OLANA MISSURA

DYNAMIC DIFFICULTY ADJUSTMENT

DYNAMIC DIFFICULTY ADJUSTMENT

Dissertation

zur

Erlangung des Doktorgrades (Dr. rer. nat.)

der

Mathematisch-Naturwissenschaftlichen Fakultät

der

Rheinischen Friedrich-Wilhelms-Universität Bonn

vorgelegt von

Olana Missura

aus

Donetsk, USSR

Bonn, Januar 2015

Angefertigt mit Genehmigung der
Mathematisch-Naturwissenschaftlichen Fakultät der
Rheinischen Friedrich-Wilhelms-Universität Bonn

1. GUTACHTER: Prof. Dr. Thomas Gärtner
2. GUTACHTER: Prof. Dr. Stefan Wrobel

TAG DER PROMOTION: 02.10.2015
ERSCHEINUNGSJAHR: 2015

One day I will find the right words, and they will be simple.

— Jack Kerouac [47]

ABSTRACT

One of the challenges that a computer game developer faces when creating a new game is getting the difficulty "right". Providing a game with an ability to automatically scale the difficulty depending on the current player would make the games more engaging over longer time. In this work we aim at a dynamic difficulty adjustment algorithm that can be used as a black box: universal, nonintrusive, and with guarantees on its performance. While there are a few commercial games that boast about having such a system, as well as a few published results on this topic, to the best of our knowledge none of them satisfy all three of these properties.

On the way to our destination we first consider a game as an interaction between a player and her opponent. In this context, assuming their goals are mutually exclusive, difficulty adjustment consists of tuning the skill of the opponent to match the skill of the player. We propose a way to estimate the latter and adjust the former based on ranking the moves available to each player. Two sets of empirical experiments demonstrate the power, but also the limitations of this approach. Most importantly, the assumptions we make restrict the class of games it can be applied to.

Looking for universality, we drop the constraints on the types of games we consider. We rely on the power of supervised learning and use the data collected from game testers to learn models of difficulty adjustment, as well as a mapping from game traces to models. Given a short game trace, the corresponding model tells the game what difficulty adjustment should be used. Using a self-developed game, we show that the predicted adjustments match players' preferences.

The quality of the difficulty models depends on the quality of existing training data. The desire to dispense with the need for it leads us to the last approach. We propose a formalization of dynamic difficulty adjustment as a novel learning problem in the context of online learning and provide an algorithm to solve it, together with an upper bound on its performance. We show empirical results obtained in simulation and in two qualitatively different games with human participants. Due to its general nature, this algorithm can indeed be used as a black box for dynamic difficulty adjustment: It is applicable to any game with various difficulty states; it does not interfere with the player's experience; and it has a theoretical guarantee on how many mistakes it can possibly make.

ACKNOWLEDGMENTS

During the time of working on and writing my thesis I received support and help from many people. I would like to take this opportunity to thank them.

First and foremost, I want to express my gratitude to my advisor, Thomas Gärtner. During my years as a graduate student he was a source of relentless support, limitless ideas, and neverending patience. I am grateful to him for creating a motivating and supporting working environment, for securing the financial support, for listening to me and encouraging me during the difficult times, for allowing me to make my own mistakes but always providing help when I needed it. Most of the work presented in this dissertation is a collaborative effort with Thomas. He also dedicated a lot of time and effort to proofreading multiple drafts of this thesis and provided helpful and constructive comments, that led to a substantial improvement of the final text.

Most of the experimental results presented here were made possible by students who worked with me: Laurenţiu Ilici (Chapter 3), Jiao Jian Wang (Chapters 3 and 5), and Martin Mladenov (Chapter 4). They put a lot of effort into creating the corresponding experimental environments, finding the participants, and compiling the results. I would also like to thank our participants, both the ones I knew, i. e., my colleagues and students at the Fraunhofer IAIS and the University of Bonn, and the anonymous ones who participated over the Internet.

I wish to give special thanks to Martin Mladenov for his original ideas and substantial contribution to the research presented in Chapter 4. I am also very grateful to Tamas Horvath, Mario Boley, and Ulf Brefeld for constructive discussions on research processes, work environments, and motivational problems. Helpful comments and feedback were also received during all these years from all members of the CAML work group.

I feel that my work and its presentation was much improved thanks to the anonymous reviewers, who took the time to read the paper versions of this thesis' chapters and provided many helpful comments and suggestions. I also want to thank the various people who came to my posters and talks for their interest, comments, suggestions, and unexpected connections.

Most importantly, I would like to thank my husband, Marcell Missura. Without his steady support and encouragement I would not have been able to concentrate on my work and complete my thesis.

Finally, I am grateful for the financial support that allowed me to work on my dissertation. This support was provided by a collabora-

tion project on “Learning and Inference” funded by the Fraunhofer and Max-Planck Societies, as well as by the German Science Foundation (DFG) under the reference number ‘GA 1615/1-1’.

I wish to dedicate this work to my daughter Mia, who has brought the most joy in my life and has been a source of invaluable new experiences.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

1

ROAD MAP

In recent years there was a steady interest in researching how video games can adjust themselves to their players. In this work we aim at creating an algorithm for one aspect of such an adjustment, the game's difficulty. Note that regardless of the purpose of a game, be it entertainment, education, or training, the goal is generally to keep players' attention for as long as it is reasonable. What is a certain way to lose their attention? A straightforward answer is for a game to be boring, be it because of a trivial story, lack of excitement, repetitive too difficult or too easy challenges, etc.

A game and its player are two interacting entities. Both of them have a set of goals that they attempt to achieve through the interaction. For the player the goals may be for example defeating whatever challenges the game is presenting to her, exploring the game world, or exploiting the rules of the game, but globally a typical player plays to have fun. For a typical commercial game the main goal is to maximise its sales figures, which can be achieved via different means, but the most sure one is for the game to be simply fun to play. What constitutes the *fun* when playing a game?

There are three main components in the theories on why gaming is fun: reward, flow and iteration [71]. Reward derives from our intrinsic nature to reward ourselves for doing something. Video games play on this by providing immediate in-game rewards for completing in-game tasks. Flow is the player's ability to become almost a part of the game. Being in the flow means that the player becomes immersed in the game and loses the track of reality and the sense of self. For this to occur in a game the player must be actively involved, concentrated and unaware of realities of time and space boundaries. Iteration is the games ability to be different upon repetition.

There is a theory that our brains are physiologically driven by a desire to learn something new: new skills, new patterns, new ideas [96, 6]. We have an instinct to play because during our evolution as a species playing generally provided a safe way of learning new things that were potentially beneficial for our life. Daniel Cook [12] created a psychological model of a player as an entity that is driven to learn new skills that are high in perceived value. This drive works because we are rewarded for each new mastered skill or gained knowledge: The moment of mastery provides us with the feeling of joy. The games create additional rewards for their players such as new items available, new areas to explore. At the same time there are new challenges to overcome, new goals to achieve, and new skills to learn, which cre-

ates a loop of learning-mastery-reward and keeps the player involved and engaged.

An inherent feature of any challenge (and of the learning required to master it) is its difficulty. Here the difficulty is a subjective factor that stems from the interaction between the player and the challenge: Some people find controlling a simulation of a helicopter in a three-dimensional space as easy and natural as walking, but most would struggle with it for quite a while before they master it. This example also demonstrates that the perceived difficulty is not a static property: It changes with the time that the player spent learning a skill. In general the more time and effort we invest into learning something new, the better we get at it and the easier subjectively the tasks that exercise this skill get.

To complicate things further, not only the perceived difficulty depends on the current state of the player's skills and her learning process, the dependency is actually bidirectional: The ability to learn the skill and the speed of the learning process are also controlled by how difficult the player perceives the task. If the bar is set too high and the task appears too difficult, the player will end up frustrated and will give up on the process in favour of something more rewarding. Then again if the challenge turns out to be too easy (meaning that the player already possesses the skill necessary to deal with it) then there is no learning involved: Even though the player accomplishes the task and receives the in-game rewards, she is missing out on that internal reward, the feeling of joy that the moment of mastery provides. And without it there is no sense of accomplishment, which makes the game appear boring.

For these reasons the game that strives to be fun should provide the challenges for the player of the "right" difficulty: The one that stimulates the learning without pushing the players too far or not enough. Ideally, the difficulty of any particular instance of the game should be determined by who is playing it at this moment; the game should possess an ability to change the difficulty of its challenges on the fly, in an online fashion.

The traditional way in which games treat difficulty adjustment is to provide players with a way of controlling the difficulty level themselves. To this end, typical levels would be 'beginner', 'medium', and 'hard'. Such a strategy has many problems. On the one hand, if the number of levels is small, it may be easy to choose the right level but it is unlikely that the difficulty is then set in a very satisfying way. On the other hand, if the number of levels is large, it is more likely that a satisfying setting is available but finding it becomes more difficult. The necessity of going back and forth between the gameplay and the settings when the tasks become too difficult or too easy disrupts the flow component of the game. On yet another hand, for game developers, it is not an easy task to map a complex game world into one

variable. Constructing such "mapping" requires extensive testing, resulting in time and money costs. Consider also the fact that generally games require several different skills to play them. Providing the computer with an ability to adjust the game to all these skill levels automatically is more user-friendly than offering several settings for a user to set.

Gilleade et al. [29] and Sweetser and Wyeth [84] state that providing players with a personalised, adaptive experience can sustain their attention and keep their interest for longer, which agrees with the argument that games with a personalised dynamic difficulty adjustment system are more interesting. Motivated by this, we aim at creating a mechanism for developing games that on the fly provide challenges of the "right" difficulty, i. e., such that players are stimulated but not overburdened. To this purpose, we investigate how machine learning techniques can be employed to automatically adjust the difficulty of games. A general technique for this problem has natural applications in the huge markets of video games but can also be used to improve the learning rates when applied to serious games.

This work is about the design of a dynamic difficulty adjustment algorithm that is driven by the following requirements:

- Universality: The resulting algorithm should be applicable to as many games as possible, independently of their type, structure, or features;
- Non-intrusiveness: If desired, the resulting algorithm should work in a transparent way, in real-time, not requiring an interaction with players;
- Feasibility: There should be guarantees about the resulting algorithm's performance.

The main part of the thesis is organized into three chapters, each of these dealing with the same question but applied to different and progressively more difficult settings. Here in brief are some of the main contributions. A more detailed overview follows below.

Chapter 3 starts with a precisely defined world of board games and describes a dynamic difficulty adjustment heuristic. Specifically, we propose that in a setting with clearly defined 'moves' and a way to evaluate their influence on the game's outcome, the dynamic difficulty adjustment mechanism chooses the next move based on the quality of a player's moves so far. We present the experiments demonstrating the power of this heuristic and describe its limitations.

In Chapter 4 we use the power of supervised learning to construct a dynamic difficulty adjustment algorithm. To this purpose we require that players have an ability to adjust the difficulty of the game in real time, at least during the playtesting phases of game development. To provide the learning algorithm with the training input, we collect the

data: game traces that among other features contain the difficulty adjustments chosen by players. From this data the algorithm constructs a set of players' models and learns to classify game traces into the set of models. After the learning stage is finished, any new player is provided with a difficulty adjustment that is specified by his or her model.

Finally, Chapter 5 presents an algorithm that doesn't demand any particular prerequisites from a game, apart from two that we feel are essential for any plausible dynamic difficulty adjustment mechanism:

- The game should possess states of different difficulty.
- The game should provide information on how well the current state matches the player's skill and preferences.

The algorithm is built using online learning principles and satisfies all the criteria listed above, including the theoretical guarantees on its performance.

1.1 DYNAMIC DIFFICULTY ADJUSTMENT VIA RANKING OF AVAILABLE ACTIONS

A lot of games are based on interactions between a player and one or more in-game entities, e. g. computer opponents in a real-time strategy, bots in a first-person shooter, non-player characters in a role-playing game. The problem of automatic difficulty scaling can be viewed in the context of these interactions. The player is a person who is currently playing the game. The agent is an in-game entity, the player's opponent. It is natural to assume that at any given time the agent has a set of actions (strategies) available to it. The question of how to adjust the game difficulty automatically can be formulated as which action or strategy should the in-game agent choose as next.

Consider the situation where both player's and agent's sets of actions are similar, as it may be the case in a two-player board game (i. e. tic-tac-toe, connect four, or backgammon), but also in a "real" computer game such as, for example, a real-time strategy. In any given game state some actions are better than the others or, in other words, there is a naturally occurring ranking on the available actions. Knowing the ranking, the agent can evaluate the performance of the player and choose its own actions accordingly. This approach allows us to create an online adaptive agent, where "online" means that the agent adapts to its opponent during the course of a single game.

In Chapter 3 we investigate the ranking approach to developing online adaptive agents on examples of connect four and checkers. To evaluate the resulting agents we design a test environment consisting of two parts. The first part contains several preprogrammed algorithms with distinct skill levels. The second part provides an environment where human players can play against the developed agents

and the statistics necessary for evaluation are gathered. The empirical evaluation provides insights both into strengths and weaknesses of the ranking dynamic difficulty adjustment.

1.2 DYNAMIC DIFFICULTY ADJUSTMENT VIA MODELLING OF PLAYING TYPES

Next we consider the dynamic difficulty adjustment as a supervised learning problem. Luckily, there is a large selection of supervised learning methods available. To make use of them though, we need training data.

Game development process usually includes multiple testing stages, where a multitude of players is requested to play the game to provide data and feedback. This data is analysed to tweak the games' parameters in an attempt to provide a fair and fun challenge for as many players as possible. The question we investigate in Chapter 4 is how the data from these tests can be used for designing intelligent difficulty settings with the help of supervised learning.

We assume there is a phase of the game development in which the game is played by a multitude of players and the difficulty is manually adjusted by them. If the data collected during the testing includes the difficulty adjustments, we can induce a difficulty model and build it into the game. The actual players do not notice any of this and, ideally, are always challenged at the difficulty that is estimated to be just right for them.

In the first approach, we assume that there is a finite number of player types. (This assumption is supported by the fact that a lot of games offer a player a choice from several predefined difficulty settings, such as for example 'easy', 'medium', 'hard'.) We do not predefine this number, but rather use clustering on the training data to see which number fits the data the best.

Our approach to building a difficulty model consists of three steps: (i) cluster the recorded game traces, (ii) average the supervision over each cluster, and (iii) learn to predict the right cluster from a short period of gameplay. In order to validate this approach, we use a leave-one-player-out strategy on data collected from a simple game and compare our approach to less sophisticated, yet realistic, baselines. All approaches are chosen such that the players are not bothered. In particular, we compare the performance of dynamic difficulty versus constant difficulty as well as the performance of cluster prediction versus no-cluster.

To test our approach we implemented a simple game using the Microsoft XNA framework [1] and one of the tutorials from the XNA Creators Club community, namely "Beginner's Guide to 2D Games"

1 http://msdn.microsoft.com/en-us/xna/default.aspx

[2]. During each game all the information concerning the game state (e. g. the amount of hit points, the positions of the aliens, the buttons pressed, etc) is logged together with a timestamp.

Even though our experimental results confirm that dynamic adjustment and cluster prediction together outperform the alternatives significantly, there are a few open questions. The most important ones are:

- The length of a game trace in our evaluation is rather short (no more than 100 seconds). Is it possible that longer games would decrease the performance?
- The length of a prefix that the learning is based on is a parameter. It is clear that it must have an influence on the outcome. What is its optimal value in terms of balance between the adaptation speed and performance?

We propose another supervised dynamic difficulty adjustment method that neatly sidesteps both of these questions. Instead of dividing a game trace into a prefix and a suffix, we train an 'online' predictor: It predicts only the next (time step) difficulty adjustment based on a few last time steps. The experiments performed on the same data as above show that this predictor indeed can follow the self-inflicted difficulty adjustments quite closely.

1.3 DYNAMIC DIFFICULTY ADJUSTMENT VIA ONLINE LEARNING

In the last of the main chapters (Chapter 5) we drop any and all assumptions about the game and its players apart from the following two:

1. The game should possess states of different difficulty.
2. The game should provide information on how well the current state matches the player's preferences.

The first assumption is quite natural: if there are no distinct states of different difficulty, there cannot be a system for changing it, neither dynamically, nor statically. The second assumption should hold as well, since without input of any kind it is impossible to design a learning system.

In Chapter 5 we formalise dynamic difficulty adjustment as a metagame between a *master* and a *player* in which the *master* tries to predict the most appropriate difficulty setting, played on a partially ordered set modelling the 'more difficult than'-relation. As the *player* is typically a human with changing performance depending on many hidden factors as well as luck, no assumptions about the *player* can be

2 http://creators.xna.com/en-GB/

made. The meta-game is played in turns where each turn has the following elements:

1. the *master* chooses a game's state (i. e. decides on a difficulty setting),
2. the *player* plays one 'round' of the game in this state, and
3. the *master* receives the feedback on whether the state was 'too difficult', 'just right', or 'too easy' for the *player*.

The *master* aims at making as few as possible mistakes, that is, at choosing a difficulty setting that is 'just right' as often as possible. We present an algorithm for the *master* with theoretical guarantees on the number of mistakes in the worst case while not making any assumptions about the *player*.

The crucial part of the analysis consists of representing a set of difficulty settings as a (finite) partially ordered set. As we can not make any assumptions about the *player*, we compare our algorithm theoretically and empirically with the best static difficulty setting chosen in hindsight, as is commonly the case in online learning [9]. This baseline has a natural relation to the real life: it reflects the best setting that the player would choose if she knew the game and her abilities.

While the theoretical results show that the *master's* algorithm's performance depends on the properties of the partially ordered set and in a good case is on par with a full information predictor (i. e. the one who at each round receives information about *all* states rather than one), the empirical studies we have conducted demonstrate even better results.

1.4 SUMMARY

In sum this work presents several new methods for dynamic difficulty adjustment in order of increasing generality.

Chapter 3 presents a simple way to match an opponent's skill by ranking available strategies and monitoring the opponent's choices. This heuristic works as long as (i) the opponents are relatively similar, (ii) full information (about available strategies and the opponent's action) is available, (iii) the strength of the adaptive player is unlimited.

Chapter 4 presents two supervised learning approaches to the dynamic difficulty adjustment problem, provided there is necessary training data available: the game traces from a sufficient amount and sufficiently different players. As for any other supervised learning algorithm, the quality of the resulting models depends on the quality of the provided training data.

Finally, Chapter 5 presents a theoretically sound, online dynamic difficulty adjustment algorithm that does not impose any assumptions on games or players that it can be applied to. This algorithm

works as long as (i) for any two game states it is possible to say whether any one is more difficult than the other, (ii) the oracle reporting the player's perceived difficulty of the current state (too difficult, just right, or too easy) is available.

1.5 BIBLIOGRAPHICAL NOTES

Most of the research presented in this dissertation is a joint work with Thomas Gärtner and has appeared elsewhere, in one form or another.

The results in Chapter 3 were published in the proceedings of *the 4th International Conference on Games Research and Development CyberGames* under the title "Online adaptive agent for connect four" [58].

The first algorithm presented in Chapter 4 has appeared in the proceedings of *the 12th International Conference Discovery Science* in a paper titled "Player modeling for intelligent difficulty adjustment" [59]. The second algorithm is a joint work with Martin Mladenov and was published in the proceedings of *the International Workshop on Machine Learning and Games*, colocated with ICML 2010, in a paper titled "Offline learning for online difficulty adjustment" [62].

The research in Chapter 5 was published in *Advances in Neural Information Processing Systems 24* [61] under the title " Predicting dynamic difficulty". An extended abstract describing this research appeared in the proceedings of *the 7th International Workshop on Mining and Learning with Graphs* [60].

The empirical studies in Chapters 3 and 5 is a joint work with Laurenţiu Ilici and Jiao Jian Wang. The corresponding paper, titled "Dynamic difficulty for checkers and Chinese chess", was published in the proceedings of *the 2012 IEEE Conference on Computational Intelligence and Games* [42].

2 LANDSCAPE

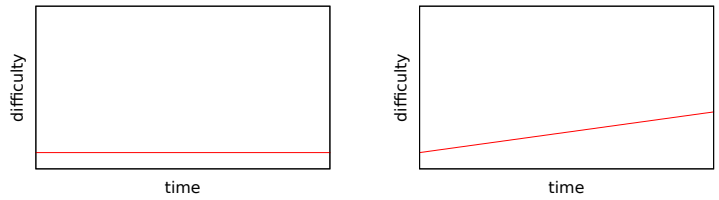
We begin with a brief history of the dynamic difficulty adjustment in video games.

The simplest difficulty a game can have is a static one. However, it is also the most boring difficulty a game can have. Game developers have used the concept of a learning curve to produce a difficulty curve in their games: The difficulty changes as the game progresses, but in a strictly prescribed way. For example, the further a player is in the game, the more difficult it becomes. The shape of the curve is something for a developer to decide, for a few examples see Figure 1. Note, that the difficulty in this case changes in exactly the same way, regardless of a particular player.

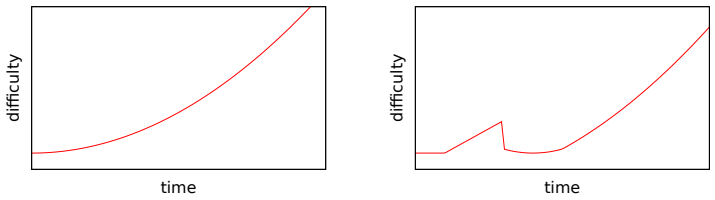
The next best thing is to provide the player with a few variants of the difficulty curve, i. e. a static choice of the difficulty level in the beginning of the game (Figure 2). Unfortunately, that can lead to a situation where if the player faces an insurmountable obstacle, she will have to restart the whole game. A logical solution here is to give the player an ability to switch between the difficulty levels on the fly. While removing one problem, this method introduces a different one: The necessity of going back and forth between the gameplay and the settings when the tasks become too difficult or too easy (i. e. the player's learning curve does not match the selected difficulty curve) disrupts the flow of the game and breaks the immersion, two of the components that are responsible for the game being enjoyable.

Furthermore, it is not an easy task to create even one difficulty curve in a satisfactory way. The obstacles multiply when game developers have to provide several different ones. For instance, how to decide how many? The choices range from the standard three ('easy', 'normal', and 'difficult') to apparently an infinity (a continuous slider on the settings screen instead of a discrete choice). How different should they be from each other? Would switching from one to another disrupt the story and the world? Answering these questions together with constructing different difficulty settings requires additional effort and extensive testing, costing developers time and money.

An alternative mechanism is to tie the difficulty adjustment not only with the player's progress through the game, but also with her performance. One of the earliest examples of this can be found in Zanac, developed in 1986 by Compile Co., Ltd. It possessed a system, called the "Automatic Level of Difficulty Control" or ALC [43], which was unique at the time. It is constantly monitoring the player's performance, keeping track of such variables as the amount and types



(a) always the same (b) steadily growing



(c) easy to learn, difficult to master (d) inventive

Figure 1: Examples of difficulty curves' shapes.

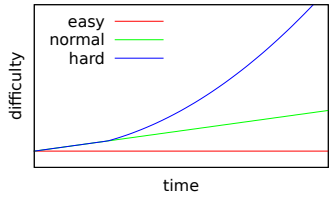


Figure 2: Several difficulty curves in a game. The choice of a difficulty setting constrains the player to following one of them.

of enemies killed, power-ups collected, lives lost, etc. Based on preprogrammed rules this data leads to changes in the challenge that the player is facing, for instance, a greater or smaller number of enemies on the screen [43]. According to the players' reports ALC produces a challenging game that adapts to the player's skill level [88].

To create a similar dynamic difficulty adjustment mechanism, one needs to answer three questions:

1. What to monitor?
2. What to adjust?
3. Plus how to adjust (2) given (1).

On the second glance, though, assuming that

- nothing stops us from monitoring everything possible, and
- everything adjustable can be adjusted to the purpose of tweaking the difficulty,

the main question really is how: How to adjust what we can, given what we observe. More than a few games published in the last twenty

years use a dynamic difficulty adjustment system in one way or another. The common theme is that all of them use a heuristic to answer that question, i. e. a set of rules tuned for this particular game that was created by game developers and tested on players up to some degree. In the following we will look at some examples and discuss their positive and negative sides.

2.1 HEURISTICS

One quite popular approach to dynamic difficulty adjustment is a so-called *rubber banding*. The name means that the player and her opponents are virtually held together by a rubber band: If the player is "pulling" in one direction (playing better or worse than her opponents), the rubber band makes sure that her opponents are "pulled" in the same direction (that is they play better or worse respectively). While the idea that the better you play the harder the game should be is sound, the implementation of the Rubber Band AI often suffers from disbalance and exploitability. A typical example of this approach can be found in some racing games, for example, the *Need for Speed* series, where the opponents' cars are driving faster the further ahead of them the player is, and slower the further behind she is. This behaviour appears unnatural and irritating if players become aware of it.

A particularly well-known example is the game called "Mario Kart" by Nintendo [55]: Players compete in go-kart races, controlling vehicles with inherently different speeds, that effectively serve as the different difficulty settings without needing to call them that way. "Mario Kart" employs *rubber banding* in a few different ways. One of them is a quite direct implementation of the above description: The speed of the computer-controlled opponents is adjusted depending on the player's performance. If the player is not doing well, all the opponents will slow down. If, on the other hand, the player is racing proficiently, the opponents will also speed up and make sure they are right there with her. Another dynamic adjustment concerns the amount and types of power-ups that are available to the player during a race. The weaker and slower players are given more and better power-ups than the faster ones.

