Opponent Modelling and Commercial Games

H.J. van den Herik, H.H.L.M. Donkers, P.H.M. Spronck
Department of Computer Science, Institute for Knowledge and Agent Technology, Universiteit Maastricht.
P.O.Box 616, 6200 MD, Maastricht, The Netherlands.
Email: herik,donkers,p.spronck@cs.unimaas.nl

Abstract- To play a game well a player needs to understand the game. To defeat an opponent, it may be sufficient to understand the opponent's weak spots and to be able to exploit them. In human practice, both elements (knowing the game and knowing the opponent) play an important role. This article focuses on opponent modelling independent of any game. So, the domain of interest is a collection of two-person games, multi-person games, and commercial games. The emphasis is on types and roles of opponent models, such as speculation, tutoring, training, and mimicking characters. Various implementations are given. Suggestions for learning the opponent models are described and their realization is illustrated by opponent models in game-tree search. We then transfer these techniques to commercial games. Here it is crucial for a successful opponent model that the changes of the opponent's reactions over time are adequately dealt with. This is done by dynamic scripting, an improvised online learning technique for games. Our conclusions are (1) that opponent modelling has a wealth of techniques that are waiting for implementation in actual commercial games, but (2) that the games' publishers are reluctant to incorporate these techniques since they have no definitive opinion on the successes of a program that is outclassing human beings in strength and creativity, and (3) that game AI has an entertainment factor that is too multifaceted to grasp in reasonable time.

1 Introduction

Ever since humans play games they desire to master the game played. Obviously, gauging the intricacies of a game completely is a difficult task; understanding some parts is most of the time the best a player can aim at. The latter means solving some sub-domains of a game. However, in a competitive game it may be sufficient to understand more of the game than the opponent does in order to win a combat. Remarkably, here a shift of attention may take place, since playing better than the opponent may happen (1) by the player's more extensive knowledge of the game or (2) by the player's knowledge of the oddities of the opponent. In human practice, a combination of (1) and (2) is part of the preparation of a top grandmaster in Chess, Checkers or Shogi. Opponent modelling is an intriguing part of a player's match preparation, since the preparing player tries to understand the preferences, strategies, skill, and weak spots of his[1] opponent.

In the following we distinguish between the player and the opponent if a two-person game is discussed. In multi-person games and in commercial games we will speak of agents. Opponent modelling is a research topic that was envisaged already a long time ago. For instance, in the 1970s chess programs incorporated a contempt factor, meaning that against a stronger opponent a draw was accepted even if the player was +0.5 ahead, and a draw was declined against a weaker opponent even when the player had a minus score. In the 1990s serious research in the domain of opponent modelling started [5, 19]. Nowadays opponent modelling also plays a part in multi-person games (collaboration, conspiracy, opposition) and in commercial games. Here we see a shift from opponent modelling towards subject modelling and even environmental entertainment modelling.

The course of the article is as follows. Section 2 defines types and roles of opponent models. In section 3 we provide a brief overview of the development of opponent models currently in use in Roshambo, the Iterated Prisoner's Dilemma, and Poker. We extrapolate the development to commercial Games. Section 4 lists six possible implementations of the opponent models. A main question is dealt with in section 5, viz. how to learn opponent models. We describe two methods, refer to a third one, and leave the others undiscussed. Section 6 focuses on the three implementations in game-tree search: OM search, PrOM search, and symmetric opponent modelling. Section 7 presents dynamic scripting as a technique for online adaptive game AI in commercial games. Finally section 8 contains our conclusions.

2 Roles of Opponent Models

In general, an opponent model is an abstracted description of a player or a player's behaviour in a game. There are many different types. For instance, a model can describe a player's preferences, his strategy, skill, capabilities, weaknesses, knowledge, and so on.

For each type we may distinguish two different roles in a game program. The first role is to model a (human or computer) opponent in such way that it informs the player appropriately in classical two-person games. Such opponent model can be *implicit* in the program's strategy or made *explicit* in some internal description. The task of such an opponent model is to understand and mimic the opponent's behaviour, in an attempt either to beat the opponent (see section 2.1) or to assist the opponent (section 2.2).

The second role is to provide an artificial opponent agent for the own agent (program or human player) using the program (see section 2.3), or an artificial agent that participates in an online multi-person game (section 2.4). Iteratively, such an opponent agent could bear in itself an opponent model of its own opponents. In most cases, the task of an

[1]In this article we use 'he' ('his') if both 'he' and 'she' are possible.

opponent model in this second role is to manifest an interesting and entertaining opponent to human players.

Regardless of its internal representation, an opponent model may range from statically defined in the program to dynamically adaptable. Opponent models that are dynamically adapted (or adapt themselves) to the opponent and other elements of the environment are to be preferred.

Below we will detail the four appearances in which opponent models are of use.

2.1 Speculation in heuristic search

The classical approach in Artificial Intelligence to board games, such as Chess, Checkers and Shogi, is heuristic search. It is based on the Minimax procedure for zero-sum perfect-information games as described by Von Neumann and Morgenstern [41]. However, the complexity of board games makes Minimax infeasible to be applied directly to the game tree. Therefore, the game tree is reduced in its depth by using a static heuristic evaluation, and quite frequently also in its breadth by using selective search. Moreover, during the detection of the best move to play next, much of the reduced game tree is disregarded by using $\alpha\beta$ pruning and other search enhancements. Actual game playing in this approach consists of solving a sequence of reduced games. Altogether, the classical approach has proven to be successful in Chess, Checkers, and a range of other board games.

In the classical approach, reasoning is based on defending against the worst case and attempting to achieve the best case. However, because heuristic search is used, it is not certain that the worst case and the best case are truly known. It means that it might be worthwhile to use additional knowledge during heuristic search in order to increase the chance to win, for instance, knowledge of the opponent. It is clear that humans use their knowledge of the opponent during game playing.

There are numerous ways in which knowledge of the (human) opponent can be used to improve play by heuristic search. One can use knowledge of the opponent's preferences or skills to force the game into positions that are considered to be less favourable to the opponent than to oneself. In the case that a player is facing a weak position, the player may try to speculate on positions in which the opponent is more likely to make mistakes. If available, a player may use the opponent's evaluation function to speculate (or even calculate) the next move an opponent will make and thus adopt its strategy to find the optimal countermoves. We will concentrate on the last approach in section 5.

2.2 Tutoring and Training

An opponent model can be used to assist the human player. We discuss two different usages: tutoring and training. Commercial board game programs (can) increase their attractiveness by offering such functionality.

In a tutoring system [20], the program can use the model of the human opponent to teach the player some aspects of the game in a personalized manner, depending on the type of knowledge present in the opponent model. If the model includes the player's general weaknesses or skills, it can be used to lead apprentices during a game to positions that help them to learn from mistakes. When the model includes the strategy or preferences of the player, then this knowledge can be employed to provide explicit feedback to the user during play, either by tricking the player into positions in which a certain mistake will be made and explicitly corrected by the program, or by providing verbal advice such as: ”you should play less defensive in this stage of the game”.

