Dynamic Difficulty for Checkers and Chinese chess

Laurențiu Ilici, Jiaojian Wang, Olana Missura and Thomas Gärtner

Abstract—We investigate the practical effectiveness of a theoretically sound algorithm for dynamic difficulty adjustment in computer games: Firstly, we show how the partially ordered set master (POSM) algorithm [11] can be incorporated in board games, taking checkers and Chinese chess as examples. Secondly, we describe an empirical study of (i) POSM on checkers against synthetic opponents of varying strength, (ii) POSM on Chinese chess against synthetic opponents of varying strength, and (iii) POSM on Chinese chess against human opponents of varying strength. Our results indicate that POSM can indeed serve as a flexible and effective subroutine for dynamical difficulty adjustment in computer games.

I. INTRODUCTION

Motivated by our enthusiasm and interest in well designed and fun games, we research ways of enhancing the player’s experience. In this paper we pursue making games more dynamic by adapting their difficulty in such a way that a player is challenged, but not overwhelmed [7]. We approach the problem from a machine learning perspective, with the intent of learning which difficulty setting is “right” for the player in a real time manner. An algorithm capable of performing this task is the partially ordered set master (POSM) [11]. This paper contains new results regarding its performance when evaluated on checkers and Chinese chess. POSM is used as a subroutine that adjusts an agent’s playing strength such that opponents of various skill levels are challenged, while still maintaining a fair chance of winning. The experimental results confirm that POSM performs this task well.

Most video games present to the players a range of difficulty settings before playing commences. Usually, due to the lack of the precise specifications about what each setting means, players have to guess which setting to pick according to their prior gaming experience. A wrong decision leads to either boring the player by making the game too easy or frustrating her by making it too hard. In an attempt to address this issue, the idea of dynamic difficulty adjustment was born, i.e., the game adjusts the difficulty setting in real time according to the player’s abilities or reactions.

Both game developer and researcher communities alike have tackled the problem of dynamic difficulty adjustment from different angles and with varying amounts of success (see Section II for several examples). In order to provide a framework for future research, Missura and Gärtner [11] presented a formalization of the dynamic difficulty adjustment problem, as well as the POSM algorithm. This work is continuing the investigations on its performance.

L. Ilici, J. Wang, O. Missura and T. Gärtner are with the University of Bonn, Bonn, Germany (email: firstname.lastname@uni-bonn.de); T. Gärtner is also with the Fraunhofer Institute for Intelligent Analysis and Information Systems, Sankt Augustin, Germany

The algorithm learns which setting is “right” for the current player based on observations made during each round of the game. These can be obtained by a variety of different methods, e.g., by monitoring the player’s facial expression with cameras or using heuristics to measure in game balance. Finding the right way of making the observations, such that they convey meaningful information, is at the moment a combination of using arcane knowledge about the game and empirical testing. However, as the experiments will show, the algorithm has the desired performance once a suitable observation method is found.

POSM has several distinct advantages over other approaches to dynamic difficulty. It is, first of all, applicable to any game genre, assuming the developer constructs a set of difficulty settings that form a “more difficult than” relation. This assumption is realistic, since games have a number of predefined settings. Secondly, in contrast to other approaches, no background knowledge about the behavior of the players is needed. Thirdly, there are theoretical guarantees on the maximum number of mistakes POSM makes before learning the appropriate setting.

We present the theoretical background and the algorithm along with an intuitive example in Section III. The purpose of the example is to illustrate both how the mechanics behind POSM work and what difficulties can be encountered.

For testing purposes, we chose two board games: checkers and Chinese chess. An overview of the implementation details of the two games, as well as their respective balance observation methods is provided in Section IV.

The experiments described in Section V represent the first evaluation of the algorithm in a real games. Last, but not least, Section VI provides a brief outlook on future work.

