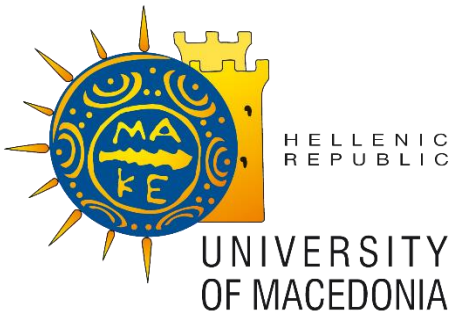


UNIVERSITY OF MACEDONIA
DOCTORAL STUDIES PROGRAMME
DEPARTMENT OF APPLIED INFORMATICS

Decision Making in Multiplayer Environments: Application in backgammon variants

Doctoral Thesis
of
Nikolaos Papahristou

Thessaloniki, 10/2015

DECISION MAKING IN MULTIPLAYER ENVIRONMENTS: APPLICATION IN BACKGAMMON VARIANTS

Nikolaos Papahristou

B.Sc. in Automation Engineering, Alexander Technological Educational Institute of Thessaloniki, 1998
M.Sc. in Applied Informatics, University of Macedonia, Greece, 2010

Doctoral Thesis

Submitted in partial fulfilment of the requirements for the

DEGREE OF DOCTOR OF PHILOSOPHY
IN APPLIED INFORMATICS

Supervisor: Dr. Ioannis Refanidis, Associate Professor

Approved by the 7-membered examining committee at /10/2015

Dr. Ioannis Refanidis Associate Professor	Dr. Nikolaos Samaras, Associate Professor	Dr. Ilias Sakellariou Lecturer
Dr. Georgios Stefanidis Professor	Dr. Maria Satratzemi, Professor	Dr. Angelo Sifaleras Assistant Professor
Dr. Konstantinos Vergidis Lecturer		

Acknowledgements

I would like to gratefully acknowledge the guidance, support and encouragement of my doctoral advisor, Dr. Ioannis Refanidis.

I would like to thank my wife, Spyridoula, for her long-lasting support and patience.

I would like to thank the Greek State Scholarship Foundation (IKY) for its financial support during my master's degree (2008-2010) as well as the first year of my PhD (2010-2011).

I would like to thank my friend and colleague Anastasios Alexiadis for lending computing power during the first year of my PhD when my computer back then was not powerful enough to conduct experiments in a timely fashion.

I would like to thank the people from all over the world who played against Palamedes and gave feedback in the form of bug reports or feature requests. Their support was critical in ensuing that Palamedes would compete without any problems in the backgammon competitions.

Finally, I would like to thank all the reviewers and editors of my published work for their helpful comments.

Ευχαριστίες

Θα ήθελα να ευχαριστήσω τον επιβλέποντα της διατριβής αυτής, Δρ. Ιωάννη Ρεφανίδη, για την καθοδήγηση, τη συμπαράσταση και την υποστήριξη που μου παρείχε κατά τη διάρκεια της διδακτορικής μου έρευνας.

Θα ήθελα να ευχαριστήσω τη σύζυγό μου, Σπυριδούλα, για τη μακρόχρονη υποστήριξη και υπομονή που επέδειξε.

Θα ήθελα να ευχαριστήσω το Ίδρυμα Κρατικών Υποτροφιών (ΙΚΥ) για την οικονομική βοήθεια που μου παρείχε μέσω υποτροφίας κατά τη διάρκεια του μεταπτυχιακού μου (2008-2010) αλλά και κατά τον πρώτο χρόνο της διδακτορικής μου έρευνας (2010-2011).

Θα ήθελα να ευχαριστήσω τον φίλο και συμφοιτητή, κ. Αναστάσιο Αλεξιάδη, για το δανεισμό υπολογιστή ισχύος κατά τον πρώτο χρόνο της διδακτορικής μου έρευνας, όταν ο υπολογιστής μου εκείνη την περίοδο δεν ήταν αρκετά δυνατός για την εκτέλεση των πειραμάτων μου σε εύλογο χρονικό διάστημα.

Θα ήθελα να ευχαριστήσω τους χρήστες του Παλαμήδη απ' όλον τον κόσμο για την ανατροφοδότηση που μου παρείχαν με τη μορφή επισημάνσεων, σφαλμάτων και αιτημάτων νέων δυνατοτήτων. Η συμβολή τους ήταν ιδιαίτερα καθοριστική για τη απρόσκοπτη συμμετοχή του Παλαμήδη στους διαγωνισμούς.

Τέλος, θα ήθελα να ευχαριστήσω όλους τους κριτές και επιμελητές των δημοσιεύσεών μου για τις εποικοδομητικές παρατηρήσεις τους.

Abstract:

Tesauro's TD-Gammon was the first major success of machine learning and artificial intelligence in general, when it demonstrated world-class performance against the human backgammon champion of that time. Even more impressively, the method used required little expert knowledge, relying on self-playing and training neural networks using reinforcement learning. However, apart from standard backgammon, several – yet unexplored – variants of the game exist, which use the same board, number of checkers and dice, but have different rules for moving the checkers, starting positions or movement direction.

In this thesis we focus our research on three such popular variants in Greece and neighboring countries, named Portes, Plakoto, and Fevga (collectively called Tavli). Motivated by the successful methods of TD-Gammon, we extend and devise new reinforcement learning methods for building artificial intelligent agents and show that expert-level play can also be achieved in these games. All the resulting agents created in this thesis are packaged in a freely available program, PALAMEDES, where everyone can play against the AI. To test the effectiveness of our approach, PALAMEDES participated in two backgammon Computer Olympiads, in 2011 and 2015, with opponents some of the best backgammon-playing programs in the world, emerging victorious in both of them.

Additionally, we used the trained agents and self-play experiments to analyze key characteristics of these games for the first time, identifying one major flaw in the Fevga variant. The resulting statistics are then used to devise better strategies when playing in a match setting.

Finally, in order to facilitate later research efforts, we devised a framework called *bcd*Gammon for reducing/extending the complexity of backgammon games, preserving the key characteristics of the originals.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

CHAPTER 1: INTRODUCTION

Games have proven to be an ideal domain for the study of artificial intelligence, as not only are they fun to play and interesting to observe, but they also provide competitive and dynamic environments that model many real-world problems. Additionally, having increased their popularity in recent years, games are now a major part of the entertainment and software industry and an important cultural phenomenon. Methods from artificial intelligence promise to have a big impact on game technology and development, assisting designers and developers and enabling new types of computer games.

1.1 Artificial Intelligence in Games

Since the beginnings of Artificial Intelligence as a subfield of computer science, games have played an important role as a testing environment for the various algorithms, providing a much harder challenge than it is typically used in computer science research, the so called "toy problems". Moreover, due to familiarity of games to the general public, strong game-playing programs generated publicity, especially when the derived systems won matches against the best human opposition.

Over the decades of game AI research, there are many examples of high-performance game-playing systems. The first successful example was the checkers program Chinook, that managed to win the world's champion checker player in 1994 (Schaeffer, 1997). TD-Gammon, a program that played backgammon was the next to reach world championship level. Even it did not win a human champion in its several matches against human experts, later analysis showed that it played better than its human opponents (Tesauro, 2002). In 1997 Logistello, a program that played Othello, defeated Takeshi Mukarami, the human world Othello champion (Buron, 1998). In 1998, Maven, a scrabble playing program defeated scrabble grandmaster Adam Logan 9 – 5 (Sheppard, 2002).

An important milestone in game AI research was IBM's Deep Blue (Campell, Hoane Jr, & Hsu, 2002), the chess machine that defeated then-reigning World Chess Champion Garry Kasparov in a six-game match in 1997. In the years following this important match, human grandmasters had some success by managing to draw several matches in

(KramniK-Deep Fritz 4-4 (2002), Kasparov-Deep Junior 3-3 (2003), Kasparov-X3D Fritz 2-2 (2003)). However, after 2005 and the loss of grandmaster Michael Dams to Hydra 5.5-0.5, and the rematch Kramnik-Deep Fritz 2-4, the superiority of the machine was obvious. Nowadays, matches between humans and computers are still played – just not on equal terms anymore. Computer programs play with odds, by giving the human player some kind of advantage.

Recently, as computer resources grow, there have been several efforts that succeeded in solving complex games. Most notable achievements were the solving of Checkers (Schaeffer, et al., 2007) and Cepheus (Bowling, et al., 2015), a program that plays a variant of poker called heads-up limit Texas hold'em essentially perfectly.

1.2 The Problem

While many games have been studied extensively by Computer Science and Artificial Intelligence scientists, many more exist that are still unexplored. In this thesis, a family of still unexplored games will be examined, the variations of backgammon that are popular in Greece, Portes, Plakoto, Fevga, collectively called Tavli. Since Portes is similar to standard backgammon, a game that expert playing programs such as the aforementioned TD-Gammon exist, we will focus most of the research to other two games.

Concretely, the main research questions of this thesis are the following:

Question 1: Can strong game-playing agents be built that can play at expert level the backgammon variants popular in Greece (Tavli – Portes, Plakoto, Fevga)?

Question 2: Can the learning algorithms and training setups be improved in order to enable AI agents to learn to play backgammon games effectively by self-play?

After successfully building an expert AI agent we will then try to answer the following secondary research question:

Question 3: Can the expert agents be used to extract useful characteristics of the games?

1.3 Contribution

During the research for this thesis, the following contributions have been made:

A. Publications and Poster Presentations

Papahristou, N., & Refanidis, I. (2011). Training Neural Networks to Play Backgammon Variants Using Reinforcement Learning, *Proceedings of Evogames 2011, EvoApplications 2011*, Part I, LNCS 6624, (pp. 113-122). Springer.

Papahristou, N., & Refanidis, I. (2012a) Improving Temporal Difference Performance in Backgammon Variants, *13th Advances in Computer Games Conference* (ACG 2011), Tilburg, The Netherlands, November 20-22, 2011, LNCS 7168, (pp 134-145). Springer.

Papahristou, N., & Refanidis, I. (2012b) On the Design and Training of Bots to Play Backgammon Variants, *8th IFIP WG 12.5 Artificial Intelligence Application and Innovations Conference* (AIAI 2012), Halkidiki, Greece, September 27-30, 2012, Proceedings, Part I, Volume 381/2012, (pp 78-87). Springer.

Papahristou, N., & Refanidis, I. (2013) AnyGammon: Playing Backgammon Variants Using Any Board Size, Entry at the Research and Experimental Festival of the *8th International Conference on the Foundations of Digital Games* (FDG 2013), Chania, Crete, May 2013, (pp. 410-412).

Papahristou, N., & Refanidis, I. (2014) Opening Statistics and Match Play for Backgammon Games, *8th Hellenic Conference on Artificial Intelligence* (SETN 2014), Ioannina, Greece, LNCS 8445, (pp 569-582). Springer.

Papahristou, N., & Refanidis, I. (2015) Constructing Pin Endgame Databases for the Backgammon Variant Plakoto, *14th Advances in Computer Games Conference* (ACG 2015), Leiden, The Netherlands, to be published by Springer.

B. Software

- Palamedes (Chapter 6)

  (http://ai.uom.gr/nikpapa/Palamedes/)
- AnyGammon (Section 2.1.6.3)

  (http://ai.uom.gr/nikpapa/AnyGammon/)

C. Competitions

- 1st place and gold medal in the 16th Backgammon Computer Olympiad at Tilburg, The Nederlands, organized by the International Computer Games Association (ICGA) from 18 November to 26 November 2011 (International Computer Games Association, 2011).

- 1st place and gold medal in the 18th Backgammon Computer Olympiad at Leiden, The Nederlands, organized by the International Computer Games Association (ICGA) from 29 June to 6 July 2015 (International Computer Games Association, 2015).

1.4 Outline

This thesis is outlined as follows:

Chapter 2 describes the necessary theoretical background. Section 2.1 describes the rules of all the backgammon variants discussed in this thesis. We also present a framework, called *bcd*Gammon, which generalizes the backgammon games. Section 2.2 gives an overview of the reinforcement learning field. Reinforcement learning is a huge field, so a subset of its algorithms, the ones most related to this work, is presented.

Chapter 3 describes the self-play training procedure we used to train Neural Networks to play tavli games at expert level. We describe all our attempts to create intelligent agents in detail, compare different training setups and the rational about which features are important to include for successful learning.

In Chapter 4, the expert-playing agents constructed in the previous chapter are used in Monte-Carlo simulations, to extract useful statistics about the games. The distribution of the outcomes, the gammon rate and the first player advantage are some interesting characteristics of the games that are measured. These statistics are then used to enhance the match strategy of our agents.

Chapter 5 shows the construction of pin endgame databases for the game of plakoto. These databases generalize a huge amount of states to only a few million records. When they are utilized in our program, it is shown that the play of the plakoto agent is improved.

In Chapter 6 we describe the program that includes all the research done in this thesis, Palamedes. Palamedes is a program that provides an attractive user interface for everyone to play against the AI in the tavli and other variants. In this chapter, the two participations of Palamedes in backgammon competitions leading in two gold medals are also discussed.

Finally, Chapter 7 concludes the thesis and avenues of future work are presented.

Chapter 2

CHAPTER 2: BACKGROUND

2.1 Rules of backgammon variants

This section presents the rules of all the backgammon variants encountered in this thesis. The main target of this thesis are the variants that are popular in Greece, Portes, Plakoto, and Fevga, collectively called *Tavli*. Several other variants are also mentioned, when we attempt to explain some of our findings. All backgammon games are played on a board consisting of 24 triangles also called *points* divided in 4 quadrants of 6 points each.

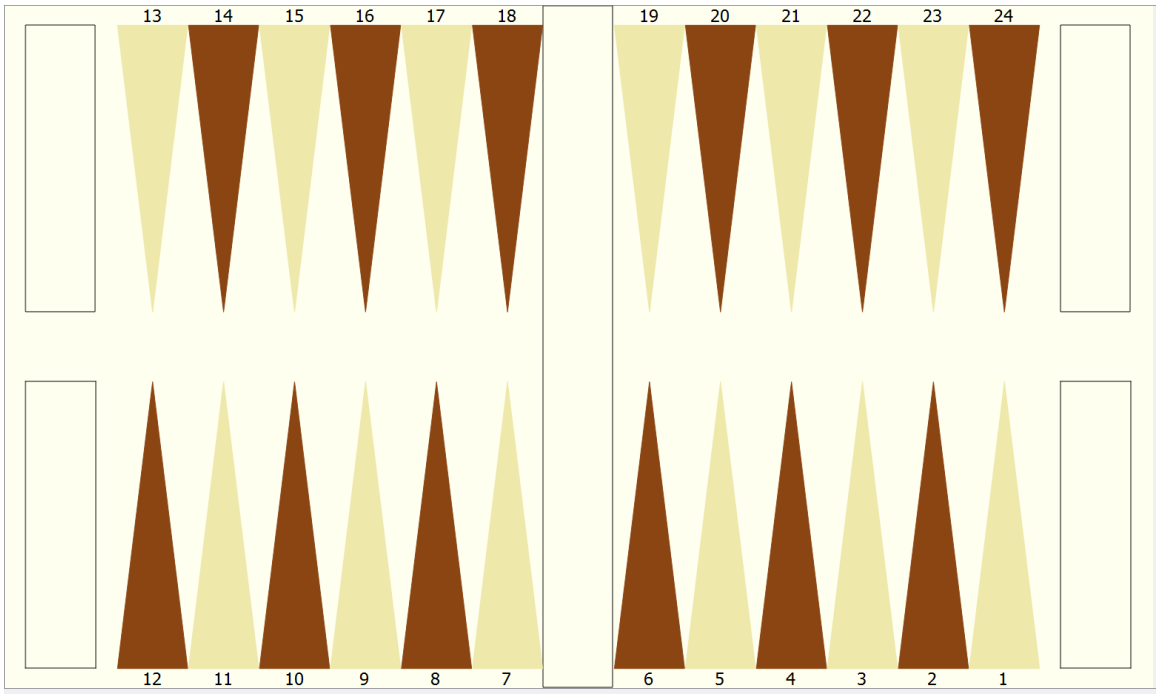


Figure 2.1: Backgammon board

The following rules are common in every variant examined in this thesis:

Each player starts the game with a number of pieces at his disposal, commonly called *checkers* (usually 15), placed in fixed starting positions.

The players take turns playing their checkers using an element of chance in the form of two six-sided dice according to the game rules. After rolling the dice, players must, if possible, move their checkers according to the number shown on each die. For example, if

the player rolls a 5 and a 4 (notated as "5-4" or “54”), the player must move one checker five points forward and another or the same checker four points forward. The same checker may be moved twice, as long as the two moves can be made separately and legally: five and then four or four and then five. If a player rolls two of the same number, called doubles, that player must play each die twice. For example, a roll of 3-3 allows the player to make up to four moves of three spaces each. On any roll, a player must move according to the numbers on both dice, if it is at all possible to do so. If one or both numbers do not allow a legal move, the player forfeits that portion of the roll and his/her turn ends. If moves can be made according to either one die or the other, but not both, the higher number must be used. If one die is unable to be moved but such a move is made possible by the moving of the other die, that move is compulsory.

When all the checkers of a player are inside his last quadrant of the board (called the *home board*), he can start removing them; this is called *bearing off*. The player that removes all his checkers first is the winner of the game.

If one player has not borne off any checkers by the time that player's opponent has borne off all fifteen, then the player has lost a double game (or *gammon* in standard backgammon terminology), which counts for double a normal loss or two points. Otherwise, the win is called “*single*” and is worth one point.

2.1.1 STANDARD BACKGAMMON

Standard backgammon is the most commonly used backgammon game in the western world. Widely known as just “backgammon”, we use the naming “standard backgammon” in this thesis in order to distinguish the game more easily from other variants which we may generally call “backgammon games” or “backgammon variants”. The following description of the rules is taken by the Wikipedia entry on backgammon (2015).

2.1.1.1 Setup and direction of movement

Each player begins with fifteen checkers, two are placed on their 24-point, three on their 8-point and five each on their 13-point and their 6-point. The two players move their checkers in opposing directions, from the 24-point towards the 1-point. Points 1 through 6

are called the home board or inner board and points 7 through 12 are called the outer board. The 7-point is referred to as the bar point and the 13-point as the midpoint.

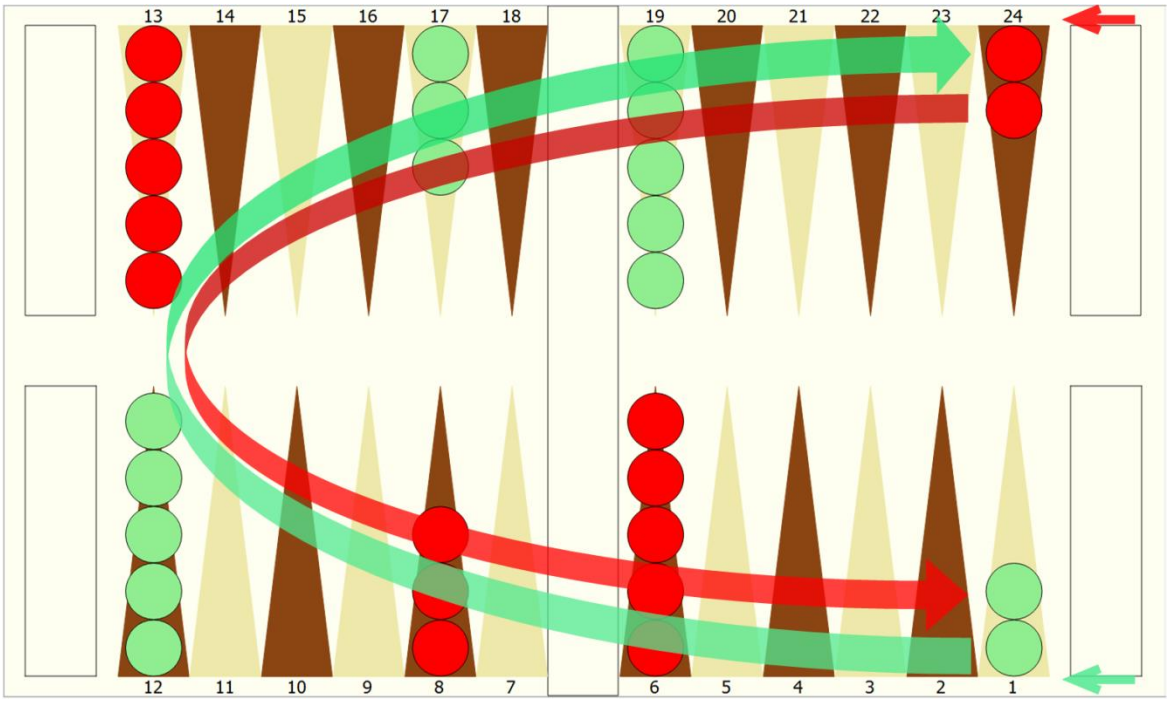


Figure 2.2: Starting position of standard backgammon and Portes

2.1.1.2 Movement rules

To start the game, each player rolls one die and the player with the higher number moves first using the numbers shown on both dice. If the players roll the same number, they must roll again. The players then alternate turns, rolling two dice at the beginning of each turn.

In the course of a move, a checker may land on any point that is unoccupied or is occupied by one or more of the player's own checkers. It may also land on a point occupied by exactly one opposing checker or "*blot*". In this case, the blot has been "*hit*" and is placed in the middle of the board on the *bar* that divides the two sides of the playing surface. A checker may never land on a point occupied by two or more opposing checkers.

Checkers placed on the bar must re-enter the game through the opponent's home board, before any other move can be made. A roll of 1 allows the checker to enter on the

24-point (opponent's 1), a roll of 2 on the 23-point (opponent's 2), and so forth, up to a roll of 6 allowing entry on the 19-point (opponent's 6). As in regular movement, checkers may not enter on a point occupied by two or more opposing checkers. More than one checker can be on the bar at a time. A player may not move any other checkers, until all checkers on the bar belonging to that player have re-entered the board. If a player has checkers on the bar, but rolls a combination that does not allow any of those checkers to re-enter, the player does not move.

When all of a player's checkers are in that player's home board, that player may start removing them; this is called "bearing off". A roll of 1 may be used to bear off a checker from the 1-point, a 2 from the 2-point and so on. A die may not be used to bear off checkers from a lower-numbered point, unless there are no checkers on any higher points.

If the losing player has not borne off any checkers and still has checkers on the bar or in the opponent's home board, then the player has lost a triple game (or *backgammon* in standard backgammon terminology), which counts for three times a normal loss or three points.

2.1.1.3 The doubling cube

To speed up match play and to provide an added dimension for strategy, a doubling cube is usually used. The doubling cube is not a die to be rolled but rather a marker with the numbers 2, 4, 8, 16, 32 and 64 inscribed on its sides to denote the current stake. At the start of each game, the doubling cube is placed on the bar showing number 64; the cube is then said to be "centered, on 1". When the cube is centered, the player about to roll may propose that the game be played for twice the current stakes. Their opponent must either accept ("take") the doubled stakes or resign ("drop") the game immediately.

Whenever a player accepts doubled stakes, the cube is placed on their side of the board with the corresponding power of two facing upward, to indicate that the right to re-double belongs exclusively to the player who last accepted a double. If the opponent drops the doubled stakes, he loses the game at the current value of the doubling cube. Although

64 is the highest number depicted on the doubling cube, the stakes may rise to 128, 256 and so on, though in expert play rarely does the cube exceed 4.

The doubling cube is a rule that was added in the 1930's in New York City. In order to take accurate doubling decisions, players must calculate the probabilities of winning/losing the game accurately, adding another strategy layer to the checker play. While it is used almost all the time in standard backgammon, the doubling cube is not used in other backgammon variants (with the exception of hypergammon).

2.1.2 PORTES

Portes is the first backgammon variant played in a tavli match. The starting position is the same with standard backgammon (Figure 2.2) and most of the rules are the same with the following exceptions:

- The winner of the opening roll rerolls for his first turn. Thus, unlike standard backgammon, a double roll is possible on the first move.
- The winner scores one point for a normal win and two points for a double win. There is no triple wins.
- There is no doubling cube.

2.1.3 PLAKOTO

The key feature of game Plakoto is the ability to pin hostile checkers, in order to prevent opponent movement. The general rules of the game are the same as Portes apart from the procedure of hitting. Players start the game with fifteen checkers placed in opposing corners and move around the board in opposite directions, till they reach the home board which is located opposite from the starting area (Figure 2.3).

When a checker of a player is alone in a point, the opponent can move a checker of his own in this point thus pinning (or trapping) the opponent's checker. This point counts then as a made point as in standard backgammon, which means that the pinning player can move checkers in this point, while the pinned player cannot. The pinned checker is allowed to move normally only when all opponent pinning checkers have left the point (unpinning).

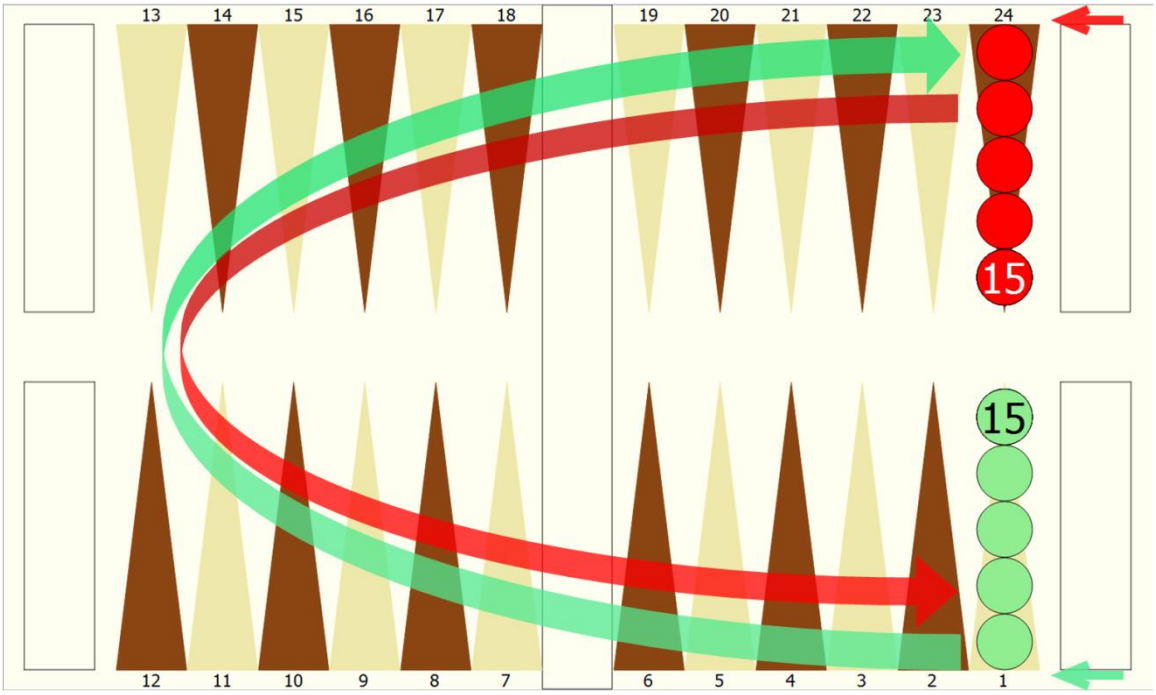


Figure 2.3: Starting position and direction of play in the Plakoto variant

Bearing off is done as usual with the following exception: bearing off is not permitted, if the opponent has one pin or more inside the player's home board. In other words, a player can be permitted to bearoff any of his checkers only when all his checkers inside his home board are pin-free.

The 24-point or the starting point is called the *mother point*. If a checker in this point gets pinned by the opponent, the game is over and you lose two points. The only exception is, if the opponent still has checkers on his starting point, since in this case his own mother is still threatened. A game in which both mothers are pinned is a tie.

2.1.3.1 Tapa subvariant

An interesting variation of Plakoto is the Tapa variant where the rules are exactly the same with Plakoto except for the initial checker placement: instead of placing all checkers in the last point the checkers are equally distributed in the last three points (5 checkers for each point (Figure 4). The change seems small, but has significant effect on important characteristics such as gammon rate and first player advantage (Chapter 5).

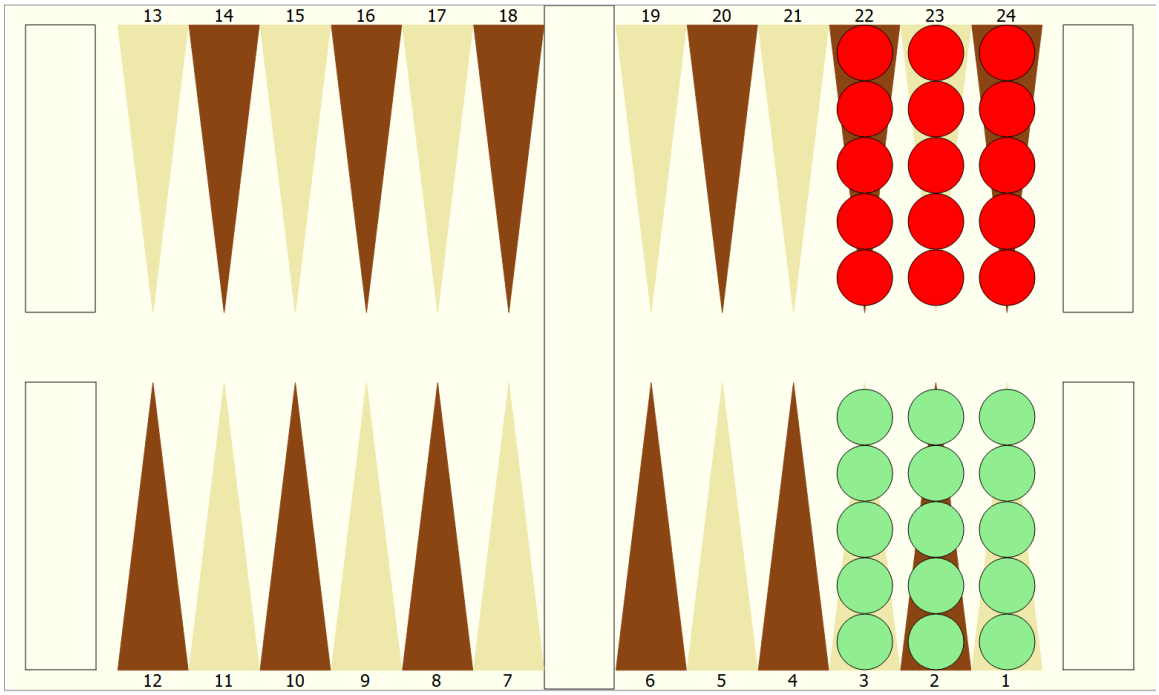


Figure 2.4: Starting position of the tapa variant

2.1.4 FEVGA

The main difference of Fevga from the other games is that there is no pinning or hitting. If the player has even a single checker in one point, this point counts as a made point, effectively preventing the movement of the opponent's checkers in this point. Each player starts with fifteen checkers on the rightmost point of the far side of the board, at diagonally opposite corners from each other, whereas the two players move to the same direction (Figure 5).