A quite different way to aid the apprentice is to provide preset opponent types. Many game programs offer an option to set the playing strength of the program. Often, this is arranged by limiting the resources (e.g., time, search depth) available to the program. Sometimes, the preferences of a program can be adjusted to allow a defensive or aggressive playing style. An explicit opponent model could assist even the experienced players to prepare themselves for a game against a specific opponent. In order to be useful, the program should in this case be able to learn a model of a specific player. In Chess, some programs (e.g., CHESS ASSISTANT[2]) offer the possibility to adjust the opening book to a given opponent, on the basis of previously stored game records.

2.3 Non-player Characters

The main goal in commercial computer games is not to play as strong as possible but to provide entertainment. Most commercial computer games, such as computer roleplaying games (CRPGs) and strategy games, situate the human player in a virtual world that is populated by computer-controlled agents, which are called ”non-player characters” (NPCs). These agents may fulfil three roles: (i) as a companion, (ii) as an opponent, and (iii) as a neutral, background character. In the first two roles, an opponent model of the human player is needed. In practice, for most (if not all) commercial games this model is implemented in an implicit way. The third role, however commercially interesting, is not relevant in the subject area of opponent modelling, and thus it is not discussed below.

In the companion role, the agent must behave according to the expectations of the human player. For instance, when the human player prefers a stealthy approach to dealing with opponents agents, he will not be pleased when the computer-controlled companions immediately attack every opponent agent that is near. If the companions fail to predict with a high degree of success what the human player desires, they will annoy the human player, which is detrimental for the entertainment value of the game. Nowadays, companion agents in commercial games use an implicit model of the human player, which the human player can tune by setting a few parameters that control the behaviour of the companion (such as ”only attack when I do too” or ”only use ranged weapons”).

In the opponent role, the agent must be able to match the

[2]See: http://store.convekta.com.

playing skills of the human player. If the opponent agent plays too weak a game against the human player, the human player loses interest in the game [34]. In contrast, if the opponent agent plays too strong a game against the human player, the human player gets stuck in the game and will quit playing too [25]. Nowadays, commercial games provide a 'difficulty setting' which the human player can use to set the physical attributes of opponent agents to an appropriate value (often even during gameplay). However, a difficulty setting does not resolve problems when the quality of the *tactics* employed by opponent agents is not appropriate for the skills of the human player.

The behaviour of opponent agents in commercial games is designed during game development, and does not change after the game has been released, i.e., it is static. The game developers use (perhaps unconsciously) a model of the human player, and a program behaviour for the opponent agents appropriate for this model. As a consequence, the model of the human player is implicit in the programmed agent behaviour. Since the agent behaviour is static, the model is static. In reality, of course, human players may be very different, and thus it is to be expected that for most games a static model is not ideal. A solution to this problem would be that the model, and thus the behaviour of the opponent agent, is dynamic. However, games' publishers are reluctant to release games where the behaviour of the opponent agents is dynamic, since they fear that the agents may learn undesirable behaviour after the game's release.

The result is that, in general, the behaviour of opponent agents is unsatisfying to human players. Human players prefer to play against their own kind, which is a partial explanation for the popularity of multi-person games [33].

2.4 Multi-person games

In multi-person games, opponent models can be used to provide NPCs as well. Clearly, the problem mentioned in the previous subsection is also present here, only in a much harder form for the opponent agents, since they have to deal with many human players with many different levels of skills in parallel.

Yet another role of opponent models comes into sight in multi-person games. There are situations in which a player is not able or willing to continue playing, but the character representing the player remains 'alive' inside the game. Such a situation could arise from (i) a connection interrupt in an online game, (ii) a 'real-world' interruption of the human player, or (iii) a human player wanting to enter multiple copies of himself in the game. An opponent model could be used in those instances to take over control of the human's alter-ego in the game, while mimicking the human player's behaviour. Of course, such a model should be adaptable to the player's characteristics.

3 Towards Commercial Games

Below we deal with three actual implementations of opponent models (3.1), viz. in Roshambo, the Iterated Prisoner's Dilemma, and Poker. From here we extrapolate the development to commercial games (3.2) with an emphasis on adaptive game AI.

3.1 Opponent models used Now

Many of the usages of opponent models as presented in the previous section are still subject of current and future research. However, in a number of games, adaptive opponent models are an essential part of successful approaches. It is especially the case in iterated games. These are mostly small games that are played a number of times in sequence; the goal is to win the most games on average. Two famous examples of iterated games are Roshambo (Rock-Paper-Scissors) and the Iterated Prisoner's Dilemma (IPD). Both games consist of one simultaneous move after which the score is determined. Roshambo has three options for each move and zero-sum scores, IPD has only two options, but has nonzero-sum scores. Both games are currently played by computers in tournaments.

In Roshambo, the optimal strategy in an infinitely repeated game is to play randomly. However, in an actual competition with a finite number of repetitions, a random player will end up in the middle of the pack and will not win the competition. Strong programs, such as IOCAINE POWDER [12] apply opponent modelling in order to predict the opponent's strategy, while at the same time they attempt to be as unpredictable as possible.

Although IPD seems not so different from Roshambo, the opponent model must take another element into account: the willingness of the opponent to cooperate. In IPD, the players receive the highest payoff if both players cooperate. Since the first IPD competition by Axelrod in 1979 [2], the simple strategy 'Tit-for-Tat' has won most of the competitions [23]. However, the 2004 competition was won by a team that used multiple entries and recognition codes to 'cheat'. Although this is not strictly opponent modelling, the incident caused the birth of a new IPD competition at CIG'05 in which multiple entries and recognition codes are allowed. IPD illustrates an aspect of opponent modelling that will play a role, in particular, in multi-person games, viz., how to measure the willingness to cooperate and how to tell friendly from hostile opponents?

A more complex iterated game is Poker. The game offers more moves than Roshambo and IPD, involves more players in one game and has imperfect information. However, the game does not need heuristic search to be played. Although many Poker-playing programs exist that do not use any opponent model, the strong and commercially available program POKI ([3]) is fully based on opponent-modelling. Schaeffer states: "No poker strategy is complete without a good opponent-modelling system. A strong poker player must develop a dynamically changing (adaptive) model of each opponent, to identify potential weaknesses." Opponent modelling is used with two distinct goals: to predict the next move of each opponent and to estimate the strength of each opponent's hand.

3.2 The Future is in Commercial Games

The answer to the question "Are adaptive opponent models really necessary?" is that adaptive opponent models are sorely needed to deal with the complexities of state-of-the-art commercial games.

