Department of Computer Science

Darmstadt University of Technology

An Intelligent Artificial Player for the Game of Risk

Diploma Thesis of Michael Wolf

Darmstadt, 2005

Für Carla

Abstract

Risk is a classical strategy board game and played in many countries around the world since 1959. Despite its popularity, the game lacks scientific attention and neither its complexity nor suitable artificial intelligence techniques are known. In this work a theoretical analysis of the *complexity* of Risk is presented. This analysis is completed by a test run measuring the complexities encountered in real games.

The design and implementation of a basic artificial player utilising a *linear evaluation function* with *handcrafted features* is presented. The deficits of this basic player are discussed and an enhanced player is introduced. This enhanced version of the player uses high-level goals and plans to overcome the drawbacks of the first one. It is shown that this enhanced player is capable of regularly defeating the average human novice and able to beat an experienced human player.

To be able to let the optimal feature weights of the evaluation function be determined automatically, *temporal difference learning* is applied to the enhanced player. Even though the increase in playing strength is only moderate, TD learning seems to be a promising technique for Risk.

The results of this work imply the presumption that dynamic, player-defined high-level goals in combination with temporal difference learning may yield a very strong artificial player.

Keywords: Risk, Artificial Intelligence, Game Playing, Linear Evaluation Function, Machine Learning, Reinforcement Learning, Temporal Difference Learning.

Zusammenfassung

Risiko ist ein klassisches Strategiespiel, das seit 1959 in vielen Ländern auf der Welt gespielt wird. Trotz seiner Beliebtheit fehlt dem Spiel die Aufmerksamkeit der Wissenschaft. Es ist weder die Komplexität von Risiko bekannt, noch wurden erfolgversprechende Techniken der Künstlichen Intelligenz veröffentlicht.

In dieser Arbeit wird eine Analyse der *Komplexität* von Risiko vorgestellt. Ein theoretischer Ansatz wird von einer Messreihe vervollständigt, bei der die Komplexität von realen Spielen gemessen wurde.

Das Design und die Implementierung eines einfachen künstlichen Spielers werden erläutert. Dieser einfache Spieler verwendet eine *lineare Evaluierungsfunktion* mit *handgefertigten Features*. Die Defizite dieses Spielers werden diskutiert und ein verbesserter Spieler vorgestellt. Diese verbesserte Version des Spielers verwendet hochstufige Zielvorgaben sowie Pläne um die Nachteile des ersten Spielers zu beheben. Es wird gezeigt, dass dieser verbesserter Spieler beim Spiel gegen menschliche Spieler in der Lage ist Anfänger regelmäßig zu besiegen es aber auch schafft gegen fortgeschrittene Spieler zu gewinnnen.

Um die optimalen Feature Gewichte der Evaluierungsfunktion automatisch bestimmen zu können, wird *Temporal Difference Lernen* verwendet. Auch wenn der Zugewinn an Spielstärke dadurch nur moderat war, so scheint TD Lernen doch eine erfolgversprechende Technik für Risiko zu sein.

Die Ergebnisse dieser Arbeit legen die Annahme nahe, dass die Kombination von dynamischen, Spieler-definierbaren hochstufigen Zielen mit Temporal Difference Lernen einen sehr starken künstlichen Spieler hervorbringen wird.

Stichwörter: Risiko, Künstliche Intelligenz, Spieltheorie, Lineare Evaluierungsfunktion, Machinelles Lernen, Reinforcement Learning, Temporal Difference Lernen.

Contents

List of Figures

List of Tables

List of Listings

Nomenclature

25P	25 Percentile.
75P	75 Percentile.
ABF	Average Branching Factor.
ACR	Adjusted Continent Rating.
AGT	Average number of game turns of a single game.
AGTC	Average Game-Tree Complexity.
AI	Artificial Intelligence.
AP	Actual Player.
AP	Australia Plan.
AT	Attack Tree.
AtPa	Attack Path.
Attack Path	A connected directed graph representing a series of battles. The nodes represent territories while the edges represent battles. Each node has to have a degree of 2.
Attack Tree	A tree representing a series of battles. The nodes represent territories while the edges represent battles from the parent nodes against the child nodes.
Battle	A series of attacks from one territory against another, continuing until either the attacked territory is captured or the attacker does not have enough armies left occupying the attacking territory to continue attacking.
Battle Tree	A tree where each node represents a battle and the edges represent attack outcomes.
Benchmark Player	The enhanced evaluation player with the Standard-Max reinforcement ditribution procedure.

BEP ..... Basic Evaluation Player.
BF ..... Branching Factor.
BP ..... Benchmark Player.
BT ..... Battle Tree.
CCP ..... Continent Conquering Plan.
Combat Move ..... The additional movement of armies from the attacking territory to the attacked territory occurring in case an attack leads to the occupation of the attacked territory.
CR ..... Continent Rating.
Decision-Type ..... The type of a decision, i.e. *Attack* decision, in contrast to a concrete decision occuring during a game.
EBR ..... Expected Battle Result.
ED ..... Expected battle result under the prerequisite of losing the battle.
EEP ..... Enhanced Evaluation Player.
Enemy Territory .... A territory not occupied by the actual player.
EP ..... Evaluation Player.
ER ..... Expected rating of a battle[1].
EV ..... Battle Result.
EV ..... Expected battle result under the prerequisite of winning the battle.
First Player ..... The player who starts the game.
Fortified Territory .. A territory which is occupied by more than one army.
Friendly Territory ... A territory occupied by the actual player.
Game Round ..... A game round starts with the beginning of the game turn of the first player and ends with the end of the game turn prior to the following game turn of the first player.
Game Turn ..... A game turn of a player starts when the last player's turn has ended and ends when he has fortified his position.
Hinterland Territory A territory which is not adjacent to any enemy territory.
ITF ..... Initial Test Factor.
MBF ..... Maximal Branching Factor.

[1]Note that battles are not rated directly by the evaluation function. Rather, one or more game states are created which are then rated by the evaluation function.

MSE ..... Mean-Squared Error.
PEP ..... Player Elimination Plan.
PF ..... Plan Factor.
PR ..... Player Rating.
RD ..... Reinforcement Distribution.
RF ..... Risk Factor.
RFW ..... Risk Framework.
SC ..... Success Chance.
Success Chance ..... The probability that a plan succeeds, i.e. all territories of the target list will be occupied.
TC ..... Target Continent.
TD ..... Temporal-Difference.
Victory Probability . The probability that a battle is victorious, i.e. the target territory will be occupied.
VP ..... Victory Probability.

1. Introduction

Risk is a popular board game played in many countries all around the globe. Despite its popularity and the various computer implementations of the game, there exists no scientific work discussing the techniques suitable for building a strong artificial Risk player. There also exists no article presenting computer implementations of the game.

This lack of scientific attention to such a widely known game is one reason why I have chosen to build an artificial player for Risk. A further reason is that the game is different from the classic games usually addressed in the field of scientific computer game playing research, thus making the creation of an artificial player for Risk more challenging and interesting.

In Risk, the player has to make several decisions in a single game turn, including the question whether to make another move or not. The number of players ranges from two to six, further differentiating Risk from classic two-player boardgames.

The game is presented in detail in Chapter 2, including the specification of the rules I used in my implementation of the game and some theoretical work on the complexity of Risk.

Chapters 3 to 6 present the practical part of my thesis, focusing on the design and architecture of the implementation. They also address the problems I encountered in the course of my work and the measures I took to solve them. To be able to start implementing a Risk player at all, a *framework* has to be built that contains the rules, the gameboard, a game manager and an interface for the players. It allows Risk to be played and is the basis for all further work. This framework is presented in Chapter 3.
The first player, the *basic evaluation player* is introduced in Chapter 4. It chooses its moves by utilising a linear evaluation function in combination with handcrafted features. Chapter 5 analyses several deficits of the basic evaluation player and introduces the *enhanced evaluation player*, which is built to overcome these flaws.
The last part of my implementation applies the technique of *TD Learning* to the en-

hanced evaluation player. This is done to be able to automatically determine the optimal feature weights of the evaluation function. This *learning player* is introduced in Chapter 6.

To provide reusability and flexibility the software is designed in a modular way. This allows to easily add new players, rule variants and game maps. The complete code is written in Java.

To be able to rate the playing skill of the players I performed several experiments setting the playing strength of the different players in relation to each other and several human players. I also measured the complexity of the games played in the Risk framework. The methods and results of these experiments are presented in detail in Chapter 7.

Finally, Chapter 8 offers a summary and discussion of this work.

2. Risk - The Game

2.1. Overview

Risk is a classical strategy board game for 2 to 6 players published by Parker Brothers in 1959. The game was originally invented in 1957 by Albert Lamorisse and called *La Conquete du Monde*. After having been adapted to satisfy *"american taste"*, it was published by Parker Brothers under the name of *Risk Continental Game* [Has, 1999]. In 1996 the game was renamed into *Risk: The Game of Global Domination* to better suit Parker Brothers' view of current world affairs. There are different version of the rules in various international editions of the game as well as updates to the original U.S. rules in 1963, 1965, 1980, 1993 and 1999. Furthermore there are two other games in the Risk series: *Risk 2210 A.D.* and *Risk The Lord of the Rings*. Since 1991 the rights to the game are owned by Hasbro Inc.

2.2. Equipment

2.2.1. Gameboard

The gameboard that I used in my work is the original Risk gameboard and consists of a map of the world divided into 6 continents and 42 territories.

Figure 2.1 shows a picture of the actual version of the Risk board game.

2.2.2. Risk Cards

For each of the territories there exists a Risk Card showing either an Artillery symbol, Cavalry symbol or an Infantry symbol. In addition to these cards there exist 2 Risk Cards, called Wild Cards, showing all three symbols at once. The Wild Cards are not linked to any territory.

Figure 2.1.: The Actual Version of the Risk Board Game [Has, 2004].

2.3. Rules

There exist many different official rule variants for the Risk board game [Lyne et al., 2004]. Not only have the rules been changed over the course of time, but there are also differences depending on the country of publication.
The rules that I used in my work are close to the German *Risiko de Luxe* rules [Has].

2.3.1. Object of the Game

The U.S. rules for the 1999 *Risk 40th anniversary Collector's Edition* [Has, 1999] summarise the object of the game as follows:

To conquer the world by occupying every territory on the board, thus eliminating all your opponents.

2.3.2. Game Setup

At the beginning of the game the territories are distributed randomly and equally among the players. All territories are distributed, even though that may result in some players owning one territory more than the other players. Each player places one of his armies on each of his territories. The Risk Cards are shuffled and the stack is placed upside-down next to the gameboard. The starting player is determined randomly.

2.3.3. Playing the Game

Risk is a turn based game where each player can perform his actions only during his turn. The game begins with the turn of the starting player. After he has finished his moves the next player's turn begins. If all players have completed their turns the first game round is over and the next one starts with the starting payer. This is repeated until all players except for one have been eliminated.[1] The remaining player has won the game.

During his turn a player can perform actions corresponding to the sequence:

Trading in Risk Cards - Placing new Armies - Attacking - Fortifying the Position.

Figure 2.2 shows an activity diagram of a game turn in Risk.

2.3.4. Trading in Risk Cards

A player can trade in three of his Risk Cards to gain additional reinforcement armies. The number of Risk Cards currently in his possession determine whether he is allowed to make a trade or whether he is forced to make a trade. If he has less then five Risk Cards in his possession he has a choice, otherwise he has to trade in a tradeable set of his Risk Cards.

A set of Risk Cards is tradeable if it contains three Risk Cards either each showing the same symbol or each showing a different symbol. Wild Cards are treated like regular Risk Cards showing a symbol of the player's choice.

[1] A player is eliminated if he does not own any territory anymore.

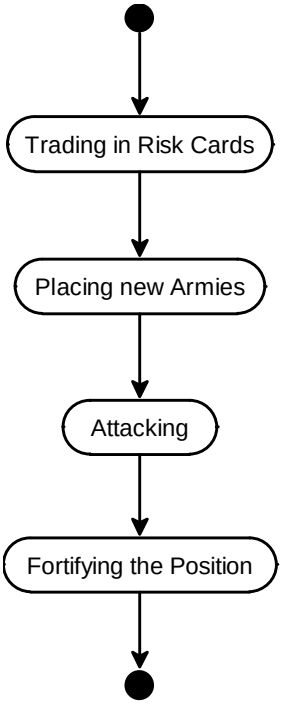

Figure 2.2.: Game Flow of a Game Turn

Trading in Risk Cards allows the player to place an extra set of new armies. As shown in table 2.1 the army bonus is determined by the symbols on the Risk Cards of the traded set. In addition to these armies, which are placed during the *Place new Armies* game phase, the player gets two new armies for each Risk Card of the set, whose corresponding territory is occupied by him. These new armies are placed immediately on the territory corresponding to the Risk Card.

Figure 2.3 shows an activity diagram of the *Trading in Risk Cards* game phase.

2.3.5. Placing new Armies

The number of armies that have to be reinforced is based on the number of territories the player occupies, the continents he controls and the army bonus from trading Risk Cards.

Symbol 1	Symbol 2	Symbol 3	Army Bonus
Infantry	Infantry	Infantry	4
Cavalry	Cavalry	Cavalry	6
Artillery	Artillery	Artillery	8
Infantry	Cavalry	Artillery	10

Table 2.1.: Army Bonuses for Valid Sets of Risk Cards

The total number of reinforcements is the sum of the reinforcements from territories, continents and Risk Cards.

$$\begin{aligned} \text{Total Reinforcements} \quad = \quad & \text{Territory Reinforcements} \; + \\ & \text{Continent Reinforcements} \; + \\ & \text{Trading Reinforcements} \end{aligned}$$

Territory Reinforcements For every three territories a player occupies, he has to reinforce one army. These reinforcements have a minimum of three, so that each player will at least get three reinforcement armies regardless of the number of territories he occupies.

$$\text{Territory Reinforcements} = \max\left(\left\lfloor \frac{\text{Occupied Territories}}{3} \right\rfloor, 3\right)$$

Continent Reinforcements For each continent which is completely occupied by the player[2], he receives additional reinforcement armies.

Table 2.2 shows the reinforcements for the continents of the original Risk gameboard.

Trading Reinforcements If the player has traded in a set of Risk Cards at the beginning of his turn, he gets the army bonus listed in table 2.1.

The reinforcement armies have to be placed on territories already occupied by the player. The player is not forced to place all of his reinforcements on the same territory; he may,

[2] A continent is completely occupied by a player if that player occupies every territory of the continent.

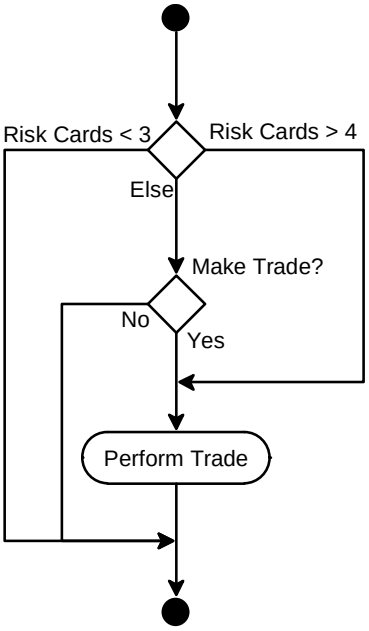

Figure 2.3.: Game Flow of the *Trading in Risk Cards* Game Phase

on the contrary, place each reinforcement army independently.

Figure 2.4 shows an activity diagram of the *Placing new Armies* game phase.

2.3.6. Attacking

A player may attack a territory which is occupied by another player from an adjacent territory occupied by him. The territory from where the attack is launched has to be occupied by a minimum of two armies. The attacking player chooses the number of attacking armies, which has to be a natural number of less than four. It is also limited by the number of armies occupying the territory from where the attack is launched minus one.

$$\text{Attacking Armies} \in [1, \min(\text{Armies in Attacking Territory} - 1, 3)]$$

Continent	Army Bonus
Australia	2
South America	2
Africa	3
Europe	5
North America	5
Asia	7

Table 2.2.: Army Bonuses for Completely Occupied Continents

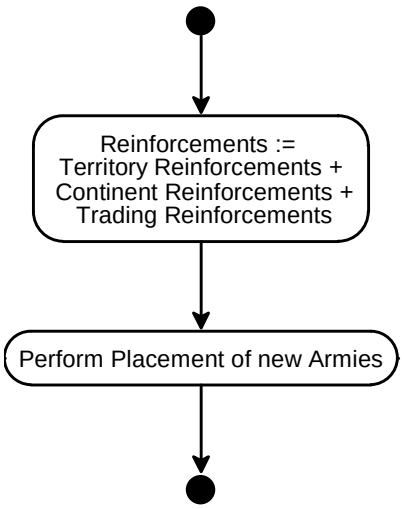

Figure 2.4.: Game Flow of the *Placing new Armies* Game Phase

The defending player has to choose[3] the number of defending armies, which has to be a natural number of less than three. It is also limited by the number of armies occupying the attacked territory.

$$\text{Defending Armies} \in [1, \min(\text{Armies in Defending Territory}, 2)]$$

[3]In some rule variants the defending player is allowed to choose the number of defending armies *after* the attacking player has rolled the dice for the attacking armies. It is important to note that in my implementation this is *not* the case. All the probabilities presented in Section 2.4 as well as the predication that the defender should always choose the maximum possible number of defending armies are based on this fact.

In the Risk framework the decision to choose the number of defending armies is omitted. The maximum possible number of defending armies is automatically chosen.

The battle is conducted by rolling a six-sided die for each participating army. The highest result of the attacking armies is compared to the highest result of the defending armies. If the attackers' result is higher the defender loses one army, otherwise the attacker loses one. If both sides rolled at least two dice the second-highest results of both sides are compared too, and the army losses are determined analogous. Armies that are lost are removed from the gameboard.

The player is not limited in the number of attacks he may launch. He may not attack at all or he might continue attacking as long as at least one of his territories occupied by at least two armies is adjacent to a territory held by another player.

Capturing a Territory

If the defender has lost all of his armies within the defending territory, the attacking player has captured that territory. He has to move the armies which participated in the attack from the attacking territory to the captured territory. Additionally, he may move more armies from the attacking territory to the captured territory[4]. The number of moving armies has to be natural and is limited by the number of armies occupying the attacking territory minus 1.

$$\text{Moving Armies} \in [\text{Attacking Armies}, \text{Armies in Attacking Territory} - 1]$$

If it is the first time during his current turn that the player captures a territory, he receives a Risk Card from the Risk Card stack. The player does not have to show the Risk Cards he holds to the other players. Therefore the players only know the number of Risk Cards held by the other players but neither the symbols nor the corresponding territories of these cards. If the defending player loses his last territory he is eliminated by the attacking player.

Eliminating a Player

A player is eliminated if he does not occupy any territory anymore. If a player is eliminated he does not participate in the game any longer. The Risk Cards he may have owned are given to the player who captured his last territory. If the attacking

[4] I call such a movement a *combat move*.

player now has a total of more than four Risk Cards he immediately has to trade in as many sets of Risk Cards as are necessary to reduce the number of his Risk Cards below five. The trade and the following placement of the reinforcement armies follow the rules of Section 2.3.4 and Section 2.3.5 respectively, with the exception that only the reinforcements from the traded Risk Cards are placed. After placing his reinforcement armies the attacking player continues his *Attacking* phase in the usual way.

Figure 2.5 shows an activity diagram of the *Attacking* game phase.

2.3.7. Fortifying the Position

After the player made the last attack of his current turn he may fortify his position. A player fortifies his position by moving armies from one of his occupied territories to an adjacent territory that is also occupied by him. The number of moving armies has to be natural and is limited by the number of armies occupying the territory that the armies are leaving minus 1.

$$\text{Moving Armies} \in [0, \text{Armies in Territory of Origin} - 1]$$

Figure 2.6 shows an activity diagram of the *Fortifying the Position* game phase.

2.4. Probabilities

When playing Risk it is very important to know the probabilities of the possible outcomes of an attack or a series of attacks as well as the expectation of the army losses of the attacker and defender. It is easy to compute that for both - attacker and defender - it is always favourable to use the maximum possible number of armies in the attack[5]. More participating armies always result in a higher victory probability and lower expected losses. While computing these numbers for a single attack is trivial, it is very complex to compute them for a longer series of attacks. Usually, the series of attacks most interesting to players are the continuing attacks from one territory against another, until either the territory is captured or the attacker does not have enough armies left occupying the attacking territory to continue attacking[6]. Players need to know the

[5] This is only true for the specific Rules presented here. There are rule variants where this predication does not hold.

[6] I call such a series of attacks a *battle*.

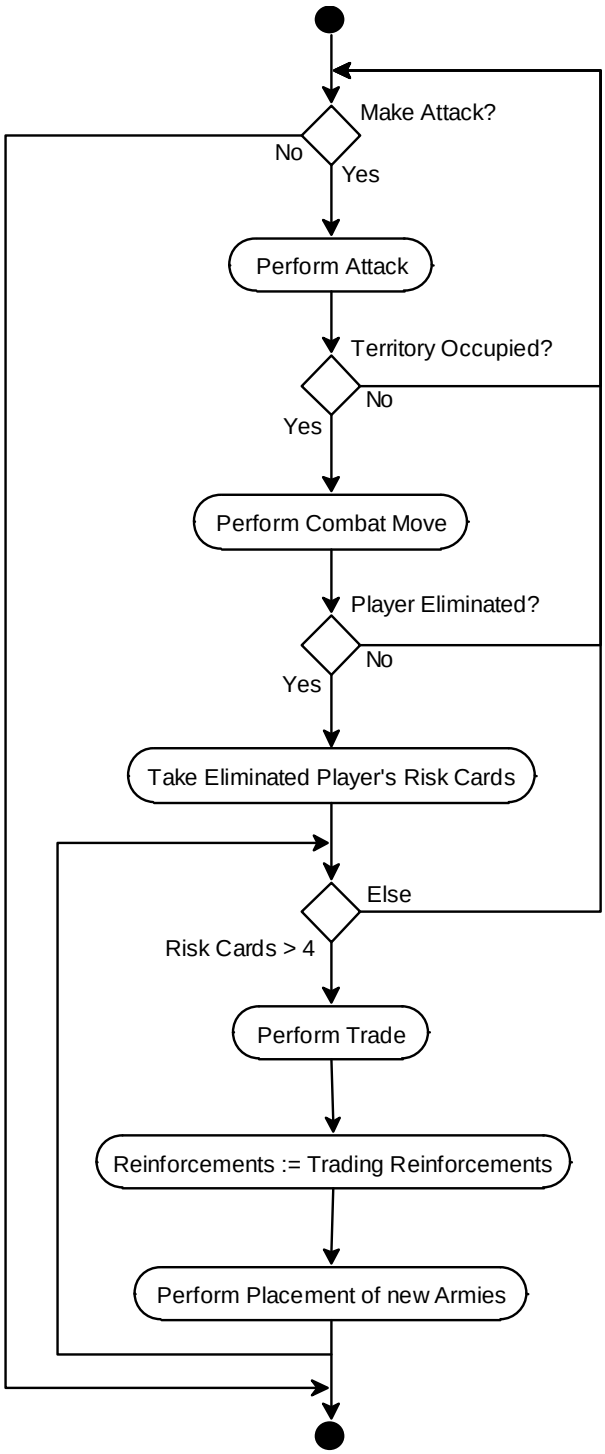

Figure 2.5.: Game Flow of the *Attacking* Game Phase

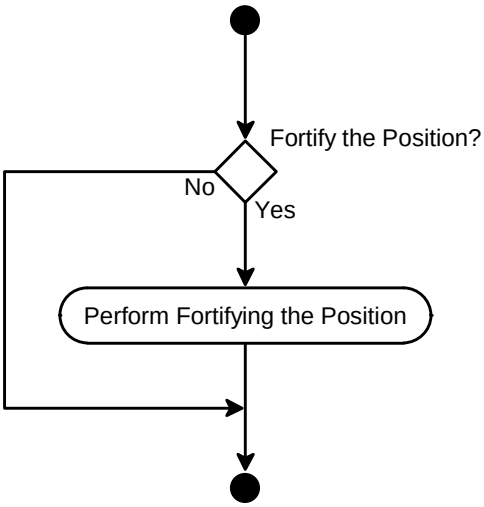

Figure 2.6.: Game Flow of the *Fortifying the Position* Game Phase

probability of capturing an enemy territory and the expectation of their army losses achieving this. Tan [Tan, 1997] first addressed this topic. In the case of rolling multiple dice for the attacker or defender, he did treat the dice independently. But because the results of the multiple dice rolls are ordered they are not independent anymore.

This issue was correctly addressed by Blatt and Osborne [Blatt, 2002, Osborne, 2003] who both showed a way of computing the victory probability and expected losses of a battle using Markov Chains. They compute a matrix S (Osborne calls the matrix F) which in turn is used to compute the probability of capturing a territory and the expected losses thereby.

The time complexity to compute these values is $O(\text{columns}(S))$ whereas the memory requirement for the matrix S is $O(\text{columns}(S) \times \text{rows}(S))$.

Let a denote the number of attacking armies, d the number of defending armies and S the minimal matrix needed to compute the victory probability $P(a, d)$ and the expected losses, then the number of rows in S is

$$\text{rows}(S) = a \times d$$

and the number of columns is

$$\text{columns}(S) = a + d$$

Armies		Result				
A	**D**	**A Loses 2**	**A Loses 1**	**Both Lose 1**	**D Loses 1**	**D Loses 2**
1	1	—	0.583	—	0.417	—
1	2	—	0.745	—	0.255	—
2	1	—	0.421	—	0.579	—
2	2	0.448	—	0.324	—	0.228
3	1	—	0.340	—	0.660	—
3	2	0.293	—	0.336	—	0.371

Table 2.3.: Probabilities of Attack Results in Risk [Lyne et al., 2004]. *A* and *D* are abbrevations for *Attacker* and *Defender* respectively.

Considering the number of armies participating in battles in the games played in the Risk framework a and d should at least number 100 armies each. Assuming 8 bytes of memory requirement for each entry in S[7] the memory requirement of S, can be computed.

$$\begin{aligned} \text{Memory}(S) &= \text{rows}(S) \times \text{columns}(S) \times \text{memory(number)} \\ &= (100 \times 100) \times (100 + 100) \times 8 \text{ bytes} \\ &= 10000 \times 200 \times 8 \text{ bytes} \\ &= 16 \text{ Megabytes} \end{aligned}$$

With that amount of memory needed to compute only a limited number of battle results, this approach is of no practical use. Instead, I chose to compute the relevant information (victory probability, expected losses in case attacker wins and expected losses in case defender wins) recursively. To reuse the results and save computation time I dynamically build a table storing the results of each computed battle situation. For detailed information see Section 3.7

Table 2.3 shows the different outcomes of single attacks and their corresponding probabilities.

[7] An entry in S consists of a floating-point number

2.5. Complexity

The complexity of a game can be measured by the *state-space complexity* and the *game-tree complexity*. The state-space complexity is a measurement of the number of different game states possible in a game. The game-tree complexity, on the other hand, is a measurement of the complexity of the decisions occurring in a game combined with the duration of the game measured in game turns. The game-tree complexity can be defined as [Wikipedia, 2004]:

> *The game-tree complexity is the number of possible different ways the game can be played*

The complexity of the decisions can be expressed in the *branching factor*.

In contrast to many other games, a game turn in Risk consists of a sequence of several different decisions. Even the number of decisions per game turn is neither fixed nor deterministic.

The following decisions occur while playing Risk:

Trade Cards In the trade cards decision the player is required to choose a tradeable set out of the Risk Cards in his possesion. He can also decide not to perform any trade if he owns less than five Risk Cards. If the player has less than three Risk Cards in his possesion, he cannot possibly have any tradeable set of Risk Cards and his only valid action would be not to trade. Therefore the decision occurs only if the player owns more than two Risk Cards.

Let P denote the number of players participating in the game, then

$$\text{Occurrence per Turn} \in \big[0, P+1\big]$$

Let tradeOptions(Set cardSet) be the function that returns the number of tradeable subsets of the set of Risk Cards cardSet and $CS_{\text{cardNumber}}$ denote the set containing all sets of Risk Cards with cardNumber Risk Cards, then

$$\text{Actions per Occurrence} \in \Big[1, \max_{cs \in CS_{10}} \big(\text{tradeOptions}(cs)\big) + 1\Big]$$

Place Armies In the place new armies decision the player is given the number of his reinforcement armies and is required to distribute these armies among his territories.

Let P denote the number of players participating in the game, then

$$\text{Occurrence per Turn} \in \big[1, P+1\big]$$

and

$$\begin{aligned}\text{Actions per Occurrence} &\in \left[1, \binom{41 + 35 + \min\left(20, (P-2) \times 20\right) - 1}{35 + \min\left(20, (P-2) \times 20\right)}\right] \\ &\in \left[1, \binom{75 + \min\left(20, (P-2) \times 20\right)}{35 + \min\left(20, (P-2) \times 20\right)}\right]\end{aligned}$$

Attack In the attack decision the player is required to choose a valid attack action or to finish his *Attacking* game phase. An attack action consists of the attacking territory, the target territory and the number of attacking armies.

It is easy to see that

$$\text{Occurrence per Turn} \in [1, \infty)$$

Let attackOptions(GameState state) be the function that returns the number of valid (attacking territory, target territory) pairs possible in the game state state. Let S denote the set containing all game states with two participating players and four armies occupying each territory, then

$$\text{Actions per Occurrence} \in \Big[0, \max_{s \in S}\big(\text{attackOptions}(s)\big) \times 3 + 1\Big]$$

Combat Move In the combat move decision the player is required to choose the number of additional armies moving from the attacking territory to the newly conquered territory. The decision occurs only if the player is able to move at least a single army.