The game begins with a starting phase, where the players must move only one checker, until it passes the opponent's starting point, before they may move any other of their checkers. The formation of primes (six consecutive made points) is easier in this game, because a made point can be constructed using a single checker. The formation of primes has the following exceptions:

1. No player can form a prime in his starting quadrant.

2. No player can form a prime immediately in front of the opponent's starting point[1].

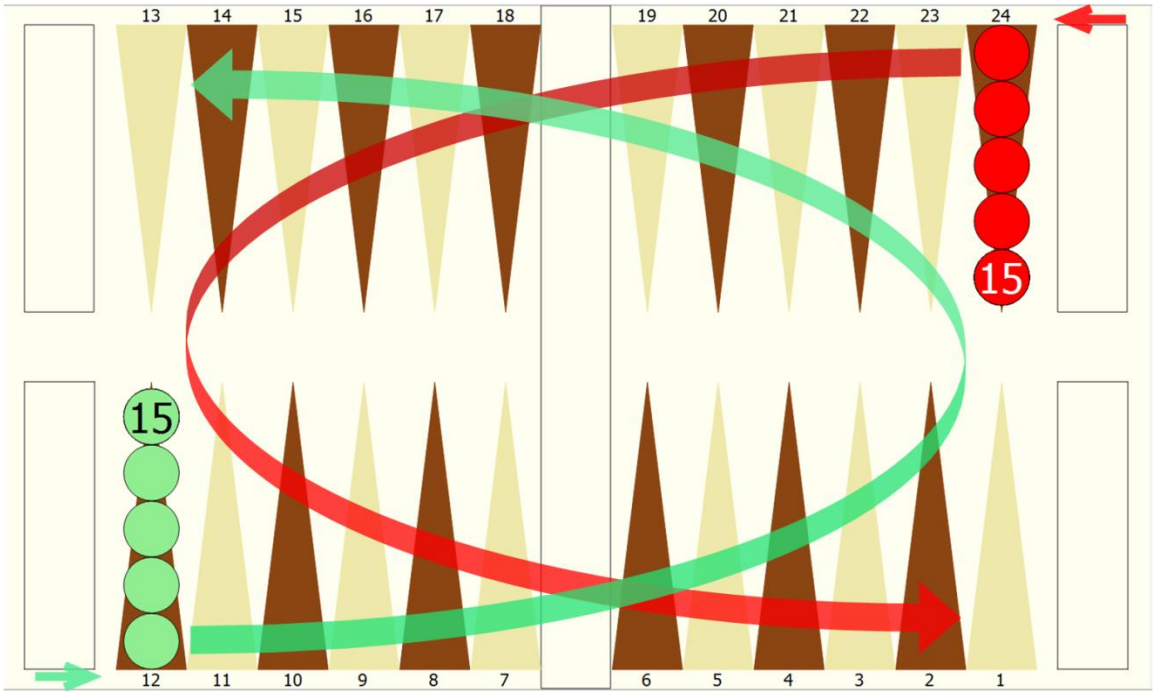


Figure 2.5: Starting position of the Fevga variant

Finally, it is not permitted to completely block the opponent (no-blocking rule). This means that for a move to be allowed, there must exist a dice roll that the opponent can use to move at least one checker[2].

As in Portes and Plakoto, triple games and the doubling cube do not exist.

[1] Rules 1 and 2 are not enforced in all Greek regions. Fevga in some regions is played with no prime exceptions, while in some others only the first exception is used. Because primes are powerful, we prefer to apply both exceptions in order to give more chances to the defending player to avoid getting primed. However, after analyzing all possible combinations, we concluded that enforcing 1 and/or 2 rule did not change the characteristics of the game in a statistically significant way.

[2] Sometimes the moving player that has completely blocked his opponent does not have a legal move to unblock the opponent. In this very rare situation, the player is allowed to play any legal move and play continues as usual (i.e. the opponent remains completely blocked).

2.1.5 MODES OF PLAY

2.1.5.1 Money Game

A money game is a style of competition where games are played individually and the participants bet on the result. At the end of each game the loser pays the winner the agreed initial stake multiplied by 2, if the result was a double win, or 3, if the result was a triple win. If the doubling cube is used, then the stake is further multiplied by the value of the doubling cube at the end of the game. From a game-theoretic point of view, a money game is like playing a match with infinite length. Unless otherwise stated, the experiments performed in this thesis are done under a money game mode; that is the agents try to maximize the result of the current game.

2.1.5.2 Match Play

This is the most common mode of competition used in tournaments on the internet and in casual play. The opponents play a series of games, until one of them reaches a predetermined number of points. Points (1, 2 or 3) are awarded normally after the end of each individual game (multiplied by the doubling cube in standard backgammon). The terminology used is "n-point match", which means that a player wins, when he acquires n points. The most popular match types used are the 7-point matches and 5-point matches because of the relative small amount of time needed to finish. In important competitions like championships, where the effect of luck needs to be reduced, longer matches are usually played (15-point or 19-point).

2.1.5.3 Tavli

In Greece, the most popular way of playing backgammon games is a Tavli match (Figure 2.6), where Portes, Plakoto and Fevga are played one after the other, until a player

reaches a predefined number of points. Using the rules described in match play, one could also play a Plakoto match or a Fevga match, but this is a very rare proposition in Greece.

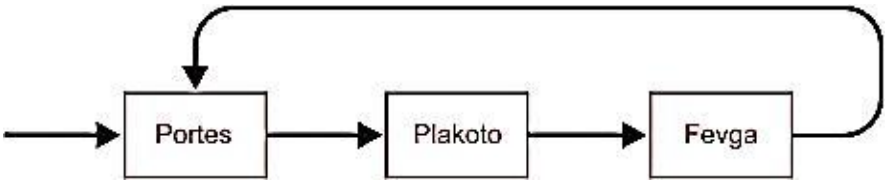


Figure 2.6 Flow of a typical Tavli match.

2.1.5.4 First player resolution

As it was shown earlier, in standard backgammon the first player to move is determined by the result of single die roll. The result of this roll is also used for the first dice roll of the game. This rule is applied before the start of each money game or, when playing a standard backgammon match, before the start of each individual match game. In a Tavli match, however, there are the following changes:

a) In the first game, the player that won the initial die roll rerolls again to begin his first turn.

b) After the first game, the winner of the previous game goes first.

These two match rules essentially permit doubles as the first roll on any tavli game, something that is not happening in standard backgammon matches. Disallowing a doubles as a starting roll, as we will see in chapter 5, has a positive effect in reducing the advantage of the first player.

2.1.6 Extending backgammon to arbitrary board sizes

Like other popular board games such as chess and go, backgammon has been studied with great interest by computer scientists. While game AI for standard backgammon has reached world-class strength (Tesauro, 2002), as it will be shown in chapter 3, the same claim cannot be stated for other variants i.e. Narde, Plakoto, Fevga, Acey-Deucey.

A popular method for developing, troubleshooting and understanding any game evaluation function in board games is to try it out first in smaller board sizes. For example in Go, apart from the standard 19x19 board, the game can be played at any board size, with 9x9 and 13x13 being the most popular ones. Standard practice for AI Go programmers is to start developing their algorithms in smaller board sizes like 9x9 and upscale from there. Furthermore, small board sized games (like 5x5 Go) can be solved more easily, giving an additional evaluation tool for the developers (van der Werf, & Winands, 2009).

In this section we attempt to reduce/extend the complexity of backgammon games in a consistent way. Previously, the only other attempt to simplify the backgammon games is the hypergammon variant (Keith, n.d.) that uses the same board size as standard backgammon but only 3 checkers for each opponent. The resulting game is simple enough in order to be strongly solved (Fang, Glenn, & Kruskal, 2008), but does not offer the strategic elements found in the original. In contrast, our underlying framework not only captures the key elements of the games in reduced versions, but also can easily extend the game into virtually any board size.

Another extendible game worth mentioning is Nannon (Pollack, 2005). This game is played on a backgammon board and can be extended on the number of checkers and the number of points on the board. However, the rules for moving the checkers are much different from the typical backgammon games (e.g. the player cannot stack checkers on a point) making the strategies required completely different. Another drawback of Nannon is that it uses a single six-sided die in all configurations, thus prohibiting the study of the effects of different chance events.

2.1.6.1 The *bcd*GAMMON Framework

In this section we present *bcd*Gammon, a framework for full parameterization of all key characteristics of a backgammon game. The name of the framework is inspired by its three core parameters, *b*, *c*, *d*:

b: is the total number of points on the board,

c: the number of available checkers for each player, and

d: the maximum number available when rolling a die.

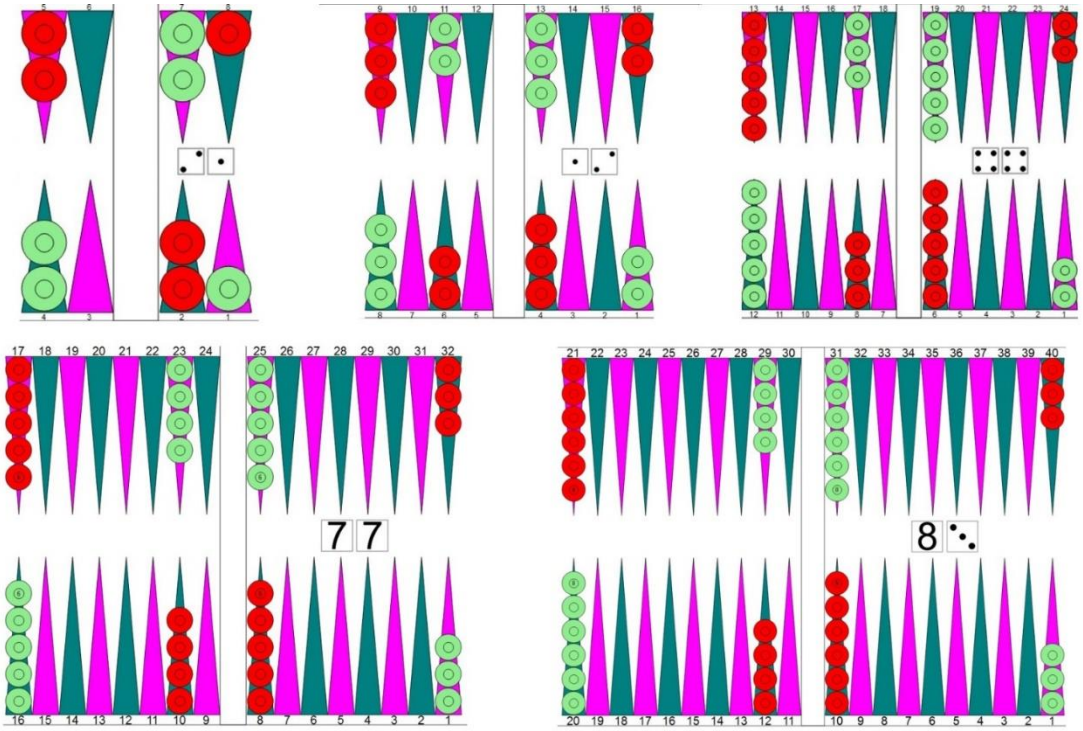


Figure 2.7: Screenshots of AnyGammon showing playable backgammon configurations. Upper Left: b=8, c=5, d=2, Upper middle: b=16, c=10, d=4, Upper right: b=24, c=15, d=6 (Standard backgammon), Lower left: b=32, c=19, d=8, Lower right: b=40, c=23, d=10

For example, parameterizing backgammon with *b*=16, *c*=10, *d*=4, named in short Backgammon(*b*=16, *c*=10, *d*=4) or Backgammon(16, 10, 4), results in a board with 16 points (4 for each quadrant), 10 checkers for each player and 4-sided dice (Figure 2.7, upper middle).

Theoretically, any number can be assigned to the three parameters, as far as b>3, c>0, d>1. In practice, and in order to preserve the look-and-feel of the original games, additional constraints should be added: b mod 4 = 0 and d = b/4. The former constraint is necessary in order to retain the look of the board as four quadrants; otherwise, the board must be represented in a straight line and additional rules regarding the home board must be added. The latter constraint is needed in order to preserve the strategic elements of the original games. We are not certain what would happen, if d is different from b/4, so we leave this investigation for future work. In all configurations, two dice are used as in the

original games. Most backgammon games examined in this thesis (standard backgammon, Portes, Plakoto, Fevga etc) can be considered a subset of the *bcd*Gammon framework where b=24, c=15, d=6.

Another crucial element of backgammon variants is the initial position. For some variants like Fevga and Plakoto, all checkers are placed in the starting point, so there is no problem in adapting any version of *bcd*Gammon. Standard backgammon, however, has a specific placement of the checkers at the start of the game. In all configurations supported, we adjusted the initial position to resemble standard backgammon.

2.1.6.3 AnyGammon: A program that supports the *bcd*Gammon framework

In order to promote and support the *bcd*Gammon framework, we authored AnyGammon (Figure 2.8), a game where players can play backgammon variants against the computer. Currently, supported game types are the tavli games (Portes, Plakoto and Fevga) but the goal of this project is to support dozens of backgammon variants. The program is available for free and can be downloaded from http://ai.uom.gr/nikpapa/AnyGammon. Currently the program runs on the Windows operating system and on Android devices (https://play.google.com/store/apps/details?id=gr.uom.ai.nikpapa.anygammon). AnyGammon was showcased at the FDG-2013 Research and Experiment Festival.

Players start a game in AnyGammon by selecting the key parameters of the game: game type, *b*, *c*, *d*. Currently, supported game types are Portes, Plakoto and Fevga variants. The goal of this project is to support dozens of backgammon variants. Notable variants planned for the immediate future are Narde, a variant similar to Fevga that is popular in Russia, and Acey-Decuey, a variant popular within the US military personnel. All games are played without the doubling cube; we plan to support this in future updates.

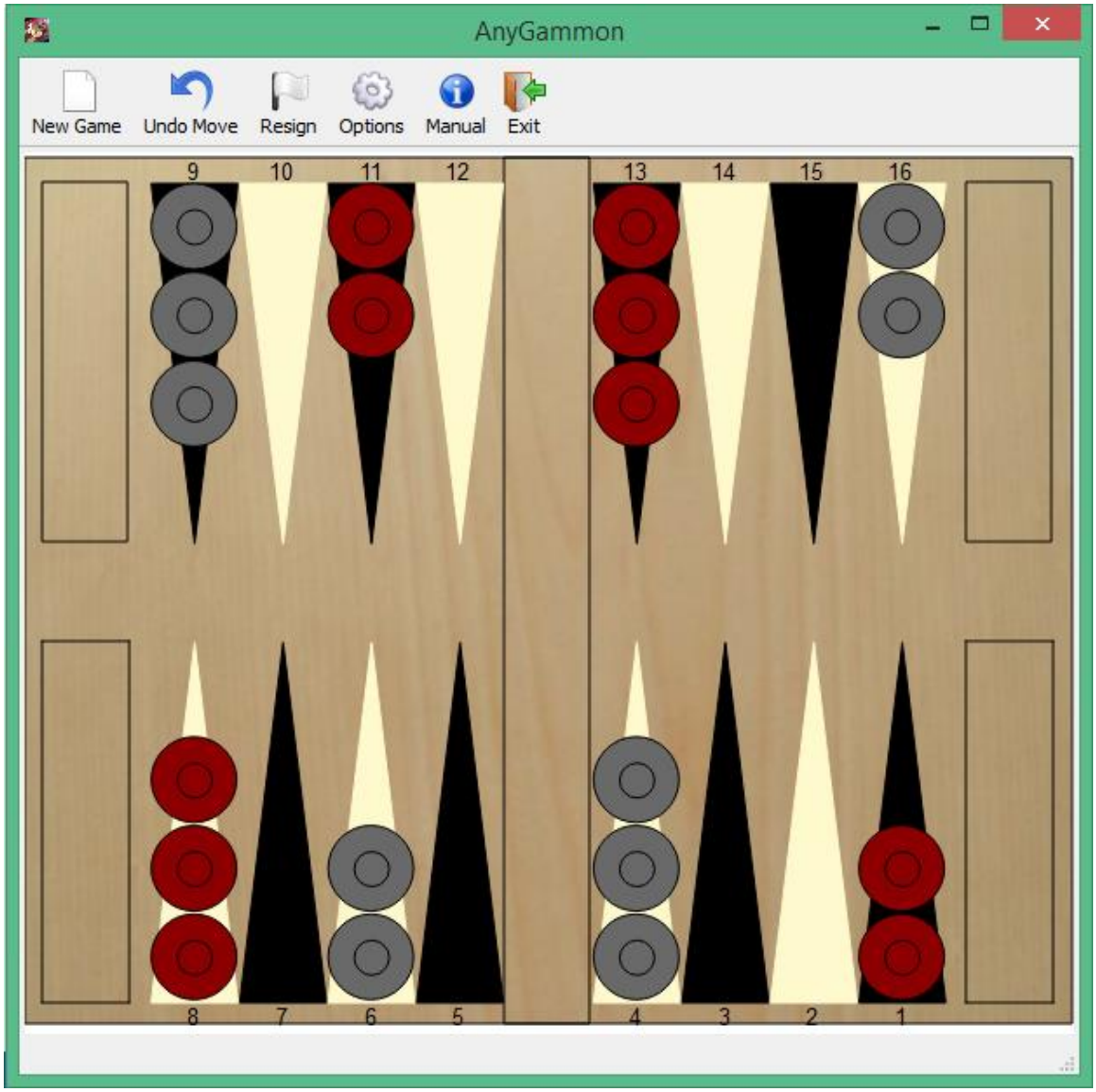


Figure 2.8: Screenshot of AnyGammon windows version

We placed several restrictions to the *b* and *d* parameters: board size (*b*) is restricted to a maximum of 40 points in increments of four and maximum number on the dice (*d*) is always *b* / 4. The number of checkers is also limited to a maximum of 30. Under these restrictions the player can select between 3x9x30=810 possible configurations at the start of a game. We plan to lift these restrictions, once we have fully investigated all aspects of these parameters.

2.1.6.4 AI agents used in AnyGammon

The AI of AnyGammon is currently in its initial stages. The user can select between two simple Monte Carlo methods, FlatMC and FlatUCB (Browne, et al., 2012), as well as the thinking time in seconds per move. Monte Carlo methods were chosen because of their simplicity and the easy application to all available configurations without parameterization. These methods can be further enhanced by inserting heuristic rules to the simulation specific to each variant. We are also planning to compare many algorithms such as the NN which is very popular in backgammon software, the NN methods we propose in chapter 4, and MCTS/UCT (Coulom, 2006), currently the dominant method in Go computer programs (Browne, et al., 2012). Finally, we want to investigate methods for transferring game evaluation functions learned for small boards to large boards and vice-versa.

2.1.6.5 Contribution and future work

The *bcd*Gammon was created primarily for research purposes as a testbed for game AI. Small board sizes make it easier to analyze algorithms and game evaluation functions. Large board sizes make the original games more challenging and interesting for the players. Especially for the smaller configurations, we believe that *bcd*Gammon will be of great importance in solving the full game for the following reason: Because backgammon games are not deterministic, every attempt to solve them requires to determine the minimum amount of precision that will be needed to store the floating point values in a database. Minimum precision is critical to reduce the memory requirements of the resulting database. This approach was used by (Bowling, et al., 2015) in the solving of the card game Heads-Up Limit Hold'em (HULFE), when another smaller card game was used (Rhode island poker) first to determine the optimal amount of precision both for reducing disk space and for best performance of the solving algorithm. A similar approach can be used for solving the game of standard backgammon: first one can try a solving algorithm in smaller *bcd*Gammon setups (e.g. b=16 or b=20), and then use the tuned precision parameter to solve the full game.

2.2 Reinforcement Learning

Reinforcement Learning (RL) is an area of Machine Learning (ML) itself, a sub-field of Artificial Intelligence (AI) and Computer Science (CS). Initially, inspired by behaviorist psychology, RL essentially gives a computational system where software agents learn behaviors in an environment under some notion of cumulative reward. This reward can be anything the agent deems valuable. Examples of reward in computer games reward are the eating of dots in pacman, the killing of an enemy in first person shooter games or the winning of a game in board games such as chess or backgammon.

The ultimate goal of RL research is to find ways to program "smart" agents without having necessarily knowledge of the environment, by giving them only rewards and punishments, and by making them react efficiently to environmental changes. In other words, learning in RL should be done using trial and error, having constant interaction with the environment. In recent years, RL has been given a lot of focus, because of the large number of practical applications that it can be used to address.

This chapter gives a brief overview of RL, focusing mainly on its main algorithms, the temporal difference type of algorithms, because these algorithms are used to construct backgammon evaluation function in later chapters. The presentation of these algorithms follows mainly (Sutton & Barto, 1998) and (Szepesvari, 2010), which we consider the main background references for RL. Obviously, an attempt of presenting the key concepts will be made having game applications in mind, and in particular, backgammon.

2.2.1 CORE ELEMENTS

The core elements found in RL problems are:

The *agent* is the learner and the decision-maker.

The *environment* is everything else outside the agent. An agent interacts continually with the environment by executing *actions* and the environment responds by presenting new information to the agent.

The *policy* that defines how the agent acts at any given moment. Policy is just a mapping of all the states of the environment to actions that can be executed in those states. In simple cases, a policy can be expressed with a lookup table; for example the game tic-tac-toe has 765 possible different positions, so a policy can be easily created by means of a table. However, real-world problems have huge amount of states that cannot be mapped in a table. In games, chess, backgammon and go are some examples with huge state spaces that are impractical to use tables.

The *reward function* is a critical element, because it defines the goal of the agent. It maps every state of the environment to a number, the *reward*, that determines how desirable is to the agent to be in this position. The goal of the agent is to maximize its long-term reward. In board games such as backgammon, the only reward that the agent gets is in the end of the game (terminal positions); for backgammon games this reward is usually +1 for a single win, +2 for a double win, -1 for a single loss and -2 for a double loss (and for standard backgammon +3, -3 for triple wins and losses respectively).

The *value function* shows the value of a state (or state-action pair) with respect to the cumulative perceived future rewards from the current state onwards. Contrast to the reward function that shows the imminent gain or loss, the value function shows if the state is good or bad long-term. A state can yield low reward, but still can have a high value, because there will be a following sequence of states that will give high reward. In game AI the value function is usually called *game evaluation function*, but has essentially the same definition: it is a value that determines how good or bad a state is with respect to the final outcome. In backgammon games another term for the value function is *equity* and is a number in the [-2, 2] interval ([-3, 3] in standard backgammon).

Finally, the environment *model* is a function that takes the current state and an action as inputs and returns the next state (also called *afterstate*) and the reward. Most RL algorithms target problems where the model of the environment is unknown, but they can also be used in environments where the model is usually known such as computer games.

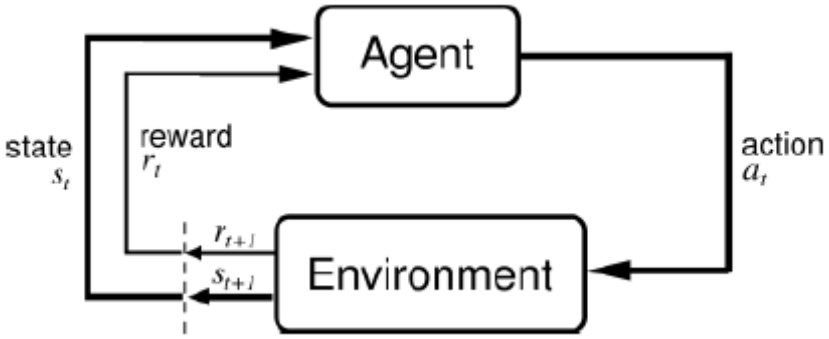


Figure 2.9: The Reinforcement Learning Framework (Sutton & Barto, 1998, pp. 71)

2.2.2 MARKOV DECISION PROCESS

RL problems are usually modelled as *Markov Decision Processes* (MDP) (Puterman, 1994). An MDP can be described as a quadruple $\{S, A, T, R\}$. S is a set of environment states, (S_1, S_2, … S_T), A is a set of actions available to the agent, with the subset of actions applicable to state s denoted as $A(s)$. T is a transition function, $P(s, a, s')$, which gives the probability of moving from state s to some other state s', provided that action a was chosen in state s. Transitions can be deterministic or non-deterministic. Finally, R is a reward function R: S ↦ ℝ, maps every possible state of the environment to a real number.

At each time t, the agent being in $s_t \in S$ chooses an action $a_t \in A(s_t)$, where $A(s_t)$ is the set of available actions in s_t, perceives the new state s_{t+1} and receives the reward $r_t = R(s_t, a, s_{t+1})$. Based on these interactions the goal of the agent is to choose a behavior that maximizes the expected return. The expected *return* for MDPs can be defined as $R_t = \sum_{k=0}^{T} \gamma^k r_{t+k+1}$, , where $0 \leq \gamma \leq 1$ is the discount rate. There is the possibility that T=∞ (for continuing tasks) or γ=1 but not both. The discount is necessary for continuing tasks, because in an infinite time horizon the sum of rewards can be infinite. Tasks that terminate

are called *episodic* tasks. In episodic tasks the agent usually tries to maximize the mean of the sum of rewards for every episode.

A desired property of state representations is to retain all relevant information leading up to the current point of time, in order for the agent to be able to make a correct decision for the future. A state having this property is called *Markov* or having the *Markov property*. In the game of backgammon for example, we can have backgammon positions of the board as states. A backgammon position has the Markov property, because we do not need any other information in order to determine the best course of action successfully. The moves that led to the current position need not be retained so are irrelevant.

2.2.3 VALUE FUNCTIONS

The success of an agent following a policy π depends on how much more reward can be accumulated in the long run. The optimal policy π^* is a policy that maximizes the expected return. A RL algorithm tries to optimize its policy so as to come as close as possible to the optimal policy π^*.

Instead of trying to directly find the optimal policy π^* a class of RL algorithms try to calculate first a *value function* and then extract the policy using this function. Value functions can be defined for states or for state-action pairs.

A value function V: S $\mapsto$ $\mathbb{R}$, maps a state to a real number value which expresses the expected return of the agent in a state, when it follows policy π from the current point of time onwards:

$$V^{\pi}(s) = E_{\pi}\{R_t | s_t = s\} = E_{\pi}\Big\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \;\Big|\; s_t = s \Big\}$$

Another way of using value functions is state-actions pairs, Q: $S \times A \mapsto \mathbb{R}$. These values are usually called Q-values and similarly to state values, express the expected return of an agent in a state s, that makes action a, and follows policy π afterwards:

$$Q^{\pi}(s,a) = E_{\pi}\{R_t | s_t = s, a_t = a\} = E_{\pi}\Big\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \;\Big|\; s_t = s, a_t = a \Big\}$$

$Q^{\pi}(s,a)$ is called *action-value* function for policy π.

In order to find an optimal policy π, the agent can learn either the optimal Q^* or the optimal V^*. A policy π is better than another policy π', if and only if $V^{\pi}(s) \geq V^{\pi'}(s)$ for every state. All optimal policies share the same optimal value function:

$$V^*(s) = \max_{\pi} V^{\pi}(s)$$

Similarly, the optimal policies have the same optimal action-value function:

$$Q^*(s,a) = \max_{\pi} Q^{\pi}(s,a)$$

2.2.4 Basic Reinforcement Learning Methods

We will examine the following methods for solving the reinforcement learning problem: Dynamic Programming (DP), Monte Carlo (MC) and Temporal Difference Learning (TDL). DP methods need a complete model of the environment and are computationally expensive, but are well understood and offer the mathematical basis for many RL algorithms. MC methods are simple and do not require a model, but do not work well for step-by-step computation. Finally, TDL methods also do not need a model, are fully incremental, but are more complex to analyze. A little more emphasis is given to MC and much more to TDL, because these are the methods used in this thesis.

2.2.4.1 Dynamic Programming

DP is a family of algorithms for finding the optimal policies in a MDP given the model of the environment. Value functions are essential here for guiding the agent towards a good policy. We first start by finding the value function of a policy π using the Bellman equation:

$$V^{\pi}(s) = \sum_{a} \pi(s,a) \sum_{s'} P^{a}_{ss'} \left[R^{a}_{ss'} + \gamma V^{\pi}(s') \right],$$

where $P^{a}_{ss'}$ is the probability of the agent being in state s',when it chooses action a in state s, and $R^{a}_{ss'}$ is the corresponding reward received.

The Bellman equation is a recursive equation that defines a |S| simultaneous linear equations in |S| unknowns. A simple iterative method to find the optimal policy involves

two steps: a) starting from some random value function (say V_0), we use the above equation iteratively to find V^{π}, b) then in the second step, we improve the policy (π') choosing the best action that maximizes the value function:

$$\pi'(s) = \arg\max_a \sum_{s'} P^a_{ss'}\left[R^a_{ss'} + \gamma V^{\pi'}(s')\right]$$

By alternating these two steps (policy evaluation and policy improvement) we are guaranteed to gradually improve the policy and eventually converge to the optimal policy. This algorithm is called *policy iteration*.

One drawback of policy iteration is that the policy evaluation step is a computational costly operation, because it needs many iterations in order to converge. If the policy evaluation step stops after only one sweep of the state space, then we have the *value iteration algorithm*:

$$V_{k+1}(s) = \max_a \sum_{s'} P^a_{ss'}\left[R^a_{ss'} + \gamma V_k(s')\right]$$

More information about DP methods is available in (Bertsekas, 1995) and (Bertsekas and Tsitsiklis, 1996).