Over the years commercial games have become increasingly complex, offering realistic worlds, a high degree of freedom, and a great variety of possibilities. The technique of choice used by game developers for dealing with a game's complexities is rule-based game AI, usually in the form of scripts [29, 40]. The advantage of the use of scripts is that scripts are (1) understandable, (2) predictable, (3) tuneable to specific circumstances, (4) easy to implement, (5) easily extendable, and (6) useable by non-programmers [40, 39]. However, as a consequence of game complexity, scripts tend to be quite long and complex [4]. Manually-developed complex scripts are likely to contain design flaws and programming mistakes [29].

Adaptive game AI changes the tactics used by the computer to match or exceed the playing skills of the particular human player it is pitted against, i.e., adaptive game AI changes its implicit model of the human player to be more successful. Adaptive game AI can ensure that the impact of the mistakes mentioned above is limited to only a few situations encountered by the player, after which their occurrence will have become unlikely. Consequently, it is safe to say that the more complex a game is, the greater the need for adaptive game AI [13, 24, 16]. In the near future game complexity will only increase. As long as the best approach to game AI is to design it manually, the need for adaptive game AI, and thus for opponent modelling, will increase accordingly.

4 How to Model Opponents

The internal representation of an opponent model depends on the type of knowledge that it should contain and the task that the opponent model should perform. Artificial Intelligence offers a range of techniques to build such models

4.1 Evaluation functions

In the context of heuristic search, an opponent model can concentrate on the player's preferences. These preferences are usually encoded in a static heuristic evaluation function that provides a score for every board position. An opponent model can consist of a specific evaluation function. The evaluation function can either be hand-built on the basis of explicit knowledge or machine-learned on basis of game records.

4.2 Neural networks

The preferences of an opponent can also be represented by a neural network or any other machine-learned function approximator. Such a network can be learned from game records or actual play. However, neural networks can also be used to represent other aspects of the opponent's behaviour. They could represent the difficulty of positions for a specific opponent [28], or the move ordering preferred. The Poker program POKI also uses neural networks to represent the opponent model.

4.3 Rule-based models

A rule-based model consists of a series of production rules, that couple actions to conditions. It is a reactive system, that tests environment features to generate a response. A rule-based model is easily implemented. It is also fairly easy to be maintained and analysed.

4.4 Finite-State Machine

A finite-state machine model consists of a collection of states, which represent situations in which the model can exist, with defined state transitions that allow the model to go into a new state. The state transitions are usually defined as conditions. The model's behaviour is defined separately for each state.

4.5 Probabilistic models

The finite-state machine model can be augmented by probabilistic transitions. It results in a probabilistic opponent model. This kind of model is especially useful in games with imperfect information, such as Poker, and most commercial games.

A second probabilistic opponent model consists of a mixture of other models (opponent types). In these models, the strategy of the opponent is determined by first generating a random number (which may be biased by certain events) and then on the basis of the outcome selecting one type out of a set of predefined opponent types.

4.6 Case-based models

A case-based model consists of a case base with samples of situations and actions. By querying the case base, the cases corresponding with the current situation are retrieved, and an appropriate action is selected by examining the actions belonging to the selected cases. An advantage of a case-based model, is that the model can be easily updated and expanded by allowing it to collect automatically new cases while being used.

5 Learning Opponent Models

A compelling question is: can a program learn an opponent model? Below we describe some research efforts made in this domain. They consist of learning evaluation functions (5.1), learning probabilistic models (5.2), and learning opponent behaviour (5.3).

5.1 Learning evaluation functions

There are two basic approaches to the learning of opponent models for heuristic search: (1) to learn an evaluation function, a move ordering, the search depth and other search preferences used by a given opponent, and (2) to learn the opponent's strategy, which means to learn directly the move that the opponent selects at every position.

The first approach has been studied in computer chess, especially the learning of evaluation functions. Although the goal often is to obtain a good evaluation function for $\alpha\beta$search, similar techniques can be used for obtaining the evaluation function of an opponent type. For instance, Anantharaman [1] describes a method to learn or tune an evaluation function with the aid of a large set of positions and the moves selected at those positions by master-level human players. The core of the approach is to adapt weights in an evaluation function by using a linear discriminant method in such a way that a certain score of the evaluation function is maximized. The evaluation function is assumed to have the following form: $V(h) = \sum_i W_i\, C_i(h)$. The components $C_i(h)$ are kept constant, only the weights W_i are tuned. The method was used to tune an evaluation function for the program DEEP THOUGHT, a predecessor of DEEP BLUE. Although the method obtained a better function than the hand-tuned evaluation function of the program, the author admits that it is difficult to avoid local maxima. Fürnkranz [15] gives an overview of machine learning in computer chess, including several other methods to obtain evaluation functions from move databases.

5.2 Learning probabilistic models

The learning of opponent-type probabilities during a game is limited since the number of observations is low. It can, however, be useful to adapt probabilities that were achieved earlier, for instance by offline learning. Two types of online learning can be distinguished: a *fast* one in which only the best move of every opponent type is used, and a *slow* one in which the search value of all moves is computed for all opponent types.

Fast online learning happens straightforwardly as follows: start with the prior obtained probabilities. At every move of the opponent do the following: for all opponent types detect whether their best move is equal to the actually selected move. If so, reward that opponent type with a small increase of the probability. If not, punish the opponent type. The size of the reward or punishment should not be too large because this type of learning will lead to the false supremacy of one of the opponent types. This type of incremental learning is also applied in the prediction of user actions [8].

Slow online learning would be an application of the naive Bayesian learner (see [9]). A similar approach is used in learning probabilistic user profiles [30]. Slow online learning works as follows. For all opponent types ω_i, the sub-game values $v_{\omega_i}(h + m_j)$ of all possible moves m_j at position h are computed. These values are transformed into conditional probabilities $\Pr(m_j|\omega_i)$, that indicate the "willingness" of the opponent type to select that move. This transformation can be done in a number of ways. An example is the method by Reibman and Ballard [31]: first determine the rank $r(m_j)$ of the moves according to $v_{\omega_i}(h+m_j)$ and then assign probabilities:

$$\Pr(m_j|\omega_i) = \frac{(1-P_s)^{r(m_j)-1}\cdot P_s}{\sum_k (1-P_s)^{r(m_k)-1}\cdot P_s} \qquad (1)$$

P_s ($\in (0,1]$) can be interpreted as the likeliness of the opponent type not to deviate from the best move: the higher P_s, the higher the probability on the best move. It is however also possible to use the actual values of $v_{\omega_i}(h + m_j)$. Now Bayes' rule is used to compute the opponent-type probabilities given the observed move of the opponent.

$$\Pr(\omega_i|m_\Omega(h)) = \frac{\Pr(m_\Omega(h)|\omega_i)\Pr(\omega_i)_t}{\sum_k \Pr(m_\Omega(h)|\omega_k)\Pr(\omega_k)_t} \qquad (2)$$

These a-posteriori probabilities are used to update the opponent-type probabilities.

$$\Pr(\omega_i)_{t+1} = (1-\gamma)\Pr(\omega_i)_t + \gamma \Pr(\omega_i|m_\Omega(h)) \quad (3)$$

In this formula, parameter γ ($\in [0, 1]$) is the learning factor: the higher γ, the more influence the observations have on the opponent-type probabilities. The approach is called *naive* Bayesian learning, because the last formula assumes that the observations at the subsequent positions in the game are independent.

