning the depth of search

Given a set of examples, each consisting of a board together with the move selected by the opponent, it is relatively easy to learn its depth of search ($d_{opponent}$) under the assumption that the opponent searches to a fixed depth and that its evaluation function is known to the learner. Since there is only a small set of plausible values for $d_{opponent}$, we can test which of them agrees best with the opponent's decisions. One possibility is to test for strict agreement, i.e., for any example $(board, move)$ we can reward any depth d for which $minimax(board, d)$ returns $move(board)$.

When $f_{model} = f_{opponent}$, such a method needs only a few examples to infer $d_{opponent}$. However, in the case where f_{model} differs from $f_{opponent}$, such an algorithm is prone to error. In order to improve the learning ability in the presence of a wrong function model, we have developed an improved algorithm that considers the relative ordering between possible moves. We want to reward every depth d if the move given in the example has indeed the highest minimax value when searching to depth d. If it is the second best move, we still want to reward it, but a little bit less. To implement such a policy that credits a depth according to the relative place of the example move, the algorithm rewards every depth with the number of moves that have lower minimax value than the example move, and penalizes it with the number of alternative moves with higher minimax value than the example move. Thus, a depth for which the example move has the highest minimax value will get a high reward. The algorithm is shown in figure 21.

```
Procedure LearnDepth(EXAMPLES)
    for each ⟨board, move⟩ ϵ EXAMPLES
        boards ← successors(board)
        for d from 1 to MaxDepth
            M ← minimax(move(board), d − 1)
            count[d] ← count[d]
                + | {b ϵ boards | minimax(b, d − 1) ≤ M} |
                − | {b ϵ boards | minimax(b, d − 1) > M} |
    return d with maximal count[d]
```

Figure 21: An algorithm for learning a model of the opponent's depth (d_{model})

In order to study the effect of the distance between the model function and the actual function on the learning ability of the algorithm, we have performed the following experiment.

1. f_{model}, used by the learning algorithm, was fixed.
2. $f_{opponent}$ was defined to return a value according to a parameter p. The function returns f_{model} with probability p and a random value with probability $1 - p$.
3. A set of games between two minimax players, *PLAYER* and *OPPONENT*, was conducted. 100 boards that OPPONENT faced, together with its chosen moves, were given as examples to the learning algorithm which updated d_{model}.

4. After each move, the current d_{model} was compared against $d_{opponent}$. If they were in disagreement, the accumulative error rate was incremented.

5. The experiment was repeated for various p.

Figure 22 shows the accumulative error rate of the algorithms as a function of the distance between f_{model} and $f_{opponent}$. It also shows the d counters after 100 examples for the case of f_{model} with probability of error equals to 0.25.