2.2.4.2 Monte Carlo methods

Contrary to DP methods, Monte Carlo methods do not need a full model of the environment; only a way of generating sample transitions and not the explicit model of transition probabilities. They utilize accumulated experience, i.e. sequences of states, actions and rewards. These sequences are frequently called trajectories or samples or simulations. MC methods are based on the fact that we could estimate a state value by averaging the returns from observed visits to that state. As more returns are observed (i.e. more samples experienced), the average should converge to the expected value.

For example, suppose we want to find $V^{\pi}(s)$, the value of state s under policy π. Under the MC method *first visit Monte Carlo*, the agent creates samples following policy π and for each time s is encountered, it records the return after the first encounter of s.

When the episode is finished, the returns are accumulated and the average of all the accumulated rewards asymptotically converges to $V^{\pi}(s)$. The same algorithm can be applied to state-action (Q) values.

2.2.4.2.1 Bandit-Based methods

The term Monte Carlo is also often used for estimation methods involving random components. An important class of problems are bandit problems, where the agent must choose between n actions in order to maximize the cumulative reward over some time period. The problem resembles slot machines, where a player would like to maximize his winnings having n slot machines at his disposal. If he knew which arm will give the best value, he will surely play only this one; without this knowledge, however, one must rely on previous observations. This leads to the exploration-exploitation dilemma: one needs to balance exploitation of the bandit currently believed to be optimal with the exploration of other bandits, which now appear suboptimal, but may turn out to be superior in the long run.

Bandit-based methods aim to minimize the agent's *regret*, in other words the expected loss due to not selecting the best bandit:

$$R \quad = \mu^* n - \mu_j \sum_{j=1}^{K} E\left[T_j(n)\right],$$

where μ^* is the best possible expected reward, n is the number of plays (simulations) so far, K is the total number of arms and $E\left[T_j(n)\right]$ is the expected selections of arm j in the first n trials. It is important to ensure that all bandits are sufficiently explored in order not to get stuck in a suboptimal arm that seems temporarily promising. In other words, it is important to place an upper confidence bound (UCB) on the rewards observed so far.

The simplest UCB policy was proposed by Auer, Cesa-Bianchi, N., & Fischer, (2012) and is called the UCB1 policy, in which the arm j selected is the one that maximizes

$$UCB1 = \bar{X}_j + \sqrt{\frac{2 \ln n}{n_j}}$$

where $\bar{X}_j$ is the average reward of move j, n is the number of times the parent of node j was traversed and n_j is the number of times that j was selected. When the reward distribution is in the [0, 1] interval, this bound is guaranteed to be optimal.

2.2.4.2.2 Monte Carlo Tree Search

Building on the above algorithms, instead of only updating the values of our possible actions, one can build a tree of underlying state-actions towards the end of the episode. This is especially important in games where the sequence of actions can be expressed as a game-tree. This class of algorithms are Monte Carlo Tree Search Algorithms (MCTS) and have shown to be effective in several games (Arneson, Hayward, & Henderson, 2010; Lorenz, 2010; Winands, Björnsson, & Saito, 2010), most notably go (Gelly, 2007; Coulom, 2007).

The basic idea of the algorithm is to progressively build a partial tree/graph in memory, guided by the results of previous simulations. For each stored node, MCTS stores statistics of the simulations traversing that node, including at least the minimum of which are the total rewards gained and the visit count, i.e. the number of simulations that traversed that node.

The following four steps are applied for each iteration of the algorithm iteratively:

1) *Selection:* Starting from the root node, a child is recursively selected according to the selection policy, until a node is found that must be expanded. Nodes must be expanded, if they are not terminal and have at least one child not in the tree.

2) *Expansion:* One or more nodes are added to the tree at the end of the selection point.

3) *Simulation:* A simulation (or playout) is run from the new node(s) according to a simulation policy, until a terminal state is reached, where we receive the total reward. The most trivial simulation policy is a uniformly random policy.

4) *Backpropagation:* The reward is "backed up" through the selected nodes of the tree, to update their statistics.

2.2.4.2.3 MC algorithms for games

In this thesis we utilize or refer to the following MC algorithms:

- *flat Monte Carlo*: a Monte Carlo method with uniform move selection and no tree growth.
- *flat UCB:* a Monte Carlo method with the bandit-based move selection *UCB1* but no tree growth.
- *MCTS*: a Monte Carlo method that builds a tree to inform its policy online.
- *UCT*: MCTS with any *UCB* tree selection policy.

The above terminology is borrowed from (Browne, et al., 2012).

2.2.4.3 Temporal Difference Methods

Temporal Difference Learning (TDL) is a branch of algorithms that combines characteristics from MC and DP methods. Like MC methods, TDL does not need the full model of the environment, just a way to extract experience (samples). Like DP methods, the updates to the value function estimations are based in part on estimations of later steps (a process also called *bootstrapping*) without waiting for the episode to end.

2.2.4.3.1 One step TDL methods

The most commonly used TDL methods are the ones that base their updates on the estimation of the next step only. We will examine three such simple methods, TD(0), Sarsa, and Q-learning.

In TD(0) the update rule is the following:

$$V(s_t) \leftarrow V(s_t) + a[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)],$$

where s_t is a state in time *t* and $0 < a \leq 1$ is a parameter called the *learning rate*. From the above equation it can be seen that TD(0) is using another estimate ($V(s_{t+1})$) to update the current value. Specifically, the value that we want the value function to shift to is $r_{t+1} + \gamma V(s_{t+1})$ term, also called the *target* of the update, and whole term $r_{t+1} + \gamma V(s_{t+1}) - V(s_t)$ is called the *TD-error*. Figure 2.10 presents the algorithm in pseudo-code form:

V(s) ←Arbitrate initialization to the policy π for all states
a ←Initialize to a fixed learning rate
γ ←Initialize to a fixed discount rate
For each episode **do**
    s ←Initial state
    Repeat (for every step of the episode)
        a ←action given by π for s
        Take action a
        s' ←next state after taking action a
        r ←reward observed after taking action a in state s
        $V(s) \leftarrow V(s) + a[r + \gamma V(s') - V(s)]$
        $s \leftarrow s'$
    Until s is terminal
End For

Figure 2.10: Tabular TD(0) for estimating V^{π}

The equivalent algorithm of TD(0) for state-action (Q) values is the *SARSA (State Action Reward State Action)* algorithm (Singh and Sutton, 1996). The update rule in SARSA becomes:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)],$$

where s_t is the state the agent is originally in time *t*, a_t is the action selected in that time step and s_{t+1} the state results in after the action a_t. The reward received is r_{t+1} and the next action taken under the current policy is a_{t+1}. The Q value function is updated after each time step, towards the target: $r_{t+1} + \gamma Q(s_{t+1}, a_{t+1})$ and the rate of learning is controlled by parameter a. Figure2.11 presents the algorithm in pseudocode:

$Q(s, a)$ ←Arbitrate initialization for all states-actions
a ←Initialize to a fixed learning rate
γ ←Initialize to a fixed discount rate
For each episode **do**
    s ←Initial state
    a ← selected action evaluating Q in s
    repeat
        Take action a
        s' ←next state after taking action a
        r ←reward observed after taking action a in state s
        a′←selected action evaluating Q in s'
        $Q(s, \mathrm{a}) \leftarrow Q(\mathrm{s}, \mathrm{a}) + \alpha[\mathrm{r} + \gamma Q(\mathrm{s}', \mathrm{a}') - Q(\mathrm{s}, \mathrm{a})]$
        $s \leftarrow s'$, a ←a′
    Until s is terminal
End For

Figure 2.11: Tabular Sarsa algorithm

Another popular TDL algorithm for state-action value is the *Q-Learning* algorithm (Watkins, 1989; Watkins & Dayan, 1992). The update rule in this algorithm is:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right]$$

The term $\max_a Q(s_{t+1}, a)$ denotes that the target state-action pair may not be the same with the state-action pair followed by the sampling policy. This is important because it allows an arbitrary selection of sampling strategies. These kind of algorithms, where the learning agent tries to find a policy that may be unrelated to the policy followed, are called *off-policy* methods. On the other hand, *on-policy* methods (like *TD(0)* and *SARSA*) learn the same policy they use to explore the environment. Provided that all state-action pairs are updated infinitely often, *Q-Learning* has been proven to converge to Q^*.

$Q(s, a)$ ←Arbitrate initialization for all states-actions
a ←Initialize to a fixed learning rate
γ ←Initialize to a fixed discount rate
For each episode **do**
s ←Initial state
repeat
a ← selected action using policy derived from Q (e.g. ε-greedy)
Take action a
s' ←next state after taking action a
r ←reward observed after taking action a in state s
$Q(s, \mathrm{a}) \leftarrow Q(\mathrm{s}, \mathrm{a}) + \alpha \left[\mathrm{r} + \gamma \max_{a'} Q(s', a') - Q(\mathrm{s}, \mathrm{a})\right]$
$s \leftarrow s'$
Until s is terminal
End For

Figure 2.12: Q-Learning algorithm

2.2.4.3.2 Multi-step TD methods

There is a way to bridge methods with one-step backups (TD(0)) and full backups (DP) by using a number of backups more than one but less than all of them until termination. For example, we could extend the TD(0) and instead of taking the target from the transition to the next state, we could use *n-step backups*, when the target of the update extends to the *n*th step under the following general case:

$$R_t^{(n)} = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots + \gamma^{n-1} r_{t+n} + \gamma^n V_t(s_{t+n})$$

Similarly, the updates can be made not only by using *n*-step returns, but also using the average of the following *n* returns. Every mix of returns can be considered as long as the weights of the returns are positive and have a sum of 1. This kind of return is called a *complex backup*. This kind of backups are used in one of the most popular TDL algorithm, TD(λ).

2.2.4.3.3 TD(λ) – forward view

The TD(λ) algorithm uses complex backups with weights proportionally to λ^{n-1} where $0 \leq \lambda \leq 1$. Adding a $(1 - \lambda)$ factor to normalize the weights to 1 we have the following return at each time step *t*:

$$R_t^{\lambda} = (1-\lambda) \sum_{n=1}^{T-t-1} \lambda^{n-1} R_t^{(n)} + \lambda^{T-t-1} R_t$$

The return of the first step has the larger weight (1 – λ), the return of the second step has the second best with (1-λ)λ, the third step reward is (1-λ)λ^2 and so forth. An important implementation point is that after a terminal state has been reached, all subsequent returns are equal to the terminal return. Another implementation point is that while theoretically we could go till the end of the episode to get to the average return, in practice we stop when the (1-λ)λ^n becomes a sufficiently small number.

In the extreme cases, when λ = 0, only the one-step return is used ($R_t^{(1)}$) making the algorithm equivalent to TD(0), while, when λ = 1, the first term is zero, so only the terminal reward (R_t) is backed-up, making the algorithm equivalent to MC methods.

This approach is called the forward view of TD(λ), because the agent at each time step must know in advance the subsequent returns in order to make the updates to the value function. In practice, this can be achieved by sampling the environment, keeping the policy "frozen" and, after the episode ends, then updating the value function.

2.2.4.3.4 TD(λ) – Backward view

The forward view has the drawback that the episode must end in order for the updates to begin to take place. In typical RL problems, it is desirable to update the value function incrementally, (i.e. at each time sampling step, also called online learning). Fortunately, there is a way to make *TD(λ)* an incremental algorithm by introducing another variable for every state *s*, called eligibility trace, denoted with $e_t(s)$. For the tabular case the algorithm is shown in Figure 2.13.

Contrary to the forward view, here we observe the current TD error and we update it backwards to every previous state proportionally to the state's current eligibility trace. Except for the accumulating traces presented in Figure 2.13, other types of eligibility traces has been proposed such as *replacing traces* and *resetting traces*, to name a few. We will not discuss further more all the different traces, since we are focused in this thesis mainly

in the forward view. It has been proved that in the online case, eligibility traces are a good, but not exactly equal, approximation to the theoretical forward view.

V(s) ←Arbitrate initialization to the policy π for all states
$e(s)$ ←0 for all states *s*
a ←Initialize to a fixed learning rate
γ ←Initialize to a fixed discount rate
For each episode **do**
    s ←Initial state
    Repeat (for every step of the episode)
        a ←action given by π for *s*
        Take action a
        s' ←next state after taking action a
        r ←reward observed after taking action a in state *s*
        $\delta \leftarrow r + \gamma V(s') - V(s)$
        $e(s) \leftarrow e(s) + 1$
        For all s **do**
            $V(s) \leftarrow V(s) + a\delta e(s)$
            $e(s) \leftarrow \gamma\lambda e(s)$
        End For
        $s \leftarrow s'$
    Until *s* is terminal
End For

Figure 2.13: Tabular *TD*(λ) with accumulating eligibility traces

2.2.4.4 Function approximation

Value functions in tasks with small state action spaces can be represented as tables where each position stores the value of a state or state-action pair. As the state (or action) space increases, however, the use of tables becomes difficult not only because of the increased memory needed, but also because of the time needed to update all the values. This is especially critical in games where the state space of most games of interest is huge. For example, standard backgammon has a state space in the vicinity of 10^{18}, a size which prohibits the use of tables, and makes some form of generalization of the value function mandatory. This kind of generalization is known as *function approximation*, since it tries to approximate a function - in our case the value function.

2.2.4.4.1 Approximating Value Functions

All the prediction methods examined so far have used incremental updates, where the value function of a state shifts towards the direction of a backed-up value also known as target value. Another way of viewing these backups is using supervised learning, where the backups translate as a training example (input-output pair) and then we interpret the derived approximation function as the prediction of the desired value function. This way we can use many supervised learning algorithms like neural networks, decision trees, linear regression, etc. In this thesis we will examine mostly neural networks, because this is the method that we used for approximating the value function in backgammon games.

In order to use function approximation to predict the value function V^{π}, we represent the value function V_t using a parameter vector $\overrightarrow{\theta_t}$. For example, we could use a neural network with $\overrightarrow{\theta_t}$ being the weights of the network. Usually the number of parameters are much smaller than the total number of states, so changing a parameter changes the value in many states. Consequently, when we apply a backup in some time step multiple states are affected.

2.2.4.4.2 Gradient-Descent Methods

An extremely popular class of methods is *gradient descent* methods, where the parameter vector is a column vector with a fixed number of real components, $\overrightarrow{\theta_t} = (\theta_t(1), \theta_t(2), \ldots, \theta_t(n))^T$ and $V_t(s)$ is a smooth differentiable function of $\overrightarrow{\theta_t}$ for every $s \in S$. These states are usually successive states derived through sampling of the environment.

A good strategy for updating $\overrightarrow{\theta_t}$ is to minimize to MSE error of the observer examples using gradient descent by adjusting the parameter vector towards the direction that reduces the error:

$$\vec{\theta}_{t+1} = \overrightarrow{\theta_t} + a[v_t - V_t(s_t)]\, \nabla_{\overrightarrow{\theta_t}} V_t(s_t),$$

where a is the usual learning rate constant, v_t is some prediction of the value function for the state s_t and $\nabla_{\overrightarrow{\theta_t}} f(\overrightarrow{\theta_t})$ is the vector of partial derivatives of our approximation

function. If v_t is an unbiased estimate for each t, then $\overrightarrow{\theta_t}$ is guaranteed to converge to a local optimum under the condition of sufficiently decreasing a.

Applying gradient descent to TD(λ) and its n-step returns averages we have to following update rule:

$$\vec{\theta}_{t+1} = \overrightarrow{\theta_t} + a\left[R_t^{\lambda} - V_t(s_t)\right] \nabla_{\overrightarrow{\theta_t}} V_t(s_t)$$

where R_t^{λ} is the usual λ-return term of the forward view TD(λ). Unfortunately, for $\lambda < 1$, R_t^{λ} is not a unbiased estimate of $V^{\pi}(s_t)$, so it is not guaranteed to converge to a local optimum. Nevertheless, such bootstrapping methods have been found to work very well in practice.

2.3 TDL in games and AI for backgammon

In this section we will examine the most important research related to the work done in this thesis. Due to the nature of the research performed in this thesis, which is mainly focused on building expert AIs on backgammon variants, the related work presented in this section is split into two areas: a) research related in TDL methods applied in games, our main focus on building successful backgammon agents and b) research related in building AI in backgammon other than TDL. Both have considerable depth, so, for space reasons, we limited our survey in research deemed as the most significant.

2.3.1 TDL METHODS IN GAMES

The earliest attempt that used TDL-like methods was Samuel's checker program (Samuel, 1959). Even though TDL methods would take around 25 years, until they were conceived, Samuel used a form of bootstrapping to update an evaluation function towards the value of a minimax search, after black and white had each played a move. We call this algorithm TD-Root following the name used in (Veness, et al., 2009). This approach enabled Samuel's program to achieve checker play equivalent to an amateur human. His efforts were limited by the computing resources of the time and by the fact that the evaluation function was directed to measure the material advantage and not the actual result of the game.

There is plenty of researchers that effectively trained game-playing agents to some degree. Being in line of our original intentions to produce expert-level playing agents, in the following sections we analyze related work that produced strong game-playing agents equivalent to human masters or above.

2.3.1.1 TD(λ) and TD-Gammon

The first major success of TDL, and probably of machine learning in general, was the TD-Gammon program (Tesauro, 1992; 1995; 2002). TD-Gammon used the TD(λ) algorithm and self-play to train from scratch a standard MLP-type neural network that learned the evaluation function of standard backgammon. TD-Gammon played several matches against backgammon world champions and, even though it was defeated in all of them, later rollout analysis showed that it made fewer errors than its human opponents (Tesauto, 2002). In this section we analyze in detail Tesauro's training setup, since it is the most relevant compared to our training enhancements presented in this thesis.

Td-Gammon's learning procedure works as follows: a sequence of states is created starting from the initial position and ending at the terminal position during the course of a self-play game (meaning that the neural network plays with itself in order to produce the game-moves). These states are represented as vectors to the inputs of the neural network (x_1, x_2, …, x_T) by means of a special encoding. Every step of the sequence represents a move from one side (1-ply in game terminology). For every vector x_t, a corresponding output vector Y_t exists that represents the estimated value of x_t. In the TD-Gammon system, the output signal is composed by four outputs, one for each possible outcome of the game ($[p_1, p_2, p_3, p_4] \rightarrow$ [win single, loss single, win double, loss double]).

With this approach the weights of the neural network are used as function approximation (Section 2.2.4.4) to the value function and the backpropagation procedure is used to compute the gradients in the gradient-descend form of TD(λ) according to the following equation for every output unit:

$$\Delta w_t = \alpha\,(Y_{t+1} - Y_t) \sum_{k=1}^{t} \lambda^{t-k} \nabla_w Y_k$$

where w is the vector of neural network weights being tuned, Y_t is the prediction for the output at time step t, $\nabla_w Y_k$ is a set of partial derivatives for each component of the weights w, α is the standard learning rate and λ is the factor controlling how much future estimates affect the current update. At the end of the game a terminal signal z, composed of 4 elements as described earlier, is returned based on the final game outcome. At the terminal position the above equation is applied as well, with one difference: the target is replaced by the terminal signal z instead of the next state Y_{t+1}. Under this condition, every output of the neural network can be thought as a probability of reaching the corresponding outcome given the input state. Combing the outputs together, an estimation of the value (also called equity in backgammon terminology) can be easily calculated.

During the learning process, the moves of the game selected by both players were determined by the learning agent with following procedure: on every time-step the agent grades all afterstate positions resulting after every possible move. The move actually played is then determined by selecting the one that leads to the afterstate with the most equity. When the training starts, the weights of the neural network were initialized uniformly in the [-0.5, 0.5] interval. Surprisingly, this setup was able to learn expert behavior from self-play, even though at the start of the training the moves selected are random.

A crucial factor for the success of the method was the encoding of the backgammon position to the input layer of the neural network. The final encoding was used after many experiments: for every point of the 24 points of the backgammon board, 4 inputs were used for every player. The first input is binary and is set when the player has exactly 1 checker on the point, the second is also binary and is set when there are 2 or more checkers of the player on the point, the third is binary as well and is set when there are 3 exactly three checkers and the fourth is a float and is proportional to the amount of checkers >= 4 the player is having on the point. Under the above reasoning, the total amount of inputs is 192. Another 6 inputs were added, 2 encoding the bar points, 2 for the beared-off checkers and 2 for the side to move. This encoding of the backgammon states at every time step without including inputs representing expert knowledge is called *raw encoding*. The neural network architecture is shown in Figure 2.14.

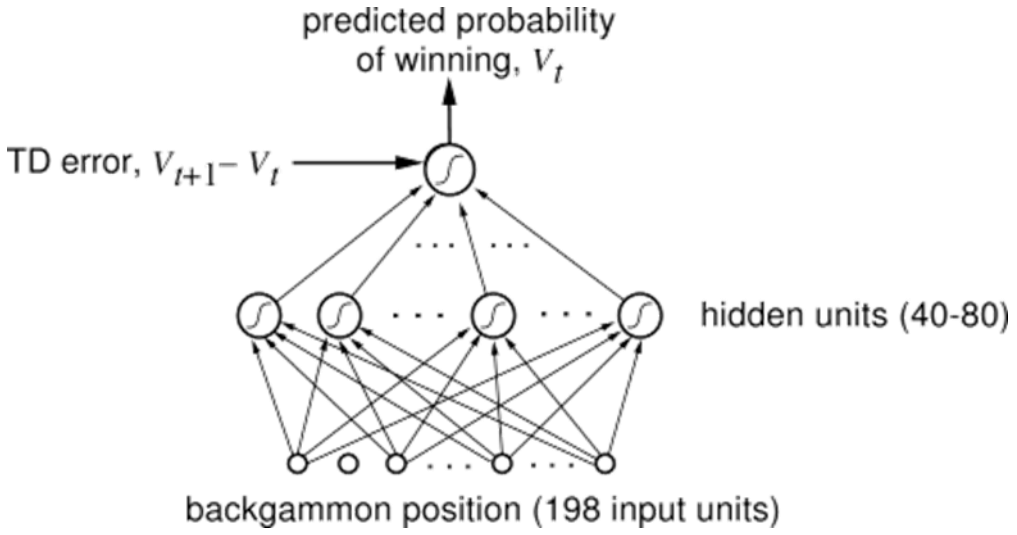


Figure 2.14: Neural network architecture of earlier TD-Gammon versions. Only 1 of the 4 outputs is shown. (Sutton & Barto, 1998)

Using the raw encoding only, TD-Gammon was able to surpass in strength all available backgammon programs of the time. Adding features representing expert knowledge of the game, e.g. "pipcount", a heuristic progress measure (the total distance of pieces from the goal) that were taken from Tesauro's previous program NeuroGammon, TD-Gammon was able to reach a playing level equaling, and maybe surpassing, the best human players of the time.

The parameters used were as follows: the learning rate *a* was fixed at 0.1 and the bootstrapping value λ initially was 0.7, but later it was reduced to 0, because no significant difference was noticed and, without eligibility traces, the updates were simpler and faster to compute. Hidden layer units started at 40 and were gradually increased to 160 in the final version. Training time was also increased as time went by, reaching 6 million games in the final version (TD-Gammon 3.1).

Eventually TD-Gammon was retired, but the influence in the backgammon world was tremendous. Firstly, many backgammon programs successfully replicated the TD(λ) + neural network combination, the first examples were the Snowie and the Jellyfish programs. Currently, all strong backgammon programs use some variant of the original TD-Gammon setup:

- BgBlitz, a shareware program authored by Frank Berger and frequent participant in backgammon computer Olympiads, uses neural networks with 200 hidden units and self-play TD(0) training procedure.
- GNUBG, an open source program also participating many times in the backgammon computer Olympiads, uses 3 different neural networks (contact, crashed, race) for the three different stages of the game. Earlier versions were trained by TD(0), whereas nowadays all training is supervised which means that the targets of the training set were calculated by rollouts by the same program. All neural networks have 128 hidden units.
- eXtreme Gammon is another commercial program believed by many to be the best in the world in standard backgammon. Little details are known except for the description in its site (Extreme Gammon, 2015): "eXtreme Gammon engine uses a Neural Network to determine the value of every position. The Neural Network has been trained for years to achieve a very high accuracy in estimating positions"

TD-Gammon and the other backgammon programs later inspired by it had also great influence to the human backgammon world. In some situations, TD-Gammon suggested different moves than the ones thought as best at the time. Expert players began carefully studying the program's evaluations and rollouts and began to change their concepts and strategies. As a result, the new knowledge generated has been widely disseminated and the overall level of play in backgammon tournaments has greatly improved in recent years.

2.3.1.2 TD-Leaf and KnightCap

While TD(λ) and neural networks were very successful in standard backgammon, researchers struggled to effectively apply the same procedure to other deterministic board games such as chess, checkers, othello or Go. In all these games, finding an evaluation function accurate enough to evaluate a position at one-ply is a very difficult task, since there may occur tactical sequences that end the game after a few moves. For example, in

chess sequences of forced mate in 2, 3, 4 or more moves frequently occur both at actual play and in analysis. Consequently, almost all successful programs in these games use some sort of deep lookahead involving variants of minimax search. As this search results in exponentially more states to be evaluated, expensive evaluation functions such as neural networks are not typically used and fast linear functions are preferred because of their speed.

The first successful TDL approach in domains, where search is important, was the chess program KnightCap (Baxter, J., Tridgell, A., & Weaver, L., 1998a; 1998b; 2000) that introduced a new TDL algorithm, TD-Leaf(λ). TD-Leaf(λ) is similar to standard TD(λ) with the only change being the root and the target of the updates; instead of the root being the current state and the target being the next state of the actual game, now the root is the leaf node of the principal variation searched in the current state and the target is the leaf node of the principal variation of the next state. Figure 2.15 explains the differences of the various backups in diagrammatic form. The idea is that the updates are now more informed about the tactical sequences found by search, so updates are more accurate in situations where tactical sequences play an important role.

KnightCap with TD-Leaf was able to reach master level play by training a linear evaluation function of 5,872 expert features in 300 games but had two major concessions. Firstly, initial results showed that self-play training was very slow, so the training was performed in an online chess server, ICC[1], with humans as opponents. For this reason additional measures were put in place, so that updates were not performed, when blunders from the opponents were recognized. Secondly, in order to give a head-start in the knowledge gained, the features representing the material value of the pieces were initialized to the default values. Good initial conditions were important for fast convergence, because without "smart" initialization of the features learning was found to be very slow.

Under this setup, KnightCap reached a performance of 2150 ELO in blitz[2] time controls without utilizing an opening tree. When an opening tree was added, KnightCap

[1] Inter Chess Club: www.chessclub.com
[2] A fast time control that gives each player at most 5 minutes for the whole game.

reached a rating 2,400-2,500 (peak 2,575) on the ICC server, a rating equivalent to a human International Master (IM) level. The parameters used were $\lambda = 0.7$ and $a = 1.0$, the latter was unusually high on purpose in order that big updates and, as consequence, faster learning would be encouraged. These parameters seems to be fixed throughout the learning process.

The authors of KnighCap developed another algorithm called TD-Directed(λ). The difference with TD-Leaf(λ) is that the updates are performed on the actual states played in the game and not on the leaves, like TD(λ). Unlike TD(λ), the states are selected based on the result of a *d*-ply search and not by 1-ply. TD-Directed was also tried on chess but found to be less fast than TD-Leaf.

A comparison with TD(λ) in standard backgammon was also performed. The authors took LGammon, an already trained neural network with the TD(λ) procedure of TD-Gammon, and tried to improved it by further training with TD-Directed and TD-Leaf. However, after 50000 training games the resulting networks were not found to be statistically better than the original. This is confirmed from our own preliminary experiments, where we tried to train a neural network from scratch using self-play, TD-Directed(0), and d=2 and found it to be very slow on the time used. This is because of the huge branching factor found in backgammon games which makes d-ply search very computationally expensive.

TD-Leaf was successfully applied to checkers (Schaeffer, Hlynka, & Jussila, 2001) where it was able to train a linear evaluation function with self-play, which was found to be equally strong to Chinook, the World Man Checkers Champion, which had an evaluation function that was manually tuned over a period of 5 years. The learning rate *a* was set to 0.01 and the λ parameter was chosen to be 0.95. These values, however, were not tuned; rather their values were influenced by the KnightCap research.

2.3.1.3 Rootstrap and Treestrap

Extending the idea of TD-Leaf algorithm, one could also update towards the trajectory of the principal variation, not only on the leaf nodes (TD-Leaf) or on the actual states

of the game (TD). This way the tree created during the search process is better utilized by the learning process, since the potential exists to update much more per move played. This is exactly what is done in the RootStrap and TreeStrap algorithms (Veness, et al., 2009). The author stresses another advantage: the updated states are more representative of the types of states that can occur in a search-based evaluation, a potential problem with TD-Leaf which only updates leaf nodes. In Rootstrap the target of the update, when the agent is in a state of "thinking" its move via search, is the leaf state in the principal variation, while in Treestrap all interior nodes of the tree are updated. The different backups of all the algorithms discussed so far are shown in Figure 2.15:

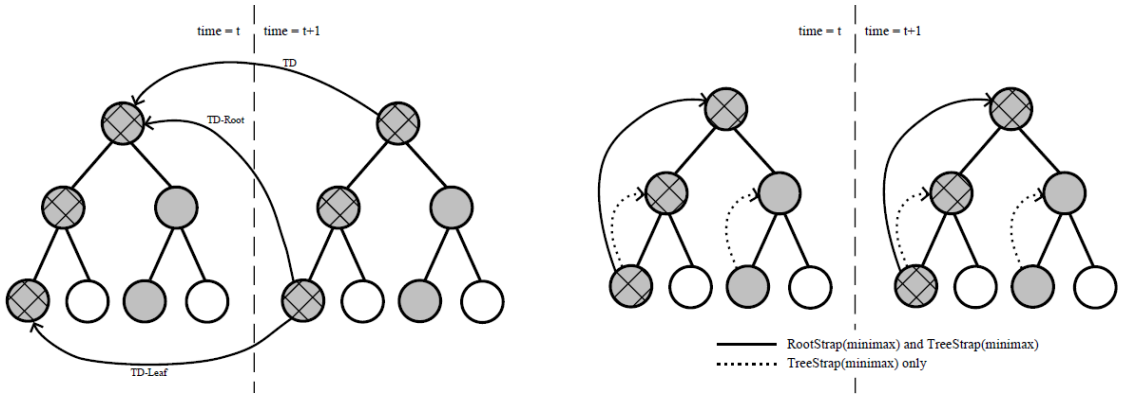


Figure 2.15: Diagrams of various TD backups (Veness, et al., 2009)

The authors also made alpha-beta versions of the algorithms exploiting the $\alpha\beta$ cut-offs produced by the alpha-beta algorithm by using a one-sided loss function and truncating the evaluation function inside the $\alpha\beta$ cutoff bounds. All the algorithms were tested in the Meep chess program, which used 1812 linear features that were initialized to small random values and a small opening book to maintain diversity in the starting moves. After 10,000 games under this training regime, Meep was able to learn a good evaluation function, with the better performing algorithm to be the TreeStrap(αβ). In blitz play on the ICC server, Meep reached 1,950-2,197 Elo under self-play training, and 2,154-2,338 Elo when trained against Shredder, a very strong chess engine.