5.3 Learning opponent behaviour

Direct learning of opponent strategies is studied extensively on iterated games [14]. For learning opponent strategies in Roshambo we refer to Egnor [12]. General learning in repeated games is studied, for example, by Carmel and Markovitch [7].

6 Opponent Models in Game-Tree Search

Junghanns [22] gave an overview of eight problematic issues when using $\alpha\beta$ in game-tree search. He also listed alternative algorithms that aimed at overcoming one or more of these problems. The four most prominent problems with $\alpha\beta$ are: (1) the *heuristic-error* problem (heuristic values are used instead of real game values), (2) the *scalar-value* problem (only a single scalar value is used to express the value of an arbitrarily complex game position), (3) the *value-backup* problem (lines leading to several good positions are preferable to a line that leads to a single good position), and (4) the *opponent* problem (knowledge of the opponent is not taken into account).

The first attempt to use rudimentary knowledge of the opponent in heuristic search is the approach by Slagle and Dixon [35] in 1970. At the base of their M & N-search method lies the observation that it is wise to favour positions in which there are several moves with good values over positions in which there is only one move with a good value. In 1983, Reibman and Ballard [31] assume that the opponent sometimes is fallible: there is a chance in each position that the opponent selects a non-rational move. In their model, the probability that the opponent selects a specific move depends on the value of that move and on the degree of fallibility of the opponent.

Below we will discuss three further approaches of dealing with Junghanns's fourth problem; viz. Opponent-Model (OM) search, Probabilistic OM (PrOM) search, and symmetric opponent modelling.

6.1 OM search

The main assumption of OM search is that the opponent (called MIN) uses a Minimax algorithm (or equivalent) with an evaluation function that is known to the first player (called MAX). Also the depth of the opponent's search tree and the opponent's move order are assumed to be known. This knowledge is used to construct a derivative of Minimax in which MAX maximizes at max nodes, but selects at min nodes the moves that MIN will select, according to MAX' knowledge of MIN's evaluation function.

For a search tree with even branching factor w and fixed depth d, OM search needs $n = w^{\lceil d/2 \rceil}$ evaluations for MAX to determine the search-tree value: at every min node, only one max child has to be investigated but at every max node, all w children must be investigated. Because the search-tree value of OM search is defined as the maximum over all these n values for MAX, none of these values can be missed. This means that the efficiency of OM search depends on how efficient the values for MIN can be obtained.

A straightforward and efficient way to implement OM search is by applying *$\alpha\beta$ probing*: at a min node perform $\alpha\beta$ search with the opponent's evaluation function (the *probe*), and perform OM search with the move that $\alpha\beta$ search returns; at a max node, maximize over all child nodes. The probes can in fact be implemented using any enhanced minimax search algorithm available, such as MTD(f). Because a separate probe is performed for every min node, many nodes are visited during multiple probes. (For example, every min node P_j on the principal variation of a node P will be probed at least twice.) Therefore, the use of transposition tables leads to a major reduction of the search tree. The search method can be improved further by a mechanism called β-pruning (see Figure 1).

The assumptions that form the basis of OM search give rise to two types of risk. The first type of risk is caused by a player's imperfect knowledge of the opponent. When MIN uses an evaluation function different from the one assumed by MAX (or uses a different search depth or even a different move ordering), MIN might select another move than the move that MAX expects. This type of risk has been described in detail and thoroughly analyzed in [18, 21]. The second type of risk arises when the *quality* of the evaluation functions used is too low. The main risk appears to occur when the MAX player's evaluation function overestimates a position that is selected by MIN. This position may then act as an attractor for many variations, resulting in a bad performance. To protect the OM search against such a performance the notion of *admissible* pairs of evaluation functions is needed: (1) MAX's function is a better profitability estimator than MIN's, and (2) MAX's function never overestimates a position that MIN's does not overestimate likewise [11].

```
**OmSearchBPb**$(h, \beta)$
1   **if** $(h \in E)$ **return** $(V_0(h), \textbf{null})$
2   **if** $(p(h) = \text{MAX})$
3       $L \leftarrow m(h)$ ; $m \leftarrow$ firstMove$(L)$
4       $m^* \leftarrow m$ ; $v_0^* \leftarrow -\infty$
5       **while** $(m \neq \textbf{null})$
6           $(v_0, mm) \leftarrow$ **OmSearchBPb**$(h + m, \beta)$
7           **if** $(v_0 > v_0^*)$ $v_0^* \leftarrow v_0$ ; $m^* \leftarrow m$
8           $m \leftarrow$ nextMove$(L)$
9   **if** $(p(h) = \text{MIN})$
10      $(v_{op}^*, m^*) \leftarrow \alpha\beta$**-Search**$(h, -\infty, \beta, V_{op}(\cdot))$
11      $(v_0^*, mm) \leftarrow$ **OmSearchBPb**$(h + m^*, v_{op}^* + 1)$
12  **return** $(v_0^*, m^*)$
```

Figure 1: β-pruning OM search with $\alpha\beta$ probing.

6.2 PrOM search

In contrast to OM search that assumes a fixed evaluation function of the opponent, PrOM search [10] uses a model of the opponent that includes uncertainty. The model consists of a set of evaluation functions, called *opponent types*, together with a probability distribution over these functions. More precisely, PrOM search is based on the following four assumptions:

(1) MAX has knowledge of n different *opponent types* $\omega_0 \ldots\ \omega_{n-1}$. Each opponent type ω_i is a minimax player that is characterized by an evaluation function V_{ω_i}. MAX is using evaluation function V_0. For convenience, one opponent type (ω_0) is assumed to use the same evaluation function as MAX uses ($V_{\omega_0} \equiv V_0$).

(2) All opponent types are assumed to use the same search-tree depth and the same move ordering as MAX.

(3) MAX has subjective probabilities $\Pr(\omega_i)$ on the range of opponents, such that $\sum_i \Pr(\omega_i) = 1$.

(4) MIN is using a *mixed strategy* which consists of the n opponent-type minimax strategies. At every move node, MIN is supposed to pick randomly one strategy according to the opponent-type probabilities $\Pr(\omega_i)$.

The fourth assumption is a crucial one because it determines the semantics of the opponent model: the mixed strategy acts as an approximation of opponent's real strategy. The subjective probability of every opponent type acts as the amount of MAX's belief that this opponent type resembles the opponent's real behaviour.