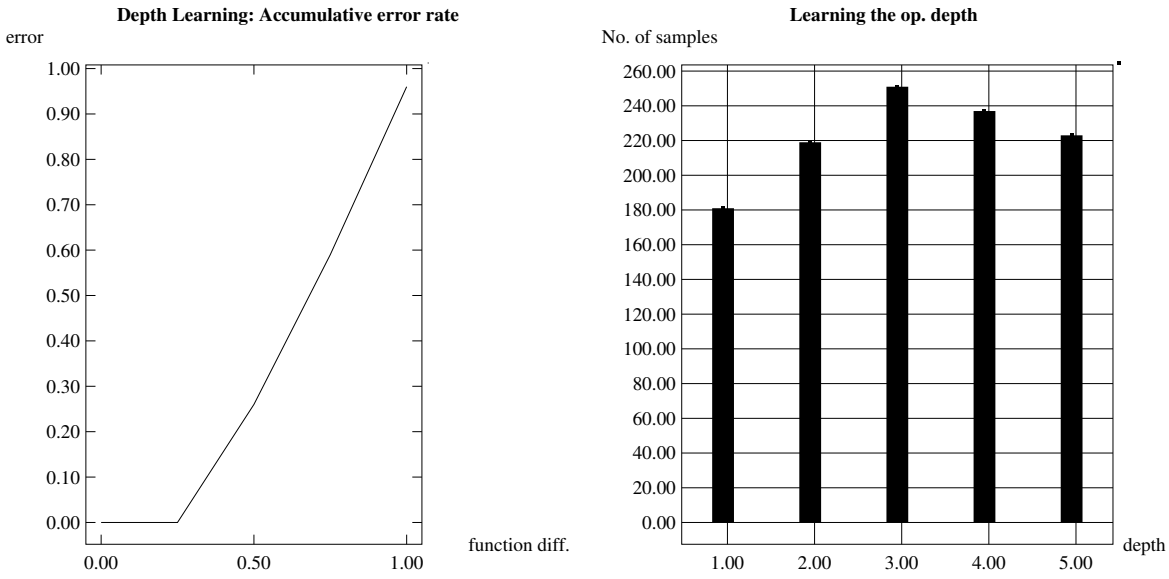

Figure 22: Learning the depth of search using 100 examples, where $d_{opponent} = 3$ and f_{model} differs from $f_{opponent}$ with probability between 0 and 1. The left graph shows the accumulative error rate as a function of the error of the function model. The right graph shows the counters of the learning algorithm for function model error of 0.25.

The accumulative error rate is the portion of the learning session where the learner has a wrong model of its opponent's depth. The experiment shows that indeed when the function model is close enough to the opponent's function, the algorithm succeeds in learning the opponent's depth. However, when the opponent's function is significantly different from the model, the algorithm's error rate increases.

Learning the opponent's depth of search may provide very useful information, especially in a game against a weaker opponent. When a player is aware of the limit of its opponent's search horizon, it can lead the opponent to trap or swindle positions [6].

6.2 Learning the opponent's strategy

The performance of the previous algorithm depends on the knowledge of the opponent's evaluation function. The natural next step is to develop an algorithm for learning evaluation functions. However, learning the opponent's function will probably depend on knowing the opponent's depth of search. In order to break this circle, we have developed an algorithm for learning the function and depth of search simultaneously.

Since learning an arbitrary real function is hard, we have made the following simplifying assumptions:

1. The opponent's function is a linear combination of features. $f(b) = \overline{w} \cdot \overline{h}(b) = \sum_i w_i h_i(b)$ where b is the evaluated board and $h_i(b)$ returns the ith feature of that board.

2. A superset of the features used by the opponent is known to the learner.

3. The opponent does not change its function while playing.

Under these assumptions, the learning task is reduced to finding the pair $(\overline{w}_{model}, d_{model})$.

The learning procedure computes for each possible depth d a weight vector $\overline{w}_d$, such that the strategy $(\overline{w}_d \cdot \overline{h}, d)$ most agrees with the opponent's decisions. The adapted model is the best pair found for all depths. For each depth, the algorithm performs a hill-climbing search, improving the weight vector until no further significant improvement can be achieved. Assume that $\overline{w}_{current}$ is the best vector found so far for the current depth. For each of the examples, the algorithm builds a set of constraints that express the superiority of the selected move over its alternatives. The algorithm performs a minimax search using $(\overline{w}_{current} \cdot \overline{h}, d-1)$, starting from each of the successors of the example board. At the end of this stage each of the alternative moves can be associated with the "dominant" board that determines its minimax value. Assume that b_{chosen} is the dominant board of the chosen move, and $b_1, \ldots, b_n$ are the dominant boards for the alternative moves. The algorithm adds the n constraints $\{\overline{w} \cdot (\overline{h}(b_{chosen}) - \overline{h}(b_i)) \geq 0 \mid i = 1, \ldots, n\}$ to its accumulated set of constraints.

The next stage consists of solving the inequalities system, i.e., finding $\overline{w}$ that satisfies the system. The method we used is a variation of the linear programming method used by Duda and Hart [4] for pattern recognition. Before the algorithm starts its iterations, it sets aside a portion of its examples for progress monitoring. This set is not available to the procedure that builds the constraints. After solving the constraints system , the algorithm tests the solution vector by measuring its accuracy in predicting the opponent's moves for the test examples. The performance of the new vector is compared with that of the current vector. If there is no significant improvement, we assume that the current vector is the best that can be found for the current depth, and the algorithm repeats the process for the next depth using the current vector for its initial strategy.

