Learning Models of Opponent's Strategy in Game Playing

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

Shaul Markovitch *
Computer Science Department
Technion, Haifa 32000
Israel
shaulm@cs.technion.ac.il

Abstract

Most of the activity in the area of game playing programs is concerned with efficient ways of searching game trees. There is substantial evidence that game playing involves additional types of intelligent processes. One such process performed by human experts is the acquisition and usage of a model of their opponent's strategy.

This work studies the problem of opponent modelling in game playing. A simplified version of a model is defined as a pair of search depth and evaluation function. M*, a generalization of the minimax algorithm that can handle an opponent model, is described. The benefit of using opponent models is demonstrated by comparing the performance of M* with that of the traditional minimax algorithm. An algorithm for learning the opponent's strategy using its moves as examples was developed. Experiments demonstrated its ability to acquire very accurate models. Finally, a full model-learning game-playing system was developed and experimentally demonstrated to have advantage over non-learning player.

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." [8]

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 of the 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 hilights one type of such a process that is performed by expert human players: acquiring a model of their opponent's strategy.

Several researchers have pointed out the importance of modelling the opponent's strategy, [10, 2, 6, 7, 1, 11], but the acquisition and use of an opponent's model have not received much attention in the computational games research community.

Human players take advantage of their modelling ability when playing against game playing programs. International Master David Levy [8] testified that he had specialized in beating stronger chess programs (with higher ELO rating), by learning their expected reactions. Samuel [10] also described a situation where human experts who had learned the expected behavior of his famous checkers program, succeeded better against it than those who did not. Jansen [5] studied the implications of speculative play. He paid special attention to swindle and trap positions and showed the usefulness of considering such positions during the game.

The work described in this paper makes one step into the understanding of opponent modelling by game playing programs. In order to do so, we will make an attempt to find answers to the following questions:

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

*This research was partially supported by the Fund for the Promotion of Research at the Technion

We will start by defining a simplified framework for opponent's strategy. Assuming a minimax search procedure, a strategy consists of the depth of the search and the static evaluation function used to evaluate the leaves of the search tree. The traditional minimax procedure assumes that the opponent uses the same strategy as the player. In order to be able to use a different model we have come up with a new algorithm, M*, that is a generalization of minimax.

The potential benefits of using an opponent's model is not obvious. If the opponent has a better strategy than the player, and the player possesses a perfect model of its opponent, then the player should adapt his opponent's strategy. If the player has a better strategy than the opponent's, then playing regular minimax is a good cautious method, but not necessarily the best. Setting traps, for example, would be excluded most of the time. In the case where the two strategies are different but neither is better than the other, the minimax assumptions about the opponent's moves may be plainly wrong.

In order to measure the potential benefits of using opponent modelling, we have conducted a set of experiments comparing the performance of the M* algorithm with the performance of the standard minimax algorithm. Finally, we have studied the problem of the modelling process itself. We have tested some learning algorithms that use opponent's moves as examples, and learn its depth of search and its evaluation function.

Section 2 deals with the first two questions: Defining a model and developing an algorithm for using a model. Section 3 deals with the third and the fourth questions: Measuring the potential benefits of modelling and testing the effects of modelling accuracy on its benefits. Section 4 discusses the fifth question: Learning opponent's models. Section 5 concludes.

2 Using opponent models

In this section we answer the first two questions raised in the introduction: what is an opponent model, and how can we use such a model.

2.1 Definitions and assumptions

Our basic assumption is that the opponent employs a basic minimax search and evaluates the leaves of the search tree by a static evaluation function. The opponent may use pruning methods, such as $\alpha\beta$ [6], that select the same moves as minimax. We assume that the opponent searches to a fixed and uniform depth: no selective deepening methods, such as quiescence search or singular extensions are used. We shall also assume that the opponent does not use any explicit model of the player.