The applicability of $\alpha\beta$ probing in PrOM search is clear (see Figure 2). The values of $v_{\omega_i}(P)$ and the best move P_j for opponent type ω_i at min node P, can safely be obtained by performing $\alpha\beta$ search at node P, using evaluation function $V_{\omega_i}(\cdot)$. Notice that an $\alpha\beta$ probe has to be performed for every opponent type separately. These $\alpha\beta$ probes can be improved by a number of search enhancements. If transposition tables are used, then a separate table is needed per opponent type. The transposition table for an opponent type must not be cleared at the beginning of each probe, but only

```
PromSearchBPb($h, \bar{\beta}$)
    if ($h \in E$) return ($V_0(h)$, null)
    if $p(h) = $ MAX
        $L \leftarrow m(h)$ ; $m \leftarrow$ firstMove($L$) ; $m_0^* \leftarrow m$
        $v_0^* \leftarrow -\infty$
        while ($m \neq$ null)
            $(v_0, mm) \leftarrow$ PromSearchBPb($h + m, \bar{\beta}$)
            if ($v_0 > v_0^*$) $v_0^* \leftarrow v_0$ ; $m_0^* \leftarrow m$
            $m \leftarrow$ nextMove($L$)
    if $p(h) = $ MIN
        $L \leftarrow \varnothing$
        for $i \in \{0, \ldots, n-1\}$
            $(\bar{v}_i^*, \bar{m}_i^*) \leftarrow$ $\alpha\beta$-Search($h, -\infty, \bar{\beta}_i, V_i(\cdot)$)
            $L \leftarrow L \cup \{\bar{m}_i^*\}$
        $v_0^* \leftarrow 0$; $m_0^* \leftarrow$ null ; $m \leftarrow$ firstMove($L$)
        while ($m \neq$ null)
            for $i \in \{0, \ldots, n-1\}$
                if ($m = \bar{m}_i^*$) $\bar{\beta}_i \leftarrow \bar{v}_i^* + 1$ else $\bar{\beta}_i \leftarrow \infty$
            $(v_0, mm) \leftarrow$ PromSearchBPb($h + m, \bar{\beta}$)
            for $i \in \{0, \ldots, n-1\}$
                if ($m = \bar{m}_i^*$) $v_0^* \leftarrow v_0^* + \Pr(\omega_i)\, v_0$
            $m \leftarrow$ nextMove($L$)
    return ($v_0^*, m_0^*$)
```

Figure 2: β-pruning PrOM search with $\alpha\beta$ probing.

at the start of the PrOM search so that knowledge of the search tree is shared between the subsequent probes for the same opponent type.

Because of the usage of multiple opponent models, the computational efforts for PrOM search are larger than those needed for OM search. However, the risk while using PrOM search is lower than while using OM search, when MAX uses the own evaluation functions as one of the opponent types. Experimental results indicate that when computational efforts are disregarded, PrOM search performs better than OM search with the same amount of knowledge of the opponent and with the same search depth.

6.3 Symmetric Opponent Modelling

Instead of the asymmetric opponent models in OM search and PrOM search, it might be more natural to assume that both players use an opponent model of each other of which they are mutually aware. In the context of heuristic search it means that both players agree that they have different (i.e., non-opposite) evaluation values for positions. The key concept is common interest. Evaluation values are based on many factors of a position. Some of these factors are pure competitive, such as the number of black pieces on a chess board, other factors are of interest of both players. Carmel and Markovitch [6] give an example for the game of checkers. Another example is the degree to which a chess position is 'open' or 'closed'. An open position (in which many pieces can move freely) is favoured by many players over closed positions. Therefore, the openness of a position is a common interest of both players.

Assume that the competitive factors of a position count S and the common-interest factors count C, then the value for the first player would be $C+S$. In the standard zero-sum approach, the opponent would be assumed to use the value $-(C+S)$ for the same position, which would mean that the opponent would award the common interest of the position with $-C$. However, it seems more natural that the second player uses the value $C-S$ for the position. In the model of Carmel and Markovitch [6], only one of the players is assumed to be aware of this fact. However, why should we not assume knowledge symmetry and let both players agree on the size of C and S? When the two players receive different pay-offs (e.g., $C+S$ and $C-S$) and these pay-offs are common knowledge, we achieve a nonzero-sum game of perfect information. In such a game there is both opponent modelling and knowledge symmetry, leading to symmetric opponent modelling. It should be noted that in any nonzero-sum game, it is possible to describe the pay-offs in terms of competitive and common-interest factors. If the first player receives A and the second player B, the common interest C is equal to $(A+B)/2$ and the competitive part S is equal to $(A-B)/2$.

The use of a nonzero-sum game as a means of symmetric opponent model introduces two challenges: (1) how to select the best equilibrium and (2) how to search efficiently. In contrast to zero-sum games in which all equilibria have the same value, in nonzero-sum games equilibria can co-exist with different values. Although all equilibria of a nonzero-sum game of perfect information can be found easily by backward induction (similar to Minimax, see Figure 3), the selection of the best one among them is hard. Moreover, the basic backward induction procedure is not feasible for large game trees, so an $\alpha\beta$-like pruning mechanism and other enhancements are asked for.

```
BackInd($h$)
1   if ($h \in E$) return ($V_1(h), V_2(h)$, null)
2   $v^* \leftarrow -\infty$, $L \leftarrow \varnothing$
3   for $m \in m(h)$
4       $(\cdot, v_1, v_2) \leftarrow$ BackInd($h + m$)
5       if ($v_{p(h)} > v^*$)   $L \leftarrow \{(m, v_1, v_2)\}$
6                               $v^* \leftarrow v_{p(h)}$
7       if ($v_{p(h)} = v^*$)   $L \leftarrow L \cup \{(m, v_1, v_2)\}$
8   select $(m, v_1, v_2) \in L$
9   return ($m, v_1, v_2$)
```

Figure 3: Backward Induction.

Both tasks can be helped by restricting ourselves to games with bounded common interest. These are nonzero-sum games where the value of C is bounded to an interval $[-B/2, B/2]$ around zero and where B is (much) smaller than the largest absolute value of S in any pay-off. The profit of using this bound is that it allows for pruning during game-tree search since the difference between the value for player 1 and 2 in each equilibrium is restricted to B. Moreover, the range of values of those equilibria is restricted, as we will show below. We will call this types of games: *BCI* *games* (Bounded Common Interest games). It can be proven that the bound B on the common interest puts a bound on the values that the equilibria can take. For trees of depth d, the range is $v^* \pm B(d-1)$ for Player 1 and $-v^* \pm Bd$ for Player 2. These ranges indicate the 'damage' that has to be feared when selecting a suboptimal equilibrium. The ranges can also be used to rule out moves that cannot lead to any equilibrium.