II. RELATED WORK

Commercial games have been criticized for having their content predefined in advance, e.g., monster spawn locations, map configurations or non-player characters behavior. This gives players a chance to exploit weaknesses in the game environment and often leads to the problem of games becoming too easy. In an attempt to address the issue, both the developer and research communities are working on methods of adding dynamic content in games, e.g., altering non-player characters’ performance or generating levels according to the player’s skills [14]. A good survey over the academic research directions in this area was made by Lopes and Bidarra [8].

In this paper we focus on online dynamic difficulty adjustment. The term online, in this context, means that the game adapts the difficulty setting during the course of one game. As of today, there are a number of different approaches to
this problem which will be briefly described below. A number of offline approaches exist as well, e.g., Charles and Black [1] and Togelius et al. [15], but they are out of our scope.

Probably one of the oldest methods used by commercial developers for adaptive AI is the “rubber band”. This method has been used especially in racing games, e.g., Mario Kart game series by Nintendo[12]. It provides both computer and human players with a realistic chance at winning no matter how far behind they are, by adjusting the speed of the opponents. Unfortunately, this kind of adaptation becomes obvious to a human player, which may results in loss of interest for playing competitively and ultimately in loss of interest for the game itself.

A more recent and sophisticated method of dynamic difficulty adjustment is used in the Left 4 Dead game developed by Valve. They have implemented a system referred to as AI Director1. The main concept behind this approach is that instead of having fixed spawn locations for in game items and monsters, there is a management system that decides their distribution according to the players current situation. This results in a different playing experience even when completing the same level over and over again. To the best of our knowledge, the only criticism towards this approach is that once the player gains experience with the game, the system becomes readable. This means she can alter her playing style in order to indirectly control the level design, which defeats the purpose of the system, i.e., providing unpredictable challenges to the player.

Danzi et al. [3] proposed a reinforcement learning approach. The authors present a modified version of the Q-learning algorithm: Instead of maximizing the expected reward the agent tries to keep it close to zero. The reasoning behind is that a reward of zero means the two players are balanced. The algorithm: Instead of maximizing the expected reward the algorithm can be easily ported between games and game genres, and it has proven theoretical guarantees on the number of mistakes it can make. POSM performed well in the synthetic experiments conducted in the original paper, however, it lacked evaluation in a real game environment. We present novel results regarding the capabilities of the algorithm in this paper.

Another machine learning approach was proposed by Missura and Gartner [11]. They provided both a formalization of the dynamic difficulty adjustment problem and a new algorithm (POSM) for predicting the right difficulty. The algorithm learns from observations made during gameplay and predicts the appropriate setting. It exhibits several distinct advantages over the approaches presented above: (i) POSM requires no offline training, (ii) the algorithm can be easily ported between games and game genres, and (iii) it has proven theoretical guarantees on the number of mistakes it can make. POSM performed well in the synthetic experiments conducted in the original paper, however, it lacked evaluation in a real game environment. We present novel results regarding the capabilities of the algorithm in this paper.

III. DYNAMIC DIFFICULTY ADJUSTMENT

This section both summarizes the theoretical background and gives the exponential update algorithm for predicting the “right” difficulty setting. We reiterate through a very small part of the content of the original publication [11] in order to make this paper self contained.

A. Theoretical Background

We define the problem of dynamic difficulty adjustment as a game played between a “master” and a “player” on a finite partially ordered set of difficulty settings that models a “more difficult than”-relation. The game is played in turns consisting of three stages:

1) the master predicts a difficulty setting,
2) the player plays the original game in this setting for a fixed amount of time, e.g., one ply in the case of turn based games, and
3) the master receives a feedback on how well this difficulty setting fits the player.

The feedback received after each turn is either:

- “too easy” if the master picked a difficulty setting that wasn’t hard enough;
- “right” if the master picked the appropriate setting; or
- “too hard” if the master picked a setting that exceeds the players skill level.

1 More information regarding Mario Kart and “rubber banding” can be found at: http://en.wikipedia.org/wiki/Mario_Kart
2 Half Life, Valve, Left 4 Dead are trademarks and/or registered trademarks of Valve Corporation. Mario Kart is a trademark of Nintendo. All other trademarks are property of their respective owners.
3 More information regarding the AI Director used in the Left 4 Dead game can be found at: http://left4dead.wikia.com/wiki/The_Director
This feedback is the only piece of information the master receives about the player. The whole process repeats until the game ends. As the master’s objective is to provide the player with the right difficulty setting as often as possible, its performance is measured by the amount of mistakes made. We consider the master has made a mistake whenever it chose an inappropriate difficulty setting.