Under the above assumptions we can define a playing strategy:

Definition 1 *A* playing strategy *is a pair (f, d) where f is a static evaluation function, and d is the depth of the minimax search.*

Definition 2 *An* opponent model *is a playing strategy. We will denote an opponent model by $S_{model} = (f_{model}, d_{model})$ while the actual strategy used by the opponent will be denoted by $S_{op} = (f_{op}, d_{op})$. The strategy used by the player will be denoted by $S_{player} = (f_{player}, d_{player})$.*

Definition 3 *A player is a pair of strategies, (S_{player}, S_{model}).*

2.2 The M* algorithm

Assuming that S_{player} and S_{model} are given, how do we incorporate them into the search for the right move? The M* algorithm, listed in figure 1, is a generalization of minimax that considers both the player and its opponent strategies. The algorithm takes S_{player} and S_{model} as input. It develops the game tree in the same manner as minimax does, but the values are propagated back in a different way.

```
$M^*(pos, depth, S_{player}, S_{model})$
  if $depth = 0$
    return $\langle f_{player}(pos), f_{model}(pos) \rangle$
  else $SUCC \leftarrow MoveGen(pos)$
    for each $succ \in SUCC$
      $v \leftarrow M^*(succ, depth - 1, S_{player}, S_{model})$
      if $(d_{model} + depth = d_{player} - 1)$
        $v_{model} \leftarrow f_{model}(succ)$

      if $MAX$ turn:
        $best_{player} \leftarrow \max(best_{player}, v_{player})$
        $best_{model} \leftarrow \min(best_{model}, v_{model})$

      if $MIN$ turn:
        if $best_{model} < v_{model}$
          $best_{model} \leftarrow v_{model}$
          $best_{player} \leftarrow v_{player}$
        else
          if $(best_{model} = v_{model})$
            $best_{player} \leftarrow \min(best_{player}, v_{player})$
    return $\langle best_{player}, best_{model} \rangle$
```

Figure 1: A simplified version of the M* algorithm

Since according to our assumptions, the opponent will use a regular minimax search with S_{model}, the opponent's reactions to each of our moves is decided according to S_{model}. However, the *value* that we attach to each node should be according to our strategy, S_{player}. Therefore, we propagate back two values: v_{model}, the value computed by f_{model} and v_{player}, the value computed by f_{player}. At the leaves level, we compute both

f_{player} and f_{model} for each board. At a MAX level, v_{model} receives the value of the minimal v_{model} value of all the successors, while v_{player} receives the maximal v_{player} value of the successors. At a MIN level, v_{model} gets the value of the maximal v_{model} value of all the successors. v_{player} gets the v_{player} value of the node selected by MIN (the one with the highest v_{model} value). If there is more than one node with maximal v_{model} value, then M* passes up the minimal v_{player} value in that set of nodes with maximal v_{model}. That is because MIN may select any of those. If an interior node in the search tree is found on the search frontier of MIN, v_{model} is adapted by calling f_{model} on that node. Figure 2 shows an example where minimax and M* recommend different moves.

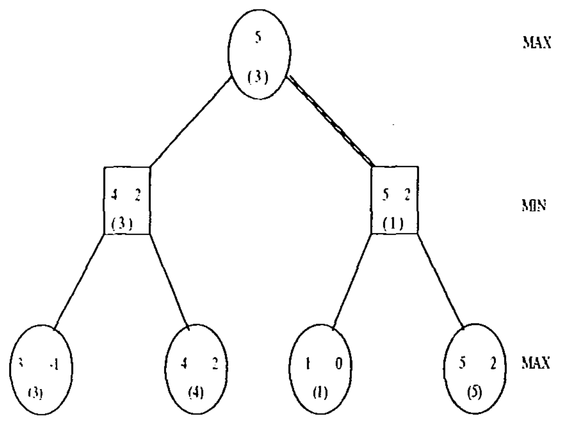


Figure 2: The tree spanned by M*. $S_{player} = (f_{player}, 2)$, $S_{model} = (f_{model}, 1)$. Left values are node evaluations by S_{player}. Right values are node evaluations by S_{model}. Minimax values are in brackets. M* chooses the right move while minimax chooses the left.