The bound on common interest, B, also allows for pruning in an $\alpha\beta$-like manner. This pruning is based on the fact that in case of bounded common interest, the difference between the values for Player 1 and Player 2 is also bounded at any position in the tree. So, the value for one player can be used to predict the value for the other player, and bounds on the value for one player can be used to bound the value for the other player. In this way, shallow and deep pruning is possible, but the amount of pruning depends on the value of B and on the depth of the tree. With every additional level of depth, the bounds on the values are widened by B, leading to less and less pruning.

Two-player nonzero-sum games of perfect information can be used for symmetric opponent modelling. A fundamental difference with the standard zero-sum games is that several equilibria can exist in one game and that selecting a good equilibrium is very hard. We proved that when bounded common interest is assumed, the range of values that equilibria can take on is also bounded. Furthermore, BCI games allow pruning during the determination of the equilibria in a game tree. BCI games offer an alternative to Minimax-based algorithms and to Opponent-Model Search in heuristic search, but experimental evidence has to be collected on the practical usability and effectiveness of the approach. The BCI game also offers an opportunity to apply a range of search techniques from Artificial Intelligence to a class of games that is of interest to a broader audience than the traditional one.

7 Opponent Models with Dynamic Scripting

In this section we present dynamic scripting as a technique that is designed for the implementation of online adaptive game AI in commercial games. Dynamic scripting uses a probabilistic search to update an implicit opponent model of a human player, to be able to generate game AI that is appropriate for the player. Those interested in a more detailed exposition of dynamic scripting are referred to [37].

Dynamic scripting is an unsupervised online learning technique for games. It maintains several rulebases, one for each class of computer-controlled agents in the game. The rules in the rulebases are manually designed using domain-specific knowledge. Every time a new agent of a particular class is generated, the rules that comprise the script controlling the agent are extracted from the corresponding rulebase. The probability that a rule is selected for a script is proportional to the weight value that is associated with the rule. The rulebase adapts by changing the weight values to reflect the success or failure rate of the associated rules in scripts. A priority mechanism can be used to let certain rules take precedence over other rules. Dynamic scripting has been demonstrated to be fast, effective, robust, and efficient. The dynamic scripting process is illustrated in Figure 4 in the context of a game.

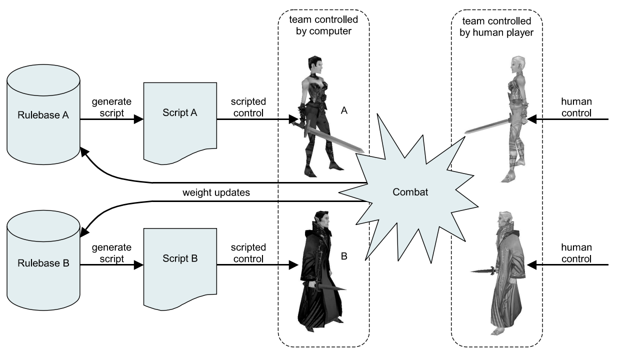

Figure 4: Dynamic scripting.

The learning mechanism in the dynamic-scripting technique is inspired by reinforcement learning techniques [38, 32]. 'Regular' reinforcement learning techniques, such as TD-learning, in general need large amounts of trials, and so are usually not sufficiently efficient to be used in games [27, 26]. Reinforcement learning is suitable to be applied to games if the trials occur in a short timespan (as in the work by [17], where fight movements in a fighting game are learned). However, for the learning of complete tactics, such as scripts, a trial consists of observing the performance of a tactic over a fairly long period of time. Therefore, for the online learning of tactics in a game, reinforcement learning will take too long to be particularly suitable. In contrast, dynamic scripting has been designed to learn from a few trails only.

In the dynamic-scripting approach, learning proceeds as follows. Upon completion of an encounter (i.e., a fight), the weights of the rules employed during the encounter are adapted depending on their contribution to the outcome. Rules that lead to success are rewarded with a weight increase, whereas rules that lead to failure are punished with a weight decrease. The remaining rules are updated so that the total of all weights in the rulebase remains unchanged.

Weight values are bounded by a range $[W_{min}, W_{max}]$. The size of the weight change depends on how well, or how badly, a computer-controlled agent behaved during an encounter with the human player. It is determined by a fitness function that rates an agent's performance as a number in the range $[0, 1]$. The fitness function is composed of four indicators of playing strength, namely (1) whether the team to which the agent belongs won or lost, (2) whether the agent died or survived, (3) the agent's remaining health, and (4) the amount of damage done to the agent's enemies. The new weight value is calculated as $W + \triangle W$, where W is the original weight value, and the weight adjustment $\triangle W$ is expressed by the following formula:

$$\triangle W = \begin{cases} -\lfloor P_{max}\dfrac{b-F}{b}\rfloor & \{F < b\} \\ \lfloor R_{max}\dfrac{F-b}{1-b}\rfloor & \{F \geq b\} \end{cases} \tag{4}$$

In equation 4, $R_{max} \in \mathbb{N}$ and $P_{max} \in \mathbb{N}$ are the maximum reward and maximum penalty respectively, F is the agent fitness, and $b \in \langle 0, 1 \rangle$ is the break-even value. At the break-even point the weights remain unchanged.

In its pure form, dynamic scripting does not try to match the human player's skill, but tries to play as strongly as possible against the human player. That, however, is in conflict with the goal of commercial games, namely providing entertainment.

A variation on dynamic scripting allows it to adapt to meet the level of skill of the human player. This variation uses a fitness scaling technique that ensures that the game AI enforces an 'even game', i.e., a game where the chance to win is equal for both players. The domain knowledge stored in the rulebases used by dynamic scripting has been designed to generate effective behaviour at all times. Therefore, even when enhanced with a fitness-scaling technique, against a mediocre player dynamic scripting does not exhibit stupid behaviour interchanged with smart behaviour to enforce an even game, but it exhibits mediocre behaviour at all times.

We called the difficulty-scaling technique that was the most successful 'top culling'. Top culling works as follows.

In dynamic scripting, during the weight updates, the maximum weight value W_{max} determines the maximum level of optimisation that a learned strategy can achieve. A high value for W_{max} allows the weights to grow to large values, so that after a while the most effective rules will almost always be selected. This will result in scripts that are close to a presumed optimum. With top culling activated, weights are allowed to grow beyond the value of W_{max}. However, rules with weights higher than W_{max} will be excluded from the script generation process. If the value of W_{max} is low, effective rules will be quickly excluded from scripts, and the behaviour exhibited by the agent will be inferior (though not ineffective).

To determine the value of W_{max} that is needed to generate behaviour at exactly the level of skill of the human player, top culling automatically changes the value of W_{max}, with the intent to enforce an even game. It aims at having a low value for W_{max} when the computer wins often, and a high value for W_{max} when the computer loses often. The implementation is as follows. After the computer has won a fight, W_{max} is decreased by W_{dec} per cent (with a lower limit equal to the initial weight value W_{init}). After the computer has lost a fight, W_{max} is increased by W_{inc} per cent.