The inner loop of the algorithm, that searches for the best function for a given depth, is similar to the method developed by Christensen and Korf [8] and used by DEEP THOUGHT [5] and by CHINOOK [18] for tuning their evaluation function from book moves. However, these programs assume a fixed small depth for their search. Meulen [20] used a set of inequalities for book learning, but his program assumes only one level depth of search.

The algorithm is listed in figure 23.

The strategy learning algorithm was tested by the following experiment. Three fixed strategies $(f1, 8)$, $(f2, 8)$ and $(f1, 6)$, were used as opponents, where $f1$ and $f2$ are two variations of Samuel's function. Each strategy was used to play games until 1600 examples were generated and given to the learning algorithm. The algorithm was also given a set of ten features, including the six features actually used by the strategies.

The algorithm was run with a depth limit of 11. The examples were divided by the algorithm into a training set and a testing set of size 800. For each of the eleven depth values, the program performed 2-3 iterations before moving to the next depth. Each iteration included using the linear programming method for a set of several thousand constraints[5].

[5] We have used the very efficient lp_solve program, written by M.R.C.M. Berkelaar, for solving the constraints system

Procedure $LearnStrategy(examples)$
$\overline{w}_0 = \overline{1}$
for d **from** 1 to $MaxDepth$
    $\overline{w}_d \leftarrow \overline{w}_{d-1}$
    Repeat
        $\overline{w}_{current} \leftarrow \overline{w}_d$
        $\overline{w}_d \leftarrow FindSolution(examples, \overline{w}_{current}, d)$
        $progress \leftarrow |score(\overline{w}_d, d) - score(\overline{w}_{current}, d)| \geq \epsilon$
    Until no progress
return $(\overline{w}_d, d)$ with the maximal score.

Procedure $FindSolution(EXAMPLES, \overline{w}_{current}, d)$
  $Constraints \leftarrow \phi$
  for each $\langle board, chosen\_move \rangle \epsilon\, EXAMPLES$
    $SUCC \leftarrow MoveGen(board)$
    for each $succ \epsilon\, SUCC$
      $dominant_{succ} \leftarrow Minimax(succ, \overline{w}_{current}, d-1)$
      $Constraints \leftarrow Constraints \cup \{\overline{w}(\overline{h}(dominant_{chosen\_move}) - \overline{h}(dominant_{succ})) \geq 0\}$
  return $\overline{w}$ that satisfy $Constraints$

Figure 23: An algorithm for learning a model of the opponent's strategy

The results of the experiment for the three strategies is shown in figure 24. The algorithm succeeded for the three cases, achieving an accuracy of 100% for two strategies and 93% for the third. Furthermore, the highest accuracy was achieved for the actual depth used by the strategies.

6.3 OLSY: An Opponent Learning System

In order to test how well the M^* algorithm and the learning algorithm can be integrated, we have built a game playing program, OLSY, that is able to acquire and maintain a model of its opponent and use it to its advantage. The system consists of two main components, a game-playing program and a learning program. The game-playing component uses the M^*_{1-pass} algorithm. The learning program accumulates the moves performed by the opponent and modifies the opponent model while playing. After each k moves (10 in our experiments), the system updates its model of opponent using the algorithm in Figure 23.

The OLSY system was tested in a realistic situation by letting it play a sequence of checkers games against regular minimax players (all programs used pruning, but it did not effect the results, since all algorithms searched to the same depth and no time limit was given.) OLSY and its two opponents each used a different strategy but with a roughly equivalent playing ability. They all searched to depth 6 and they all used variations of Samuel's static evaluation function. We stopped the games several times to test how well OLSY performs against its opponents. The test was conducted by turning off the learning mechanism and performing a tournament of 100 games between the two.