The learning rates used were: 10^{-6} for TreesStrap (minimax) and 10^{-7} for Treestrap(αβ). Both KnightCap and Meep used online incremental updates, meaning that the updates were performed after each move of a training game.

2.3.1.4 Self-play or expert tutoring?

When we have an expert, it is tempting to use it to learn against it instead of self-play training. As it was shown in the previous sections, KnightCap and Meep benefited from having an expert as an opponent; KnightCap by playing against humans (not always expert) and Meep by playing against a strong chess engine (Shredder). This was also confirmed in a study in the game of backgammon in (Wiering, 2010). Three paradigms were tested, self-play training, training by watching experts and training by playing against an expert. Training against an expert was the fastest. The TD(λ) setup used had many similarities to the one proposed in this thesis; a) neural networks were used as function approximators, as it is usual in backgammon, b) learning was offline, meaning that the updates were performed after the game ended, c) the second player position was inverted, making the network see only the position as the first player. However, unlike our setup, the learning trajectory started from the beginning, probably resembling the forward offline method discussed in Section 3.2.2.1.

The experiments showed that learning by observing an expert was approximately two or three times slowest than the other methods. Learning against an expert was the fastest, closely followed by self-play. Unfortunately, the resulting networks were tested only against themselves and not against some known benchmark such as pubeval (Tesauro, 1994), so we do not know exactly how strong they were. The learning rate parameter *a* was set at 0.01, and interestingly, the λ value of 0.6 was found to give the best performance, something not observed in TD-Gammon or in our results. In line with our observations, starting with high values of λ (0.8) seemed to make the training faster.

2.3.1.5 Learning from databases

One could also try to learn from a database of games already played by expert players. Looking at the results of the previous paragraph, this seems to be the same as training

by watching an expert, however there are several advantages. Firstly, the agent does not need to waste resources on exploring the environment, which is important in games where the function that produces the available moves is costly. Secondly, not learning the values of weaker moves, the disadvantage of learning only from expert play can be easily mitigated by introducing randomness or other techniques. A successful example is the Giraffe chess program (Lai, 2015), where a variant of TD-Leaf was used to train a deep neural network utilizing a database of random positions taken by databases of computer games.

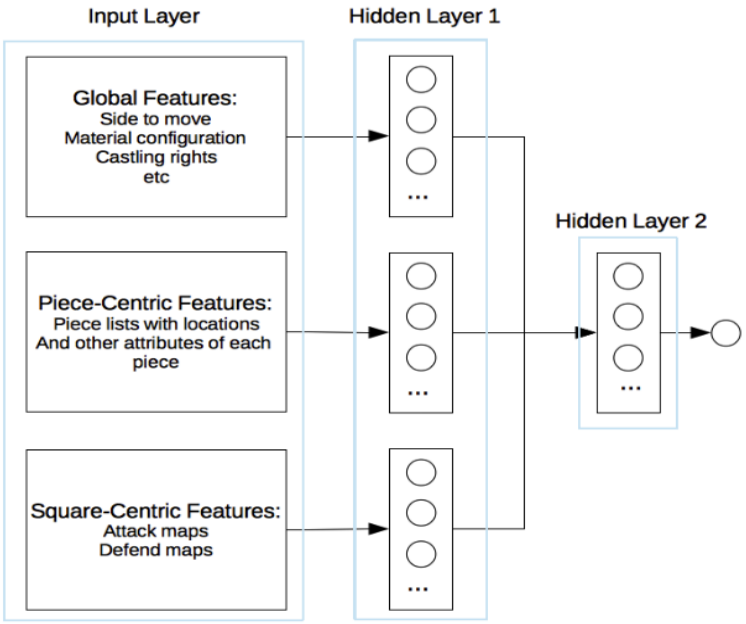


Figure 2.16: Giraffe's neural network architecture (Lai, 2015)

Instead of using each position directly, the author introduced variations by randomly applying a legal move to each position before putting it in the actual training database. This approach gives variety to the positions and at the same time it keeps the distribution close to one actually encountered during game-play. These positions were then the start of a ten-move self-play game that was used by the TD-Leaf algorithm to train the network using standard backpropagation. The learning rate was auto-tuned by the Adadelta algorithm (Zeiler, 2012) and the λ value was fixed to 0.7. The resulting network reached the performance of an international master (IM) in blitz play.

We believe that whenever possible database of experts should be utilized at least for initial training. If this is not enough to reach adequate performance, self-play can be used afterwards. However, for our purposes of building a strong agent for the backgammon variants in which we are interested, such databases do not exist for the games Plakoto and Fevga. So we concentrate our efforts on learning by self-play, a useful paradigm for agents trying to learn new problems in general.

2.3.1.6 Summary

The most key characteristics of the TDL training setups described so far, along with our method, are shown in Table 2.1. In the first row the program name is shown except for the backgammon program trained in (Weiring, 2012) where the name of the researcher is shown.

Table 2.1: Summary of the key characteristics of various TDL applications in games

Program	**TDGammon**	**KnightCap**	**Chinook**	**Meep**	**Weiring Bg**	**Giraffe**	**Palamedes**
Game	BG[1]	Chess	Checkers	Chess	BG[1]	Chess	BG[1]
Algorithm	TD(λ)	TD-Leaf(λ)	TD-Leaf	TreeStrap (αβ)	TD(λ)	TD-Leaf(λ)	TD(λ)
Parameter λ	0.7, 0	0.7	0.95	-	0.6	0.7	decreasing
Parameter α	0.1	1.0	0.01	10^{-7}	0.01	Autotuned (AdaDelta)	decreasing
Function Approx.	Neural Networks	Linear Weights	Linear Weights	Linear Weights	Neural Networks	Deep Networks	Neural Networks
Training setup	Online Self-play	Online vs Expert	Online Self-play	Online vs Expert	Offline vs Expert	Online play from db of positions	Reverse Offline Recalc Self-play

The λ parameter selected in most setups is 0.7 or close to it, noting that most attempts did not optimize this value. An exception is TreeStrap, because the λ parameter does not exist in this algorithm. The learning rate has much diversity in the setups examined, starting from 10^{-7} in Meep and reaching 1.0 in KnightCap. This value seems to be highly dependent on the features and setup used. Objectively speaking, an automatic tuning

[1] BG = Backgammon games

algorithm (like Giraffe) or a decreasing rate like the one we used in our setup should be used, in order to be able to escape local optima when stuck. In practice, these two algorithms are very difficult to tune together, because the time needed for all the training runs is usually in the factor of days due to the complexity of the games.

2.3.2 OTHER METHODS IN BACKGAMMON

After the success of TD-Gammon, many researchers tried to apply different techniques to learn to play standard backgammon but without much success. This section presents some of the most important work that used different kind of methods other than TDL+NN.

The first attempt after TD-Gammon was by Pollack, Blair and Land (1997), when they presented HC-Gammon, a program that used a much simpler Hill-Climbing algorithm that trained the weights of neural networks. Under their model the current network is declared `Champion,' and by adding Gaussian noise to the biases of this champion network, a `Challenger' is created. The Champion and the Challenger then engage in a short tournament and, if the Challenger outperforms the Champion, small changes are made to the Champion in the direction of the Challenger biases. HC-Gammon won only 40% of the games against the pubeval program.

Another interesting work is that of Sanner et al. (2000), whose approach is based on ACT-R theory of cognition (Anderson & Lebiere, 1998). Rather than trying to analyze the exact board state, they defined a representational abstraction of the domain, consisting of general backgammon features such as blocking, exposing and attacking. They maintained a database of feature neighborhoods, recording the statistics of winning and losing for each such neighborhood. All possible moves were encoded as sets of the above features; then, the move with the highest win probability (according to the record obtained so far) was selected. This system reached a 45.94% win rate against pubeval.

Darwen (2001) studied the coevolution of standard backgammon players using single and multi-node neural networks, focusing on whether non-linear functions could be discovered. He concluded that with coevolution there is no advantage in using multi-node

networks and that coevolution is not capable of evolving non-linear solutions. His best agent scored 52.7% against pubeval.

Qi and Sun (2003) presented a genetic algorithm-based multiagent reinforcement learning bidding approach (GMARLB). The system comprises several evolving teams, each team composed of a number of agents. The agents learn through reinforcement using the Q-learning algorithm. Each agent has two modules, Q and CQ. At any given moment only one member of the team is in control and chooses the next action for the whole team. The Q module selects the actions to be performed at each step, while the CQ module determines whether the agent should continue to be in or relinquish control. Once an agent relinquishes control, a new agent is selected through a bidding process, whereby the member which bids highest becomes the new member-in control. Their system reached a 56% winning rate against the pubeval benchmark.

GP-Gammon was another line of research by Azaria and Sipper (2005), where Genetic Programming (GP) (Koza, 1992) was applied to evolve computer programs to play backgammon. GP starts with an initial set of general and domain specific features and then lets evolution evolve the structure of backgammon-playing strategy. In addition, GP readily affords the easy addition of control structures such as conditional statements, which may also evolve automatically. Two methods were tried, evolving backgammon strategies with external opponent as teacher and GP with self-learning. The self-learning approach was the better of the two, scoring 62.4% wins against pubeval, whereas the teacher-learning approach scored 56.8% wins.

All the above approaches have one major flaw: the performance is measured in win rate against pubeval and not in points per game, from which we assume that the learned agents try to learn the game only to win and do not account the double wins as something more valuable. However, building a win only agent takes much of the complexity out of the game, since the double wins are very common in standard backgammon (Section 4.3.1).

Moreover, some of the above results were evaluated with very few test games against pubeval: GMARLB-Gammon only with 50, HC-Gammon used 200 and GP-

Gammon 1000. We believe that only ACT-R-Gammon, which used 5,000 games, and Darwen which used 10,000 games, have sufficient amount of games to compensate for the randomness of the game.

Finally, the MCTS algorithm (Section 2.2.4.2.2) was also applied to standard backgammon in (Van Lishout, Chaslot, & Uiterwijk, 2007). In this work, the UCT algorithm was used in the selection phase and random games in the playout phase. They also built a high performance move generation algorithm so that the random games can be finished quickly. The resulting program, MC-Gammon 1.0, makes a simplification of the game rules by declaring the game won, when all the checker are inside the home board. An updated version, which played the full game, competed in the backgammon computer Olympiad in 2007, losing all its games.

Chapter 3

CHAPTER 3: TRAINING NNS TO PLAY BACKGAMMON GAMES USING TD

3.1 Learning architecture

The architecture of the learning system that we used for all our experiments is shown in Figure 3.1.

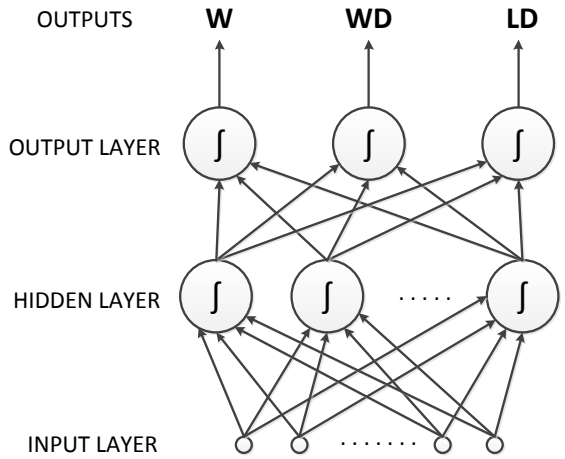


Figure 3.1: The neural network architecture used in our learning system. All units of the hidden and output layer use sigmoid transfer functions.

The learning procedure is executed as follows: we start by creating a sequence of game positions beginning with the starting position and ending in the last position, when the game is over. For each of these positions we use the backpropagation procedure of the neural network to compute the TD(λ) update. The various methods of selecting input-output-target sequences are mentioned in Section 3.3.2.1. The self-play learning process is repeated, until we can no longer improve the NN.

The inputs of the NN may represent the board position and/or some other features. We used a modified version of the unary truncated encoding scheme used by TD-Gammon (Section 2.3.1.1) to map the board positioning of the checkers to the inputs of the neural network. We used three binary outputs to describe the final outcome of the game from the side of the first player. The first output (W) represents the outcome of the game, win or loss; the second output (WD) represents whether a double game is won; and the third output (LD) represents whether a double game is lost.

Under these training conditions, the neural network learns the "probability" of all three outputs at any time in the game, also called position equity. For the creation of the game sequences, we used the same NN currently in training to select the moves for both sides. Whenever a move selection must be made, the agent scores the states resulting from all legal moves by combining all three outputs. At every time-step the agent scores all legal moves available and selects the one with the highest score.

For the evaluation of the learned agents, three procedures were examined:

a) Evaluation against an independent benchmark opponent, the open source program Tavli3D 0.3.4.1 beta (Varouhakis, 2007), the only freely available program that can play all the backgammon variants that we are interested in.

b) Evaluation against stored weights taken by the agent at different stages of the learning process. Examples are weights after 10^4 training games, weights after 10^5 training games etc.

c) Evaluation against previously trained agents.

During the training procedure the weights of the network were periodically saved and tested with procedures (a) and (b), until no more improvement was observed. All the tests were conducted in matches of 10000 games each. The result of the tested games sum up to the form of estimated *points per game* (ppg) and is calculated as the mean of the points won and lost.

3.2 Initial Experiments

3.2.1 DETERMINING THE EFFECT OF EXPERT VS RAW FEATURES

Our first learning experiment was done to determine how the agents learned with and without expert features. This was mainly done to Plakoto and Fevga variant since in Portes we can use the standard backgammon research available by TD-Gammon (Tesauro, 1992; Tesuaro 1995;Tesauro, 2002). We used the same approach for all the variants examined: First, we trained a neural network with inputs consisting only of the raw position of the board. As with TD-Gammon, we observed a significant amount of learning even without the addition of "smart" features. It only took few thousands learning games for the

agent to surpass the playing performance of the Tavli3D benchmark program for Plakoto and Fevga games and the pubeval benchmark program for Portes. We evaluated the resulting agent and tried to improve its evaluation function by identifying expert features. A second neural network was trained from scratch including these expert features to the inputs of the previous NN architecture.

3.2.1.1 Experiments in Fevga

The raw board position in the game of Fevga was encoded as follows: for every point of the board four binary inputs were used, each one designating whether there was one, two, three, or four and more checkers in the point. This coding thus used 96 input units for every player to encode the checkers inside the board and additional 2 units to encode the number of checkers off the board, for a total of 194 units. We named the agent trained with this coding scheme and the procedure described earlier **Fevga-1.**

Fevga-1 was assessed as average in strength by human standards. Concepts learned include the understanding of protecting the starting quadrant and attacking the starting quadrant of the opponent in the early phase of the game, as well as the smooth spreading of checkers. However, a major weakness was also found: a complete disregard for primes. The creation and sustainment of the prime formation is considered by human experts the most powerful strategy available in the Fevga variant.

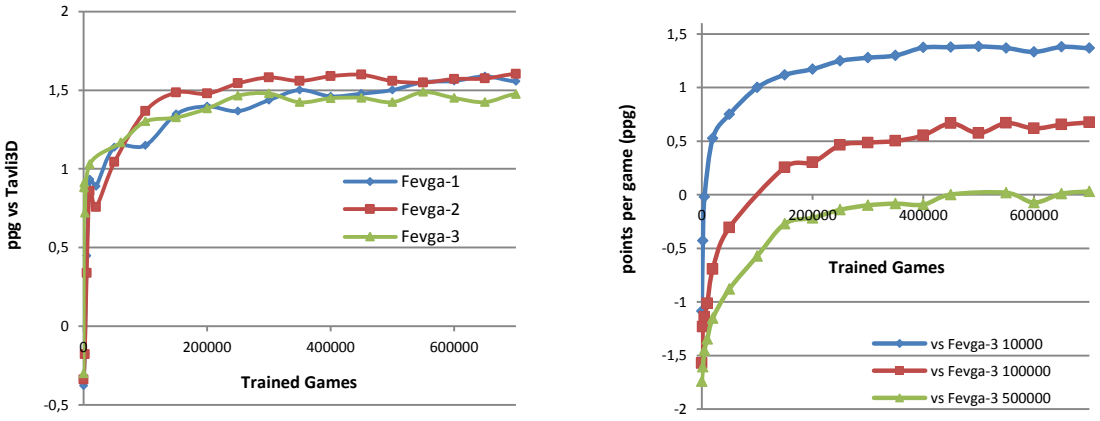


Figure 3.2: Left. Training progress of all agents against the Tavli3D benchmark program. **Right.** Training progress of Fevga-3 against stored weights.

Adding expert features. Given the drawback in the playing strategy of Fevga-1, it was decided to add the special knowledge of primes in the inputs of the neural network as smart (or expert) features. The different formations of primes were divided in two categories according to their significance: a) early primes that are formed in the first two quadrants of the player and b) late primes that are formed in the last two quadrants as well as between the 4th and the 1st quadrant. Late primes are more powerful, because they restrict the opponent earlier in his/her development and frequently result in the winning of a double game. These features take the form of four binary input units of the neural network that are enabled, when the player and/or the opponent makes a prime and at least one opponent checker is left behind it. In addition, two more special features common to regular backgammon were also added: a) one input unit for the *pipcount* of each player, which is the total amount of points (or *pips*) that a player must move his checkers to bring them to the home board and bear them off, and b) two input units for the existence of a *race* situation, which is a situation in the game where the opposing forces have disengaged, so there is no opportunity of further blocking. The total number of input units in this encoding (which we named **Fevga-2**) is 201. The evaluation of Fevga-2 (Figure 3.2 Left) showed only a marginal increase in performance that was verified by manual human analysis: while not totally ignorant of the value of prime formation as Fevga-1, Fevga-2 failed to grasp the essence of primes.

Adding intermediate reward. To clarify more precisely the importance of primes, a third neural network was trained where the agent learned with the same input units as Fevga-2, but with one important difference: when reaching a position with a prime formation, the target of the TD update was made a constant value instead of the next position value. This constant value was for primes of type (a) equivalent with winning a single game and for primes of type (b) equivalent with winning a double game. In other words, intermediate rewards were introduced, when primes were formed in the game. This had the result that the strategy learned was a strategy based on the creation of primes, which is roughly equivalent to what is perceived by experts as the best strategy. We named this agent **Fevga-3**. Its training progress can be seen in Figure 3.2 Right.

Indeed, after manual examination, the playing style of Fevga-3 was very similar to the way humans play the game. Not only did it recognize the value of primes and did not lose opportunities to make one, but was also able to play moves that facilitated the creation of primes at later stages. The results of the evaluation against the Tavli3D benchmark show that Fevga-2 gains slightly more points per game than Fevga-3 (+1.61ppg vs +1.52ppg). However, when we compared Fevga-2 to Fevga-3 by selecting the best set of weights and playing 5000 games against each other, the results of this match showed a marginal superiority of the Fevga-3 player (+0.03ppg).

Table 3.1. Analysis of the match Fevga-2 vs Fevga-3

Result/Points	Fevga-2	Fevga-3	Total
Single Wins	1704 (34.08%)	2513 (50.26%)	4217 (84.34%)
Double Wins	556 (11.12%)	227 (4.54%)	783 (15.66%)
Total Wins	2260 (45.2%)	2740 (54.8%)	5000
Total Points	2816	2967	5783

The analysis of the match between Fevga-2 and Fevga-3 (Table 3.1) gives some interesting information. The "human" strategy of Fevga-3 seems to win more games (54.8%). Nevertheless, the final result is almost equal, because Fevga-2 wins more double games (11.12% vs 4.54%). This is also confirmed after careful analysis of the results against Tavli3D: the two agents win the same number of games, but the Fevga-2 emerges superior, because it wins more double games. We believe that the Fevga-2 strategy is better against weak opponents, because in the long run it wins more points than Fevga-3 due to more double games won. But when playing against a strong opponent, a little better strategy seems to be the more "human-like" strategy of Fevga-3, which maximizes total won games at the cost of doubles. Looking it from another perspective, we can say that the two versions have different playing styles: Fevga-2 plays more aggressively, trying to win more double games, while Fevga-3 plays more cautiously, paying more attention in securing the win than risking for doubles.

3.2.1.2 Experiments in Plakoto

Encoding the raw board position. The input units of the neural network for the first version of Plakoto were the same 194 units of Fevga-1 plus 24 binary units for every player that indicated if the player had pinned a checker of his opponent at each point of the board. Thus, there were 242 input units in total. The agent with this coding scheme was named **Plakoto-1**. As in Fevga, after only a few thousand games Plakoto-1 easily surpasses in strength the Tavli3D benchmark program. This improvement finally reaches a peak performance of about 1.15ppg (Figure 3.3, Left).

Using manual play, the level of Plakoto-1 was assessed as average by human standards. Strong aspects were the understanding of the value of pinning the opponent, especially in the home board. At the same time, it was also careful not to leave open checkers, thus not giving the opponent the chance to pin, because it understood that this will greatly increase the chances of losing. Mistakes, however, occurred often, when it had to select a move that left at least one checker open: it did not take into account the possibility of the opponent pinning the open checker in the next move, thus rejecting moves resulting in positions with little or no chance for the opponent to pin and preferring moves resulting in open checkers very easily pinned.

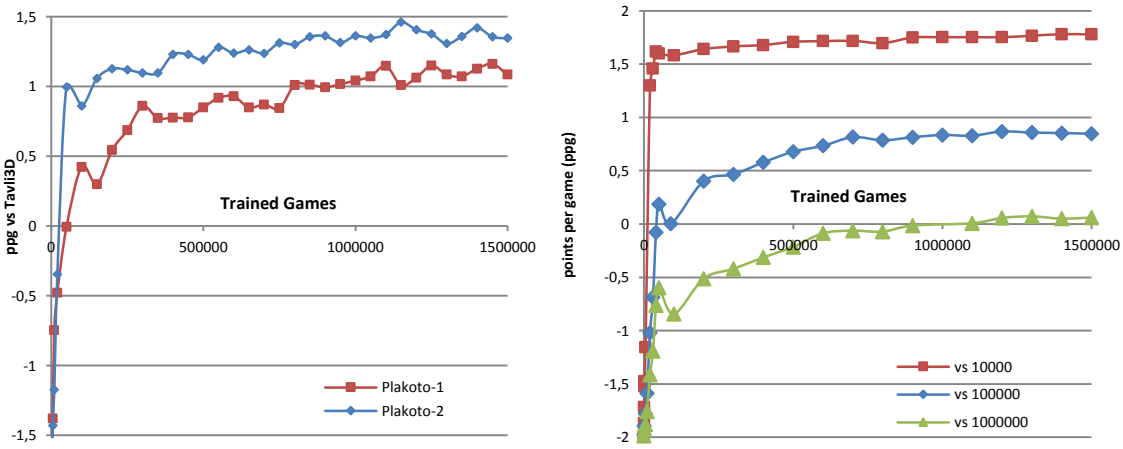


Figure 3.3: Left. Training progress of Plakoto-1 and Plakoto-2 against Tavli3D. **Right.** Training progress of Plakoto-2 against stored weights at 10,000, 100,000, and 1,000,000 games trained.

Adding expert features. The following single feature was able to increase the performance considerably: the 24 binary inputs representing the pins of the opponent at each

point were replaced by the probability of the opponent pinning a point, if an open checker of the agent exists. This probability starts at 0/36 = 0, when no dice roll exists that can pin the open checker or no open checker exists and maxes to 36/36 = 1, when all dice rolls pin the open checker or the checker is already pinned. This added feature required no additional input units, as it utilized units already used by the neural network, only a little more computational overhead for computing the pinning probabilities. The resulting agent was named **Plakoto-2**. Compared to Plakoto-1, Plakoto-2 achieved better peak performance by about 0.3 ppg against Tavli3D (Figure 3.3, Left). A match of 5000 games between the two agents resulted in a comfortable win for Plakoto-2 (6687-1771, +0.98 ppg), further confirming the superiority of Plakoto-2. The level of Plakoto-2 was assessed as that of an experienced player.

Figure 3.2 Right and Figure 3.3, Right show the training progress of Fevga-3 and Plakoto-2 against previously stored weights. In both figures we see that the initial strategy improves rapidly for the first few thousand games and then improves more slowly to its peak performance.

3.2.2 DETERMINING THE TARGET OF THE UPDATE

In order to find the best move in a given situation, backgammon programs usually score each possible afterstate (that is the states resulting, *after* the player has played a move) and select the move that produces the afterstate with the biggest score.

An important implementation detail for a TD+NN learning system is the selection of input-target pairings for the TD update. In previous work we split up each training game into two training sequences, one for the afterstates of the first player and another for the afterstates of the second player, and we updated these sequences separately (Figure 3.4b). In this work we made one simple, yet very effective improvement: instead of splitting up each training game in two, we keep one training sequence and we update each player's afterstate using as target the inverted value of the other's player afterstate on the next move (Figure 3.4c). Both methods flip the board, so as both players' afterstates are given to the neural network, as if it is the first player to move. This is different from the approach used

originally by TD-Gammon (Figure 3.4a), where there was no flipping of the board and the neural network learned to play the game for both sides, identifying the side to move by two binary inputs. We believe the board-flipping approach has the potential of getting improved performance, as the expressiveness of the neural network is increased.

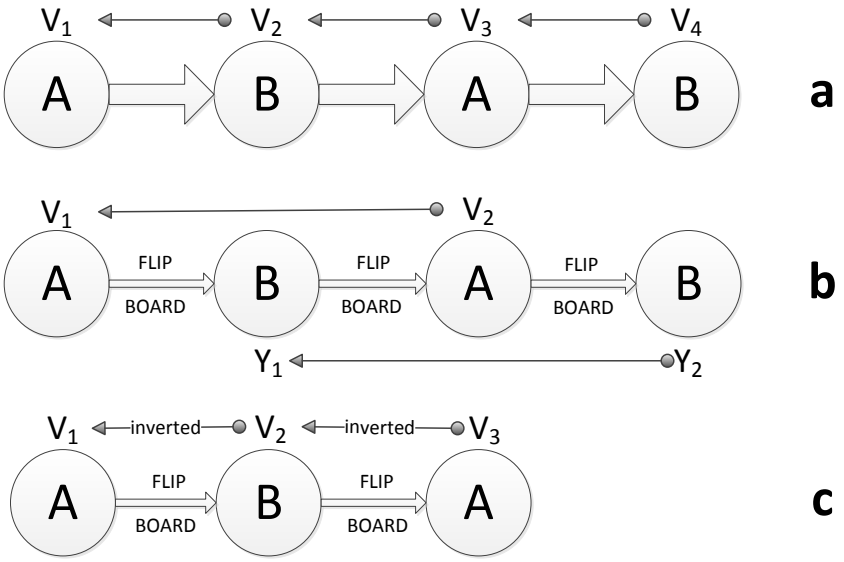


Figure 3.4: Alternate updating methods of the temporal difference in two player zero-sum games. Method a: Update the values without flipping the board. Requires input(s) to designate which player is on the move. Method b: Updates are split in two. Method c: Updates are done on the inverted value of the next player. Circles indicate a position after a player (A or B) has made a move (afterstate).

3.2.2.1 Sequence creation and how to update

Contrary to standard backgammon, when we started making programs for the variants Plakoto and Fevga, there was no other program close to expert play, nor were databases of games available. Therefore, self-play was for us the only option for creating a game sequence. We examined the following options for creating and updating a self-play game:

- Learning online (each update is done immediately after a move is played).
- Learning offline (updates are done incrementally after the game ends).
- Forward offline: Updates are done starting from the first position of the game and ending at the terminal position.
- Reverse offline: Updates are done starting from the terminal position of the game and ending at the first.

- Reverse offline recalc: As previous, but recalculate target value after each update.

The intuition of updating backwards an offline game is that updates of non-terminal states will be more informed as the reward of the outcome of the game is received on the first update of the game. This is enhanced with the addition of the recalculation of the target value. Online updates have the benefit of learning while the game is in progress; however, there is a chance that at the start of training, where moves are more or less random, the agent will get stuck or progress slowly.

Preliminary experiments with all of the above methods showed that the slowest method was forward offline, particularly in the Fevga variant, with the others resulting in more or less the same performance (Figure 3.5). The reverse offline method with recalculation of the target value learns the fastest than all others at the start of the training and continues to have good performance afterwards. The downside is that more computation is needed in order to recalculate the target value at every step. However, this was not felt in our case since the creation of a game sequence is much more time consuming than the time to make the updates. Even with slower learning progress, all methods were found to reach the same level, so whatever final performance gains described later in the paper were only due to changing the updating method from (b) to (c) (Figure 3.4).