It is easy to see that

$$\text{Occurrence per Turn} \in [0, \infty)$$

and

$$\text{Actions per Occurrence} \in [2, \infty)$$

Fortify Position In the fortify position decision the player is required to choose the number of armies moving from a friendly territory to an adjacent friendly territory.

It is easy to see that

$$\text{Occurrence per Turn} = 1$$

and

$$\text{Actions per Occurrence} \in [1, \infty)$$

As these theoretical values for the decision occurrences and valid actions are no help in getting an impression of the average complexity of a game of Risk, I measured the values occurring in a series of test games. Section 7.1 presents both the methods and results of the experiment in detail.

2.5.1. State-Space Complexity

The state-space of a game is the set of all legal board states. It is easy to see that the state-space of Risk is infinite.

Theorem:
The state-space of Risk is infinite.

Proof:
There is no limit on the number of armies in the game. Each turn a player has to reinforce at least three armies while no attack - the only action that removes armies from the gameboard - is enforced. As there is no limit on the number of armies in the game, the maximal possible number of armies occupying a single territory is unlimited too[8].

Let s be an arbitrary state, t an arbitrary territory and s^x the state that arises when placing x armies in state s on territory t, then all states $s^n, n \in \mathbb{N}$ are valid game states. The states are also distinct, because in each one a different number of armies is occupying territory t. We can deduce, therefore, that the state-space of Risk is (countably) infinite. In practice, there are rarely more than 1,000 armies on the gameboard in a game played in the Risk framework[9]. It is, therefore, possible to calculate the size of a limited state-space still big enough to be sufficient for almost all games played in the Risk framework. For simplicity's sake I completely ignored the Risk Card distribution among the players.

Let M denote the maximum number of armies on the gameboard, T the number of

[8] Assuming the number of territories is finite - which is naturally given in any playable Risk map

[9] The number is based on the results of the complexity measurement test run presented in Section 7.1. I have done no statistical analysis though, the number is solely based on my personal observation.

territories on the map and P the number of players participating in the game, then using the formula[10]

$$\begin{aligned}
\text{Game States}(M,P) &= \text{Army Distribution} \times \text{Territory Distribution} \\
&= \sum_{A=T}^{M} \binom{T+(A-T)-1}{(A-T)} \times \binom{P+T-1}{T} \\
&= \sum_{A=T}^{M} \binom{A-1}{A-T} \times \binom{P+T-1}{T}
\end{aligned}$$

we can calculate the size of the reduced state-space with a maximum of 1,000 armies on the gameboard and four players participating in the game[11].

$$\begin{aligned}
\text{Game States}(1,000,4) &= \sum_{A=42}^{1,000} \binom{A-1}{A-42} \times \binom{45}{42} \\
&= \sum_{A=42}^{1,000} \frac{(A-1)!}{((A-1)-(A-42))! \times (A-42)!} \times \frac{45!}{(45-42)! \times 42!} \\
&= \sum_{A=42}^{1,000} \frac{(A-1)!}{(42-1)! \times (A-42)!} \times \frac{45!}{3! \times 42!} \\
&= 2.972 \times 10^{74} \times 14,190 \\
&= 4.218 \times 10^{78} \\
&\approx 10^{78}
\end{aligned}$$

When humans play Risk far less armies are commonly used. In my personal experience I never observed more than 200 armies on the gameboard in any game. Using the same

[10] It should be noted that I also did not include the current player in the definition of a game state, therefore reducing the state-space to the space of the possible gameboard states. Of course, a complete game state specification would also need to encompass the distribution of the Risk Cards among the players as well as the definition of the current player.

[11] Assuming the classic Risk map with $T = 42$ is used.

formula we can calculate the complexity of the reduced state-space with a maximum of 200 armies on the gameboard and four players participating in the game[12].

$$
\begin{aligned}
\text{Game States}(200,4) &= \sum_{A=42}^{200} \binom{A-1}{A-42} \times \binom{45}{42} \\
&= \sum_{A=42}^{1{,}000} \frac{(A-1)!}{((A-1)-(A-42))! \times (A-42)!} \times \frac{45!}{(45-42)! \times 42!} \\
&= \sum_{A=42}^{200} \frac{(A-1)!}{(42-1)! \times (A-42)!} \times \frac{45!}{3! \times 42!} \\
&= 3.029 \times 10^{43} \times 14,190 \\
&= 4.298 \times 10^{47} \\
&\approx 10^{47}
\end{aligned}
$$

The game itself supplies approximately 200 army playing pieces per player[13], therefore a maximum of 800 armies could be on the gameboard in any four player game.

2.5.2. Branching Factor

The branching factor is a measurement of the complexity of the decisions of a game. That encompasses the frequency of the decisions per game turn as well as the number of valid actions for the decisions. I distuinguish three different branching factors:

Branching Factor The branching factor (BF) of a game state is the number of different game states[14] that can be reached from that state by the actions of a player in a single game turn. The branching factor of a game turn is exactly the complexity, i.e. the number of leaf nodes, of its *turn-tree*. The turn-tree is a tree whose nodes correspond with game states. The child nodes of a parent node are the nodes corresponding to all the game states that can be reached from that node during a decision in the game turn[15]. The root node is the node corresponding to the game state at the beginning of the game turn.

[12] Assuming the classic Risk map with $T = 42$ is used.

[13] I counted the playing pieces in a German Risk version sold in 2005.

[14] If a game state can be reached by several different sequences of actions, the game state is counted for each of them.

[15] It should be noted, that there may well be multiple nodes corresponding to the same game state.

Distribution	Min	25P	Median	75P	Max	Average
BF Approximation	7	10^6	10^{11}	10^{18}	10^{90}	10^{85}

Table 2.4.: Characteristics of the Distribution of the Expanded Turn-Tree Complexities

The branching factor of a game state is very hard to compute in Risk. Using the data of the complexity measurement (Section 7.1) we can compute approximations of branching factors of the game states encountered in the test run.

Let options(Decision$d$) be the function that returns the number of valid game states reachable by choosing a valid action of d[16] and D_t denote the set of all decisions occurring in game turn t, then the branching factor for t is approximated using the formula

$$BF_t \approx \prod_{d \in D_t} \text{options}(d)$$

By using this formula the path through the turn-tree actually taken by the player is expanded to form an approximation of the turn-tree. Every (virtual) path through this approximated turn-tree has exactly the same sequence of decisions and all decisions on the same level of the tree have the same number of valid actions. Of course, the expanded part of such an approximated turn-tree does not correlate with valid game states and actions anymore.

Figure 2.7 shows a histogram of the approximated branching factors calculated from the data of the complexity measurement while table 2.4 offers some characteristics of the distribution of the approximations.

Maximal Branching Factor The maximal branching factor (MBF) of a game is the maximum of the branching factors of all game states.

Theorem:
The maximal branching factor of Risk is infinite.

Proof:
For each branching factor $n \in \mathbb{N}$ it is easy to construct a game state with a branching factor $n' > n$. Let s be the game state with the branching factor n. Let

[16] In case of a decision with deterministic actions only, i.e. all decisions in Risk except the *Attack* decision, the number of reachable game states is equal to the number of valid actions.

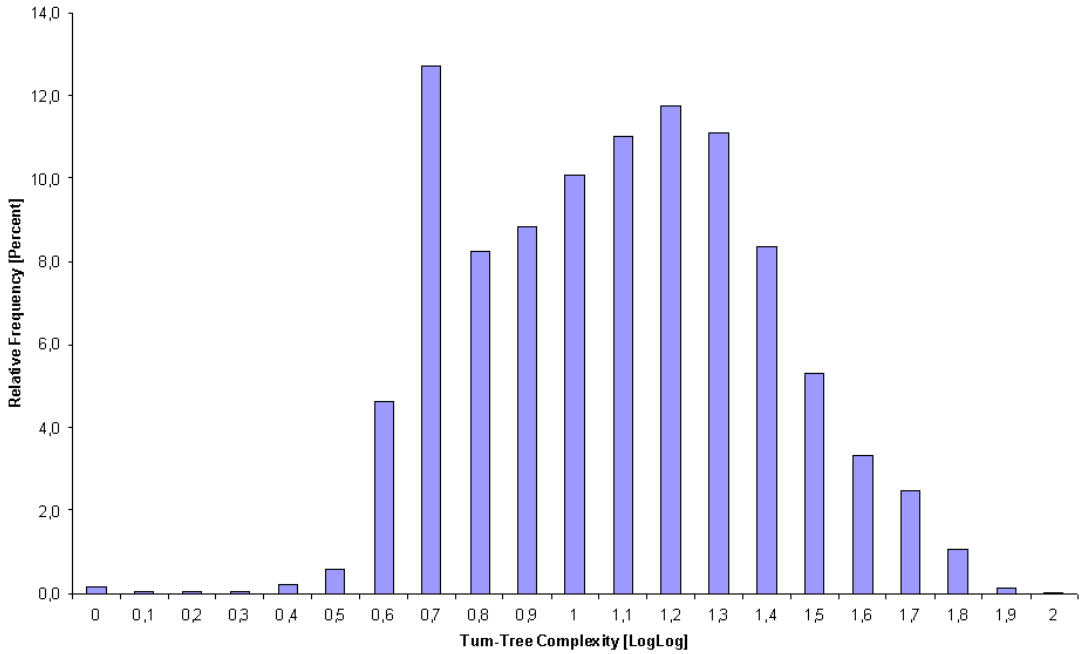

Figure 2.7.: Distribution of the Expanded Turn-Tree Complexities

s' be the state that is created by placing a single army on an arbitrary territory t occupied by the actual player, which is adjacent to a territory occupied by a different player. There exists such a territory, otherwise s would be a final state with a branching factor of zero.

The number of game states that can be reached in the *Trading in Risk Cards* and *Placing new Armies* game phases is equal for s and s'. In the *Attacking* game phase, however, the number of game states reachable from s' is greater than the number of game states reachable from s. By first attacking[17] an arbitrary territory from t with a single army and losing that army, we can reach all the states reachable by s. On the other hand, by not attacking at all, we have states unreachable from s.

We have shown that all the states reachable from s in the players turn can also be reached from s', but there exist states reachable from s' but not from s. Therefore the branching factor n' of s' is greater than n.

[17]This attack is possible because of the definition of t.

Of course, in each finite game the maximal branching factor is also finite.

Average Branching Factor The average branching factor (ABF) of a game is the arithmetic average of the branching factors of all game states. With no exact data on the individual branching factors of the game states and the huge number of game states it is not possible to compute the exact value. Because the average branching factor is used to compute the game-tree complexity of a game, it is important to get at least an approximation of the exact value.

I have computed two different approximations of the average branching factor:

Average-Decision The average-decision approximation makes use of the average frequency of the decisions and the average numbers of valid actions per occurrence of the decisions. These values are calculated from the data of the complexity measurement test run (Section 7.1).

Let options(Decision$d$) be the function that returns the number of valid game states reachable by choosing a valid action of d[18], frequency(Decision-Type$D$) the function that returns the occurrence per turn of D and $\varnothing$(Data data) the function that returns the arithmetic mean of data over the values measured in the complexity measurement. Let DT denote the set of all decision-types of the game, then

$$ABF = \prod_{D \in DT} \varnothing_{d \in D}\big(\text{options}(d)\big)^{\varnothing(\text{frequency}(D))}$$

The average branching factor computed with the average-decision approximation using the data of the complexity measurement is 6.133×10^{33}.

Average-Branching-Factor The average-branching-factor approximation uses the approximations of the branching factors of the game states and calculates the arithmetic mean over these approximations. The approximations themselves are computed and measured in the complexity measurement (Section 7.1).

Let $\varnothing$(Data data) be the function that returns the arithmetic mean of data. Let T denote the set of all measured game turns, then

$$ABF = \varnothing_{t \in T}\left(BF_t\right)$$

[18] In case of a decision with deterministic actions only, i.e. all decisions in Risk except the *Attack* decision, the number of reachable game states is equal to the number of valid actions.

Average Branching Factor	Game-Tree Complexity
Average-Decision Approximation	10^{2350}
Average-Branching-Factor Approximation	10^{5945}

Table 2.5.: Average Game-Tree Complexity for Different Average Branching Factors

The average branching factor computed with the average-branching-factor approximation using the data of the complexity measurement is 2.774×10^{85}.

2.5.3. Game-Tree Complexity

The game-tree is a tree whose nodes correspond with game states. The child nodes of a parent node are the nodes corresponding to all the game states that can be reached from that node during the player's turn[19]. The root node is the node corresponding to the game state at the beginning of the game. The game-tree is used, for example, in the widely known *minimax* or *expectimax* algorithms [Russell and Norvig, 1995].

The average game-tree complexity (AGTC) depends on the average branching factor (ABF) and the average number of game turns (AGT)[20]:

$$AGT = ABF^{AGT}$$

Table 2.5 shows the average game-tree complexities for Risk. The complexities are computed using the two average branching factors presented in Section 2.5.2 and the average number of game turns measured in the complexity measurement test run (Section 7.1).

2.5.4. Comparison with Classic Games

When comparing these results with other board game complexities, it is easy to see that Risk is far more complex than any of the traditional boardgames. Table 2.6 shows the state-space complexity and the game-tree complexity of several popular boardgames summarised by Jaap van den Herik [Jaap van den Herik et al., 2002] in comparison to Risk. Additionally the complexities of the restrained state-space Risk versions, as calculated in Section 2.5.1, are shown.

[19]It should be noted, that there may well be multiple nodes corresponding to the same game state.
[20]The complexity of the game-tree is the number of leaf nodes of the tree.

Game	State-Space Complexity	Game-Tree Complexity
Nine Men's Morris	10^{10}	10^{50}
Checkers	10^{18}	10^{31}
Othello	10^{28}	10^{58}
Chess	10^{46}	10^{123}
Risk (200 armies)	10^{47}	$10^{2350} \mid 10^{5945}$
Shogi	10^{71}	10^{226}
Risk (1000 armies)	10^{78}	$10^{2350} \mid 10^{5945}$
Go (19 x 19)	10^{172}	10^{360}
Risk	∞	$10^{2350} \mid 10^{5945}$

Table 2.6.: Complexities of Classic Boardgames and Risk

It is important to note, though, that the game-tree complexities of Risk are approximations of the average game-tree complexity, while the game-tree complexities of the classic games refer to the maximal size of the game-tree. The maximal game-tree complexity of Risk is infinite.

2.5.5. Adaptions of the Risk Framework

To be able to cope with the complexity of Risk, I simplified two decisions for the artificial players in the Risk Framework.

As shown in Section 2.4 it is always favourable[21] to choose the maximal possible number of attacking repectively defending armies in an attack. The artificial players automatically choose the maximal possible number of attacking armies while the number of defending armies is set to the maximal possible number for all players.

The very high branching factor of Risk is making it infeasible to build a game-tree and use a traditional minimax algorithm. Even evaluating and comparing the $\approx 10^{33}$ different game states of the first level of the game-tree is infeasible. By separating the different decisions and treating each one independently as well as splitting up the *Place new Armies* game phase into an independent decision for each reinforced army, I was

[21] In terms of victory probability and estimated losses.

able to cut down the number of evaluated game states to $\approx 10^6$ each turn[22]. This allows computing the decisions in a reasonable amount of time[23].

On the other hand, this approach also neglects all strategically important dependences between the different decisions. Even an average playing skill in Risk demands combining the decisions of at least the current game turn into an unified strategy. For example capturing a territory does not only require making the relevant attack decisions but also placing enough reinforcement armies on the territory from where the attack is launched. Chapter 5 describes in detail how this problem is at least partially solved.

2.6. Existing Implementations

There exist many computer implementations of the Risk board game. Some are freeware, some shareware and some are regular commercial products. The two most known versions are *Risk* and *Risk 2* by Hasbro Interactive. To the extend of my knowledge none of these implementations is open source nor is there any scientific publication addressing the artificial intelligence of computer implementations of Risk.

[22] See Section 4.3 for details.

[23] Both numbers are based on the average-decision approximation of the branching factor.

3. The Risk Framework

As the basis for all other work a framework is needed that allows Risk to be played and takes care of the rules, the gameboard and the flow of the game. This chapter describes the software architecture and the important classes of that framework.

3.1. Overview

The main components of the framework are the classes GameManager, Gameboard, Rules and GameMap. The class GameManager is the core of the architecture, it controls the flow of the game and checks whether the player's actions comply with the rules of the game. The class Gameboard is a representation of the gameboard as it can be perceived by the players at any given situation. It encompasses the distribution of the armies on the territories as well as the number of Risk Cards each player owns. The class Rules represents the part of the Risk rules that may change when playing different variants of the game. The core game concepts are not in this class but generally incorporated in the code at the relevant positions, because they are equal among all common rule variants. The class GameMap implements a graph representation of the map of the gameboard.

Figure 3.1 shows an informal representation of the core architecture of the Risk framework.

3.2. Game Manager

The main task of the class GameManager is to control the flow of the game. It also keeps track of the actual game by maintaining and updating a private gameboard. The game manager communicates with the players telling them to make the appropriate decisions. After receiving the chosen action the game manager checks it for validity and updates the gameboard accordingly.

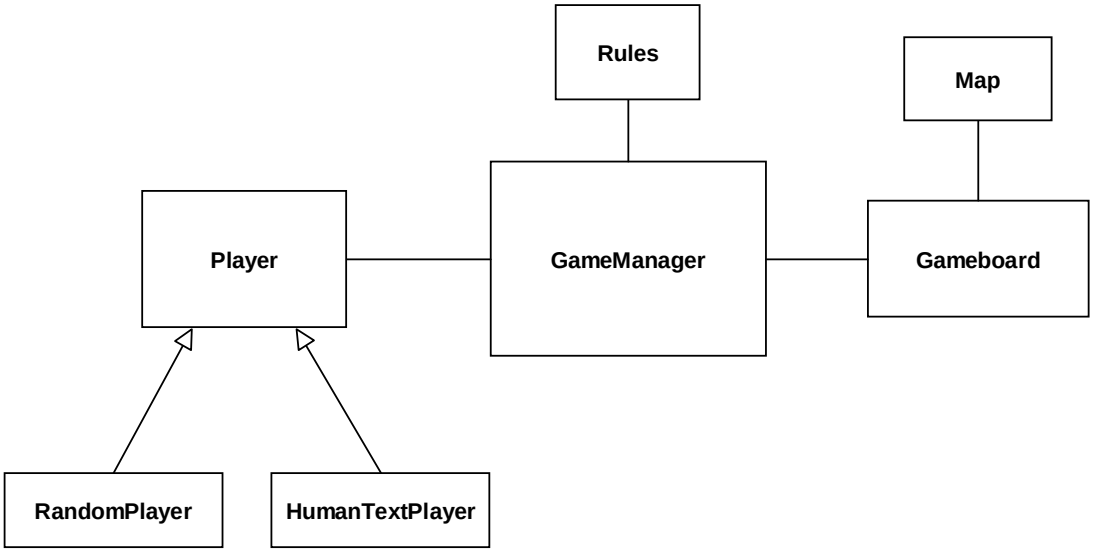

Figure 3.1.: The Core Architecture of the Risk Framework

Listing 3.1 shows a pseudocode representation of the main loop of the game flow, while listings 3.2 to 3.5 show the handling of the different decisions in more detail.

3.3. Gameboard

The class Gameboard keeps track of the part of the game state which is visible to the players. For each territory the class gameboard stores which player occupies it as well as the number of occupying armies. It also keeps track of the number of Risk Cards each player holds, it does not, however, store which Risk Cards are owned by the players.

The gameboard has several methods that, given a deterministic action or an attack result, will update the gameboard to the new situation. Additionally it offers several methods supplying information derived from the game situation. This includes, for example, the total number of armies and territories of each player as well as the number of armies of each player on a specific continent.

```
while not gameboard.gameIsOver() {
  if not gameboard.playerIsDefeated(actualPlayer) {
    processTrade()
    processReinforcement()
    processAttack()
    processMove()
  }
  actualPlayer ← nextPlayer()
}
```

Listing 3.1: The Main Loop of the Risk Framework

```
if gameboard.getNumberOfPlayerCards(actualPlayer) > 2 {
  chosenTrade ← actualPlayer.makeTrade()
  if checkTrade(rules,chosenTrade) {
    executeTrade(rules,chosenTrade)
  }
}
```

Listing 3.2: The processTrade Method of the Abstract Class Player

3.4. Rules

The class Rules is an abstract class that contains a set of abstract methods that all concrete subclasses have to implement. By providing a fixed interface this architecture allows a new Rule system to be added very easily.

It also provides methods for each decision that return all valid actions as well as methods that verify whether a given action is legal in a given game state. It thereby provides players with a set of legal actions to choose from and allows the game manager to easily check chosen actions for validity.

```
chosenReinforcement ← actualPlayer.distributeReinforcement()
checkReinforcement(rules, gameboard, chosenReinforcement)
executeReinforcement(rules, gameboard, chosenReinforcement)
```

Listing 3.3: The processReinforcements Method of the Abstract Class Player

```
do{
  chosenAttack ← actualPlayer.makeAttack()
  if checkAttack(rules, gameboard, chosenAttack) {
    executeAttack(rules, gameboard, chosenAttack)
    if gameboard.occupationOccurred and
    gameboard.combatMovePossible {
      chosenCombatMove ← actualPlayer.makeCombatMove()
      if checkCombatMove(rules, gameboard, chosenCombatMove) {
        executeCombatMove(rules, gameboard, chosenCombatMove)
      }
    }
  }
}while chosenAttack ≠ null
```

Listing 3.4: The processAttack Method of the Abstract Class Player

3.5. Map

The abstract class GameMap implements a graph representation of the map of the gameboard. By providing a fixed interface this architecture allows a new map to be added very easily.

The map can be interpreted as a graph where each node represents a territory and each edge a border between the corresponding territories of the nodes of the edge. The graph itself is implemented using an adjacency list representation which in turn is implemented by two parallel arrays (as thouroughly explained in [Cormen et al., 2001]).

Additionally the map stores the sets of territories that form the continents of the map, the reinforcement bonus of the continents, the symbols of the corresponding Risk Cards

```
chosenMove ← actualPlayer.makeMove()
if checkMove(rules,gameboard,chosenMove) {
  executeMove(rules,gameboard,chosenMove)
}
```

Listing 3.5: The processMove Method of the Abstract Class Player

of the territories, the number of Wild Cards and, of course, the names of the territories and continents.

The class also provides many useful graph level methods like computing the set of adjacent territories of a given territory, testing whether two given territories are adjacent or computing the distance of two given territories.

3.6. Player

The abstract class Player defines an interface between the game manager and the players. This architecture allows a new player to be added very easily.

The class offers some methods that form the core of the game manager to player communication. These methods are:

void startTurn(int gameRound) This method is called by the game manager at the start of a player's turn to indicate that his turn has just begun. The parameter of the method is the number of the actual game round. The player can use this information to initialize any calculation that is not depending on a specific decision but rather on the actual situation at the start of the turn. For example, players could calculate a turn specific goal that they may want to achieve during this turn or learning players could perform a learning step at the beginning of each game turn.

Trade makeTrade(int reinfSoFar, boolean tradeNecessary) This method is called by the game manager if a trade is theoretical possible, i.e. the player has at least three Risk Cards. It corresponds to the game phase *Trading in Risk Cards* (Section 2.3.4). However, it may still be that the player does not own a tradeable set of Risk Cards, in that case the player has to return *null*. The method has two parameters, first the number of reinforcements the player will receive without

performing a trade, and second whether the rules demand that he has to trade in a set of Risk Cards. Returning *null* is interpreted as not performing a trade. This is, of course, not legal when the second argument has the value *true*.

Reinforcement distributeReinforcements(int reinf) This method is called at least once every game turn and corresponds to the game phase *Placing new Armies* (Section 2.3.5). The parameter of the method is the number of reinforcements to distribute, while the return value is a reinforcement distribution.

Attack makeAttack() This method corresponds to the game phase *Attacking* (Section 2.3.6) and is only called during that game phase. The return value is the chosen attack. The method is called repeatedly until *null* is returned, which ends the attacking phase.

Move makeCombatMove(Move combatMove) This method is called whenever an attack leads to the occupation of the attacked territory and the attacking player has more than one army left in the attacking region[1]. The parameter is already a valid combat move, more precisely, it is the combat move with the maximal number of moving armies. This parameter is not necessary for the player to make his decision, but rather simplifies the processing of the decision for the player. The return value of the method is the chosen combat move. Returning *null* is treated by the game manager like returning a valid combat move with 0 moving armies.

Move makeMove() This method is called exactly once every game turn and corresponds to the game phase *Fortifying the Position* (Section 2.3.7). The return value is the chosen move.

void endTurn(int gameRound) This method is called by the game manager at the end of a player's turn to indicate that his turn is ending now. The parameter of the method is the number of the actual game round. The player can use this information to initialize any calculation that is not depending on a specific decision but rather on the actual situation after the player made his moves. For example learning players could perform a learning step at the end of each turn.

[1] After the armies participating in the last attack have been moved.

This abstract class acts as the interface between the game manager and various player implementations. The first two concrete player implementations are the *Random Player* and the *Human Text Player*.

3.6.1. Random Player

The class RandomPlayer implements a minimalistic artificial Risk player. As the name suggests, the random player completely lacks any systematic decision-making. The algorithm is simple and straightforward. For each decision, the player creates a list of valid actions and picks one of them randomly.

It is interesting to note that even though this player lacks any skill in playing Risk, a feature that is usually associated with a good playing skill turns up nevertheless. The territories occupied by random players will eventually form connected clusters which often contain hinterland territories. This can be explained by the fact that the chance of attacking an enemy territory grows linearly with the number of borders the random player already has to that territory. Usually territories close to a cluster of friendly territories will have more borders to friendly territories and, therefore, will be attacked and consequently conquered more often than far off territories with few borders to friendly territories.

3.6.2. Human Text Player

The class HumanTextPlayer implements a basic interface for a human to play Risk. The interface is text based and allows the human player to make his decisions by typing the appropriate numbers to the console. A prompt usually asks the human player to type the number of a territory or the number of armies depending on the current decision. The gameboard is not graphically shown but the user can always access a text description of the actual game situation.

For convenience, I recommend using a regular Risk gameboard to visualize the game situation.

3.7. Battle Computer

The class BattleComputer provides methods to predict the outcome of an attack or a series of attacks. The series has to be a battle, i.e. an attack from a single territory

against a single territory that has to end either when the attacked territory is conquered or the attacker does not have enough armies left to continue attacking. The interesting values are the victory probability of a battle[2] and the expected army losses of the attacker in case of a victory as well as the expected army losses of the defender in case of a defeat.

The prediction is calculated by recursively computing the results of the attacks. This is possible because the results of the first attack of a battle are always either a final state where the battle is either won or lost, or other battles with fewer armies participating. With a maximum of three different possible outcomes of an attack, computing the victory probability and the expected losses recursively creates a tree with a maximal branching factor of 3. Figure 3.2 shows an example of such a battle tree. The height of the tree is

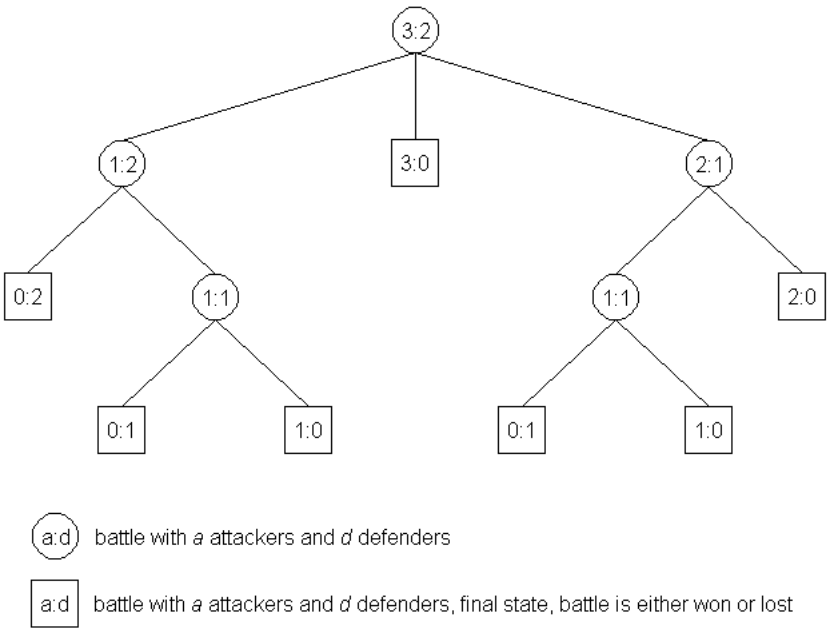

Figure 3.2.: Example Battle Tree

limited by the maximum of the participating armies of the attacker and defender. Thus we can compute an upper bound on the number of nodes of the battle tree.

Let a and d denote the number of attacking respectively defending armies in a battle, then

$$\text{Upper Bound} = 3^{\max(a,d)}$$

[2]The probability that the attacked territory will be conquered

Even if we limit the number of attacking and defending armies to 100 each, which would be sufficient for almost all battles fought in games played in the Risk framework, the upper bound of the battle tree would be $3^{100} = 5.154 \times 10^{47}$. With a function call for each node, this would result in a recursion tree of the same size, which, of course, is infeasible to compute during each battle.

Fortunately, we can cut down the number of actually needed recursive function calls to a very small fraction of this upper bound. This is possible because the outcome of an attack only depends on the number of attacking and defending armies. If a armies attack d armies only $a \times d$ different battles can occur in the battle tree. In the example of both 100 attacking and 100 defending armies, that would be 10^4 different battles; thus, on average, each unique battle is processed roughly 10^{43} times.