A partial order on the set of difficulty settings is both natural and essential. Without it, POSM has no way of knowing which setting is more difficult than another setting. We assume this set to be finite, as in real games you would also expect only several distinct difficulty settings.

Notice, that no assumptions are made about the player. As players are varied in skill level and tend to improve their skill by playing, making assumptions about them would reduce the value of the results.

In order to better illustrate the setting described above and how to model it, we give a simple example. Imagine a game where the player gets points for clicking falling balloons on the screen. The game has ten distinct natural numbers for the falling speed of the balloons. These speed values form the difficulty settings and provide a total order. Intuitively, as the speed increases, it becomes more difficult for the player to click on all the balloons. Assume that, by surveying a number of different players, the game designers reach the conclusion that a player should be able to click eighty to ninety percent of all balloons, regardless of her skill level, to consider the game difficulty appropriate.

The master takes the role of choosing the right difficulty setting. It receives feedback about the player’s performance after each round, of, say, fifteen seconds, and uses this information to adjust its difficulty prediction. This means that whenever the percentage of balloons clicked, in a fifteen second period, stays within the bounds, the master observes “right”, for any lower or higher percentage it observes “too hard” or “too easy” respectively.

With the above example in mind, the problems that the master faces become apparent. Firstly, whenever it makes a wrong decision it has to wait for a feedback before being able to reassess the situation. Secondly, if a mistake was made, the observation only states that the difficulty setting was too easy or too hard, but it does not give any precise indication about what the right setting should have been. Moreover, as the game progresses, the player will probably get more skillful and a setting that was labeled for example too hard at first, will become right.

### B. Algorithm

We will now introduce the partially ordered set master (POSM) algorithm, which was originally published by Missura and Gärtner [11]. This algorithm assumes the role of the master in the problem formalization described above. The goal of the algorithm is to predict the right difficulty setting from a finite partially ordered set \((K, \succeq)\) of possible settings. The order is such that \(\forall i, j \in K\) we write \(i \succeq j\) if \(i\) is more difficult than \(j\). The possible responses \(o_t\) that the algorithm observes are:

- +1 if the difficulty setting was “too easy”;
- 0 if it was “right”, or
- -1 if it was “too hard”.

POSM maintains the belief \(w_t\) about the “right”-ness of each setting and updates it when the observed response was either too easy or too hard.

#### Algorithm 1 Partially-ordered-Set Master (POSM)

<table>
<thead>
<tr>
<th>Require:</th>
<th>(\beta \in (0, 1)), (K) difficulty settings, a partial order (\succeq) on (K), a sequence of observations (o_1, o_2, \ldots)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td>(\forall k \in K: \text{let } w_1(k) = 1)</td>
</tr>
<tr>
<td>2:</td>
<td>for (t = 1, 2, \ldots) do</td>
</tr>
<tr>
<td>3:</td>
<td>(\forall k \in K: \text{let } A_t(k) = \sum_{x \in K, x \succeq k} w_t(x))</td>
</tr>
<tr>
<td>4:</td>
<td>(\forall k \in K: \text{let } B_t(k) = \sum_{x \in K, x \nless k} w_t(x))</td>
</tr>
<tr>
<td>5:</td>
<td>Predict (k_t = \text{argmax}_{x \in K} \min{B_t(x), A_t(k)})</td>
</tr>
<tr>
<td>6:</td>
<td>Observe (o_t \in {-1, 0, 1})</td>
</tr>
<tr>
<td>7:</td>
<td>if (o_t = +1) then</td>
</tr>
<tr>
<td>8:</td>
<td>(\forall k \in K: \text{let } w_{t+1}(k) = \begin{cases} \beta w_t(k), &amp; k \leq k_t \ w_t(k), &amp; \text{otherwise} \end{cases})</td>
</tr>
<tr>
<td>9:</td>
<td>end if</td>
</tr>
<tr>
<td>10:</td>
<td>if (o_t = -1) then</td>
</tr>
<tr>
<td>11:</td>
<td>(\forall k \in K: \text{let } w_{t+1}(k) = \begin{cases} \beta w_t(k), &amp; k \geq k_t \ w_t(k), &amp; \text{otherwise} \end{cases})</td>
</tr>
<tr>
<td>12:</td>
<td>end if</td>
</tr>
<tr>
<td>13:</td>
<td>end for</td>
</tr>
</tbody>
</table>