Figure 25 shows the results of this experiment. The players start with almost equivalent ability. However, after only few examples, the learning program becomes significantly stronger than its

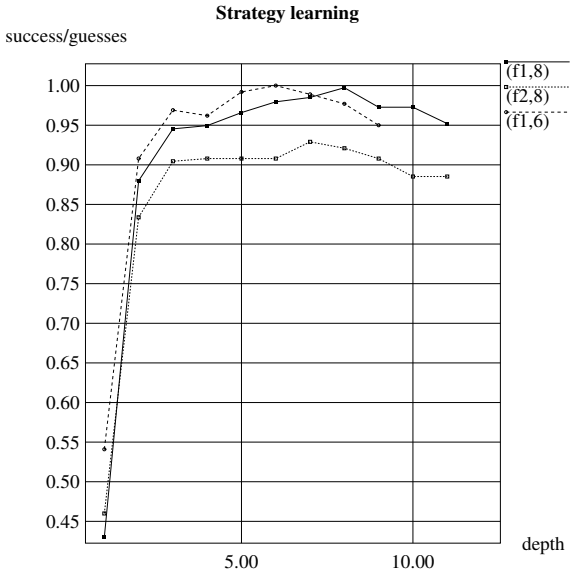

Figure 24: Learning opponent's strategy

non-learning opponents. The performance of OLSY kept increasing until about 50 examples were processed. After that, the learning curve was leveled. We inspected the accuracy of the models as determined by the learning procedure. After 100 examples, OLSY succeeded in acquiring models for its two opponents with a quality of 95% for the first and 98% for the second. Learning more examples did not cause any further modification of the models.

7 Conclusions

The minimax algorithm assumes that the opponent uses the same strategy as does the player. In this paper we presented a generalized version of the minimax algorithm that can utilize different opponent models. We presented the M^* algorithm, which simulates the opponent's search to determine its expected decision for the next move, and evaluates the resulted board by searching its associated subtree using its own strategy. We then presented the M^*_{1-pass} algorithm, which is a version of M^* that expands the tree only once, but propagates all the necessary values. The M^* algorithm (as well as Minimax) assumes that the opponent searches as deep as the player's search horizon. We extended the M^* and M^*_{1-pass} algorithms to enable the utilization of models of the opponent's depth of search.

Experiments performed in the domain of checkers demonstrated the advantage of M^* over minimax but also showed that an error in the model can deteriorate performance significantly. We developed an algorithm, M^*_ϵ, 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.

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. When the bound approaches zero, the amount of pruning approaches that of $\alpha\beta$.

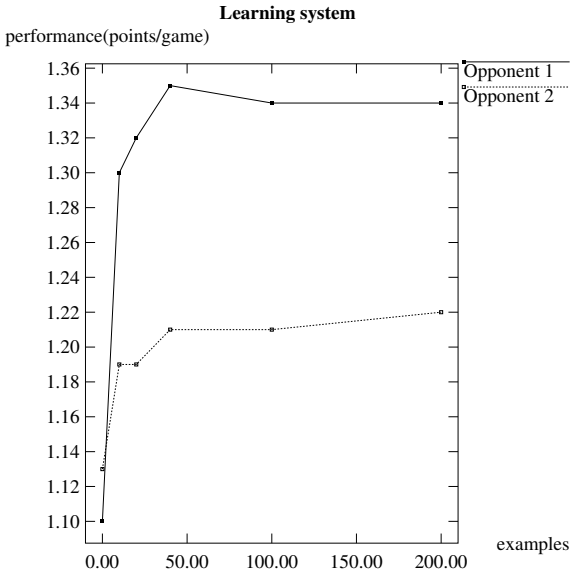

Figure 25: The performance of the learning system as a function of the number of examples. Measured by mean points per game.

In the second part of the paper we tackled the problem of learning an opponent's model by using its moves as examples. We developed an algorithm for learning an opponent model, both depth and evaluation function. The algorithm works by iteratively increasing the model depth and learning a function that best predicts the opponent's moves for that depth. For testing the algorithm, we tried to learn minimax players that search to a fixed depth and use an evaluation function based on a linear combination of features, known to the learner. The results show that few examples are needed for learning a model that agrees almost perfectly with such a player. In the future, we mean to investigate the algorithm's ability to model more sophisticated players. We tested the algorithm in a game-playing learning system, named OLSY. The system was tested in a realistic situation, learning its opponent while playing against it. The system indeed showed an improvement as more examples of opponent's moves became available.