By storing the outcomes of already computed battles in a hashtable, the computation is reduced to calculating so far unencountered battle situations and looking up known results. The memory requirement is the space for $a \times d$ battle results with a memory requirement for the data of 24 bytes each. In the case of the 100×100 armies example, it amounts to 240 Kilobytes of memory and 10^4 computations in the case of a formerly empty hashtable. If the battle is encountered a second time, the battle computer just looks up the results[3] in the hashtable.

In the actual implementation the hashtable is initialized and dynamically filled with computed results every game anew. This drawback can easily be remedied by making the hashtable persistent.

Table 2.3 shows the different outcomes of single attacks and their corresponding probabilities.

[3] Computed and stored in the hashtable when the battle was encountered first.

4. Basic Evaluation Player

The basic evaluation player chooses its moves by utilising a linear evaluation function in combination with handcrafted features.

4.1. Overview

The basic evaluation player makes his decisions by choosing the move that will lead to the game state with the highest expected value. This is done independently for each decision and even for each reinforced army in the *Placing new Armies* game phase. That is, the player simulates all valid actions of the current decision, rates and compares the resulting game states with each other and picks the action that leads to the game state with the highest rating. For example, the evaluation player compares all different game states that arise when placing a single army in the *Placing new Armies* game phase, then chooses the territory yielding the highest rated game state, completely indifferent to the other decisions. While this does not restrict the accessible state-space it does, however, reduce the overall playing strength of the player. Chapter 5 presents the negative effects of this approach as well as several measures to cope with them.

To be successful the player needs to be able to predict the game state that will arise when applying any one action as well as a way to rate the different game states.
The prediction is easy for most of the possible actions, because they are deterministic and the player can correctly compute the resulting game state. In the case of an attack or a battle, however, the resulting game state is not deterministic, because it depends on the results of multiple dice rolls.

The rating is done by the *evaluation function*, which, given a gameboard and a player, will return the rating of the game state. To do this the evaluation function uses handcrafted features. The features are also functions that are given a game state and a player and return a feature value. The evaluation function returns the weighted sum of all the feature values, therefore it can be classified as a *linear* evaluation function.

Figure 4.1 shows an informal representation of the core architecture of the basic evaluation player.

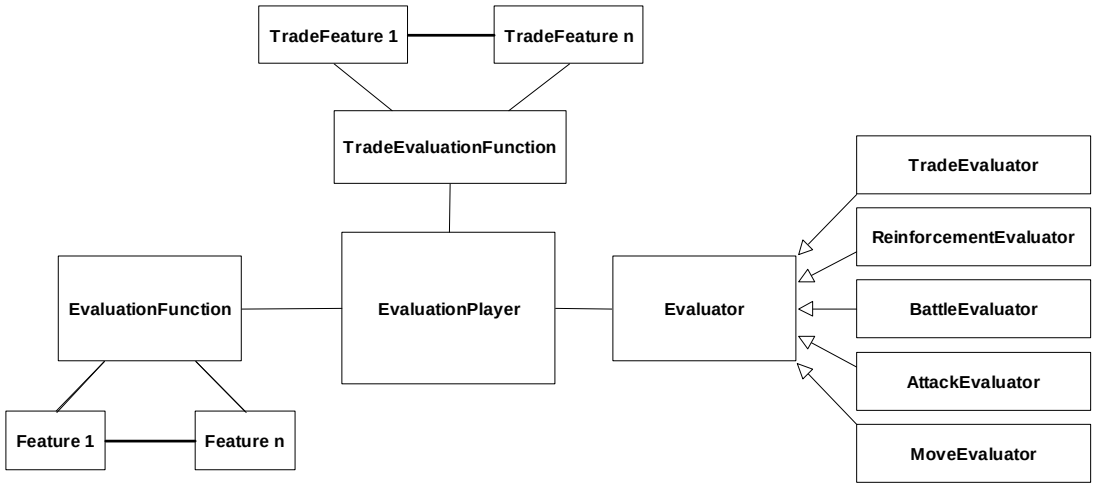

Figure 4.1.: The Core Architecture of the Basic Evaluation Player

4.2. Useful Functions

This class offers a few methods that are quite useful while trying to play Risk successfully. Among others these methods include a function calculating an estimation of the number of reinforcements for a given player, as well as a method computing the probability that a given territory will be conquered by other players if they try to do so.

It also offers a method that utilises an A*-Search [Russell and Norvig, 1995] to compute the attack path with the highest victory probability given a start territory and a destination territory. The method assumes that the start territory and the destination territory are occupied by different players. It returns an ordered list of territories (the attack path) as well as the probability that all the path territories will be successfully conquered by the player occupying the start territory.

4.2.1. Continent Rating

The continents in Risk have several characteristics which determine their usefulness for the players. In general, it is a good idea to conquer a continent, but the decision *which*

continent to conquer first is not as simple. The game state, of course, determines the general circumstances for all strategic decisions. But the characteristics of the continents themselves, regardless of the actual game state, must be considered too. The three distinct charatcteristics of the continents are the *army bonus*, the *border number* and the *territory number*.

Army Bonus The army bonus is the number of additional reinforcement armies which the player will get who completely occupies the continent. This is the sole reason why continents are important in Risk. In the original map the army bonuses of the continents are vital to winning the game.

Border Number The number of borders of a continent determine how easily it can be defended when it is completely occupied by a single player. Because the army bonus is so important, other players will surely try to break the complete occupation of the continent - and they have to come through the border territories. The more border territories a continent has, the harder it is to defend.

Territory Number The number of territories in the continent is the least important of the three characteristics. It determines the number of battles required to conquer it and thereby influences the number of armies needed to conquer it.

To be able to rate the continents I compute a rating from the three characteristics. This rating is computed with a general formula, instead of just manually assigning the six continents a rating. That approach simplifies the introduction of new maps to the Risk Framework.

$$\text{Continent Rating} = \frac{15 + \text{Army Bonus} - 4 \times \text{Border Number}}{\text{Territory Number}}$$

Table 4.1 shows the characteristics and ratings for the continents.

4.3. Decision Making Process

The basic evaluation player makes his decisions by rating and comparing an estimation of the game states that will probably result from his actions. The rating of the game states is done by the evaluation function, while the class EvaluationPlayer creates the expected game states and compares the ratings returned by the evaluation function. To be more precisely, the evaluation player creates a set of legal actions for the current

Continent	Army Bonus	Border Number	Territory Number	Rating
Australia	2	1	4	3.250
South America	2	2	4	2.250
North America	5	3	9	1.222
Africa	3	3	6	1.000
Europe	5	4	7	0.571
Asia	7	5	12	0.167

Table 4.1.: Characteristics and Ratings for the Continents

decision whose elements are then evaluated by the *Evaluators* which utilise the evaluation function for this task.

The *makeAttack* method is a little different than the other decision making methods. Actually, it utilises the chooseBest method twice. First, it chooses the best battle to be fought and, if a battle is chosen, it returns the attack corresponding to that battle. If no battle is chosen, it chooses and returns the best attack to be fought. That enables the player not only to launch attacks of worthwhile battles, but also to launch attacks where it is not advisable to continue fighting the whole battle.

Sections 4.3.1 and 4.3.2 describe this process in more detail while figure 4.2 shows a sequence diagram of the interactions of an arbitrary single decision. The method *makeDecision* can be substituted by any of the specific decision-methods introduced in Section 3.6, while *<Decision>Evaluator* is the approprioate Evaluator for the current decision as introduced in Section 4.3.2.

Listing 4.1 shows a pseudocode representation of such a *makeDecision* method in the general case, while listing 4.2 depicts the makeAttack method.

```
EvaluationPlayer::makeDecision(decision parameters){
  Actions ← createSetOfLegalActions()
  bestAction ← this.chooseBest(Actions, decisionEvaluator)
  return bestAction
}
```

Listing 4.1: Decision-Making Methods of the Evaluation Player

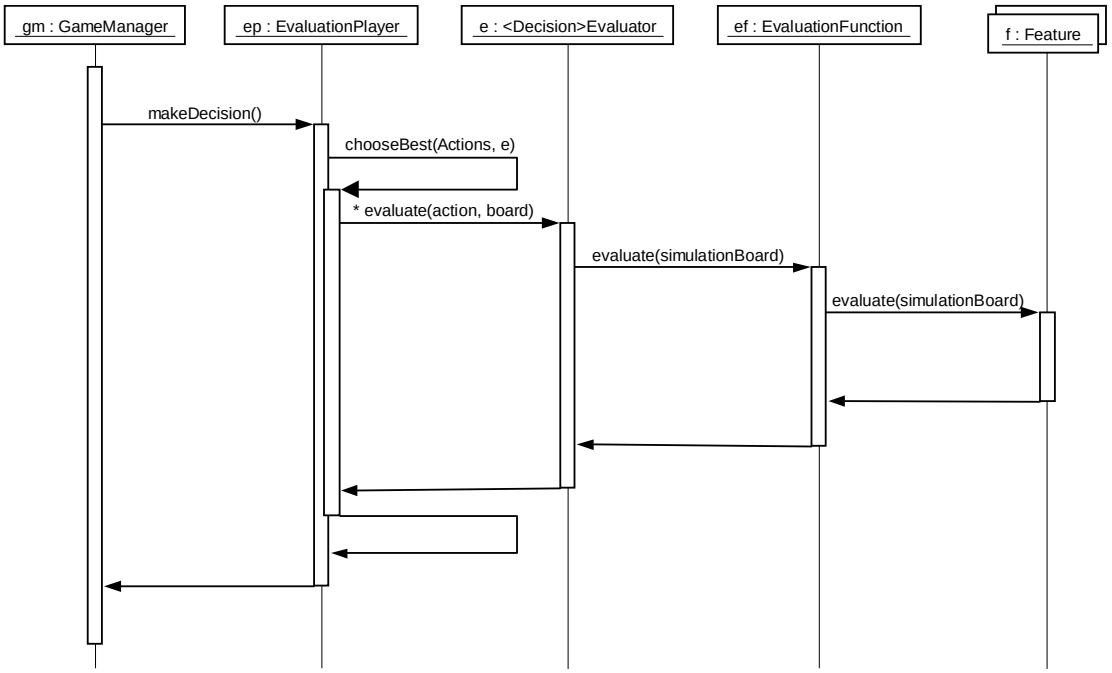

Figure 4.2.: Interactions of the Decision-Making Process of the Evaluation Player

```
EvaluationPlayer::makeAttack(){
  Battles ← createSetOfLegalBattles()
  bestBattle ← this.chooseBest(Battles, BattleEvaluator)
  bestAttack ← getCorrespondingAttack(bestBattle)
  if bestAttack = null {
    Attacks ← createSetOfLegalAttacks()
    bestAttack ← this.chooseBest(Attacks, attackEvaluator)
  }
  return bestAttack
}
```

Listing 4.2: *makeAttack* Method of the Evaluation Player

4.3.1. chooseBest

This method of the class EvaluationPlayer expects a set of possible actions as well as an appropriate Evaluator. The method is completely independent of any specific decision. The decision specific processing was done earlier when the method corresponding to the decision type created the set of legal actions and called the chooseBest method. There is also a specific evaluator for each decision that incorporates the information how to generate the expected game states as well as the information necessary to compute the expectation of the rating of the expected game state.

For each action of the given set, the method chooseBest calls the given evaluator's *evaluate* method, which in turn returns the expected rating of the game state that will probably result from this action. The method keeps track of the best rating so far computed for this decision and returns the best action of the set. In the case of several actions resulting in game states with equal expected ratings, the method chooses one of these actions randomly and equably.

Listing 4.3 shows a pseudocode representation of the chooseBest method.

```
EvaluationPlayer::chooseBest(Actions,evaluator){
  bestEvaluation ← −∞
  equalEvaluations ← 0
  for each action in Actions {
    actualEvaluation ← evaluator.evaluate(action,gameboard)
    if actualEvaluation > bestEvaluation {
      bestAction ← action
      bestEvaluation ← actualEvaluation
      equalEvaluations ← 1
    }
    if actualEvaluation = bestEvaluation {
      equalEvaluations++
      randomNumber ← createRandomNumber()
      if randomNumber × equalEvaluations < 1 {
        bestAction ← action
        bestEvaluation ← actualEvaluation
        equalEvaluations ← 1
```

```
        }
      }
    }
    return bestAction
}
```

Listing 4.3: The chooseBest Method of the Evaluation Player

4.3.2. Evaluators

There are different evaluators for the different decisions:

- Trade Evaluator
- Reinforcement Evaluator
- Attack Evaluator
- Battle Evaluator
- Move Evaluator

The evaluators are responsible for generating the expected game state resulting when applying an action on the current game state, thus simulating the execution of an action. They also use the evaluation function to compute the rating of the generated game states. For most decisions this is fairly easy, because the actions are deterministic and the evaluator just creates a copy of the actual game state and tells the gameboard to update itself with the actual action. This game state is then rated by the evaluation function and the result returned to the method *chooseBest*.

The only nondeterministic decision (i.e. the attack decision) is a little more complicated and the approach I chose is presented in the following subsections.

Listing 4.4 shows a pseudocode representation of the *evaluate* method of the evaluator in the general case.

```
Evaluator::evaluate(action,gameboard){
  simulation ← simulateActionExecution(action,gameboard)
  evaluation ← EvaluationFunction.evaluate(simulation)
```

```
  return evaluation
}
```

Listing 4.4: The evaluate Method of an Evaluator

Battle Evaluator

An attack is not deterministic, making it harder for the evaluator to predict the game state resulting from applying the attack action to the actual game state.

A straightforward approach would be to compute the expected result of the battle and rate that game state. However, even though this would be suitable for battles with a very high or low victory probability it is inadequate for uncertain battles. When computing the expectation of the battle outcome before rating the game state vital information could be lost. For example, if a battle is evaluated and the expected outcome is a loss and the rating of the resulting game state is worse than the rating of the current one, then the corresponding attack will not be launched. It might well be though, that there was a reasonable chance[1] to win the battle and that the rating of the resulting game state was very high. The straightforward player would almost never fight a battle with less than 50% victory probability, even if a victory would result in a significant gain for the player while a defeat could well be borne.

Let Eval(State gs) be the evaluation function[2], State(BattleResult br) the function that returns the game state incorporating the battle result br and P(BattleResult br) the function that returns the probability of br. Let BR(Battle b) be the function that returns the set of all possible results of b and $EBR(b)$ be the function that returns the expected battle result[3] of a given battle b. Then the expected rating (ER) of a given battle B is

$$\begin{aligned} ER(B) &= \left(\sum_{br \in BR(B)} P(br)\right) \times \text{Eval}\left(\text{State}\left(\sum_{br \in BR(B)} P(br) \times br\right)\right) \\ &= 1 \times \text{Eval}\Big(\text{State}\big(EBR(B)\big)\Big) \\ &= \text{Eval}\Big(\text{State}\big(EBR(B)\big)\Big) \end{aligned}$$

[1] Though less than 50%

[2] Ignoring that it also expects to be given a player as an additional argument.

[3] The EBR of a given battle could easily be provided by the battle computer (Section 3.7).

The solution to this drawback is to create all game states that can possibly arise in a given battle along with their probabilities. The different game states will be rated and afterwards multiplied with their probability, and then added up to the rating of the expected outcome of the battle. This approach, which applies the rating before calculating the expectation, will provide accurate ratings of battle outcomes. But it has its drawback, too. The sheer number of possible game states in bigger battles which have to be rated in combination with the fact that the rating of a game state and the calculation of the probability of the corresponding battle outcome requires a considerable amount of computation time makes this approach infeasible.

Let Eval(State gs) be the evaluation function[4], State(BattleResult br) the function that returns the game state incorporating the battle result br and P(BattleResult br) the function that returns the probability of br. Let BR(Battle b) be the function that returns the set of all possible results of b. Then the expected rating (ER) of a given battle B is

$$ER(B) = \sum_{br \in BR(B)} P(br) \times \text{Eval}\big(\text{State}(br)\big)$$

Fortunately, it is possible to get a meaningful ER with an acceptable amount of coputation time. I compute the victory probability of a battle and the expected battle outcome under the prerequisite that the attacker is victorious as well as the expected battle outcome under the prerequisite that the attacker suffers a defeat. The two game states corresponding to the battle outcomes will be rated and then multiplied by their probability and added up to get the expected rating of the outcome of the battle. To understand this, it is important to know that the part of the outcome of a battle which affects the territory distribution, i.e. victory or defeat, has a far greater impact on the rating of the game state than the part affecting the army distribution, i.e. army losses. Therefore, this approach requires only the computation of a single victory probability and two battle outcomes as well as two game state ratings while keeping the important differentiation whether the battle was victorious or not.

Let Eval(State gs) be the evaluation function[5], State(BattleResult br) the function that returns the game state incorporating the battle result br and P(BattleResult br) the function that returns the probability of br. Let V(Battle b) be the function that returns

[4]Ignoring that it also expects to be given a player as an additional argument.
[5]Ignoring that it also expects to be given a player as an additional argument.

the set of all victorious results of b and $D(\text{Battle } b)$ be the function that returns the set of all non-victorious results of b. Let $EV(b)$ be the function that returns the expected battle result of a given battle b only considering victorious battle results and $ED(b)$ be the function that returns the expected battle result of a given battle b only considering battle results leading to a defeat. Let $VP(\text{Battle } b)$ be the function that returns the victory probability of b[6]. Then the expected rating (ER) of a given battle B is

$$\begin{aligned} ER(B) &= \left(\sum_{v \in V(B)} P(v)\right) \times \text{Eval}\left(\text{State}\left(\sum_{v \in V(B)} P(v) \times v\right)\right) + \\ &\quad \left(\sum_{d \in D(B)} P(d)\right) \times \text{Eval}\left(\text{State}\left(\sum_{d \in D(B)} P(d) \times d\right)\right) \\ &= VP(B) \times \text{Eval}\Big(\text{State}\big(EV(B)\big)\Big) + \\ &\quad \big(1 - VP(B)\big) \times \text{Eval}\Big(\text{State}\big(ED(B)\big)\Big) \end{aligned}$$

Listing 4.5 shows a pseudocode representation of the *evaluate* method of the battle evaluator.

```
BattleEvaluator::evaluate(attack, gameboard){
  simVictory ← simulateVictory(attack, gameboard)
  simDefeat ← simulateDefeat(attack, gameboard)

  evaluationVictory ← evaluationFunction.evaluate(simVictory)
  evaluationDefeat ← evaluationFunction.evaluate(simDefeat)

  evaluationVictory ← evaluationVictory × getVictoryProb()
  evaluationDefeat ← evaluationDefeat × (1 − getVictoryProb())

  evaluation ← evaluationVictory + evaluationDefeat
  return evaluation
}
```

Listing 4.5: The evaluate Method of the Battle Evaluator

[6]The EV, ED and VP of a given battle are provided by the battle computer (Section 3.7).

Attack Evaluator

An attack is not deterministic, making it harder for the evaluator to predict the game state resulting from applying the attack action to the actual game state.

The attack evaluator is principally very similar to the battle evaluator, but rather than simulating the victory and defeat of a whole battle it simulates the 2-3 possible outcomes of a single attack.

4.4. Evaluation Function

The evaluation function has the task of rating a game state. It is a function that expects a game state and a player and returns a real value. So even the simple function that, for every game state and player, always returns the constant *1* is a valid evaluation function.

The challenge is to design an evaluation function that maps higher ratings to *'better'* game states and thereby imposes an order on the game states with respect to their *'quality'*. Of course, *'quality'* has to be defined. In a non-deterministic game like Risk the probability that a given player in a given game state will eventually win the game is a good definition of the quality of a game state. That still leaves the challenge to determine which features of the game state are crucial for winning the game and what interpedences exist between them.

I designed several *features* from my personal Risk experience. Each *feature* is a function that expects a game state and a player as input and returns a real value. That is exactly the definition of an evaluation function. In fact, each feature is an evaluation function but with a different definition of the quality of the game state. It does not try to order the game states according to the probability of winning the game from the actual game state, but rather concentrates on the much smaller scale of its definition. Ordering the game states according to specific small scale features of the game states is much easier than ordering them according to the probability of winning the game.

This approach (similar to *divide and conquer*) requires a way to merge the results of the small scale features into a combined rating that correctly orders the game states according to the probability of winning the game. This is done using a *linear* function:

For each feature there exists a positive real value[7] called the *feature weight* which adjusts the feature result to the feature's relative importance in relation to all other

[7] To be precisely it is element of R_0^+

features. Because the features are not restricted in their co-domain and do not necessarily return comparable results, the feature weight itself does not necessarily correlate with its feature's importance.

The evaluation function just adds up the adjusted feature results

$$\text{Game State Rating} = \sum_{\text{Features}} (\text{Feature Weight} \times \text{Feature Result})$$

This linear approach makes the evaluation function comparatively fast. It also allows the feature weights to be manually tuned in a relative simple though cumbersome way. A more complex non-linear approach on the other hand would have allowed for a wider range of possible functions that would also be able to capture non-linear correlations between features. A brief overview of the advantages and disadvantages of linear and non-linear evaluation functions respectively can be found in [Fürnkranz, 2001].

4.5. Features

This section presents the handcrafted features I created to capture the strategically important aspects of the game state. The features themselves can be interpreted as small scale evaluation functions which rate a game state not on the high-level goal of winning the game, but on the rather specific definition of the feature. Each feature expects a game state and a player as input and returns a real value.

To make the evaluation of a game state faster I introduced *decision types*, i.e. the feature has another input parameter specifying the actual decision the player has to make. Thereby all features that are irrelevant to this decision do not have to be evaluated. To save computation time the irrelevant features just return 0 instead of calculating the real feature result. For each feature the game phases where the feature is irrelevant are listed in the individual feature description.

There exists another positive real value for each feature called the *feature scale factor* which is used to roughly scale the feature results into the co-domain $[0, 1]$ if possible. This creates a rough correlation between a feature's weight and its importance.

4.5.1. Armies Feature

The Armies Feature returns the number of armies of the actual player (AP) in relation to the total number of armies on the gameboard.

$$\text{Feature Result} = \frac{\text{Armies of } AP}{\text{Total Armies}}$$

The feature result is always in the interval $[0, 1]$. This feature is not applied in the *Placing new Armies* and *Fortifying the Position* phases of the game.

4.5.2. Best Enemy Feature

The Best Enemy Feature returns a negative measure of the power of the best enemy player. The power of the enemy players is measured by their averaged relative army strength and relative territory strength.

$$\text{Feature Result} = -1 \times \max_{\text{Enemy Player } i} \left(\left(\frac{\text{Armies of } i}{\text{Total Armies}} + \frac{\text{Territories of } i}{\text{Territories on Map}} \right) /2 \right)$$

The feature result is always in the interval $[-1, 0]$. This feature is not applied in the *Placing new Armies* and *Fortifying the Position* phases of the game.

4.5.3. Continent Safety Feature

The Continent Safety Feature returns a negative measurement of the threat from enemy players against continents which are completely occupied by the actual player. The threat is also weighted by the rating of the continents.

Let threat(Territory t) be the function that computes the probability that the given territory t will be occupied by enemy players if they try to conquer it. It is implemented in the class UsefulFunctions (Section 4.2). Let CR(Continent c) be the function that returns the continent rating of c. Let C denote all continents fully occupied by the actual player and B the border territories of the current continent c.

$$\text{Feature Result} = -1 \times \sum_{c \in C} \left(\left(\left(\sum_{b \in B} \text{threat}(b)^2 \right) + \left(\max_{b \in B} (\text{threat}(b)) \right) \right) \times CR(c) \right)$$

4.5.4. Continent Threat Feature

The Continent Threat Feature returns a measurement of the threat from the actual player against continents which are completely occupied by enemy players. The threat

is also weighted by the rating of the continents.

Let threat(Territory t) be the function that computes the probability that the given territory t will be occupied by enemy players if they try to conquer it. It is implemented in the class UsefulFunctions (Section 4.2). Let CR(Continent c) be the function that returns the continent rating of c. Let C denote all continents fully occupied by enemy players and B the border territories of the current continent c.

$$\text{Feature Result} = \sum_{c \in C} \left(\left(\sum_{b \in B} \text{threat}(b)^2 \right) \times CR(c) \right)$$

4.5.5. Distance to Frontier Feature

The Distance to Frontier Feature returns a measurement of the army distribution throughout the actual player's territories. Armies positioned far away from territories occupied by enemy players result in a lower feature value than armies positioned on border territories. The function distance(territory) is implemented in the class UsefulFunctions and computes the distance of a given friendly territory to the nearest enemy territory.

Let T denote the territories occupied by the actual player (AP).

$$\text{Feature Result} = \frac{\text{Total Armies of } AP}{\sum_{t \in T} \text{Armies in } t \times \text{distance}(t)}$$

The feature result is always in the interval $(0, 1]$. This feature is not applied in the *Attacking* phase of the game.

4.5.6. Enemy Estimated Reinforcements Feature

The Enemy Estimated Reinforcement Feature returns the negative estimation of the total number of armies the enemy players will be able to reinforce in the course of the next game round[8].

$$\text{Feature Result} = -1 \times \sum_{\text{Enemy Player } i} \text{Army Reinforcement Expectation of } i$$

This feature is not applied in the *Placing new Armies* and *Fortifying the Position* phases of the game.

[8] In this context the *next game round* is defined as the time period which starts immediately after the actual player's current turn ends and lasts until the actual player's next turn starts.

4.5.7. Enemy Occupied Continents Feature

The Enemy Occupied Continents Feature returns the number of continents which are completely occupied by enemy players.

This feature is not applied in the *Placing new Armies* and *Fortifying the Position* phases of the game.

4.5.8. Hinterland Feature

The Hinterland Feature returns the percentage of the territories of the actual player (AP) which are hinterland territories, i.e. which are not adjacent to an enemy territory.

$$\text{Feature Result} = \frac{\text{Hinterland Territories of } AP}{\text{Total Territories of } AP}$$

The feature result is always in the interval $[0, 1]$. This feature is not applied in the *Placing new Armies* and *Fortifying the Position* phases of the game.

4.5.9. Maximum Threat Feature

The Maximum Threat Feature returns a measurement of the probability that the actual player is able to successfully occupy at least a single enemy territory during his next *Attacking* game phase. Of all possible battles that the actual player is able to initiate the feature computes the victory probability and the maximum of these probabilities is returned.

Let Battles denote all the battles that the actual player would be able to initiate if he was in his *Attacking* phase.

$$\text{Feature Result} = \max_{b \in \text{Battles}} \big(\text{victoryProbability}(b)\big)$$

The feature result is always in the interval $[0, 1]$. This feature is not applied in the *Attacking* phase of the game.

4.5.10. More Than One Army Feature

The More Than One Army Feature returns the percentage of the territories of the actual player (AP) which are fortified territories, i.e. which are occupied by more than one army.

$$\text{Feature Result} = \frac{\text{Fortified Territories of } AP}{\text{Total Territories of } AP}$$

The feature result is always in the interval $[0, 1]$.

4.5.11. Occupied Territories Feature

The Occupied Territories Feature returns the number of territories which are occupied by the actual player in relation to the total number of territories on the map. In the case of the original map, which I used throughout my work, the total number of territories is 42.

$$\text{Feature Result} = \frac{\text{Territories of } AP}{\text{Total Territories}}$$

The feature result is always in the interval $(0, 1]$. This feature is not applied in the *Placing new Armies* and *Fortifying the Position* phases of the game.

4.5.12. Own Estimated Reinforcements Feature

The Own Estimated Reinforcement Feature returns the expectation of the total number of armies the actual player (AP) will be able to reinforce in his next *Placing new Armies* game phase.

$$\text{Feature Result} = \text{Army Reinforcement Expectation of } AP$$

This feature is not applied in the *Placing new Armies* and *Fortifying the Position* phases of the game.

4.5.13. Own Occupied Continents Feature

The Own Occupied Continents Feature returns the number of continents which are completely occupied by the actual player.

This feature is not applied in the *Placing new Armies* and *Fortifying the Position* phases of the game.

4.5.14. Own Occupied Risk Card Territories Feature

The Own Occupied Risk Card Territories Feature returns the number of Risk Cards in posession of the actual player whose corresponding territory is occupied by the actual player.

This feature is not applied in the *Placing new Armies* and *Fortifying the Position* phases of the game.

4.5.15. Risk Cards Feature

The Risk Cards Feature returns the number of Risk Cards which are in posession of the actual player.

This feature is not applied in the *Placing new Armies* and *Fortifying the Position* phases of the game.

4.6. Trade Evaluation Function

The trade evaluation function is an evaluation function in the domain of the Risk Card trades which is used in the *Trading in Risk Cards* game phase. It is similar to, though completely independent of the evaluation function (Section 4.4) that evaluates the game states in the *Placing new Armies*, *Attacking* and *Fortifying the Position* game phases. The trade evaluation function expects a game state and a potential trade of Risk Cards as parameters and is using this input to compute a real valued rating of the trade. This is done by simply adding up the evaluations of the features. Because the rating of a trade is much simpler than the rating of the complete game state the trade evaluation function needs less features than the evaluation function.

4.7. Trade-Features

The trade-features are similar to the features presented in Section 4.5 with the difference that the trade-features rate a trade rather than a game state.

To retain simplicity the trade-features are crafted in a way that the corresponding feature weights all have the value 1.0 and that no feature scale factors are needed. The trade-features themselves measure their feature results in army numbers that the trade is worth. Accordingly the features all return 0 in case the *'no trade'* trade is evaluated.

4.7.1. Occupied Territories Trade-Feature

The Occupied Territories Trade-Feature returns the number of Risk Cards whose corresponding territory is occupied by the actual player (AP) multiplied by 2. This correlates to the additional armies the trading player receives for occupying territories shown on the traded set of Risk Cards as introduced in Section 2.3.4.