Vector \(A_t\) in Line 3 collects the belief about all difficulty settings that are known to be more difficult than the current setting at the current time step, while \(B_t\) in Line 4 does the same for easier settings. If, according to the observation, POSM made a mistake it will update the belief for either the too easy or the too hard settings, in lines 8 or 11 respectively. We provide a sample run through of the algorithm below.

For simplicity, we will restrict the example to the balloon example given in Section III-A, which uses a total order on the set of difficulties rather than a partial order. Recall that we defined the difficulty settings as having ten distinct values and they are ordered by their ascending numerical values. The algorithm starts by initializing the beliefs \(w_1(k)\) to 1 for all difficulty settings. Intuitively, this means that the master does not have any information about the right setting and therefore, maintains an equal amount of belief about the “right”-ness of each setting. However, it still needs to predict a setting for the player. In order to be able to update the belief of as many settings as possible in case it makes a mistake, it first calculates \(A_t = (1, 2, 3, \ldots, 10)\) and \(B_t = (10, 9, 8, \ldots, 1)\). The master then predicts the difficulty setting \(k\) according to the formula in Line 5. Notice that we have a tie in this example for \(k = 5\) and \(k = 6\). How to break ties is left for the developer to decide (we randomly pick one setting). Let the algorithm predict \(k = 6\) for this example. The master then pauses until it receives an observation.
After observing one round of play, if the value of the observation is not “right”, the master will update the belief of the current setting as well as either the beliefs of the settings known to be more difficult than the current setting or the less difficult in line 9 or 12 respectively. Continuing the example this would mean that, given an observation of, say, −1 (too hard), it would update the \( w_1(k) \) for \( k \in [6, 10] \) (we denote with \([a, b]\) the set of natural numbers in the interval \([a, b]\)) by multiplying them with the learning rate \( \beta \). As \( \beta \in (0, 1) \) it will lower the belief that the current setting is right setting as well as the beliefs about all harder settings are right. The process then repeats, with the newly calculated \( w_2 \), until the game ends.

The example above could be extended to a partial order, if we extend the game described in Section III-A to also include falling bombs mixed among the balloons and a setting for the frequency of generating bombs. Let clicking a bomb cause the player the inability to click anything else on the screen for, say, two seconds. Now our difficulty settings become a tuple of the form \((\text{balloonSpeed, bombFrequency})\). These difficulty settings no longer provide a total order because, while we can safely assume a tuple with both higher balloon speed and frequency is harder than one with lower values for both settings, we cannot assume the same for one that has one value higher and the other lower (example, setting \((5, 5)\) is intuitively harder than \((4, 4)\), however, we cannot compare \((5, 5)\) with \((4, 6)\) or \((6, 4)\) as some players will cope better with speed rather than bombs and vice versa). The algorithm works in this setting as well by just using the information on the difficulty settings that can be directly compared.

### IV. Tested Games

In this section we introduce the games we have chosen for testing: checkers and Chinese chess. We will also provide a short overview about the related work for these games. Last but not least, we will provide the reader with some basic knowledge about the static evaluation functions that were used for scoring the various board configurations.

The main reasons for choosing these two games are the following: Both games

- have been thoroughly studied by other researchers, which provides us with a framework to build upon,
- are relatively easy to learn and difficult to master by humans and computers alike, and
- are zero sum games, meaning we can use the Minimax [9] algorithm. Using Minimax provides us with an obvious choice of a totally ordered set of difficulty settings, i.e., the search depth up to which we explore the game tree.