To evaluate the effect of top culling to dynamic scripting, we employed a simulation of an encounter of two teams in a complex computer roleplaying game, closely resembling the popular BALDUR'S GATE games. We used this environment in earlier research to demonstrate the efficiency of dynamic scripting [37]. Our evaluation experiments aimed at assessing the performance of a team controlled by the dynamic scripting technique using top culling, against a team controlled by static scripts. In the simulation, we pitted the dynamic team against a static team that uses one of five, manually designed, basic strategies (named 'offensive', 'disabling', 'cursing', 'defensive', and 'novice'), or one of three composite strategies (named 'random team', 'random agent' and 'consecutive').

Of the eight static team's strategies the most interesting in the present context is the 'novice' strategy. This strategy resembles the playing style of a novice BALDUR'S GATE player. While the 'novice' strategy normally will not be defeated by arbitrarily picking rules from the rulebase, many different strategies exist that can be employed to defeat it, which the dynamic team will quickly discover. Without difficulty-scaling, the dynamic team's number of wins will greatly exceed its losses. Details of the experiment are found in [36].

For each of the static strategies, we ran 100 tests without top culling, and 100 tests with top culling. We recorded the number of wins of the dynamic team for the last 100 encounters. Histograms for the tests with the 'novice' strategy are displayed in Figure 5. From the histogram it is immediately clear that top culling ensures that dynamic scripting plays an even game (the number of wins of the dynamic player is close to 50 out of 100), with a very low variance. The same pattern was observed against all the other investigated tactics. We can therefore conclude that dynamic scripting, enhanced with top culling, is successful in automatically discovering a well-working implicit model of the human player. As a perk, this model will be automatically updated when the human player learns new behaviour.

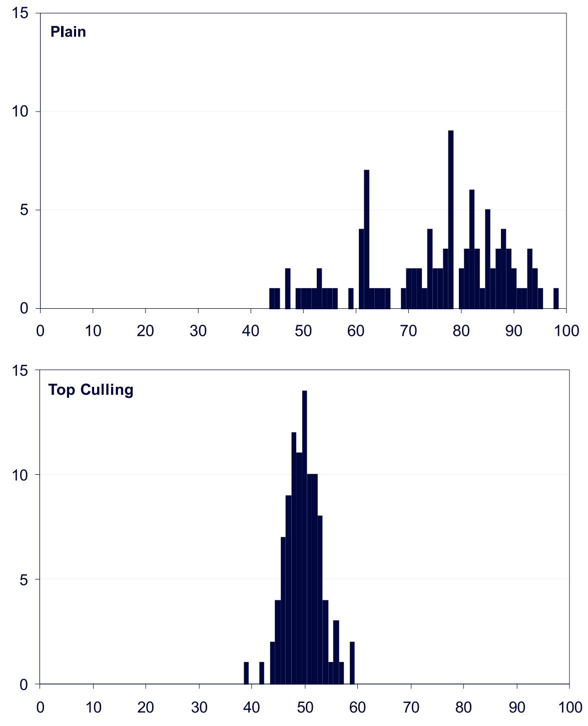

Figure 5: Histograms of 100 tests of the achieved number of wins in 100 fights, against the 'novice' strategy. The top graph is without difficulty scaling, the bottom graph with the application of top culling.

8 Conclusions

For human beings, opponent modelling is an essential and intriguing part of a player's match preparation. In this contribution we have discussed how opponent models can be implemented in computer programs. We investigated the full collection of games, ranging from classical two-person games via multi-person games to commercial games. Although opponent modelling is on the research table almost from the beginning of computer game research, serious implementation started in 1993 and the realization of most ideas is still in its infancy. There are three successful instances of actual implementation, viz. in Roshambo, the Iterated Prisoner's Dilemma, and Poker. Yet we may conclude that there is a wealth of techniques that are waiting for implementation in actual games.

In the contribution we have discussed OM search, PrOM search, and symmetric opponent modelling for classical games, and dynamic scripting for commercial games. In the last application (i.e., dynamic scripting and in particular in top culling) we see a shift in the goal to be reached. In classical games opponent modelling is used to raise the playing strength, in commercial games opponent modelling has as its main goal raising the entertainment factor. Currently, it is not clear to what extent both goals (i.e., raising the playing strength and raising the entertainment factor) are interchangeable. This is a topic of future research. However, at this moment it leads us to the conclusion that the undecided research question has as consequence that commercial games' publishers are reluctant to incorporate these techniques since they do not know whether a program that is outclassing human beings in strength and creativity will also raise the level of entertainment. From the research performed so far in this new area we may conclude that game AI (our current research tool for raising entertainment) has an entertainment factor that is too multifactored to grasp in reasonable time. Hence, new ideas should be developed that bring us a new classification of entertainment factors (types and roles) and will shed new light on the trade-off between issues on raising the playing strength and raising the entertainment.

Bibliography

[1] Anantharaman, T. (1997). Evaluation tuning for computer chess: Linear discriminant methods. *ICCA Journal*, Vol. 20, No. 4, pp. 224–242.

[2] Axelrod, R.M. (1984). *The Evolution of Cooperation.* BASIC Books, New York.

[3] Billings, D., Davidson, A., Schaeffer, J., and Szafron, S. (2000). The challenge of poker. *Artificial Intelligence*, Vol. 134, No. 1–2, pp. 201–240.

[4] Brockington, M. and Darrah, M. (2002). How *Not* to implement a basic scripting language. *AI Game Programming Wisdom* (ed. S. Rabin), pp. 548–554, Charles River Media, Inc., Hingham, MA.

[5] Carmel, D. and Markovitch, S. (1993). Learning models of opponent's strategies in game playing. *Proceedings AAAI Fall Symposion on Games: Planning and Learning*, pp. 140–147, Raleigh, NC.

[6] Carmel, D. and Markovitch, S. (1998). Pruning algorithms for multi-model adversary search. *Artificial Intelligence*, Vol. 99, No. 2, pp. 325–355.

[7] Carmel, D. and Markovitch, S. (1999). Exploration strategies for model-based learning. *Autonomous Agents and Multi-Agent Systems*, Vol. 2, No. 2, pp. 141–272.

[8] Davison, B.D. and Hirsh, H. (1998). Predicting sequences of user actions. *Predicting the Future: AI Approaches to Time-Series Problems*, pp. 5–12, AAAI Press, Madison, WI. Proceedings of AAAI-98/ICML-98 Workshop, published as Technical Report WS-98-07.

[9] Domingos, P. and Pazzani, M. (1997). On the optimality of the simple bayesian classifier under zero-one loss. *Machine Learning*, Vol. 29, pp. 103–130.

[10] Donkers, H.H.L.M., Uiterwijk, J.W.H.M., and Herik, H.J. van den (2001). Probabilistic opponent-model search. *Information Sciences*, Vol. 135, No. 3–4, pp. 123–149.