Overall, the system serves its purpose in creating a game that everyone can play and win, regardless of their skill level. It also makes sure that there are some opponents around the player at all times, which is important in a combat-oriented racer. Unfortunately, apart from the fact that the system appears (i) to punish the skilful players, and (ii) to be easily exploitable: The best strategy seems to be to artificially slow yourself down to ensure that opponents do not go too fast and that you get all the powerful weapons. Which is probably not what the game designers intended.

Another example of dynamic difficulty adjustment is provided by most of the modern computer role-playing games (CRPG): As soon as the player's character gains a new level, the monsters' characteristics (health, inflicted damage) or the areas (types and amount of monsters) are adjusted accordingly, the "level" value being considered a universal indicator of a player's strength. Unfortunately, even in the very numerical world of a typical CRPG, with all properties expressed in numbers and their relations in formulas, this single value represents neither the character's strength, nor the player's skill with the game. It is all too easy to create and develop a character who is not fit for the game world and its levelling up does not mean it becomes any stronger, but the game follows its logic and becomes too difficult as it progresses. On the other hand, give a strong character together with the instructions on how to develop it during the game to a newbie, and the game would be too difficult for this player, even though the character can deal with it potentially.

The downsides of this become obvious in an open-world game, where game designers have to ensure that the player can go wherever she wants to and do whatever she sees fit without breaking the story or the feeling of slowly gaining power. An example of such a game together with its typical problems is "The Elder Scrolls IV: Oblivion" by Bethesda Game Studios [87]. In accordance with the principles described above, in "Oblivion" enemies scale their levels and their presence based on the player's level. City guards are always 2-5 levels above the player, bandits 2-5 below, and so on. This helps to make sure that no matter where the player decides to go, she does not feel as if she couldn't go there. Unfortunately, it is rather difficult to decide what to scale exactly and how much. In "Oblivion" scaling permeates nearly every aspect of gameplay, from how strong enemies are, to which enemies actually spawn, to what they carry, even to what items shops could sell. As a result of this, after the player reaches a certain level, the lower branches of enemies' hierarchy, such as wolves and lower-end Daedra, simply disappear from the world. This not only brings with itself the problem that the world looks unbalanced. Rather, certain quests become impossible to complete as they depend on the presence of creatures that are not there any more. On the other hand, common road bandits would start wearing the most powerful armour in the game, disrupting the economics of the world. The quest enemies would be summoned of such a high level, that the player's allies wouldn't have any chance against them. These problems were important enough to include warnings about them into the "Oblivion" own official strategy guide [64].

Apart from breaking the story and disrupting the player's sense of growing power, this also creates a ground for exploits. The best, even if probably unsatisfying, way to ensure that the game stays easy, is

for the player to choose to never level up, which goes rather against the whole mechanics of a CRPG.

One of the seemingly simplest examples of the 'observe → adjust' approach can be found in "Half-Life 2" by Valve Corporation [32]. Commentary Mode in Episode 1 and 2 reveals that the game slightly modifies the contents of the supply boxes the player encounters, depending on the protagonist's status at the time the player breaks them. The full health signifies a relatively easy play-through and results in smaller healing items or random ammo, while the low health prompts the game to put something more useful in the boxes, such as +20 first-aid kits. The observation consists of a single parameter and the game world is modified in a way that does not interfere with the player's experience.

The next one by complexity is "Max Payne" by Remedy Entertainment [56]. While the game has an explicit choice for the difficulty setting (Easy, Medium, and Hard), it also includes a dynamic difficulty, whose way of adjusting depends on the setting chosen by the player. The observable parameter is a number of times the player has died repeatedly. On all settings multiple deaths prompts the game to provide the player with more health items, but the more difficult the setting, the smaller the amount of these bonuses [27]. There is some evidence that apart from that the game also very slightly adjusts the level of aim assistance the player gets, and increases the enemy's health a bit depending on her estimated skill level and the projected success [92]. Note that also in this case the adjustments are minor and easy to calculate in, taking into account that the enemies are all spawned in the predefined locations. That ensured that game designers could test the effects and tune them to desirable values.

The Capcom's "Resident Evil 4" [68] went much further than that both in terms of observables and adjustables. Even though the full information about the algorithm is not disclosed, it is known that it is a quite complex heuristic that takes into account almost everything that the player does in the game, from how many times she died to how fast she went through the areas. Based on this data, the algorithm adjusts in a probabilistic way multiple parameters of the game world, from the amount of the opponents to their behaviour. Judging from the players' responses this is a particularly successful implementation of dynamic difficulty adjustment.

The game that is famous for, among other features, its dedication to produce adjusting challenges is "Left 4 Dead" by Valve Corporation [50]. Here, the game's dramatics, pacing, and difficulty are controlled, or to express it more precise, created by the artificial intelligence called the "Director". It is entirely procedural, which means that none of the interactive elements of a level (triggers, flags, or enemies) is predefined. Rather, they are all placed and spawned anew,

depending on each player's current situation, status, skill and location, creating a new experience for each playthrough [1]. To quote:

> The Director takes into account the "stress level" of every individual survivor. It does not want anyone to experience a boring game, nor does it want someone to get a heart attack by having a constant stream of zombies, which would really make you numb to the excitement. ... The Director is a wonderful thing – it keeps you guessing, it makes every experience unique, and most importantly by changing it up it keeps you fully immersed and does not let you fall into the same boring pattern of play.

The benefits of a well-designed system in this style are that it yields consistent, well-paced play, and since it is transparent and non-intrusive, it is easy for the player to forget that it is there, so they feel a greater sense of accomplishment when they complete a particular challenge, even if the bar was continually lowered until it was within their reach.

One of the downsides is that a fraction of players feels cheated if the game's challenge was being altered behind their backs while they were playing. Another one is that a system like this is relatively reactive, i. e. it has to observe you for a while until it is able to make a good guess about how to react to your play.

To finish our review of different dynamic difficulty adjustment systems employed in commercial video games, we will talk about "God Hand" by Clover Studio [30]. On the one hand, it follows the above strategy (observe $\rightarrow$ adjust). On the other hand, it is strikingly different because it does not hide anything from the player. Rather, the observations and the resulting adjustments are an important part of the interface: In the bottom left hand corner of the screen there is an indicator of the actual difficulty meter. As the player successfully progresses through the game, the meter fills up and when it reaches the top, the difficulty increases and the meter is emptied. As the player sustains the damage, the meter decreases, and with it the difficulty. The difficulty is adjusted by letting the enemies use better strategies and hit harder. The game rewards the player both implicitly, by giving her the chance to succeed over impossible odds, and explicitly, by awarding her extra points which are converted to cash at the end of each level. Cash is spent on new moves to help the player beat down more thugs.

Interestingly enough, "God Hand" was created by the same person as "Resident Evil 4", Shinji Mikami. Judging by the players' responses, both systems provide excellent pacing and a sense of accomplishment. However, due to the fact that "God Hand" does not attempt to hide anything and makes the whole dynamic difficulty adjustment explicit to the player, it avoids the "unfair play" problem.

As we have seen above, there is quite some interest in creating and incorporating dynamic difficulty adjustment systems into commercially produced games. As we will see below, there are also a few researchers from the machine learning and artificial intelligence communities who also investigate the question of how to create a dynamic difficulty adjustment mechanism.

2.2 ACADEMIC APPROACHES

We will start our review from the body of work where the same general scheme as above is assumed: observations $\rightarrow$ adjustments. As such, the dynamic difficulty adjustment in this research is explicit, and usually is the ultimate goal of the algorithms being developed. On the way we will also look at a couple of examples of how the observations of players' state can be made. After that we will talk about the implicit systems, where the dynamic difficulty adjustment is not the goal, but rather a side product of an adaptive or intelligent game engine.

2.2.1 *Explicit dynamic difficulty adjustment*

Most academic researchers of the explicit dynamic difficulty adjustment base their investigations on the same problem structure as the one used by game developers:

- repeat until the game ends:
  - observe the player's state;
  - adjust game parameters.

The open questions in this case are again the same as the ones game developers are striving to answer:

1. What to observe?
2. What to adjust?
3. What is the mapping between observations and adjustments?

From these three questions the second one is the least susceptible to the scientific approach. As we have seen above, in the dynamic difficulty adjustment systems employed in actual games, the parameters that are being adjusted range from something as simple as an item in the next chest to pretty much the whole layout of the next level, including the strategies of opponents generated on this level. We feel that it is the natural state of things: The developers of any particular game know best which parameters influence the difficulty and how. Even though it is tempting to imagine a future where this information is available automagically, to the best of our knowledge there are currently no attempts at solving this problem.

Below we will review existing approaches to the other two questions, observations and mappings.

2.2.1.1 *Observations*

The goal of having any observations at all in the context of dynamic difficulty adjustment is to provide an input for the adjustment mapping. In other words, we have to be able to tell based on some information, when to adjust. Recall that the purpose of tuning the difficulty on the fly is to keep the player in the "Zone" or in the "Flow" [84], to match the challenge to her current abilities. Therefore, we need to be able to tell when she leaves the "Zone", when the ability does not match the challenge any more.

Even the games without the dynamic difficulty adjustment usually possess a so-called difficulty curve, where the game difficulty changes depending on how far the player is in the game. It is assumed in this case that the progress alone serves as an indicator of the ability, i. e. if she has gotten that far, she needs to and should be able to deal with the new level of challenge.

In the heuristics described in the previous section the observations usually are made of the player's performance; the choice and the conversion of primitive attributes (health points, time to progress, etc.) to a single variable, i. e. "the game needs to be harder/easier", is implemented on a case by case basis by game developers. Some of the researchers adopt a similar approach: They develop a dynamic difficulty adjustment algorithm, put it into a game of their choice, and create an observational heuristic tailored to that game that provides the algorithm with observations.

Fortunately, there are also a couple of methods that can collect the observations for the dynamic difficulty adjustment in a more independent and universal manner. The first one that we will look at is Player Modelling, which is a relatively new area of interest for the researchers in the context of computer games [40]. Player Modelling is used mainly for two different goals. On the one hand, it allows, at least in theory, to transfer human behaviour onto computer opponents (we will talk about this application in more detail later). On the other hand, creating, collecting, and processing models of the players leads to the reality-based player types as opposed to the predefined {"beginner", "intermediate", "advanced"} set. Once these player types are discovered, game developers can tailor their game to each type separately. Any new player is then monitored as well, but not to decide about her state, rather, to determine to which type she belongs and to offer her the most fitting experience.

Charles and Black [10] discuss the need of player modelling for creating games that adapt to players, where adaptation is not limited to the choice of the right challenge. They describe a general framework that places together player's preferences, in-game observations,

learned models, and in-game adaptations. They also suggest how the learning tasks of the framework can be solved by using neural networks, both in a supervised and non-supervised settings. These tasks, however, can be solved with a multitude of machine learning tools. We have to remember though, that they have their own particular properties that render some of the approaches less applicable than others. For instance, in a standard release-once mode of game development, when after the development process is finished the game is released to the public and no further development takes place, the training data that is representative of the whole player-base could be costly to obtain. The learning or the prediction or both should happen online, in other words during a single game, and the acceptable performance should be achieved as quickly as possible.

Schadd et al. [72] describe a way to classify the playing style of an opponent into one of the several pre-defined models, using an RTS game as a test bed. The models are manually created beforehand and depend heavily on the game in question. The authors state that to simplify the classification process, the models are organized in a hierarchical way. The classification then happens on each level of the hierarchy independently. The classifiers are adapted from the repeated games theory, their parameters are adjusted based on the training data (games versus particular opponents). Both classifiers show good performance during a game, but with a significant delay (one with seven minutes, another after forty minutes in games lasting on average for fifty minutes). All this shows the potential problems of such approach: (i) difficulties with creating and the inherent heuristic nature of hand-crafted models, (ii) the parameters have to be adjusted based on the real players, since the artificially created training data (i.e. games played versus a specific bot or bots) most probably fails to represent them sufficiently well, (iii) by the time the needed confidence level of the prediction is reached, it is probably too late to make any significant changes.

Hawkins et al. [35] show how particle filtering [66] can be used to learn the models by collecting the data from a multitude of players rather than create them explicitly by hand. Another advantage of this approach is that the observable parameters are translated into model parameters and vice versa explicitly, which means that (i) assigning a new player to one of the models is an automatic, dynamic process; (ii) the dynamic adjustments can be read off the models.

Yannakakis and Hallam [99] and their previous work describe results of using neuro-evolution [98] together with a Bayesian network to create an explicit player's model in a modified version of Pac-Man or a self-made game Dead End. They compare the interestingness achieved by using player models with hand-crafted and with random choice of parameters. From the results one may conclude that using an explicit player model is better than leaving things to chance.

One of the few examples of player modelling done on real world data is the work of Drachen et al. [19]. The authors use self-organized maps (SOMs) [49] to infer player models from the data collected from Tomb Raider: Underworld [91]. The raw data consists of game sequences recording events that a player goes through. For the study, six high level features were extracted from the collected data. Note that the features are hand-crafted by the researchers themselves. The data then was clustered (using a SOM), and the resulting clusters interpreted in terms of the features used, showing four distinct player types.

Note that there is an implicit assumption used above that the player's performance (i. e. the parameters that can be read from a game's state) is correlated with her emotional state or her place inside/outside of the "Zone". While there is definitely some strong evidence in favour of this assumption, it also raises a question: Why not measure her emotional state directly? If we can reliably tell, whether the player is bored or stressed out, that would be a perfect input for a dynamic difficulty adjustment algorithm.

Emotional state of players (and humans in general) can be perceived and estimated through a variety of observable signals, in particular physiological ones (such as cardiovascular responses, muscle tension, respiratory rate etc.) [65]. Especially of interest for dynamic difficulty adjustment is the anxiety level. Liu et al. [53, 54] describe how it could be used for adjusting the game's difficulty level on the fly. The procedure is as follows (on a person by person basis):

1. To generate a model:
    - collect the signal data (ECG (Electrocardiogram), ICG (Impedance Cardiogram), PPG (Photoplethysmogram), Heart Sound, GSR (Galvanic Skin Response), peripheral temperature, and EMG);
    - extract multiple features (the detailed description can be found in [53]);
    - use a learning method to extract the underlying state (in this particular case, a regression tree), the known labels are extracted from the self-reports;
2. To infer and use the players' emotional state, use the generated model to estimate the anxiety level and act accordingly.

Note that in these papers the difficulty levels are predefined (as usually, designing them would be left for game developers). Also, a relatively simple schema for adjustment is used (i. e. if the anxiety is high $\rightarrow$ go to the lower level, if it is low $\rightarrow$ go to the higher one). Nevertheless, according to the empirical data received, it results in a better game experience (better performance and better self-reported level of engagement) for the players involved.

This evidence shows that affective computing can be used as a complementary approach to player modelling based on the performance features to design a universal way to answer the question number one from above: "What to observe?" Next on our list is "How to adjust?"

2.2.1.2 *Adjustments*

After the observations, the next component in a dynamic difficulty adjustment system is the mapping between them and the adjustables. Once the input and the output (the player's state and the adjustable parameters' space) are fixed, we have on our hands a learning problem: given a particular player's state, find the matching set of adjustments. Which method to choose in order to solve it would usually depend on external constraints, e. g. can the learning be performed in an offline mode or should it be online, how much of the training data is available, etc. Below we will describe a few published approaches that were applied in different circumstances.

An example of an in-game online adaptive agent based on a modified Reinforcement Learning (RL) approach is presented in the work of Danzi de Andrade et al. [15]. Informally, in an RL setting an agent is interacting with an environment by perceiving it, executing actions and receiving rewards. The agent's goal is to learn to execute such actions (depending on its state) that maximize its reward in the long run (for a formal definition and ways to solve an RL problem see the book of Sutton and Barto [83]). Danzi de Andrade et al. [15] use Q-learning together with a challenge function to build intelligent agents that automatically control the game difficulty level. Q-learning produces a ranking on a set of actions available to the agent in any given state. The ordinary Q-learning agent learns the value function by iteratively collecting the information about the environment, and at every step chooses the best possible action given the current value function. In contrast to that, the agent designed by the authors at every step chooses the action depending on the current value function and the manually created challenge function, which reports to the agent whether its performance is better or worse than the opponent's. The output of the challenge function is used as an indicator of the perceived difficulty: If the opponent is doing worse than the agent, the game is too difficult, and vice versa. First, the available actions are ordered with regard to their values: The maximum value signifies the best action to take to win the game, the minimum one, the worst. Next, if according to the challenge function, the game appears to be too difficult (easy) for the opponent, the agent chooses the action that is worse (better) than the one it made before. This approach was evaluated empirically in the context of a fighting game, Knock 'em, and proved successful against the algorithmic agents developed by the authors. Unfortunately, this approach faces all the problems associated with reinforcement learning [44]: bad scaling to high-dimensional

spaces, difficulties in dealing with continuous spaces, long learning times, which means that it would require a substantial amount of improvement to use it with games that people play.

Another example is described in the work of Hunicke and Chapman [41]. They look at the problem of dynamic difficulty adjustment in computer games from the point of view of inventory theory. Games are viewed as the flow of supply and demand in players' inventories. The game engine traces and evaluates the player's performance and attempts to adapt the game world in such way that the game keeps being challenging for the player. The adaptation takes the form of adjusting the characteristics of the player's opponents, their numbers and locations, etc. While this approach certainly has its place in solving the problem of developing games that adapt themselves to the players, there exist games with rigid rules, where changing the game world is not possible or where players do not possess inventories in any sense.

A setting of interacting entities is considered in work of Herbrich et al. [38]. In this case the difficulty adjustment happens through creating and matching teams of players in such a way that the skill ratings of resulting teams are approximately the same. The authors propose and evaluate a novel approach to estimate the skill ratings based on the outcomes of multiple games. A possible application of this approach to "one player vs one (or more) in-game agents" would be to have a set of in-game agents possessing different skill ratings. Then, based on the estimated rating of the player, an appropriately skilled agent can be chosen to create a "right" difficulty level. Still, to produce the estimation the player needs to play multiple games.

2.2.2 *Implicit dynamic difficulty adjustment*

Above we looked at the examples where researchers tackle an explicit dynamic difficulty adjustment problem, i. e. creating a system of observations together with adjustments whose purpose is to keep the game's difficulty level matched to the player's state. There is also a separate area of research that deals with a more global problem of creating an intelligent or an adaptive game AI, such that the games employing it appear intelligent, flexible, and possessing human-like characters. As a consequence of these qualities in such systems the automatic adjustment of the difficulty happens or should happen on the fly and automagically. In the following we will look at the directions that this research takes.

2.2.2.1 *Adaptive game AI*

So far we have made no assumption about the power or the intelligence of the game engine. If we look at the games available today though, programming a decent AI for quite a large fraction of them is

a challenging task (real-time strategies come to mind first, but also tactical games that would benefit from players' opponents being smarter and able to work cooperatively). Therefore, it does not make much sense to talk about dynamic difficulty adjustment for these games without considering the bigger question of developing a better or, more specifically, intelligent, context-aware, and dynamic AI.

One example of such research can be found in work by Bakkes et al. [4]. The authors apply case-based reasoning together with an "adaptive mechanism" that is treated as a black box (as a possibility reinforcement learning is mentioned): Games are represented as feature vectors (features being time stamps) and in any given game and time the system looks for a most similar game from already recorded and recovers the best strategy in this situation. The approach was tested in a commercial real-time strategy (RTS) game. While the authors do not focus on dynamic difficulty adjustment, they suggest that it happens as a side-effect of on overall adaptive AI.

Another example of Adaptive AI is called Dynamic Scripting, introduced by Spronck et al. [81, 82], Szita et al. [85], (as opposed to the static scripting of game's characters): The scripts are broken into basic, elementary pieces and the AI learns to associate these proto-scripts with in-game situations depending on the positive or negative feedback. The feedback can be tailored to match the purpose of adaptation. If the learning is allowed to continue during the play as well, the AI becomes adaptive, able to change depending on new circumstances, for example, the change in player's skills. Thus, dynamic difficulty adjustment happens as a natural consequence.

2.2.2.2 *Opponent modelling*

We have talked about Player Modelling for the purpose of creating more truthful observations of players. There is also a different application of it that could potentially lead to dynamic difficulty adjustment, namely transferring human behaviour onto computer opponents. The closest example of this is presented by Bauckhage et al. [5]. The authors use a combination of machine learning methods to construct opponents in a first person shooter game with the sole purpose of producing behaviour that visually appears human-like. To evaluate the performance they played short video clips of people and computer opponents playing the game and asked the viewers their opinion on whether the player was a human or a computer. The results are very encouraging, the viewers misidentified the imitation opponents as humans 69% of the time and correctly identified human players also 69% of the time. It remains to be seen, though, how this can be used in creating a dynamic difficulty adjustment system.

Ponsen et al. [67] pursue a related but not quite the same goal. The purpose of opponent modelling in their case is to enhance the strength of the opponent in a situation where the assumption of ra-

tional play does not hold, i. e. the humans are incapable of playing perfectly, and, therefore, it is worthwhile to explore other options rather than the moves optimal under the rationality assumption.

Yannakakis and Hallam [100, 101] argue against an explicit difficulty adjustment, rather they want to change the behaviour of opponents to optimize the game's 'interestingness', where the challenge is taken into account as part of the 'interestingness' (the second one being the 'curiosity', which they define as the unpredictability of opponents). It is indeed rather curious how neatly the opponents' behaviour can be split in this particular case into challenge (speed) and curiosity (variance). To create a model, first, the data from the played games was collected together with the players own evaluations of what was more fun. Then, a neural network was trained (evolved) [63] with this data to produce a mapping: game and player features $\rightarrow$ 'fun'. Using the resulting model, the authors designed an adaptive mechanism and tested it in games with children. The first results demonstrate that the participants tend to rate the adaptive version of the game as more fun than the static one.

The following example of opponent modelling goes even further away from dynamic difficulty adjustment. Kang and Tan [45] attempt to create an adaptive agent with the purpose of giving players personalized recommendations and advices. They use a combination of reinforcement learning and neural networks to create and train player models. Unfortunately, the authors evaluate their agents on hand-crafted simulation of players. While it follows from the empirical results that having an adaptive agent is better than a static one in terms of correct recommendations, it is also obvious that the proposed approach is too slow to deal with actual humans. Even though eventually the agent reaches the accuracy of $70 - 80\%$, which means that three out of four recommendations are approved ('liked') by the player, it takes more than a hundred interactions with a player to reach the accuracy of 50%.

2.2.2.3 *Procedural content generation*

An orthogonal idea to Opponent Modelling, i. e. creating adaptive opponents, is to create an adaptive game world. For example, if we consider a racing game, a suitable adaptation mechanism could be creating racing tracks on demand, such that they fit the player's style and skill. That is what Togelius et al. [90] describe. Their approach could be summarized as follows:

1. Build a set of player models;
2. extract an appropriate set of parameters from them (in this case, the fitness values of the racing tracks, i. e. how much a track fits a player model);

3. generate (in this case, evolve) the game world/levels/maps (in this case, racing tracks) tailored to a particular model (set of parameters).

The problem is, everything is a problem here. Building player models: difficult. Choosing parameters: unclear. Constructing levels to fit to parameters: arcane knowledge. Nevertheless, the potential for (implicit) dynamic difficulty adjustment is there, even though the authors never mention it.

On a more general level, there is a question whether it is possible to automatically create a game itself for a particular player. Halim and Baig [33] attempt to answer this question by generating the game world or 'rules' that adapt to a player. Unfortunately, what the authors call 'rules' is rather like choosing a point in a predefined parameter space. All games generated that way are essentially one game, which is underlined by the fact that they use a single agent controller based on human-defined rules to play and evaluate all generated games. It makes one wonder: If a set of rules defined once and for all is sufficient to play all games, do they really provide different levels of entertainment? On the other hand, it could be a valid approach for finding the optimal or close to the optimal set of parameter values. That still leaves the question open: How does that game adapt to a particular player? The authors do not answer this question, but one can imagine combining the search ('generation') procedure with player modelling.

Which is what's happening in the work of Shaker et al. [78]: First the authors create player models using neural networks with measurable features (observables in our previous terminology) and controllable parameters of the world (adjustables) as the input. The player models predict the levels of fun, challenge, and frustration. Then they are used together with an exhaustive search of controllable parameters' space to generate a next level that is supposedly adapted to the current player. Unfortunately, the experiments were conducted on a small scale, which prevents one from making a definite conclusion.

Since this particular line of research is only vaguely related to the topic of this work, we refer a curious reader to other sources about various approaches to generating games' levels, such as a survey on procedural content generation by Hendrikx et al. [37], as well as newer work by Liapis et al. [51], Cardamone et al. [8], Shaker et al. [79]. For work related to generating games' rules see the results published by Hom and Marks [39], Browne [7], Togelius and Schmidhuber [89], Font et al. [23].

2.3 CONCLUSION

We have attempted to review the different approaches to dynamic difficulty adjustment existing at the moment: heuristics, explicit, and

implicit methods. From the limited amount and the breadth of presented approaches it is apparent that this area of research is not yet mapped very well. A learning problem involving humans, their perceptions and reactions, is inherently a difficult one, since humans are difficult to formalise. In our work presented in the following chapters we try to move away from heuristics or specialized approaches and in the end we arrive to a universal dynamic difficulty adjustment mechanism.

3

ADAPTIVE RANKER

3.1 INTRODUCTION

The problem of a dynamic difficulty adjustment can be viewed in the context of an interaction between a player and one or more in-game entities, e. g. computer opponents in a real-time strategy, bots in a first-person shooter, non-player characters in a role-playing game. We will call a specific in-game entity involved in this interaction the agent. It is natural to assume that at any given time the agent has a set of actions (strategies) available to it. The question of how to adjust the game difficulty automatically can be formulated as what action (strategy) should the agent choose as next.