#### A. Checkers

Checkers or English Draughts is a well known board game, in terms of popularity among human players and in terms of academic research. While it is out of the scope of this paper to present the game itself, there are numerous tutorials available online\(^4\). In this work we follow the American rules.

\(^4\) Tutorial for checkers: [www.jimloy.com/checkers/rules2.htm](http://www.jimloy.com/checkers/rules2.htm)

From a computer science point of view the game has been studied for almost 60 years. Most publications deal with creating a computer opponent that plays at a very high level. This research direction has been exhausted as the game was solved by Schaeffer et al. [13]: assuming perfect play from both sides the game always ends in a draw. To the best of our knowledge, the exact algorithm developed by this research is not fully disclosed.

Although creating the best player is irrelevant for the intent of this paper, we still need a static evaluation function of our own. By consulting some of the knowledge published by J. Schaeffer [12] about the important features of a checkers board configuration, we have built two static evaluation functions. These will be used by the Minimax algorithm to assess utility of the possible moves in the game tree.

First we developed a rather simple evaluation function, henceforth denoted by \textsc{Material}. It counts the number of pieces on the board for one player and subtracts the number of pieces of the other player. For each normal piece it awards 1 point and for each king 1.5 points. It also takes into account winning or losing positions, awarding \(\infty\) for a win and \(-\infty\) for a loss. Although this heuristic is very simple, it would be sufficiently powerful to play a perfect game if we could traverse the entire game tree. As the search depth will be limited, its performance will be restricted and can be rather poor when pinned against human opponents.

The second evaluation function is more complex (henceforth denoted by \textsc{Positional}). It has the same basic construction as \textsc{Material} but takes into account nine more features. For example it’s important which player controls the middle of the game board and if there are pieces on the edges because they cannot be jumped. The weights for the nine new features have been evolved via a genetic algorithm approach. We will not discuss the details of this procedure due to space restrictions, however, in the experiments we will show that \textsc{Positional} outperforms \textsc{Material}.

By combining these two heuristics with the Minimax algorithm applied at different search depths, we obtain a large enough spectrum of differently skilled computer players. Furthermore, by choosing one of the two evaluation functions for \textsc{Posm’s} move picking policy and the other one for the opponent’s policy, we have the added benefit of a better approximation of a real world environment; where each player has its own strategy.

One important issue with the game of checkers is that the official rule book does not contain a robust means of declaring a game as ending in a draw. In a championship game only an independent referee can decide this. To circumvent this problem we declare a draw if after one hundred ply no one can be declared a winner.

As mentioned in Section III-A, \textsc{Posm} relies on observations whether the difficulty setting was appropriately chosen. For the game of checkers this observation will be provided by the best move possible according to \textsc{Positional}(8) (we will henceforth denote the computer players by combining the name of the evaluation function and the search depth at which
the minimax search is applied). If the score of this move surpasses a certain threshold, POSM will receive -1 or 1 according to the sign of the score, otherwise it will receive 0. This means that POSM will always assess the game under the assumption that it was playing against POSITIONAL(dif) and, thus, be biased to overestimate a player like MATERIAL(2) or underestimate a human player if she knows a better way of picking her moves. However, trying to avoid this assumption seems infeasible, as it is unclear how to perfectly learn the opponent’s strategy within a few game moves.

POSM picks its move according to POSITIONAL(dif) where dif is the current predicted difficulty setting. This means POSM cannot adapt to any player that has a skill level which surpasses the POSITIONAL(8) evaluation function, due to the limitations we imposed on the range of difficulty settings we chose for the game, particularly, search depth settings in [2, 8]. Choosing a broader set of settings would not induce any problem for POSM, but would have an increasingly high effect on the running time of the experiments, due to the complexity of the Minimax algorithm.