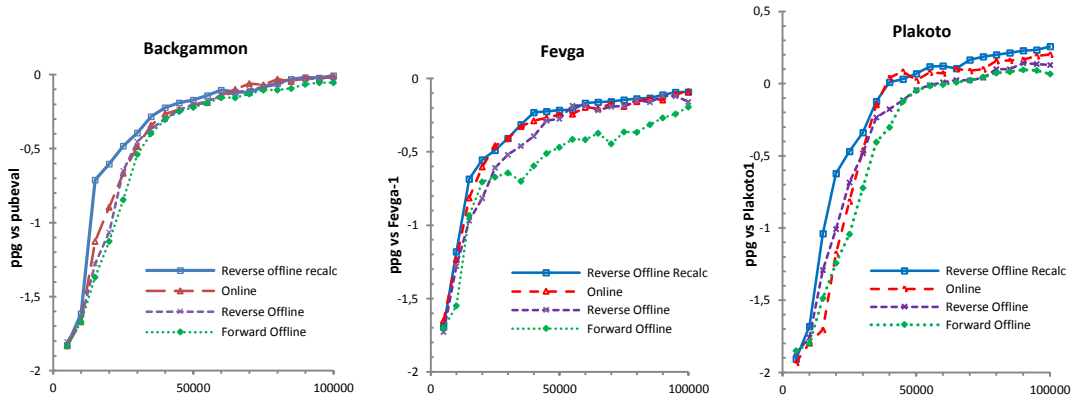


Figure 3.5: Training progress of methods for sequence creation and update in Backgammon (left), Fevga (middle) and Plakoto (right). Every line is the average of 10 different training runs starting from the same random weights. For speed reasons, NNs in all games have 10 hidden units and no expert features. Benchmark opponents are pubeval for backgammon, Fevga-1 for Fevga and Plakoto-1 for Plakoto.

In our previous experiments, we used the forward offline method. Following the experiments mentioned in this section, all experiments from now on were conducted using reverse offline recalc.

3.2.2.2 Results in the Plakoto and Fevga variants with expert features

We compared the proposed training method to the previous one (Section 3.2.1) by training again the NNs with the added expert features. For Plakoto, the new agent was named Plakoto-3 and has exactly the same inputs as Plakoto-2. For Fevga, we trained two new NNs: Fevga-4 has the same procedure as Fevga-2, whereas Fevga-5 has the same intermediate reward as Fevga-3. Table 3.2 shows all the techniques used by the various versions examined in this paper.

Results in 3.2.1.1 showed that Fevga-2's strategy was much different from the one considered by the human experts, even with features recognized the presence of primes (six consecutive made points) in a position. To clarify the importance of primes more precisely, a new NN was trained (Fevga-3), where the agent learned with the same input units as Fevga-2, but with one important difference: when reaching a position with a prime formation, the target of the TD update was made a constant value instead of the next position value. This had the result that the learned strategy was based on the creation of primes, which is roughly equivalent to what is perceived by experts as the best strategy. Results showed that the riskier strategy of Fevga-2 scores more points against the benchmark program Tavli3D than Fevga-3, but when getting them to play against each other Fevga-2 was a little bit inferior. To preserve continuity with our previous work, we continued to benchmark our training progress with the open source program Tavli3D, which at the time of writing was the only open source program that can play these variants.

All networks had 100 hidden neurons and were trained to 1.5 million games. For simplicity, we fixed the value of λ to zero for the experiments conducted in this paper. For $\lambda>0$ and reverse updates, care must be taken when taking future time steps into consideration: since every time step is viewed as the first player, any value taken by future time steps that is not a move by the player making the current update must be inverted. As the initial

training of Fevga-2 and Fevga-3 were only 700,000 games, we extended their training (with the same initial λ=0.7) to match the new ones. During the training, we periodically saved the weights of each NN and we tested the networks against Tavli3D for 10,000 test games each, half as the first player and half as the second player (Figure 3.6). The result of the tested games sum up to the form of estimated *points per game* (ppg) and is calculated as the mean of the points won and lost.

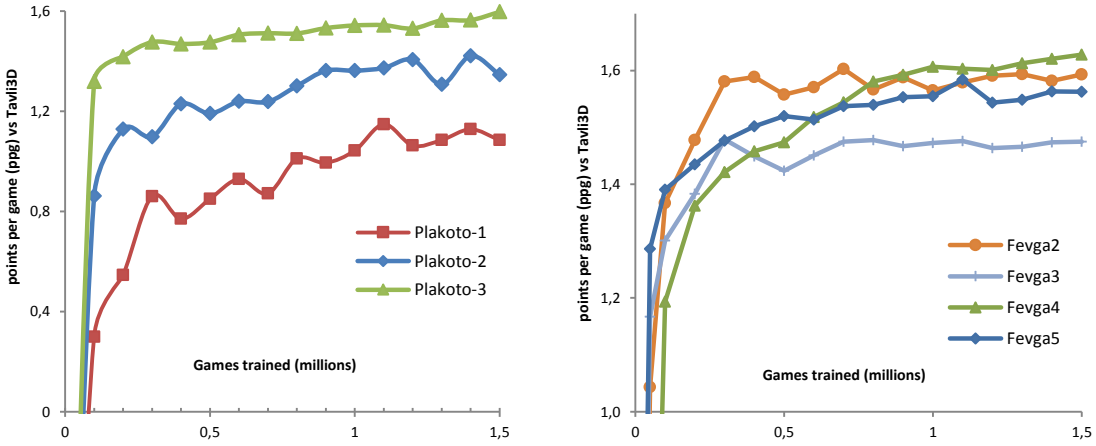


Figure 3.6: Training progress of all trained NNs against the Tavli3D benchmark program in the Plakoto variant (**Left**) and the Fevga variant (**Right**).

We also tested the best set of weights of each NN by playing tournaments against each other at (1-ply) as well as by implementing a simple look-ahead procedure using the expectimax algorithm (Michie, 1966) at 2-ply depth (Table 3.3). In order to speed up the testing time, this expansion of depth-2 was performed only for the best 15 candidate moves (forward pruning). For the same reason, the total amount of testing games using 2-ply was reduced to 1,000 per test.

The results in Plakoto show a significant increase in final performance. The performance of Plakoto-3 at 1-ply is equivalent to the performance of Plakoto-2 at 2-ply against Tavli3D. Additionally, Plakoto-3 learns faster than the other two agents.

Table 3.2: Summary of techniques used by the various agents

Plakoto Agent	Updating method (Figure 3.4)	Sequence creation and update direction	Fevga agent	Updating method (Fig. 3.4)	Sequence creation and update direction	Intermediate reward
Plakoto-1	b	Forward offline	**Fevga-2**	b	Forward offline	No
Plakoto-2	b	Forward offline	**Fevga-3**	b	Forward offline	Yes
Plakoto-3	c	Reverse offline recalc	**Fevga-4**	c	Reverse offline recalc	No
			Fevga-5	c	Reverse offline recalc	Yes

Table 3.3: Comparison of various agents at 1-ply and 2-ply for Plakoto (Left) and Fevga (Right). All results are in points per game (ppg) with respect to the player on the row. Players on columns always use 1-ply.

	Tavli3D	Plakoto1	Plakoto2		Tavli3D	Fevga-2	Fevga-3	Fevga-4
Plakoto-1	1-ply: +1.15 2-ply: +1.36	*	*	**Fevga-2**	1-ply: +1.60 2-ply: +1.61	*	*	*
Plakoto-2	1-ply: +1.46 2-ply: +1.60	1-ply: +0.98 2-ply: +1.35	*	**Fevga-3**	1-ply: +1.52 2-ply: +1.53	1-ply: +0.03 2-ply: +0.49	*	*
Plakoto-3	1-ply: +1.60 2-ply: +1.68	1-ply: +1.10 2-ply: +1.24	1-ply: +0.35 2-ply: +0.62	**Fevga-4**	1-ply: +1.63 2-ply: +1.64	1-ply: +0.35 2-ply: +0.53	1-ply: +0.26 2-ply: +0.48	*
				Fevga-5	1-ply: +1.58 2-ply: +1.59	1-ply: +0.42 2-ply: +0.60	1-ply: +0.32 2-ply: +0.45	1-ply: +0.02 2-ply: +0.14

Table 3.4: Analysis of some of the matches of Fevga-4 and Fevga-5

Match:	Fevga-5 vs Fevga-4		Fevga-4 vs Tavli3D		Fevga-5 vs Tavli3D	
	Fevga-5	**Fevga-4**	**Fevga-4**	**Tavli3D**	**Fevga-5**	**Tavli3D**
Single Wins	47.54%	39.52%	28.93%	2.74%	32.84%	2.86%
Double Wins	4.9%	8.04%	68.32%	0.01%	64.26%	0.04%
Total Wins	52.44%	47.56%	97.25%	2.75%	97%	3%
Final Score	+0.02ppg	-0.02ppg	+1.63ppg	-1.63ppg	+1.54ppg	-1.54ppg

In Fevga, Fevga-4 outperforms both Fevga-2 and Fevga-3 agents, while Fevga-5 outperforms all others except in the Tavli3D benchmark, where it is inferior to Fevga-4 and Fevga-2 (Table 3.3). The explanation of this phenomenon is shown at Table 3.4: Against an inferior opponent, Fevga-4 achieves more points, because it wins more doubles due to its riskier strategy, while Fevga-5's safer strategy of building primes wins the same amount of games overall but fewer doubles. These new results show that the strategy learned by Fevga-4 and Fevga-5 is no different than the one learned from their previous

counterparts (Fevga-2 and Fevga-3); simply the proposed training method learns the strategies better.

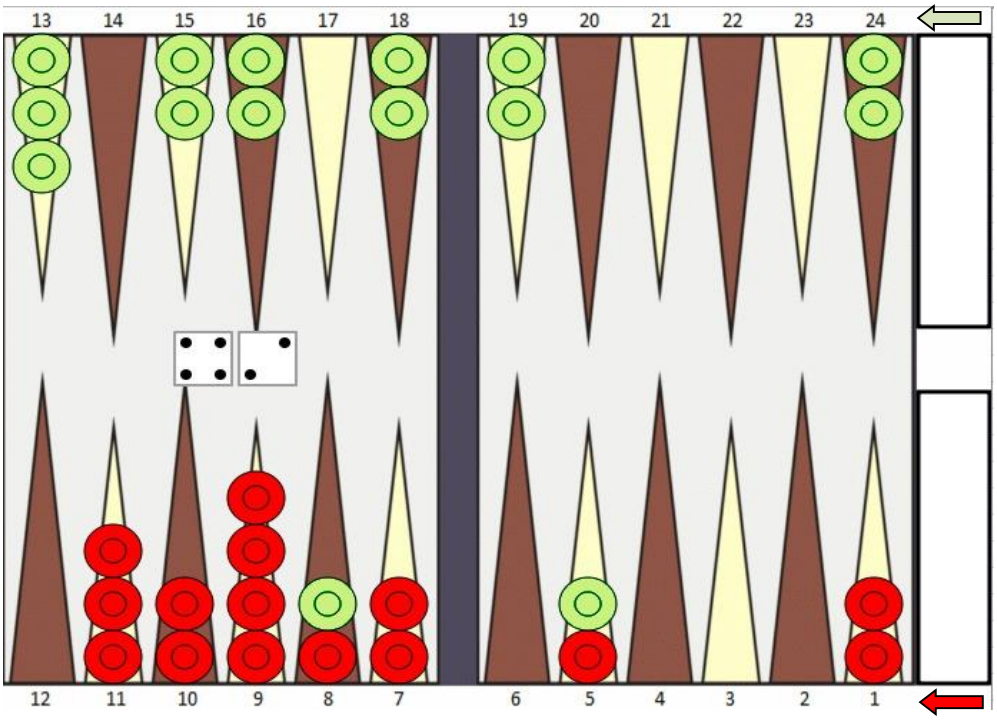


Figure 3.7: Example of a position where agents Plakoto1-3 fail to produce the best move. The green player is to play roll 42. The best move here is 24/18, since the 24-point cannot be pinned by any dice roll. However, Plakoto1-3 agents prefer the clearly inferior move 24/20, 24/22 which gives the opponent a pinning opportunity to get back into the game.

3.2.2.3 Mother point feature selection in the Plakoto variant

An interesting observation was made while testing the strategies that were learned in the Plakoto variant. The resulting strategy was very conservative with regard to its starting point (also called the "mother" point). The agent correctly identified that it must not let the starting point with one checker, as it would be potentially open to a pinning attack that would automatically lose the maximum amount of points (double game). However, it could not discriminate the positions that such an attack could not be carried out by the opponent and protected its first point even after we added the expert feature of pinning probability in Plakoto2 and Plakoto3. This resulted in obvious errors in a small number of positions. For

example, the amount of equity lost for selecting the wrong move in Figure 3.7 was calculated to 0.276ppg by making a 100,000 games rollout on each of the moves in question (Table 3.5).

We suspected that the agent learned the harmful concept of leaving the first point open by the four raw features instead of the added expert feature "pinning probability at point 1". In order to confirm this, we trained another agent (Plakoto-4) without the first of the four features for point 1, leaving only three features, one, if 2 or more checkers are present, another, if 3 or more checkers are present and a last one, if 4 or more checkers are present. The resulting agent confirmed our suspicions, as it managed to learn the concept of "leaving the first point unprotected is bad" in a correct way, without committing the same mistakes of its predecessors. Evaluating Plakoto-4 final performance of 1.5 million trained games against Plakoto-3 in a 10,000 tournament resulted in equal performance. This may mean either that: a) positions of this kind do not appear frequently and when they appear they do not seem to have a significant impact to the result or b) Plakoto-4 simply needs more training for the difference to tell.

Table 3.5: Evaluation and rollout analysis of the two best moves of the position in Figure 3.7. The first four columns show the evaluation of the Plakoto-3 and Plakoto-4 NNs after 1-ply and 2-ply look-ahead. The fifth and sixth column show the equity of the position by making a rollout analysis using Plakoto-3 and Plakoto-4. The last column shows the equity that was lost by selecting the inferior move. The equity loss was calculated on the average of the two rollouts.

Move	Plakoto-3 (1-ply) eval	Plakoto-3 (2-ply) eval	Plakoto-4 (1-ply) eval	Plakoto-4 (2-ply) eval	Rollout Plakoto-3	Rollout Plakoto-4	equity loss
24/22 24/20	1.020	1.048	0.942	1.046	0.983	0.968	0.276
24/18	0.692	1.082	1.140	1.248	1.259	1.243	-

Why was this concept not learned correctly by the other agents, especially when the other points where learned correctly? The concept of protecting the 1st point is one of the first things the agents learn, because it is the closest to the terminal position, the only position that receives reward, and because the random character of the first self-play training games result in many "mother doubles". When confronted with two features to learn the concept, one being a binary input, and one a float input between 0 and 1, the neural

network chooses the first one because it is the easier and the faster to learn. It would appear that the agent would have a chance to "unlearn" this later, as learning progresses, when the estimates of the NN are closer to the optimal. However, this is never done, as these kind of positions rarely appear, because the agent has learned how (wrongly) to defend against.

3.3 Final Learning Setup

After determining the target of the update and some expert features we were able to finalize the training procedure. We also reevaluated the play of the agents and decided to add a few more expert features that will be shown in this section.

Another parameter that needed resolution was how many hidden layers should we have used and what their size would be. Following the successful application of TD-Gammon we used one hidden layer in all our backgammon NNs. The number of hidden neurons is 160 for backgammon, 100 for Fevga and 100 for Plakoto. These numbers were chosen based on preliminary experiments. A higher number of hidden neurons increases performance cost for evaluating each state. This results in increased thinking time for each move, especially when utilizing lookahead in greater depths . Thus, the number of hidden neurons chosen is a compromise between performance and computational cost. 160 hidden neurons were also used by the TD-Gammon program.

Using the above architecture, the procedure of obtaining an estimation of the game-theoretic value of each state is straightforward: set the inputs of the NN according to the board positioning, execute the forward-propagate procedure of the NN to update the outputs, and finally linearly combine the outputs according to the following formula: V= 2 * W - 1 + WD – LD.

3.3.1 Training the NN using TDL

Training a neural network requires training examples in a supervised learning setting. We use TD(λ) algorithm (Sutton, 1988) and the NN's backpropagation algorithm to update the TD error. The exact training procedure is summarized in Algorithm 1. This

training scheme, named *reverse offline recalc*, was selected among several similar self-play methods (Section 3.3.2.1).

Algorithm1. Training a backgammon NN using TD(0)

```
    // nn:  the neural network that we want to train
    //       nn.inputs: a vector representing the input layer
    //  nn.outputs: a vector representing the output layer (W, WD, LD)
    //  nn.target: a vector representing the target of the update
    // states: a vector holding the all the positions of a game
 1. nn.initialize(input layer size, hidden layer size, output layer size = 3, learning rate α)
 2. randomize(nn)      // randomize all weights to [-0.5, 0.5]
 3. while (stopping condition) do
 4.    states = selfplaygame(nn)
 5.    for (t=T to 1 step -1) do
 6.       if(states(t) is terminal)
 7.          nn.targets = reward(states(t))
 8.       else
 9.          nn.inputs = encoding(states(t+1))
10.          nn.forwardpropagate()  // calculate outputs
11.          nn.targets = invert(nn.outputs)
12.       endif
13.       nn.inputs = encoding(states(t))
14.       nn.forwardpropagate()   // calculate outputs
15.       nn.backpropagate()        // apply backpropagation algorithm
16.    end for
17. end while
```

Figure 3.8: Reverse offline recalc algorithm with TD(0)

In the adopted training procedure, the updates are applied (Lines 5-15), after a self-play game (Line 4) is ended, starting from the last position of the game and ending at the first (Line 5). At each time step, we recalculate the target for each update (Lines 9-11), in order to get as much accuracy for the estimation of the example label as possible. The function encoding (Lines 9, 13) encodes the raw and expert features in their predefined positions at the input layer. Note that the value of the next state is inverted (Line 11). This is necessary because the NN plays the game for both sides always as the first player. When all the moves up to the first are updated, the algorithm starts a new self-game producing the moves according to the updated NN. The procedure is repeated, until the selected stopping criterion is satisfied. Possible stopping criteria are: (1) a predefined number of self-

play games is reached or (2) no more performance improvement according to a predefined benchmark is found after a prespecified number of self-play games.

Algorithm 1 uses TD(λ) with λ=0, that is the current state is updated only according to the estimation of the next state (Lines 9-11). Thus, the target of the update is $V_{target}(s_t) = V(s_{t+1})$. If we want the target of the update to be based on more than one future move estimates, we can use the forward view of TD(λ) (Section 2.2.4.3.3), where ($0<\lambda\leq 1$), and the target of the update becomes

$$V_{target}(s_t) = (1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1}V(s_{t+n}) + \lambda^{T-t-1}V(s_T)$$

In case of λ>0, lines 8-10 of Algorithm 1 must be changed accordingly. Similarly to $V(s_{t+1})$, all values $V(s_{t+n})$ for *n* being any odd number must be inverted.

The updates of the network weights are done incrementally and not in a batch setting. This procedure is similar to stochastic or "online" training (Wilson & Martinez, 2003). The main difference is that there are no fixed labels in the training examples; the labels are given by TD(λ). We prefer incremental training, because it has been shown to perform at least equally to the standard batch training using fewer computational resources (Wilson & Martinez, 2003).

3.3.2 CHOOSING LEARNING RATE A AND PARAMETER Λ

One of the advantages of incremental training is that one can use a larger learning rate than in a batch setting. We also made some experiments with different values of λ with mixed results. In the Plakoto variant, values of λ>0.6 resulted in divergence, whereas lower values sometimes became unstable. So it was decided to keep λ=0 for this variant. For Portes and Fevga variants it was possible to increase the λ value without problems and this always resulted in faster learning, but unlike other reported results (Wiering, 2010) final performance did not exceed experiments with λ=0.

Table 3.6: Selected values of α and λ parameters.

Games Trained	Portes	Plakoto	Fevga
0-10,000	λ=0.7 α=1	λ=0 α=0.3	λ=0.7 α=1
10,000-100,000	λ=0.7 α=0.3	λ=0 α=0.3	λ=0.7 α=0.3
100,000-250,000	λ=0.7 α=0.1	λ=0 α=0.1	λ=0.7 α=0.1
250,000-500,000	λ=0 α=0.3	λ=0 α=0.1	λ=0 α=0.3
500,000-1,500,000	λ=0 α=0.1	λ=0 α=0.1	λ=0 α=0.1
1,500,000-5,000,000	λ=0 α=0.1	λ=0 α=0.01	λ=0 α=0.01
5,000,000-	λ=0 α=0.01	-	-

Previous experiments were conducted with constant λ and α=0.1. Following the above preliminary experiments we use a decreasing value for λ and α for the experiments in this paper (with the exception of Plakoto where λ is kept constant to zero). Starting with high values of λ=0.7 and α=1 we gradually decrease these values, when performance starts to flatten. The exact values of these parameters are shown in Table 3.6. Using this setup the performance of Plakoto and Fevga variants maxes out at 5 million games and Portes at around 15 million games.

3.3.3 EXPERT FEATURES

The features included in the input layer of each NN are divided to "raw" and "expert" features. Raw features represent the placement of each checker on the board, while expert features are important game concepts that would otherwise be very difficult for the NN to infer from the raw encoding alone. The raw features of Plakoto and Fevga are presented in 3.2.1, while the raw features of our Portes NN are exactly the same as used in (Tesauro, 1992). The remaining of this section presents the selected expert features for the Portes game as well as the new expert features that we used in Plakoto and Fevga. The remaining expert features of Plakoto and Fevga are described in 3.2.1.

3.3.3.1 Expert features for Portes/Backgammon

All the expert features of our Portes/Backgammon bot are shown in Table 3.7. The features capture important game playing concepts according to the current literature from expert backgammon players. For example EnterFromBar_1 and EnterFromBar_2 capture the concept of home board strength. This feature, however, is useless, when the position

has no contact (race feature). The NN takes care of combining the features in the correct way taking the current position into account. Additionally, the hidden neurons can create features not existent in the expert list, if necessary. For example, we found that the prime formation (six consecutive made points) was handled correctly by the program, so we did not include it in the list of expert features, even if it is an important concept. The features PipDiff_1, PipDiff_2, PipBearoff_1, PipBearoff_2 were normalized to the [0, 1] interval by a dividing with 60.

Table 3.7: Expert features for the Portes/backgammon variant.

Feature name	Description
HitProb_1	Probability of one player checker being hit on the next roll
HitProb_2	Probability of two player checkers being hit on the next roll
Race	Boolean feature showing the position is a no contact position
PipDiff_1	Pipcount difference when the player is behind (when ahead = 0)
PipDiff_2	Pipcount difference when the player is ahead (when behind = 0)
PipBearoff_1	Pipcount to bearoff for player on roll
PipBearoff_2	Pipcount to bearoff for opponent
EnterFromBar_1	Probability of player entering from bar
EnterFromBar_2	Probability of opponent entering from bar
OppContain_1	Probability of opponent's last checker escaping from player's home board
OppContain_2	Probability of opponent's second to last checker escaping from player's home board
UsContain_1	Probability of player's last checker escaping from opponent's home board
UsContain_2	Probability of player's second to last checker escaping from opponent's home board

3.3.3.2 New expert features for Plakoto

After manual examination and with the help of comments from users that downloaded *Palamedes*, we identified two key problems of our Plakoto bots. The first one presented itself in positions, when the bot has pinned the opponent inside the bot's home board. In such positions it is advisable for the bot to "stack" its checkers in the pinned

point, whenever possible, so as to prolong the duration of the pin even in the bearoff situation. Such a strategy most often leads to a double game. However, our bots were positioning their checkers, as if it was a normal bearoff, greatly reducing their chances of a double game. This problem was addressed by adding the ChFrontOfPin_1 and the ChFrontOfPin_2 features. These two features were scaled to [0, 1] interval by dividing each by 14. We also added the Esc_Prob1 and EscProb2 features hoping that the bot can advance its made points more fluidly, not leaving behind made points that cannot escape easily. Finally we added five features from Portes that are relevant to Plakoto as well. The complete set of features is shown in Table 3.8.

Table 3.8: Expert features for the Plakoto variant.

Feature name	Description
Race	Boolean feature showing if the position is a no contact position
PipDiff_1	Pipcount difference when the player is behind (when ahead = 0)
PipDiff_2	Pipcount difference when the player is ahead (when behind = 0)
PipBearoff_1	Pipcount to bearoff for player on roll
PipBearoff_2	Pipcount to bearoff for opponent
ChFrontOfPin_1	Number of player checkers in front of last pin when the player has the opponent pinned in the player's homeboard
ChFrontOfPin_2	Number of opponent checkers in front of last pin when the opponent has the player pinned in the opponent's homeboard
Esc_Prob1	Escape probability of player's last made point
Esc_Prob2	Escape probability of opponent's last made point

3.3.3.3 New expert features for Fevga

The most important concept in the Fevga variant is the existence of a prime formation. In previous work we addressed this by adding one binary feature for every type of prime, when it was encountered in the game. While this resulted in the desired effect of the NN learning the concept of making primes when necessary, it did not always understand when it was important to prevent the opponent from making primes of its own. The bot could not understand by this feature alone when the opponent was close to making a prime so as to take immediate measures to disrupt his plan. The inclusion of 2-ply look-ahead

improved the situation, as now the bot had access to the next moves of the opponent, but it would be desirable to have this knowledge without reverting to the computational expensive procedure of looking ahead at greater depths.

To address this problem we changed the binary features of making primes in the following way: When a prime is made, the feature is set to one as before. When there is no prime present, instead of setting the feature to zero, we replaced it with a heuristic that computes the probability of making the prime. This was done both for the primes of the bot as well as for the primes of the opponent. Computing accurately this heuristic is very complex and takes much time especially for middle game positions. In order to keep the computational requirements low, we compute the heuristic only for the most common scenario: when there is only one checker left to make the prime. Positions where the prime needs two or more checkers to be achieved are less frequent and usually have smaller probability of success. Thus, the resulting heuristic is a compromise between accuracy and executing time.

These updated features resemble the way we added the pinning probabilities in the Plakoto variant as shown in 3.2.1.2. It has the advantage of putting knowledge in the NN while at the same time keeping low the size of the inputs. We also added the features PipDiff_1, PipDiff_2, PipBearoff_1, PipBearoff_2 of Portes and Plakoto, because they are relevant to Fevga as well.

We also experimented by combining the above new features with the intermediate reward procedure during the training of Fevga3 and Fevga5 bots (3.2.1.1). Such a procedure results in a strategy that tries to build primes and maintain them at all cost. While the resulting performance was higher than previous bots, it was lower than Fevga6, i.e. without the intermediate reward. One possible explanation is that without the intermediate reward the bot can identify situations where a prime is not the best course of action. It seems that finding exceptions to the rule of building primes even with an incomplete heuristic is more fruitful than a “dogmatic” behavior regarding primes.

3.3.4 EXPERIMENTAL RESULTS

Being consistent with our previous naming scheme, we name the new bots Plakoto-5 and Fevga-6. We compare them by taking the best set of trained weights and make them playing a tournament against a benchmark opponent without look-ahead (1-ply). For Plakoto and Fevga this benchmark is our best previous bot, namely Plakoto-4 and Fevga-4 respectively. For the Portes/Backgammon we chose the pubeval benchmark, because we can indirectly compare the performance with others backgammon bots that published results against it. We also report on the performance when applying a simple look-ahead procedure using the expectimax algorithm (Michie, 1966) at 2-ply depth. The bot is awarded a +1 point for a single win, +2 points for a double win, -1 for a single loss, -2 for a double loss. The result of the tested games sum up to the form of estimated *points per game* (ppg) and is calculated as the mean of the points won and lost. The number of games played are 100,000 for 1-ply and 10,000 for 2-ply. In order to speed up the testing time of 2-ply, the expansion of depth-2 was performed only for the best 15 candidate moves (forward pruning). Table 3.9 presents the results.

The performance of the Portes/Backgammon bot is comparable to most top playing bots. TD-Gammon 2.1 reported a 0.596 performance against pubeval (Tesauro, 2011), while another backgammon program, GNUBG (Gnubg.org, 2015), frequent participant to backgammon Computer Olympiads, recently reported similar performance (0.6046 ppg) in its mailing list, while using a more complex training scheme and three different NNs for three different stages of the game (Gnubg mailing list, 2012).

Table 3.9: Performance of the new bots against benchmark opponents

Bot	Opponent	ppg
Portes-1(1-ply)	Pubeval (1-ply)	0.603
Plakoto-5(1-ply)	Plakoto-4(1-ply)	0.356
Plakoto-5(2-ply)	Plakoto-4(1-ply)	0.422
Fevga-6(1-ply)	Fevga-4(1-ply)	0.215
Fevga-6(2-ply)	Fevga-4(1-ply)	0.323

Since the training procedure and the NN architecture is the same for the old and new bots for the Fevga and Plakoto variants, it is safe to assume that the gain was due to the addition/alteration of the expert features. We believe that the common features of Portes that were added to Plakoto and Fevga played a minor role to the improved performance. More important for Fevga was the alteration of the prime features and for Plakoto the addition ChFrontOfPin_1 andChFrontOfPin_2.

Chapter 4

CHAPTER 4: OPENING STATISTICS AND MATCH PLAY

4.1 Introduction

In this chapter we use our expert playing agents of Palamedes to make the first ever computer assisted analysis of the opening rolls for the backgammon variants Portes, Plakoto and Fevga (collectively called Tavli in Greece). We then use these results to build effective match strategies for each game variant.

Our methodology is similar to the one used in (Keith, 2006): After the opening roll and for each roll, the most promising continuations are analyzed by means of rollout analysis, a Monte Carlo method that is commonly used in backgammon. The rollouts start from the resulting position after each candidate move and a fixed number of games is played, until a terminal position is reached. Counting the results of these games we can finally get the probabilities of single wins (WS), double wins (WD), single losses (LS) and double losses (LD). Based on these probabilities, we can then compute the estimated equity of each position using the following equation:

$$E = WS - LS + 2 * (WD - LD)$$

This kind of evaluation is considered to offer accurate results in standard backgammon, despite the fact that the move selection algorithm of the rollout phase is not so strong in terms of performance (Tesauro, 2002). Rollouts can also be truncated, which means that they could stop after a fixed amount of plies (instead of going till the end of the game) and average together the estimates of the resulting positions, with a negligible change in their estimates. In the presence of an endgame database that can offer the exact equity of endgame positions (e.g. a 2-sided endgame database), a rollout can go as far as the first position encountered in the endgame database and return the database value.