Korf [7] proposes a similar algorithm for using a model of the opponent evaluation function. The algorithm evaluates each leaf twice - using the opponent's function and the player's function. A pair of values is then propagated up, selecting the pair with the highest player's component in MAX level, and taking the pair with the lowest opponent's component in MIN level. There is a major difference between M* and Korf's algorithm. While M* uses one level of modelling, i.e., it assumes that the opponent does not use a model of the player, Korf's algorithm assumes maximal level of modelling. It assumes that the player possesses a model of the opponent's function, and a model of the opponent's model of its function, and a model of the opponent's model of that model etc.

It is possible to add $\alpha\beta$ pruning to M* by transferring two pairs of values, α_{player}, β_{player} and α_{model}, β_{model}. Pruning will take place only if both MAX and MIN agree that it is useless to continue developing that part of the tree. Therefore, M* will have fewer cutoffs than regular Minimax, costing more search time. The payoff in search time depends on the similarity between f_{player} and f_{model}. When the two functions always agree on the relative ordering between positions, M^* will prune the same branches as minimax does. When the two functions always disagree, M^* will never prune. We did not include $\alpha\beta$ pruning in figure 1 for sake of clarity.

2.3 Properties of M*

It is easy to show that M* is a generalization of the minimax algorithm.

Lemma 1 *Assume that M* and Minimax use the same* S_{player} *strategy.*
By using $S_{model} = (-f_{player}, d_{player} - 1)$, *M* becomes identical to minimax.* [1]

$$Minimax(position, depth) = \\ M^*(position, depth, S_{player}, S_{model}).$$

It is also easy to show by induction on the depth of search that M* always selects a move with a value greater or equal to the one selected by minimax.

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

$$Minimax(position, depth) \leq \\ M^*(position, depth, S_{player}, S_{model})$$

for any S_{model}.

The intuition behind the above lemma is, that if the opponent is weaker than the player, and the player knows this, then the player can make less conservative assumptions about the opponent's reactions, increasing the value returned by the procedure. The fact that M* returns a higher value does not mean that it always selects better moves. If it underestimates the opponent, then the higher value will not be materialized.

The risk taken by using M* depends on the quality of the model. However, when using M* to play against a weaker opponent, there is a higher risk in trusting its static evaluation function as a predictor. Therefore, for the experiments described in the next section, we have used a modified version of M*. The reactions of the opponent for each of the alternative following moves is computed using S_{model} as before. However, in deeper MIN levels of the tree, M* passes up the minimal v_{player} instead of the value of the board selected by MIN. This method reduces the risk, especially in the case of weaker opponent's model.

[1] M* can not handle d_{model} greater than $d_{player} - 1$. Also, we can consider an evaluation function that receives the active player as an additional parameter and returns the value accordingly. Therefore, from now on, we can say that the standard minimax algorithm uses $S_{model} = S_{player} = (f_{player}, d_{player})$

3 The potential benefits of using opponent models

Now that we have an algorithm for using an opponent's model, we would like to evaluate the potential benefit of using this algorithm. In order to do so, we have conducted a set of experiments comparing the M* algorithm that has a perfect model of its opponent (i.e., $S_{model} = S_{op}$) to the regular Minimax algorithm.

3.1 Experimentation methodology

The experiments described in the following subsections involve the following players:
$MM = (S_{player}, S_{player})$
$M^* = (S_{player}, S_{model})$. $(S_{model} = S_{op})$
$OP = (S_{op}, S_{op})$
A basic test consists of a set of 100 games played between M* and OP, and another set of 100 games played between MM and OP. Both competitors, M* and MM, were allotted the same search resources. The benefit of the M* algorithm over MM is measured by the difference between the mean points per game (2 points for a win, 1 point for a draw).