B. Chinese chess

Chinese chess or Xiangqi is a two-player Chinese board game, which is similar to Western chess. It is one of the most popular board games in Chinese communities and has a long history. There are many hypotheses on its origin, however, its modern form was first introduced in the Song dynasty (960 – 1279). As is the case with the game of checkers, there are online rule books.

The first scientific paper on computer Chinese chess was published by T. Y. Zang [17]. The earliest human vs computer competition was the annual ACER cup, which was held in Taiwan from 1985 to 1990.

Based on the work published by Luo et al. [16] and H. Chen [2] we have again constructed a static evaluation function. This function takes into account three major features and combines them in as an linear weighted sum to compute the final score of a given board configuration.

The first major feature and the most important one, as is the case with the game of checkers, is material. We calculate this feature by adding up the values of the pieces each player possesses on the board and subtracting the two sums from each other. The value for each piece is determined by its type and position. A piece-square table is used to store the weights for each of these (piece type-position) tuples. The values contained in the table are based on both the published training results of a genetic algorithm [16] and a Xiangqi program called ElephantEye [2].

The second major feature is called tactical formation. The Cannon piece can capture an enemy piece if and only if there exists exactly one piece in between. A common strategy is to place one of our own Cannons in a position such that it attacks the opponent’s king directly. Although it would not be able to capture it, this prevents an opponent from placing any piece in between, thus, impairing the opponent’s defense.

Last but not least we look at mobility. On a basic level it measures the number of legal moves a player has in a given position. In addition it takes into account the fact that the Horse and Chariot pieces have their strength and mobility highly correlated. Therefore, it adds a penalty to these pieces according to how restricted their mobility is by diminishing their material value.

The game of Xiangqi presents us with the same problem as the game of checkers, i.e., a draw is usually declared by a referee. Furthermore, we have to create a system that avoids repetitions, otherwise the computer players will get stuck in a cyclic array of positions. To circumvent these issues we have developed a simple mechanism that deals with the most basic kinds of repetitions and limited the games to two hundred ply per player before calling the game a draw. In order to illustrate the effects of repetitions we can make a comparison between Western chess, where a threefold repetition means the game ends in a draw, and Xiangqi, where it leads to a loss for the player causing the repetition.

To demonstrate the flexibility of POSM we use a different observation method for Xiangqi. It receives information about the game state by applying CHESSAI(dif) (we will henceforth denote our non-adaptive player with ChessAI) where dif is the current difficulty setting and observe how the move score returned by the Minimax algorithm varies from ply to ply. To this end, we subtract the score of the previous move from the score of the move we would be picking with the current difficulty setting. If the absolute value of this difference surpasses a threshold value, POSM receives a feedback of -1 or +1 according to the sign of the the difference, otherwise it receives 0.

V. Results

This section describes the experiments conducted on the two games. First we introduce a performance measure. Then we describe the experiments and provide their results.

While there are well established systems for measuring a player’s skill, e.g., the TrueSkill [5] or the Elo [4], they are more powerful than what is needed in this work. To this end, we introduce a simple measure called STRENGTH. To calculate it, we pin two players against each other for a fixed number n of games and observe their outcome:

- 1 if the first player wins (the total number of wins is denoted by wins);
- 0 if we have a draw (the total number of draws is denoted by draws);

We then compute the formula \((\text{wins} + \text{draws}/2)/n\), in order to combine all the observations in a single easy to understand number that takes values in \([0, 1]\). We call this number STRENGTH. The closer it gets to a value of 1 the more it indicates that the first player is overpowering the second one. On the other hand, a value closer to 0 would indicate that the second player is more powerful. Equilibrium is achieved when the STRENGTH has a value of 0.5. POSM would ideally perform equally against each of its opponents. As the results

\footnote{Tutorial for Chinese chess: www.chessvariants.org/xiangqi.html}
will show it is able to maintain an almost constant Strength close to 0.5 across all opponents.

A. Checkers

![Figure 1](image1.png)

**Figure 1.** Positional(8,6,4,2) vs Material(i) Strength. Each point is calculated after running an episode of 100 games. This figure shows how Positional scales against Material.