Consider the situation where player's and agent's goals are mutually exclusive, i. e. when one of them 'wins', the other one 'loses'. This is often the case in two-player board games, e. g. in tic-tac-toe, connect four, or backgammon, but also in "real" computer games such as real-time strategies. Assuming that the game contains states labelled 'winning' and 'losing', in any given game state actions leading to 'winning' states are better than the ones leading to 'losing' states. In other words there is a naturally occurring ranking on the available actions. Knowing the ranking, the agent can evaluate the performance of the player by looking at the rank of actions performed by the player and choose its own actions accordingly. This principle allows us to create an online adaptive agent, ADAPTIVE RANKER (Algorithm 1).

Under the assumption that the player wants to win the game, the ranking of {action, state} pairs should reflect the quality of an action with regard to its relation to the winning and losing states. In an extreme case an action that allows the player to win immediately is the best one and should have the highest rank, while an action that forces the player to lose is the worst one and, correspondingly, should have the lowest rank. For all the actions whose influence cannot be seen immediately, there should be a way to look ahead, to be able to determine their future consequences.

One particular way to enable the looking ahead is to represent a game with a tree. A *game tree* is a directed tree, where game states are represented by vertices and actions are represented by edges that lead from a vertex to its children. The *complete game tree* is the game tree starting at the initial state and containing all possible legal states. A game representation by a tree in this way is also called an extensive form of a game. Note that with a complete game tree one can find an optimal sequence of moves for both players that will guarantee either

Algorithm 1 ADAPTIVE RANKER

Require: a ranking function for a set of {action, game state} pairs
  repeat
    Wait for the player to finish her turn.
    Retrieve a set of {action, state} pairs, where actions are the actions that were available to the player on the previous turn, and states are the states that these actions would have led to.
    Retrieve action a performed by the player on the previous turn that led to the current state s
    Retrieve the rank of $\{a, s\}$ using given ranking function.
    Retrieve a set of {action, state} pairs available to self on the current turn.
    Return the action of an {action, state} pair with the same rank as $\{a, s\}$.
  until the game is over.

a win for one of them or a tie. This is called *solving a game*. For formal definitions as well as a discussion of various aspects of game trees we refer the reader to a book by Fudenberg and Tirole [26].

The leaves of the complete game tree provide the genuine information about which actions (or rather action sequences) lead to which outcomes, allowing to evaluate them correspondingly. Starting from these values and traversing the tree upwards, one can assign labels to every node in a game tree. Starting with the leaves, we attach a label $+1$ to the agent's winning states, 0 to the draws, and -1 to the losing ones. Note that in the parent state of any leaf the choice for either the player, or the agent is clear: The former would choose an action leading to the -1-state, while the latter the one leading to the $+1$-state. Therefore, depending on whose turn it is in the leaf's parent, we can assign it the appropriate label. Applying the same logic, the labels can be propagated upwards in the tree all the way to the root, marking the quality of all the states and of the actions leading to them. See Figure 3 for an example.

Due to the complexity of most of the games that people play, the corresponding game trees are so large that the traversal of paths in them would take an unreasonably long time. Consider, for example, checkers: Solving this game took several years as reported in a book by J. Schaeffer [73]. The other problem with using a game tree to evaluate game states is presented by games with imperfect information, such as Poker.

In such situations one could use various machine learning techniques to learn an evaluation function for a given game or forego the evaluation scores completely and directly learn the ranking on state-action pairs or possibly state-action pair sequences (taking into account the history).

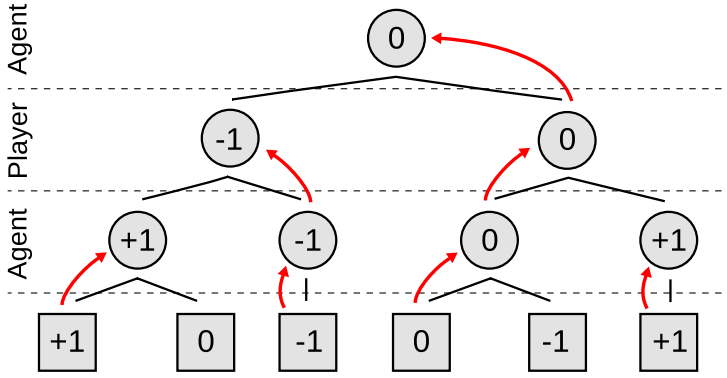


Figure 3: A fictional complete game tree with propagated labels. The square nodes represent leaves. The red arrows indicate how the labels are propagated upwards depending on who chooses the next action. Note, the agent is maximizing the value of the next state, while the player is minimizing it.

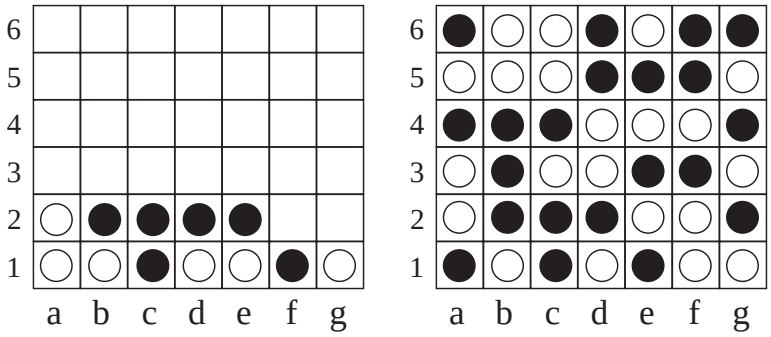
As a proof of concept in this chapter we present ADAPTIVE RANKER implementations for two board games, connect four (Section 3.2) and checkers (Section 3.3). Since our work does not include research questions connected to dealing with large game trees or learning evaluation functions and rankings, we limit ourselves to using the depth search restricted MINIMAX algorithm (described below) with manually designed evaluation functions. To evaluate the resulting agent we design a test environment consisting of two parts. The first part contains computer opponents with distinct skill levels. The second part provides an environment where human players can play against the developed agents.

3.2 CONNECT FOUR

Connect four is a game for two players. Each player has 21 identical stones. We assume that one set of stones is white and the other is black. The game is played on a rectangular board consisting of 7 vertical columns of 6 squares each. If a stone is "dropped" in one of the columns, it will "fall down" to the lowest unoccupied square. No stones can be dropped into columns that are already full. One move consists of placing one stone in one of the columns.

The players make their moves in turn. The goal of the game for each player is to get four of her own stones connected either vertically, or horizontally, or diagonally. If all 42 stones are placed on the board and no such group was created, the game is a draw. In the game displayed in Figure 4a the player with the black stones has built a winning horizontal group. Figure 4b displays a possible draw situation.

In this work we assume that the player with the white stones makes the first move and in the following will be referred to as White. The player with the black stones will be called Black.



(a) Black wins: b2-e2.

(b) After 42 moves no winning group was created.

Figure 4: Possible endgame situations in connect four.

Before describing the adaptive agent for connect four in Section 3.2.3, we present in Section 3.2.1 a set of algorithms implemented to play connect four with distinctly different skill levels and report on how they were used to evaluate the skill levels of human players in Section 3.2.2.

3.2.1 *Testing ground*

The purpose of the testing environment was to provide a set of algorithms that play connect four with distinctly different skill levels, so that (i) they can be used to evaluate the strength of human players, and (ii) the adaptive agent can be tested against them to see how well it can adapt. For this we have implemented four algorithms: Naive, Simple, MiniMax, and Optimal. To establish their relative skill we let them play against each other in several episodes, each episode consisting of 1000 games, where for half of the games one of the algorithms was playing the white side, and for the other half it was playing the black side. From the resulting statistics, i. e. how many times each algorithm has won, lost, or led the game to a draw, we have established the following ranking:

1. Naive. The easiest algorithm to beat.
2. Simple. It almost always beats Naive.
3. MiniMax. The performance of this algorithm depends on its limited search depth; its value was set in such way as to ensure that
   - MiniMax reliably beats Simple and Naive;
   - MiniMax plays as good as possible while still requiring no more than a few seconds for each move.
4. Optimal. The optimal algorithm always wins when playing White.

Table 1: Players' payoffs in connect four.

		Outcome of a strategy		
		White wins	Black wins	Draw
Payoff for	White	+1	−1	0
	Black	−1	+1	0

In the following we describe all four of the non-adaptive computer opponents used in the testing ground and the results of their playing against each other.

Optimal

In 1988 Victor Allis designed the optimal algorithm based on nine strategic rules that describe local situations on the board and proved that White always wins if it plays according to his algorithm [2]. An implementation of the optimal algorithm developed by Giuliano Bertoletti was used in our environment. The engine is written in C and was modified slightly to allow the interaction with the other algorithms in the testing ground.

MiniMax

In terms of game theory a game consists of a set of players, a set of moves (or strategies) available to these players and a specification of players' payoffs for every combination of strategies [26]. For connect four the set of players consists of Black and White and the set of strategies is a set of all sequences of legal moves.

With payoffs defined in Table 1 connect four turns into a two-players zero-sum game: The total sum of gains and losses of both players is zero for any game ending. Two-players zero-sum games are solvable by applying the MiniMax algorithm [70].

Recall that a complete game tree allows to solve the game by traversing it from the leaves up to the root and assigning the appropriate labels to the nodes (Figure 3). In the practical implementations of MiniMax it is usually only a subtree that gets investigated. It is built from the complete game tree by considering a subtree whose root is the node representing the current board situation, and following it downwards for a fixed amount of levels. In this case the tree does not contain the full information any more. While genuine leaves of the original game tree still possess their exact evaluation scores, the leaves of the subtree that result from cutting off the original tree should be evaluated using an evaluation function.

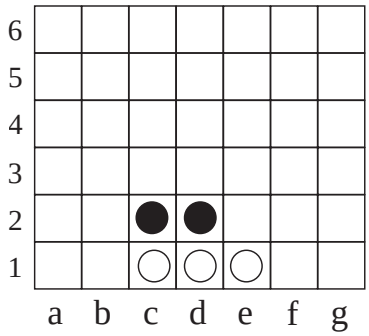


Figure 5: Black has to refute two White threats, one on b1 and one on f1, in one turn, which is not possible.

To construct it, we consider two features of a game state, open lines and forks. Both represent an advantage for their owner, since both can be turned into a win, given an opportunity.

More specifically, four squares on the board forming a straight line either vertically, or horizontally, or diagonally such that three or less of them are taken by one player, and none by the other, are called an *open line*. A former open line of the player is called *broken* if the opponent placed a stone in it effectively discarding this threat.

A *fork* is a connected group of squares of size four or five lying on a straight line, such that (i) both squares on the ends of the group are empty, and (ii) all the squares apart from the ones on the ends are taken by one player. A White fork built with three stones on Figure 5 presents immediate loss situation for Black, since it cannot be refuted. In general, an opponent's fork built with two stones should be refuted as soon as possible to prevent it from becoming a 3-fork. On the other hand building of player's own forks should be preferred, since they offer great advantage.

Given a state, the evaluation function calculates the weighted sum of player's forks, her open lines, and opponent's forks and broken lines. The resulting number is the evaluation score of the state.

The function operates with three weights: w_f, w_{pl}, w_{ol}. Assuming $f_p(v)$ and $l_p(v)$ are the numbers of player's forks and open lines in state v, $l_o(v)$ and $f_o(v)$ are the numbers of her opponent's broken lines and forks in state v, the evaluation score is

$$eval(v) = w_{pl} \cdot l_p(v) + w_f \cdot f_p(v) + w_{ol} \cdot l_o(v) - w_f \cdot f_o(v)$$

The larger the value of $eval(v)$, the better the state v is from the heuristic's point of the view. The values for the weights w_f, w_{pl}, and w_{ol} were chosen empirically and set to 5, 5, and 1 respectively [57].

When using a subtree MiniMax cannot be sure any more that it finds the optimal path through the tree, because it replaces the exact information contained in the genuine leaves with heuristic values. We will call the height of the subtree that MiniMax traverses *search depth*. Increasing the search depth improves the performance of MiniMax

Algorithm 2 MINIMAX

Require: game tree node v, depth d, maximizing $mx \in \{true, false\}$, evaluation function for a game tree node *eval*()
  if $d = 0$ or v is a leaf **then**
    Return $eval(v)$
  end if
  if mx is *true* **then**
    for each child c of v **do**
      $s_c = \text{MINIMAX}(c, d-1, false)$
    end for
    Return maximum of s_c over all children of v
  else
    for each child c of v **do**
      $s_c = \text{MINIMAX}(c, d-1, true)$
    end for
    Return minimum of s_c over all children of v
  end if

at the cost of time needed to evaluate the game tree. The choice of the particular search depth represents a trade-off between how well we want MINIMAX to play and the time spent on calculating one move. While playing versus humans, this time should not exceed a few seconds. The search depth restricted variant of MINIMAX is presented in Algorithm 2.

To improve the performance of MINIMAX the amount of nodes visited for a given search depth can be reduced using Alpha-Beta pruning [70]. Alpha-Beta terminates an evaluation of the move when at least one possibility has been found that proves this move to be worse than the current optimum value. The efficiency of Alpha-Beta depends on the order in which the nodes are evaluated. It has been shown that in optimal conditions, when the first child considered for a given node is enough to prune the others, Alpha-Beta doubles the effective search depth of MINIMAX algorithm [70] (see Figure 6). Randomizing the order in which moves are evaluated results in the average total number of traversed nodes being of the order of $O(b^{3d/4})$, where b is the average branching factor, d is the search depth [48].

Already for search depth 11 on the equipment and the implementation used, the time that it takes for MINIMAX to make a move can reach up to 4-5 minutes. Therefore, we decided to set the search depth of MINIMAX to 10. At this value MINIMAX takes up to several seconds to make a move, which is still tolerable for games played against human players.

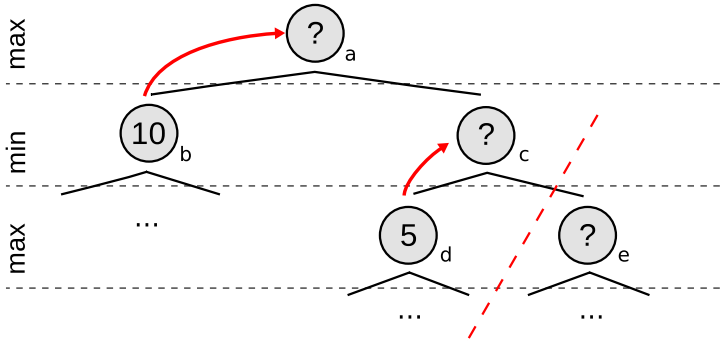


Figure 6: An example of Alpha-Beta pruning. On a level marked *max* (*min*) the maximum (the minimum) of the children's values is assigned to the parent. After node b is assigned 10 and node d is assigned 5, the subtree starting from node e does not need to be evaluated any more: $c = \min\{d, e\} \leqslant 5 < 10 \leqslant \max\{b, c\}$, and is, therefore, irrelevant for evaluating node a.

Simple *and* Naive

The adaptive agent should be able to adapt even to the players of a very low skill level. Therefore, we needed an algorithm that performed badly without making deliberately bad moves. The stronger of two weakest algorithms, Simple, is MiniMax with the search depth set to 1. While quite weak, it still possesses some amount of intelligence due to the evaluation function. Naive, on the other hand, discerns only winning and losing moves.

A move is called *winning*, if it creates for the player a group of four stones connected either vertically, or horizontally, or diagonally. A move is called *preventive*, if it would be a winning move for the opponent.

Naive tests the given board for the presence of the winning and preventive moves. Naive chooses the next move from available ones with the following priorities:

1. Choose uniformly at random one of the winning moves, if there is at least one.

2. Else, choose uniformly at random one of the preventive moves, if there is at least one. (Note that if there is more than one such move, Naive will lose the game regardless of which move it makes.)

3. Else, choose uniformly at random one of the moves that do not make Naive to lose immediately, if there is at least one.

4. Else, choose uniformly at random one of the available moves. (Note that all of these moves are losing ones.)

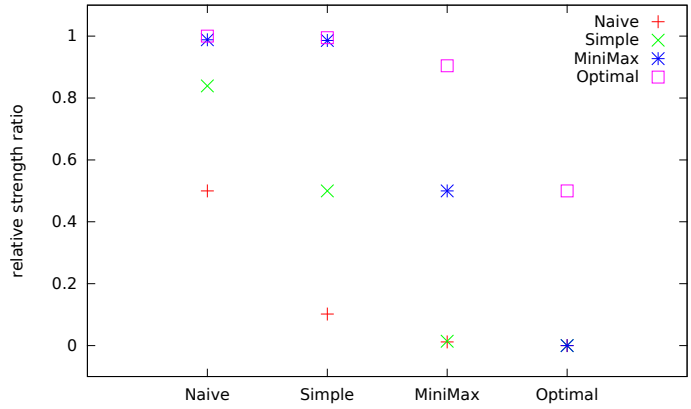


Figure 7: The relative strength ratio of the non-adaptive computer opponents in the testing ground when playing against each other.

Results

To evaluate the performance of the computer opponents, we introduce a metric called *relative strength ratio*. Let i and j be two opponents, who have played $n_{i,j}$ games against each other, $win_{i,j}$ is the amount of games i has won, $draws_{i,j}$ is the amount of draws. Then the relative strength ratio $r_{i,j}$ is defined as follows:

$$r_{i,j} = \frac{win_{i,j} + draws_{i,j}/2}{n_{i,j}} \,. \tag{1}$$

Note that $r_{i,j} \in [0, 1]$ with $r_{i,j} = 0.5$ meaning that the opponents are of equal strength.

The results of all non-adaptive computer opponents playing against each other are presented in Figure 7. From the values of the relative strength ratio we see that they indeed possess distinctly different skill levels, with OPTIMAL being the strongest, and NAIVE the weakest of the four.

3.2.2 *Human testing ground*

(Un)fortunately humans are very good at learning and exploiting the determinism and flaws of computer algorithms. Because of that it was especially interesting to see how the developed adaptive algorithm will score against human players. For that purpose a testing environment was developed to let people play against any of the algorithms used for the testing ground described in Section 3.2.1 as well as the adaptive algorithm described in Section 3.2.3. During the period of three weeks 52 participants played 1347 games.

The algorithms from the testing ground were used to try and determine the skill levels of human participants. Even though each player was asked to play at least 10 games against one of the non-adaptive algorithms and 10 games against the adaptive algorithm, practically

no one held to this suggestion. Instead, most of the players played no more than 4 or 5 games versus each of non-adaptive agents and on average 14-15 games against the adaptive one.

The fact that most of the participants played against more than two non-adaptive algorithms allowed us to attempt to determine the skill level of each player. Let us denote with t_1, t_2, t_3, t_4 the total amount of games played by a specific player p against NAIVE, SIMPLE, MINIMAX, and OPTIMAL respectively, with pl_1, pl_2, pl_3 the percentage of games won by the same player p against NAIVE, SIMPLE, and MINIMAX respectively, and with pl_4 the percentage of games drawn against OPTIMAL. It is natural to assume that players winning against stronger opponents are themselves stronger than players winning against weaker opponents. Therefore, in calculating the skill level of a player we weigh the values pl_1, pl_2, pl_3, pl_4 with weights that reflect the skill level of the respective algorithm. To take into account the fact that most of the players didn't play against all of the algorithms presented on the playground, we sum the statistics only over the algorithms that a particular player played against. Based on these considerations we estimate player's p skill level as follows:

$$s(p) = \frac{\sum_{i:t_i \neq 0} w_i pl_i}{\sum_{i:t_i \neq 0} w_i}, \tag{2}$$

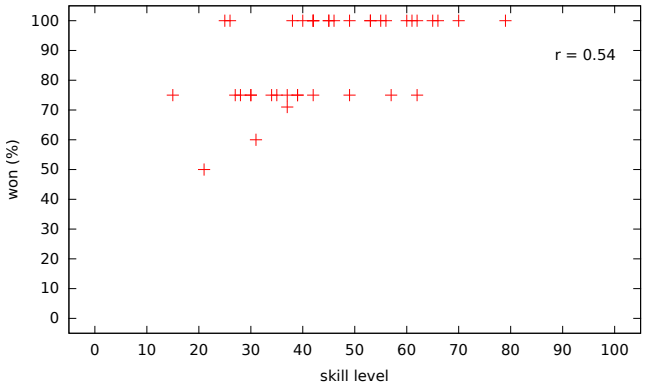
where $w_1 = 1$, $w_2 = 2$, $w_3 = 3$, and $w_4 = 4$ are the weights assigned to NAIVE, SIMPLE, MINIMAX, and OPTIMAL respectively. The weights were set in an ad hoc way, to represent the relative skill of the algorithms.

The results of the experiments are presented in the form of plots of skill level versus percentage of games won (or, in case of OPTIMAL, drawn) in Figure 8. The correlation coefficient values confirm that there exists some dependency between the chosen skill level function (Equation 2) and the amount of games players win against every non-adaptive algorithm. From this data we conclude that $s(p)$ bears some resemblance with the actual skill level of player p.

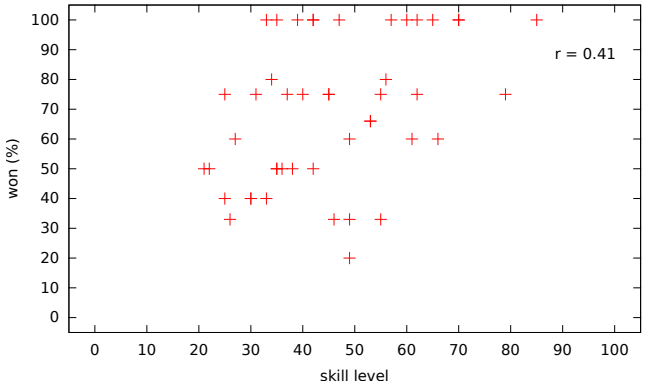
The data obtained from the games played by humans versus the adaptive agent is described in Section 3.2.4.

3.2.3 ADAPTIVEMINIMAX

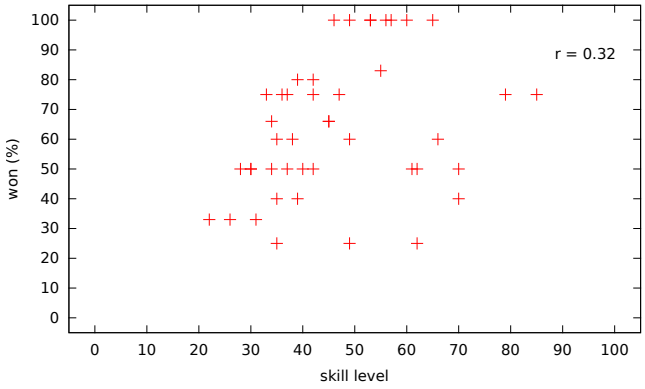
The basic idea behind the proposed adaptive agent is that, given a way to rank the player's and the agent's actions, throughout the game the agent should choose actions that have the same rank as the ones chosen by the player in the previous turns. In the implementation of the adaptive ranking agent for connect four, we use the MINIMAX algorithm to establish the ranking and call our adaptive agent ADAPTIVEMINIMAX.



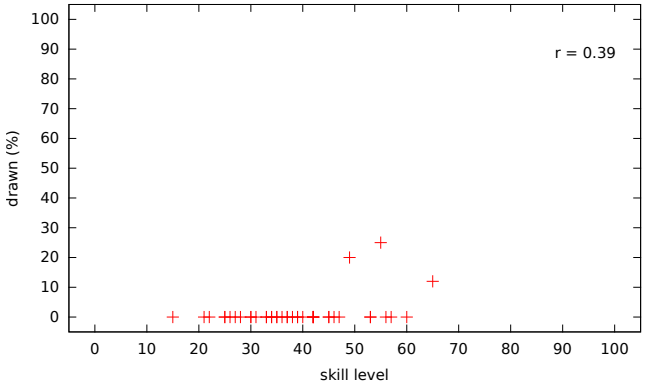
(a) Humans vs NAIVE. The correlation coefficient = 0.54.



(b) Humans vs SIMPLE. The correlation coefficient = 0.41.



(c) Humans vs MINIMAX. The correlation coefficient = 0.32.



(d) Humans vs OPTIMAL. The correlation coefficient = 0.39.

Figure 8: The plots of human players' winning rates (Y) against their skill levels (X) in the games versus the non-adaptive algorithms presented in the playground.

Recall that, given a complete game tree, MiniMax assigns the following evaluations to all the vertices in the tree: $+1$ to the winning states (and, correspondingly, to the moves that lead to them), -1 to the losing ones, and 0 to the ones resulting in a draw. For the purpose of ranking the moves, this gradation is not detailed enough. Consider the situation, where the agent has three available moves (m_1, m_2, m_3) with evaluation scores being $(0, -1, -1)$. That would result in move m_1 having the rank of 1, while moves m_2 and m_3 will get a rank drawn uniformly at random without replacement from $\{2, 3\}$.

While both moves are losing ones, their quality could differ. Imagine that m_2 forces MiniMax to lose in one turn, but m_3 results in the game going on for 20 turns more. The first option corresponds to a much weaker opponent. Also, assuming the player does not follow an optimal strategy, m_3 allows for other options to be discovered further into the game, possibly even changing its outcome.

To achieve a more detailed ranking we modified the evaluation procedure of the game tree leaves in such way that it also reflects the length of the path leading from the root of the tree to the leaf. To implement this idea we replaced $+1$ and -1 with sufficiently large numbers $+L$ and $-L$. Sufficiently large in this context means that the numbers should allow the modification of the winning/losing scores without losing their signature properties: They should be easily distinguishable from scores returned by a heuristic evaluation function applied to the pruned leaves of the MiniMax subtree. Positive numbers should indicate the winning path of MAX player, while negative numbers should indicate the winning path of MIN player. To include the information about the length of the winning (or losing) path on every tree level along the winning path of MAX player, 1 was subtracted from the previous score. Similarly, 1 was added to the previous score on every level along the losing path of MAX player. Hence, the move with the evaluation score of $L - 5$ means that there exists a winning path for MAX player starting from this move and it is six moves long, while the move with the evaluation score of $L - 1$ means that MAX player can win in two moves if she chooses this one.