Let territory(Risk Card rc) be the function that returns the territory of rc and

occupied(Territory t, Player p) be the function which returns 1 if t is occupied by p and zero otherwise. Let RC denote the set of Risk Cards of the trade excluding Wild Cards.

$$\text{Feature Result} = \sum_{c \in RC} \text{occupied}\big(\text{territory}(c), AP\big) \times 2$$

4.7.2. Unoccupied Territories Trade-Feature

The Unoccupied Territories Trade-Feature returns a negative measurement of the potential opportunity costs of reinforcement armies if this trade is performed now instead of being performed in a later *Trading in Risk Cards* game phase. For each Risk Card of the set, excluding Wild Cards, whose corresponding territory is occupied by an enemy player the feature computes the probability that the actual player would conquer that territory if he tried. Trades are rated less when the territories shown on the traded Risk Cards are not currently occupied by the actual player, but could easily be conquered.

Let threat(Territory t) be the function that computes the probability that the given territory t will be occupied by the actual player if he tries to conquer it. If t is already occupied by the actual player the function returns 0. It is implemented in the class UsefulFunctions (Section 4.2). Let territory(Risk Card rc) be the function that returns the territory of rc and RC denote the set of Risk Cards of the trade set excluding Wild Cards.

$$\text{Feature Result} = -1 \times \left(\sum_{c \in RC} \max\Big(\text{threat}\big(\text{territory}(c)\big) - 0.25, 0\Big) \right) \times 2$$

4.7.3. Trade Value Trade-Feature

This feature works only with the rules of the *Trading in Risk Cards* game phase that are introduced in Section 2.3.4, especially the army bonuses presented in table 2.1.

The Trade Value Trade-Feature returns a negative measurement of the potential opportunity costs of reinforcement armies if this trade is performed now instead of being performed in a later *Trading in Risk Cards* game phase. More precisely, it returns the negative difference between the maximal army bonus of a trade and the army bonus of the actual trade. Generally speaking, the less the army bonus of a trade, the less the rating of the trade.

$$\text{Feature Result} = -1 \times (10 - \text{Army Bonus of Trade})$$

5. Enhanced Evaluation Player

The drawbacks that arise because the decisions are treated independently are discussed together with the measures I took to lessen their effect. These measures include defining a target continent, using plans to achieve specific pre-defined goals as well as different strategies for the distribution of the reinforcement armies.

5.1. Drawbacks of the Basic Evaluation Player

The basic evaluation player presented so far has several major disadvantages: Armies are not placed where they are needed and important battles are not fought. The cause of these bad decisions is the fact that the evaluation player does not look ahead. Not only does it not search a game tree of the next game turns[1], but there is also no coordination between the different decisions of a single game turn. For example, the basic evaluation player has no way to realise that placing armies on a fought-over continent, or conquering a territory in that continent increases the chance to — several moves or turns later — occupy the whole continent.

Looking ahead in Risk is very time consuming, because it is so complex. As shown in Section 2.5, even searching a game tree of the decisions of a single game turn is infeasible.

To cope with the worst manifestations of this problem I introduced the *target continent* along with two new features utilising it as well as three tactical goals for which series of coordinated battles are planned and several different methods of placing the reinforcement armies.

Figure 5.1 shows an informal representation of the core architecture of the enhanced evaluation player.

[1] Including the other player's turns

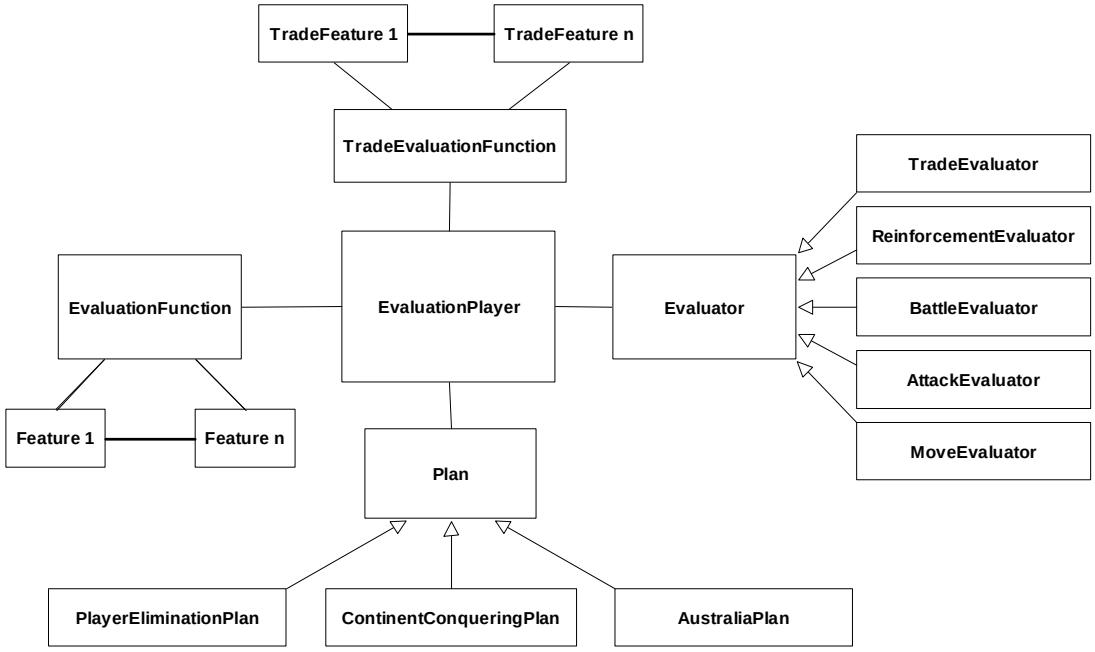

Figure 5.1.: The Core Architecture of the Enhanced Evaluation Player

5.2. Target Continent

The target continent tells the evaluation player on which continent to concentrate its efforts. This is done by using two features: the *Continent Army Domination Feature* and the *Continent Domination Feature*.

The Continent Army Domination Feature provides that placing armies on territories in the target continent is preferred during the *Placing new Armies* game phase. It also encourages moving armies from territories not part of the target continent into territories of the target continent during the *Fortifying the Position* game phase.

The Continent Domination Feature, on the other hand, provides that conquering territories in the target continent is preferred during the *Attacking* game phase. It also encourages moving armies from territories not part of the target continent into territories of the target continent in case of a combat move.

If the feature weights are set accordingly, this approach provides game play more concentrated on conquering a single continent.

The continent target is computed anew for every decision[2], so that in case of different circumstances the continent target can be adapted. It is chosen from all the continents not completely occupied by the actual player. The decsision is based on the continent rating (Section 4.2.1) and the actual army and territory distribution. The continent rating is multiplied with a factor measuring the relative power of the actual player (AP) in that continent. The continent with the highest *adjusted continent rating* (ACR) is chosen as the continent target. Steadyness of the continent target is guaranteed by the fact that the decisions which are encouraged by the two new features increase the relative power of the player in the continent target, thus increasing the adjusted continent rating even further.

Let $CR(\text{Continent}c)$ be the function that returns the continent rating of c, then for a continent C

$$ACR\,(C) = \left(\left(\frac{\text{Armies of } AP \text{ on } C}{\text{Total Armies on } C} + \frac{\text{Territories of } AP \text{ on } C}{\text{Territories on } C} \right) /2 \right) \times CR\,(C)$$

5.2.1. Continent Army Domination Feature

The Continent Army Domination feature returns the number of armies the actual player (AP) has on the target continent (TC) divided by the total number of armies on the target continent.

$$\text{Feature Result} = \frac{\text{Armies of } AP \text{ on } TC}{\text{Total Armies on } TC}$$

The feature result is always in the interval $[0, 1]$. This feature is not applied in the *Attacking* phase of the game.

5.2.2. Continent Domination Feature

The Continent Domination Feature returns a measurement of the relative power of the actual player (AP) on the target continent (TC) multiplied by the rating of the target continent.

Let $CR(\text{Continent}c)$ be the function that returns the continent rating of c.

$$\text{Feature Result} = \left(\left(\frac{\text{Armies of } AP \text{ on } TC}{\text{Total Armies on } TC} + \frac{\text{Territories of } AP \text{ on } TC}{\text{Territories on } TC} \right) /2 \right) \times CR(TC)$$

[2] An exception is the *Trading Cards* decision.

This feature is not applied in the *Placing new Armies* and *Fortifying the Position* phases of the game.

5.3. Plans

From all the flaws arising from the original evaluation player's inability to look ahead a series of battles, I chose the three most important ones and developed a method to cope with them. I created the abstract class *Plan* which allows arbitrary territory targets to be specified and computes a forest of *attack trees*[3] containing all the target territories. So far this cannot be done dynamically. Instead, a concrete subclass of the class Plan has to be created. For each of the three major flaws of the basic evaluation player during the *Attacking* game phase[4] I built a concrete subclass of the class Plan which creates the required list of target territories.

The class Plan offers a few methods that should be noted:

void initialize(Vector targetList) This is the core method of the class. From the given list of target territories it computes a forest of attack trees containing all of the target territories. The created forests are not necessarily optimal, in the sense that they do not take into account the number of armies currently positioned on the root territories of the attack trees. Hence, it is possible that a different attack tree with the same number of nodes exists, that has more armies positioned on its root territory, thus having a higher chance of successfully occupying all target territories in the tree.

Listings 5.1 to 5.4 show the pseudocode version of the algorithm creating the forest of attack trees.

double getSuccessChance(Gameboard b) This method, given the current game situation, simply returns the probability that the plan succeeds, i.e. all target territories are occupied by the actual player when the plan finishes. The success chance

[3] An *attack tree* specifies a series of battles, which can contain several battles starting from the same territory.

[4] Namely the inability to efficiently eliminate enemy players, the inability to snatch the continent Australia from an enemy player in case the opportunity exists and the slowness with which continents are conquered due to inefficient sequences of battles.

```
Plan::initialize(Vector targetList) {
  while(not targetList.isEmpty()) {
    t ← this.chooseStartTerritory(targetList)
    path.add(t)
    targetList.remove(t)
    while(t ≠ null) {
      t ← this.chooseNextTerritory(targetList, t)
      path.add(t)
      targetList.remove(t)
    }
  path ← this.addOwnTerritory(path)
  this.removePathTerritoriesFromTargetList()
  }
  pathList.add(path)
  path ← new Vector()
}
```

Listing 5.1: The *initialize* Method of Class Plan

```
Plan::chooseStartTerritory(Vector targetList) {
  targetList ← this.sortTerritories(targetList)
  // sorts the target territories with respect to the number
  // of adjacencies towards each other, ascending
  for each t in targetList {
    if board.territoryIsInFrontOfPlayer(actualPlayer, t){
      return t
    }
  }
  return targetList.getFirstTerritory()
}
```

Listing 5.2: The *chooseStartTerritory* Method of Class Plan

```
Plan::chooseNextTerritory(Vector targetList, Territory pTer) {
  targetList ← this.sortTerritories(targetList)
  // sorts the target territories with respect to the number
  // of adjacencies towards each other, ascending
  for each t in targetList {
    if board.adjacent(pTer,t){
      return t
    }
  }
  return null
}
```

Listing 5.3: The *chooseNextTerritory* Method of Class Plan

is computed by multiplying the individual success chances of the various attack trees of the plan.

double getSuccessChanceWithReinf(int ra, Gameboard b) This method, given the current game situation and a number of additional army reinforcements, returns the probability that the plan succeeds, i.e. all target territories are occupied when the plan finishes. The success chance is computed by multiplying the individual success chances of the various attack trees of the plan. In contrast to the previous method, this method internally uses the *distributeReinforcements* method to hypothetically distribute the reinforcement armies in a way that increases the success chance of the plan. The method returns this increased success chance.

Additionally the class Plan has methods of gameplay similar to the ones that the abstract class Player (Section 3.6) stipulates. That allows the Plan, in cooperation with teh enhanced version of the evaluation player, to effectively take over the gameplay from the player.

Reinforcement distributeReinforcements(int r, Gameboard b) This method creates and returns a valid distribution of a number of reinforcement armies equal to, or less than the given number of reinforcement armies.

```
Plan::addOwnTerritory(Vector path) {
  pair ← getClosestTerritoryPair(ownTerritories,path)
  // returns the pair (OwnTerritory,pathTerritory)
  // with shortest distance
  attackPath ← UseFulFunctions.getBestAttackPath(pair)
  if(this.overlappingPaths(attackPath, pathList) {
    this.removeTerritoriesBeforeOverlap(attackPath)
    this.appendToEarlierPath(attackPath)
  }
  meeting = this.checkWherePathsMeet(attackPath,path)
  switch(meeting)
    case 0 {
      // start of path
      attackPath.append(path)
    }
    case 1 {
      // end of path
      path.reverse()
      attackPath.append(path)
    }
    case 2 {
      // middle of path
      firstPart ← this.getFirstPart(path)
      firstPart.reverse
      secondPart ← this.getSecondPart(path)
      attackPath.append(firstPart,secondPart)
    }
  return attackPath
}
```

Listing 5.4: The *addOwnTerritory* Method of Class Plan

The reinforcement armies are distributed only among the root territories of the attack trees of the plan. A single reinforcement army is continually placed on the root territory whose attack tree has the least individual success chance. This is done until either the given number of reinforcement armies is distributed, or the overall success chance exceeds 0.95. If less than the given number of reinforcements are distributed the improved evaluation player distributes the remainder of the reinforcement armies on its own.

Attack makeAttack(Gameboard b) This method creates and returns a valid attack. The algorithm making the decision is quite simple, it chooses the first attack of the first attack tree with the maximal possible number of attacking armies. If an attack results in the occupation of the target territory a combat move decision is made if necessary and the attack tree is updated to reflect the new game situation. If the number of child nodes of the root node exceeds 1, the attack tree is divided into two independent attack trees with the same root node. The segmentation is done in a way that the root node of the new attack tree containing the executed attack has a degree of 1. After that segmentation, the root node of the attack tree including the executed attack necessarily has a branching factor of 1[5]. If the node of the target territory is a leaf of the former attack tree, i.e. no attack is started from the newly conquered territory, the attack tree including the executed attack is deleted. Otherwise the root node of the attack tree including the executed attack is removed and the node of the target territory is the new root node of the tree.

Figure 5.2 shows several steps in the update process of an example attack tree.

Move makeCombatMove(Move combatMove, Gameboard b) This method decides how many armies will be moved along in a newly conquered territory. If the root node of the attack tree of the last attack has a branching factor of 1[6], the maximal possible number of armies is moving along. Otherwise the armies so far remaining in the start territory of the attack have to be divided. The method distributes the available armies to the different attack trees that arise when the current attack tree will be split. This is done similar to the plan-controlled distribution of armies in the *Placing new Armies* game phase. A single army is continually allocated to

[5] If the segmentation was not necessary, the root node had a brachning factor of 1 to start with.
[6] Note that this method is executed *before* the attack tree is upated.

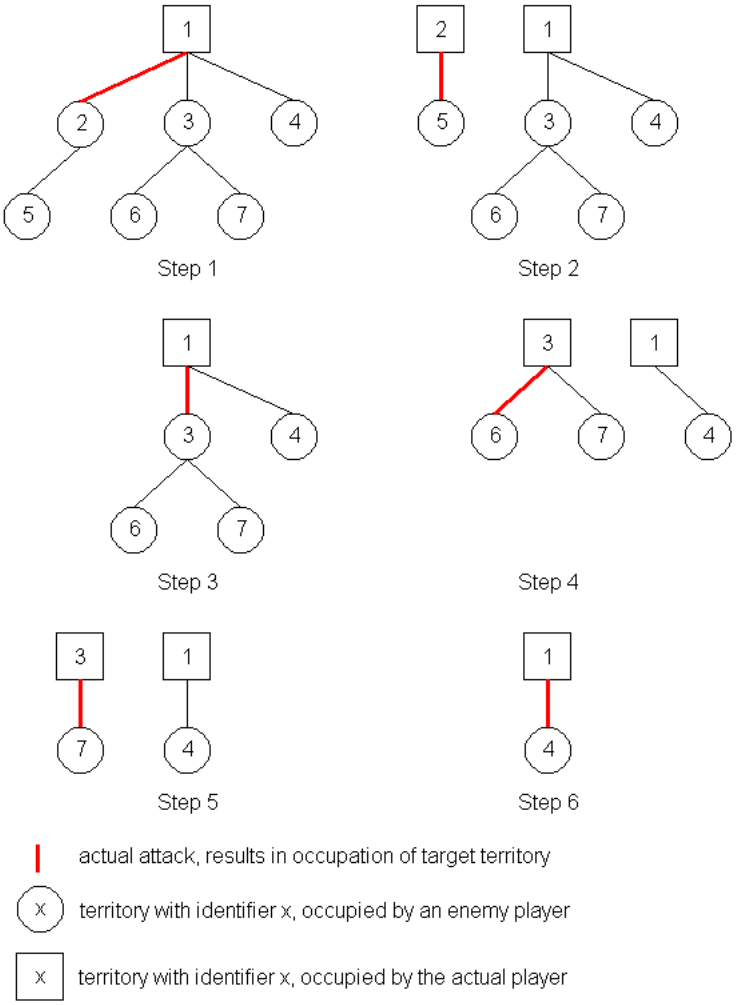

Figure 5.2.: Updating of an Example Attack Tree

the attack tree with the lowest individual success chance until all availbale armies have been distributed. The number of armies making the combat move is set to the number of armies allocated to the attack tree including the last executed attack.

In case the node of the target territory was a leaf of the former attack tree, i.e. no attack is started from the freshly conquered territory, the enhanced evaluation player is given control of the combat move decision.

When the improved evaluation player activates the plan, the plan will have complete control over the player's decisions as long as it is executing. A plan finishes either when it is successful or when it failes, i.e. the success chance has fallen to 0. Plans can start at the beginning of a player's game turn and in the *Attacking* game phase after a previous plan has finished. In general, all plans finish during the *Attacking* game phase of the same game turn in which they started.

5.3.1. Activating Plans

Plans can start at the beginning of a player's game turn and in the *Attacking* game phase after a previous plan has finished. In these situations the improved evaluation player checks whether a new plan should be initiated. All three plans are checked for initiation one after the other as long as no plan has already been chosen. This imposes an order to the plans. The *Player Elimination Plan* has the highest priority, followed by the *Australia Plan* and finally by the *Continent Conquering Plan*.

The checking routine consists of three different tests:

Initial Test The initial test - performed before a plan is created - is a rough estimate whether a plan is feasible at all. The main purpose of the initial test is to reduce the computation time required by the creation of plans with a insufficient success chance. Generally speaking, it denies the creation of a new plan when the number of reinforcement armies the player has at its disposal[7] is equal or less than an estimate of the number of armies that would have to be destroyed multiplied with the *initial test factor* (ITF).

The creation of a plan is allowed if

$$\text{Maximal Reinforcements} > ITF \times \text{Estimated Resistance}$$

As the result of several short intuitive trials I have set the initial test factor to 1.25. The estimated number of resisting enemy armies depends on the plan itself and is elucidated in the individual sections of the different plans.

Standard Test After the plan has been created, the standard test decides whether the plan is executed based on the success chance (SC) of the plan and a plan-specific

[7] Containing regular reinforcement armies for territories and continents as well as the maximal number of reinforcements the player could receive for trading in a set of Risk Cards

plan factor (PF). In the calculation the test makes use of the *risk factor* (RF), which individual players can use to influence the minimal success chance that a plan needs to provide before it is allowed to execute.

Let SCR(int reinforcements) be the function that returns the success chance of the plan after reinforcements additional reinforcement armies have been distributed by the plan. A plan is executed if

$$SCR(\text{Reinforcements without Trade}) > 1 - RF \times PF$$

It is important to note that the standard test does not take into account any reinforcement armies received from trading in Risk Cards. This implies that when a plan is executed after the standard test the player does not necessarily have to trade in a set of Risk Cards.

After performing several test runs, I set the risk factor to 0.08, though values between 0.04 and 0.12 performed almost equally well. The plan factor for the three plans is elucidated in the sections of the plans themselves.

Trade Test If the plan has been rejected by the standard test the trade test decides whether the plan will be executed in combination with a trade of Risk Cards. By adding the maximal number of reinforcement armies achievable by trading in Risk Cards to the distributable reinforcement armies the success chance of the plan is inreased[8]. This happens at the price of losing Risk Cards as well as the opportunity costs which arise when trading in a set of Risk Cards in a later game turn would result in a greater number of reinforcement armies. The trade test takes this into consideration and adjusts the risk factor for trades with army bonuses below the maximal army bonus. The trade test is only applied if the player can trade in at least a single set of Risk Cards.

Let SCR(int reinforcements) be the function that returns the success chance of the plan after reinforcements additional reinforcement armies have been distributed by the plan and MTR denote the maximal number of reinforcements achievable by a trade of Risk Cards. A plan is executed if

$$SCR(\text{Maximal Reinforcements}) > 1 - \left(RF - \frac{10 - MTR}{125}\right) \times PF$$

[8]This predication is only true if the player is able to trade in at least one valid set of Risk Cards. Therefore the trade test is only applied in that case.

If a plan is executed due to the trade test, the enhanced evaluation player is forced to choose a set of Risk Cards to trade for reinforcement armies. But the decision which set to trade is still made by the player, it just is not able to choose the option of not trading.

As the result of several short intuitive trials I set the denominator of the fraction to 125. The plan factor for the three plans is elucidated in the sections of the plans themselves.

Figure 5.3 shows an activity diagram of the plan activation process.

5.3.2. Player Elemination Plan

One of the worst flaws of the basic evaluation player was its inability to efficiently eliminate enemy players. The player did not utilise an opportunity to wipe out an enemy player unless the enemy player had only a single territory left. But eliminating enemy players, especially if they posses many Risk Cards, is vital to winning the game. There are two reasons for this importance. First, enemy players possesing many Risk Cards are likely to trade in a set of these cards, resulting in a major boost to the number of their reinforcement armies. Second, when a player is eliminated, the eliminating player captures all Risk Cards currently in possession of the eliminated player. Eliminating a player with many Risk Cards may even result in an immediate trade of Risk Cards by the eliminating player. This is the only possible way of placing additional reinforcement armies in the middle of the *Attacking* game phase.

The Player Elimination Plan is designed to cope with this problem. The list of target territories consists of all the territories currently occupied by the to-be-eliminated player.

But the Player Elimination Plan is not without risk, if it fails it may very well result in a worse game situation for the actual player. Not only did he use his resources in a way that might not be matching his overall strategy, but he also nearly extinguished an enemy player possesing a potential high number of Risk Cards. That might allow other players the opportunity to easily finish the actual player's failed attempt to eliminate the target player.

The estimated resistance used in the initial test is the total number of armies of the to-be-eliminated player, while the plan factor of the standard and trade tests is the number of Risk Cards currently in possession of that player. Therfore, the higher the

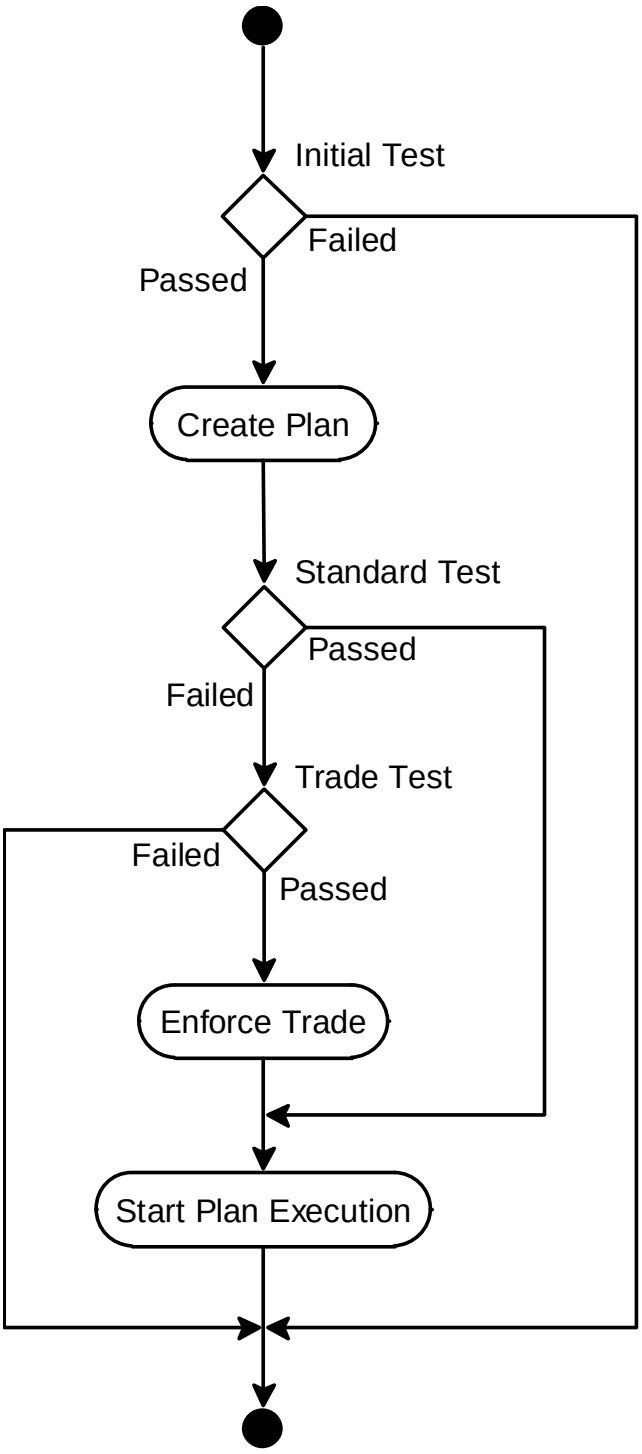

Figure 5.3.: The Plan Activation Process

number of Risk Cards in possession of the to-be-eliminated player, the less the minimum required victory chance for a plan to be executed.

5.3.3. Australia Plan

Australia is an important continent, especially in the early part of the game. It consists of only four territories making it easy and fast to conquer, while it has just a single border making it easy to defend. Naturally, many players try to conquer it at the beginning of the game. Usually, once the first struggle is decided, it is very hard to conquer the continent. But the situation might arise that the player occupying Australia reduces the number of defending armies and the actual player has sufficient reinforcement armies to successfully conquer the continent.

This plan is designed to use such an opportunity and snatch Australia away from the occupying player. It is only executed when Australia is completely occupied by a single enemy player[9] and only when there is a decent chance of success. It is of no use to the actual player to destroy the defending armies and not to be able to conquer the whole continent, or not to be able to defend it in the case of a success.

The estimated resistance used in the initial test is the total number of armies positioned on the continent of Australia and the territory of Siam[10], while the plan factor of the standard- and trade test is 2.

5.3.4. Continent Conquering Plan

The Continent Conquering Plan strives to improve the conquest of the target continent. The target list consists of the territories of the target continent currently not occupied by the actual player. In contrast to the other two plans, even a failed plan only improves the overall strategic goal of the actual player. Therefore, even if the chance to completely conquer a continent is rather low, the plan may be executed.

The estimated resistance used in the initial test is the total number of enemy armies on the target continent, while the plan factor of the standard- and trade test is 5.

[9] The impact of that restriction has not been researched yet.
[10] The armies positioned on Siam are only included if Siam is not occupied by the actual player.

5.4. Reinforcement Distribution

Another flaw of the basic evaluation player is the completely independent army placement in the *Placing new Armies* game phase. Let us assume, for example, that a territory was in need of armies to defend itself or to conquer an adjacent one. Let us further assume that the victory probability was very low, so that quite a few a armies were needed to achieve an expected victory. Even though the basic evaluation player might have enough armies to place in that territory he would not do so. This is due to the placement of single armies on at a time. In case of very low victory probabilities, placing a single army on the territory will only very slightly increase the rating of the game state, while placing the army on a different territory is likely to yield a better overall rating.

To cope with the problem I changed the way the armies are distributed. As explained in Section 2.5, placing the armies in a single action is not feasible. So I extended the one-at-a-time placement and added the possibility to place all reinforcemet armies on a single territory. After all armies are placed the final reinforcement distribution is compared to the different distributions placing all reinforcement armies on a single territory. The distribution yielding the best evaluation is chosen. I also developed procedures were the armies are not completely placed one after another, but in several packages with each having a decreasing number of armies.

I created five different army placement procedures:

One This is the distribution procedure of the basic evaluation player. Single armies are successively placed on the territory yielding the highest gain in the overall game state evaluation.

Listing 5.5 shows a pseudocode version of this placement procedure.

One-Max Single armies are successively placed on the territory yielding the highest gain in the overall game state evaluation. In addition to the One procedure, the resulting distribution is compared to the different distributions that arise when all reinforcement armies are placed on a single territory. The distribution yielding the best evaluation is chosen.

Listing 5.6 shows a pseudocode version of this placement procedure while listing 5.7 shows the max-placement method.

```
EvaluationPlayer::standard(int armies,Reinforcement reinf) {
  Territories ← board.getTerritories(actualPlayer)
  while armies > 0 {
    for each territory in Territories {
      tempReinf ← new Reinforcement(territory,1)
      ReinfList.add(tempReinf.mergeReinf(reinf))
    }
    reinf.mergeReinf(chooseBest(ReinfList,evaluator))
    armies--
  }
  return reinf
}
```

Listing 5.5: The *Standard* Distribution Procedure

```
EvaluationPlayer::standardMax(int armies) {
  reinf ← this.standard(armies,new Reinforcement())
  return this.max(armies,reinf)
}
```

Listing 5.6: The *Standard-Max* Distribution Procedure

Half-Quarter-One-Max This dsitribution procedure places half of the reinforcement armies as a bundle in the first step of the process. The second step consists of placing a bundle of a quarter of the reinforcement armies, after that the procedure distributes the remainder of the reinforcement armies similar to the standard procedure. In addition, the resulting distribution is compared to the different distributions that arise when all reinforcement armies are placed on a single territory. The distribution yielding the best evaluation is chosen.