The experiments were conducted for two different games: Tic-tac-toe on an 3 × 3 board with the well known "open lines advantage" evaluation function, and checkers with an evaluation function based on the one used for Samuel's checkers player[9].

3.2 The effect of the level difference on the benefit of modelling

For the first experiment described here, we have fixed f_{player} and f_{op} and varied values of the depth components of the strategies. For the second experiment, the depth parameters of all strategies were kept constant. The function f_{player} was set to be Samuel's evaluation function while $f_{model} = f_{op}$ was formed by destroying f_{player} values that reside outside the range $-a \ldots a$.

$$f^a_{model}(x) = \begin{cases} f_{player}(x) & \text{if } |(f_{player}(x)| < a \\ 0 & \text{otherwise} \end{cases}$$

As a becomes smaller, $f_{player} - f_{op}$ becomes larger and the function quality drops.

The experiments can be interpreted in two ways:

- Testing the effect of level difference between the two players on the benefit of using a perfect model over standard minimax.
- Testing the effect of the distance between the model and the actual strategy on its benefit, or, in different words, testing the importance of modelling accuracy.

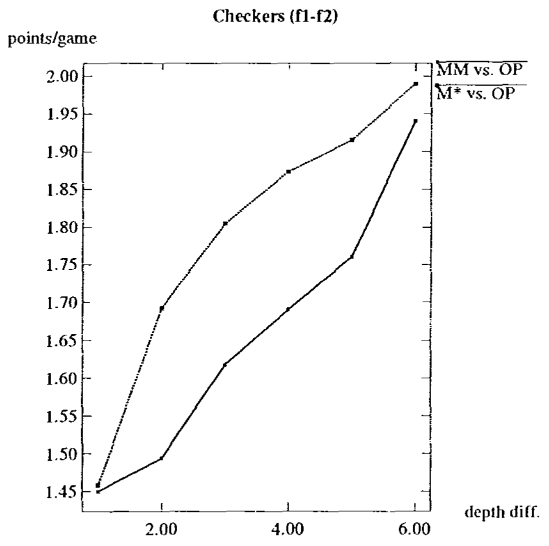


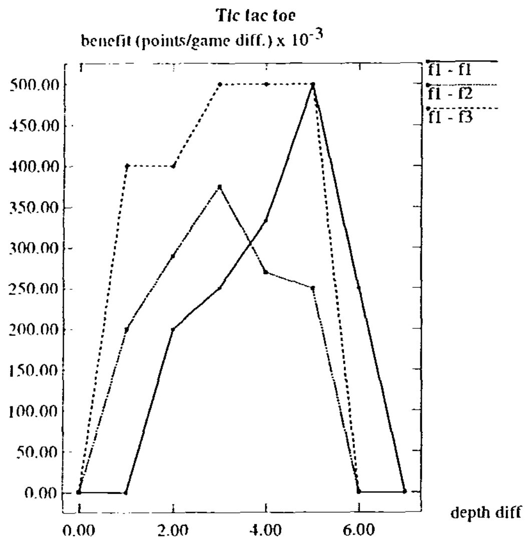
Figure 3: The performance of M* vs. the performance of Minimax as a function of search depth difference. Measured by mean points per game.

Figure 3 exhibits the full results of one checkers tournament. We can see that M* always performs better then Minimax.

Figures 4,5 summarize the results of the experiments. In these graphs we plot the difference in performance between the two algorithms. All graphs exhibit similar behavior: The benefit of opponent modelling increases with the difference in level up to a certain point where the benefit starts to decline. The increase in the benefit can be explained by the observation that Minimax is being too careful in predicting its opponent's moves, while M* utilizes its model and exploits the weaknesses of its opponent to its advantage. Alternatively, we can say that harm is caused by incorrect modelling, and is increased with the difference between the model and the actual strategy. Overestimating the opponent will usually cause too defensive strategy. When the level difference becomes larger, Minimax wins in almost all games. In such a case there is little place for improvement by modelling.