We refined the gradation of available moves further using the heuristic evaluation function of MiniMax (Section 3.2.1) for the pruned leaves of the MiniMax's subtree. Taking into account that the chosen heuristic provides the scores in the range $[-20, 20]$ and that the maximum height of connect four game tree is 42, L was set to 100. This choice satisfies the properties mentioned earlier: Even the modified winning and losing scores cannot overlap with the range of the scores provided by the heuristic. These scores allow us to impose a ranking on a set of available moves in every game state. As mentioned above, for the moves with the same evaluation scores the corresponding ranking scores are drawn uniformly at random without replacement from the respective range.

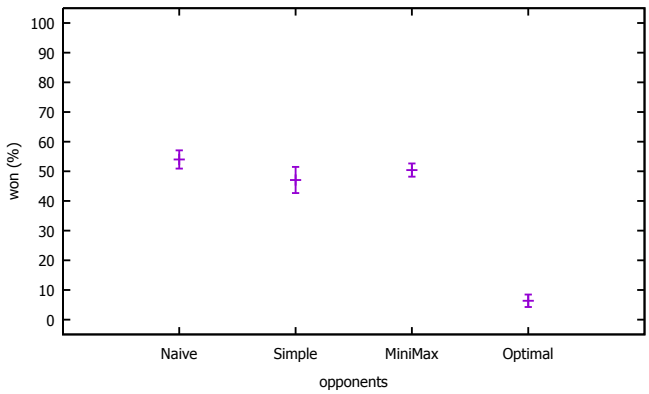


Figure 9: The mean and the standard deviation of the percentage of games won by AdaptiveMiniMax against all the non-adaptive algorithms in the testing ground (drawn in the case of Optimal). The statistics are calculated over 10 episodes of 1000 games each.

Before evaluating the available moves and making a decision, AdaptiveMiniMax evaluates the moves that were available to its opponent and calculates a ranking score for each. To decide which move to take, AdaptiveMiniMax calculates the average ranking score of the move made by the opponent: $r = \Sigma_{i=1}^{t} r(m_i)/t$, where t is the amount of moves made by the opponent so far, and m_i is the move made by the opponent at turn i. After evaluating and ranking its own available moves AdaptiveMiniMax chooses not the optimal move, but the one with the ranking score closest to the average ranking score of the opponent, r.

3.2.4 *Experimental results*

Recall that Naive, Simple, MiniMax, and Optimal algorithms have distinctly different skill levels when playing connect four. In Figure 9 we show the performance of AdaptiveMiniMax versus all of them. Against an opponent of a weaker or an equal skill, AdaptiveMiniMax adapts well, winning approximately half of the games. The data collected from the games of AdaptiveMiniMax versus Optimal clearly shows the agent's disadvantage. Since it is built on top of MiniMax, it can play only as good as MiniMax itself. Therefore, it cannot adapt to any opponent that is stronger than MiniMax.

One of the results of the experiments with human players is that clearly participants preferred to play against an adaptive opponent. When they chose one of the others they usually found very fast (in less than 5 games on average) how relatively easy or difficult this specific opponent was to beat and we can say "got bored" with it and moved onto another one. Whereas against an adaptive opponent they spent much more time (on average $14 - 15$ games). We can speculate

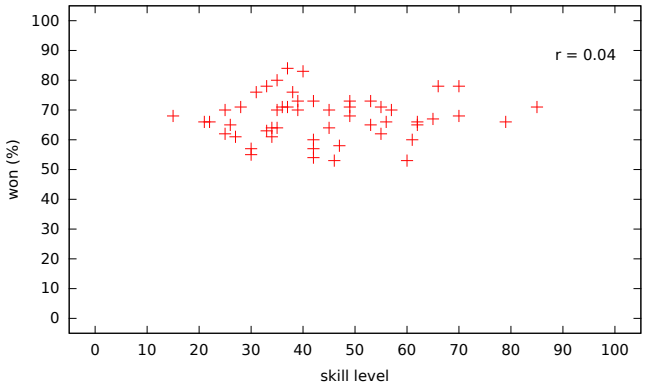


Figure 10: Players' skill levels (X) against their winning rate (Y) versus AdaptiveMiniMax. The correlation coefficient = 0.04

that it was influenced by the fact that this opponent is adaptive and at least to some extent is playing at the same level as the human players.

The plot of players skill levels (Eq. 2) against their winning rates in the games with AdaptiveMiniMax is presented on Figure 10 (cf. Figure 8). We can see that independently of the players' skill levels all the plotted points lie in a region between 50% and 80%. It is also interesting to note that the weakest and the strongest players have approximately the same winning rate, 70%. Furthermore, the correlation coefficient of 0.04 tells us that there is no substantial correlation between the players' skill levels calculated according to Equation (2) and their winning rate against AdaptiveMiniMax. We conclude that in the experiments with human players AdaptiveMiniMax shows good adaptive qualities.

Comparing Figures 8c and 10 we see that even though AdaptiveMiniMax is built on top of MiniMax algorithm and supposedly it can play only as strong as MiniMax, players who displayed perfect performance when playing against MiniMax, could not repeat their success when playing against AdaptiveMiniMax. In fact, AdaptiveMiniMax's evaluations are different from MiniMax's ones, because they include information about the length of the strategies into resulting moves' scores. It is possible that as the result AdaptiveMiniMax can play stronger than MiniMax. Another possible reason for the different experimental data is the fact that all the players played very few games against MiniMax (no more than six, and the strongest players no more than four). If they played more games, the results could have been different.

We conclude that, at least for connect four, the principle of adaptive ranking together with the hand-made evaluation function results in the agent that successfully adapts both to computer opponents and to human players of various skill levels. In the next section we will see that for another well known board game this is not the case.

3.3 CHECKERS

Checkers or English Draughts is a well known board game, in terms of popularity among human players as well as in terms of academic research. It is played by two players on a fixed size chequerboard (8x8 for the English Draughts variant) using two sets of pieces, dark and light (Figure 11). Players make their moves in turns by moving one of their pieces in a prescribed way: The pieces can either move to a free adjacent diagonal square, or jump diagonally over an opponent's piece, if the square to land on is unoccupied. For a comprehensive description of the game we refer the reader to the corresponding Wikipedia article[1].

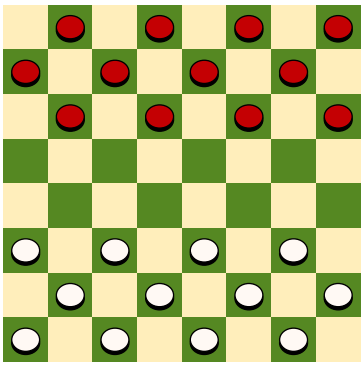


Figure 11: Starting position in checkers.

From a computer science point of view the game has been studied for almost 60 years. The research has been mostly focused on creating strong computer opponents. This direction has been exhausted as checkers were solved by Schaeffer et al. [74]: Assuming perfect play from both sides the game always ends in a draw.

Just as for connect four, we provide the adaptive ranking agent with computer opponents of various skill levels, which we describe in Section 3.3.1. Compared to connect four, checkers is more complex in various ways, in particular, it has a higher average branching factor, which makes comparing the moves' rankings more complex as well. We talk about the differences in Section 3.3.2, then in Section 3.3.3 we suggest several variants of the AdaptiveMiniMax algorithm tailored to address the problems coming from the higher average branching factor. In the same section we present the experimental results.

3.3.1 *Computer opponents*

Jonathan Schaeffer and his colleagues created an optimal checkers algorithm, Chinook [73]. In contrast to the optimal algorithm for connect four that exploits the structural properties of the game, Chinook includes among other things an algorithm to perform the deep search of the game tree and a linear hand crafted evaluation function that considers several features of the game board. The endgame database of Chinook is publicly available, but since AdaptiveRanker requires ranking of all available moves, rather than simply the optimal one, we have decided not to use it.

1 http://en.wikipedia.org/wiki/Draughts

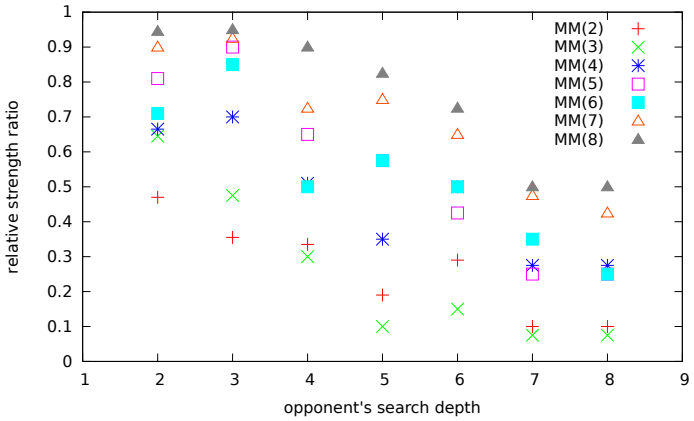


Figure 12: The relative strength ratio of the computer opponents with the search depth ranging from 2 to 8, when playing against each other.

Instead we have created a set of computer opponents based on the open source checkers project called raven-checkers[2] that uses MiniMax algorithm and Martin Fierz's Simple checkers evaluation function[3]. To regulate the strength of the opponents we vary the search depth of MiniMax from 2 to 8.

To illustrate the skill levels of computer opponents produced this way, we show the plots of relative strength ratio (Eq. 1), where i and j encode the search depths of the corresponding pair of opponents in Figure 12.

3.3.2 *Differences between checkers and connect four*

Both checkers and connect four are zero-sum games, but checkers have higher complexity than connect four: The average depth of a game tree in checkers is 70 versus 36 in connect four. The branching factor (the amount of available moves) varies from 1 to 17, the average branching factor is 8 versus 4 in connect four. The estimate of the game tree complexity, i. e. how many nodes the MiniMax algorithm would need to traverse to evaluate the root, is 10^{31} [74] versus 10^{21} in connect four [2].

The difference most important for the adaptive ranking agent is the large variance of the branching factor. If the opponent had a choice of two moves, while the agent is faced with nine or ten or vice versa, it is to be expected that their rankings are not directly comparable.

From the highly variable branching factor follows another difference, the sensitivity towards the quality of the move. The position in the ranking does not represent the quality of the move in the context of winning the game: The third best move for one of the players could

2 https://code.google.com/p/raven-checkers/
3 http://www.fierz.ch/engines.php

be still a good move according to the evaluation function, while the third best move for the other could force it to lose the game.

To address these difficulties we explore below different variants of using the ranking on the moves to produce the next move for the adaptive ranking agent.

3.3.3 AdaptiveMiniMax

The first AdaptiveMiniMax variant we have implemented uses the basic idea of picking the move with the same rank as the one that the opponent has made with the following modification: If the move that AdaptiveMiniMax is supposed to take has a negative evaluation score, AdaptiveMiniMax climbs up the ranking until it finds the first move with a non-negative evaluation score and uses this one. In the case there was no move with the given rank (which happens when the opponent had more moves available than AdaptiveMiniMax), AdaptiveMiniMax uses the lowest move in the ranking with a non-negative evaluation score. In this way we ensure that no obviously bad moves are made by AdaptiveMiniMax regardless of the opponent's choices.

The second AdaptiveMiniMax variant deals with the problem of the varying branching factor by mapping the absolute ranking of opponent's moves and of its own moves to the $[0, 1]$ interval and picking a move that has the new relative rank on $[0, 1]$ closest to that of the opponent's move. It also applies the same correction for negative evaluation scores as above.

The third AdaptiveMiniMax variant ignores the ranking altogether and instead chooses a move that has the evaluation score closest to that of the opponent's move in terms of Euclidean distance. In this variant AMM does not eliminate moves with the negative evaluation scores.

All three variants take the history into account by using the decayed average of the rank (resp. the evaluation score) instead of their current values. Let v_i be the rank or the evaluation score of the opponent's move at turn i, $t+1$ a total amount of opponent's moves so far, $\delta \in [0, 1]$ the decay factor. The decayed average rank (resp. evaluation score) v at turn $t+1$ is $v = \Sigma_{i=0}^{t} \delta^{t-i} v_i / (t+1)$.

As was the case with connect four, all three variants of AdaptiveMiniMax are built on top of the MiniMax algorithm. Their search depth was set to 7, while their opponents were instances of MiniMax with the search depth running from 2 to 8. The results of these games are presented in Figure 13.

The first thing to note is that as the opponent's search depth becomes larger than 7, the performance of all three variants start to deteriorate. Furthermore, as expected, due to the varying branching factor and the sensitivity of the game's outcome to the quality of

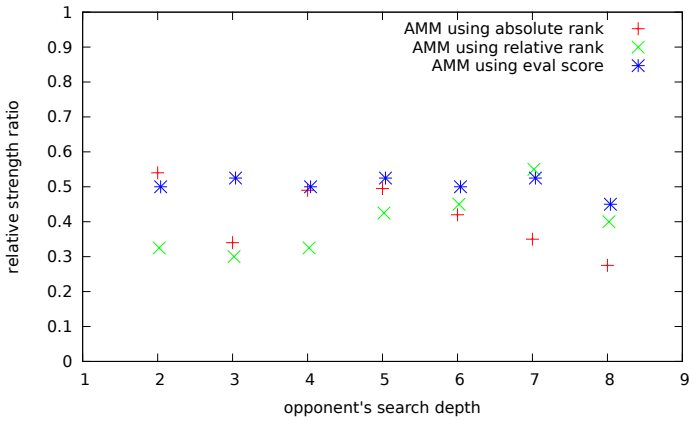


Figure 13: The relative strength ratio of the AdaptiveMiniMax variants when playing checkers against MiniMax with the search depth ranging from 2 to 8.

moves, the AdaptiveMiniMax variant using the absolute rank performs the worst both in terms of adapting to its opponents and in terms of its own performance. Using the relative rank appears to improve the adaptive qualities of AdaptiveMiniMax somewhat, but the real winner in this setting is the variant choosing the moves that have their evaluation score closest to that of the opponent's moves.

3.4 CONCLUSION

In this chapter we described an approach to develop an online adaptive agent for games where: (i) the players' goals are mutually exclusive, and (ii) there is a ranking on players' moves available that represents their respective merit with regard to achieving one's goals.

By "adaptive" we mean that the agent is able to modify its playing strategy depending on the abilities of its opponent, and "online" means that the agent's reasoning is based on the events happening in the *current* game only.

The main idea behind the adaptive ranking agent was to establish a ranking on the moves available to the agent and to its opponent and choose the moves with the ranking scores 'close' to the ranking scores of the opponent's moves (see Algorithm 1). The way the ranking is constructed and how the 'closeness' of ranking scores is defined has to be decided based on the particular game's properties. The quality of the ranking bounds the quality of the agent: if it consistently misplaces moves, the agent would lose its adaptive properties against the opponents who are better at estimating the moves' strength.

As a proof of concept we have implemented the adaptive ranking agent based on MiniMax algorithm (AdaptiveMiniMax) for games of connect four and checkers. In connect four AdaptiveMiniMax showed good performance when playing against the computer op-

ponents, adapting well to their respective skill levels, with exception of OPTIMAL, to which it was losing steadily. In the experiments with the human players, data confirming the adaptive qualities of ADAPTIVEMINIMAX was obtained. There is hardly any correlation between the skill level of a specific player and the percent of games this player won against ADAPTIVEMINIMAX. From the same data it seems that for the majority of players ADAPTIVEMINIMAX choose a strategy that was weaker than the corresponding player's skill level, i. e. the percentage of the games won by human players is mostly greater than 50%.

Checkers is an example of a board game where opponents' situations on consecutive turns often are not similar. Thus, this game illustrates the limitation of ADAPTIVERANKER: If the rankings are not comparable, it does not adapt well. In this case we could still create an adaptive agent by choosing moves with similar evaluation scores, but the success of this approach also depends on certain similarity of opponents' situations. Hence, ADAPTIVERANKER does not fulfill the first of our requirements for a dynamic difficulty adjustment algorithm, universality. We move on to explore how supervised machine learning can be used to create a universally applicable dynamic difficulty adjustment.

4

DIFFICULTY MODELLING

4.1 INTRODUCTION

The game development process usually includes multiple testing stages, where a multitude of players is requested to play the game to provide data and feedback. This data is analysed to tweak the game parameters in an attempt to provide a fair challenge for as many players as possible. The question we investigate in this chapter is how a dynamic difficulty adjustment mechanism can be created using the data from the alpha and beta testing together with supervised learning.

Our aim is to devise a difficulty adjustment algorithm that does not bother the actual players. For that, we assume there is a phase of the game development in which the game is played by testers and the difficulty is manually adjusted to be just right by each tester individually. In the background the data that represents the game states and players' decisions is collected. From this data, we induce a difficulty model and build it into the game. The players do not notice any of this and are always challenged at the difficulty that is estimated to be just right for them.

We present two different approaches to implement such a system: *static* (Section 4.3) and *dynamic* (Section 4.6). The names reflect how the prediction is made: The static algorithm predicts the difficulty setting for a new player exactly once, while the dynamic algorithm predicts the difficulty adjustment at every time step. Before going into the details we also describe in Section 4.2 a simple game that was used to simulate the data collection phase and to evaluate both algorithms.

The static algorithm consists of two phases. First, it uses all the data collected to build several difficulty models by clustering the recorded game traces and averaging the supervision over each cluster. After that it learns to predict the correct model from a short period of gameplay. In Section 4.4 we show how this algorithm performs using the data collected from human players. We compare the resulting models to less sophisticated, yet realistic, baselines. In particular, we look at the performance of changing with time versus constant difficulty as well as the performance of cluster prediction versus no-cluster. While the results are encouraging (i. e. non-constant adjustment and cluster prediction together outperform the alternatives significantly), it is important to note that the success depends on several open questions (e. g. how does the performance depend on the prefix/suffix

partition?) as well as the choice of values for several parameters (see Section 4.5 for a discussion).

Seeing that a game trace is nothing else than a time sequence leads us to the dynamic algorithm (Section 4.6). The question of how to adjust the difficulty at the next time step turns into a time series prediction problem. The algorithm learns to predict the next difficulty value from a very short period of gameplay using the support vector regression with a Gaussian kernel. We evaluate its performance using the same setup as for the static algorithm and present the results in Section 4.7.

4.2 EXPERIMENTAL SETUP

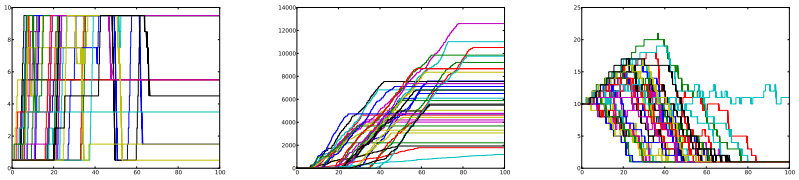
To test both algorithms we implemented a simple game, Aliens, using the Microsoft XNA framework[1] and one of the tutorials from the XNA Creators Club community, namely "Beginner's Guide to 2D Games"[2]. The player controls a cannon that can shoot cannonballs. The gameplay consists of shooting down the alien spaceships while they are shooting at the cannon.

A total of five spaceships can be simultaneously on the screen. They appear on the right side of the game screen and move on a constant height from the right to the left. The spaceships are generated so that they have a random speed within a specific interval from a given average speed. Whenever one of the spaceships is shot down or leaves the game screen, a new one is generated. At the beginning of the game the player's cannon has a certain amount of hit points, which is reduced by one every time the cannon is hit. At random time points a repair kit appears on the top of the screen, floats down, and disappears again after a few seconds. If the player manages to hit the repair kit, the cannon's hit points are increased by one. The game is over if the hit points are reduced to zero or a given time limit of 100 seconds is up.

Figure 14: Screenshot of Aliens.

Additionally to the controls that allow the player to rotate the cannon and to shoot, there are also two buttons by pressing which the player can increase or decrease the difficulty at any point in the game. Behind the scenes the difficulty is controlled by the average speed of the alien ships. For every destroyed spaceship the player receives a certain amount of score points, which increases quadratically with

1 http://msdn.microsoft.com/en-us/xna/default.aspx
2 http://creators.xna.com/en-GB/



(a) Difficulty level. (b) Score. (c) Health.

Figure 15: Game traces from one player. Different colours represent different game traces. A lot of the game traces end prematurely (i. e. the health goes to zero and the game ends) and are filled up with the last recorded feature values. Since the data is logged when at least one of the feature values changes, every trace is a step-wise constant function.

the difficulty level. During each game all the information concerning the game state (e. g. the amount of hit points, the positions of the aliens, the buttons pressed, etc.) is logged together with a timestamp. For the empirical evaluation we held one gaming session with 17 participants and collected the data on how the players behave in the game with the total of 232 game traces.

Out of all logged features we restricted our attention to the three: the difficulty level, the score, and the health, as they seem to represent the most important aspects of the player's state. The game traces of one randomly chosen player are shown in Figure 15. There are a few things to note about them. First, we can see that one player exhibits various behaviours, to the point that it is difficult to discern any clear patterns. Second, out of all game traces of this player, only one reaches the time limit of 100 seconds. (It is the cyan trace that has the difficulty of approximately 3.5, the score slowly increasing to about 1100 points, and the health points staying close to 11.) Comparing this trace with the others, we conclude that our player does not settle for the difficulty that allows her to get through the game. Rather, she actively engages in the process of tuning the difficulty to give her the experience she wants. It is, in fact, the typical behaviour for our participants.

4.3 STATIC ALGORITHM

Recall, that in this chapter we focus on the question of how a game can adapt to a particular playing type given two sources of information: the data collected from the alpha/beta-testing stages (offline phase), as well as the data collected from the new player (online phase). By playing type we mean a certain pattern in behaviour with regard to challenges. To simplify the problem of difficulty adjustment it is the common practice to discretise the space of all possible playing types. The simplest such discretisation is into beginners, averagely

skilled players, and experts (corresponding to easy, average, and hard settings). In our experiments we do not predefine the types, rather, we infer them by clustering the data collected in the offline phase. After that, we learn a mapping from a set of game traces into the set of difficulty adjustments that correspond to the types. In the online phase, given a new player, the game needs only to determine which type she belongs to and then apply the learned model.

Let $\mathcal{T}$ be a set of collected game traces. The algorithm's outline consists of the following steps:

1. Cluster the set $\mathcal{T}$.
2. For each cluster construct a difficulty adjustment model.
3. Given a new player, decide on which cluster she belongs to and predict the difficulty adjustment using the corresponding model.

Note that it is desirable to adapt to the new player as quickly as possible. To this purpose we propose to split each game trace T into two parts:

- a prefix, T^{pre}, the relatively short beginning that is used for the training of the predictor in the offline phase;
- a suffix, T^{post}, the rest of the trace that is used for constructing the difficulty adjustment model.

In our experiments we used the K-means algorithm [34] for the clustering step and an SVM with a Gaussian kernel function [13] for the prediction step of the algorithm outlined above. In the following we describe the static algorithm in detail.

4.3.1 *Preprocessing*

As mentioned above, the game traces are split into two parts, $T = (T^{pre}, T^{post})$, such that T^{pre} contains the first 30 seconds of the trace T, and T^{post} contains everything after the first 30 seconds from the trace T. The *pre* parts of the traces are used for training the predictor. The *post* parts of the traces are used for constructing difficulty models.

First thing to note about the collected data, is that even though the game ends after the 100 seconds, a lot of the game traces are shorter than that because the cannon's hit points drop to zero. To equalize the duration of all collected data, we fill up the traces that lasted less than the predefined time with the last achieved features' values, making them all to last for 100 seconds.

Next thing to consider is that to reduce the computational load the data is logged only when the game's or the player's state changes (in the case of a simple game used in our experiments it may seem a

trivial concern, but this is important to keep in mind for more complex games). As a result for two different game traces their time lines are different. One way to produce identical time lines for each game trace is to sample at the regular intervals the piece-constant functions that the collected data represent (see Figure 15). Note that in this case every game trace becomes an instance in a (number of game features) $\times$ (number of time steps)-dimensional space.

To reduce the dimensionality of the instance space, we construct a set of polynomial fits for each game trace prefix and suffix, a polynomial for each game feature, such that they minimize the root-mean-square error (RMSE). In the following T refers to a single game prefix or suffix, $N = |T|$. Let u_i be the value of the feature u at time t_i, $u_i = u(t_i)$. Let P_u be a polynomial of degree d, $P_u(x) = \sum_{k=0}^{d} a_{u_k} x^k$. We want to find $a_{u_0}, \ldots, a_{u_d} \in \mathbb{R}$ such that the sum over i of the squared differences between $P_u(t_i)$ and u_i is minimal. Formally,

$$(a_{u_0}, \ldots, a_{u_d}) = \underset{(a_0,\ldots,a_d)\in\mathbb{R}^{d+1}}{\operatorname{argmin}} \frac{1}{2} \sum_{i=1}^{N} \left(\sum_{k=0}^{d} a_k t_i^k - u_i \right)^2 .$$

Hence, we can find $a_{u_0}, \ldots, a_{u_d}$ by solving a system of linear equations:

$$\frac{\partial}{\partial a_{u_j}} \frac{1}{2} \sum_{i=1}^{N} \left(\sum_{k=0}^{d} a_{u_k} t_i^k - u_i \right)^2 = \sum_{i=1}^{N} \sum_{k=0}^{d} \left(a_{u_k} t_i^k - u_i \right) t_i^j = 0,$$

where $j = 0, \ldots, d$.