4.2 Experimental setup and results

We used our latest and best Neural Networks (NN) game evaluation functions for selecting each move on the rollouts. For Portes we used Portes_ACG13 NN, for Plakoto we used Plakoto5 and for Fevga we used Fevga6 (Section 3.3.4). The rollouts were performed using 1-ply playing mode, which means that Palamedes looked ahead only at the

current roll for each play during the rollouts, selecting the best play of each trial. After the opening roll, we rolled out the five most promising candidate moves (selected using 2-ply evaluation), using 100,000 games per position. The standard error of the estimated equity E when performing this number of trials is less than 0.02. Rollouts were performed using **cubeless untruncated money play**. *Cubeless* means that games are played without a doubling cube. *Untruncated* means that rollouts were played out until the end of the game. *Money play* means that each game is played individually and not as a part of a match.

Table 4.1: Best move of all opening rolls per variant examined

	PORTES		**PLAKOTO**		**FEVGA**	
ROLL	**BEST MOVE**	**EQ**	**BEST MOVE**	**EQ**	**BEST MOVE**	**EQ**
SINGLE ROLLS						
21	24/23 13/11	0.006	24/22 24/23	0.042	24/21	-0.030
31	8/5 6/5	0.155	24/21 24/23	0.037	24/20	0.012
41	24/23 13/9	0.002	24/20 24/23	0.070	24/19	0.086
51	24/23 13/8	0.011	24/19 24/23	0.043	24/18	0.090
61	13/7 8/7	0.108	24/18 24/23	0.097	24/17	0.194
32	24/21 13/11	0.017	24/21 24/22	0.050	24/19	0.086
42	8/4 6/4	0.110	24/20 24/22	0.065	24/18	0.090
52	24/22 13/8	0.015	24/19 24/22	0.066	24/17	0.194
62	24/18 13/11	0.017	24/18 24/22	0.106	24/16	0.259
43	24/20 13/10	0.015	24/20 24/21	0.056	24/17	0.194
53	8/3 6/3	0.059	24/19 24/21	0.039	24/16	0.259
63	24/18 13/10	0.018	24/18 24/21	0.096	24/15	0.336
54	24/20 13/8	0.029	24/19 24/20	0.073	24/15	0.336
64	8/2 6/2	0.016	24/18 24/20	0.121	24/14	0.385
65	24/18 18/13	0.072	24/18 24/19	0.117	24/13	0.440
DOUBLE ROLLS						
11	8/7 (2) 6/5(2)	0.213	24/23 (4)	0.129	24/20	0.012
22	13/11(2) 6/4(2)	0.240	24/20 24/22 (2)	0.137	24/16	0.259
33	8/5 (2) 6/3 (2)	0.259	24/18 24/21 (2)	0.187	24/15	0.336
44	24/20(2) 13/9(2)	0.348	24/16 (2)	0.247	24/16	0.259
55	13/8 (2) 8/3 (2)	0.160	24/14 24/19 (2)	0.361	24/9 24/19	0.831
66	24/18(2) 13/7(2)	0.398	24/12 (2)	0.521	24/18	0.090

Opening rolls were split in two groups, single and double, in order to shed more light into the effect of rolling a double at the start of the game. This is most useful in standard backgammon, which does not allow a double opening roll like the Portes variant does.

The move selected for each roll was picked as the best after rolling out the most promising candidate moves available. These figures were constructed by singling out the move with best equity after each roll. The actual moves selected can be seen in Table 4.1.

Figures 4.1 – 4.3 summarize the results for each roll and game variant and compares the games. All numbers shown are with regard to the first player making the move. Averages of all single rolls are marked with the word 'SINGLE'. Averages of all double rolls are marked with the word 'DOUBLE'. Finally the word 'ALL' is the weighted (according to the probability of each roll) average of all 21 rolls.

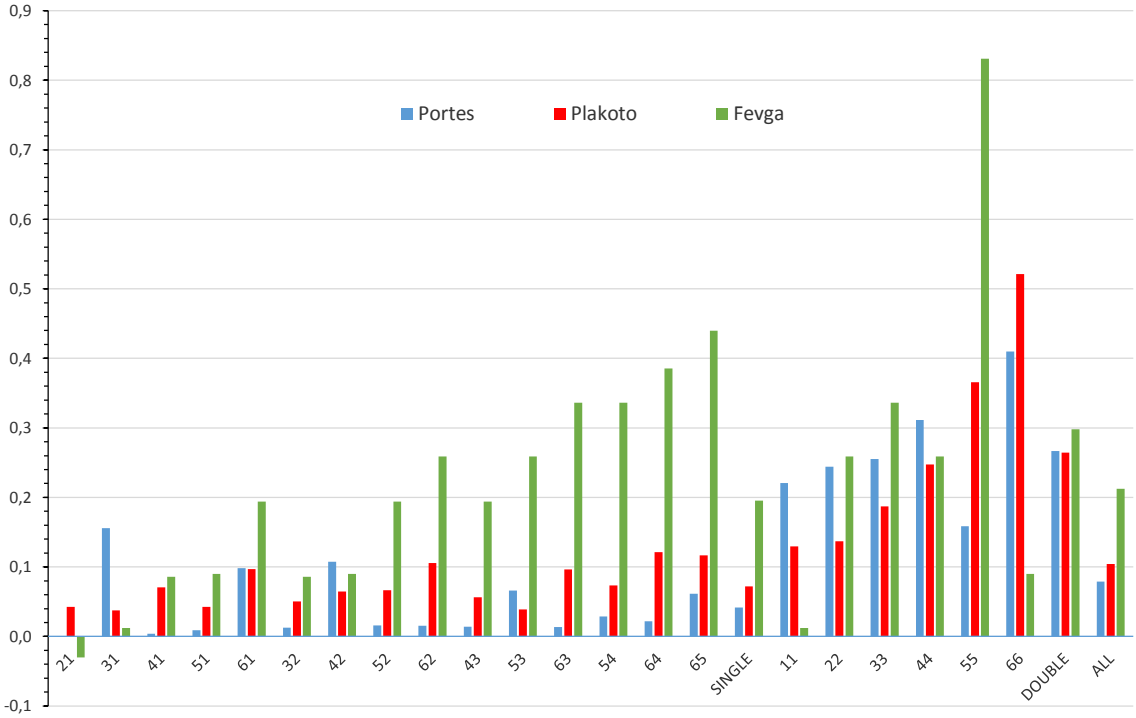


Figure 4.1: Comparison of estimated equity of all opening rolls

In Figure 4.1 the estimated equity of all opening moves for all games is presented. The starting roll with the greatest equity is by far the 55 in Fevga, while the least useful roll is the 21 in Fevga.

Figure 4.2 summarizes the outcome of all rolls to produce the expected result of the first player. From this figure we can derive the percentage of games that result in doubles, also called "gammon rate", by adding WD and LD (Table 4.2).

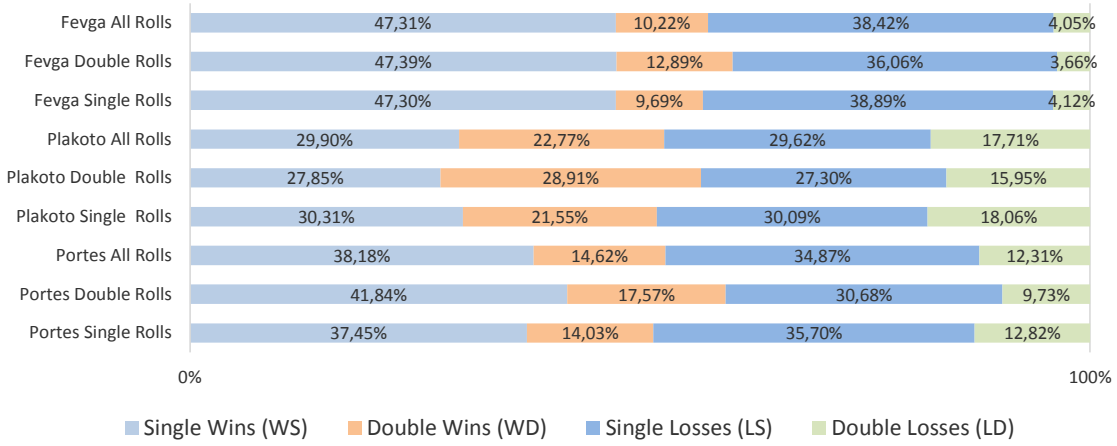


Figure 4.2: Expected outcome (%) of the first player

Table 4.2: Gammon rates of Tavli variants

Variant	Gammon Rate
Portes	26.85%
Plakoto	40.48%
Fevga	14.27%

Perhaps the most interesting result of this study is the total estimated equity of the first player shown in Figure 4.3. Ideally, a perfectly designed backgammon game would give zero equity to the first player. This would mean that the opening roll does not favor one player over the other. Our study shows that the "best" variant in that regard is Portes, closely followed by Plakoto. On the other hand, Fevga gives a significant advantage to the first player.

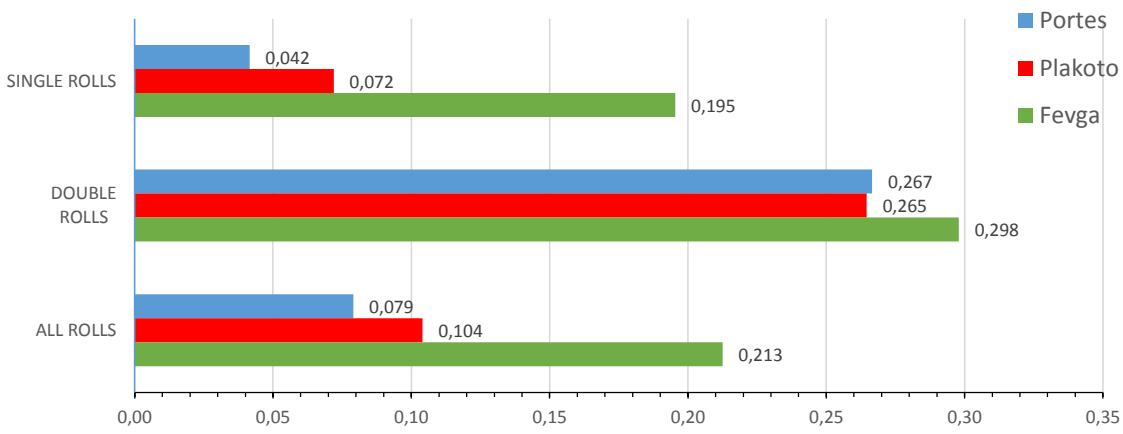


Figure 4.3: Total estimated equity of the first player

4.3 Discussion

This section discusses and compares the results of the three games to each other, as well as to previous similar studies. We also attempt to explain some of the results found from a strategic point of view.

4.3.1 PORTES

The results for the single rolls of the Portes variant are very similar to a previous study on standard backgammon openings (Keith, 2006). In that study, the rollouts were performed by GnuBG (Gnubg.org, 2015), a very strong open source backgammon program at a 2-ply depth. The estimated equity of all single rolls in (Keith, 2006) is 0.039, ours is 0.042. Almost all best opening moves coincide with our best selected moves. The gammon rate is estimated in (Keith, 2006) at 27.6%. If we count the backgammons, which according to Portes rules are counted as gammons, this rate is increased to 28.8%. Our results estimate this at a more modest 26.9%, almost a 2% difference. We give two possible explanations for this behavior: a) 1-ply rollouts are not accurate enough and b) the playing style of Palamedes is more conservative compared to that of GnuBG, resulting in somewhat fewer gammons.

Since the analysis of the single opening rolls is nothing new, we concentrate the discussion on the effect of the double rolls. The inclusion of doubles in the opening roll

gives more advantage to the first player. The average equity of all double rolls is 0.267 (Figure 4.3), six times larger than the equity of the single rolls. This was expected, since: a) doubles usually result in more distance travelled than the average single roll and b) even small doubles like 11 give the opportunity to construct strategically made points without risking getting hit by the opponent. The best double roll is 66 with 0.41 equity; even the worst double roll (11, E=0.22) is better than the best single roll (31, E=0.16). The effect of doubles can be seen in the weighted average of all rolls (Figure 4.3, E=0.079), which is almost twice that of the single rolls.

4.3.2 PLAKOTO

Plakoto results, compared to the other games, demonstrate an increased gammon rate. 41% of Plakoto games are won as doubles, 14% more than the rate we calculated in Portes. This rate can be explained by the strategic strength of pinning an opponent checker inside his home board. It is well known that this kind of pin can result in double games, because, if the pinning player manages not to get pinned himself, he can place his checkers in such a way that during bear off most of his pieces will be borne off, before the pinning checker is unpinned. This places the pinned player at a great disadvantage, because usually he does not have enough time to return the last checker to his home board and avoid the double loss. Of course, one can play a very conservative game and avoid leaving lone checkers in his home board at all costs. However, this can lead to other problems: building large stacks of checkers, that are extremely inflexible and also minimize the chances of hitting lone checkers of the opponent. For this reason, Palamedes and most expert players prefer a “restrained aggressive approach” during the opening, leaving some lone checkers open, when there is a small chance that the opponent can pin them. This strategy, nevertheless, inevitably falls victim to a lucky pinning roll by the opponent, which may be enough to result in a double loss. This reasoning strongly suggests that the starting position of Plakoto greatly influences the gammon rate and the equity of the first player.

In order to test the hypothesis above, we made another experiment changing the starting position: Instead of having all 15 checkers at the starting point, the checkers are

distributed evenly in the first three points. This variant is known in some regions as the *Tapa* variant (Section 2.1.3.1) and we will use this naming also in this paper. The starting position of Tapa makes the pinning of checkers inside the home board during the opening more difficult, because the players can construct made points more easily during the start of the game. We used the same methodology and the same Neural Network (Plakoto-5) for the rollouts. Even if this NN was not trained for this specific variant, we believe that it is sufficient to produce strong play, because the type of positions resulting from a Tapa game are well within the range of positions the NN has seen during self-play training[1].

Table 4.3: Comparison of Tapa and Plakoto estimated results for the first player

Vari-ant	WS (%)	WD (%)	LS (%)	LD (%)	EQUITY	GAMMON RATE (%)
Plakoto	29.89	22.77	29.62	17.71	0.104	40.48
Tapa	37.40	13.12	37.91	11.55	0.026	24.67

The results of the Tapa experiment (Table 4.3) confirm our hypothesis. The gammon rate is reduced from 40.48 to 24.67%. Also the equity of the first player is reduced to 0.026, which is even lower than the equity of the single rolls in the Portes variant (Figure 4.3).

Another notable point that can be seen in Table 4.3 is that the first player wins about the same amount of single games as the second player (29.9% vs 29.6%). Consequently all the advantage that the first player has can be attributed to the difference in double games won, which is 22.77% compared to 17.71% of the second player.

4.3.3 FEVGA

The first interesting result in the Fevga experiments is that the expected equity of the first player (0.213) is the highest amongst all games examined, more than twice that of

[1] The opposite situation could be problematic: a Tapa trained NN may not evaluate correctly Plakoto's opening positions with early home board pins in points 2 and 3, because this kind of experience would have been extremely rare in its self-play training.

the Plakoto (0.104). Winning 57.5% of the games gives the player who plays first a distinctive advantage. Fevga also has the roll with the most gained equity in all games, the 55 roll at 0.84 equity. We also observe that all high sum rolls (e.g. 63, 64, 65) give very high equity for the first player, with 65 (E=0.44), even surpassing the best Portes roll (66, E=0.41). However, unlike the two other variants, doubles do not increase the equity of the single rolls that much (from 0.19 to 0.21). This can be attributed to the fact that apart form 55, the other two large doubles (44 and 66), that typically have increased equity, have a reduced effect because of Fevga's starting rule (Section 2.1.4). Overall, we note that the further the starting checker is able to move during the first roll, the better the chances are for the first player. This observation fully justifies the name of the game ('Fevga' means 'run' in Greek).

Another surprising observation is that the gammon rate (14.27%) is very low compared to the other variants. The greatest factor that affects this statistic is the very small chance of the second player winning a double game. With 4.05% the second player wins less than half doubles that the first player does (10.22%).

4.4 Match Play

In this section we show how we can use the statistics from the previous sections to construct effective match strategies for Tavli variants. When playing a match, the goal of the players is to win the match and not to maximize their expected reward at the individual games. For this reason all strong backgammon programs select the best move by approximating the Match Winning Chance (MWC) at each move selection. We present a simple method, similar to the one used in backgammon, for approximating MWC, using the estimates of the NN evaluations and the gammon rate computed in Table 4.2. For simplicity, we examine only matches of the same game type where the player that starts each game is determined randomly.

First, we build a table estimating MWC before the start of the game for all possible score differences during the course of the match. In the most simple case, that is, when the

score is tied, the players have the same chance of winning the match. The table is calculated using the following recursive definition:

*mwc(A,B) = S * mwc(A-1, B) + D * mwc(A-2, B) + S * mwc(A, B-1) + D * mwc(A, B-2)*

where A is the remaining points left for player A to win the match, B is the remaining points left for player B to win the match, mwc(A,B) is the table entry specifying the probability of winning the match for the A player when the current score is A points away – B points away, S is the probability each player has of winning a single game (= (1 - gammon rate) / 2), D is the probability each player has of winning a double game (=gammon rate / 2). Tables 4.4, 4.5, 4.6 show the tables computed with this method for the games Portes, Plakoto, Fevga respectively and match away scores up to 9.

Table 4.4: MWC (%) for player A on Portes variant

A away		**MATCH WINNING CHANCE (MWC)**								
	B away	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
1		50.00	68.28	81.68	89.04	93.53	96.16	97.73	98.65	99.20
2		31.73	50.00	65.85	76.78	84.56	89.83	93.37	95.72	97.25
3		18.32	34.15	50.00	62.91	73.20	80.98	86.72	90.84	93.75
4		10.96	23.22	37.09	50.00	61.39	70.85	78.41	84.26	88.69
5		6.47	15.44	26.80	38.61	50.00	60.25	69.07	76.36	82.23
6		3.84	10.17	19.02	29.15	39.75	50.00	59.41	67.68	74.71
7		2.27	6.63	13.28	21.59	30.93	40.59	50.00	58.74	66.56
8		1.35	4.28	9.16	15.74	23.64	32.32	41.26	50.00	58.20
9		0.80	2.75	6.25	11.31	17.77	25.29	33.44	41.80	50.00

Table 4.5: MWC (%) for player A on Plakoto variant

A away		**MATCH WINNING CHANCE (MWC)**								
	B away	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
1		50.00	64.88	79.43	86.77	91.90	94.91	96.85	98.03	98.78
2		35.12	50.00	65.87	75.78	83.47	88.67	92.34	94.84	96.55
3		20.57	34.13	50.00	61.90	71.98	79.55	85.33	89.56	92.65
4		13.23	24.22	38.10	50.00	60.91	69.87	77.20	82.97	87.43
5		8.10	16.53	28.02	39.09	50.00	59.69	68.13	75.17	80.93
6		5.09	11.33	20.45	30.13	40.31	50.00	58.94	66.82	73.60
7		3.15	7.66	14.67	22.80	31.87	41.06	50.00	58.29	65.75
8		1.97	5.16	10.44	17.03	24.83	33.18	41.71	50.00	57.79
9		1.22	3.45	7.35	12.57	19.07	26.40	34.25	42.21	50.00

Table 4.6: MWC (%) for player A on Fevga variant

A away		**MATCH WINNING CHANCE (MWC)**								
	B away	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
1		50.00	71.43	84.18	91.18	95.09	97.27	98.48	99.15	99.53
2		28.57	50.00	66.69	78.37	86.25	91.39	94.68	96.74	98.02
3		15.82	33.31	50.00	63.91	74.72	82.70	88.39	92.33	95.00
4		8.82	21.63	36.09	50.00	62.19	72.20	80.03	85.93	90.26
5		4.91	13.75	25.28	37.81	50.00	60.98	70.32	77.92	83.88
6		2.73	8.61	17.30	27.80	39.02	50.00	60.07	68.85	76.19
7		1.52	5.32	11.61	19.97	29.68	39.93	50.00	59.35	67.65
8		0.85	3.26	7.67	14.07	22.08	31.15	40.65	50.00	58.77
9		0.47	1.98	5.00	9.74	16.12	23.81	32.35	41.23	50.00

Finally, for move selection, a similar equation is used for determining the MWC of each move:

*MWC = WS * mwc(A-1, B) + WD * mwc(A-2, B) + LS * mwc(A, B-1) + LD * mwc(A, B-2),*

where WS, WD, LS and LD are the output estimations of our neural network evaluation function.

4.4.1 Experiments in Match Play

In order to test the above method, we made an experiment playing 10,000 5-point matches in the three variants examined, where one player uses the "match" strategy and the other player uses the "money play" strategy that tries to maximize the value of each individual game. The match started half the time by the "match" player and the other half by the "money" player. The results along with some useful statistics that we stored during the course of the matches are shown in Table 4.7. All results are from the point of the match player.

Table 4.7: Performance of match strategy vs money play strategy in 10000 5-point matches

Variant	Match Wins	Diff. moves	Games WS	Games WD	Games LS	Games LD	Total game points
Portes	5144±98	7.1%	22937	7094	19558	9066	-565
Plakoto	5103±98	4.6%	15994	10627	15238	11007	-4
Fevga	5067±98	5.3%	28395	4453	27358	5401	-635

The performance of the match strategy is better than the money-play strategy in all games, in terms of matches won by the match player, although the total points won by the match player are less than the points won by the money player. In other words, the match player is able to win the points, when they are more important, in order to win the current match. This observation is clearer in Portes and Plakoto and less significant in Fevga, due to the low gammon rate of Fevga that does not give many opportunities for the players to take justified risks for a gammon. We also kept counters whether the money player would play the same move with the match strategy in a non trivial decision (number of possible moves > 1) when it was the turn of the match player (column Diff. moves). As it can be seen in this column, the two strategies differ very slightly and this can be an explanation why the match strategy is only better by a small margin.

Finally, we also measured the result of each game (columns: WS, WD, LS, LD) and the total game points from the point of the match player. Interestingly, the match player wins more single games and less double games in all three variants. This can be explained with the following reasoning: when the match player is ahead on the score, it will play more conservatively trying to keep its lead and not take unnecessary chances to win a gammon that could give also winning chances to the opponent. On the other hand, when he is behind, he will go more aggressively for a gammon in order to try to close the gap, before it is too late. This risky strategy will be sometimes successful, but most of the times it will result in gammons for the opponent.

4.5 Conclusions and future work

In this chapter we used Palamedes bot to conduct rollout experiments on the opening moves of the first player for three popular backgammon games: Portes, Plakoto and Fevga. Our findings for Portes without the double rolls are very close to those found in the literature. To the best of our knowledge, this is the first time that an analysis of the opening moves was conducted for the other two variants, Plakoto and Fevga.

Our results show that the advantage of the first player is significant in the Fevga variant, small in Plakoto and very small in Portes. The superiority of the Portes variant in this statistic was expected, because Portes (and backgammon) has the advantage of a specially crafted starting position, which is not present in the other variants. Another interesting result is that the gammon rates of the three games fall in completely different ranges. The smallest gammon rate is for the Fevga variant (14.27%), followed by Portes/Backgammon (26.9%), whereas Plakoto has the largest rate (at 41%).

We also showed the effect of the starting position on the statistics examined in the Plakoto variant. Changing the starting position (Tapa variant) only slightly, we managed to lower the gammon rate and the advantage of the first player significantly, making Tapa the most “fair” backgammon variant examined so far. It would be interesting to try the opposite procedure in the backgammon/Portes variant: what would be the gammon rate and equity of a variant with the same rules as backgammon but a starting position, where

all starting checkers are placed in the player's first point? If the results of our Plakoto/Tapa experiments are any indication, we suspect that we would see an increase in both of these measurements. We could have tried out an experiment using the Portes NN in this variant. However, unlike the Plakoto/Tapa case, here the change of the starting position is significant, so we feel that the Portes NN will not generalize well. A new NN-based evaluation function should be self-trained, but as this is not trivial, we leave it for future work.

Finally, as a practical application, we used the computed gammon rates to construct a match strategy that outperformed our previous money play strategy when playing 5-point matches in Portes and Plakoto. In the future we plan to extend this method in matches, where the starting player of the game is the one that wins the previous game, and in matches that consist of different game types like a Tavli match.

Chapter 5

CHAPTER 5: CONSTRUCTING PIN ENDGAME DATABASES FOR PLAKOTO

Computer game programs have been using endgame databases to great effect, especially in board games. Examples of complex games benefiting from such databases are chess (Nalimov, Haworth & Heinz, 2000), Chinese chess (Fang, Hhu & Hhu, 2002), checkers (Schaeffer et al., 2003), awari (Romein,& Bal, 2003), Kriegspiel (Ciancarini & Favini, 2010) and nine-men morris (Gasser, 1996), to name a few. Moreover, endgame databases are catalytic in every attempt to solve a game, as it can be seen in solved games like checkers (Schaeffer et al., 2007), nine-men morris (Gasser, 1996) and more recently heads-up limit texas holdem poker (Bowling et al., 2015).

An endgame database usually contains precomputed game-theoretical values (or near perfect heuristics) for each position record. The game playing program can use this database by searching the records, when an endgame position contained in the database is reached by the AI search. The benefits for the program are multiple: firstly, the value retrieved from the database is more accurate than the program's evaluation function; secondly, the retrieval of the database value is typically faster than the evaluation function execution speed; thirdly, there is no need to search any further down the tree.

The endgame databases can also provide a powerful analytical tool for game professionals and for understanding the game in general. A prominent example is chess, where positions which humans had analyzed as draws were proven winnable and vice-versa. Also the database constructed when heads-up limit texas holdem poker was solved (Bowling et al., 2015) offered insights that contradicted some human beliefs about the best play in this game.

Backgammon programs also make use of endgame databases. These usually cover the positions where both players have their checkers in the bearoff quadrant (also known as *bearoff* databases). In the two-sided version, these databases offer the game-theoretic value of the position, whereas in the one-sided version, the goal is to minimize the average number of rolls to bearoff, so the values stored represent a distribution of the expected

number of rolls to bearoff. The one-sided version is much smaller than the two-sided one, but it is not as accurate with respect to finding the best move.

The aforementioned bearoff endgame databases can be used in many of the variants in which we are interested in (Portes, Plakoto, Fevga, Narde), since the bearoff positions of backgammon can occur in all of these games as well. For this purpose, Palamedes already contains a two-sided bearoff database that was constructed using similar techniques used by other programs. This database gives the game theoretical value of all bearoff positions when the doubling cube is not used and is 5.48GB in size.

This chapter describes our efforts of our first attempt to construct endgame databases for positions only seen in the Plakoto variant. To the best of our knowledge, this is the first time this kind of endgame database is constructed. We believe this is the first step towards constructing bigger and better endgame databases for the game of Plakoto in the future.

5.1 Endgames with pins

Strategically thinking, pinning is the most important characteristic of the Plakoto variant for the following reasons:

- A “made point” can be constructed with only one checker instead of the usual two, which makes primes and other formations easier.
- Players can nullify bad luck, when they roll small rolls and/or the opponent rolls big rolls. This is true because running to the bearoff phase is unimportant, when one or more checkers are trapped.
- The side that has pinned without getting pinned usually gets a few rolls ahead in the bearoff race. The further ahead the pin is, the bigger the advantage.

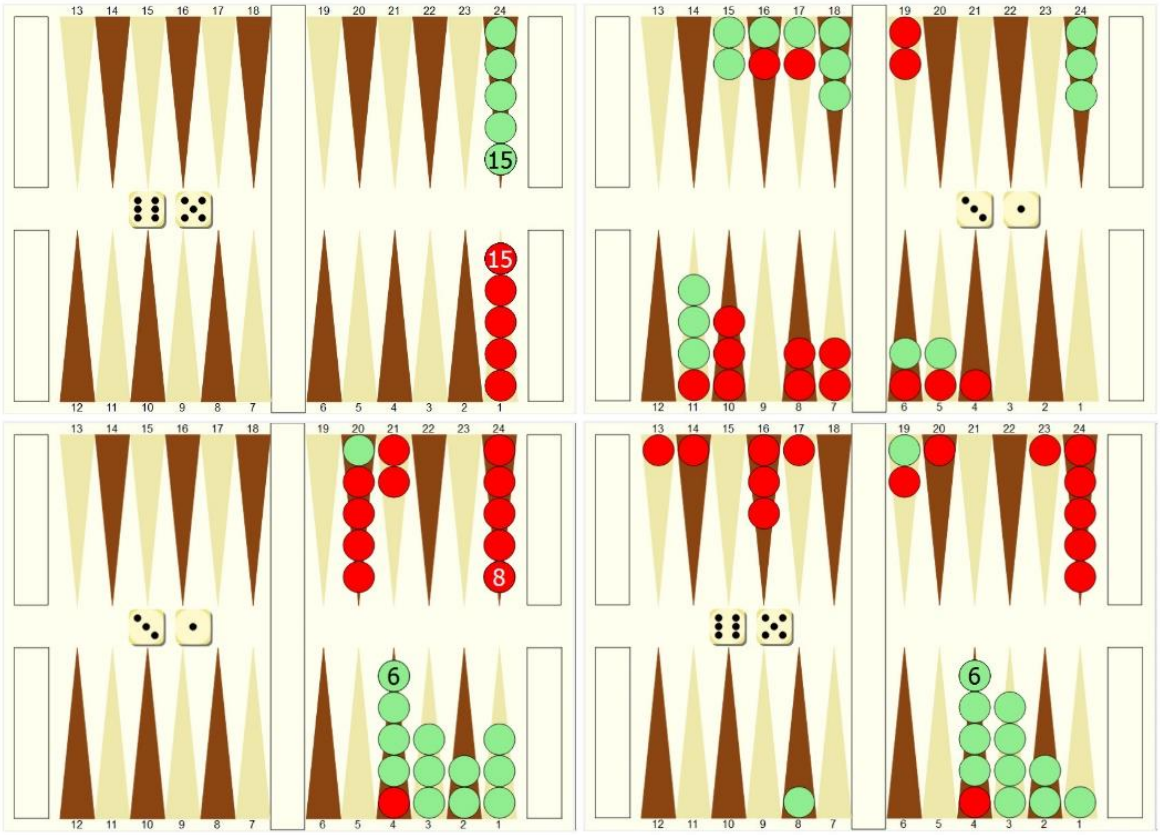


Figure 5.1: Various Plakoto positions a) Upper left: Starting position. Red player starts at point 1 and bears off at point 24, while green player starts at point 24 and bears off at point 1, b) Upper right: Typical middle-game position c) Lower Left: Endgame position where both players have pins in their bearoff quadrant d) Lower right: Both players have pins in their bearoff quadrants and some checkers in the previous quadrant.