4 Learning a model of the opponent's strategy

The last section demonstrated the potential benefit of using an opponent's model. In this section we will discuss methods for acquiring such a model. We assume the framework of learning 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.



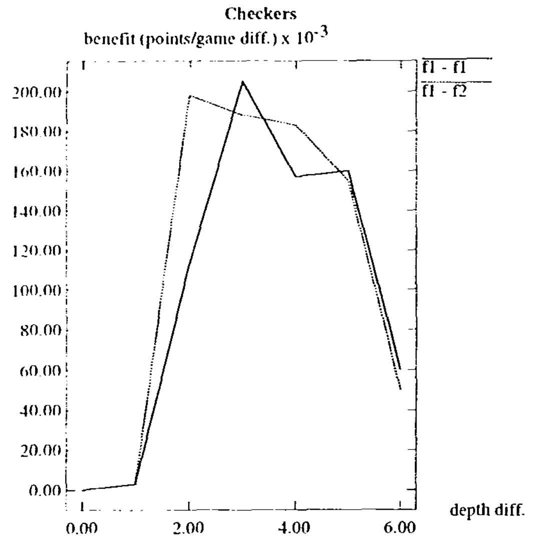


Figure 4: The benefit of using M* over minimax as a function of the search depth difference. Measured by mean points per game.

According to our assumptions, the opponent searches to a fixed depth, therefore learning d_{model} involves selecting a depth from a small set of plausible values. The space of possible functions is nevertheless infinite, and the task of learning f_{model} is therefore much harder.

4.1 Learning the depth of search

Given a set of examples, each consists of a board together with the move selected by the opponent, it is relatively easy to learn the depth. Since there is only a small set of plausible values for d_{op}, we can check which of them agrees best with the opponent decisions. The algorithm for learning the depth is listed in figure 6.

When $f_{model} = f_{op}$, the above algorithm needs few examples to infer d_{op}. It can be proven that d_{op} will

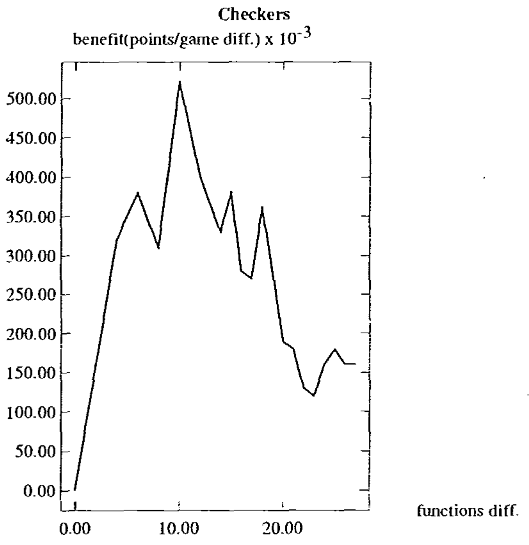


Figure 5: The benefit of using M* over minimax as a function of the function difference. Measured by mean points per game.

```
LearnDepth(examples)
  for each $\langle board, move\rangle \in examples$
    $boards \leftarrow successors(board)$
    for $d$ from 1 to $MaxDepth$
      $M \leftarrow minimax(move(board), d)$
      $count[d] \leftarrow count[d]$
        $+ \mid \{b \in boards \mid minimax(b,d) \le M\} \mid$
        $- \mid \{b \in boards \mid minimax(b,d) > M\} \mid$
  return $d$ with maximal $count[d]$
```

Figure 6: An algorithm for learning a model of the opponent's depth (d_{model})

always be in the set of depth counters with maximal values. However, in the case that f_{model} differs from f_{op} the algorithm can make an error. Figure 7 shows the counters of all depths after searching 100 examples. The algorithm succeeds to predict d_{model} in the presence of imperfect function model. Figure 8 shows the accumulative error rate of the algorithm as a function of the distance between f_{model} and f_{op}. 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 perfect, the algorithm succeeds in learning the opponent's depth after a few examples. However, when the opponent's function is even slightly different than the model, the algorithm's error rate increases significantly.