To choose an appropriate polynomial degree, we have calculated the average RMSE over the available data for degrees ranging from 1 up to 19. Note, we are constructing a descriptive as opposed to a predictive model. Hence, we are not concerned about overfitting and testing the predictive accuracy on the unseen data. Rather, we want to minimize the RMSE on the data present. The resulting average values of the error for every feature are shown on Figure 16. Note that the difficulty level appears to be the most difficult to fit. Based on these results we have decided in the following to use $d = 7$.

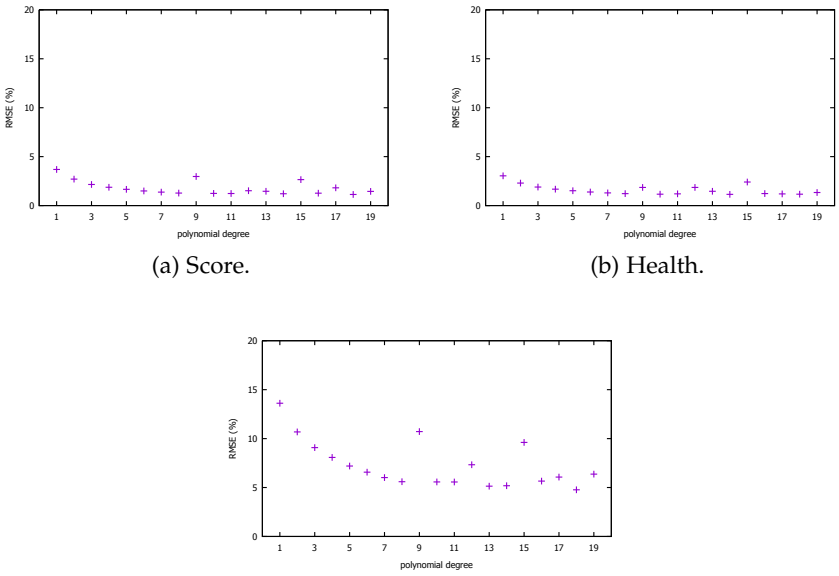
After constructing the fit, all game trace prefixes and suffixes are replaced with their corresponding sets of the polynomials :

$$\widehat{T} = \{P_u, P_v, \ldots\} .$$

To de-clutter the notation in the following we write T instead of $\widehat{T}$ meaning the representation of game traces, as well as their prefixes and suffixes, by the corresponding polynomials.

4.3.2 *Clustering*

The next step is based on the assumption that there is a finite, small number of playing types. To discover them we cluster the game traces,



(a) Score.

(b) Health.

(c) Difficulty level.

Figure 16: RMSE of polynomial fits for different features. The error is displayed in percentages of the corresponding feature's range.

specifically, the polynomials corresponding to the suffixes of the difficulty feature. For this purpose we use K-means clustering [34]. The idea of the algorithm is to minimize the intra-cluster distances over all clusters. To achieve that, the algorithm takes as a parameter the amount of clusters and the initial choice of cluster centroids (in our implementation they are drawn uniformly at random from $\mathbb{R}^{d+1}$). Then it iteratively (i) assigns data instances to the nearest centroid, and (ii) recalculates the centroids. The algorithm stops when the clusters don't change any more.

We represent each playing type by the index of the cluster. The indices will be used in the classification step to train the SVM. Furthermore, the instances belonging to each cluster will be used together to produce various difficulty models and to evaluate the overall performance of the static algorithm.

For illustrative purposes, in Figure 17 we show the result of clustering the game traces from Figure 15 using 3 clusters. (In this case, to make the comparison with the original traces easier, the whole game traces are clustered using their difficulty feature, not only their suffixes.) We can see that what appeared chaotic in Figure 15, in fact, demonstrates two clear patterns: one with the difficulty increasing steeply right from the start and staying at the maximum (the top cluster), and the other with the difficulty starting to increase somewhat later during the game, not quite as steep as in the first cluster, and also staying at the maximum once it reached it (the centre cluster). The bottom cluster contains the traces that are sufficiently remote from both of these patterns. This demonstrates that while a player

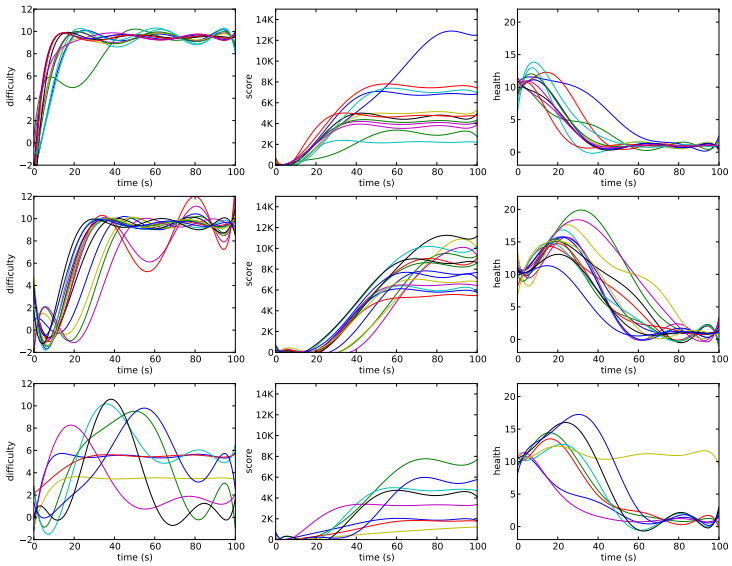


Figure 17: Game traces from Figure 15 after the interpolation and clustering. We can see two playing types: (i) in the top cluster increase the difficulty as much as possible, reach moderate score values, and die quickly; (ii) in the centre cluster wait somewhat before increasing the difficulty, reach higher score values, increase health points (and die not so quickly). The bottom cluster contains outliers.

can exhibit a seemingly indiscernible variety of behaviours, it is a reasonable idea to cluster her game traces to find patterns, what we call playing types. Note also, that even though we use only the difficulty feature for clustering, the other two features follow the resulting clusters. If that was not the case, we would advise using a combination of features for clustering.

4.3.3 *Classification*

In the classification step we want to obtain a mapping that, given a prefix of a game trace, returns the corresponding cluster index. Recall that in a supervised learning problem there is a training set $X \subseteq \mathcal{X}^m$, a set of corresponding labels $Y \subseteq \mathcal{Y}^m$, and an unknown underlying process mapping instances to labels, $g : \mathcal{X} \to \mathcal{Y}$. The learner's task is to find a function $f : \mathcal{X} \to \mathcal{Y}$ that approximates g *good enough*, that is that minimises some measure of error.

The measure of error usually employed is the *empirical risk*,

$$R_{emp}[f] = \frac{1}{m} \sum_{i=1}^{m} l\left(x_i, y_i, f(x_i)\right) \ ,$$

where m is the size of the training set, $x_i \in X$, $y_i \in Y$, $l : \mathcal{X} \times \mathcal{Y} \times \mathcal{Y} \to [0, \infty)$ is a loss function, such that $l(x, y, y) = 0$ for all $x \in X, y \in Y$. In the case when the class of possible solutions f is sufficiently limited we can hope to approximate the minimizer of the expected error over all possible training patterns (the *expected risk*) by minimizing the empirical risk [94, 95].

Our learner of choice is a support vector machine (SVM) [13]. To solve a binary classification problem, that is $\mathcal{Y} = \{\pm 1\}$, it relies on two ideas:

- It utilises the *kernel trick* [75], that is it uses a *kernel function* to map the original instances to a high-dimensional feature space, and finds a hyperplane that separates two classes.
- To achieve the higher stability against instance noise, it maximises the decision gap between the training instances belonging to different classes.

For a detailed information on SVMs and their use for classification problems, we refer the reader to an excellent book by Cristianini and Shawe-Taylor [14].

Note that, since the cluster indices run from 1 to k, we have a multi-class classification problem. We solve it by using one-versus-all strategy [21]: We train k different SVMs such that the i-th SVM gets as an input a set of prefixes with their labels set to 1 if their corresponding suffixes belong to cluster i, otherwise their labels are set to -1. Let m again be the size of the training set, and function $\mathrm{ind}\,()$ return a cluster index for a given game trace prefix, $\mathrm{ind}\,(T_j^{pre}) \in \{1, \ldots, k\}$,

$$
\begin{aligned}
X_i &= \left\{ T_j^{pre} | j = 1, \ldots, m \right\}, \\
Y_i &= \left\{ y_{i_j} | j = 1, \ldots, m \right\}, \\
y_{i_j} &= \begin{cases} 1, & \text{if } \mathrm{ind}\left(T_j^{pre}\right) = i \\ -1, & \text{otherwise.} \end{cases}
\end{aligned}
$$

Instead of a class label, each SVM outputs a real-valued confidence value. Given a new prefix and k trained SVMs, the predicted cluster index is the index of the SVM with the largest confidence value. Each SVM uses a Gaussian kernel [76].

4.3.4 *Difficulty models*

After the preprocessing and the training, for each new prefix T^{pre} the classifier predicts i, the index of the cluster that the corresponding suffix T^{post} belongs to. Let cluster i contain traces $T_1, \ldots, T_m$. Let v_j be the feature corresponding to the difficulty adjustment in T_j, and $v_{j_l} = v_j(t_l)$. The next step in the algorithm is to use the information stored in $T_1, \ldots, T_m$ to predict the difficulty adjustment curve for the trace T. To this purpose we consider the following approaches:

1. The constant model. Given the cluster, this function averages over all instances in the cluster and additionally over the time, resulting in a constant difficulty adjustment.

$$y_{\text{const}} = \frac{1}{Nm} \sum_{j=1}^{m} \sum_{l=1}^{N} v_{j_l} \ .$$

2. The mean model. Given the cluster, this function for each time point returns the average over all instances in the cluster.

$$y_{\text{mean}}(t) = \frac{1}{m} \sum_{j=1}^{m} v_j(t) \ .$$

For the next model, given the cluster, we train the regularised least squares regression (RLSR) [69] with the Gaussian kernel on its instances. As the classification, the regression is also a supervised learning problem. In contrast with the classification problem, here $\mathcal{Y} \subseteq \mathbb{R}$.

RLSR minimizes the mean squared error between the given labels and the predicted values, while controlling the complexity of the predictor by simultaneously minimizing its norm:

$$\min_f \frac{1}{2} \sum_{i=1}^{m} (f(x_i) - y_i)^2 + \frac{\lambda}{2} \|f^2\|_K \ ,$$

where $x_i \in X$, $y_i \in Y$, and λ is the regularization parameter. By applying the Representer Theorem [77] and properties of kernel functions the minimizer of the problem above can be found as a solution of a system of linear equations:

$$\boldsymbol{c} = (K + \lambda I)^{-1} \boldsymbol{y} \ ,$$

$$f(x') = \sum_{i=1}^{m} c_i k(x_i, x') \ ,$$

where $\boldsymbol{c} \in \mathbb{R}^m$, $\boldsymbol{y} = (y_1, \ldots, y_m)^T$, $x' \in \mathcal{X}$, $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ is a kernel function, $K = (k(x_i, x_j))_{i,j=1}^{m}$ is the kernel matrix calculated on the training set. Hence, the third difficulty model that we consider is:

3. The regression model. It maps the prefixes of cluster i to the suffixes of cluster i. Let cluster i contain game traces $T_1, \ldots, T_m$. Recall that all game trace prefixes and suffixes are represented by sets of polynomials of degree d, each polynomial, in turn, by a vector of its coefficients. The training instances are the prefixes of the game traces in cluster i, $X = \{T_j^{pre} | j = 1, \ldots, m\}$. Their labels are the polynomials corresponding to the difficulty adjustment feature in their suffixes, $Y = \{P_{v_j^{post}} | j = 1, \ldots, m\}$. Note, in contrast to the RSLR formulation above, each label is not a scalar, but a vector, $y_j \in \mathbb{R}^{d+1}$. Hence, both $\boldsymbol{y}$ and $\boldsymbol{c}$

are matrices with m rows and $d+1$ columns. Taking into account that we have three game features, the regression predictor f maps $\mathbb{R}^{3(d+1)}$ to $\mathbb{R}^{d+1}$. Given a new instance $x' = T^{pre}$, $f(x') = \sum_{l=1}^{m} c_l k(T_l^{pre}, x') = (a_0, \ldots, a_d)^T$, and the model's output is

$$y_{regr}(x', t) = \sum_{k=0}^{d} a_k t^k \,.$$

4.4 EVALUATION

To evaluate the performance of a predictor the standard approach is to use cross-validation [17] in one of its forms. Cross-validation is a technique to estimate the accuracy of a predictor on unseen data. The general idea is to consider various splits of the training data into a train and a test set, where the predictor is trained on the train set and its predictive error is calculated on the test set. The error averaged over all splits is an estimate of the predictive error on the unseen data.

To evaluate the performance of the static algorithm we conduct a cross-validation on the data that is similar to the standard "leave one out" approach. For each player presented we construct a following train/test split:

- training set consists of the game traces played by all players except this one;
- test set consists of all the game traces played by this player.

Constructing the train and test sets in this way models a real-life situation of adjusting the game to a previously unseen player.

As performance measures we use two metrics: the classification accuracy, which is the percentage of correctly classified instances in the test set, and the RMSE calculated on the exhibited behaviour in the test instances and the behaviour prescribed by the models of the predicted cluster.

To provide the baselines for the performance evaluation, we construct for each test instance a sequence of "cheating" predictors: The first (best) one chooses a cluster that delivers a minimum error; the second best chooses the cluster with the minimum error among the remaining clusters, and so on. We call these predictors "cheating" because they have access to the test instances' data before they make the prediction. For each "cheating" predictor the error is averaged over all test instances and the error of the SVM predictor is compared to these values. As the result we can see which place in the ranking of the "cheating" predictors the SVM one takes.

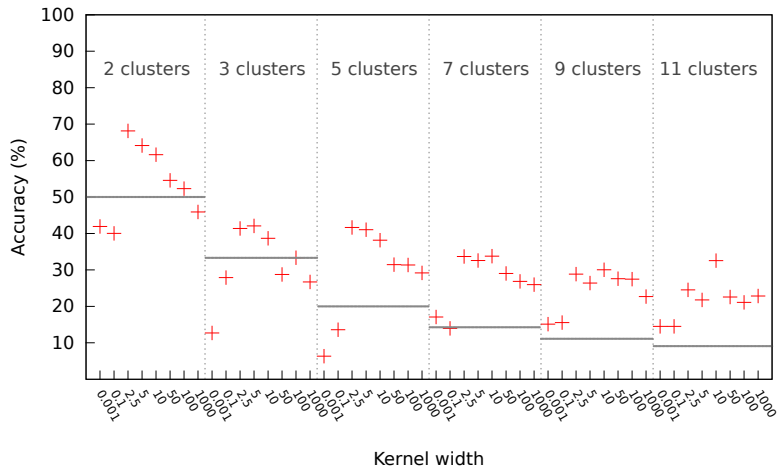


Figure 18: The accuracy of the SVM predictor for various values of the kernel width and for various amounts of clusters. The grey horizontal lines represent a baseline for each amount of clusters: A predictor that assigns each instance to one of the clusters uniformly at random.

4.4.1 *Learning parameters*

The first part of the evaluation concerns the choice of two learning parameters, the Gaussian kernel widths for the SVM predictor and for the regression model.

Figure 18 illustrates how the accuracy of the SVM predictor depends on the amount of clusters and the value of its kernel width. For two clusters we see a clear winner, with the kernel width equal to 2.5 the accuracy is close to 70%. As the amount of clusters increases, the winner becomes less clear-cut, but until we reach eleven clusters, the value of 2.5 still appears to be a good choice. As we will see later, we will restrict our evaluation to the amount of clusters ranging from one to ten. Hence, in the following we set the SVM kernel width to 2.5

Next, we investigate the performance of the regression model for different values of the kernel width, while varying the amount of clusters (Figure 19). At the same time we look at the error values of the other two models. As a result we arrive at two conclusions: (i) the best error values of the regression model are achieved for the kernel width equal to one; (ii) the errors of the constant and the mean models are very close to each other. Hence, in the following we set the regression kernel width to one, and consider only the regression and the constant models.

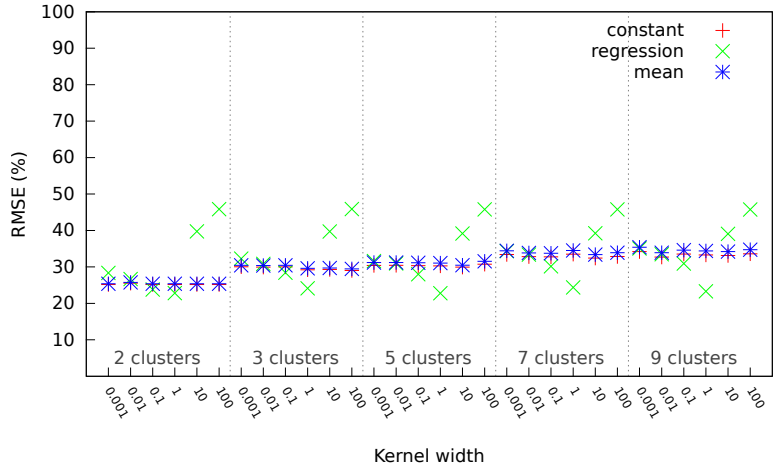


Figure 19: The RMSE of all three models for various values of the kernel width and for various amount of clusters. The regression model performs best with the width set to one. The error values of the other two models are very close to each other.

4.4.2 *Regression versus constant model*

We want to verify the hypothesis that a difficulty function that depends not only on the cluster, but also on the prefix itself (e.g. the regression model) is more appropriate than the one depending on the cluster alone. To this aim we performed proper statistical tests with the null hypothesis that the models perform equally well. As suggested recently [16] we used the Wilcoxon signed ranks test.

The Wilcoxon signed ranks test is a non-parametric test to detect shifts in populations given a number of paired samples. The underlying idea is that under the null hypothesis the distribution of differences between the two populations is symmetric about 0. It proceeds as follows:

1. compute the differences between the pairs,
2. determine the ranking of the absolute differences, and
3. sum over all ranks with positive and negative difference to obtain W_+ and W_-, respectively.

The null hypothesis can be rejected if W_+ (or $\min(W_+, W_-)$, respectively) is located in the tail of the null distribution which has sufficiently small probability.

For settings with a reasonably large number of measurements, the distribution of W_+ and W_- can be approximated sufficiently well by a normal distribution. Unless stated otherwise, we consider the 5% significance level (the corresponding critical value t_0 is 1.78).

To eliminate all other influences, we considered first and foremost only a single cluster. In this case, as expected, the adjustment pre-

Table 2: t-values for comparison of the constant model versus the regression model.

c	best-const vs best-regr	worst-const vs worst-regr
1	8.46	8.46
2	6.12	9.77
3	5.39	12.64
4	5.26	11.37
5	4.90	12.62
6	4.77	11.05
7	4.80	10.38
8	4.62	6.83
9	4.61	7.20
10	4.63	4.36
11	4.55	0.71
12	4.68	-0.77
13	4.60	-9.16
14	4.50	-5.54
15	4.57	-13.26

dicted by the regression model significantly outperforms the constant setting (the corresponding t-value is 2.67).

We also wanted to compare the performance of regression and constant models for larger numbers of clusters. To again eliminate all other influences, we considered the best and the worst cluster for either strategy. The t-values for these comparisons are displayed in Table 2.

While varying the amount of clusters from one to fifteen we found out that the regression model always significantly outperforms the constant model when choosing the best cluster. The same effect we can observe for the worst cluster, but only while the amount of clusters used is less than eleven. For more clusters the constant model starts to outperform the regression one, probably due to there being an insufficient amount of instances in some clusters to train a good regression model. Based on these results in the following we consider only the regression model and vary the amount of clusters from one to ten.

4.4.3 *Right versus wrong choice of cluster*

As a sanity check, we next compared the performance of the best choice of a cluster versus the worst choice of a cluster. To this end we

found—very much unsurprisingly—that for any non-trivial number of clusters, the best always significantly outperforms the worst.

This means there is indeed room for a learning algorithm to fill. The best we can hope for is that in some settings the performance of the cluster predicted by SVM is close to, i. e., not significantly worse than, the best predictor while always being much, i. e., significantly, better than the worst predictor.

4.4.4 *One versus many types of players*

The last parameter that we need to check before coming to the main part of the evaluation is the number of clusters. It can easily be understood that the quality of the best model improves with the number of clusters while the quality of the worst degrades even further. Indeed, on our data, having more clusters was always significantly better than having just a single cluster for the best predictor using the regression model.

Under the assumption that we do not want to burden the players with choosing their difficulty, this implies that we do need a clever way to automatically choose the type of the player. Adjusting the game just to a single type is not sufficient.

4.4.5 *Quality of predicted clusters*

We are now ready to consider the main evaluation of how well the type of the player can be chosen automatically. As mentioned above the best we can hope for is that in some settings the performance of the predicted cluster is close to the best cluster while always being much better than the worst cluster. Another outcome that could be expected is that performance of the predicted cluster is far from that of the best cluster as well as from the worst cluster.

To illustrate the quality of the SVM predictor we look at its place in the ranking of the "cheating" predictors while varying the amount of clusters. The results of the comparison of the predictors' performance for the regression model are shown in Table 3. Each line in the table corresponds to the amount of clusters specified in the first column. The following columns contain values 'w', 's', and 'b', where 'w' means that the SVM predictor displayed the significantly worse performance than the corresponding "cheating" predictor, 'b' for the significantly better performance, and 's' for the the cases where there was no significant difference. The columns are ordered according to the ranking of the "cheating" predictors, i. e. 1 stands for the best possible predictor, 2 for the second best, and so on.

We can observe a steady trend in the SVM predictor's performance: Even though it is always (apart from the trivial case of one cluster) significantly worse than that of the best possible predictor, it is also

Table 3: Results of the significance tests for the comparison of performance of the SVM predictor and "cheating" predictors using the regression model. Here 'w' stands for significantly worse, 'b' for significantly better, 's' for no significant difference.

	1	2	3	4	5	6	7	8	9	10
1	s									
2	w	b								
3	w	s	b							
4	w	b	b	b						
5	w	s	b	b	b					
6	w	w	b	b	b	b				
7	w	s	b	b	b	b	b			
8	w	s	b	b	b	b	b	b		
9	w	w	s	b	b	b	b	b	b	
10	w	w	s	b	b	b	b	b	b	b

always significantly better than that of the most other predictors. In other words, regardless of the amount of clusters, the SVM predictor always chooses a reasonably good one.

This last investigation confirms our hypothesis that predicting the difficulty-type for each player based on short periods of gameplay is a viable approach to taking the burden of choosing the difficulty from the players.

4.5 DISCUSSION

Above we presented and investigated one possible use of supervised learning for dynamical difficulty adjustment. Our aim was to devise a difficulty adjustment algorithm that does not bother the actual players. Our approach to building a difficulty model consists of clustering different types of players, finding a good difficulty adjustment for each cluster, and predicting the cluster for short traces of gameplay. Our experimental results confirm that dynamic adjustment and cluster prediction together outperform the alternatives significantly.

Note that there are a few aspects that can influence the performance of the algorithm. First, it requires training data, which could be both an advantage and a disadvantage. In this particular case, the data consists of players' game traces: for each time step a record of a game state. It is fairly straightforward to obtain, there is nothing in there that needs new methods or is costly as such, and it is being collected routinely for other purposes. As is usual, the performance of the learning algorithm directly depends on the quality and the amount of training data.

In our experiments, there was a competition element (the high scores information was available to all participants), and since the score points were driven by the difficulty as well as other factors, the players were motivated to increase their difficulty past the point where they were comfortable. The models constructed incorporated this behaviour accordingly. One must be aware that if the data does not represent the behaviour of the majority of players, the learning algorithm will not be able to produce reliable predictions.

Another potential problem concerns our choice of models' construction, namely clustering. The drawbacks of K-means clustering are well known: The number of clusters has to be defined in advance and the algorithm is dependent upon the starting centroid locations.

One further parameter left out in our investigation is the length of the prefix that is used for the prediction. Note that the game traces in the considered test setup were very short (100 seconds). It is plausible that this plays a crucial role in the quality of the model and for the longer games the short prefix of the game trace does not provide enough information to make accurate predictions. Longer game traces can probably be broken into smaller pieces, for which it is established in some way (for instance, with the help of grid search) what length of prefix and suffix works sufficiently well. This idea leads us directly to the next algorithm presented in Section 4.6: Let us use very short pieces of a game trace (for instance, three time points) to predict a single difficulty adjustment for the next time point.

Formulated that way, we have a time series prediction problem. Non-linear techniques for time series prediction are used to model a wide variety of processes in fields like finance [93], [86], telecommunications [22] and autonomous systems [36]. Black box modelling techniques are the usual way to deal with non-linearity [80], among which backpropagation neural networks are a classic [24]. In recent years, however, support vector machine-based approaches [31] have been gaining popularity due to their theoretical guarantees, relative ease of model selection, and good generalization abilities.

In the following section we present an algorithm that learns a mapping from short intervals of game traces to the difficulty adjustments and predicts appropriate ones given a new player and a short trace of in-game observations.

4.6 DYNAMIC ALGORITHM

In Section 4.3 the main assumption is that there exists a finite number of types of players, where by type we mean a certain pattern in behaviour with regard to challenges. Having access to data collected from players in the testing stage allows us to infer these types by clustering rather than define them manually. Furthermore, a function is learned for each type of the player: given a short trace of the start of

the game from a new player this function tells us how the difficulty should be changed in the rest of the game. In this section we investigate how this approach can be modified to make sequential online predictions as the player progresses through the game.