The occurring fractions are rounded to the closest natural number.

Listing 5.8 shows a pseudocode version of this placement procedure.

Quarter-Quarter-One-Max This distribution procedure places a quarter of the reinforcement armies as bundles in the first two steps of the process. After that, the

```
EvaluationPlayer::max(int armies, Reinforcement reinf) {
  Territories ← board.getTerritories(actualPlayer)
  for each territory in Territories {
    ReinfList.add(new Reinforcement(territory, armies))
  }
  ReinfList.add(reinf)
  return chooseBest(ReinfList, evaluator)
}
```

Listing 5.7: The *Max-Placement* Method

procedure distributes the remainder of the reinforcement armies similar to the standard procedure. In addition, the resulting distribution is compared to the different distributions that arise when all reinforcement armies are placed on a single territory. The distribution yielding the best evaluation is chosen.

The occurring fractions are rounded to the closest natural number.

Listing 5.9 shows a pseudocode version of this placement procedure while.

Quarter-One-Max This distribution procedure places a quarter of the reinforcement armies as a bundle in the first step of the process. After that the procedure distributes the remainder of the reinforcement armies similar to the standard procedure. In addition the resulting distribution is compared to the different distributions that arise when all reinforcement armies are placed on a single territory. The distribution yielding the best evaluation is chosen.

The occurring fractions are rounded to the closest natural number.

Listing 5.10 shows a pseudocode version of this placement procedure.

The different army placement procedures require different amounts of computation time, whereas the procedures presented in this section all have comparably modest requirements and are feasible to implement.

Table 5.1 shows the computation time demand of the different army placement procedures measured in evaluations of game states.

```
EvaluationPlayer::HalfQuarterOneMax(int armies) {
  half ← round(armies / 2.0)
  quart ← round(armies / 4.0)
  Territories ← board.getTerritories(actualPlayer)
  for each territory in Territories {
    ReinfList.add(new Reinforcement(territory,half))
  }
  reinf ← chooseBest(ReinfList,evaluator)
  ReinfList.clear()
  for each territory in Territories {
    tempReinf ← new Reinforcement(territory,quart)
    ReinfList.add(tempReinf.mergeReinf(reinf))
  }
  reinf.mergeReinf(chooseBest(ReinfList,evaluator))
  reinf.mergeReinf(this.standard(armies - half - quart,reinf))
  return this.max(armies,reinf)
}
```

Listing 5.8: The *Half-Quarter-One-Max* Distribution Procedure

```
EvaluationPlayer::QuarterQuarterOneMax(int armies) {
  quarter ← round(armies / 4.0)
  Territories ← board.getTerritories(actualPlayer)
  for each territory in Territories {
    ReinfList.add(new Reinforcement(territory,quarter))
  }
  reinf ← chooseBest(ReinfList,evaluator)
  ReinfList.clear()
  for each territory in Territories {
    tempReinf ← new Reinforcement(territory,quarter)
    ReinfList.add(tempReinf.mergeReinf(reinf))
  }
  reinf.mergeReinf(chooseBest(ReinfList,evaluator))
  reinf.mergeReinf(this.standard(armies - 2 * quarter,reinf))
  return this.max(armies,reinf)
}
```

Listing 5.9: The *Quarter-Quarter-One-Max* Distribution Procedure

```
EvaluationPlayer::QuarterOneMax(int armies) {
  quarter ← round(armies / 4.0)
  Territories ← board.getTerritories(actualPlayer)
  for each territory in Territories {
    ReinfList.add(new Reinforcement(territory,quarter))
  }
  reinf ← chooseBest(ReinfList,evaluator)
  reinf.mergeReinf(this.standard(armies - quarter,reinf))
  return this.max(armies,reinf)
}
```

Listing 5.10: The *Quarter-One-Max* Distribution Procedure

Procedure	**Evaluations**	**A = 9, T = 13**
One	$A \times T$	117
One-Max	$(A+1) \times T$	130
Half-Quarter-One-Max	$(\lfloor \frac{A}{4} \rfloor + 3) \times T$	65
Quarter-Quarter-One-Max	$(A - 2 \times \lfloor \frac{A}{4} + 0.5 \rfloor + 3) \times T$	104
Quarter-One-Max	$(A - \lfloor \frac{A}{4} + 0.5 \rfloor + 2) \times T$	117
Optimal	$\binom{T+A-1}{A}$	293,930

Table 5.1.: Computation Time Demand of Different Reinforcement Distribution Procedures. The demand is measured in game state evaluations. A and T are abbrevations for the number of *Armies* and *Territories* of the reinforcing player. The example features the average number of armies and territories in the *Placing new Armies* game phase measured in the complexity measurement test run (Section 7.1).

6. Learning Player

To be able to automatically determine the optimal feature weights of the evaluation function I applied the technique of *TD-Learning* to the enhanced evaluation player.

6.1. Temporal-Difference Learning

6.1.1. Introduction

Temporal-Difference (TD) learning is a *reinforcement learning* technique that combines *Monte Carlo* ideas with *dynamic programming* ideas [Sutton and Barto, 1998]. Sutton [Sutton and Barto, 1998] introduces temporal-difference learning as follows:

> *Like Monte Carlo methods, TD methods can learn directly from raw experience without a model of the environment's dynamics. Like dynamic programming, TD methods update estimates based in part on other learned estimates, without waiting for a final outcome.*

Because TD learning is learning *a guess from a guess* [Sutton and Barto, 1998], it is called a *bootstrapping* method.

In TD learning the previous estimates of the learning task are changed towards the current one. Therefore, it is possible to continually update the estimate whithout having to wait for the final outcome. For a detailed introduction on TD learning interested readers refer to Sutton's work [Sutton and Barto, 1998].

6.1.2. The TD(Lambda) Learning Algorithm

The TD(λ) learning algorithm was introduced by Sutton in 1988 [Sutton, 1988]. It changes the weights of the prediction according to the difference of the last two predictions in a way that reduces the error of the previous predictions. The parameter λ[1] is

[1] $0 \leq \lambda \leq 1$

used to weigh the impact of the previous predictions. $\lambda = 0$ does only consider the last error while $\lambda = 1$ gives equal weight to all the previous errors.

Let P_t be the prediction of time-step t and w the vector of weights.

$$w_{t+1} = w_t + \alpha \times (P_{t+1} - P_t) \times \sum_{k=1}^{t} \lambda^{t-k} \Delta_w P_k$$

with α a positive step-size parameter and $\Delta_w f(w)$, for any function f, denotes the vector of partial derivatives with respect to w.

In the case of the linear evaluation function the vector of partial derivatives is just the vector of the feature values.

Let F(GameState s) be the linear evaluation function and f_i(GameState s) the function evaluating the ith feature. Let x_t denote the game state of time-step t, then the formula used to update the weight of the ith feature is [Fürnkranz, 2001]

$$w_{i,t+1} = w_{i,t} + \alpha \times \big(F(x_{t+1}) - F(x_t)\big) \times \sum_{k=1}^{t} \lambda^{t-k} f_i(x_k)$$

Sutton [Sutton and Barto, 1998] shows that the vector of the feature weights will converge to an optimum as long as the step-size parameter α is reduced over time according to the conditions

$$\sum_{k=1}^{\infty} \alpha_k = \infty$$

and

$$\sum_{k=1}^{\infty} \alpha_k^2 < \infty$$

6.1.3. TD Learning in Risk

Reinforcement

The goal of the learning algorithm is to modify the evaluation function in a way that it converges to the function that returns the probability of winning the game from a given game state. This can be achieved by setting the reinforcements of winning and losing a game to one and zero respectively.

Let x_T denote the game state of the last time-step T, then

$$F(x_T) = 1$$

if the player has won the game, and

$$F(x_T) = 0$$

otherwise.

Learning Steps

When learning is applied to a traditional board game, there is one learning step, i.e. one weight update, after each turn of the learning player. In Risk, on the other hand, it is also possible to learn after each individual decision in the course of the player's turn. Learning after each decision clearly accelerates the learning process, but introduces the problem of inhomogenous learning steps. Most of the time only the player's last action would have changed the game state without any of the other players acting in between, but once every turn not only the player's last action but also all of the other players actions would have changed the game state. The game state after the other players made their moves is rather likely to be worse for the learning player than the game states occurring after performing the player's action only. I assume this would cause the learning algorithm to overrate the actions without the interference of other players as well as underrating the actions with the interference of the other players. Therefore, I chose to perform only a single learning step at the end of the player's turn.

Normalization

After implementing the formula shown in Section 6.1.2, I observed that the feature weights of the evaluation function were continually growing until they exceeded the representation limit of the underlying programming language. To cope with this problem I *normalized* the feature vectors used in calculating the new feature weights.

Let $F(\text{GameState } s)$ be the linear evaluation function, $f(\text{GameState } s)$ the function that returns the vector of all feature evaluations of game state s and $f_i(\text{GameState } s)$ the function evaluating the ith feature. Let x_t denote the game state of time-step t, then the formula used to update the weight of the ith feature including normalization is

$$w_{i,t+1} = w_{i,t} + \alpha \times \big(F(x_{t+1}) - F(x_t)\big) \times \frac{\sum_{k=1}^{t} \lambda^{t-k} f_i(x_k)}{\|f(x_t)\| \times \left\|\sum_{k=1}^{t} \lambda^{t-k} f(x_k)\right\|}$$

with $\|\cdot\|$ the Euclidean Norm.

6.2. Implementation of the Learning Process

To implement the learning process the learning player extends the enhanced evaluation player by adding a *learn* method, methods to make the feature weights persistent and a data structure storing feature evaluations of past turns required to compute the current feature weight updates.

The learn method is called at the end of each game turn of the learning player as well as once after the player is eliminated from the game. After the last learning step of the game has been performed the current feature weights are stored in a file. They are reloaded when the player begins its next game.
When a learning player plays its first game, the feature weights are randomly generated with

$$\text{weights} \in [0, 0.001]$$

The learning player encompasses two parameters: α the step-size parameter and λ the parameter specifying the impact of earlier game states on the learning process.

7. Experiments

To be able to rate the playing skill of my players I performed several experiments setting the playing strength of the various players in relation to each other and several human players. I also measured the complexity of the games played in the Risk framework.

7.1. Complexity Measurement

In the complexity measurements I measur the frequency of each decision and the number of valid actions per occurrence of the decision.

7.1.1. Methods

To measure the complexity of the decisions in the Risk framework, I developed a tool monitoring the occurrence of decisions in the game manager. The tool keeps track of the number of times the decision occurred in each game turn as well as the number of valid actions for each occurrence of a decision.

As explained in Section 2.5.5, I simplified the *Attack* and *Place new Armies* decisions in the Risk framework. Therefore, in addition to measuring the number of valid actions actually occurring in the Risk framework, I also calculate the number of valid actions that would occur without the simplifications. In case of the *Placing new Armies* decision I also measure the number of reinforcement armies the players receive as well as the number of territories they occupy at the time of the decision.

Additionally, I keep track of the duration of the games, i.e. the number of game rounds as well as the number of game turns.

Table 7.1 summarises the settings used in the experiment.

Parameter	Value
Number of Games	1,000
Number of Players	4
Measured Players	4
Player	Enhanced Evaluation Player
Reinforcement Distribution	One-Max

Table 7.1.: Settings for the Risk Complexity Measurement

7.1.2. Results

Some characteristics of the distributions are shown in tables 7.2 and 7.3 while table 7.4 presents the results of the measurement of the game duration. It is easy to see that the

Decision	Absolute	Occurrence per Turn					
		Min	25P	Median	75P	Max	Average
Trade Cards	44,144	0	0	1	1	3	0.634
Place Armies	71,990	1	1	1	1	3	1.034
Attack	492,987	1	4	5	9	74	7.084
Combat Move	98,852	0	0	1	2	19	1.420
Fortify Position	69,589	1	1	1	1	1	1.000

Table 7.2.: Frequencies of Risk Decisions

Place new Armies decision is the single most complex decision in Risk. The number of valid actions is roughly 19 orders of magnitude greater than the second largest one. The impact of the simplified placement method used in the Risk Framework is also clearly visible from the measured data. The number of valid actions in that case is only little more than twice as high as the second largest one.

The histograms of the distributions of the frequencies of the decisions as well as the histograms of the distributions of the valid actions per decision occurrence are depicted in Appendix A. The histogram showing the number of occupied territories at the time of the *Place new Armies* game phase is also shown in figure 7.1.

Decision	Min	25P	Median	75P	Max	Average
Trade Cards	1	1	2	3	85	2.761
Place Armies	1	165	3,003	2.496×10^6	10^{24}	10^{21}
Place Armies, RFW	3	27	65	160	1,845	147.945
Attack	1	5	8	11	62	8.724
Attack, RFW	1	5	7	8	37	8.372
Combat Move	2	3	6	10	171	7.655
Fortify Position	1	7	26	80	1,413	67.913

Table 7.3.: Valid Actions of Risk Decisions

Parameter	Value
Number of Games	1,000
Number of Game Rounds	21,616
Number of Game Turns	69,589
Average Game Rounds per Game	21.616
Average Game Turns per Game and Player	17.397

Table 7.4.: Durations of Risk Games

Besides being useful for analysing the complexity of Risk (Section 2.5), the histogram shows the impact of the *Enemy Estimated Reinforcements Feature*. From 12 territories onwards, groups of territories with roughly the same relative frequency can be identified. These clusters result from players striving to reduce the number of reinforcement armies of the enemy players. Players occupying 12, 15, ..., 39 territories tend to be attacked more often, hence losing a territory more often[1]. This explains the reduced frequency of these territory numbers as well as the increased frequency of the territory numbers with one territory less than the next cluster, i.e. 11, 14, ..., 38.

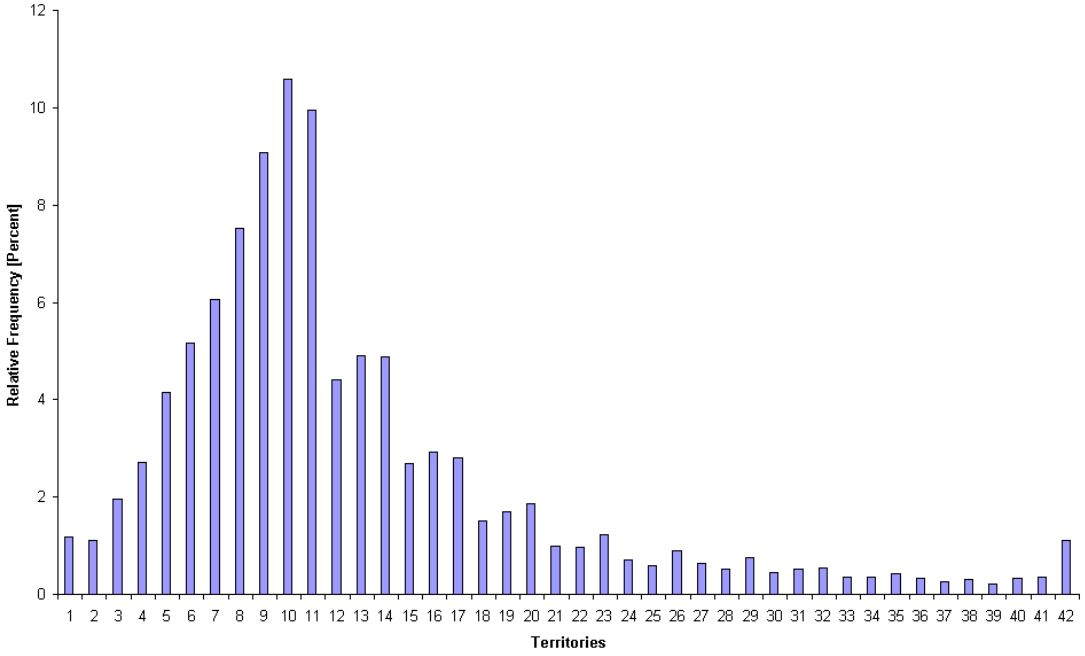

Figure 7.1.: Distribution of the Number of Occupied Territories in the *Place Armies* decision

7.2. Rating System

To be able to compare the playing skill of different players I defined a benchmark player and a rating system.

[1] In the case of these number of occupied territories, the loss of one territory subsequently reduces the number of reinforcement armies by one.

7.2.1. Benchmark Player

The benchmark player is the enhanced evaluation player as presented in Chapter 5 with the One-Max reinforcement distribution procedure.

7.2.2. Rating a Player

The rating of the playing strength of a player is a measurement of its playing skill relative to the playing skill of the benchmark player. Naturally, the benchmark player has a *player rating* (PR) of 100%.

The player rating is defined by letting the test player play against the benchmark player for several games. This is done in four player games with one instance of the test player and three instances of the benchmark player.

Let GP denote the number of played test games and GW the number of test games won by the test player, then the player rating of a test player is

$$PR = \frac{GW}{GP} \times 4$$

7.3. Enhanced Evaluation Player

To be able to rate the impacts on the playing strength of the changes to the basic evaluation player, I compared several versions of the evaluation player against the benchmark player.

7.3.1. Methods

I performed ratings for the basic evaluation player and for each of its three enhancements. All of these ratings are based on test runs consisting of at least 1,000 games each.

Basic Evaluation Player

The basic evaluation player (Chapter 4) is rated.

Target Continent

The basic evaluation player with the addition of the target continent (Section 5.2) is rated.

Plans

The basic evaluation player with the addition of the plans (Section 5.3) is rated. For each of the three different plans the basic evaluation player with the addition of just that plan, as well as a version with the addition of all three plans, is rated.

Reinforcement Distribution

The basic evaluation player with the addition of the different reinforcement distribution procedures (Section 5.4) is rated. For each of the different distribution procedures a basic evaluation player with the addition of that procedure is rated.

7.3.2. Results

Table 7.5 and figure 7.2 show the player ratings of all the players rated in this experiment.

The results evince that the enhanced version of the evaluation player has a significantly improved playing skill. It is also easy to see that the plans have the most impact on the improvement of the BEP, each one of them increases the player rating from 1.2% to roughly 77%. Surprisingly, the combination of all three plans leads only to a slightly higher rating as each plan on its own.
In comparison to these improvements the impact of the target continent might seem small, but nevertheless it results in a victory probability almost 20 times as high as the one of the BEP.
The least impact is achieved by the different reinforcement distribution procedures. Even though an increase in the player rating can be measured, the improvement is, especially in relation to the other enhancements, way behind my expectations. In the case of the Quarter-One-Max procedure, the player rating is even less than the one of the BEP.

7.4. TD Learning

To be able to measure the success of the learning player, I let several players train on a number of games and rate them during the course of their progres.

Player	Won	Played	Rating
Basic Evaluation Player	3	1,000	1.2%
BEP with TC	56	1,000	22.4%
BEP with PEP	391	2,000	78.2%
BEP with AP	374	2,000	74.8%
BEP with CCP	385	2,000	77.0%
BEP with all Plans	201	1,000	80.4%
BEP with One-Max	9	1,000	3.6%
BEP with Half-Quarter-One-Max	6	1,000	2.4%
BEP with Quarter-Quarter-One-Max	5	1,000	2.0%
BEP with Quarter-One-Max	1	1,000	0.4%
Benchmark Player	—	—	100.0%

Table 7.5.: Summary of the Player Ratings of the BEP Enhancements

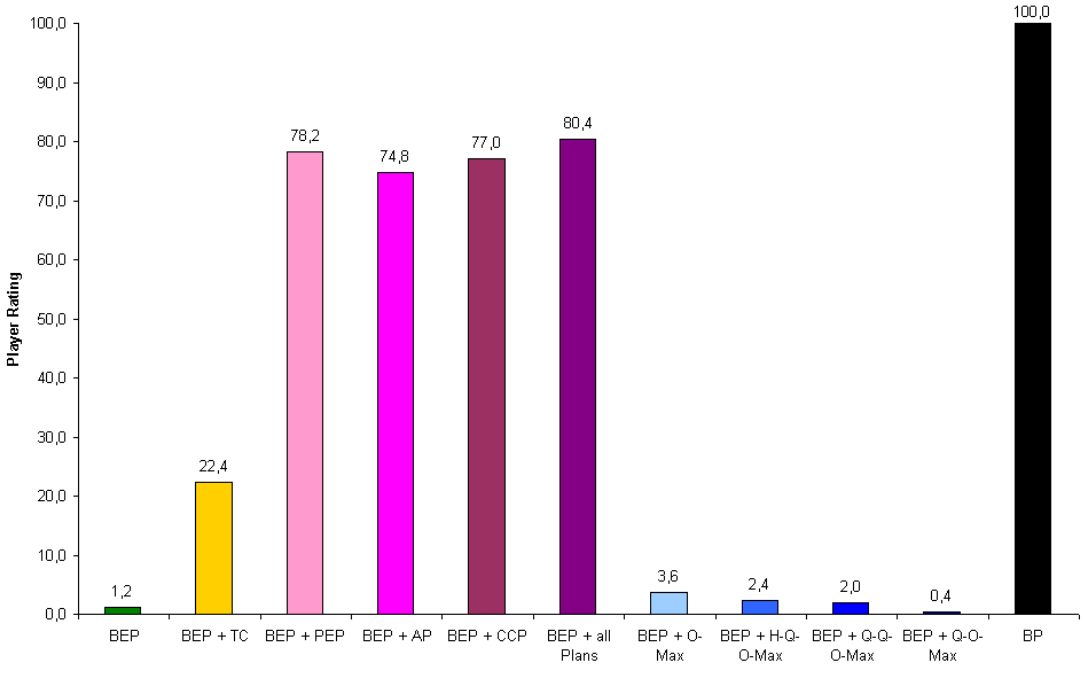

Figure 7.2.: Summary of the Player Ratings of the BEP Enhancements

Player	α	λ
Learning Player Red	$1/\lceil n/1,000 \rceil$	0.5
Learning Player Black	$\frac{1}{n}$	0.5
Learning Player Yellow	0.25	0.5
Learning Player Blue	if n $<$ 2,000: $1 - 0.00025 \times n$ else: $1/\lceil n/2,000 \rceil$	0.5

Table 7.6.: Players Participating in the Training Games. With n the number of games the learning player has already experienced.

7.4.1. Methods

This experiment consists mainly of two parts. First, the training of the learning players and second, the rating of the players at certain steps in the course of the training.

Training

I let four different learning players, each with a different α-function, learn on 10,000 training games. All of the learning players use the One-Max reinforcement distribution procedure. The training games are played by six players, four independent instances of the learning players and two benchmark players. This is done to average the impact of different outcomes of the training games on the learning process of the players.

Table 7.6 summarises the settings for the four learning players while figure 7.3 shows a graph of the different alpha functions.

Rating

I rate all learning players every 1,000 training games, with all ratings based on test runs consisting of 250 games each.

7.4.2. Results

Table 7.7 and figure 7.4 compare the average player ratings of the various learning players.

The detailed results of the individual players are presented in Appendix B and the development of the averaged feature weights during the learning process can be found

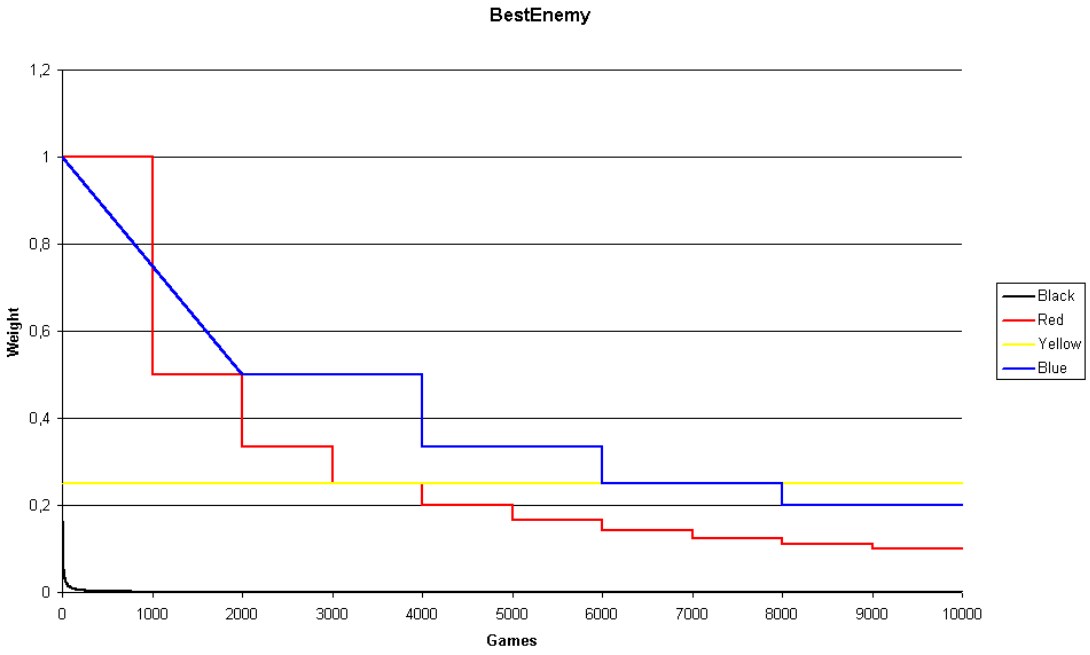

Figure 7.3.: Graphs of the *alpha* functions of the different learning players

in Appendix C.

When comparing the learning curves of the four players it can be seen that the players Red, Yellow and Blue all three increase their player rating by a minimum of 20%, while player Black does not seem to make any progres. This is due to fact that the alpha function of player black approaches zero very fast. By analysing the feature weight changes[2] of player Black I realised that significant changes were made in the first 50 games only. After that many games the alpha value was too low to allow the learning algorithms to make any considerable change to the feature weights. The feature weights of the other players (Appendix C) on the other hand all seem to converge to the same values.

In general, the feature weights do not seem to have as big an effect as I thought. Most probably this is due to the fact that the plans (Section 5.3) do not rely on the evaluation function and have a major impact on the playing strength of the players (Section 7.3.2).

[2]Because the weight changes of player Black do not offer anything interesting besides this conclusion, they are omitted in this work.

Experience	Player			
	Red-⌀	Black-⌀	Yellow-⌀	Blue-⌀
0,000	85.2%	90.4%	66.8%	84.0%
1,000	98.4%	75.6%	114.8%	94.4%
2,000	86.0%	86.4%	98.0%	89.2%
3,000	89.2%	90.0%	98.4%	97.6%
4,000	90.4%	90.0%	111.6%	86.0%
5,000	101.2%	98.4%	106.0%	88.8%
6,000	100.4%	100.0%	110.0%	92.0%
7,000	110.8%	86.0%	98.0%	103.2%
8,000	101.6%	92.8%	100.8%	101.2%
9,000	100.8%	88.4%	106.8%	99.6%
10,000	108.4%	92.8%	100.8%	104.0%

Table 7.7.: Average Player Ratings of the Various Learning Players

7.5. Human Opponents

To be able to compare the artificial players with external ones I rated the playing strength of several human players using the rating system introduced in Section 7.2.

7.5.1. Methods

Rating human players accurately is more difficult than rating artificial players. Naturally, the number of games played in the experiments is rather limited compared to the number of games played in the rating of artificial players. Furthermore, human players themselves are very heterogeneous in their playing strength.

In total I rated eight different human players. To get an indication of their playing skills and experience, I let every human classify himself according to his Risk experience either as a *beginner* or an *advanced* player.

7.5.2. Results

Table 7.8 shows the player ratings of the different human players I rated while figure 7.5 depicts a comparison between the benchmark player and the human players.

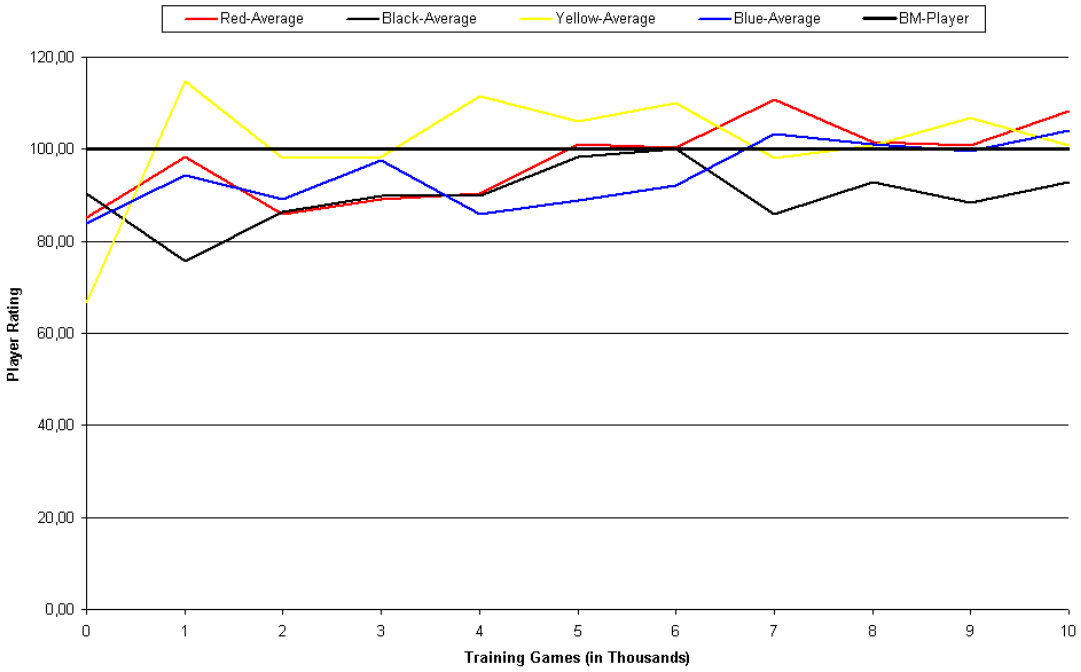

Figure 7.4.: Average Player Ratings of the Various Learning Players

The result from the experiment shows that the average human player has a higher playing rating than the benchmark player. Considering the deficiencies of the evaluation player (Section 5.1), even though lessened by the enhancements of the EEP, this outcome is not surprising. On the contrary, it is amazing to realise that a relative simple player, who does not look ahead even a single decision, is capable of regularly defeating the average human novice and able to beat an experienced human player.