![Figure 2](image2.png)

**Figure 2.** Material(8,6,4,2) vs Material(i) Strength. Each point is calculated after running an episode of 100 games. This figure shows how search depth affects playing strength of the computer players.

In the first series of experiments we needed to assess how the various combinations of evaluation functions and search depths fair against each other. To this end, we ran episodes of one hundred games for every possible combination of evaluation function and search depth. A number of these are both displayed by the pieces and explained below.

Figure 1 displays a few representative results on how Positional scales when playing against Material([2, 8]). The Positional(8) vs Material(i) and Positional(2) vs Material(i) are very important, as they form an upper and lower bound on the playing strength of POSM. This is due to the fact that, as stated in Subsection IV-A, POSM uses the Positional evaluation function in order to both assess the game balance and return its move.

Figure 2 focuses on displaying how the search depth alone affects the playing strength. One of the players is fixed to being one of Material(2,4,6,8) while the opponent is using Material(i) with i in [2, 8]. Notice that as the search depths of players are closer together the Strength approaches 0.5. These results are not only what we would intuitively expect, but also serve as a sanity check for this measuring strategy.

Figure 3 keeps the Positional(8) vs Material(i) and Positional(2) vs Material(i) points, just to put things into perspective. We can observe that POSM manages to keep a Strength within roughly the interval [0.4, 0.5] across all the six opponents which are playing using the Material evaluation function. This means it correctly predicts the strength of the opponent in most of the games, in spite of having a different and more advanced playing strategy than its opponents. Furthermore, in these experiments POSM plays against the same opponent a batch of one hundred games, but we reset the weight vector \( w_t \) after each match. This makes it harder for the algorithm as no information whatsoever is passed between games, thus, simulating playing against a new player in every game.

<table>
<thead>
<tr>
<th>&amp; Positional(8)</th>
<th>POSM</th>
</tr>
</thead>
<tbody>
<tr>
<td>win</td>
<td>draw</td>
</tr>
<tr>
<td>Material(2)</td>
<td>97</td>
</tr>
<tr>
<td>Material(3)</td>
<td>86</td>
</tr>
<tr>
<td>Material(4)</td>
<td>79</td>
</tr>
<tr>
<td>Material(5)</td>
<td>76</td>
</tr>
<tr>
<td>Material(6)</td>
<td>54</td>
</tr>
<tr>
<td>Material(7)</td>
<td>52</td>
</tr>
<tr>
<td>Material(8)</td>
<td>34</td>
</tr>
</tbody>
</table>

Table I lists the actual number of wins, draws and losses for Positional(8) and POSM when playing against Material([2, 8]). As we can see both the win and the draw rates go up when POSM is adjusting the difficulty when compared to the rates Positional(8) is achieving. While we would like to see a more even distribution over the wins and losses from POSM, the increase in the number of draws shows that the
algorithm is indeed adapting well to its opponent.

All of the above results were achieved by using the following parameters: 0.5 for POSM’s learning rate and \( \{-2.5, 2.5\} \) for the threshold value of the observation method. While the learning rate was kept constant, we have tried several other values ( \( \{-1, 1\}, \{-0.5, 0.5\}, \{-1.5, 1.5\}\) ) for the threshold but these, \( \{-2.5, 2.5\} \), provided the best results. We did not try an iterative search for the perfect threshold in order to avoid overfitting. However, these values will only work with the evaluation function used in our experiments. A different evaluation function and/or observation method will need a different set of parameters.

B. Chinese chess

The experiments conducted on this game are complementary to the ones conducted on the checkers game. They mainly focus on the ability of POSM to pick the opponent’s search depth, as well as on how well it fares against humans.

We will first describe the computer player results. To see how well POSM can predict the opponent’s search depth we ran episodes of 1000 games of POSM vs CHESSAI(\( i \)) with \( i = [2, 6] \). Figure 4 illustrates the results. On the horizontal axis we have a counter for the moves within the games, while on the vertical axis we have the average predicted difficulty for the opponent. As stated in Section III-A the algorithm always picks the medium difficulty setting at start. For this batch of experiments this is 4 because the range of possible settings is \([1, 8]\). As the games progress and POSM receives more feedback about the state of the games, we can see how the predicted difficulty setting reaches the opponent setting.