The idea we present is rather simple. Rather, than breaking up a given time trace of a game trace into a prefix and a suffix and learning the mapping from a set of prefixes into a set of suffixes, we propose breaking a time trace into a sequence of rather short intervals. The next step is to learn a mapping between an interval and the difficulty value that succeeds it, so that for any player the game can observe his current "state" and use the learned mapping to predict the difficulty adjustment for the next time step.

A single point in a game trace consists of values of different features that describe the game state at this time point. In our approach we want to find a mapping between the state of the game and the game difficulty. This mapping should model as close as possible the way players adjusted the difficulty in the testing phase. We can regard the process of difficulty adjustment as a non-linear dynamic system with inputs corresponding to the features of the game and the output being the desired difficulty. Let us denote the difficulty setting at time t_i by y_i, and all the other features at time t_i by $\boldsymbol{u}_i$. We could formulate the mapping as

$$y_{n+1} = f(y_n, ..., y_{n-k}, \mathbf{u}_n, ..., \mathbf{u}_{n-k}),$$

where f is an unknown non-linear function.

A critical decision is whether to include the difficulty value itself as a feature of a training instance. On the one hand, including it provides the learning algorithm with important piece of information. On the other hand, in this case a simple predictor returning the same value as was used as a part of the input becomes very hard to beat due to the long constant periods of difficulty in any given time trace. Note that even though this predictor will produce a small error using for example the mean square distance, it will completely miss all the important points, i. e. the ones where the player wanted to change the difficulty.

As a result, we considered our main task to construct a predictor that does not use the difficulty level itself as a part of the input. That means that the variable y_n is not available during training, and the dynamic model appears to be feed-forward [80], i. e. not taking into account any feedback,

$$\hat{y}_{n+1} = \hat{f}(\mathbf{u}_n, ..., \mathbf{u}_{n-k}).$$

Nonetheless, it is likely that the past behaviour of the game difficulty does contain information which is beneficial for the prediction, thus we want to include it in our model in some way.

To address this issue, we train a second stage predictor where we add the output of the first predictor $\hat{y}$ to the list of features with the hope that it is a substitute for the unobservable variable y. In this way we arrive at the prediction equation

$$\bar{y}_{n+1} = \bar{f}(\hat{y}_n, ..., \hat{y}_{n-k}, \mathbf{u}_n, ..., \mathbf{u}_{n-k}).$$

The learning and predicting is done using the support vector regression (SVR) with a Gaussian kernel [20, 75]. The parameters for the SVR were chosen via a grid search and set to

- for the first stage: $\gamma = 0.1, C = 100$;
- for the second stage: $\gamma = 0.1, C = 10$.

4.7 EVALUATION

To test the 'online' algorithm we use the same experimental setup and data as above (Section 4.2). Note that in contrast to above, we use very short intervals of game traces as instances. Thus, there is no reason to use polynomial fits to represent the data. To overcome the irregularity of the time steps we construct a stepwise-constant fit for each trace and then sample the fit every 0.25 of a second to produce the time sequences with identical time steps.

We conduct the leave-one-player-out cross-validation on the data to evaluate the performance of the predictor. Just as above, for each player presented we construct the following train/test split:

- the training set consists of the game traces played by all players except this one;
- the test set consists of all the game traces played by this player.

As a performance measure we use the mean square difference between the exhibited behaviour in the test instances and the behaviour predicted by the SVR model. The mean is calculated over the time and over the test instances.

As baselines we use two simple models. The first one is a "random walk" predictor, i. e. a model that at each time step decides uniformly at random whether to increase the difficulty, to decrease it, or to stay at the same level.

The second baseline represents the best possible choice of the constant difficulty level in hindsight, i. e. given a test instance of a game trace it calculates the mean of the difficulty levels chosen by the player and at each time predicts this value.

The error values averaged over the test sets used in leave-one-player-out cross-validation are presented in Table 4. To illustrate the quality of the SVR predictor we have included plots of the randomly picked games from the test sets corresponding to different players. As you

Table 4: Comparison of performance of the SVR predictor and the baselines.

SVR	**0.027 ± 0.004**
Random	0.238 ± 0.013
Mean	0.086 ± 0.007

can see on Figure 20 the predicted difficulty (the black line) matches the difficulty set by the players themselves (the red line) fairly closely. It is all the more impressive if you keep in mind that there is no feedback to the predictor about the true values of difficulty level during the game trace.

4.8 CONCLUSION

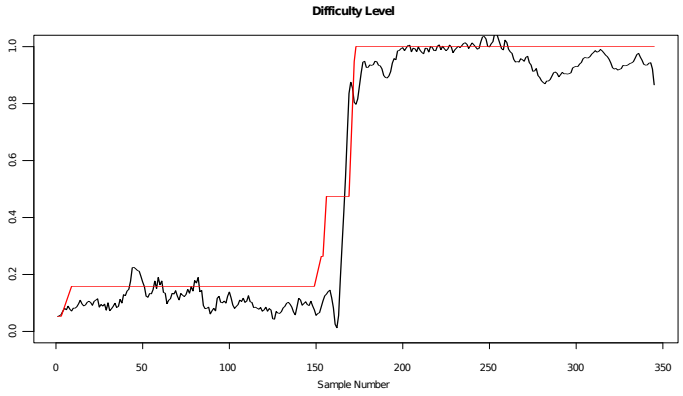
In this chapter we have shown how one can create a dynamic difficulty adjustment mechanism using off-the-shelf methods of machine learning, such as clustering, support vector classification, and support vector regression. The data that is required for these methods to work can be collected during such standard phases of game development as alpha and beta testing.

Our first approach to building a difficulty model consists of clustering game traces to infer playing types, finding a good difficulty adjustment for each cluster, and predicting the cluster for short traces of gameplay. Our experimental results confirm that the regression model and cluster prediction together outperform the alternatives significantly.

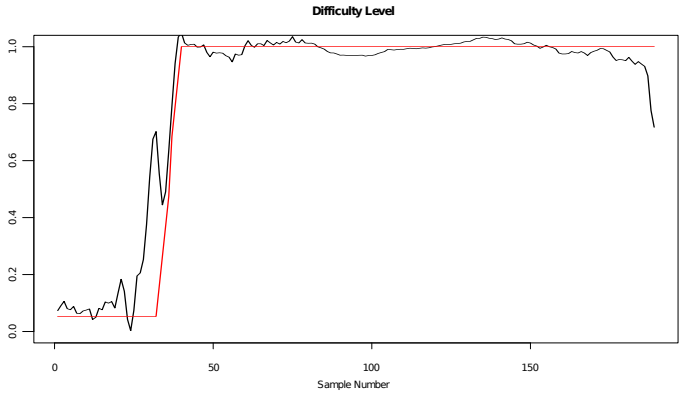
One important parameter that could influence the quality of the prediction is the length of the prefix used for the mapping. While we haven't investigated the limits of its influence, we have proposed a way to avoid this question completely by constructing a dynamic difficulty adjustment algorithm that uses extremely short time intervals of gameplay (three time steps in our experiments) to predict the next difficulty adjustment.

The error values as well as the visual investigation of the output of the dynamic algorithm allows us to state confidently that predicting the difficulty adjustments for each player based on short periods of gameplay is a viable approach for taking the burden of choosing the difficulty away from the players.

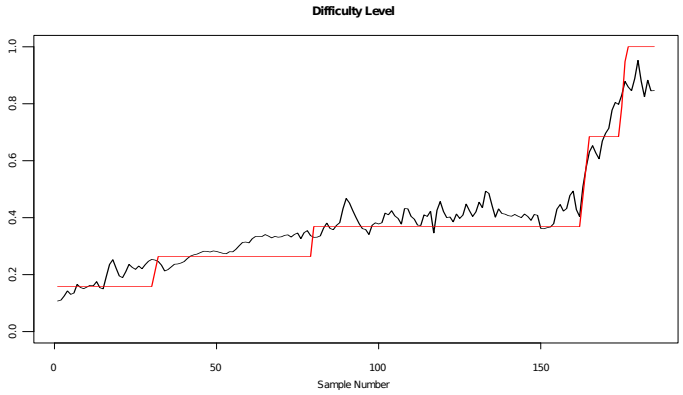
The quality of learned models critically depends on the quality of training data. Additionally, even though empirical results are encouraging, there are no guarantees on the performance of the presented algorithms. In the next chapter we will formulate the dynamic difficulty adjustment problem in such a way that it allows us to dispense with the need for training data, as well as to create an algorithm with an upper bound on the amount of mistakes it can make.



(a)



(b)



(c)

Figure 20: The comparison of the predicted difficulty values (the black lines) versus the actual difficulty values set by the players (the red line). Different plots represent randomly chosen games from the test sets corresponding to different players. The x-axis is the time, the y-axis is the difficulty rescaled to the $[0; 1]$ interval.

5

MASTER OF PARTIALLY ORDERED SETS

Imagine that you are a game developer, who wants to implement a dynamic difficulty adjustment mechanism in a new game. In an attempt to save time and money, you start by looking at what has been done by others in a similar situation. You find several heuristics and a few published algorithms. How should you decide, whether to take one of the existing routines, or to create you own?

The main motivation for this work was a wish to create a black box mechanism for dynamic difficulty adjustment: Something that any game developer can take and use in any game, while spending minimum time on adjusting it to their particular needs. Let's look at what properties game developers would wish for in such a device.

First, any such mechanism should be as independent of a game, as possible. This, in turn, means that it can be applied to any game, regardless of its type or structure.

Second, it should be as independent of and as non-intrusive to a player, as possible. It is perfectly fine to collect some information from a player engaged in a game with a purpose of improving her experience. But it is rather undesirable to subject a player to a lengthy training procedure.

Third, making it work should not cost a fortune in terms of time or money: No excessive training times, or excessive training data, or excessive setting up.

Fourth, it should work! There should exist guarantees about its performance. Heuristics don't have these by definition, and existing publications (algorithms) didn't seem to be interested in this question. Their performance is confirmed by empirical studies, which are usually small, limited to one game, or unrealistic (with the notable exception of the work by Herbrich et al. [38]).

Fifth, preferably, it should be understandable by a human. This simply because people tend to use things that they believe they understand.

In this chapter we present our answer to all of these requirements: An online dynamic difficulty adjustment algorithm, which is universal, usable almost as is, learning in an online fashion, possessing theoretical performance guarantees, and also quite simple.

5.1 INTRODUCTION

When game developers create controls for a player to modify the difficulty setting, they explicitly attach difficulty values to game states:

Each setting constrains the player to a subset of all possible game states of a particular difficulty value. A much less daunting task is, when presented with a pair of game states, to decide which one, if any, is more difficult than the other. Note, that some states can be incomparable in this way, inducing a partial order on all game states.

A player induces her own structure on game states at any particular time: Her skills and other qualities create a labelling that could be unknown even to the player herself, where some states are marked as 'too difficult', some as 'too easy', and others as 'just right'. Two things are worth noting about this labelling:

- It respects the difficulty relation between the states. If state i is more difficult than state j, their respective labels should reflect that.
- It changes with time. The skills development, the level of boredom or tiredness, or even change of a player would shift the labels.

The task of adjusting the difficulty automatically can be formulated in terms of online learning problem [3]: There are K arms (game states), to which an adversary (a player) assigns loss values on each iteration (0 to the 'just right' states, 1 to all the others). The goal of the algorithm is to choose a 'just right' arm as often as possible, or, equivalently, choose an arm on each iteration to minimize its overall loss. In Section 5.2 we briefly review partially ordered sets and classical multi-armed bandit setting.

Note that in contrast to the classical setting there is the additional information in the form of the structure over game states. The partial order allows the learning algorithm to infer more from the feedback on each iteration. In Section 5.3 we formalise the learning problem and propose a learning algorithm for it in Section 5.4. For this algorithm we then give a bound on the number of proposed difficulty settings that were not just right (in Section 5.5). The bound limits the number of mistakes the algorithm can make relative to the best static difficulty setting chosen in hindsight. For the bound to hold, no assumptions whatsoever need to be made on the behaviour of the player.

Last but not least we empirically study the behaviour of the algorithm under various circumstances (in Section 5.6). In particular, we investigate the performance of the algorithm 'against' statistically distributed players by simulating the players as well as 'against' adversaries by asking humans to try to trick the algorithm in a simplified setting. Furthermore, we present the results and challenges of implementing our algorithm into a real game and testing it on real human players.

5.2 PRELIMINARIES

In this chapter we are exclusively interested in applying adversarial online learning to partially ordered sets (posets). We begin with reviewing the needed properties of posets and the classical multi-armed bandit setting of online learning.

5.2.1 *Partially ordered sets*

A partially ordered set is a mathematical structure that consists of a set together with a binary relation that, given a pair of elements from the set, establishes a precedence of one of the pair's elements over the other. Note that the relation does not necessarily cover every pair of elements, this is reflected in the term 'partially'.

Definition 5.2.1. A *partial order* on a set $\mathcal{K}$ is a binary relation $\succeq \subseteq \mathcal{K} \times \mathcal{K}$, which is reflexive, antisymmetric, and transitive, i. e., for all $a, b, c \in \mathcal{K}$ holds:

- $a \succeq a$ (reflexivity),
- $a \succeq b, b \succeq a \Rightarrow a = b$ (antisymmetry),
- $a \succeq b, b \succeq c \Rightarrow a \succeq c$ (transitivity).

The pair $(\mathcal{K}, \succeq)$ is referred to as a *partially ordered set*, or a *poset* for short.

For $a, b \in \mathcal{K}$, if $a \succeq b$ or $b \succeq a$, then a and b are *comparable*, otherwise they are *incomparable*. A partial order under which every pair of elements is comparable is called a *total order* or *linear order*; a totally ordered set is also called a chain. A poset in which no two distinct elements are comparable is called *independent* or an *antichain*. A poset is called *dependent* if it has two distinct elements that are comparable. An element a *covers* an element b if $a \succeq b$, $a \neq b$, and there is no element $c \in \mathcal{K}$ with $a \succeq c$, $c \succeq b$.

An important notion for the algorithm that follows is a path cover. To introduce it, first we note that there is a direct connection between posets and directed acyclic graphs. Given a poset $(\mathcal{K}, \succeq)$, we construct a corresponding directed acyclic graph (dag) $G = (\mathcal{K}, E)$ by (i) identifying a set of vertices with $\mathcal{K}$, and (ii) creating an edge (i, j) for each pair of vertices $i, j \in \mathcal{K}$ such that i covers j.

Given a directed graph $G = (\mathcal{K}, E)$, a *directed path* is a sequence of edges in E, $P = (v_1, v_2), (v_2, v_3), \ldots, (v_{n-1}, v_n)$, such that (i) all involved vertices are distinct, and (ii) all edges are distinct. A *path cover* is a set of directed paths such that every vertex $k \in \mathcal{K}$ belongs to at least one path. A *minimum path cover* is a path cover containing the least amount of paths.

Recall that an antichain is a poset in which no two distinct elements are comparable. In terms of the corresponding dag that means that the set of edges E is empty, i. e. no two vertices are adjacent. The minimum path cover of this graph consists of paths of length zero: Each vertex belongs to its own path. Therefore, the size of the minimum path cover is exactly the size of the vertices' set.

Consider antichains that are subsets of the original poset $(\mathcal{K}, \succeq)$. Each of them has a corresponding *independent set* in G, i.e a set of vertices no two of which are adjacent. In any path cover all the vertices of such set have to belong to distinct paths. Hence, the size of the minimum path cover is at least the size of the maximum antichain in $(\mathcal{K}, \succeq)$. In fact, it is exactly this size, as stated by Dilworth's theorem [18]:

Theorem 5.2.1 (Dilworth's Theorem)**.** *Let every set of* $k+1$ *elements of a partially ordered set* P *be dependent, while at least one set of* k *elements is independent. Then* P *is a set sum of* k *disjoint chains.*

5.2.2 *Online learning*

Online learning is concerned with learning in a sequential setting. In the original formulation, on each round the learner (i) is presented with some information about the world it is in, (ii) makes its prediction, (iii) and is informed in some way about the quality of its prediction. Then the game moves on to the next round. The goal of the learner is to make as few mistakes as possible [52]. In classical online learning, the first step of each round consists of giving the learner an instance from the learning domain, while in the full information or bandit games the learner receives information about the structure of the loss function [25, 3]. Note also that in contrast to the setting presented by Littlestone [52], the main focus of research in this area has been on minimizing the total loss achieved by the learner over the T rounds, rather than the amount of mistakes.

The bulk of research on online learning has been concentrating on the case where the problems' domain is finite, i. e. there are K actions, arms, experts, etc, and on each round the learner chooses one (or several) of them. The multi-armed bandit setting refers to the case, where the feedback is restricted to the loss incurred by the arm the learner predicted. The traditional measure of the learner's performance is the so-called regret: the difference between the loss of the best static choice in hindsight and of the learner (the expected regret in case of randomized algorithms). The overall goal is to create a learner that achieves a regret sublinear in t. Various bounds on the regret were proven for various assumptions on the adversary's (environment's) behaviour, for more information we refer the reader to the excellent book by Cesa-Bianchi and Lugosi [9].

The best known to-date algorithm that does not use any additional information is the Improved Bandit Strategy (called ImprovedPI in the following) [9]. The upper bound on its regret is of the order $\sqrt{KT\ln(T)}$, where T is the amount of iterations. ImprovedPI will be the second baseline after the best static in hindsight (Bsih) in our experiments.

5.3 FORMALISATION

To be able to theoretically investigate dynamic difficulty adjustment, we view it as a meta-game between a *master* and a *player*, played on a partially ordered set modelling the 'more difficult than'-relation. The meta-game is played in turns where each turn has the following elements:

1. the *master* chooses a difficulty setting,
2. the *player* plays one 'round' of the game in this setting, and
3. the *master* experiences whether the setting was 'too difficult', 'just right', or 'too easy' for the player.

The *master* aims at making as few as possible mistakes, that is, at choosing a difficulty setting that is 'just right' as often as possible. In this chapter, we aim at developing an algorithm for the *master* with theoretical guarantees on the number of mistakes in the worst case while not making any assumptions about the *player*.

To simplify our analysis, we make the following, rather natural assumptions:

- the set of difficulty settings is finite and
- in every round, the (hidden) difficulty settings respect the partial order, that is,
  - no state that 'is more difficult than' a state which is 'too difficult' can be 'just right' or 'too easy' and
  - no state that 'is more difficult than' a state which is 'just right' can be 'too easy'.

Even with these natural assumptions, in the worst case, no algorithm for the *master* will be able to make even a single correct prediction. As we can not make any assumptions about the *player*, we will be interested in comparing our algorithm theoretically and empirically with the best statically chosen difficulty setting, as is commonly the case in online learning [9].

This setting is related to learning directed cuts with membership queries. For learning directed cuts, i. e., monotone subsets, Gärtner and Garriga [28] provided algorithms and bounds for the case in which the labelling does not change over time. They then showed that

the intersection between a monotone and an antimonotone subset in not learnable. This negative result is not applicable in our case, as the feedback we receive is more powerful. They furthermore showed that directed cuts are not learnable with traditional membership queries if the labelling is allowed to change over time. This negative result also does not apply to our case as the aim of the *master* is "only" to point at a green vertex as often as possible and as we are interested in a comparison with the best static vertex chosen in hindsight.

5.4 ALGORITHM

In this section we give a multiplicative update algorithm for predicting a vertex that corresponds to a 'just right' difficulty setting in a finite partially ordered set $(\mathcal{K}, \succ)$ of difficulty settings. The partial order is such that for $i, j \in \mathcal{K}$ we write $i \succ j$ if difficulty setting i is 'more difficult than' difficulty setting j. The learning rate of the algorithm is denoted by $\beta \in (0, 1)$. The response that the *master* algorithm can observe o_t is $+1$ if the chosen difficulty setting was 'too easy', 0 if it was 'just right', and -1 if it was 'too difficult'. The algorithm maintains a belief w of each vertex being 'just right' and updates this belief if the observed response implies that the setting was 'too easy' or 'too difficult'.

Algorithm 3 PARTIALLY-ORDERED-SET MASTER (POSM) for Difficulty Adjustment

Require: parameter $\beta \in (0, 1)$, K difficulty Settings $\mathcal{K}$, partial order $\succ$ on $\mathcal{K}$, and a sequence of observations $o_1, o_2, \ldots$

1: $\forall k \in \mathcal{K}$: let $w_1(k) = 1$
2: **for** each turn $t = 1, 2, \ldots$ **do**
3: $\quad \forall k \in c_t$: let $A_t(k) = \sum_{x \in \mathcal{K} : x \succeq k} w_t(k)$
4: $\quad \forall k \in c_t$: let $B_t(k) = \sum_{x \in \mathcal{K} : x \preceq k} w_t(k)$
5: $\quad$ PREDICT $k_t = \operatorname{argmax}_{k \in \mathcal{K}} \min\{B_t(k), A_t(k)\}$
6: $\quad$ OBSERVE $o_t \in \{-1, 0, +1\}$
7: $\quad$ **if** $o_t = +1$ **then**
8: $\quad\quad \forall k \in \mathcal{K}$: let $w_{t+1}(k) = \begin{cases} \beta w_t(k) & \text{if } k \succeq k_t \\ w_t(x) & \text{otherwise} \end{cases}$
9: $\quad$ **end if**
10: $\quad$ **if** $o_t = -1$ **then**
11: $\quad\quad \forall k \in \mathcal{K}$: let $w_{t+1}(k) = \begin{cases} \beta w_t(k) & \text{if } k \preceq k_t \\ w_t(x) & \text{otherwise} \end{cases}$
12: $\quad$ **end if**
13: **end for**

The main idea of Algorithm 3 is that in each round we want to make sure we can update as much belief as possible. The significance

of this will be clearer when looking at the theory in the next section. To ensure it, we compute for each setting k the belief 'above' k as well as 'below' k . That is, A_t in line 3 of the algorithm collects the belief of all settings that are known to be 'more difficult' and B_t in line 4 of the algorithm collects the belief of all settings that are known to be 'less difficult' than k. Upon each mistake, depending on the observation, we will be able to either update the believes above or below the chosen setting and as we are considering a worst case scenario, we can only guarantee to update an amount of belief larger than or equal to $\min\{B_t(k), A_t(k)\}$. To achieve the best performance, we choose the k that gives us the best worst case guarantee in line 5 of the algorithm.

5.5 THEORY

We will now show a bound on the number of inappropriate difficulty settings that are proposed, relative to the number of mistakes the best static difficulty setting makes. We denote the number of mistakes of the *master* algorithm until time T by m and the minimum number of times a statically chosen difficulty setting would have made a mistake until time T by M. Furthermore, we denote the total amount of belief on $\mathcal{K}$ at time t by $W_t = \sum_{k\in\mathcal{K}} w_t(k)$.

The analysis of the algorithm relies heavily on a path cover of $\mathcal{K}$ that we denote by $\mathcal{C}$. To achieve the tightest bound, the minimum path cover of $\mathcal{K}$ should be chosen. The minimum path cover can be found in time polynomial in $|\mathcal{K}|$ and its size is equal to the size of the largest antichain in $(\mathcal{K}, \succ)$.

For all $c \in \mathcal{C}$ we denote the amount of belief on every chain by $W_t^c = \sum_{x\in c} w_t(x)$, the belief 'above' k on c and 'below' k on c by $A_t^c(k) = \sum_{x\in c: x\succeq k} w_t(x)$ and by $B_t^c(k) = \sum_{x\in c: x\preceq k} w_t(x)$, respectively. To relate the amount of belief updated in the algorithm to the amount of belief on each chain observe that

$$\begin{aligned}
&\max_{k\in\mathcal{K}} \min\{A_t(k), B_t(k)\} \\
=&\max_{c\in\mathcal{C}} \max_{k\in c} \min\{A_t(k), B_t(k)\} \\
\geqslant&\max_{c\in\mathcal{C}} \max_{k\in c} \min\{A_t^c(k), B_t^c(k)\} .
\end{aligned}$$

Denote the 'heaviest' chain by

$$c_t = \operatorname*{argmax}_{c\in\mathcal{C}} \max_{k\in c} \min\{A_t^c(k), B_t^c(k)\} . \tag{3}$$

We will now show that in each iteration in which our algorithm proposes an inappropriate difficulty setting, we update at least half of the weight of the heaviest chain. That is, we will first show that $\max_{k\in c_t} \min\{A_t^{c_t}(k), B_t^{c_t}(k)\} \geqslant W_t^{c_t}/2$. For that, we choose

$$i = \operatorname*{argmax}_{k\in c_t} \{B_t^{c_t}(k) \mid B_t^{c_t}(k) < W_t^{c_t}/2\}$$

and

$$j = \operatorname*{argmin}_{k \in c_t} \{B_t^{c_t}(k) \mid B_t^{c_t}(k) \geqslant W_t^{c_t}/2\} .$$

This way, we obtain $i, j \in c_t$ for which $B_t^{c_t}(i) < W_t^{c_t}/2 \leqslant B_t^{c_t}(j)$ and which are consecutive, that is, $\nexists k \in c : i \prec k \prec j$. Such i, j exist and are unique as $\forall x \in \mathcal{K} : w_t(x) > 0$. We then have $B_t^{c_t}(i) + A_t^{c_t}(j) = W_t^{c_t}$ and thus also $A_t^{c_t}(j) > W_t^{c_t}/2$. This immediately implies

$$W_t^{c_t}/2 \leqslant \min\{A_t^{c_t}(j), B_t^{c_t}(j)\} \leqslant \max_{k \in c_t} \min\{A_t^{c_t}(k), B_t^{c_t}(k)\} .$$

That is, for each mistake we update an amount of belief greater than or equal to $W_t^{c_t}/2$. As, furthermore $W_t^{c_t} \geqslant W_t/|\mathcal{C}|$, the total amount of belief changes such that

$$\begin{aligned} W_{t+1} &\leqslant W_t^{c_t}/2 + \beta W_t^{c_t}/2 + (W_t - W_t^{c_t}) \\ &\leqslant \frac{W_t}{2|\mathcal{C}|} + \beta \frac{W_t}{2|\mathcal{C}|} + (W_t - W_t/|\mathcal{C}|) \\ &= \left[\frac{|\mathcal{C}| - 1}{2|\mathcal{C}|} + \frac{\beta}{2|\mathcal{C}|}\right] W_t . \end{aligned}$$

Applying this bound recursively, we obtain for time T

$$\begin{aligned} W_T &\leqslant W_0 \left[\frac{|\mathcal{C}| - 1}{2|\mathcal{C}|} + \frac{\beta}{2|\mathcal{C}|}\right]^m \\ &\leqslant |\mathcal{K}| \left[\frac{|\mathcal{C}| - 1}{2|\mathcal{C}|} + \frac{\beta}{2|\mathcal{C}|}\right]^m . \end{aligned}$$

As we only update the weight of a difficulty setting if the response implied that the algorithm made a mistake, β^M is a lower bound on the weight of one difficulty setting on this chain and hence also $W_T \geqslant \beta^M$. Solving

$$\beta^M \leqslant |\mathcal{K}| \left[\frac{|\mathcal{C}| - 1}{2|\mathcal{C}|} + \frac{\beta}{2|\mathcal{C}|}\right]^m$$

for m, we obtain

$$m \leqslant \left\lfloor \frac{\ln |\mathcal{K}| + M \ln 1/\beta}{\ln \frac{2|\mathcal{C}|}{2|\mathcal{C}| - 1 + \beta}} \right\rfloor .$$

Note, that this bound is similar to the bound for the full information setting [9] despite much weaker information being available in our case. The influence of $|\mathcal{C}|$ is the new ingredient that changes the behaviour of this bound for different partially ordered sets.