During the course of the experiment I got the impression that the human players (especially the advanced ones) quickly recognize patterns in the actions of the benchmark player and start to adapt their strategy according to their predictions of the artificial players behaviour. While being far from perfect, I generally perceived an improvement of the playing strength of the human players as a consequence of this adaption. Limited by the insufficient number of test games, I did not research this topic any further.

If further games including human players should be done, it would be advisable to implement a more user-friendly graphical user interface. Currently there is only a very basic and tiresome text-based interface for the human player 3.6.2.

Player	Risk-Experience	Won	Played	Rating
Human 1	Beginner	1	8	50.0%
Human 2	Beginner	1	9	44.4%
Human 3	Beginner	2	7	114.3%
Human 4	Advanced	6	20	120.0%
Human 5	Advanced	3	8	150.0%
Human 6	Advanced	5	8	250.0%
Human 7	Advanced	5	10	200.0%
Human 8	Advanced	2	5	160.0%
All Beginners	—	4	24	66.7%
All Advanced	—	21	51	164.7%
All Players	—	25	75	133.3%
Benchmark Player	—	—	—	100.0%

Table 7.8.: Player Ratings of Several Human Players

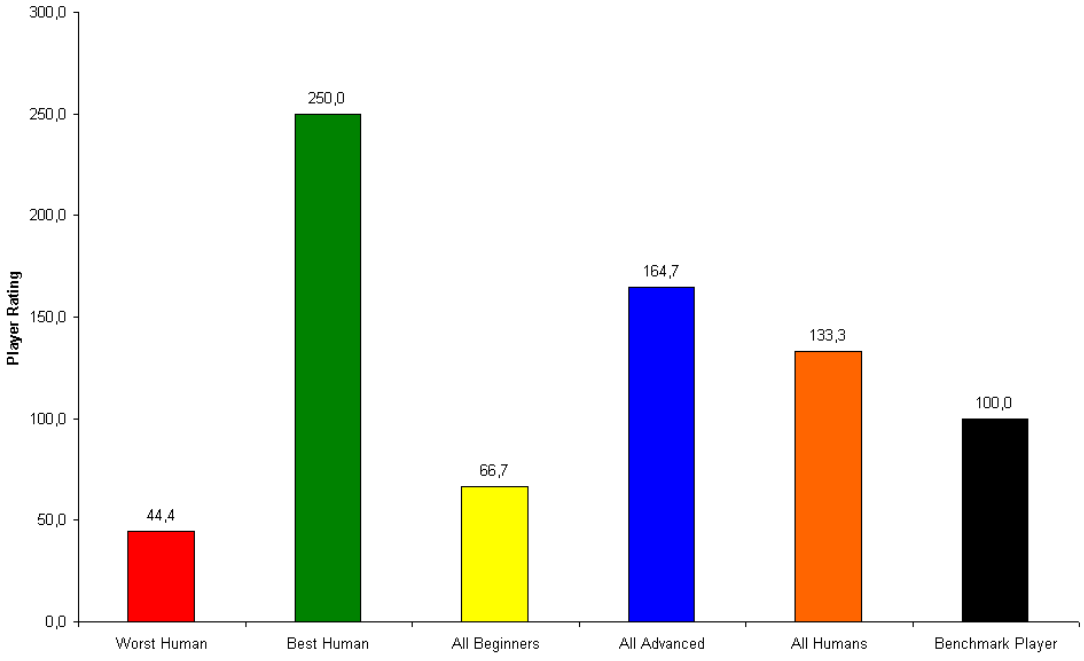

Figure 7.5.: Comparison of Human Players with the Benchmark Player

8. Conclusion

Risk is a very complex game, with both infinite game-state and game-tree complexities. Even calculating a good approximation of the average complexity of the game proves to be complex.

My work has shown that it is crucial in Risk to combine several actions to achieve a high-level goal. The addition of just a single pre-defined high-level goal paired with the means to combine several actions to achieve it, has increased the probability to win a game by more than 60 times (Section 7.3.2), while the addition of just a high-level goal has increased the probability by almost 20 times.

Even though the enhanced evaluation player significantly surpasses its predecessor, it does not eliminate the deficits of the basic player, but rather strives to eliminate the most apparent symptoms of these deficits.

Looking ahead is not likely to solve the flaws, because the complexity of Risk makes it infeasible to look ahead more than a few (simplified) actions. But, of course, it will surely further increase the playing strength of the player.

I believe that the key to building very strong Risk players are dynamic plans. Instead of having three pre-defined high-level goals, the player would have to define reasonable goals[1] on its own using the evaluation function. Then a plan would have to be created that would connect the actions of the player in a way that provides the highest probability of achieving that goal. Of course, after each decision the situation would have to be re-evaluated, though that might prove to be infeasible.

The results of TD learning seem very promising, even though the player rating was only increased by an average 20%. As explained in Section 7.4.2, I believe this is due to the little impact the evaluation function has on the overall playing strength of the EEP.

[1] In this context a goal is not limited to a single favourable fact, i.e. eliminating an enemy player, but it can be any combination of such fovourable facts, i.e. eliminating an enemy player and destroying the occupation of a continent by an enemy player and completely conquering a continent for the actual player.

If the evaluation function determined for which high-level goal a plan would be made, the evaluation function, and subsequently TD learning, too, would have a major effect on the playing strength.

A. Complexity Measurement Histograms

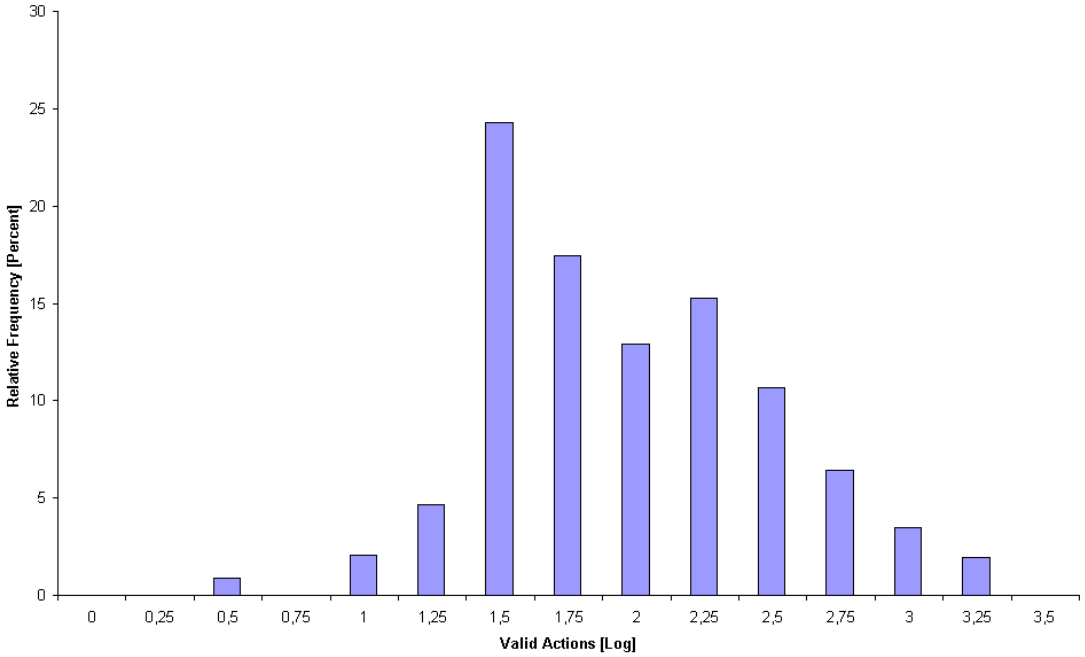

Figure A.1.: Distribution of the valid actions of the *Place Armies* decisions as implemented in the Risk Framework

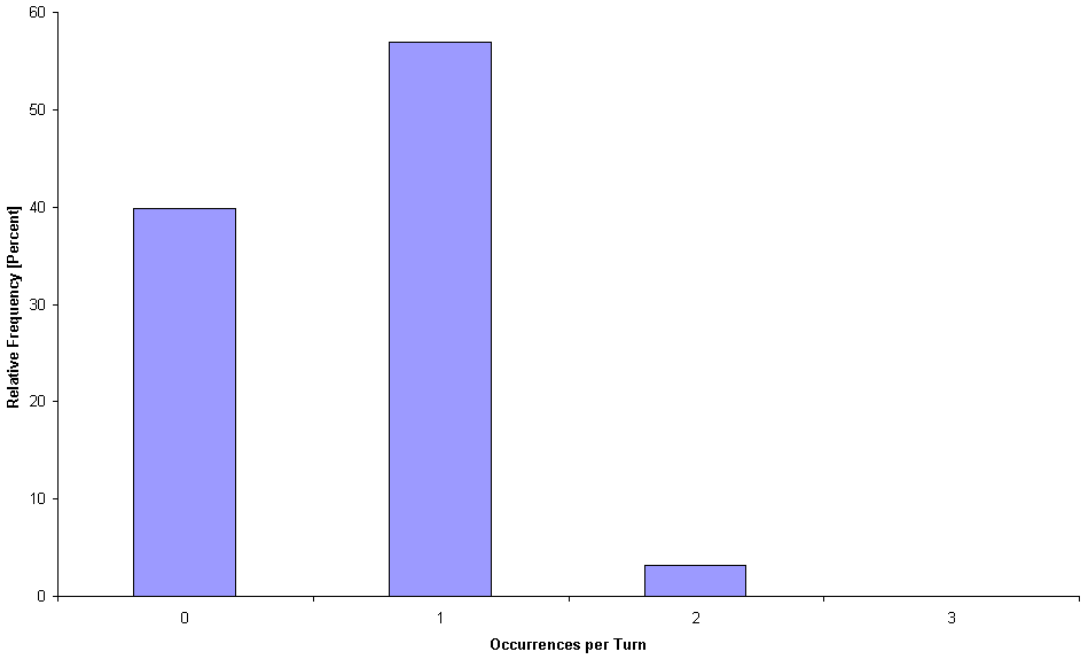

Figure A.2.: Distribution of the occurrences of the *Trade* decision

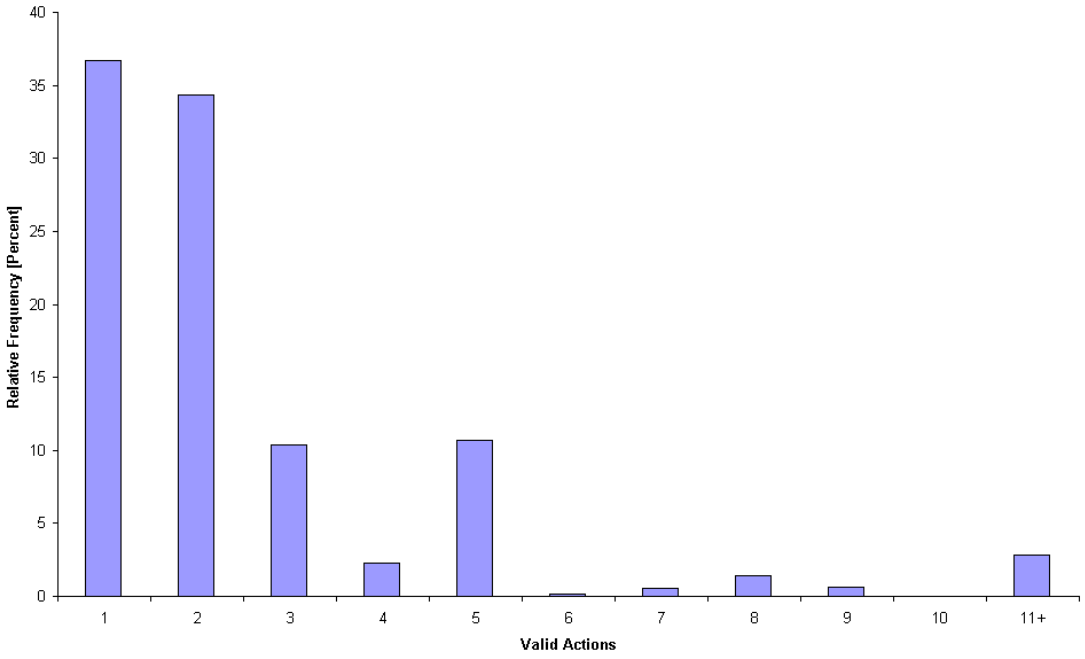

Figure A.3.: Distribution of the valid actions of the *Trade* decisions

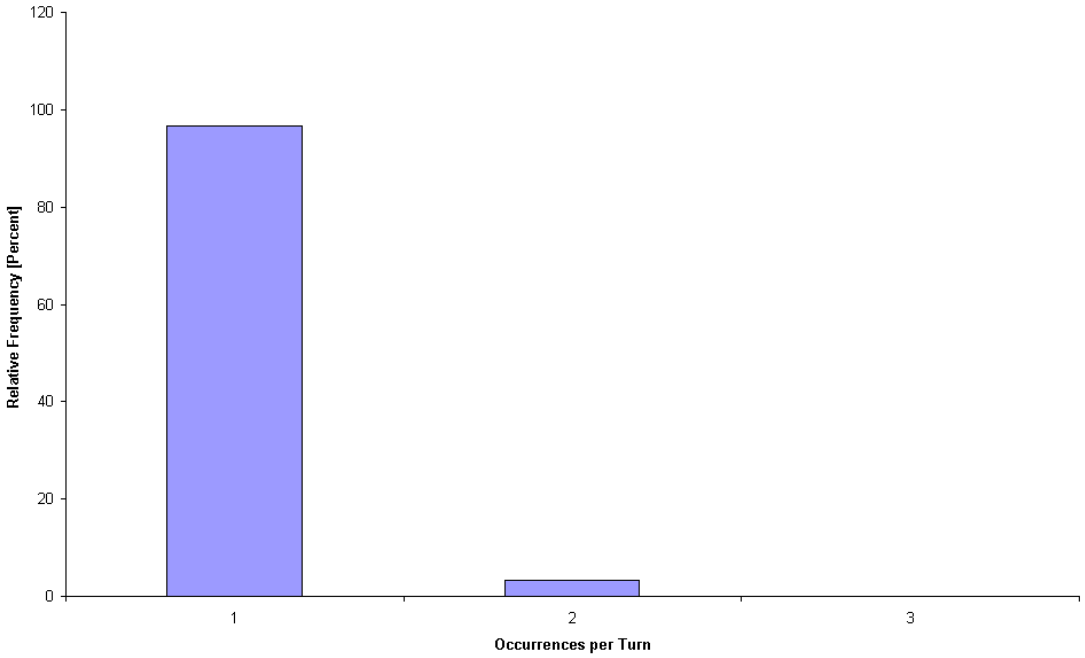

Figure A.4.: Distribution of the occurrences of the *Place Armies* decision

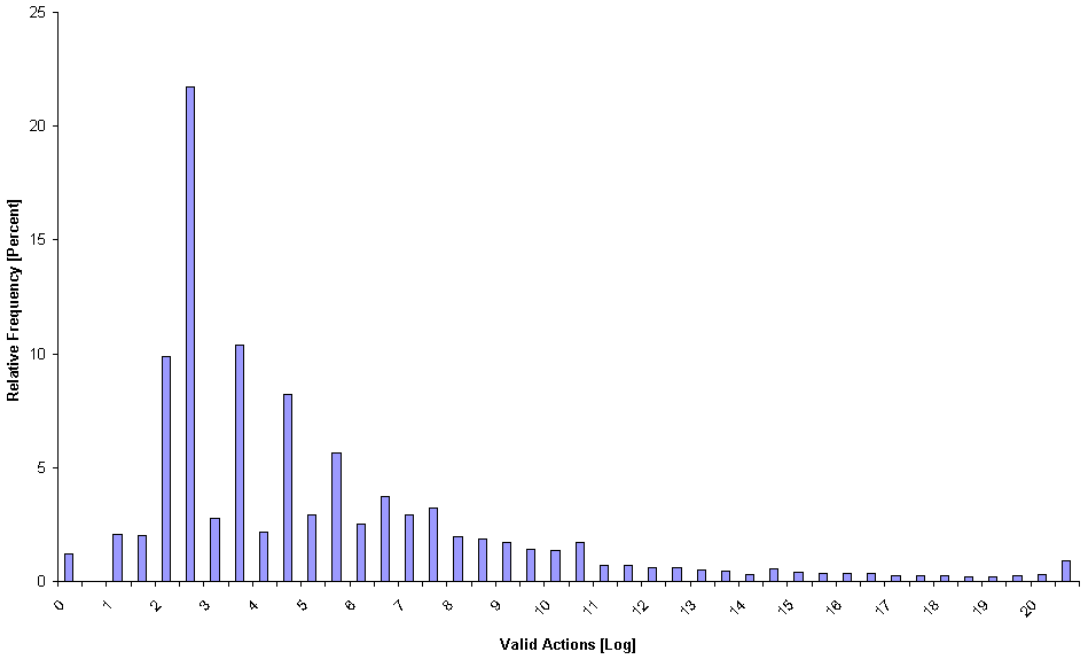

Figure A.5.: Distribution of the valid actions of the *Place Armies* decisions

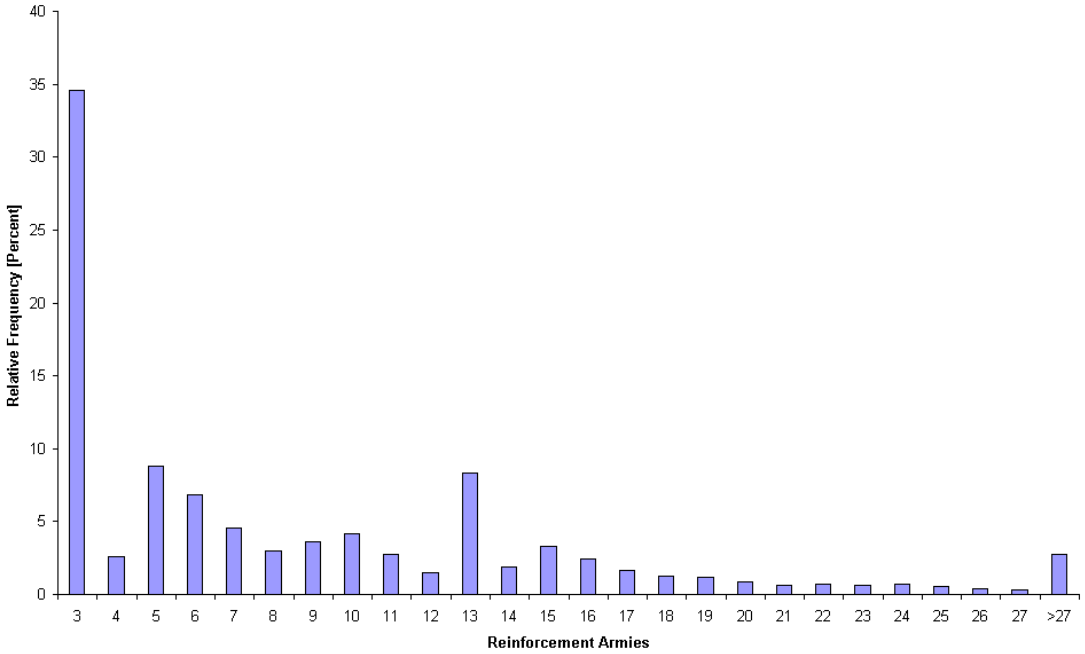

Figure A.6.: Distribution of the Number of Reinforcement Armies in the *Place Armies* decision

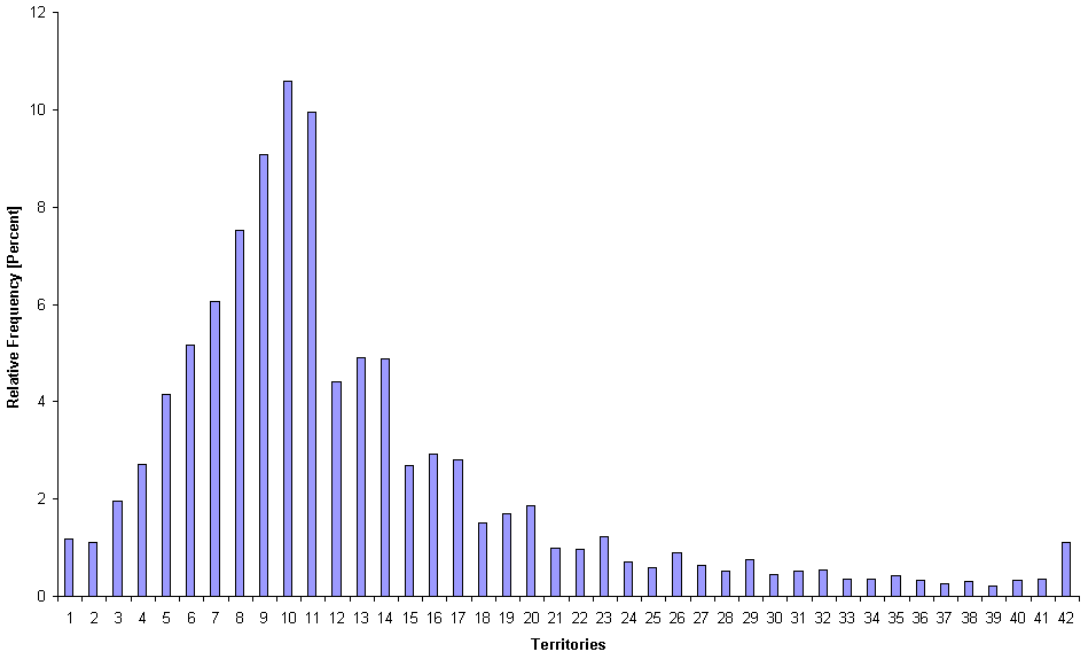

Figure A.7.: Distribution of the Number of Occupied Territories in the *Place Armies* decision

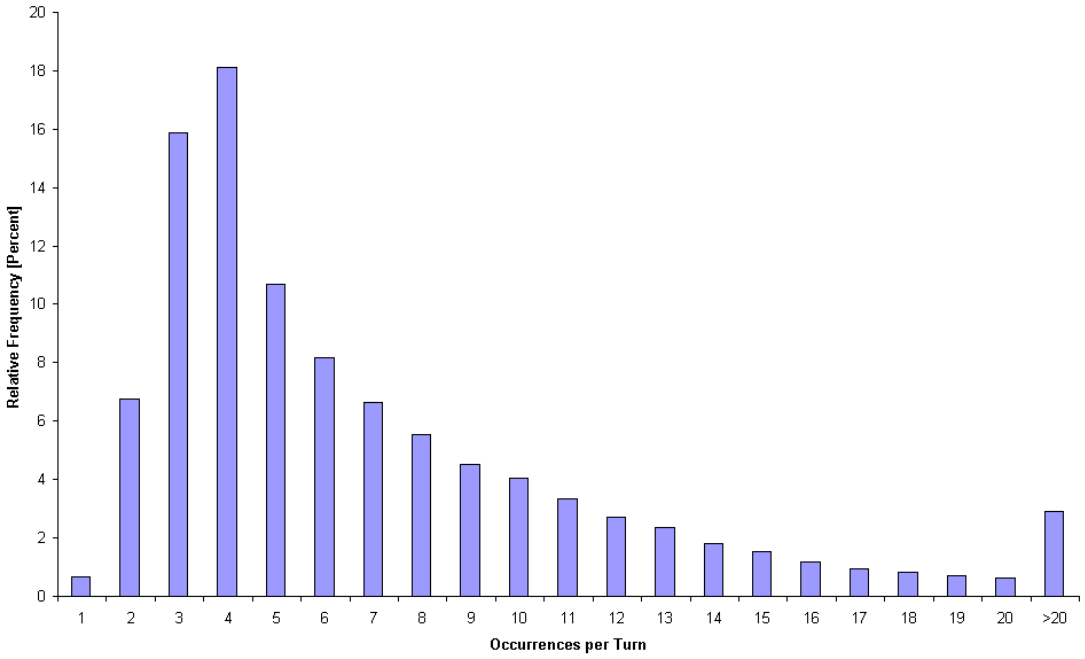

Figure A.8.: Distribution of the occurrences of the *Attack* decision

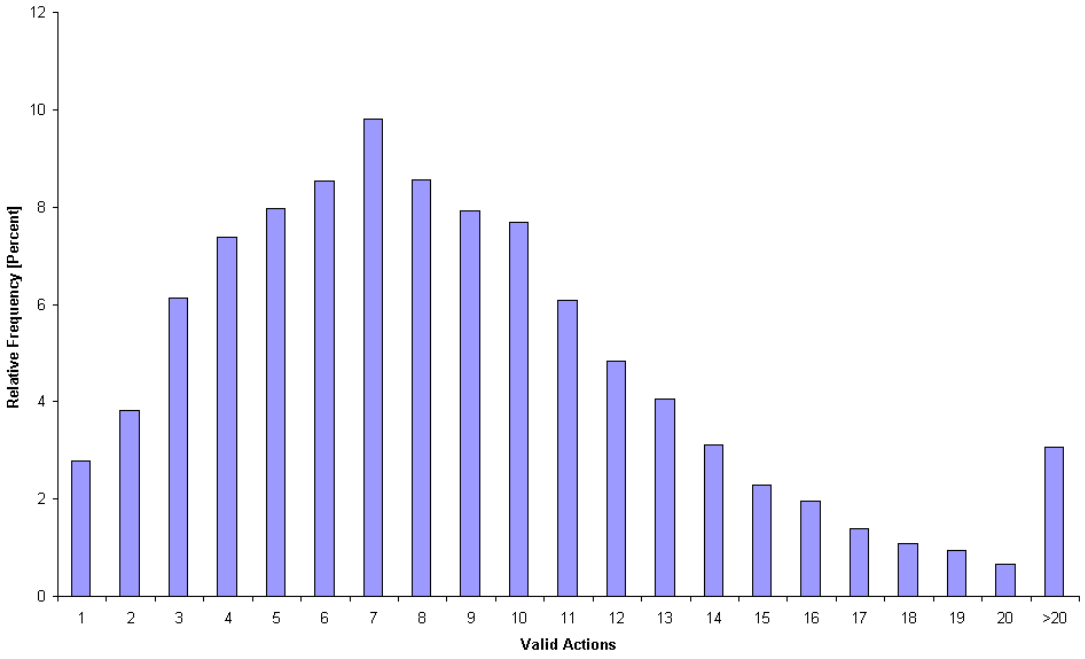

Figure A.9.: Distribution of the valid actions of the *Attack* decisions

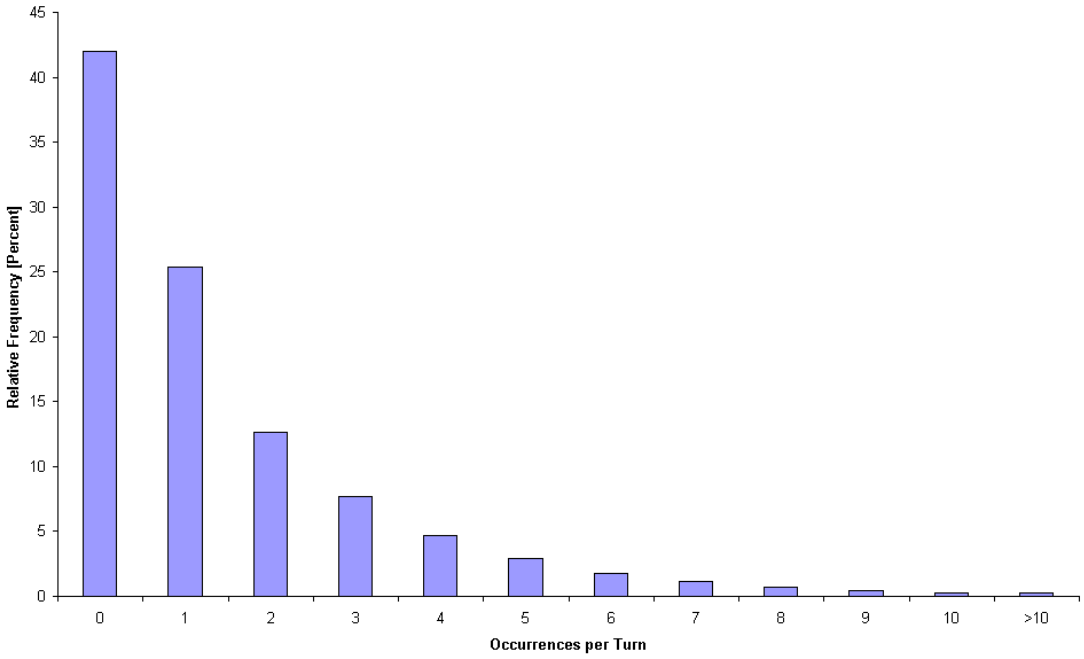

Figure A.10.: Distribution of the occurrences of the *Combat Move* decision

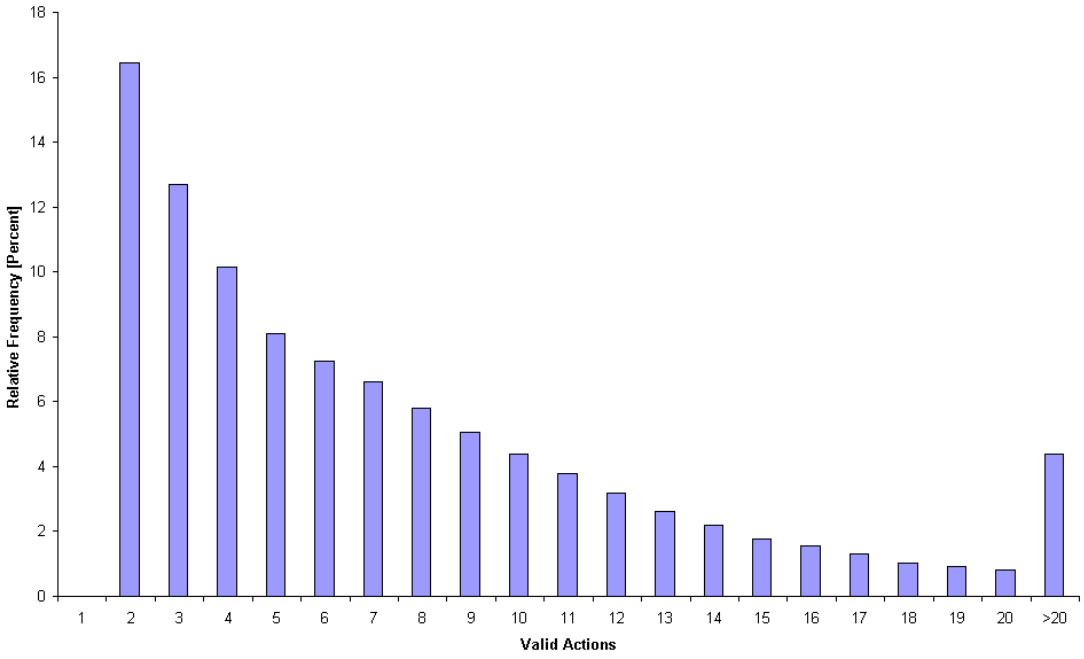

Figure A.11.: Distribution of the valid actions of the *Combat Move* decisions

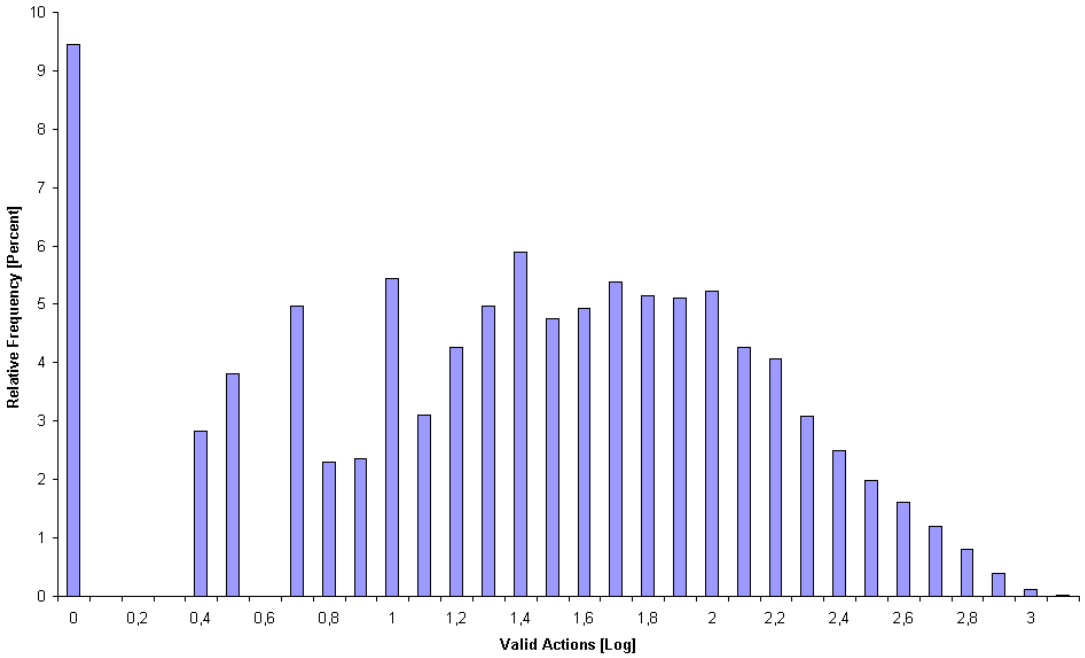

Figure A.12.: Distribution of the valid actions of the *Fortify Position* decisions