A mojor issue for further research is the tradeoff between the playing ability gained by modelling versus the ability lost by the reduced pruning. More work need to be done in order to show under what considitions this tradeoff is beneficial.

We believe that this work presents significant progress in the area of *using* opponent models. The M^* family of algorithms presented in this paper are all generalizations of minimax that allow us to use n-level opponent models together with bounds on their errors. This work also presents initial steps in the area of *learning* opponent models. The task of learning n-level models is extremely difficult and deserves further research.

8 Acknowledgments

We would like to thank David Lorenz and Yaron Sella for letting us use their efficient checker playing code as a basis for our system. We would also like to thank Arie Ben-Ephraim for helping us in the early stages of this work. Finally, we thank M.R.C.M. Berkelaar from Eindhoven University of Technology, The Netherlands for making his extremely efficient lp_solver program available to the public.

References

[1] B. Abramson. Expected outcome: A general model of static evaluation. *IEEE Trans. on Pattern Analysis and Machine Intelligence 12,182-193*, 1990.

[2] H. Berliner. Search and knowledge. In *Proceeding of the International Joint Conference on Artifical Intelligence (IJCAI 77)*, pages 975–979, 1977.

[3] H. Berliner. The b* tree search algorithm: A best-first proof procedure. *Artifical Intelegence 12, 23-40*, 1979.

[4] R. O. Duda and P. Hart. *Pattern Classification and Scene Analysis*. New York: Wiley and Sons, 1973.

[5] F.-H. Hsu, T. Ananthraman, M. Campbell, and A. Nowatzyk. Deep thought. In T. Marsland and J. Schaeffer, editors, *Computers, Chess and Cognition*, pages 55–78. Springer New York, 1990.

[6] P. Jansen. Problematic positions and speculative play. In T. Marsland and J. Schaeffer, editors, *Computers, Chess and Cognition*, pages 169–182. Springer New York, 1990.

[7] D. Knuth and R. Moore. An analysis of alpha-beta pruning. *Artifical Intelligence 6, no.4, 293-326*, 1975.

[8] R. Korf. Depth-first iterative deepening: An optimal admissible tree search. *Artifical Intelegence 27, 97-109*, 1985.

[9] R. E. Korf. Generalized game trees. In *Proceeding of the International Joint Conference on Artifical Intelligence (IJCAI 89)*, pages 328–333, Detroit, MI, Aug. 1989.

[10] R. E. Korf. Multy-player alpha-beta pruning. *Artifical Intelegence 48, 99-111*, 1991.

[11] D. Levy and M. Newborn. *How Computers Play Chess*. W.H. Freeman, 1991.

[12] R. D. Luce and H. Raiffa. *Games and Decisions*. New York: Wiley and Sons, 1957.

[13] C. A. Luckhardt and K. B. Irani. An algorithmic solution of n-person games. In *Proceeding of the Ninth National Conference on Artifical Intelligence (AAAI-86)*, pages 158–162, August 1986.

[14] D. A. McAllester. Conspiracy numbers for min-max search. *Artifical Intelegence 35, 287-310*, 1988.

[15] E. H. D. P. J. Gmytrasiewicz and D. K. Wehe. A decision theoretic approach to coordinating multiagent interactions. In *Proceedings of the International Joint Conference on Artifical Intelligence (IJCAI 91)*, pages 62–68, 1991.

[16] A. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal, 3, 211-229*, 1959.

[17] A. Samuel. Some studies in machine learning using the game of checkers ii–recent progress. *IBM Journal, 11, 601-617*, 1967.

[18] J. Schaeffer, J. Culberson, N. Treloar, B. Knight, P. Lu, and D. Szafron. A world championship caliber checkers program. *Artifical Intelegence 53, 273-289*, 1992.

[19] C. E. Shannon. Programming a computer for playing chess. *Philosophical Magazine, 41, 256-275*, 1950.

[20] M. van der Meulen. Weight assessment in evaluation functions. In D. Beal, editor, *Advances in Computer Chess 5*, pages 81–89. Elsevier Science Publishers, Amsterdam, 1989.