5.6 EXPERIMENTS

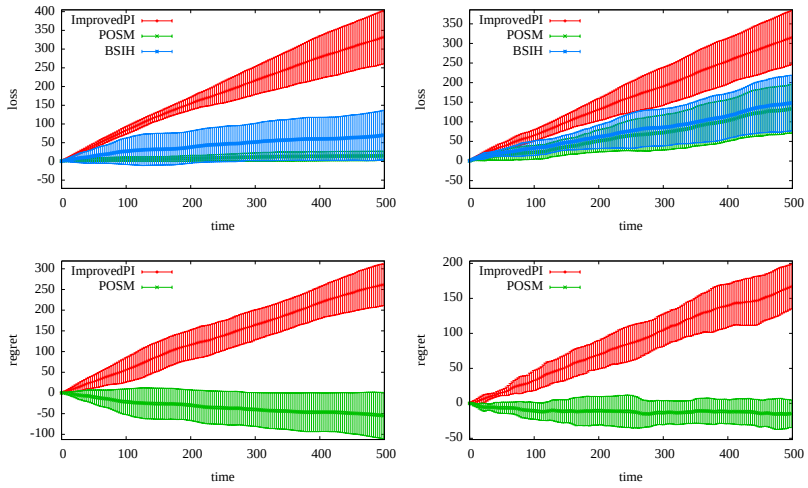
We performed two sets of experiments: (i) in an artificial environment we evaluated how well POSM can minimize the regret when dealing with various kinds of adversaries; (ii) in Chinese chess we evaluated how well POSM can perform against various opponents, how well it can predict the skill level of an opponent, and how satisfied human players were with POSM's playing skill.

5.6.1 *Artificial environment*

To evaluate the performance of POSM independently of any real game, we created an artificial environment replicating the online learning setting on K arms with a partial order. To this purpose we simulated a stochastic opponent, as well as used human players to provide our algorithm with a non-stochastic non-oblivious adversary. In this environment we compare the total loss of our algorithm with two baselines. The first one is the best static difficulty setting in hindsight: it is a difficulty that a player could pick if she knew her skill level in advance and had to choose the difficulty only once. The second one is the ImprovedPI algorithm [9].

Note that because of our assumptions about the nature of difficulty settings and the specifics of the poset, the labels build three distinct "zones": all the states that are 'too easy', all the states that are 'just right', and all the states that are 'too difficult'. In the following we refer to the set of vertices with 'just right' labels as the *zero-zone* (because in the corresponding loss vector their components are equal to zero). The *border* of the zero-zone Z is a set of vertices $B \subseteq \mathcal{K}$ such that each vertex in B has at least one neighbour in the zero-zone and at least one neighbour in the complement of the zero-zone, $B = \{b \in \mathcal{K} \mid N(b) \cap Z \neq \emptyset \quad \text{and} \quad N(b) \cap (\mathcal{K} \setminus Z) \neq \emptyset\}$, where $N(b) = \{k \in \mathcal{K} \mid k \text{ covers } b \quad \text{or} \quad b \text{ covers } k\}$.

In both stochastic and adversarial scenarios we consider two different settings: so called 'smooth' and 'non-smooth' one. The settings' names describe the way the zero-zone changes with time. In the 'non-smooth' setting we don't place any restrictions on Z apart from its size, while in the 'smooth' setting no more than one vertex from B can change its label at each iteration. These two settings represent two extreme situations: one player changing her skills gradually with time causes the labels on the poset to change 'smoothly'; switching players during the course of a single game makes the zero-zone 'jump' over the poset. In a more realistic scenario the zero-zone would change 'smoothly' most of the time, but sometimes it would perform jumps.



(a) On a chain of 50 vertices (b) On a grid of 7×7 vertices

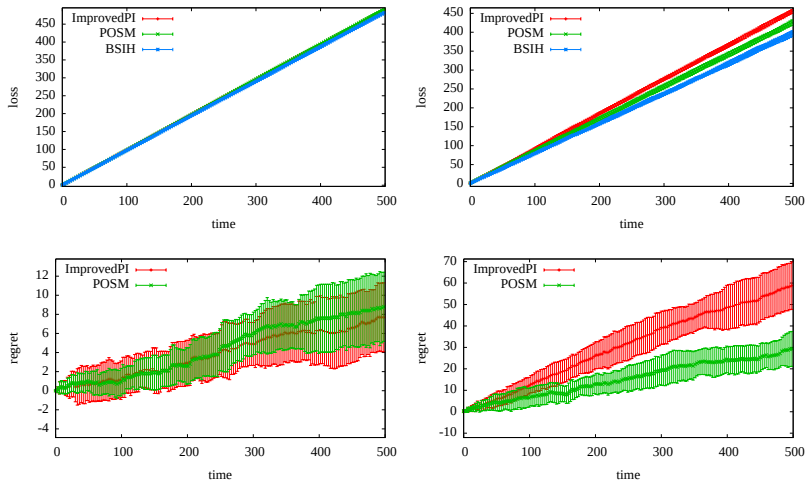
Figure 21: Stochastic adversary, 'smooth' setting.

5.6.1.1 *Stochastic adversary*

In the first set of experiments we performed, the adversary is stochastic: On every iteration the zero-zone can change in a pre-defined way with a pre-defined probability. In the 'smooth' setting only one of the border vertices of the zero-zone at a time can change its label to 1, -1, or stay as it is with an equal probability. Note that altering a vertex' label may break the consistency of the labelling with regard to the poset. The necessary repair procedure may result in more than one vertex being relabelled at a time.

For the 'non-smooth' setting we consider two cases: an easier one, where the zero-zone may contain up to 20% of all the vertices in the graph, and an especially difficult one, with the zero-zone limited to always containing only one vertex. The zero-zone is created on each iteration anew. First, a vertex in a graph is chosen uniformly at random and its label is set to zero. Then, the zero-zone is 'grown', with one neighbouring vertex at a time chosen uniformly at random and its label also set to zero, until the size limit is reached. After that all the vertices upstream and downstream from the new zero-zone are labelled in a consistency preserving way.

We consider two graphs that represent two different but typical game structures with regard to the difficulty: a single chain and a 2-dimensional grid. A set of progressively more difficult challenges that can be found for instance in a puzzle or a time-management game can be directly mapped onto a chain of a length corresponding to the amount of challenges. A 2- (or more-) dimensional grid on the other hand is more like a skill-based game, where depending on the choices players make, different game states become available to them. Structurally, these two posets have very different path covers.



(a) One vertex in the zero-zone

(b) Up to 20% of vertices in the zero-zone

Figure 22: Stochastic adversary, 'non-smooth' setting, on a chain of 50 vertices.

The minimal path cover of a chain consists of exactly one path, the chain itself, while for the 2-dimensional grid its size is the size of the largest antichain, the diagonal of the grid. In our experiments the chain contains 50 vertices, while the grid is built on 7×7 vertices (the size of the grid is chosen so that it is close to the length of the chain), the size of its minimal path cover is 7. Hence, we expect the performance of POSM to be worse on the grid than on the chain.

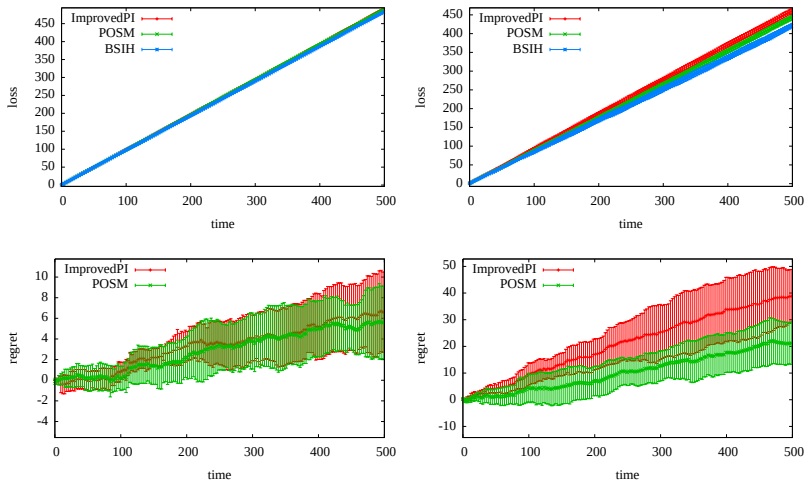
In all considered variations of the setting the game lasts for 500 iterations and is repeated 10 times. The resulting mean and standard deviation values of loss and regret, respectively, are shown in the following figures: The 'smooth' setting in Figure 21; The 'non-smooth' setting in Figures 22, and 23.

Note that in the 'smooth' setting Posm is outperforming Bsih and, therefore, its regret is negative. As expected, both the loss and the regret are worse on the grid. In the considerably more difficult 'non-smooth' setting all algorithms perform badly (as expected). Nevertheless, in the slightly easier case of larger zero-zone, Posm performance in terms of regret gets better.

While Bsih is a baseline that can not be implemented as it requires to foresee the future, Posm is a correct algorithm for dynamic difficulty adjustment. Therefore, it is surprising that Posm performs almost as good as Bsih or even better.

5.6.1.2 *Evil adversary*

While the experiments in our stochastic environment show encouraging results, of real interest to us is the situation where the adversary is



(a) One vertex in the zero-zone (b) Up to 20% of vertices in the zero-zone

Figure 23: Stochastic adversary, 'non-smooth' setting, on a grid of 7×7 vertices.

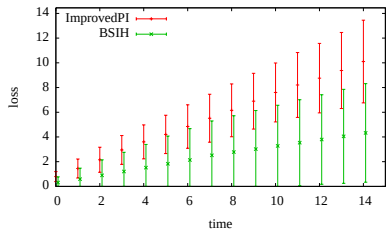
'evil', non-stochastic, and non-oblivious. In dynamic difficulty adjustment the algorithm will have to deal with people, who are learning and changing in hard to predict ways.

We limit our experiments to the case of a linear order on difficulty settings, in other words, the chain. Even though it is a simplified scenario, this situation is rather natural for games and it demonstrates the power of our algorithm.

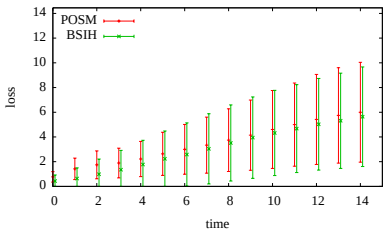
Instead of implementing the adversary as was the case for the stochastic scenario, we use people as adversaries. Just as in games with the dynamic difficulty adjustment players often are not supposed to be aware of the mechanics, our methods and goals were not disclosed to the testing persons. Instead they were presented with a modified game of cups: On every iteration the casino is hiding a coin under one of the cups; after that the player can point at two of the cups. If the coin is under one of these two, the player wins it.

Behind the scenes the cups represented the vertices on the chain and the players' choices were setting the lower and upper borders of the zero-zone. If the algorithm's prediction was wrong, one of the two cups was decided on randomly and the coin was placed under it. If the prediction was correct, no coin was awarded.

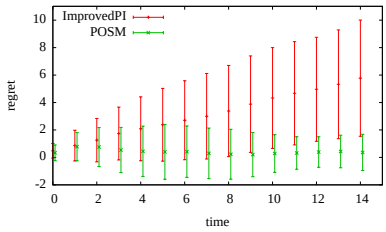
Unfortunately, using people in such experiments places severe limitations on the size of the game. Without the internal game appeals, such as a story, achievements, or social interactions, and without any extrinsic rewards they can only handle short chains and short games before getting bored. In our case we restricted the length of the chain to 8 and the length of each game to 15 iterations.



(a) Games vs ImprovedPI.



(b) Games vs Posm.



(c) Regret.

Figure 24: Evil adversary, 'smooth' setting, a single chain of 8 vertices.

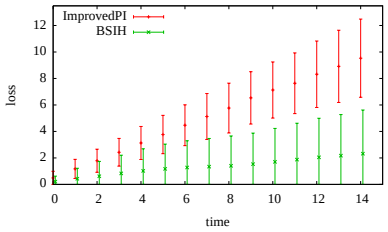
Again, we created the 'smooth' and 'non-smooth' settings by placing or removing restrictions on how players were allowed to choose their cups. To each game either ImprovedPI or Posm was assigned uniformly at random. The results for both settings are presented in Figures 24 and 25. Note, that due to the fact that this time different games were played by ImprovedPI and Posm, we have two different plots for their corresponding loss values.

We can see that in the 'smooth' setting again the performance of Posm is very close to that of Bsih. In the more difficult 'non-smooth' one, the results are also encouraging. Note, that in the games that people played against Posm, the loss of Bsih appears to be worse. A plausible interpretation is that players had to follow more difficult (less static) strategies to fool Posm to win their coins. Nevertheless, the regret of Posm is small even in this case.

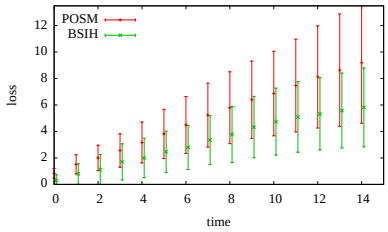
The experiments performed in this synthetic environment show that the additional information encoded in the structure of a poset is indeed beneficial and helps POSM to achieve better loss and regret values, when compared to ImprovedPI. How much this information helps depends on the structure of the poset, namely on the size of its minimal path cover, as well as on the maliciousness of the adversary. Next we will look what POSM can achieve in a real game, namely, in Chinese chess.

5.6.2 *Chinese chess*

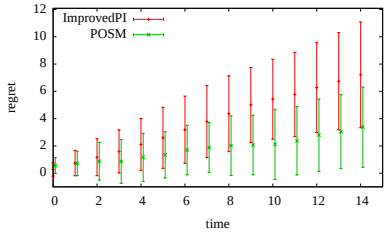
Chinese chess or Xiangqi is a two-player Chinese board game (Figure 26), which is similar to Western chess. It is one of the most popular



(a) Games vs ImprovedPI.



(b) Games vs Posm.



(c) Regret.

Figure 25: Evil adversary, 'non-smooth' setting, a single chain of 8 vertices.

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). For a comprehensive overview of the rules we refer the reader to one of the online sources[1]. The first scientific paper on computer Chinese chess was published by Zhang [102]. The earliest human versus computer competition was the annual ACER cup, which was held in Taiwan from 1985 to 1990.

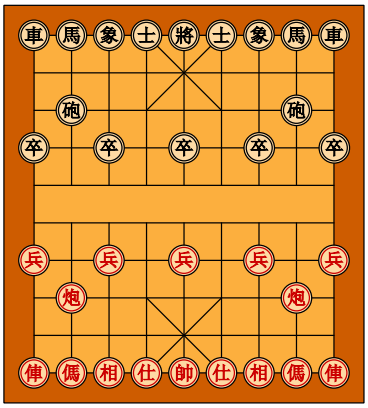


Figure 26: Xiangqi board in the starting position.

Our computer opponents use the MiniMax algorithm with a static evaluation function, which we have constructed using the work published by Wang et al. [97] and Chen [11]: It is a sum of three major features of a given board configuration, each feature represented by a piece-square table.

The first major feature is *material*. We calculate it by adding up the values of the pieces each player still has on the board and subtracting the two sums from each other. The value for each piece is determined by its type and position.

1 Tutorial for Chinese chess: www.chessvariants.org/xiangqi.html

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 defence.

The third major feature we use is *mobility*. 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.

Each score is calculated using the values in the corresponding piece-square tables, then the evaluation function sums up all three scores for a given board state. In the following we refer to a computer opponent using the MINIMAX algorithm with the search depth i and this evaluation function CHESSAI(i).

In the game of Xiangqi a draw is usually declared by a referee. Furthermore, we had 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.

POSM requires an observation oracle that provides feedback about the predicted difficulty setting. The observation oracle that we implemented for Xiangqi applies to the current state the evaluation mechanism of CHESSAI(i), where i is the current difficulty setting, and observes how the move's evaluation score 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.

5.6.2.1 *Results*

When evaluating the Xiangqi games that POSM played against computer and human opponents, we focused on two questions: (i) how well POSM can learn the opponent's skill; (ii) how well human players are satisfied with its skill.

We will first describe the computer player results. To see how well POSM can predict the opponent's search depth we ran episodes of

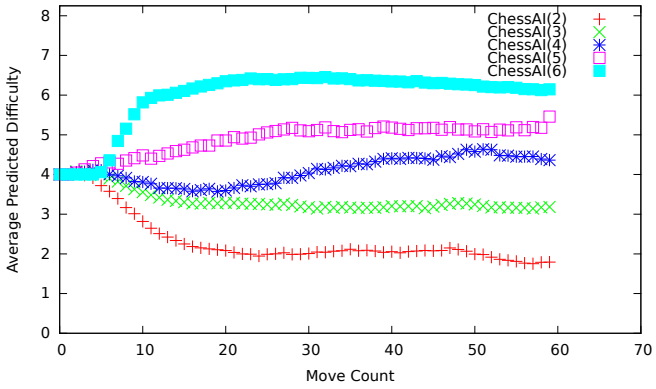


Figure 27: Posm versus ChessAI(2-6). Average predicted difficulty over episodes of 100 games against each algorithmic opponent.

100 games of Posm versus ChessAI(i) with $i \in \{2, \ldots, 6\}$. Figure 27 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 5.4 the algorithm always picks the medium difficulty setting at the start, in this case 4. As the game progresses and Posm receives more feedback, the predicted difficulty setting reaches the opponent's skill value.

To answer the second question, we have provided an implementation of Xiangqi with Posm to human players. 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[2], 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 28. The results of the survey as well as the spread of the players are depicted in Figure 29. To have a quantitative representation of human players' skills, we use the same *relative strength ratio* metric as in Section 3.3.1. A scatter plot of the strength results for human players is shown in Figure 30.

A short analysis of Figure 29 and Figure 30 shows a strong correlation between the results of the survey and the relative strength ratio, i. e., if the strength is high the survey says the opponent was too easy and vice versa. The survey results are biased by human emotion and

2 Game platform used to rank the human players: www.qqgame.qq.com/

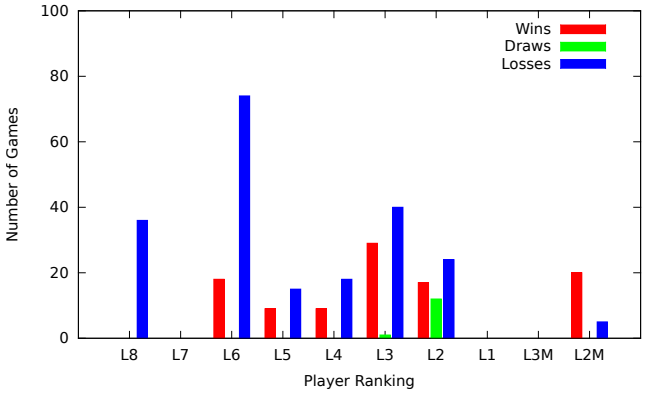


Figure 28: POSM versus human testers. The number of games ending with wins/draws/loses for each category of player.

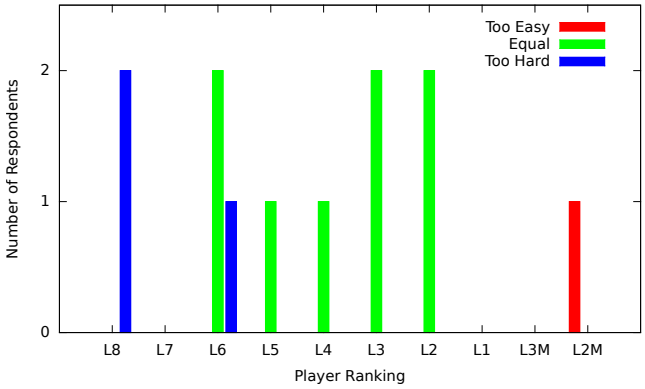


Figure 29: Survey results when grouped by category of players. The results show the answers to the question "How difficult did you find the game?".

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 is illustrated by the Level 6 players, for whom the proportion of won and lost games is far from 50%, but their responses in the survey show that they perceived their opponent as being equally strong.

5.7 CONCLUSION

In this chapter we formalised dynamic difficulty adjustment as an online learning problem on partially ordered sets and proposed an online learning algorithm POSM for dynamic difficulty adjustment. Using this formalisation, we were able to prove a bound on the performance of POSM relative to the best static difficulty setting chosen in hindsight BSIH. To validate our theoretical findings empirically, we

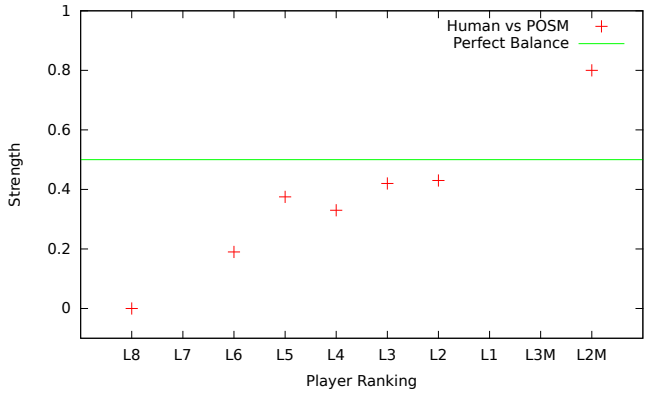


Figure 30: Humans versus POSM STRENGTH scatter plot. The STRENGTH achieved by each category of players can be seen on the y-axis.

performed a set of experiments, comparing POSM and another state-of-the-art algorithm to BSIH in two settings (a) simulating the player by a stochastic process and (b) simulating the player by humans that are encouraged to play as adversarial as possible. These experiments showed that POSM performs very often almost as well as BSIH and, even more surprisingly, sometimes even better. As this is also even better than the behaviour suggested by our mistake bound, there seems to be a gap between the theoretical and empirical performance of our algorithm.

Apart from the synthetic experiments, we have also shown how POSM can be incorporated in board games, by taking Chinese chess as an example 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 weak, or too smart.)

6 TRAVEL DIARY

6.1 GOAL

In this work we concentrated on the question of creating a dynamic difficulty adjustment algorithm. The hypothesis of usefulness of such an algorithm is based on two theories:

- Play is important for brain development [96] and for acquiring new skills [6];
- Games that adapt to their players can sustain their attention and keep their interest for longer [29, 84].

Additional evidence for the usefulness of dynamic difficulty adjustment algorithms comes from the industry. As reviewed in Chapter 2, for the last twenty to thirty years commercial game developers were creating various mechanisms to adapt the game difficulty to players. Most, if not all, of their approaches are heuristics and suffer from typical heuristics problems, the biggest of which is non-transferability: for a new game a new system has to be created.

Motivated by this, in the process of working on this thesis our decisions were guided by the following requirements:

- Universality: The resulting dynamic difficulty adjustment system should be applicable to as many games as possible, independently of their type, structure, or features;
- Non-intrusiveness: If desired, the resulting system should work in a transparent way, in real-time, not requiring an interaction with players;
- Feasibility: There should be guarantees about the resulting system's performance.

6.2 LANDSCAPE

Reviewing the academic results in the area of dynamic difficulty adjustment showed that the majority of the existing approaches are based on reinforcement learning or evolutionary algorithms. These methods do not satisfy the requirements above:

- They require a reward or fitness function, designing which demands both the domain knowledge and the algorithm knowledge. The structure of this function critically influences the performance of the learner.

- While both have various guarantees on their convergence, in practical terms they have long learning times and do not scale well to the high-dimensional state-spaces.
- The long learning times lead to the training using artificial opponents with unclear results concerning adaptation to human players.

All of the above makes the proposed approaches not very suitable for using in games created and played by humans.