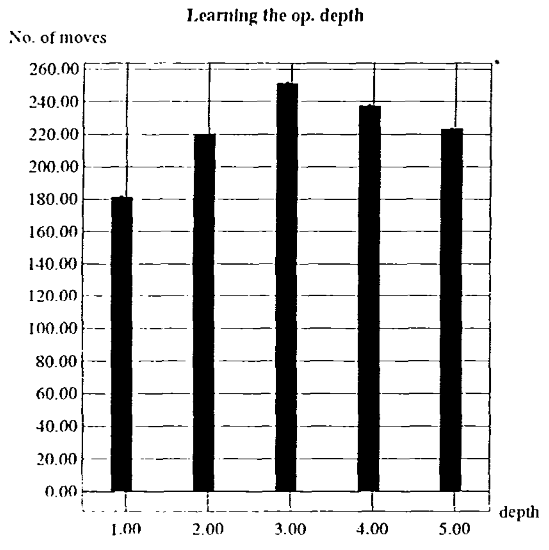


Figure 7: Learning the depth of search by 100 examples, $d_{op} = 3$ f_{op} return a random value with probability 0.25

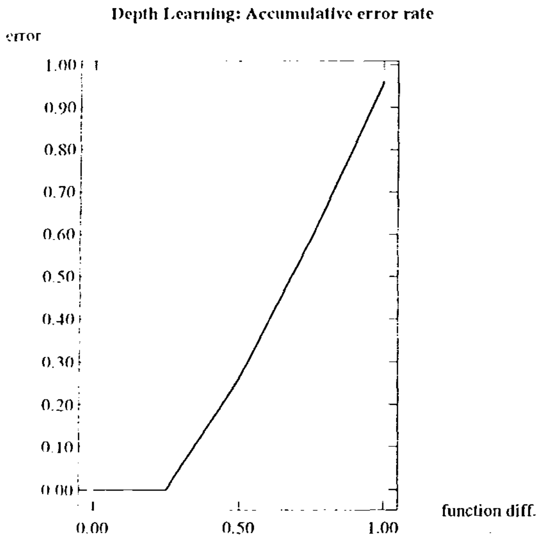


Figure 8: The error rate of the algorithm as a function of the functions difference