![Fig. 4. POSM vs CHESSAI(2-6) Average predicted difficulty over episodes of 100 games against each player.](image)

Although testing against computer players gave us a good indication whether or not the algorithm is working properly, we also conducted experiments to see how the algorithm performs against humans.

A total of twelve testers volunteered for the study. We asked them to play at least ten games against POSM. The outcome of each game was recorded. We have also conducted a survey where the players were asked “How difficult did you find the game?” with three possible answers “too easy”, “too hard” and “equal”. This question was asked only once, when they submitted their final results.

All of the testers were rated beforehand by an outside ranking system\(^6\), thus dividing them into several categories according to their skill level. The possible categories were Level 8 through 1 players, Level 8 being the lowest ranking skill, followed by Master 3 through 1, Master 1 being the highest possible ranking. The results of the games played against the different categories of testers can be seen in Figure 5. The results of the survey as well as the spread of the players are depicted in Figure 6. A scatter plot showing the STRENGTH is shown in Figure 7.

The limitations of this approach can be seen in Figure 5, where we can see that Level 8 players haven’t been able to obtain even a single victory. Both tests against human and computer players alike have shown that POSM cannot adapt to an opponent that has a random move picking policy. This fact may however be attributed to a shortcoming of having a deterministic algorithm. Imagine that a completely random player may choose a very good move or even worse a sequence of very good moves by chance, followed by a sequence of very bad moves. Because of the sequence of good moves chosen by the opponent at the beginning, the algorithm will increase the difficulty setting and its belief about the opponent being skillful. Reversing this process takes a potentially long series of observations and ultimately leads to the defeat of the random player. Furthermore POSM cannot play worse than random, because even if it sets the difficulty setting to one (the minimum admissible value such that the Minimax algorithm works) the evaluation function will still be applied, thus returning a move which is on average better than a random one.

![Fig. 5. POSM vs Human testers. The number of games ending with wins/draws/loses for each category of player. Categories for which no volunteers were found are displayed on the X axis, in order to provide a broader overview of the skill levels.](image)

A short analysis of Figure 6 and Figure 7 shows a strong correlation between the results of the survey and the STRENGTH, i.e., if the STRENGTH is high the survey says the opponent was too easy and vice versa. The survey results are

\(^6\)Game platform used to rank the human players: www.qqgame.qq.com/
biased by human emotion and are more meaningful than the STRENGTH results, because they capture the in-game balance information as opposed to only the end result. After all, a game can be perfectly balanced for its whole timespan and then just at the end a few mistakes can lead to a loss for either of the players. This bias is illustrated by the Level 6 players which although lost a lot more games when playing against POSM than the other categories, still responded that they perceived their opponent as being equally strong.

The human experiments, although providing encouraging results, were not made on a sufficiently large player base to enable statistically significant statements to be made. Performing the experiments on a larger player base is a task for future work. Another interesting research direction that we will pursue is applying the algorithm to other game genres, e.g., real time strategy or racing games.

VI. Conclusions

We investigated the practical performance of a theoretically sound algorithm for dynamic difficulty adjustment, namely, the partially ordered set master POSM algorithm [11]. We have shown how it can be incorporated in board games, by taking checkers and Chinese chess as examples and providing the design details for our game balance observation methods.

The tests, consisting of games versus synthetic opponents of various strength, as well as versus human players, have shown that in almost all different settings POSM adjusts the difficulty properly. (The exceptional cases are, as expected, the opponents who are either too random, or too smart.)

The results show the answers to the question “How difficult did you find the game?”. Categories for which no volunteers were found are displayed on the X axis as well.

VI. Acknowledgments

Part of this work was supported by the German Science Foundation (DFG) under ’GA 1615/1-1’. The authors thank their co-workers and reviewers for their valuable comments and suggestions. Special thanks go to the testers which volunteered for our human experiments.

References