6.3 PLACES VISITED

On the way to our destination, we have first considered games as a series of interactions between the player and in-game entities, where in each interaction both the player and her opponent has a selection of moves to choose from. The simple dynamic difficulty adjustment heuristic we have proposed, ADAPTIVE RANKER, is based on the idea that *the choices the player makes during the game are based on her skill*. Hence, if we give the opponent a way to estimate the skill and to choose the move that represents the same level of skill, the resulting game should appear to the player balanced in terms of difficulty. As a way to estimate the skill of the player we use the ranking on the available moves produced by an evaluation function.

We have shown that in a good case, i. e. when the evaluation function estimates the quality of the moves sufficiently well, and the player and her opponent have a similar choice of moves, ADAPTIVE RANKER performs well, adapting both to artificial opponents and to human players. We have also shown that in a different case, where the choices available to the player and her opponent are sufficiently different, ADAPTIVE RANKER fails to adapt. While this doesn't disprove the principle, it shows that this approach does not fulfil the requirements listed above.

Next, we have proposed to use the testing phases of game development process as a source of training data for a supervised learning approach to dynamic difficulty adjustment. The assumption here is that *the difficulty adjustments created by the testers represent the difficulty adjustments that players would do in similar game situations.* Hence, a supervised learner creates a mapping between sequences of game states and (models of) difficulty adjustments. The experiments we have performed show that this approach works well, predicting the difficulty for an unseen player that is close to the one she has chosen herself. It remains to be seen, how satisfied human players will be by the suggested difficulty adjustments. The difficulty modelling fulfils two of the requirements listed above, universality and non-intrusiveness, with the only constraint that it needs training data, i. e. game traces

that include difficulty adjustments and represent the players' population.

Finally, we have formulated the dynamic difficulty adjustment problem as an online learning problem on structured data. The crucial insight was that, *with regard to the difficulty, the game states form a partially ordered set*. The formulation led to the online learner on a partially ordered set, POSM. Applied to the dynamic difficulty adjustment problem, POSM attempts to choose for the player game states of "just right" difficulty, while maximising the amount of information it would gain upon a mistake. POSM fulfils all the requirements above. It can be taken as is and implemented in any game that possesses states of varying difficulty. It adapts to the player in real-time. We have proved an upper bound on the amount of mistakes that POSM will make in T iterations. Due to this, we can say that the goal we have set to ourselves has been achieved.

6.4 FURTHER DIRECTIONS

One particular aspect of our work that requires further attention, is conducting, on a larger scale, experiments involving human players. As they stand at the moment, our experimental results with humans have an encouraging but tentative character. Every time human players participated in our studies, we discovered some unexpected, surprising behaviour that directly influenced the results. Hence, we feel there is still much to learn by exposing our algorithms to humans.

Another interesting direction to pursue is online learning in non-standard spaces. While there was a lot of research done in the recent years about online learning in a Euclidean space, very few people considered the idea that convexity, the ground concept on which online learning is based, is not constrained to a Euclidean space. Rather it is a combinatorial notion, applicable to any space where a convex hull operator can be defined. An example of POSM shows that convexity induces a structure on the input space, which is beneficial for an online learner, since it provides additional information. Exactly how beneficial this information is and how it can be used, remains to be discovered.

> *Our battered suitcases were piled on the sidewalk again; we had longer ways to go. But no matter, the road is life.*
>
> — Jack Kerouac [46]

BIBLIOGRAPHY

[1] Left 4 Dead 411. Left 4 Dead hands-on preview. http://www.left4dead411.com/left-4-dead-preview-pg1, 2008. Accessed: 14/12/2014.

[2] Victor Allis. A knowledge-based approach of connect-four. The game is solved. Master's thesis, Free University of Amsterdam, October 1988.

[3] Peter Auer, Nicolò Cesa-Bianchi, Yoav Freund, and Robert E. Schapire. Gambling in a rigged casino: The adversarial multi-armed bandit problem. *Annual IEEE Symposium on Foundations of Computer Science*, 0:322, 1995. ISSN 0272-5428. doi: 10.1109/SFCS.1995.492488.

[4] Sander Bakkes, Pieter Spronck, and Jaap van den Herik. Rapid and reliable adaptation of video game AI. *IEEE Transactions on Computational Intelligence and AI in Games*, 1(2):93–104, 2009.

[5] C. Bauckhage, B. Gorman, C. Thurau, and M. Humphrys. Learning human behavior from analyzing activities in virtual environments. *MMI-Interaktiv*, 12:3–17, 2007. URL http://www.mmi-interaktiv.de/uploads/media/mmiij-vr_02.pdf.

[6] Irving Biederman and Edward Vessel. Perceptual pleasure and the brain. *American Scientist*, 94(3), 2006.

[7] Cameron Browne. *Automatic Generation and Evaluation of Recombination Games*. PhD thesis, Queensland University of Technology, February 2008. URL http://dev.pubs.doc.ic.ac.uk/phd-game-design/phd-game-design.pdf.

[8] Luigi Cardamone, Daniele Loiacono, and Pier Luca Lanzi. Interactive evolution for the procedural generation of tracks in a high-end racing game. In *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*, GECCO '11, pages 395–402, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0557-0. doi: 10.1145/2001576.2001631.

[9] Nicolò Cesa-Bianchi and Gábor Lugosi. *Prediction, learning, and games*. Cambridge university press, 2006. ISBN 978-0-521-84108-5.

[10] Darryl Charles and Michaela Black. Dynamic player modeling: A framework for player-centered digital games. In *Proceedings of the International Conference on Computer*

Games: Artificial Intelligence, Design and Education, pages 29–35, 2004. URL http://research.rmutp.ac.th/paper/cu/DynamicPlayerModelling.pdf.

[11] H. Chen. Elephanteye. v3.13l. *Source code published at: http://sourceforge.net/projects/xqwizard/*, 2008.

[12] Daniel Cook. The chemistry of game design. Gamasutra, 07 2007. URL http://www.gamasutra.com/view/feature/1524/the_chemistry_of_game_design.php?print=1. Accessed: 24/12/2014.

[13] Corinna Cortes and Vladimir Vapnik. Support vector networks. *Machine Learning*, 20:273–297, 1995.

[14] Nello Cristianini and John Shawe-Taylor. *An introduction to support vector machines and other kernel-based learning methods*. Cambridge university press, 2000.

[15] Gustavo Danzi de Andrade, Hugo Pimentel Santana, André Wilson Brotto Furtado, André Roberto Gouveia do Amaral Leitão, and Geber Lisboa Ramalho. Online adaptation of computer games agents: A reinforcement learning approach. *II Workshop de Jogos e Entretenimento Digital*, pages 105–112, 2003.

[16] Janez Demšar. Statistical comparisons of classifiers over multiple data sets. *Journal of Machine Learning Research*, 7(1), 2006.

[17] Pierre A. Devijver and Josef Kittler. *Pattern recognition: A statistical approach*, volume 761. Prentice-Hall London, 1982.

[18] Robert P. Dilworth. A decomposition theorem for partially ordered sets. *Annals of Mathematics*, 51(1):161–166, January 1950.

[19] Anders Drachen, Alessandro Canossa, and Georgios N. Yannakakis. Player modeling using self-organization in Tomb Raider: Underworld. *IEEE Symposium on Computational Intelligence and Games*, pages 1–8, September 2009. doi: 10.1109/CIG.2009.5286500. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5286500.

[20] Harris Drucker, Chris J. C. Burges, Linda Kaufman, Alex J. Smola, and Vladimir Vapnik. Support vector regression machines. In M. C. Mozer, M. I. Jordan, and T. Petsche, editors, *Advances in Neural Information Processing Systems 9*, pages 155–161, Cambridge, MA, 1997. MIT Press.

[21] Kai-Bo Duan and S. Sathiya Keerthi. Which is the best multiclass SVM method? An empirical study. In Nikunj C. Oza, Robi Polikar, Josef Kittler, and Fabio Roli, editors, *Multiple Classifier Systems*, volume 3541 of *Lecture Notes in Computer Science*,

pages 278–285. Springer Berlin Heidelberg, 2005. ISBN 978-3-540-26306-7. doi: 10.1007/11494683_28.

[22] T. Edwards, D. S. W. Tansley, R. J. Frank, and N. Davey. Traffic trends analysis using neural networks. *Proceedings of the International Workshop on Applications of Neural Networks to Telecommuncations*, pages 157–164, 1997.

[23] José María Font, Tobias Mahlmann, Daniel Manrique, and Julian Togelius. Towards the automatic generation of card games through grammar-guided genetic programming. In *Proceedings of International Conference on the Foundations of Digital Games, Chania, Crete, Greece, May 14-17, 2013*, pages 360–363. Society for the Advancement of the Science of Digital Games, 2013.

[24] R. J. Frank, N. Davey, and S. P. Hunt. Time series prediction and neural networks. *Journal of Intelligent & Robotic Systems*, 31(1-3): 91–103, 2001. ISSN 0921-0296. doi: 10.1023/A:1012074215150.

[25] Yoav Freund and Robert E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55(1):119–139, August 1997.

[26] Drew Fudenberg and Jean Tirole. *Game Theory*. MIT Press, 1991. ISBN 9780262061414. URL http://books.google.de/books?id=pFPHKwXro3QC.

[27] Rockstar Games. Information about difficulty settings in Max Payne 3. http://support.rockstargames.com/entries/21456896-information-about-difficulty-settings-in-max-payne-3, 2012. Accessed: 14/12/2014.

[28] Thomas Gärtner and Gemma C. Garriga. The cost of learning directed cuts. In *Proceedings of the 18th European Conference on Machine Learning*, 2007.

[29] Kiel Mark Gilleade, Alan Dix, and Jen Allanson. Affective videogames and modes of affective gaming: Assist me, challenge me, emote me. In *Proceedings of Digital Games Research Association*, volume 2005. Citeseer, 2005.

[30] God Hand. [PlayStation 2]. Japan, North America: Capcom, 2006.

[31] Arthur Gretton, Arnaud Doucet, Ralf Herbrich, Peter J. W. Rayner, and Bernhard Schölkopf. Support vector regression for black-box system identification. In *Proceedings of the 11th IEEE Signal Processing Workshop on Statistical Signal Processing*, 2001.

[32] Half-Life 2. [Download]. USA: Valve Corporation, 2004.

[33] Zahid Halim and A. Rauf Baig. Automatic generation of interesting games. In *Proceedings of the 2nd International Conference on Computer Science and its Applications 2009*, pages 1–6. IEEE, 2009. ISBN 9781424449453. doi: 10.1109/CSA.2009.5404283. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5404283.

[34] John A. Hartigan and Manchek A. Wong. Algorithm AS 136: A k-means clustering algorithm. *Applied statistics*, pages 100–108, 1979.

[35] Guy Hawkins, Keith Nesbitt, and Scott Brown. Dynamic difficulty balancing for cautious players and risk takers. *International Journal of Computer Games Technology*, 2012:1–10, 2012. ISSN 1687-7047. doi: 10.1155/2012/625476. URL http://www.hindawi.com/journals/ijcgt/2012/625476/.

[36] Heiko Helble and Stephen Cameron. 3-d path planning and target trajectory prediction for the oxford aerial tracking system. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 1042–1048. IEEE, 2007. doi: 10.1109/ROBOT.2007.363122.

[37] Mark Hendrikx, Sebastiaan Meijer, Joeri van der Velden, and Alexandru Iosup. Procedural content generation for games: A survey. *ACM Transactions on Multimedia Computing, Communications and Applications*, 9(1), 2011. URL http://www.st.ewi.tudelft.nl/~iosup/pcg-g-survey11tomccap_rev_sub.pdf.

[38] Ralf Herbrich, Tom Minka, and Thore Graepel. TrueSkill™: A Bayesian skill rating system. In B. Schölkopf, J. Platt, and T. Hoffman, editors, *Advances in Neural Information Processing Systems 19*, pages 569–576. MIT Press, Cambridge, MA, 2006.

[39] Vincent Hom and Joe Marks. Automatic design of balanced board games. In *AI Interactive Digital Entertainment (AIIDE 2007)*, pages 25–30, Stanford, California, 2007. URL https://www.aaai.org/Papers/AIIDE/2007/AIIDE07-005.pdf.

[40] Ryan Houle and Henke Associates. Player modeling for adaptive games. In *AI Game Programming Wisdom 2*, chapter 10, pages 557–566. Charles River Media, 2003. ISBN 1584502894.

[41] Robin Hunicke and Vernell Chapman. AI for dynamic difficulty adjustment in games. *Proceedings of the Challenges in Game AI Workshop, 19th National Conference on Artificial Intelligence*, 2004.

[42] Laurentiu Ilici, Jiao Jian Wang, Olana Missura, and Thomas Gärtner. Dynamic difficulty for checkers and Chinese chess.

In *Proceedings of the 2012 IEEE Conference on Computational Intelligence and Games*, pages 55–62. IEEE, September 2012. doi: 10.1109/CIG.2012.6374138.

[43] Fujisankei Communications International. Zanac instruction manual. NES-ZA-USA, 1986.

[44] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement learning: A survey. *Computing Research Repository*, cs.AI/9605103, 1996. URL http://arxiv.org/abs/cs.AI/9605103.

[45] Yilin Kang and Ah-Hwee Tan. Learning personal agents with adaptive player modeling in virtual worlds. In *Proceedings of the 2010 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology*, pages 173–180. IEEE, August 2010. ISBN 978-1-4244-8482-9. doi: 10.1109/WI-IAT.2010.201. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5615251.

[46] Jack Kerouac. *On the Road*. Viking Press, 1957.

[47] Jack Kerouac. *The Dharma Bums*. Penguin UK, 2000.

[48] Donald Ervin Knuth. *Selected papers on analysis of algorithms*. CSLI, 2000.

[49] Teuvo Kohonen. *Self-organizing maps*, volume 30. Springer, 2001.

[50] Left 4 Dead. [CD-ROM], USA: Valve Corporation, 2008.

[51] Antonios Liapis, Héctor P. Martínez, Julian Togelius, and Georgios N. Yannakakis. Adaptive game level creation through rank-based interactive evolution. In *Proceedings of the IEEE Conference on Computational Intelligence and Games (CIG)*, pages 71–78, 2013.

[52] Nick Littlestone. Learning quickly when irrelevant attributes abound: A new linear-threshold algorithm. *Machine Learning*, pages 285–318, 1988.

[53] Changchun Liu, Pramila Agrawal, and Nilanjan Sarkar. Maintaining optimal challenge in computer games through real-time physiological feedback mechanical engineering. In *Proceedings of the 11th Human Computer Interaction International Conference*, 2005.

[54] Changchun Liu, Pramila Agrawal, Nilanjan Sarkar, and Shuo Chen. Dynamic difficulty adjustment in computer games through real-time anxiety-based affective feedback. *International Journal of Human-Computer Interaction*, 25(6):506–529, 2009. doi: 10.1080/10447310902963944.

[55] Mario Kart. [Super Nintendo]. Japan: Nintendo, 1992.

[56] Max Payne. [CD-ROM]. USA: Gathering of Developers, 2001.

[57] Olana Missura. Adaptive agents in the context of connect four. In *LWA 2007: Lernen - Wissen - Adaption, Halle, Workshop Proceedings*, pages 165–166. Martin-Luther-University Halle-Wittenberg, 2007.

[58] Olana Missura and Thomas Gärtner. Online adaptive agent for connect four. In *Proceedings of the Fourth International Conference on Games Research and Development CyberGames 2008*, pages 1–8, 2008.

[59] Olana Missura and Thomas Gärtner. Player modeling for intelligent difficulty adjustment. In *Proceedings of Discovery Science, 12th International Conference, Porto, Portugal*, volume 5808 of *Lecture Notes in Computer Science*, pages 197–211. Springer, 2009. doi: 10.1007/978-3-642-04747-3_17.

[60] Olana Missura and Thomas Gärtner. Classification on graphs for dynamic difficulty adjustment. In H. Blockeel, K. Borgwardt, and X. Yan, editors, *7th International Workshop on Mining and Learning with Graphs, Leuven, Belgium, Extended Abstracts*, 2009.

[61] Olana Missura and Thomas Gärtner. Predicting dynamic difficulty. In J. Shawe–Taylor, R. S. Zemel, P. Bartlett, F. C. N. Pereira, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 24*, pages 2007–2015, 2011.

[62] Martin Mladenov and Olana Missura. Offline learning for online difficulty adjustment. In *Proceedings of MLG 2010: the International Workshop on Machine Learning and Games, colocated with ICML 2010, Haifa, Israel*, 2010.

[63] David J. Montana and Lawrence Davis. Training feedforward neural networks using genetic algorithms. In *Proceedings of the International Joint Conferences on Artificial Intelligence*, volume 89, pages 762–767, 1989.

[64] Peter Olafson. *Elder Scrolls IV: Oblivion Official Game Guide*. Prima Games; Revised & expanded edition, 2006.

[65] Rosalind W. Picard. *Affective computing*. MIT Press, Cambridge, MA, USA, 1997. ISBN 0-262-16170-2.

[66] Michael K. Pitt and Neil Shephard. Filtering via simulation: Auxiliary particle filters. *Journal of the American Statistical Association*, 94(446):590–599, 1999. URL http://home.gwu.edu/~stroud/classics/PittShephard99.pdf.

[67] Marc J. V. Ponsen, Geert Gerritsen, and Guillaume Chaslot. Integrating opponent models with Monte-Carlo tree search in poker. In *Interactive Decision Theory and Game Theory*, 2010. URL http://aaai.org/ocs/index.php/WS/AAAIW10/paper/view/1984/2462.

[68] Resident Evil 4. [DVD-ROM]. USA: Capcom, 2005.

[69] Ryan Michael Rifkin. *Everything Old is New Again: A Fresh Look at Historical Approaches to Machine Learning*. PhD thesis, MIT, 2002.

[70] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, 2nd edition, 2003.

[71] Katie Salen and Eric Zimmerman, editors. *The Game Design Reader: A Rules of Play Anthology*. MIT Press, 2005. ISBN 0262195364, 9780262195362.

[72] Frederik Schadd, Sander Bakkes, and Pieter Spronck. Opponent modeling in real-time strategy games. In *GAMEON*, pages 61–70, 2007. URL http://ticc.uvt.nl/~pspronck/pubs/SchaddBakkesSpronckGAMEON07.pdf.

[73] Jonathan Schaeffer. *One jump ahead: computer perfection at checkers*. Springer, 2008.

[74] Jonathan Schaeffer, Neil Burch, Yngvi Björnsson, Akihiro Kishimoto, Martin Müller, Robert Lake, Paul Lu, and Steve Sutphen. Checkers is solved. *Science*, 317(5844):1518–1522, 2007. doi: 10.1126/science.1144079. URL http://www.sciencemag.org/content/317/5844/1518.abstract.

[75] Bernhard Schölkopf and Alex J. Smola. *Learning with Kernels*. MIT Press, Cambridge, MA, 2002.

[76] Bernhard Schölkopf, Kah-Kay Sung, Christopher J. C. Burges, Federico Girosi, Partha Niyogi, Tomaso Poggio, and Vladimir Vapnik. Comparing support vector machines with Gaussian kernels to radial basis function classifiers. *IEEE Transactions on Signal Processing*, 45(11):2758–2765, 1997.

[77] Bernhard Schölkopf, Ralf Herbrich, and Alex J. Smola. A generalized representer theorem. In David Helmbold and Bob Williamson, editors, *Computational Learning Theory*, volume 2111 of *Lecture Notes in Computer Science*, pages 416–426. Springer Berlin Heidelberg, 2001. ISBN 978-3-540-42343-0. doi: 10.1007/3-540-44581-1_27.

[78] Noor Shaker, Georgios N. Yannakakis, and Julian Togelius. Towards automatic personalized content generation for platform

games. In *Proceedings of Artificial Intelligence and Interactive Digital Entertainment (AIIDE 10)*, 2010. URL http://aaai.org/ocs/index.php/AIIDE/AIIDE10/paper/view/2135.

[79] Noor Shaker, Georgios N. Yannakakis, and Julian Togelius. Towards player-driven procedural content generation. In *Proceedings of the 9th Conference on Computing Frontiers*, CF '12, pages 237–240, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1215-8. doi: 10.1145/2212908.2212942. URL http://doi.acm.org/10.1145/2212908.2212942.

[80] Jonas Sjöberg, Qinghua Zhang, Lennart Ljung, Albert Benveniste, Bernard Deylon, Håkan Hjalmarsson, and Anatoli Juditsky. Nonlinear black-box modeling in system identification: A unified overview. *Automatica*, 31:1691–1724, 1995.

[81] Pieter Spronck, Ida Sprinkhuizen-Kuyper, and Eric Postma. Online adaptation of game opponent AI in simulation and in practice. In *Proceedings of the 4th International Conference on Intelligent Games and Simulation*, pages 93–100, 2003.

[82] Pieter Spronck, Marc Ponsen, Ida Sprinkhuizen-Kuyper, and Eric Postma. Adaptive game AI with dynamic scripting. *Machine Learning*, 63(3):217–248, 2006.

[83] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1998. URL citeseer.ist.psu.edu/sutton98reinforcement.html.

[84] Penelope Sweetser and Peta Wyeth. GameFlow: A model for evaluating player enjoyment in games. *Computers in Entertainment (CIE)*, 3(3):3, 2005.

[85] Istvan Szita, Marc J. V. Ponsen, and Pieter Spronck. Effective and diverse adaptive game AI. *IEEE Transactions on Computational Intelligence and AI in Games*, 1(1):16–27, 2009.

[86] Francis E. H. Tay and Lijuan Cao. Application of support vector machines in financial time series forecasting. *Omega*, 29(4):309 – 317, 2001. ISSN 0305-0483. doi: 10.1016/S0305-0483(01)00026-3. URL http://www.sciencedirect.com/science/article/B6VC4-43CH9VY-2/2/bd7990e5f3c02971f2e5310db0cbd48c.

[87] The Elder Scrolls IV: Oblivion. [DVD-ROM]. USA: 2K Games, 2006.

[88] Lucas M. Thomas. Zanac review: Ask your doctor if prescription Zanac is right for you. http://wii.ign.com/articles/839/839646p1.html, IGN, 2007. Accessed: 08/06/2008.

[89] Julian Togelius and Jürgen Schmidhuber. An experiment in automatic game design. In *Proceedings of the 2008 IEEE Symposium On Computational Intelligence and Games*, pages 111–118. IEEE, December 2008. ISBN 978-1-4244-2973-8. doi: 10.1109/CIG.2008.5035629. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5035629.

[90] Julian Togelius, Renzo De Nardi, and Simon M. Lucas. Making racing fun through player modeling and track evolution. In *SAB'06 Workshop on Adaptive Approaches for Optimizing Player Satisfaction in Computer and Physical Games*, pages 61–70, 2006.

[91] Tomb Raider: Underworld. [CD-ROM]. North America, Europe, Australia: Eidos Interactive, 2008.

[92] unangbangkay. Good idea, bad idea: Dynamic difficulty adjustment. http://www.destructoid.com/good-idea-bad-idea-dynamic-difficulty-adjustment-70591.phtml, 2008. Accessed: 14/12/2014.

[93] Tony Van Gestel, Johan A. K. Suykens, Dirk-Emma Baestaens, Annemie Lambrechts, Gert Lanckriet, Bruno Vandaele, Bart De Moor, and Joos Vandewalle. Financial time series prediction using least squares support vector machines within the evidence framework. *IEEE Transactions on Neural Networks*, 12(4): 809–821, 2001.

[94] Vladimir N. Vapnik and Alexey Ya. Chervonenkis. On the uniform convergence of relative frequencies of events to their probabilities. *Theory of Probability & its Applications*, 16(2):264–280, January 1971.

[95] Vladimir N. Vapnik and Alexey Ya. Chervonenkis. The necessary and sufficient conditions for consistency in the empirical risk minimization method. *Pattern Recognition and Image Analysis*, 1(3):283–305, 1991.

[96] Lev Semenovich Vygotsky. Play and its role in the mental development of the child. *Journal of Russian and East European Psychology*, 5(3):6–18, 1967.

[97] Jiao Wang, Tao Wang, Yan-hong Luo, and Xin-he Xu. Implementation of adaptive genetic algorithm of evaluation function in chinese chess computer game system. *Communications and Information Technology, 2005. ISCIT 2005. IEEE International Symposium*, 2005.

[98] Georgios N. Yannakakis and John Hallam. Evolving opponents for interesting interactive computer games. In *Proceedings of the 8th International Conference on Simulation of Adaptive Behavior*, pages 499–508. MIT Press, 2004. doi: 10.1.1.60.

6629. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.60.6629.

[99] Georgios N. Yannakakis and John Hallam. Towards optimizing entertainment in computer games. *Applied Artificial Intelligence*, 21(10):933–971, November 2007. ISSN 0883-9514. doi: 10.1080/08839510701527580. URL http://www.tandfonline.com/doi/abs/10.1080/08839510701527580.

[100] Georgios N. Yannakakis and John Hallam. Game and player feature selection for entertainment capture. In *IEEE Transactions on Computational Intelligence and AI in Games*, pages 244–251, 2007. URL http://ieeexplore.ieee.org/xpl/login.jsp?tp=&arnumber=4219050.

[101] Georgios N. Yannakakis and John Hallam. Real-time adaptation of augmented-reality games for optimizing player satisfaction. *IEEE Transactions on Computational Intelligence and AI in Games*, 1:121–133, 2009.

[102] T. Y. Zhang. Application of artificial intelligence in computer chinese chess. *M.Sc. thesis, Department of Electrical Engineering, National Taiwan University*, 1981.

COLOPHON

This thesis was typeset by the author using the LaTeX typesetting system originally developed by Leslie Lamport, based on TeX created by Donald Knuth, and the typographical look-and-feel `classicthesis` developed by André Miede. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*".

Final Version as of October 3, 2015 (`classicthesis` version x.y).