A typical occurrence in a Plakoto game is for both players to have pinned each other. Then the best strategy usually is to try to maintain the pin(s) for as long as possible trying to make the opponent unpin his own pins. This is especially true in race situations (like Figure 5.1.c, Figure.5.1.d.), where no more pins are possible. For this initial exploration on Plakoto pin databases, we are interested in positions with the following characteristics:

- The side to move has pinned the opponent exactly once inside her bearoff quadrant (points 2-6)[1].
- The opponent has pinned the moving player exactly once.
- No further pins are possible. In this paper, these no-contact positions are also called *race* positions.

These endgames eventually resolve by one player unpinning his pin, followed by the other player moving his newly freed checker to begin the bearoff. One reason that we are interested in these endgames is that they take place frequently in practice. In an initial 100,000-game self-play experiment with Palamedes best neural network Plakoto-5 (Section 3.3.4) at the highest settings, we found out that these endgames occur in 14% of games played.

5.2 Number of endgame positions

The number of positions (R) of C checkers residing inside P points can be calculated by the following formula (Ross, Benjamin & Munson, 2007):

$$R = \binom{C + P - 1}{C} = \frac{(P + C - 1)!}{C!\,(P - 1)!}$$

The number of checkers for the positions of interest is 13 (one checker is pinned by the opponent and one checker must always be at the pinning point to maintain the pin). Depending on the memory needs of the game playing program a different number of points (P) can be used. For example, for P = 6 (all the non-pinned checkers are under the 6-point, i.e. inside the bearoff quadrant) the total number of positions is 8568 per pin placement. Such a position is shown in Figure 5.1.c. For the remainder of this paper all discussion takes place under the assumption that all unpinned checkers of the player to move is under the 12 point (P = 12, R = 2496144). A sample position can be seen in Figure 5.1.d. Note

[1] No-contact positions where a player has pinned the 1-point (also known as "mother" point) are proven double wins for the pinning player except for the rare cases, when the opponent has also pinned the 1-point (tie).

that in both Figure 5.1.c and Figure 5.1.d the position is valid for database retrieval for either player to move.

We have constructed a different database for each possible pin point of the moving player (2-6), so we have 5 databases and 12,480,720 positions in total for the 12-point version. This database is one-sided and corresponds to half the board. If we assume that the opponent has a similar position to the other half, the total possible 2-sided "true" positions that these databases can apply is $12{,}480{,}720^2 = 155{,}768{,}371{,}718{,}400$. If we further assume that the opponent is pinning at the full half of his board (points 13-23), then the total applicable positions are 12,480,720 x 2,496,144 x 11 = 342,690,417,780,480. This number is the lower bound, because the endgame characteristics set in the previous section can be met in positions where the opponent player has checkers below the 12 point.

5.3 Algorithm

The goal of the players in the endgame positions already discussed is to maintain his pin as long as possible. Essentially, the player is playing a mini-game where he tries to maximize the number of moves keeping the pin. Since the game has a chance layer, this goal becomes the maximization of the average distance to unpin. Due to the fact that there is no contact, this metric can be computed using a one-sided database.

5.3.1 PLAKOTO ENDGAME PIN DATABASE ALGORITHM

The procedure we use is inspired by retrograde analysis (Thompson, 1986), where the algorithm starts from a terminal position and works backwards. In our case we do not have terminal positions, but we start at a position where all checkers have been moved the furthest. This is the position where all 13 checkers are placed at the last point (point 1). The procedure then works backwards as usual.

The database creation algorithm is shown in Figure 5.2. For every position encountered and all 21 rolls, we find all the legal afterstates, retrieve the distances and return the max distance. The distance of the current position is then calculated as the weighted average of all rolls and stored in the database. The algorithm increments the position and begins

the next iteration, until all positions are exhausted. The position is incremented in such a way that the resulting afterstates will always have a distance in the database. The only exception is when the roll has no moves, but we can find the distance of this case with a simple recursive operation.

During actual play the database is activated, when the position before the roll has the characteristics described in section 2.2. We retrieve the distances of all the afterstates and we select the move which results in the largest distance.

Algorithm1. Plakoto endgame pin database creation

```
pinDatabase(p, pinPlacement)
position ← createStartPosition(pinPlacement)
endPos ← createEndPosition(pinPlacement, p)
while position is not endPos do
  saveInDB(hash(position), findDistance(position))
  increment(position)
end while

function findDistance(position)
  avgDistance ← 0
  for every roll d of the 21 possible rolls do
    afterStates ← findMoves(position, d)
    distances ← readDistancesFromDB(afterStates)
    distance ← max(distances)
    if d is double roll
      avgDistance += distance
    else
    avgDistance += 2 * distance
    end if
  end for
  return avgDistance / 36
```

Figure 5.2: Plakoto endgame pin database algorithm

5.3.2 Storage and Hashing

Important properties for many endgame databases are the storage and the compression mechanisms used. We use a modified version of the hashing function used in (Benjamin, Ross & Andrew, 1996) to encode the board position to a 32-bit integer. This function is fast, gives a perfect hash and can be easily decoded for the reversed procedure (int to

position). Since the number of records is relatively small, we have not made any attempts to compress the database. For the same reason, we store the distance value as a double for maximum precision, although it may not be needed. The minimum amount of precision that is acceptable for best play is left for future work. The final database size is 19MB for the 12-point and 67Kb for the 6-point version per pin placement.

5.4 Discussion

In this section we discuss potential problems with the one-sided databases and conduct two experiments to evaluate our existing AI in positions from the database.

5.4.1 Potential problems with one-sided databases

One problem with one-sided databases is that it may give errors in actual play, when we take the opponent into account. This is already documented for the one-sided bear-off databases used in backgammon (Ross, Benjamin & Munson, 2007). We identified one possible problem case in our databases in a very rare situation. where the player to move has a high average distance to unpin for all available moves and the opponent is almost ready to unpin. In this case, because the unpinning of the opponent is almost certain, it may be best for the moving player to prepare for a better placement in the bearoff quadrant instead of continuing to maximize his distance to unpin. However, rollout experiments in 5 samples of such cases have not given evidence that one strategy is better than the other. We believe the problem exists in the bearoff databases, because the problematic bearoff positions are near the end of the game, while our "problematic" positions, being much further away from terminal, allow the luck factor to "wash out" any small errors.

5.4.2 Using the databases to evaluate the neural networks

Another interesting use of endgame databases (or databases of solved games) is to evaluate existing AI implementations. We conducted experiments with Palamedes using the best neural network (NN) available for Plakoto: a) firstly, for all database positions and all possible rolls we checked if the best move of the NN coincided with the best as seen in the databases and b) secondly, we played 100,000 self-play games with the NN and, when

a database position was encountered, we compared the move chosen by the NN to the database's optimal. For the first experiment we constructed the opponent position as a mirror of the player to move.

Table 5.1: Evaluation of Palamedes AI in Plakoto pin endgames

Comparison Method	Correct moves by the NN (%)
All positions	15%
Self-play positions	64%

As it can be seen, the NN does not select the best move 85% of the time in the first test, however it does noticeably better at positions found in practical play. We believe this is normal behavior for the NN to score so low in the first test, because the self-play procedure used to train the network certainly could not generalize well to all possible cases most of which are corner cases rarely to be seen in expert play. The result of the second test shows the importance of such databases to enhance the move selection mechanism of the existing AI.

5.5 Conclusion and future work

We have presented an algorithm that created several one-sided endgame databases for the game of Plakoto. The databases are small but can be applied to a huge number of endgame positions. To the best of our knowledge, this is the first time that endgame databases are created for the game of Plakoto. We have also shown that the usage of these databases greatly enhances our AI's move selection.

There are several avenues to build upon these results. An obvious one is to construct more databases with the same method. We have only built databases for 2-6 pinned points, pinned points 7-18 can be easily created. Also, databases with more than one pin per side are possible. The conversion of our algorithm to race endgames where the opponent has pinned more than one checker is straitghtforward. A more difficult case is when the moving player has two or more pins.

With the presence of these databases our neural network evaluation function does not need to generalize in these types of positions. We could improve the representation power of our network by retraining the NN without taking into account these endgames.

Finally, we would also like to explore compression techniques for storage. This will be essential for the creation of larger pin endgame databases.

Chapter 6

CHAPTER 6: PALAMEDES

All the agents described in this thesis are packaged in the Palamedes program. The users of Palamedes can play against the AI agents through an attractive graphical user interface (Figure 6.1). Palamedes is freely available for Windows from the web page of the project[1] or from the Google Play Store[2] for Android devices. The description in this chapter is based on the Windows Palamedes version 0.50 unless otherwise stated expressively. The Android version has limited configurability, following the general practice in mobile games "as simple as possible".

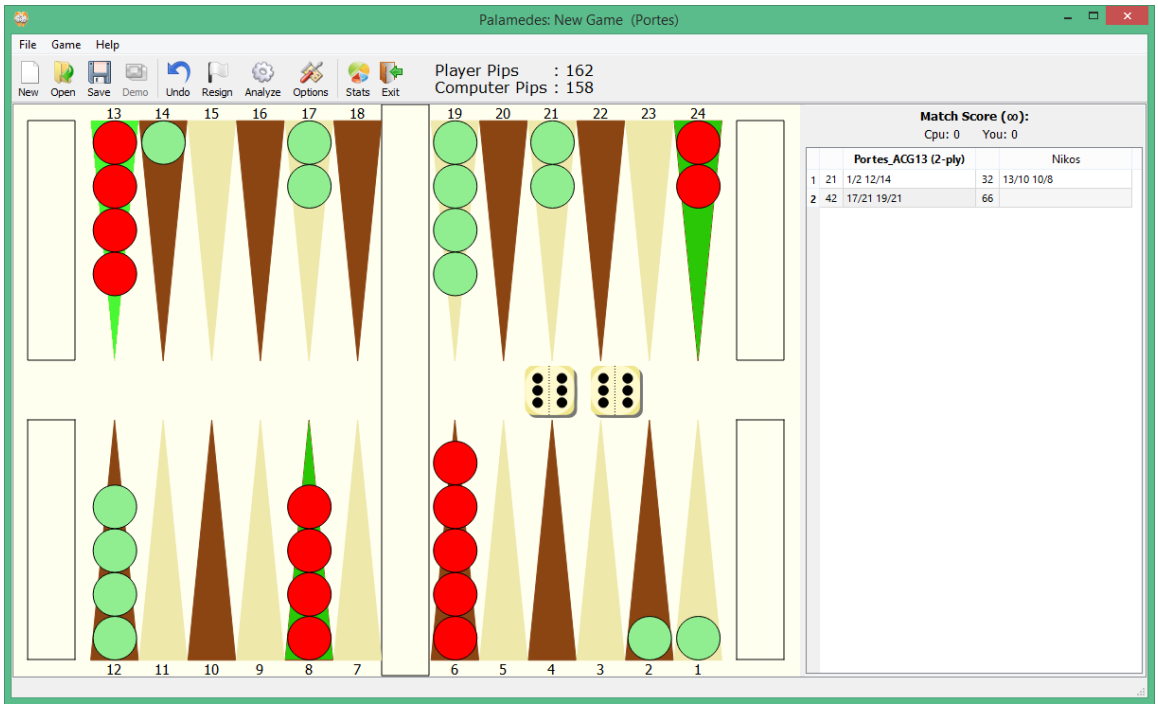


Figure 6.1: Typical Palamedes Screen when playing a game (Windows version)

Palamedes is programed in the C++ language. The graphical interface is provided by the Qt framework[3] (Open Source Version, LGPL licence) and the neural networks are implemented with Eigen (Guennebaud, et al., 2010), a matrix and linear operations library.

---

[1] http://ai.uom.gr/nikpapa/Palamedes
[2] https://play.google.com/store/apps/details?id=gr.nikpapa.palamedes
[3] http://www.qt.io/

Palamedes does not provide a learning component, meaning that all agents have fixed, deterministic strategies.

6.1 Feature list

6.1.1 VARIANTS SUPPORTED

Palamedes 0.50 supports the following backgammon variants: Portes, Plakoto, Fevga, Narde, HyperGammon, NackGammon, Tapa, Takhteh and standard backgammon with more variants planned in the future. Most of these variants have been discussed in Section 2.1. The remaining variants are discussed here.

HyperGammon is a variant that has the same rules as standard backgammon, with the only exception that the players have only 3 checkers instead of 15, and these checkers start at the first points (24, 23, 22). Palamedes support of Hypergammon extends these rules by allowing 3 to 6 checkers per side.

Narde is a variant popular in Russia that is similar to Fevga. The starting position and the direction of movement are the same. The differences in the rules are the following:

- Players can move only on checker off the starting point each turn.The first point is only allowed one checker movement each roll. There is no staring rule as in Fevga.
- Primes are allowed only when a checker of the opponent has moved ahead of the last point of the prime we want to make.
- All other special rules of Fevga (blocking rule, prime rules) do not apply.

We use the Fevga NNs for the AI in Narde. The play is “good enough” but a specialized network trained specifically for this game would certainly be more effective. This is left for future work.

Nackgammon is a variant of standard backgammon invented by Nack Ballard that has a different starting position, adding 2 checkers at point 23 and removing 1 checker from the two big stacks (6, 13). The games tend to be quite a bit longer, because one cannot

easily run quickly with the back checkers. It is a positional game, with more emphasis on priming and back games, and less on attacking and blitzing.

Takhteh is a variant similar to Portes that is popular in the Middle East and has the following additional rules:

- No hit and run: When a checker hits an opponent checker, this checker can be moved again in the same turn.
- No pip wastage in bearoff: This means that one should always bear off a checker where possible rather than use a smaller number to move that checker forward.

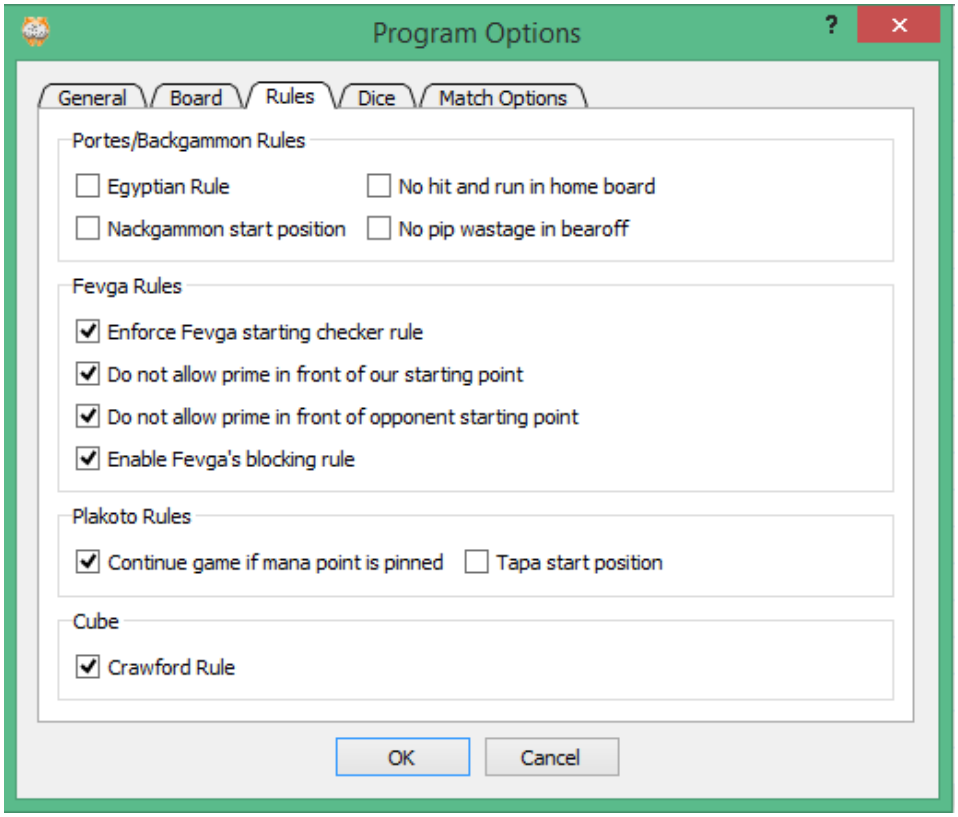


Figure 6.2: Palamedes options for changing the game rules

Palamedes offers some support for enabling/disabling some of the rules of the games (Figure 6.2). This is mainly used in Fevga, in order to support the different variations popular in some Greek regions.

The android version currently allows only Portes, Plakoto and Fevga games without the option of changing the game rules.

6.1.2 HUMAN VS AI PLAY

The user can play against any of the available neural network agents. In the 0.50 version the following NNs are available:

- Portes, Backgammon, Hypergammon, Takhteh:
    - Portes_ACG13. This is the best NN trained in Section 3.3.
    - Portes160. This is a NN with 160 hidden nodes but no expert features, trained using the same procedure as the other NNs. This NN is weaker than Portes_ACG13 NN.
- Plakoto, Tapa:
    - Plakoto-3
    - Plakoto-4
    - Plakoto-5
- Fevga, Narde:
    - Fevga-4
    - Fevga-5
    - Fevga-6

The other NNs mentioned in Chapter 3 can be found in earlier program versions and can be downloaded in the program's site. There is also an option to load a valid neural network from file. This option is mainly used for testing and troubleshooting. In the android version, only the best neural network for each game is installed.

Finally, the AI can resign the game, when its evaluation function shows that it has very little chances to win the game. This gives Palamedes human-like behavior and speeds-

up gameplay. Palamedes resigns a double game, when the probability of losing a double game is perceived above 99% (LD > 0.99). Similarly, a single game resignation occurs, when the probability of winning a single game is calculated by the NN to be less than 1% (W < 0.01).

6.1.3 LOOK-AHEAD AND DIFFICULTY

All the agents can be modified to play in two modes 1-ply and 2-ply, as it is described in section 3.2. The 2-ply look-ahead can be refined using forward pruning through configuration of two settings in the general program settings (Figure 6.3) under AI pruning.

Both options are based on the fact that at 1-ply all candidate moves are graded by the agent and then sorted. The first option (*maximum number of moves expanding in 2-ply*) expands to 2-ply only up to the number of moves selected. This is useful, because sometimes the available moves are very high (up to 1000) and expanding all moves is very computationally expensive. If we believe that the NNs are accurate enough to ensure that the best move lies in the best x moves as graded by 1-ply, then we can confidently set this option to x. The second option (*Prune when value difference greater than*) prunes the 2-ply expansion based on the value difference from the best move. Moves that have values greater than value of (first move + this option value) are not expanded.

The depth of look-ahead can be selected at the start of each game/match. In the android version, there is no such selection, the player just selects a difficulty setting (easy, normal, expert) at game start. The normal difficulty is equivalent to 1-ply, the expert difficulty is 2-ply with pruning the 15 moves pruning and the easy difficulty is 1-ply look-ahead but instead of selecting the best move, the agent selects one move randomly from the top five moves.

Searching at greater depths is straightforward and it is planned for inclusion in future program updates.

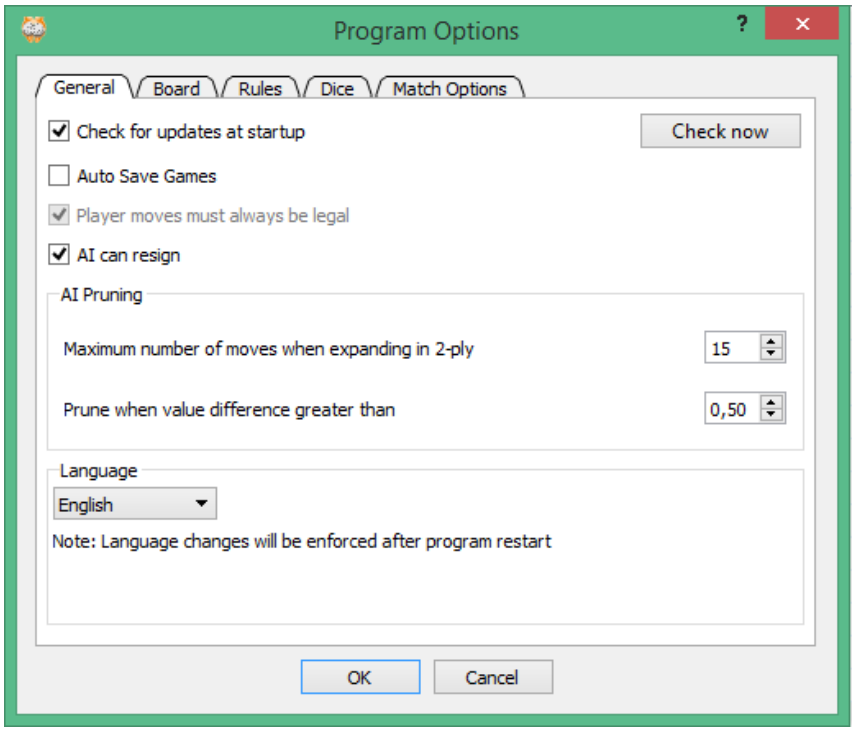


Figure 6.3: Palamedes general program options

6.1.4 Endgame databases

Palamedes supports the Plakoto Endgame Databases created with the methods presented in Chapter 5. For space reasons, all Palamedes versions include the "small" version, i.e. the 6-point databases. In the future, when the "large" 12-point databases will be compressed sufficiently enough, these databases could be included.

Palamedes also supports a two-sided 6-point bearoff database that can be used in all games. This database is very large (5.48 GB) and so it is not included in the available version for download. This database is mainly used in competitions. The program searches if this database exists at program startup.

6.1.5 Modes of play

Palamedes can play both "money games" and matches in any variant supported (Figure 6.4). Match length can be 1, 3, 5, 7, 9, 11, 13 or 15 (Android: 1, 3, 5, 7). Money games are identified by the infinity symbol (∞). AI decisions in a match are influenced by

the match score as described in Section 4.4. The doubling cube can be used only in standard backgammon, in the future we plan to make it available to the other variants as well.

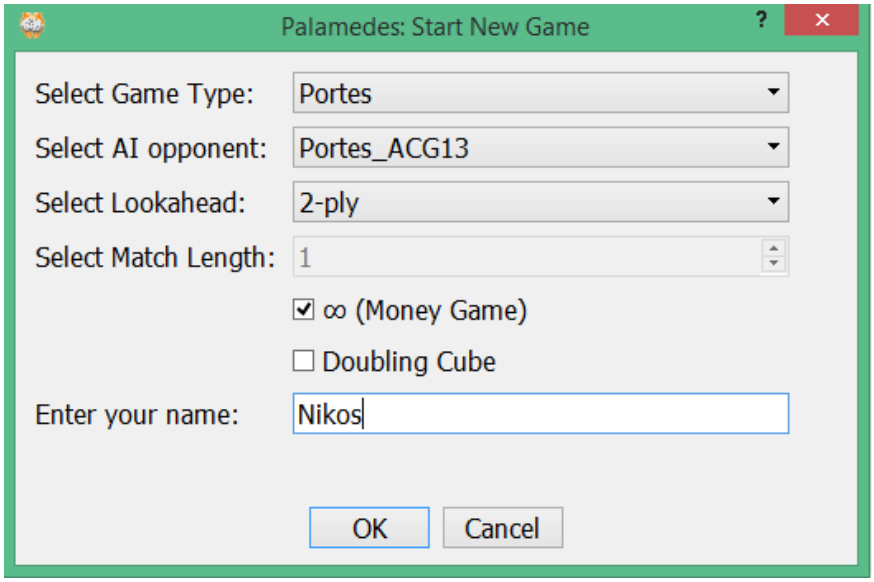


Figure 6.4: Game/Match Start

6.1.6 PLAYER STATISTICS

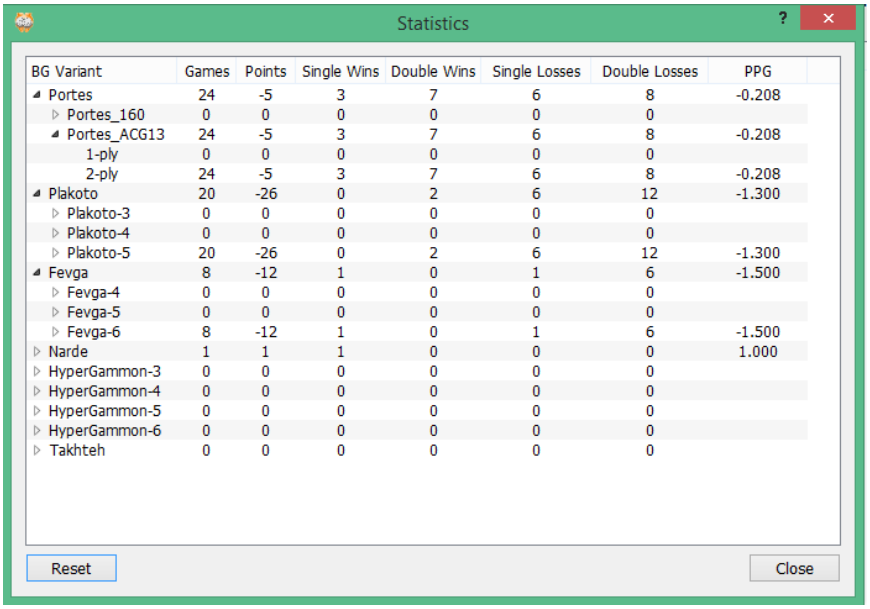
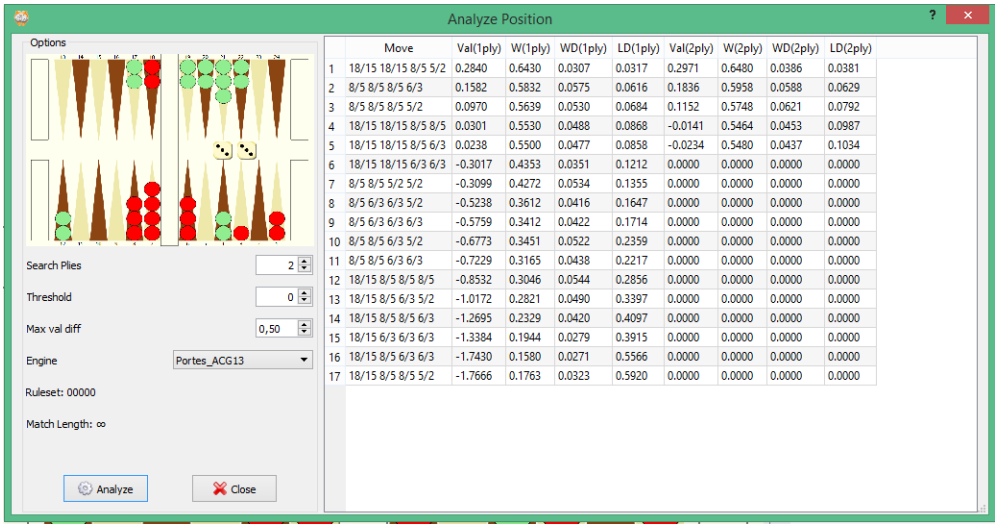


Figure 6.5: Player statistics in Palamedes

Results of every game are recorded in a database where the player can see his/her overall points against any combination of Game/NN/Look-ahead (Figure 6.5). Values stored are how many single/double games were won/lost and the average score of the player (in PPG – Points per Game).

6.1.7 ANALYSIS

Users can analyze a game, after it is finished. Selecting a position or dice roll in the scoresheet and pressing the Analyze button shows the Analysis window (Figure 6.6).



	Move	Val(1ply)	W(1ply)	WD(1ply)	LD(1ply)	Val(2ply)	W(2ply)	WD(2ply)	LD(2ply)
1	18/15 18/15 8/5 5/2	0.2840	0.6430	0.0307	0.0317	0.2971	0.6480	0.0386	0.0381
2	8/5 8/5 8/5 6/3	0.1582	0.5832	0.0575	0.0616	0.1836	0.5958	0.0588	0.0629
3	8/5 8/5 8/5 5/2	0.0970	0.5639	0.0530	0.0684	0.1152	0.5748	0.0621	0.0792
4	18/15 18/15 8/5 8/5	0.0301	0.5530	0.0488	0.0868	-0.0141	0.5464	0.0453	0.0987
5	18/15 18/15 8/5 6/3	0.0238	0.5500	0.0477	0.0858	-0.0234	0.5480	0.0437	0.1034
6	18/15 18/15 6/3 6/3	-0.3017	0.4353	0.0351	0.1212	0.0000	0.0000	0.0000	0.0000
7	8/5 8/5 5/2 5/2	-0.3099	0.4272	0.0534	0.1355	0.0000	0.0000	0.0000	0.0000
8	8/5 6/3 6/3 5/2	-0.5238	0.3612	0.0416	0.1647	0.0000	0.0000	0.0000	0.0000
9	8/5 6/3 6/3 6/3	-0.5759	0.3412	0.0422	0.1714	0.0000	0.0000	0.0000	0.0000
10	8/5 8/5 6/3 5/2	-0.6773	0.3451	0.0522	0.2359	0.0000	0.0000	0.0000	0.0000
11	8/5 8/5 6/3 6/3	-0.7229	0.3165	0.0438	0.2217	0.0000	0.0000	0.0000	0.0000
12	18/15 8/5 8/5 8/5	-0.8532	0.3046	0.0544	0.2856	0.0000	0.0000	0.0000	0.0000
13	18/15 8/5 6/3 5/2	-1.0172	0.2821	0.0490	0.3397	0.0000	0.0000	0.0000	0.0000
14	18/15 8/5 8/5 6/3	-1.2695	0.2329	0.0420	0.4097	0.0000	0.0000	0.0000	0.0000
15	18/15 6/3 6/3 6/3	-1.3384	0.1944	0.0279	0.3915	0.0000	0.0000	0.0000	0.0000
16	18/15 8/5 6/3 6/3	-1.7430	0.1580	0.0271	0.5566	0.0000	0.0000	0.0000	0.0000
17	18/15 8/5 8/5 5/2	-1.7666	0.1763	0.0323	0.5920	0.0000	0.0000	0.0000	0.0000

Figure 6.6: Analysis window

There are options to change all the AI settings available (NN, LookAhead, etc). In the right part of the dialog the report of the analysis is shown. All the available moves are shown along with the output values from the evaluation function (Val columns) and the outputs of the neural network (W, WD, LD). 2-ply columns are computed taking the weighted average of all 21 opponent rolls. The moves are sorted by descending value (Best moves first). When the positions belongs to an endgame database loaded by the program the values from the databases are shown instead of the NN.