4.2 Learning the opponent's strategy

In order to learn the opponent's strategy, will make the following 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. The feature set of 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 listed in figure 9 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.

```
$LearnStrategy(examples)$
$\overline{w}_0 = \overline{1}$
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.

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

Figure 9: An algorithm for learning a model of the opponent's strategy $(\overline{w}_{model}, d_{model})$

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 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 determine 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 [3] 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 our algorithm, that searches for the best function for a given depth, is similar to the method used by DEEP THOUGHT [4] and by Chinook [11] for tuning their evaluation function from book moves. However, these programs assume a fixed small depth for their search. Meulen [12] used a set of inequalities for book learning, but his program assumes only one level depth of search.

4.3 Strategy learning experiments

The strategy learning algorithm was tested by two experiments. The first experiment tests the prediction accuracy of models acquired by the algorithm. 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 to 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 programing method for a set of several thousands constraints[2].

The results of the experiment for the three strategies is shown in figure 10. 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.

The second experiment tested the usage of the model learning algorithm by a playing program. A model-learning playing system was built by using the model learning algorithm for acquiring opponent's model, and the M* for using it. The system accumulates the opponents moves during the game. After each alternating game, the model learning procedure is called with the total accumulated set of opponent moves. The learned model is then adapted for use by the M* algorithm.

The system was tested in a realistic situation by letting it play a sequence of games against regular minimax players that use different strategies with roughly equivalent playing ability. After each model modification by the learning program, a tournament of 100 games, between the competitors, was conducted for measuring their relative performance. Obviously, the learning mechanisms, including move-recording, were turned off

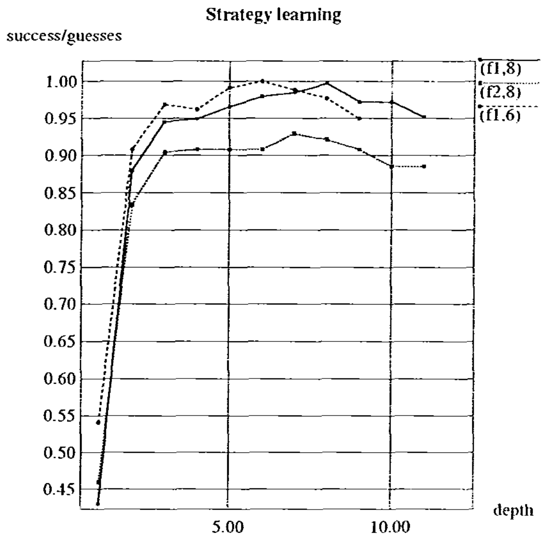


Figure 10: Learning opponent's strategy

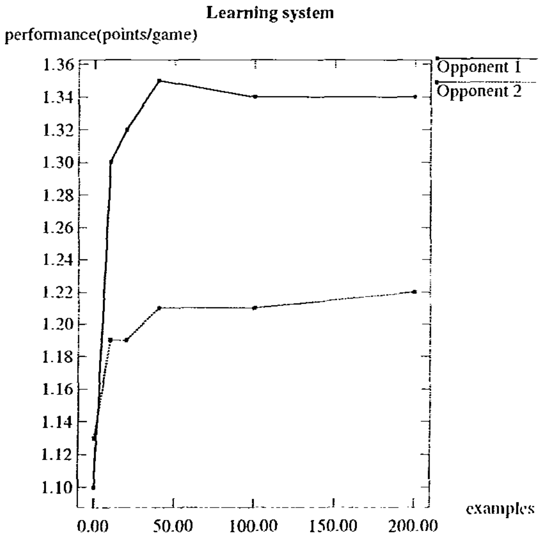


Figure 11: The performance of learning program as a function of the number of learning games. Measured by mean points per game.

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

for the whole duration of the testing phase.

Figure 11 shows the results of this experiment. The players start of with almost equivalent ability. However, after several games, the learning program becomes significantly stronger than its non-learning opponents.

5 Conclusions

This work takes one step into understanding the process of opponent's modelling in game playing. We have defined a simplified notion of opponent's model – a pair of an evaluation function and a depth of search. We have also developed M*, a generalization of the minimax algorithm that is able to use an opponent model. The potential benefits of the algorithm over standard minimax were studied experimentally. It was shown that as the opponent becomes weaker, the potential benefit over minimax increases. The same experiments also demonstrate the potential harm of overestimating the opponent.

After establishing the benefit of using accurate model, we proceeded with tackling the problem of learning opponent's model using its moves as examples. We have come out with an algorithm that quickly learns the depth of the opponent search in the presence of imperfect model of the opponent function.

Next, we have developed an algorithm for learning an opponent model (both depth and evaluation function), using its moves as examples. The algorithm works by iteratively increasing the model depth and learning a function that best predicts the opponents moves for that depth.

Finally, a full playing system was built, that is able to model its opponent while playing with it. Experiments demonstrated that the learning-player advantage over non-learning player, increases with the number of games.

One of the simplified assumptions that we have made, is a fixed-depth search by the opponent. Obviously, this is not a realistic assumption. We intend to examine what are the consequences of removing this assumption.

The algorithm developed for learning opponent model proved to be extremely efficient in acquiring an accurate model of the opponent. It would be interesting to test whether it can achieve similar results when used for book learning.

6 Acknowledgements

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

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

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

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

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

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

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

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

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

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

[11] 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.

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