[11] Donkers, H.H.L.M., Uiterwijk, J.W.H.M., and Herik, H.J. van den (2003). Admissibility in opponent-model search. *Information Sciences*, Vol. 154, No. 3-4, pp. 119–140.

[12] Egnor, D. (2000). Iocaine powder. *ICGA Journal*, Vol. 23, No. 1, pp. 33–35.

[13] Fairclough, C., Fagan, M., MacNamee, B., and Cunningham, P. (2001). Research directions for AI in computer games. *12th Irish Conference on Artificial Intelligence & Cognitive Science (AICS 2001)* (ed. D. O'Donoghue), pp. 333–344.

[14] Fudenberg, D. and Levine, D.K. (1998). *The Theory of Learning in Games.* MIT Press, Cambridge, MA.

[15] Fürnkranz, J. (1996). Machine learning in computer chess: The next generation. *ICCA Journal*, Vol. 19, No. 3, pp. 147–161.

[16] Fyfe, C. (2004). Independent component analysis against camouflage. *Computer Games: Artificial Intelligence, Design and Education (CGAIDE 2004)* (eds. Q. Mehdi, N.E. Gough, S. Natkin, and D. Al-Dabass), pp. 259–262, University of Wolverhampton, Wolverhampton, UK.

[17] Graepel, T., Herbrich, R., and Gold, J. (2004). Learning to fight. *Computer Games: Artificial Intelligence, Design and Education (CGAIDE 2004)* (eds. Q. Mehdi, N.E. Gough, S. Natkin, and

D. Al-Dabass), pp. 193–200, University of Wolverhampton, Wolverhampton, UK.

[18] Iida, H., Uiterwijk, J.W.H.M., and Herik, H.J. van den (1993a). Opponent-model search. Technical Report CS 93-03, Universiteit Maastricht, Maastricht, The Netherlands.

[19] Iida, H., Uiterwijk, J.W.H.M., Herik, H.J. van den, and Herschberg, I.S. (1993b). Potential applications of opponent-model search. Part 1: the domain of applicability. *ICCA Journal*, Vol. 16, No. 4, pp. 201–208.

[20] Iida, H., Handa, K-i, and Uiterwijk, J. (1995). Tutoring strategies in game-tree search. *ICCA Journal*, Vol. 18, No. 4, pp. 191–204.

[21] Iida, H., Kotani, I., Uiterwijk, J.W.H.M., and Herik, H.J. van den (1997). Gains and risks of om search. *Advances in Computer Chess 8* (eds. H.J. van den Herik and J.W.H.M. Uiterwijk), pp. 153–165, Universiteit Maastricht, Maastricht, The Netherlands.

[22] Junghanns, A. (1998). Are there practical alternatives to alpha-beta? *ICCA Journal*, Vol. 21, No. 1, pp. 14–32.

[23] Kendall, G. (2005). Iterated prisoner's dilemma competition. http://www.prisoners-dilemma.com/.

[24] Laird, J.E. and Lent, M. van (2001). Human-level's AI killer application: Interactive computer games. *Artificial Intelligence Magazine*, Vol. 22, No. 2, pp. 15–26.

[25] Livingstone, D. and Charles, D. (2004). Intelligent interfaces for digital games. *Proceedings of the AAAI-04 Workshop on Challenges in Game Artificial Intelligence* (eds. D. Fu, S. Henke, and J. Orkin), pp. 6–10, AAAI Press, Menlo Park, CA.

[26] Madeira, C., Corruble, V., Ramalho, G., and Ratitch, B. (2004). Bootstrapping the learning process for the semi-automated design of challenging game ai. *Proceedings of the AAAI-04 Workshop on Challenges in Game Artificial Intelligence* (eds. D. Fu, S. Henke, and J. Orkin), pp. 72–76, AAAI Press, Menlo Park, CA.

[27] Manslow, J. (2002). Learning and adaptation. *AI Game Programming Wisdom* (ed. S. Rabin), pp. 557–566, Charles River Media, Inc., Hingham, MA.

[28] Markovitch, S. (2003). Learning and exploiting relative weakness of opponent agents. NWO-SIKS Workshop on Opponent Models in Games, Maastricht, the Netherlands.

[29] Nareyek, A. (2002). Intelligent agents for computer games. *Computers and Games, Second International Conference, CG 2000* (eds. T.A. Marsland and I. Frank), Vol. 2063 of *Lecture Notes in Computer Science*, pp. 414–422, Springer-Verlag, Heidelberg, Germany.

[30] Pazzani, M. and Billsus, D. (1997). Learning and revising user profiles: The identification of interesting web sites. *Machine Learning*, Vol. 27, pp. 313–331.

[31] Reibman, A.L. and Ballard, B.W. (1983). Nonminimax search strategies for use against fallible opponents. *AAAI'83*, pp. 338–342, Morgan Kaufmann Publ., San Mateo, CA.

[32] Russell, S. and Norvig, P. (2003). *Artificial Intelligence: A Modern Approach*. Prentice Hall, Pearson Education, Upper Saddle River, NJ, second edition edition.

[33] Schaeffer, J. (2001). A gamut of games. *Artificial Intelligence Magazine*, Vol. 22, No. 3, pp. 29–46.

[34] Scott, B. (2002). The illusion of intelligence. *AI Game Programming Wisdom* (ed. S. Rabin), pp. 16–20, Charles River Media, Inc., Hingham, MA.

[35] Slagle, J.R. and Dixon, J.K. (1970). Experiments with the M & N tree-searching program. *Communications of the ACM*, Vol. 13, No. 3, pp. 147–154.

[36] Spronck, P.H.M., Sprinkhuizen-Kuyper, I.G., and Postma, E.O. (2004a). Difficulty scaling of game AI. *GAME-ON 2004 5th International Conference on Intelligent Games and Simulation* (eds. A. El Rhalibi and D. Van Welden), pp. 33–37, EUROSIS, Ghent, Belgium.

[37] Spronck, P.H.M., Sprinkhuizen-Kuyper, I.G., and Postma, E.O. (2004b). Online adaptation of game opponent AI with dynamic scripting. *International Journal of Intelligent Games and Simulation*, Vol. 3, No. 1, pp. 45–53.

[38] Sutton, R.S. and Barto, A.G. (1998). *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA.

[39] Tomlinson, S.L. (2003). Working at thinking about playing or a year in the life of a games AI programmer. *Proceedings of the 4th International Conference on Intelligent Games and Simulation (GAME-ON 2003)* (eds. Q. Mehdi, N. Gough, and S. Natkin), pp. 5–12, EUROSIS, Ghent, Belgium.

[40] Tozour, P. (2002). The perils of AI scripting. *AI Game Programming Wisdom* (ed. S. Rabin), pp. 541–547, Charles River Media, Inc., Hingham, MA.

[41] von Neumann, J. and Morgenstern, O. (1944). *Theory of Games and Economic Behavior*. Princeton University Press, Princeton, NJ.