B. Detailed Learning Curves

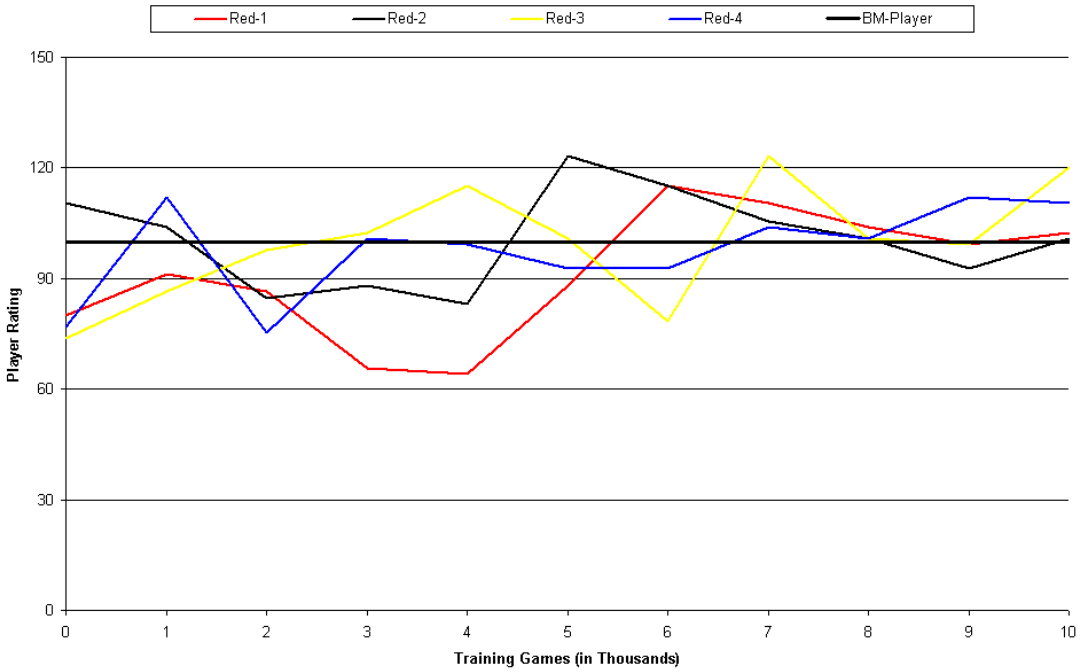

Figure B.1.: Player Rating of the Learning Player Red

Experience	Player			
	Red-a	**Red-b**	**Red-c**	**Red-d**
0,000	80.0%	110.4%	73.6%	76.8%
1,000	91.2%	104.0%	86.4%	112.0%
2,000	86.4%	84.8%	97.6%	75.2%
3,000	65.6%	88.0%	102.4%	100.8%
4,000	64.0%	83.2%	115.2%	99.2%
5,000	88.0%	123.2%	100.8%	92.8%
6,000	115.2%	115.2%	78.4%	92.8%
7,000	110.4%	105.6%	123.2%	104.0%
8,000	104.0%	100.8%	100.8%	100.8%
9,000	99.2%	92.8%	99.2%	112.0%
10,000	102.4%	100.8%	120.0%	110.4%

Table B.1.: Player Ratings of the Learning Player Red

Experience	Player			
	Black-a	**Black-b**	**Black-c**	**Black-d**
0,000	107.2%	102.4%	88.0%	64.0%
1,000	78.4%	86.4%	84.8%	52.8%
2,000	108.8%	76.8%	105.6%	54.4%
3,000	89.6%	113.6%	86.4%	70.4%
4,000	97.6%	86.4%	97.6%	78.4%
5,000	113.6%	88.0%	100.8%	91.2%
6,000	102.4%	128.0%	91.2%	78.4%
7,000	96.0%	100.8%	70.4%	76.8%
8,000	88.0%	118.4%	78.4%	86.4%
9,000	102.4%	81.6%	92.8%	76.8%
10,000	89.6%	110.4%	94.4%	76.8%

Table B.2.: Player Ratings of the Learning Player Black

Experience	Player			
	Yellow-a	**Yellow-b**	**Yellow-c**	**Yellow-d**
0,000	73.6%	86.4%	80.0%	27.2%
1,000	128.0%	113.6%	113.6%	104.0%
2,000	104.0%	91.2%	96.0%	100.8%
3,000	89.6%	81.6%	97.6%	124.8%
4,000	92.8%	94.4%	140.8%	118.4%
5,000	105.6%	94.4%	116.8%	107.2%
6,000	104.0%	121.6%	104.0%	110.4%
7,000	88.0%	104.0%	104.0%	96.0%
8,000	94.4%	121.6%	83.2%	104.0%
9,000	115.2%	102.4%	113.6%	96.0%
10,000	99.2%	91.2%	113.6%	99.2%

Table B.3.: Player Ratings of the Learning Player Yellow

Experience	Player			
	Blue-a	**Blue-b**	**Blue-c**	**Blue-d**
0,000	72.0%	75.2%	89.6%	99.2%
1,000	113.6%	72.0%	91.2%	100.8%
2,000	94.4%	84.8%	83.2%	94.4%
3,000	102.4%	94.4%	112.0%	81.6%
4,000	92.8%	91.2%	81.6%	78.4%
5,000	105.6%	81.6%	64.0%	104.0%
6,000	72.0%	89.6%	83.2%	123.2%
7,000	104.0%	118.4%	94.4%	96.0%
8,000	105.6%	110.4%	96.0%	92.8%
9,000	105.6%	78.4%	89.6%	124.8%
10,000	112.0%	96.0%	102.4%	105.6%