6.1.8 DICE GENERATORS

Palamedes also lets users control the pseudo-random generator used by the program for producing the dice rolls (Figure 6.7).

The available pseudo-random generation algorithms are: Linear Congruential, Mersenne Twister and Ranlux. All these are default algorithms supplied by the C++11 <random> library. The default algorithm is Mersenne Twister.

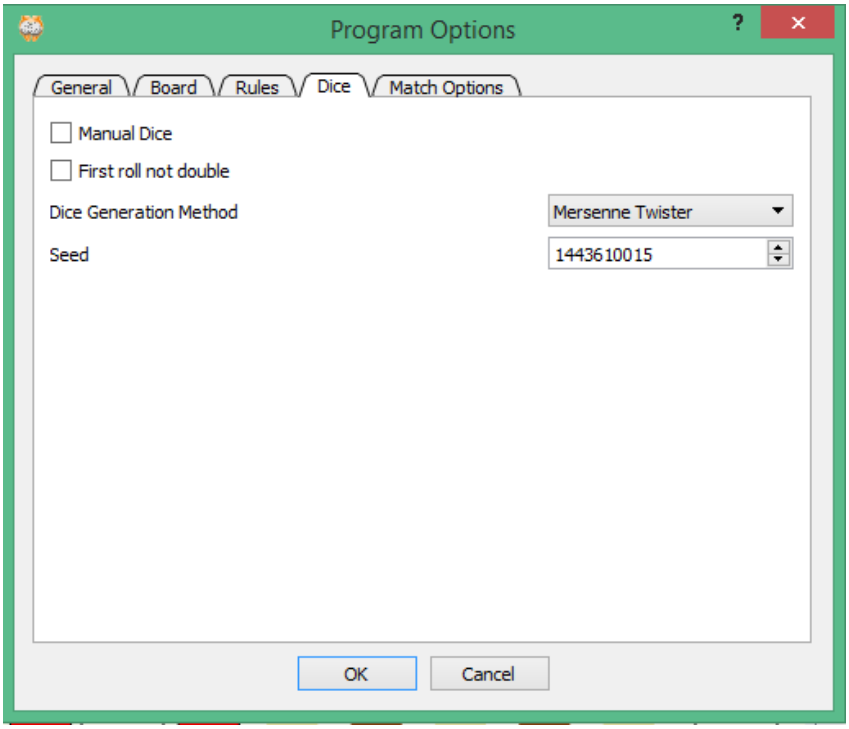


Figure 6.7: Dice options

The seed of the algorithm is randomly set at program startup. The user has the option to change the seed to whichever value he wants. Setting the seed to a value will always reset the dice algorithm to a state determined by the seed value. This means that the dice rolls after setting the seed will always be the same, if the same seed is entered.

There is also the option for entering the dice manually. When setting this option, a dialog appears prompting the user to enter the dice roll, when a player must roll the dice. The manual dice option can be used in combination with physical dice or with independent dice generators such as the Aias Floating Dice Roller[1] app on Android devices.

[1] https://play.google.com/store/apps/details?id=com.aiassoft.floatingDice

6.1.9 LOAD/SAVE GAMES

All games can be saved in files. These files are text files, have a .tavli extension and have a special formatting that supports backgammon variants. This function is useful for loading previously saved games and analyzing them. The current version of Palamedes supports a single game per file.

6.2 Backgammon Computer Olympiad Participation

Palamedes participated two times in the backgammon computer Olympiad organized by the ICGA, in 2011 and in 2015. This event gathers every other year researchers and programmers that are interested in making computer programs for board games and games in general. In the backgammon tournament, only a standard backgammon tournament is run. This is done mostly because: a) the other variants are not so popular globally and b) only a few backgammon programs know how to play backgammon variants other than the standard game. The Computer Olympiad is organized since 1987. Palamedes won the first place out of three participants (GNUBG, BgBlitz, see Section 2.3.1.1) in both of its participations.

6.2.1 BACKGAMMON COMPUTER OLYMPIAD 2011

The 2011 Computer Olympiad was staged in Tilburg, The Nederlands. At that time Palamedes (version 0.41) did not yet know how to play with the doubling cube. The organizers gratefully allowed Palamedes to participate as a full participant, with opponents making the necessary adjustments to disable the doubling cube when playing against Palamedes. However, the triple wins of standard backgammon were kept intact, something that was problematic for Palamedes, since its trained NN was trained having Portes in mind, without taking triple wins into account. Thankfully, triple wins in standard backgammon are very rarely encountered, about 1% of all games. Moreover, in these days, Palamedes played in “money-game” mode only, that is it did not take the match score into account when making decisions.

The tournament format was round-robin with each round consisting best of three 15-point matches and best of three 7-point matches. The results of all rounds were:

1. Palamedes – BgBlitz 2-0 (7-5, 7-6)
2. GnuBg – BgBlitz 2-1 (14-15, 15-12, 16-12)
3. Palamedes – GnuBg 2-1 (7-5, 5-9, 7-3)

Palamedes made some small errors mainly, because it did not take the match score into consideration. Also, at the match lost against GnuBg, it lost a triple game that could easily be avoided, if it had trivial knowledge of triple wins. Despite these shortcomings, Palamedes managed to win the tournament and the gold medal.

6.2.2 BACKGAMMON COMPUTER OLYMPIAD 2015

After four years the same participants gathered again for a rematch of the 2011 tournament[1]. The opponents of Palamedes all had trained slightly better Neural Networks than their respective versions in 2011. Palamedes did not have a new NN, but was able to play according to match score (Match mode) and had a simple doubling algorithm, based on publicly available market tables. To counter triple loss situations, a special function was constructed called "backgammon avoidance" which checks if the agent is in danger of losing a triple game, and when triggered, discards the normal NN evaluation function in favor of another function that tries to avoid a triple game.

The tournament format was the same as last time: best of 3 15-point matches. In case of a tie the tiebreaker was agreed to be the number of matches won. The results were the following:

1. Palamedes-BGBlitz 2-1 (9-20, 17-10, 16-11)
2. GnuBg-BGBlitz 1-2 (15-10, 6-16, 1-15)
3. Palamedes-GnuBg 1-2 (15-11, 7-16, 7-17)

The result was dead equal with match points also the same. In this case, it was agreed to play another (smaller) match in 7 points. Palamedes won both the tiebreak

[1] In the 2013 Computer Olympiad the backgammon competition didn't take place.

matches thus winning its second gold medal in two appearances. The full results of the tiebreaks were:

1. GnuBg-BGBlitz 1-0 (9-4)
2. Palamedes-BGBlitz 1-0 (8-4)
3. Palamedes-GnuBg 1-0 (11-0)

The “backgammon-avoidance” function was triggered only once in all the games played.

Chapter 7

CHAPTER 7: CONCLUSION AND FUTURE WORK

This chapter summarizes the results of the thesis and shows some avenues for future work. The chapter is split in four sections, referencing the four main chapters (Chapters 3-6) of this thesis.

7.1 TDL training of NNs

In Chapter 3 it was shown that Temporal Difference Learning combined with artificial neural networks as function approximators is capable of producing high performance game playing programs in backgammon variants Portes, Fevga and Plakoto. For the games of Plakoto and Fevga the resulting agents greatly outperform the only available program for comparison, Tavli3D. In Portes and standard backgammon, Palamedes showed its strength by winning two times the Backgammon Computer Olympiad organized by the ICGA, the most prestigious competition for backgammon software.

These results answer the first research question (Section 1.2) of this thesis: *that strong game-playing agents can be built that can play at expert level the backgammon variants popular in Greece: Portes, Plakoto and Fevga.*

In all games we used expert features to enhance performance. The problems found by learning overlapping features indicate that one must choose the features to be trained very carefully, or else risking suboptimal performance.

We have managed to increase the performance of our temporal difference learning architecture by making the target of the update the inverted value of the opponent's next state and by updating the game sequence starting from the terminal and working to the starting position, a procedure we call *reverse offline recalc*. This algorithm was found to be the most effective compared to several different training algorithms that we experimented with.

Our experiments with different values for the learning rate α and the λ parameter show that the best choice for either of them is domain specific. Using our setup, it is possible to start the training with high values and gradually decrease them.

The proposed method answers the second research question (Section 1.2) of this thesis: *that the learning algorithms and training setups can be improved in order to enable AI agents to learn to play backgammon games effectively by self-play.*

In the future, we intend to investigate further improving the playing strength of the agents by adding or modifying more features. In the Fevga variant for example, the heuristic for calculating the probability of making a prime formation on the next roll can be improved by including cases with two or more missing checkers and by making it faster to compute. An automatic process of selecting, comparing and training the available features could be used in order to detect the beneficial from the problematic ones. This process, however, can be very time consuming, especially when many games must be played for good learning (as is in backgammon) or the number of features is large (as is in chess for example). These enhancements can be used in other games as well as in conjunction with other TD learning algorithms.

Ultimately, however, it would be best if no expert features were added by the programmer and these features where automatically detected by the agents. A training setup where a self-trained agent reached expert knowledge of a complex game without including expert knowledge as features would be a major scientific breakthrough.

The learning hyperparameters, α and λ, were manually tuned. As we did not exhaust all possible combinations, it may possible that an even more aggressive approach could yield faster learning. It would be interesting to investigate an algorithm that automatically decreases these parameters during training, as it would free the human designer of the otherwise cumbersome trial and error approach.

Finally, we would like to apply the proposed method, reverse offline recalc, to other games.

7.2 Generating Statistics for Tavli games

In Chapter 4, we used the trained NNs of Palamedes to extract useful statistics for the Tavli variants that we are interested in, that is Portes, Plakoto and Fevga. Rollout experiments were conducted, where the following was calculated: the distribution of all the

result outcomes, the gammon rate, and the advantage of the first player. Our findings for Portes (without a starting double roll) are very close to those found in the literature. As far as we know, these statistics were constructed for the first time for the other two variants, Plakoto and Fevga.

The gammon rates, interestingly, fall in different ranges for each game. The smallest gammon rate is for the Fevga variant (14.27%), followed by Portes/Backgammon (26.9%), whereas Plakoto has the largest rate (at 41%). As for the advantage of the first player, this is significant in the Fevga variant, small in Plakoto and very small in Portes. The superiority of the Portes variant in this statistic was expected, because Portes (and backgammon) has the advantage of a specially crafted starting position, which is not present in the other variants.

The effect of the starting position on the statistics examined in the Plakoto variant was also shown. Changing the starting position of Plakoto only slightly, transforming it to the Tapa variant, had the effect of lowering the gammon rate and the advantage of the first player significantly, making Tapa the most "fair" backgammon variant examined so far.

Finally, as a practical application, the computed gammon rates was used to construct tables to be used when a match strategy is required. Experiments showed that such a strategy outperforms the money play strategy when playing 5-point matches in Portes and Plakoto.

These results answer the third research question (Section 1.2) of this thesis: *that the expert agents can be used to extract useful characteristics of the games.*

One of the conclusions of Chapter 4 was that the first player has a large advantage over the second player in the Fevga variant. It would be preferable if this advantage were as small as possible. What changes can we make to the Fevga rules so as to make the game fairer to the second player? Also, another interesting experiment is to compare Fevga with Narde, a variant with similar rules. However, we would need to train an expert agent for the Narde variant, so we leave this for future work.

One interesting experiment would be to try the following procedure in the backgammon/Portes variant: what would be the gammon rate and equity of a variant with the

same rules as backgammon but a starting position, where all starting checkers are placed in the player's first point? This would show how much the starting position of standard backgammon influences the outcomes of the game. If the results of our Plakoto/Tapa experiments are any indication, we suspect that an increase in both of these measurements is expected. We could have tried out an experiment using the Portes NN in this variant. However, unlike the Plakoto/Tapa case, here the change of the starting position is significant, so we feel that the Portes NN will not generalize well. A new NN-based evaluation function should be self-trained, but as this is not trivial, this is left for future work.

The match strategies created in this thesis can be applied to matches of the same game type, when at the start of a game the first player is determined randomly. In the future, we plan to extend this method to matches, where the starting player of the game is the one that wins the previous game, and in matches that consist of different game types like a Tavli match.

7.3 Plakoto Pin Endgame Databases

In chapter 5, an algorithm was presented that created several one-sided endgame databases for the game of Plakoto. These databases improve the AI's move selection when these endgames are encountered. The databases are small but can be applied to a huge number of endgame positions. To the best of our knowledge, this is the first time that endgame databases are created for the game of Plakoto.

The Plakoto endgame databases built in this thesis cover only the special case when there is a race situation, when both players have a pinned point, with the moving player having the pin in points 2-6. Also all checkers of the moving player must be in the last 12 points. An obvious improvement is to construct more databases with the same method. Databases with pinned points 7-18 and/or checker placement under the 12 point can be easily created. Also, databases with more than one pin per side are possible. The conversion of the proposed algorithm to race endgames where the opponent has pinned more than one checker is straitghtforward. A more difficult case is when the moving player has two or more pins.

With the presence of these databases our neural network evaluation function does not need to generalize in these types of positions. We could improve the representation power of our network by retraining the NN without taking these endgames into account.

Finally, an important improvement would be to compress the databases. This will be essential for the creation of larger pin endgame databases. A simple way would be to investigate if the float values can be stored with lower precision than the current (double). More complex compression techniques can be tried, like the one used by Tammelin, et al. (2015) in Texas Holdem‘ poker.

7.4 Palamedes program

The Palamedes program offers an attractive graphical interface where anyone can play against the AI agents shown in this thesis in several variants (Chapter 2). Palamedes includes the neural networks trained in Chapter 3, it can play in a match setting (Chapter 4), and supports the Plakoto Endgame Databases created in Chapter 5. Palamedes is available for free for the Windows and Android platforms.

Palamedes can be improved in several ways. Firstly, a human vs human mode can be made for the users to be able to play with one another, either on the same device or via the internet. Also, in order to be able to test its agents against other agents, a backgammon connection protocol could be developed. Saved games, at the time of writing this thesis, are not compatible with other backgammon programs. It would be helpful, if Palamedes could save the games/matches in a format readable by other programs, at least for standard backgammon.

We also plan to increase the number of backgammon variants that can be handled by Palamedes. Interesting candidates towards this direction are the acey-deucey, gioul and gul-bara variants. Finally, we plan to improve the look-ahead procedure by searching in greater depths and by utilizing cutoff algorithms as in (Hauk, Buro, & Schaeffer, 2006).

REFERENCES

Allis, L.V., van der Meulen M., & van den Herik H.J. (1994). Proof-Number Search. *Artificial Intelligence*, Vol. 66, ISSN 0004-3702, (pp. 91–124).

Anderson, J. R. & Lebiere C. (1998). *The Atomic Components of Thought*. Lawrence Erlbaum Associates.

Andrews R., Diederich J., & Tickle A. (1995). Survey and critique of techniques for extractingrules from trained artificial neural networks. *Knowledge-Based Systems* Vol 8(6), (pp. 373-389).

Arneson, B., Hayward, R. B., & Henderson, P. (2010). Monte Carlo tree search in hex. *IEEE Transactions of Computational Intelligence AI Games*, vol. 2(4), (pp. 251–258).

Auer, P., Cesa-Bianchi, N., & Fischer, P. (2002). Finite-time analysis of the multiarmed bandit problem," *Machine Learning,* Vol. 47, no. 2, (pp. 235–256).

Azaria, Y. & Sipper M. (2005). GP-gammon: Genetically programming backgammon players, *Genetic Programming and Evolvable Machines*, vol. 6(3), (pp. 283–300).

Backgammon (2015), Wikipedia, Accessed September 18, 2015 at http://en.wikipedia.org/wiki/Backgammon

Baxter, J., Tridgell, A., & Weaver, L. (1998a). Tdleaf(): Combining temporal difference learning with game-tree search. *Australian Journal of Intelligent Information Processing Systems*, Vol 5(1), (pp. 39-43).

Baxter, J., Tridgell, A., & Weaver, L. (1998b). Knightcap: a chess program that learns by combining td(lambda) with game-tree search. *15th International Conference on Machine Learning* (pp. 28-36). Morgan Kaufmann, San Francisco, CA.

Baxter, J., Tridgell, A., & Weaver, L. (2000). Learning to Play Chess Using Temporal Differences. *Machine Learning,* 40(3), (pp 243-263)

Benjamin, A., & Ross, A.M. (1996). Enumerating backgammon positions: the perfect hash. *Interface: Undergraduate Research at Harvey Mudd College*, Vol 16 (1), (pp. 3-10).

Bertsekas, D. (1995) Dynamic Programming and Optimal Control. Athena Scientific.

Bertsekas, D., & Tsitsiklis, J. (1996) Neuro-Dynamic Programming. Athena Scientific.

Bowling, M., Burch, N., Johanson, M., & Tammelin, O. (2015). Heads-up limit hold'em poker is solved. *Science*, Vol 347(6218), (pp. 145-149).

Browne, C.B., Powley, E., Whitehouse, D., Lucas, S.M., Cowling, P.I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., & Colton, S. (2012). A Survey of Monte

Carlo Tree Search Methods. *IEEE Transactions on Comp. Intell. and AI in Games*, Vol 4(1), (pp. 1-43).

Buro, M. (1998). From Simple Features to Sophisticated Evaluation Functions. *Proceedings of the 1st International Conference on Computers and Games*. Springer, LNCS(1558), (pp. 126-145).

Campell, M., Hoane Jr, A.J., Hsu, F-h. (2002). Deep Blue. *Artificial Intelligence*, Vol 134(1-2), (pp 57-83).

Cazenave, T., & Saffidine, A. (2011). Score Bounded Monte-Carlo Tree Search. *Computeras and Games Conference*, (CG 2010), LNCS, Vol. 6515, (pp. 93–104). Springer, Heidelberg.

Ciancarini, P., & Favini, G.P. (2010). Solving kriegspiel endings with brute force: the case of KR vs. K. *Advances in Computer Games* (ACG 2010). (pp. 136-145).

Coulom.R. (2006). Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. *5th International Conference on Computers and Games*, Turin, Italy, (pp. 72-83).

Coulom, R. (2007). Monte-Carlo tree search in crazy stone. *Procedeeings of Game Programming Workshop*, Tokyo, Japan, (pp. 74–75).

Darwen, P. (2001). Why Co-Evolution beats Temporal-Difference Learning at Backgammon for a Linear Architecture, but not a Non-Linear Architecture. *Proceedings of the 2001 Congress on Evolutionary Computation (CEC-01)*. Seoul, Korea, (pp.1003-1010).

Extreme Gammon. (2015). About XG Mobile, Available from: <http://www.xg-mobile.com/about.aspx>. [8 September 2015]

Fang, H.R., Glenn, J. & Kruskal, CP. (2008). Retrograde approximation algorithms for jeopardy stochastic games. *ICGA Journal*, (pp. 29-2).

Fang, H.R., Hsu, T.-s., & Hsu, S.C. (2002). Construction of Chinese chess endgame databases by retrograde analysis. *Computers and Games Conference* (CG 2002). (pp. 96–114). Springer, Heidelberg.

Finnsson, H., & Björnsson, Y. (2008). Simulation-based approach to general game playing. *23rd Association of Advancement of Artificial Intelligence Conference*, (AAAI 2008), (pp. 259–264). AAAI Press.

Gasser, R. (1996). Solving nine men's morris. *Computational Intelligence*, Vol 12(1), (pp. 24-41).

Gelly, S. (2007). A contribution to reinforcement learning; Application to computer-Go, Ph.D. dissertation, Informatique, Univ. Paris-Sud, Paris, France.

Gelly, S., & Silver, D. (2008). Achieving master level play in 9 x 9 computer Go. *23rd Association of Advancement of Artificial Intelligence Conference*, (AAAI 2008), (pp. 1537–1540). AAAI Press.

Gnubg.org, computer software 2015. Available from <www.gnubg.org> [8 September 2015].

GnuBg Mailing list post (2012), *pubeval benchmark,* Available from: <http://lists.gnu.org/archive/html/bug-gnubg/2012-01/msg00034.html> [8 September 2015].

Guennebaud, G., Beno, J. and others (2010), Eigen v3, Avalaible from: <http://eigen.tuxfamily.org> [8 September 2015].

Hauk, T., Buro, M., & Schaeffer, J. (2004). *-minimax performance in backgammon. *Proc. Computers and Games*, (pp. 35-50).

Hauk, T., Buro, M., & Schaeffer, J. (2006). *-minimax performance in backgammon. *Computers and Games* (CG 2006), LNCS, vol 3846, (pp. 51-66). Springer.

International Computer Games Association (2011). Computer Olympiad 2011 Results, Available from <https://icga.leidenuniv.nl/?page_id=106> [8 September 2015].

International Computer Games Association (2015). Computer Olympiad 2015 Results, Available from <https://icga.leidenuniv.nl/?page_id=1315#backgammon> [8 September 2015].

Keith, T. (n.d.) HyperGammon. Avalaible from: <http://www.bkgm.com/variants/ HyperBackgammon.html> [8 September 2015].

Keith, T. (2006), Backgammon openings. Rollouts of opening moves. Available from: <http://www.bkgm.com/openings/rollouts.html> [8 September 2015].

Koza, J. R. (1992). *Genetic programming: On the Programming of Computers by Means of Natural Selection.* Cambridge, MA: MIT Press.

Kuittinen, J., Kultima, A., Niemelä, J., & Paavilainen, J. (2007). Casual games discussion. *Conference on Future Play* (Future Play '07). New York, NY, USA, (pp. 105-112). ACM.

Lai, M. (2015) Giraffe: Using Deep Reinforcement Learning to Play Chess, MSc Dissertation, mperial College London.

Libro de los juegos, (2015). Wikipedia, Accessed September 18, 2015 at http://en.wikipedia.org/wiki/Libro_de_los_juegos.

Lorentz, R.J. (2008). Amazons discover monte-carlo. *Computer and Games Conference* (CG 2008). LNCS, Vol. 5131, (pp. 13–24). Springer, Heidelberg.

Lorentz, R. J. (2010). Improving Monte-Carlo tree search in Havannah. *Proceedings of Computer Games*, Kanazawa, Japan, (pp. 105–115).

Michie, D. (1996) Game-playing and game-learning automata. *Advances in Programming and Non-Numerical Computation*, (pp 183-200).

Nalimov, E.V., Haworth, G. McC., & Heinz, E.A. (2000). Space-efficient indexing of chess endgame tables. *ICGA Journal*, Vol 23(3), (pp. 148-162).

Papahristou, N., & Refanidis, I. (2011). Training Neural Networks to Play Backgammon Variants Using Reinforcement Learning. *EvoApplications 2011*, LNCS, Vol 6624, (pp. 113-122). Springer.

Papahristou, N., & Refanidis, I. (2012a). Improving Temporal Difference Learning Performance in Backgammon Variants. *Advances in Computer Games* (ACG-13). LNCS, Vol 7168, (pp 134-145). Springer.

Papahristou, N., & Refanidis I. (2012b). On the Design and Training of Bots to play Backgammon Variants. *8th Artificial Intelligence Applications and Innovations Conference.* (AIAI 2012), Halkidiki, Greece, IFIP Advances in Information and Communication Technology, Vol 381, (pp. 78-87). Springer.

Pollack J.B. (2005). Nannon: A Nano Backgammon for Machine Learning Research. In *Computation Intelligence in Games Conference* (CIG 2005).

Pollack, J.B., Blair, A.D., & Land, M. (1997). Coevolution of a backgammon player. *Artificial Life V: Proceedings of the Fifth International Workshop on the Synthesis and Simulation of Living Systems*, Cambridge, MA, MIT Press, (pp. 92-98).

Puterman. M. L. (1994). *Markov Decision Processes*. John Wiley & Sons.

Qi, D., & Sun, R. (2003). Integrating Reinforcement Learning, Bidding and Genetic Algorithms, *International Conference on Intelligent Agent Technology (IAT-2003)*, IEEE Computer Society Press, Los Alamitos, CA, (pp. 53-59).

Romein, J.W., & Bal, H.E. (2003). Solving awari with parallel retrograde analysis. *Computer*, Vol 36(10), (pp. 26-33).

Ross, A.M., Benjamin, A.T., & Munson, M. (2007). Estimating winning probabilities in backgammon races. *Optimal Play: Mathematical Studies of Games and Gambling*, Institute for the Study of Gambling and Commercial Gaming, (pp. 269-291). University of Nevada, Reno.

Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development, 3.*

Sanner, S., Anderson, J.R., Lebiere, C., & Lovett, M. (2000). Achieving Efficient and Cognitively Plausible Learning in Backgammon, *Proceedings of the Seventeenth International Conference on Machine Learning (ICML-2000)*, Stanford, California, (pp. 823-830).

Saito, J.-T., Chaslot, G., Uiterwijk, J.W.H.M., & van den Herik, H.J. (2007). Monte-carlo proof-number search for computer Go. *Computer and Games Conference* (CG 2006). LNCS, Vol. 4630, (pp. 50–61). Springer, Heidelberg.

Schaeffer, J. (1997). *One Jump Ahead: Challenging Human Supremacy in Checkers*. New York: Springer-Verlag.

Schaeffer, J., Bjornsson, Y., Burch, N., Lake, R., Lu, P., & Sutphen, S. (2003). Building the checkers 10-piece endgame databases. *Advances in Computer Games* (ACG 2013), (pp. 193-210). Kluwer Academic Publishers.

Schaeffer, J., Burch, N., Björnsson, Y., Kishimoto, A., Müller, M., Lake, R., Lu, P., & Sutphen, S. (2007). Checkers is solved. *Science*, Vol 317, (pp. 1518-1522).

Schaeffer, J., Hlynka, M., & Julissa, V. (2001). Temporal Difference Learning Applied to a High-Performance Game-Playing Program. *Proceedings IJCAI*, (pp. 529-534).

Sheppard, B. (2002), World-championship-caliber Scrabble. *Artificial Intelligence,* 134 (pp. 241–275)

Singh, S.P., & Sutton, R.S. (1996). Reinforcement learning with replacing eligibility traces. *Machine Learning*, Vol 22(1-3), (pp. 123-158).

Sutton, R.S. (1988). Learning to predict by the methods of temporal differences. *Machine Learning*, (pp 9-44).

Sutton, R.S., & Barto, A.G. (1998). Reinforcement Learning: An Introduction, MIT Press

Szepesvári, C. (2010). Algorithms for Reinforcement Learning (Electronic Draft Version), http://www.sztaki.hu/~szcsaba/papers/RLAlgsInMDPs-lecture.pdf

Tammelin, O., Burch, N., Johanson, M., & Bowling, M. (2015). Solving Heads-up Limit Texas Hold'em. *24th International Joint Conference of Artificial Intelligence* (IJCAI-15), (to be published).

Tesauro, G. (1992). Practical issues in temporal difference learning. *Machine Learning*, Vol 4, (pp. 257-277), (1992).

Tesauro, G. (1994). Benchmark player "pubeval.c". *Backgammon Galore Website*, Available from <http://www.bkgm.com/rgb/rgb.cgi?view+610>. [8 September 2015]

Tesauro, G. (1995). Temporal Difference Learning and TD-Gammon. *Communications of the ACM*, Vol 38(3), (pp. 58-68).

Tesauro, G. (2002). Programming backgammon using self-teaching neural nets. *Artificial Intelligence*, Vol 134, (pp. 181-199).

Tesauro, G. (2011). Td-Gammon computer software, Available from: <http://www.scholarpedia.org/article/User:Gerald_Tesauro/Proposed/Td-gammon>. [8 September 2015]

Thompson, K. (1986). Retrograde analysis of certain endgames. *ICCA Journal*, Vol 9(3), (pp. 131-139).

Varouhakis, J. 2007. Tavli3D computer software 2007, Available from: <http://sourceforge.net/projects/tavli3d>. [8 September 2015]

van der Werf, E. C.D., & Winands, M, H.M. (2009). Solving go for rectangular boards. *ICGA Journal,* Vol. 30-2, (pp. 77–88).

Wiering, M.A. (2010). Self-Play and Using an Expert to Learn to Play Backgammon with Temporal Difference Learning. *Journal of Intelligent Learning Systems and Applications*, Vol 2, (pp. 57-68).

Wikipedia (2015). Backgammon, Available from: <https://en.wikipedia.org/wiki/ Backgammon>. [8 September 2015]

Wilson D.R., & Martinez T.R. (2003). The general inefficiency of batch training for gradient descent learning. Neural Networks, Vol 16(10), (pp. 1429-1451).

Winands, M.H.M., Björnsson, Y., & Saito, J.-T. (2008). Monte-carlo tree search solver. *Computer and Games Conference* (CG 2008). LNCS, Vol. 5131, (pp. 25–36). Springer, Heidelberg.

Winands, M.H.M., & Björnsson, Y. (2009). Evaluation Function Based Monte-Carlo LOA. *Advances in Computer Games* (ACG 2009). LNCS, Vol. 6048, (pp. 33–44). Springer, Heidelberg.

Winands, M.H.M., Björnsson, Y. & Saito J.-T. (2010). Monte Carlo tree search in lines of action. IEEE Transanctions of Computational Intelligence AI Games, vol. 2(4), (pp. 239–250).

Van Eck, N.J., & van Wezel, M. (2008). Application of reinforcement learning to the game of Othello. *Computers and Operations Research*, Vol 35(6), (pp. 1999-2017).

Van Lishout, F., Chaslot, G., & Uiterwijk, W.H.M. (2007). Monte-Carlo Tree Search in Backgammon, Computer Games Workshop, Amsterdam, the Netherlands, (pp. 175–184).

Veness, J., Silver, D., Uther, W., & Blair, A. (2009). Bootstrapping from Game Tree Search. *Advances in Neural Information Processing Systems*, Vol 22, (pp. 1937-1945).

Zeiler, M.D. (2012). Adadelta: an adaptive learning rate method. arXiv preprint arXiv: 1212.5701.