Table B.4.: Player Ratings of the Learning Player Blue

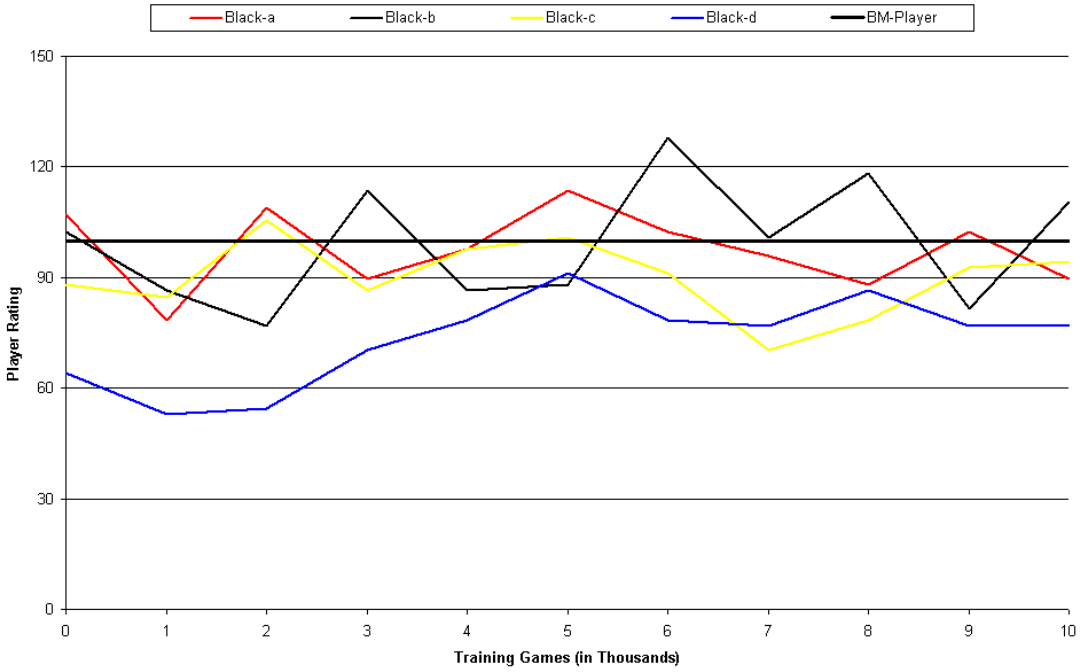

Figure B.2.: Player Rating of the Learning Player Black

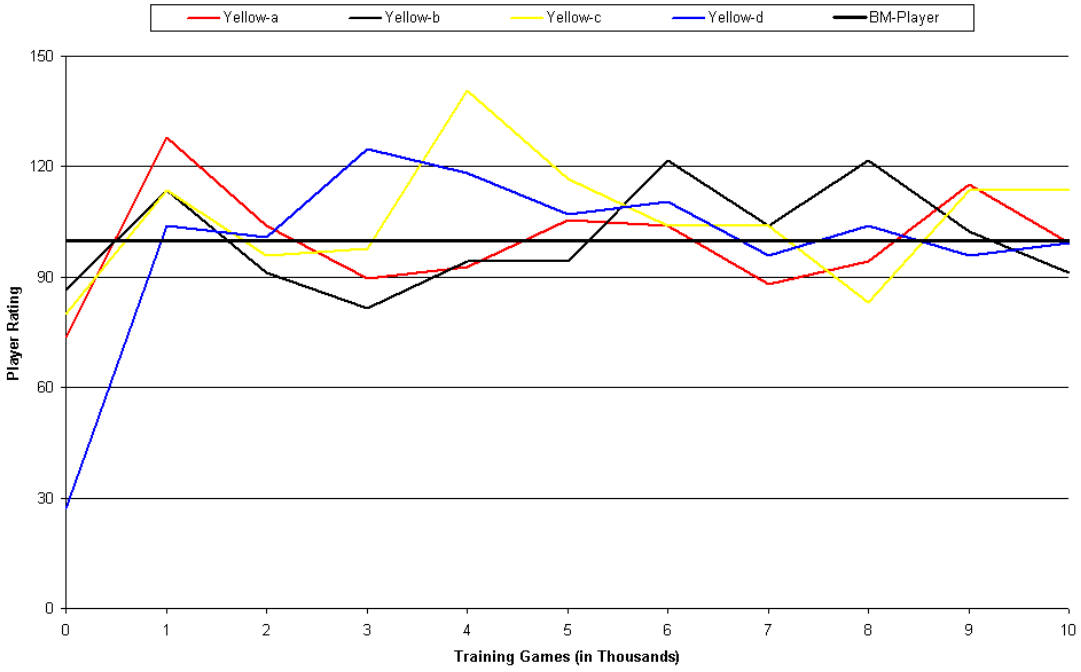

Figure B.3.: Player Rating of the Learning Player Yellow

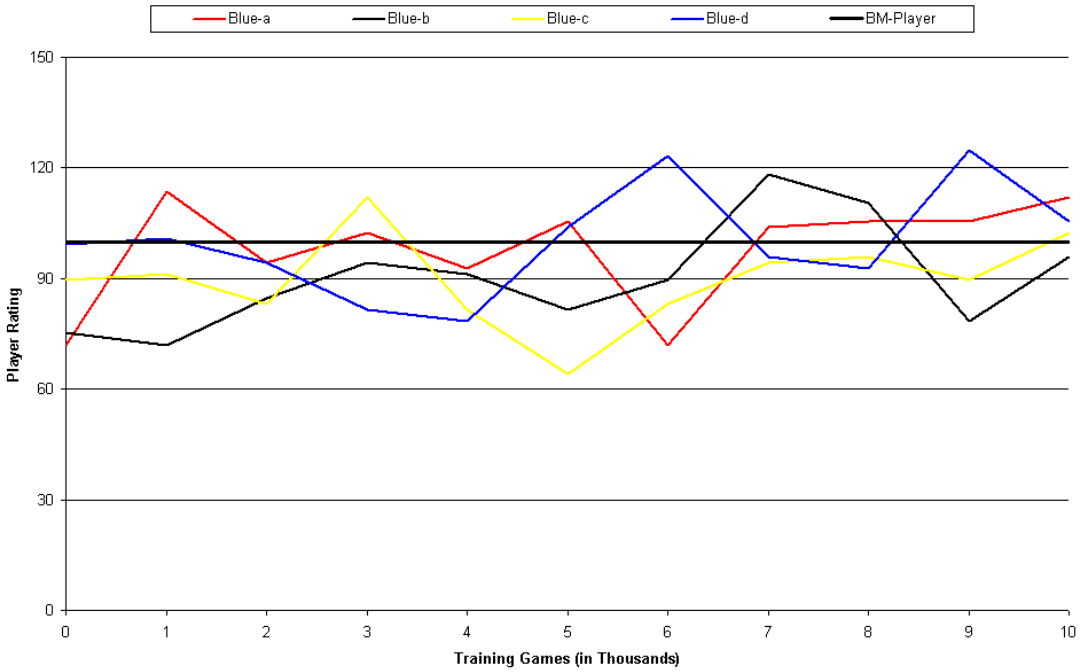

Figure B.4.: Player Rating of the Learning Player Blue

C. Feature Weight Changes

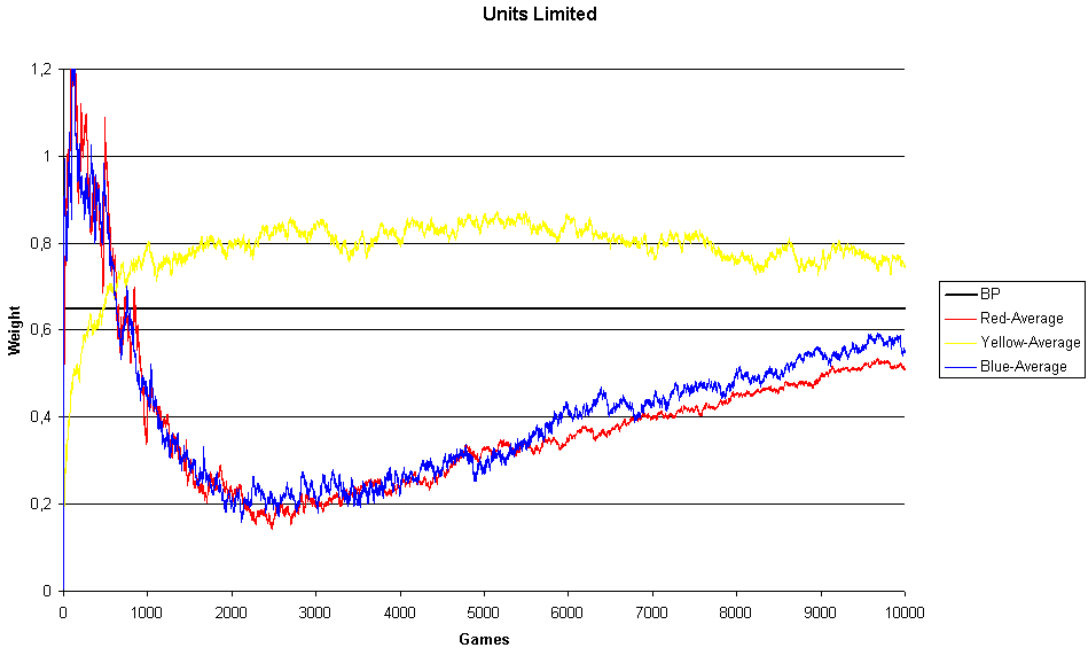

Figure C.1.: Development of the *Armies Feature*

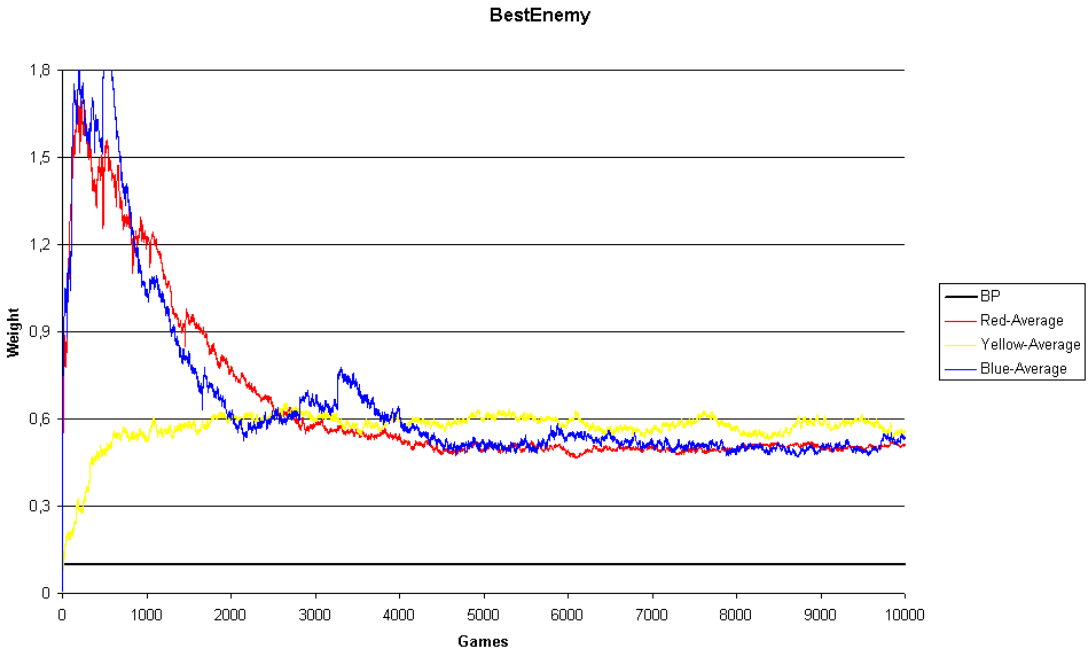

Figure C.2.: Development of the *Best Enemy Feature*

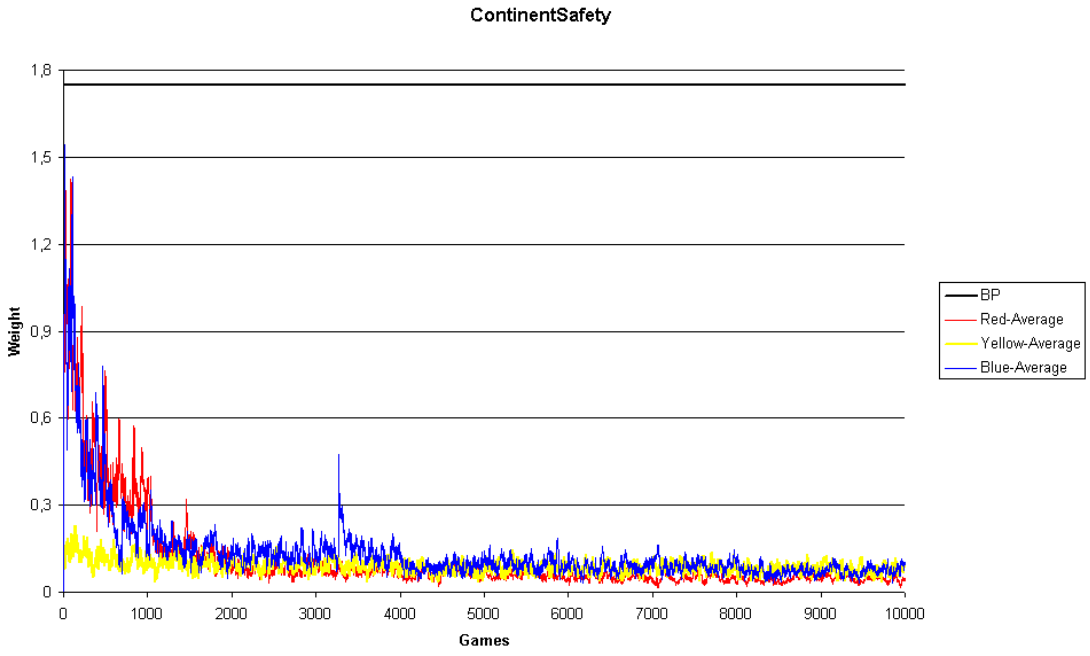

Figure C.3.: Development of the *Continent Safety Feature*

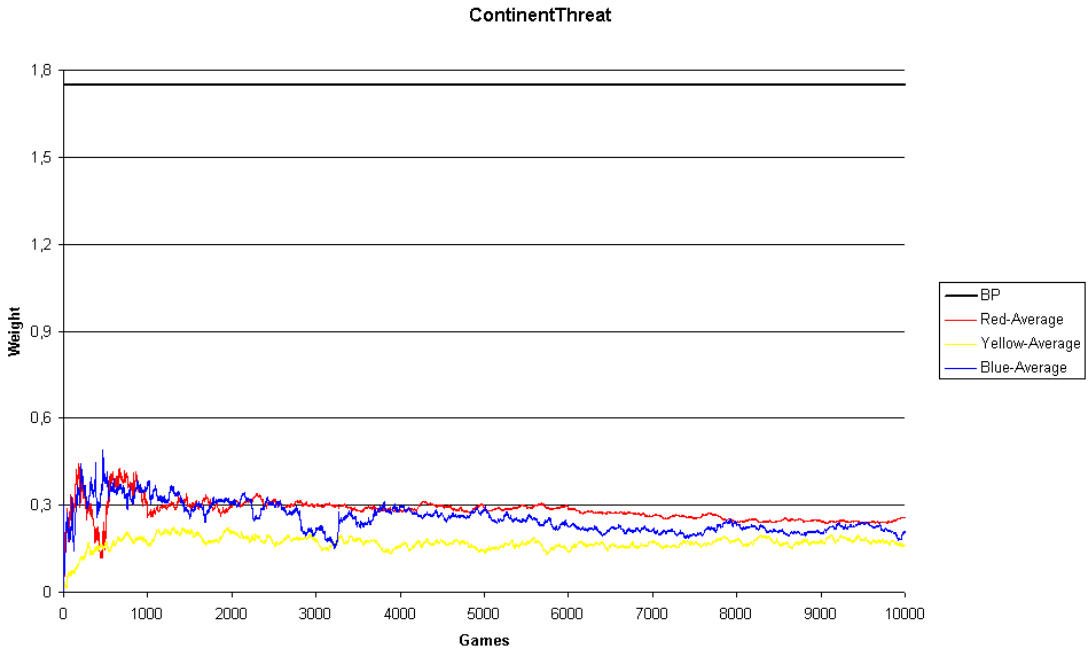

Figure C.4.: Development of the *Continent Threat Feature*

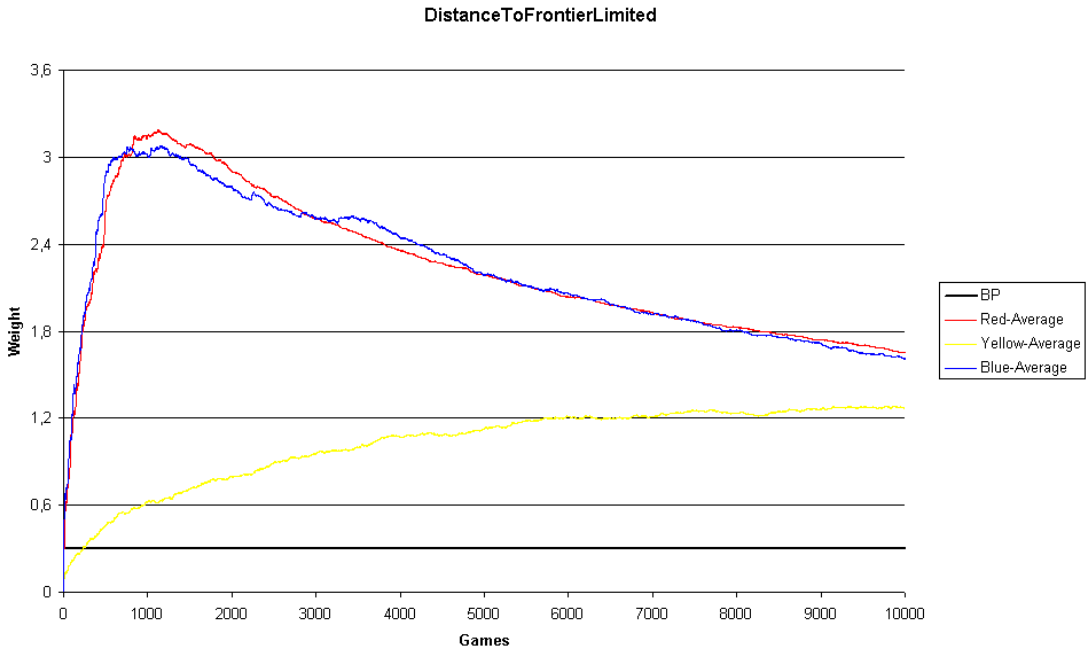

Figure C.5.: Development of the *Distance to Frontier Feature*

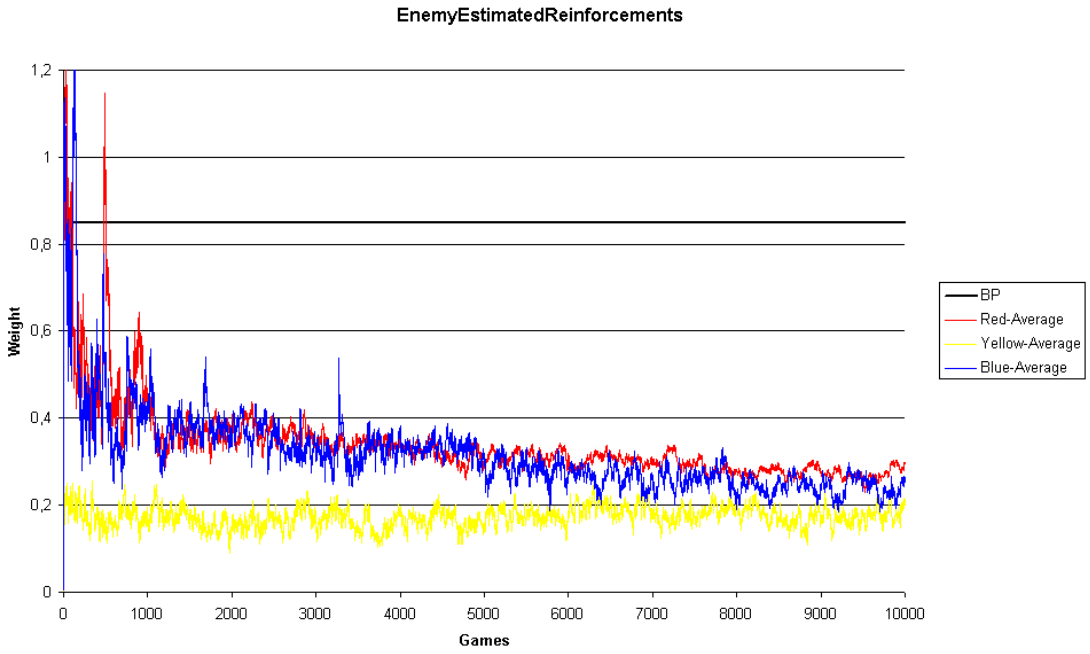

Figure C.6.: Development of the *Enemy Estimated Reinforcement Feature*

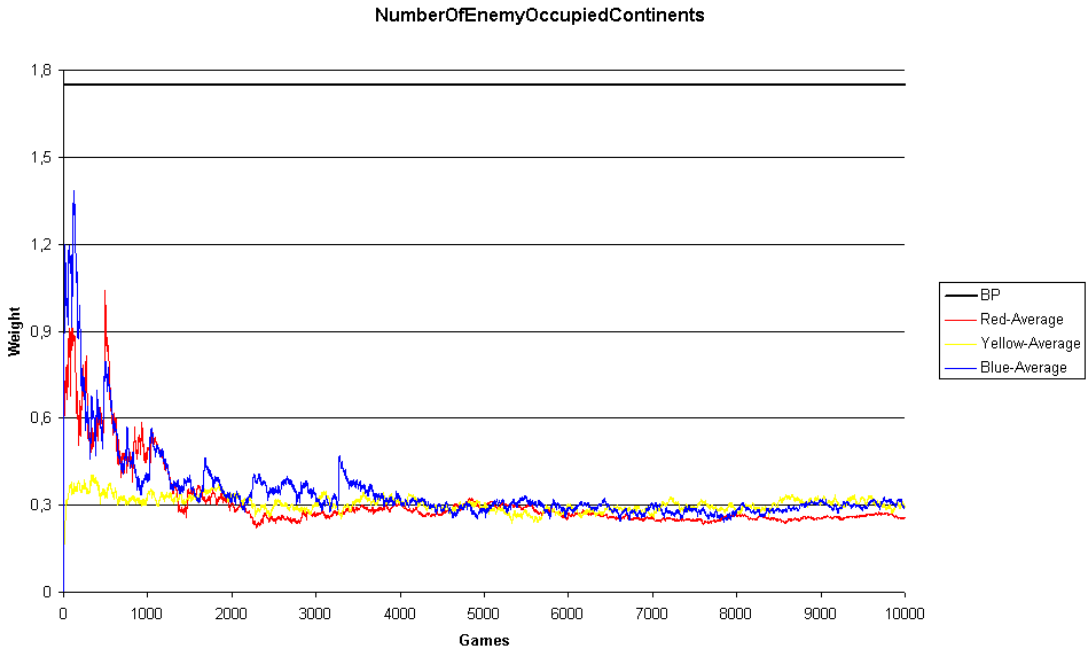

Figure C.7.: Development of the *Enemy Occupied Continents Feature*

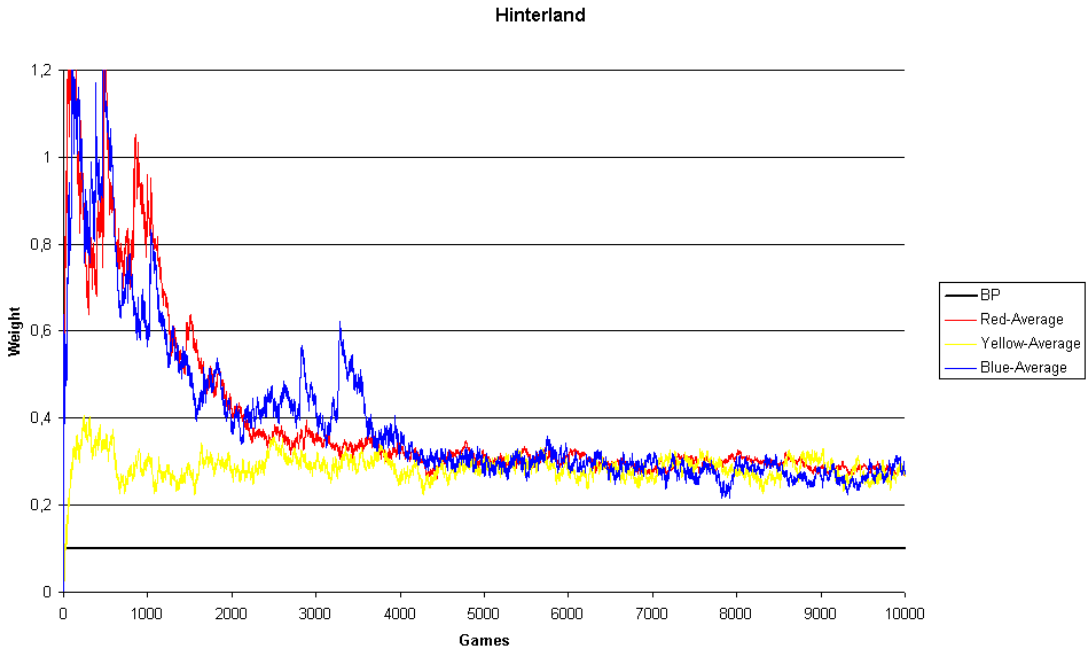

Figure C.8.: Development of the *Hinterland Feature*

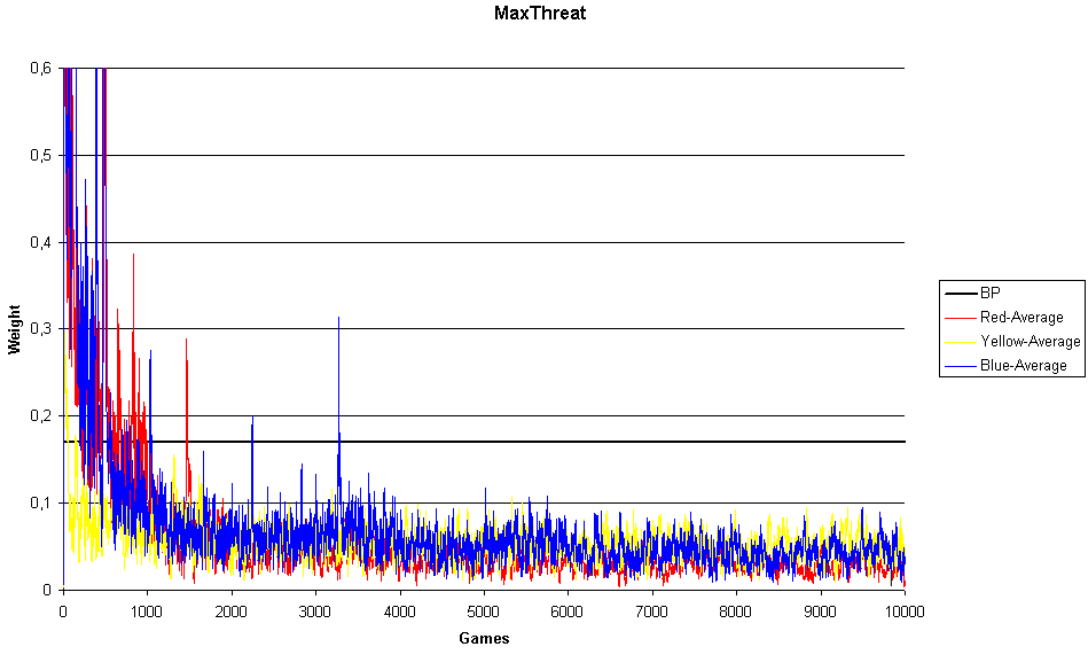

Figure C.9.: Development of the *Maximum Threat Feature*

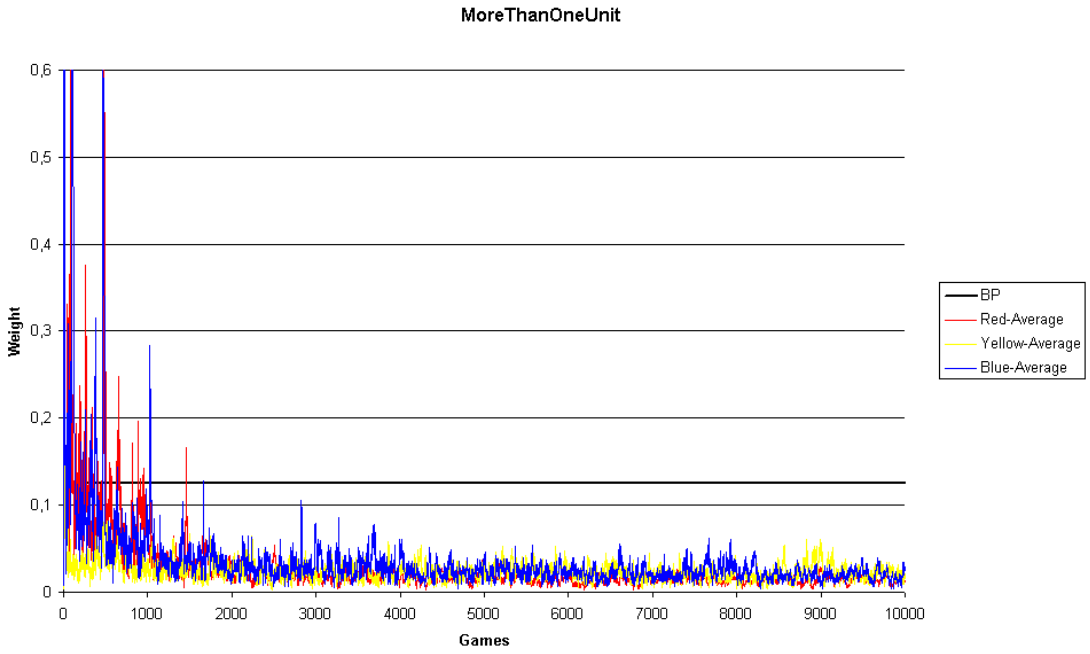

Figure C.10.: Development of the *More Than One Army Feature*

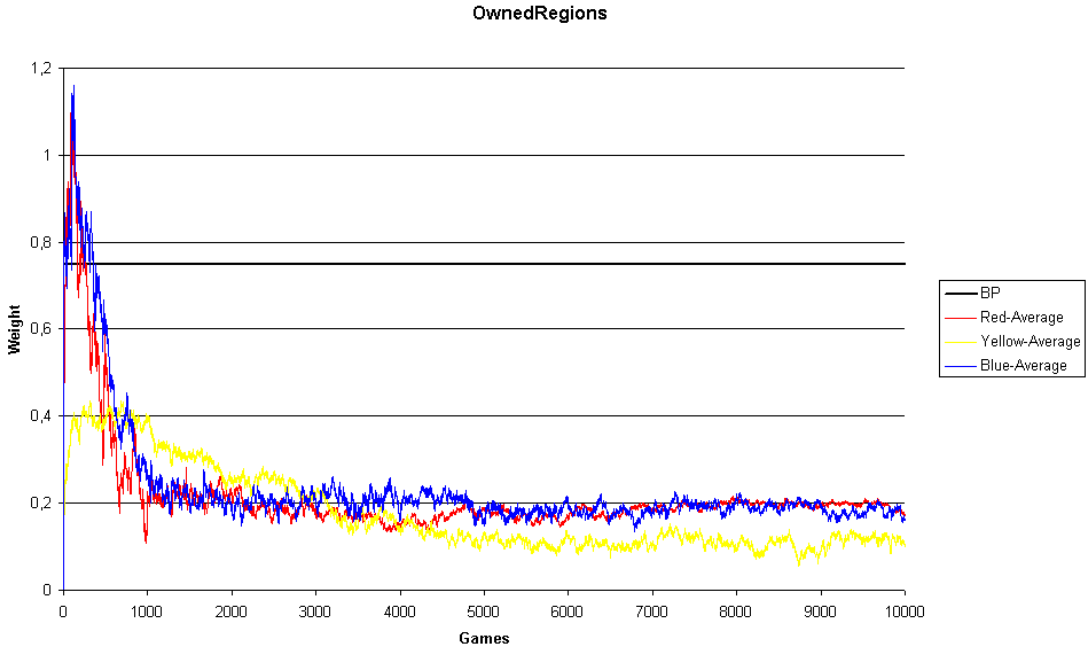

Figure C.11.: Development of the *Occupied Territories Feature*

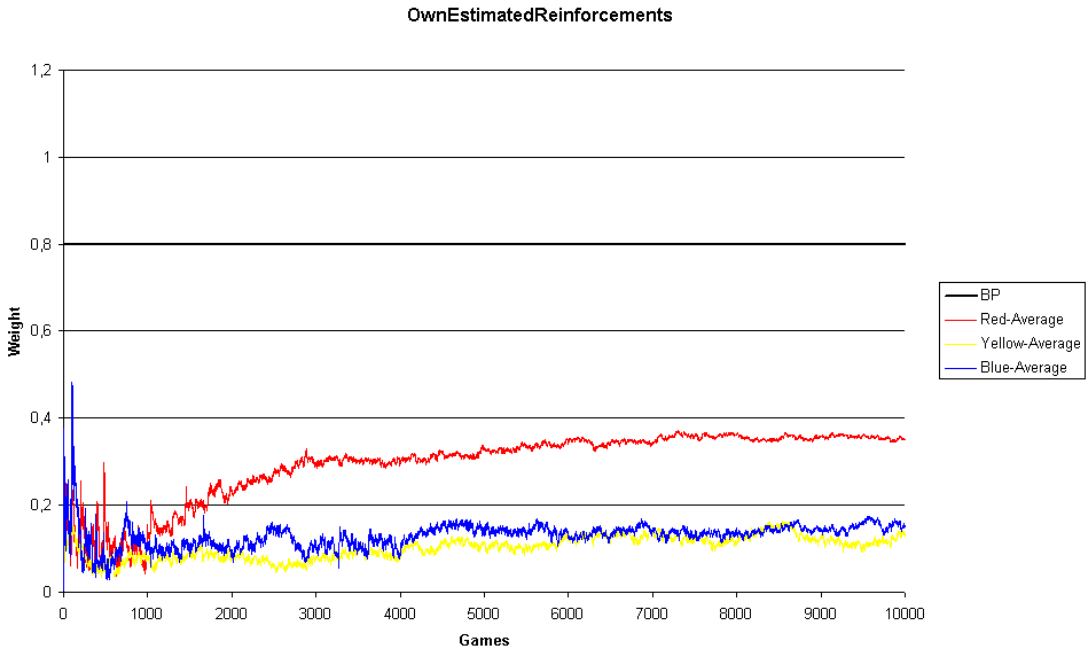

Figure C.12.: Development of the *Own Estimated Reinforcement Feature*

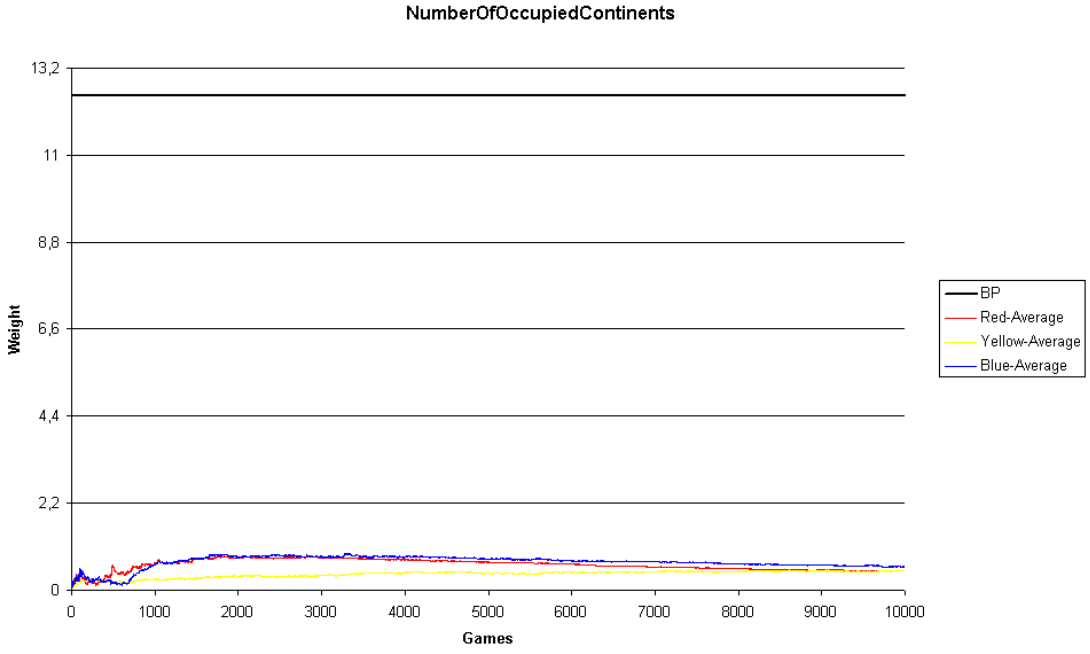

Figure C.13.: Development of the *Own Occupied Continents Feature*

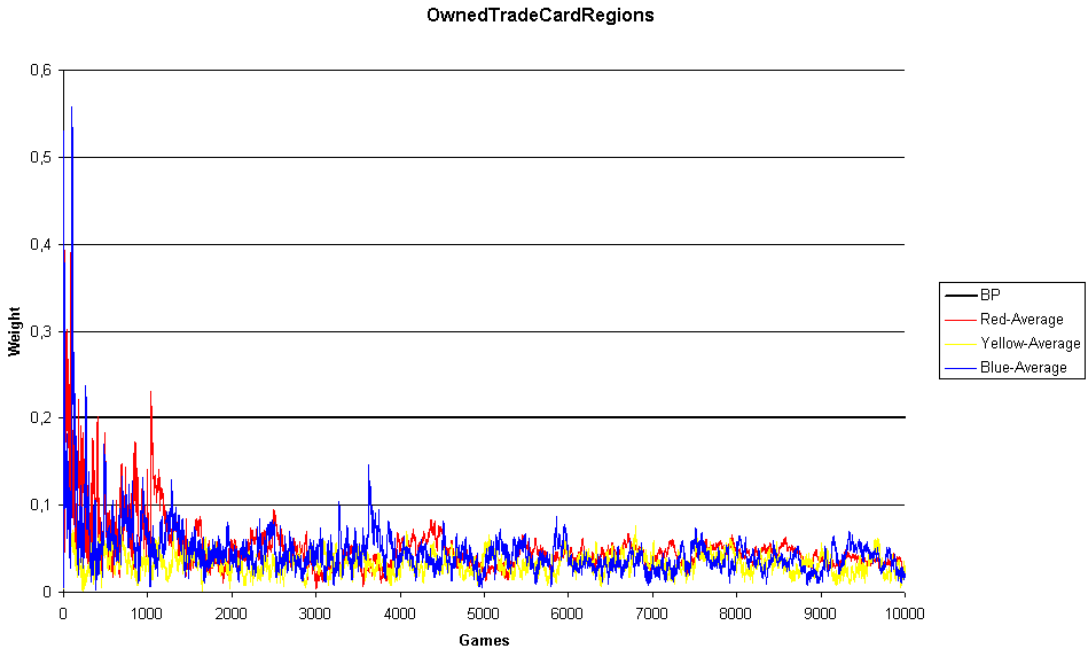

Figure C.14.: Development of the *Own Occupied Risk Card Territories Feature*

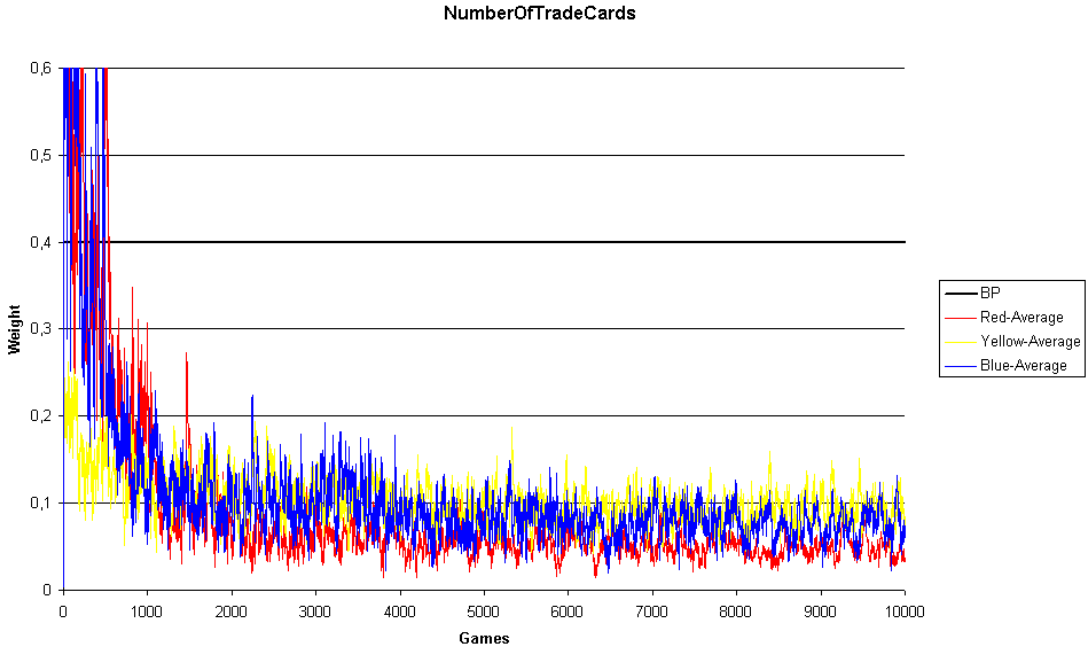

Figure C.15.: Development of the *Risk Cards Feature*

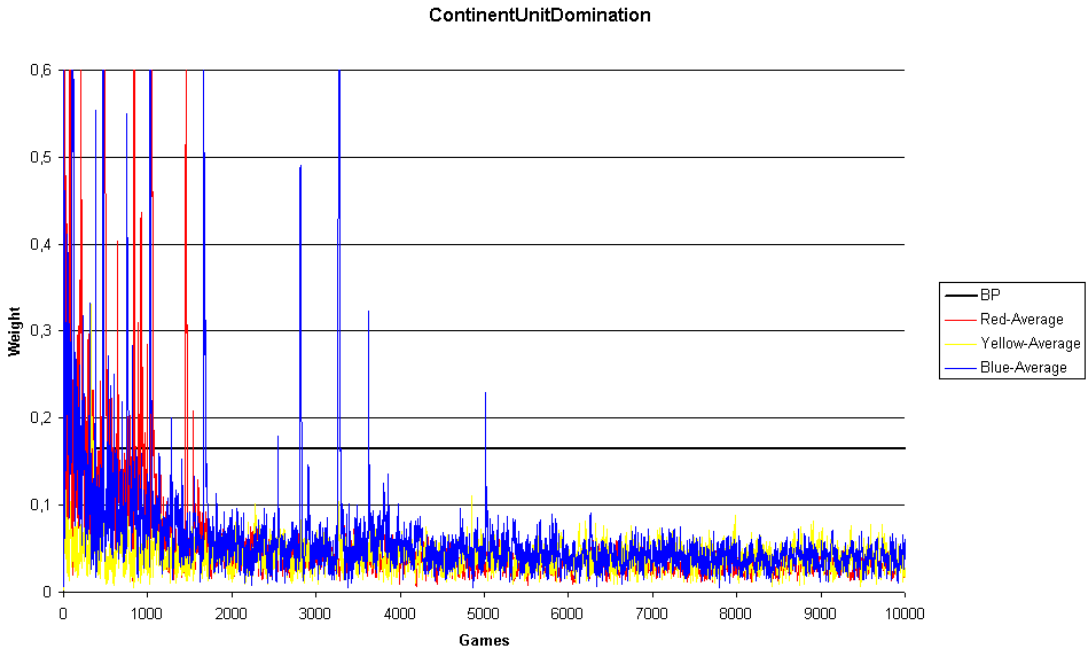

Figure C.16.: Development of the *Continent Army Domination Feature*

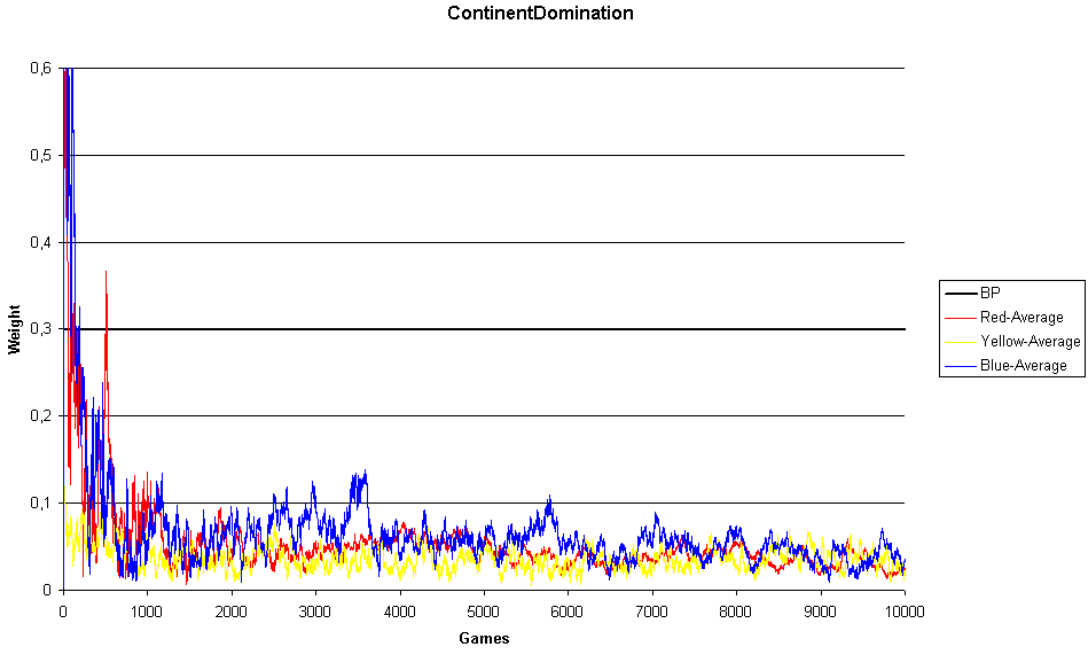

Figure C.17.: Development of the *Continent Domination Feature*

Bibliography

Sharon Blatt. Risky business: An in-depth look at the game risk. *Rose-Hulman Institute of Technology Undergraduate Mathematics Journal*, 3(2), 2002.

T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* MIT Press, 2001.

Johannes Fürnkranz. Machine learning in games: A survey. In Johannes Fürnkranz and Miroslav Kubat, editors, *Machines that Learn to Play Games*, chapter 2, pages 11–59. Nova Science Publishers, Huntington, NY, 2001. URL `http://www.ai.univie.ac.at/cgi-bin/tr-online?number+2000-31`.

Risiko de Luxe Spielanleitung. Hasbro Deutschland GmbH.

Risk Collector's 40th Anniversary Edition. Hasbro Inc., 1999.

Risk boardgame picture was published on the Hasbro website. Hasbro Inc., 2004. URL `www.hasbro.com/pl/page.viewproduct/product_id.9491/dn/games/default.cfm`.

H. Jaap van den Herik, Jos W. H. M. Uiterwijk, and Jack van Rijswijck. Games solved: Now and in the future. *Artificial Intelligence*, 134(1-2):277–311, 2002.

Owen Lyne, Leon Atkinson, Peter George, and Don Woods. Risk faq - version 5.51. http://www.maths.nottingham.ac.uk/personal/odl/riskfaq.html, 06 2004.

Jason A. Osborne. Markov chains for the risk board game revisited. *Mathematics Magazine*, 76(2):129–135, 2003.

Stuart Russell and Peter Norvig. *Artifical Intelligence: A Modern Approach,*. Prentice-Hall, Englewood Cliffs, NJ, ISBN 0-13-103805-2, 912 pp., 1995, 1995.

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

Richard S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3:9–44, 1988. URL `citeseer.ist.psu.edu/sutton88learning.html`.

Baris Tan. Markov chains and the risk board game. *Mathematics Magazine*, 70(5): 349–357, 1997.

Wikipedia. Game tree. http://en.wikipedia.org/wiki/Game_tree, 12 2004.

Acknowledgements

This work would not be what it finally has become if not for the help, guidance, counsel, encouragement and time of many people.

First of all, I want to thank Thorsten Nowak and Matthias Firner for countless hours of discussions about Risk strategies and design choices. I also thank my father, Günter Wolf, for his continual support. Furthermore, I want to emphasize the very good cooperation with my Professor, Johannes Fürnkranz. I could always discuss my ideas and was totally free in my decisions.

Finally, I want to thank all people who contributed to my work in any way, especially the volunteers participating in the Human Opponents experiment.

Alexander Kober, Christina Müller, Elena Lauer, Frank Pellkofer, Katja Kurz, Leonie von Bremen, Sabine Kaucic, Stefan Schwab, Stephan Remspecher, Stephanie Blank, Sven Rettig, Farrah Fritz and Thomas Fritz.

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Diplomarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, März 2005 Michael Wolf